ON PROGRAM DECOMPOSITION AND PARTITIONING IN DATA-FLOW SYSTEMS

Jean-Luc Gaudiot

1982
CSD-821212

UNIVERSITY OF CALIFORNIA

Los Angeles

Characterization of Intermodule Communications and Heuristic Task Allocation for Distributed Real-Time Systems

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Lance Min-Tsung Lan

1985

© Copyright by

Lance Min-Tsung Lan

1985

The dissertation of Lance Min-Tsung Lan is approved.

[illegible]

Harold Borko

[illegible]

Bruce Rothschild

[illegible]

Tomas Lang

[illegible]

Milos D. Ercegovac

[illegible]

Wesley W. Chu, Committee Chair

University of California, Los Angeles

1985

To

Show-Fung (Shirley)

and our

Moms and Dads

TABLE OF CONTENTS

page

LIST OF FIGURES

page

LIST OF NOTATION

Roman Capitals

E - Qualifier for module enablement: $V_{ij}(E;t_k,t_{k+1})$, $L_{ij}(E;t_k,t_{k+1})$.

F - File: F_k

I - Interval size between PR levels: I_{PR}

IMC - Alternative presentation of IMC Volume: $IMC_{ij}(t_k,t_{k+1})$.

J - Number of modules

K - Number of files

L - Average message length: $L_{jk}(W;t_k,t_{k+1})$, $L_{jk}(R;t_k,t_{k+1})$.

M - Module: M_i.

N - Number of module invocations: $N_j(t_k,t_{k+1})$.

Also, Number of PR levels: N_{PR}.

Q - Processing-cost matrix with elements q_{is}

R - Qualifier for read

S - Number of sites (processors, or computers)

T - Accumulative execution time (AET) of a module: $T_j(M;t_h,t_{h+1})$.

V - M-F or F-M IMC Volume: $V_{ij}(R;t_h,t_{h+1})$, $V_{ij}(W;t_h,t_{h+1})$,

W - Qualifier for write

X - Assignment matrix with elements x_{is}

Small Roman

h - Time index: (t_h,t_{h+1})

i - Module index: M_i

j - Module index

k - File index: F_k

p - probability: $p_{jn}(E;t_h,t_{h+1})$, $p_{jk}(W;t_h,t_{h+1})$, $p_{jk}(R;t_h,t_{h+1})$.

q - Element in processing-cost matrix Q: q_{is}.

r - Site (processor) index

s - Site (processor) index.

Also, memory storage requirement for module M_i: s_i.

t - Site (processor) index

x - Execution time (ET) of a module per enablement: $x_j(t_k, t_{k+1})$.

Also, Element in assignment matrix X: x_{ik}.

Greek

α - Percentage to decide IMC threshold θ_{IMC}

β - Percentage to decide processor-load threshold θ_{PL}

δ - Indicator function

γ - Importance indices for IMC and PR: γ_{IMC}, γ_{PR}

θ - IMC and processor-load thresholds: θ_{IMC}, θ_{PL}

λ - Arrival rate for tasks

ρ - Processor utilization

ACKNOWLEDGEMENTS

I would like to express special thanks to my committee chairman, Professor Wesley W. Chu, for his guidance, friendship, encouragement, and insightful comments during the entire course of this research. I also thank Mike L. Green, Edward Y. S. Lee, and Richard P.-Y. Ma of TRW for their assistance in installing the DPAD Simulator at UCLA, Kin Kwong Leung for sharing with me his analytic model for estimating module response time and task response time [CHU84b] in my study of precedence relationship, and the members of my dissertation committee: Professors Milos Ercegovac, Tomas Lang, Bruce Rothschild, and Harold Borko for their encouragement. Thanks also are due to Jung Min An, Joseph Bannister, Joseph Hellerstein, and Leslie J. Holloway, for their stimulating discussions during this research, as well as Salpy Manjikian, Wendy Hagar, Aeri Lee, and Laurel Cornachio for their secretarial and administrative assistance.

Additionally, I would like to thank my grandma Pern-Tow, my parents Chio-Dong and Shu-Kuan, and other members of my family for their love and emotional support. I would always remember the lesson my late father-in-law, Dr. Jin-Liu Hsu, taught me: 'Always be honest in research.' Unfortunately he passed away one month before I finish this dissertation. Finally, I express special thanks to my wife Show-Fung (Shirley) whose love, patience, sensitivity, and devotion kept my motivation going up to the completion of my Ph.D.

This work was supported in part by U.S. Army Contracts No. DASG60-79-C-0087 and DASG60-83-C-0019.

VITA

May 16, 1952	Born, Kangshan, Kaohsiung Hsien, Taiwan
1970-1974	B.S. in Electrical Engineering, National Taiwan University
1974-1976	System Engineer, China Steel Corp., Taiwan
1976-1977	M.S. in Computer Science, University of Nebraska – Lincoln
1977-1978	Systems Programmer, Nebraska State Department of Labor, Lincoln, Nebraska
1978-1979	Member of Technical Staff, Technology Service Corp., Santa Monica, California
1980-1981	Member of Technical Staff, OAO Corp., Los Angeles, California
1983-1984	Member of Technical Staff, RFA Associates, Consultant to Rockwell International, Los Angeles
1978-1984	Research Assistant, Computer Science Department, University of California, Los Angeles, California

PUBLICATIONS

D. W. Embley, M. T. Lan, D. W. Leinbaugh, and G. Nagy, "A Procedure for Predicting Program Editor Performance From the User's Point of View", *Intl. J. Man-Machine Studies,* Nov. 1978, pp. 639-650.

M. T. Lan and G. Moring, "JOVIAL 73 Automated Complexity Analyzer", Contract Final Report, #F04704-79-C-0059, OAO Corp., Los Angeles, Calif., Sep. 30, 1980, for the MX Missiles Project.

W. W. Chu, L. J. Holloway, and M. T. Lan, "Task Allocation in Distributed Data Processing", IEEE *Computer* Magazine, Nov. 1980, pp. 57-69. (Also in P. L. McEntire & R. E. Larson (ed.), *Distributed Computing: Concepts and Implementations,* IEEE Press, 1984, pp. 109-119).

W. W. Chu, J. Hellerstein, and M. T. Lan, "Research on the Shared Database Kernel for the BMD Application", Contract Report CSD-820430, UCLA, April 1982, for U.S. Army, Ballistic Missile Defense Agency.

W. W. Chu, J. Hellerstein, and M. T. Lan, "The Exclusive-Writer Protocol: A Low-Cost Approach for Updating Replicated Files in Distributed Real-Time Systems", *The Third Intl. Conf. on Distributed Computing Systems,* Miami/Ft. Lauderdale, Florida, Oct. 18-22, 1982, pp. 269-277. (Also in P. L. McEntire & R. E. Larson (ed.), *Distributed Computing: Concepts and Implementations,* IEEE Press, 1984, pp. 219-227).

W. W. Chu, J. Hellerstein, and M. T. Lan, "Database Management Algorithms for Advanced BMD Applications", Contract Report CSD-830430, UCLA, April 1983, for U.S. Army, Ballistic Missile Defense Agency.

W. W. Chu, J. Hellerstein, and M. T. Lan, "Database Management Algorithms for Advanced BMD Applications", Contract Report CSD-840031, UCLA, April 1984, for U.S. Army, Ballistic Missile Defense Agency.

W. W. Chu, M. T. Lan, and J. Hellerstein, "Intermodule Communication (IMC) Estimation and Its Applications in Distributed Processing Systems", *IEEE Trans. on Computers,* vol. C-33, no. 8, Aug. 1984, pp. 691-699.

ABSTRACT OF THE DISSERTATION

Characterization of Intermodule Communications and

Heuristic Task Allocation for Distributed Real-Time Systems

by

Lance Min-Tsung Lan

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1985

Professor Wesley W. Chu, Chair

Distributed processing has the potential for providing lower cost, better response time, and higher availability than centralized processing. In a distributed real-time system, a fixed set of application program modules reside permanently on a set of computers (or, processors). Module executions are invoked by a stream of external stimuli (arrivals). A key work in the design of distributed real-time systems is task allocation (module assignment) — assigning the program modules onto the set of computers such that the processing of each stimulus (event) can be finished within a prescribed time limit.

The three important parameters in task allocation are intermodule communication (IMC), accumulative execution time (AET) of each module, and precedence relations (PR) among program modules. IMC is the communication between program modules through shared files. When a module on a computer writes to or reads from a shared file on *another* computer, IMC results in IPC (interprocessor communication), an overhead to the processor load. Therefore, a task-allocation algorithm should try to minimize IPC by assigning a pair of heavily communicating modules to the same computer. On the other hand, AET always contributes to processor load; its contribution is independent of task allocation. Constructing a simulator to measure the IMC and AET is time-consuming. A model is therefore developed to *estimate* these values, based on the control-and-data-flow graph and branching probability. Estimated results for a space-defense application match closely with the measured values obtained from a simulator of this application.

A program module can not be enabled before all its predecessor(s) finish execution; this relation is called the precedence relation (PR). Analytical results indicate that the size ratio of two consecutive modules plays an important role in task allocation. Generally if the execution time of the second module is much larger than the first one, then assigning these 2 modules to a same computer is beneficial to response time.

Given J modules and S computers, there are S^J module assignments. It is not feasible to enumerate through all these assignments to find the optimal assignment when S and J are large. A heuristic algorithm is proposed to find sub-optimal assignments, based on the concept of *minimum bottleneck*, using IMC and AET data and PR rules. Simulation results indicate that the algorithm generates good assignments.

CHAPTER 1

INTRODUCTION

1.1 WHY DISTRIBUTED PROCESSING

Although the computer speed has increased several orders of magnitude during the last 30 years, user demand for computing power is increasing more quickly. In the early '50s, it was estimated that 50 IBM Corp. model 704 mainframes would satisfy the computational requirements of the U.S. [MALL82]. By the end of that decade, every sizable business establishment, college, and government agency had at least one computer. Nowadays, the demand for computation is still challenging the computer architects.

One example is a real-time computer system embedded in a space-defense radar system where the computer must process the data stream returned from a continuously scanning radar in real time. Besides real-time applications, computer speedup is desired in applications of lengthy, repetitive nature. The quality of computational results in areas such as meteorology, cryptography, image processing, and sonar and radar surveillance is proportional to the amount of computation performed.

For example, in the VLSI automatic testing, every chip produced by a semiconductor manufacturer should be tested extensively. A speedup for testing a chip would increase the production volume and/or reduce the use of testing machines and technicians. Another example is in VLSI design automation where simulating a 10^2-instruction sequence requires 49 minutes of CPU time and simulating a 10^6-instruction sequence requires 250 hours [BLAN84]. If we can speed up the simulator, the product design cycle will be significantly shortened because many simulation runs are usually necessary before a chip design is finalized.

Many applications require speed capability not achievable with a single computer (or processor). [1] The two obvious approaches to this problem are faster circuits and distributed data processing (DDP). This thesis deals with DDP which provides for simultaneous execution of multiple program modules of a task. In DDP, all processors should be equal (i.e., no master/slave relations) and each can access any network resource without the interference from centralized controllers. If properly applied, DDP can be a much cheaper technique than a single high-speed computer. Hundreds of microprocessors can be interconnected to provide a processing entity that outperforms the cost/performance ratio of a supermainframe.

[1] "Computer" and "processor" are used interchangeably in this dissertation.

1.2 TASK ALLOCATION

Task partitioning and task allocation are two major steps in the design of distributed processing systems [CHU80]. If these steps are not done properly, an increase in the number of computers in a DDP system may actually result in a decrease of the total throughput. This is the well-known saturation effect. In this dissertation, we assume the software system for a DDP application has been well partitioned into a set of program modules (or, subroutines). We then study how to properly allocate (assign) these modules onto the multiple computers of the DDP system in order to realize the benefits of DDP. (In this dissertation, "task allocation" and "module assignment" are used interchangeably for step 2.)

The three important parameters for task allocation are intermodule communication (IMC), accumulative execution time (AET) of each module, and precedence relations (PR) among program modules. IMC is the communication between program modules through shared files. When a module on a computer writes to or reads from a shared file on *another* computer, it requires extra processing and communication overhead which we shall refer to as IPC (interprocessor communication). A task-allocation algorithm should try to minimize IPC by assigning a pair of heavily communicating modules to the same computer. The AET contribution to processor load is independent of task allocation. Finally, the precedence relation specifies that a program module can not be enabled before all its predecessor(s) finish execution. Analytical and experimental results indicate that the module-size ratio between two consecutive program modules plays an important role in task allocation.

1.3 BACKGROUND AND-RELATED WORKS

Existing module-assignment methods can be divided into four categories: queueing model approach, graph theoretic approach, integer 0-1 programming approach, and heuristic approach. Existing queueing models do not consider IMC and therefore do not provide good assignments.

1.3.1 Graph Theoretic Approach

In general, this approach can handle only 2-processor module assignment problems. Each module is represented by a node in a weighted graph where positive numbers assigned to the nodes represent the expected cost of each module due to its execution. IMC cost between each pair of modules is represented by the weight of a nondirected arc connecting the two nodes [JENN77, STON77, STON78, RAO79, BOKH79, CHOU82]. The arc weight then becomes the *interprocessor communication* (IPC) if the pair of modules are not coresident on a processor. Any pair of coresident modules is assumed to have zero IPC cost. An additional node is also provided for each processor; an arc between a module node and *one* of the two processor nodes represents the processing cost of running the module on *the other* processor.

The module assignment strategy in this model is to minimize total cost, defined as the sum of processing cost and IPC cost. In order to represent the assignment of modules to processors an assignment matrix X is defined such that

$$x_{is} = \begin{cases} 1 & \textit{if module } M_i \textit{ is assigned Processor } s \\ 0 & \textit{otherwise} \end{cases}$$

Processing cost is given by the Q matrix,

$$Q = \left\{ q_{is} \right\}, \qquad i=1, \ldots J, \qquad s=1, \ldots S$$

where J is the total number of modules, S is the number of processors (sites) in the system, and q_{is} represents the processing cost for module M_i on Processor s. A value of infinity, $q_{is} = \infty$, implies that module M_i cannot be executed at Processor s.

Let v_{ij} represent the IMC volume between M_i and M_j. The total cost for processing a given task can then be expressed as an objective function of the assignment X.

$$Cost(X) = \sum_{s} \sum_{i} \left\{ q_{is} x_{is} + \sum_{t \neq s} \sum_{j < i} w v_{ij} x_{is} x_{jt} \right\} \tag{1-1}$$

The first term of eq. (1-1) represents the processing cost for each module on its assigned processor. The second term represents the IPC cost between non-coresident modules. The normalization constant w is used to scale processing cost and IPC cost to account for any differences in measuring units. The minimum-cost module assignment is obtained by performing a *min-cut algorithm* on the graph [HARA69].

While this method is conceptually simple, it has several limitations. First, an extension of the min-cut algorithm to an arbitrary number of processors quickly becomes computationally intractable. An extension to four or more processors has been proposed for cases where the IMC pattern can be constrainted to be a tree [STON78]. Second, a weighted graph represents only the data flow; it does not show the control flow (the precedence relationship) among the modules of a task. The min-cut algorithm does not consider the precedence relationship. Third, the method assumes one-time execution of modules. But, in almost all real-time systems, program modules reside in the systems during entire mission time and a module is invoked (i.e., *enabled)* to execute by each occurrence of certain type(s) of events, e.g. the completion of a preceding module in the control flow or an arrival of a radar search return.

1.3.2 Integer 0-1 Programming Approach

This method stems from the file allocation problem where the model is formulated as an optimization problem and is solved via a mathematical programming technique [CHU69, CHU80, MA82]. As with the graph theoretic approach, the goal is to achieve optimal system performance by minimizing the total cost defined in eq. (1-1) over the module assignment X. In addition, the minimization is done subject to some constraints which may be imposed by a given environment or the design specifications. For example, a limited-memory constraint is represented by

$$\sum_i s_i x_{is} \leq R_s , \quad s=1, \ldots S$$

where s_i represents the amount of memory storage required by module M_i and R_s represents the memory capacity at Processor s. Load balancing can also be achieved through imposition of constraints.

The integer 0-1 programming approach can handle any number of processors. However, it is time-consuming to optimize the performance for a system with a large number of processors and modules. And it is rather complex to express the precedence relationship and its effect on task response time analytically.

1.3.3 Heuristic Approach

Gylys and Edwards proposed a heuristic algorithm for module clustering [GYLY76, EFE82]. The process of assigning two modules to the same processor is called *fusion*. The algorithm searches for a pair of modules with the largest IMC and checks to see whether the fusion of these two modules satisfies the real-time and memory constraints. If it does, that pair is fused. Otherwise, the pair with the next largest IMC is chosen as a possible candidate. The fusing process continues until all eligible pairs are examined. Although simple and fast, this approach again assumes one-time execution of modules. And, it does not consider precedence relationship.

After examining the above approaches, we are motivated to develop a heuristic algorithm that can remedy their shortcomings.

1.4 OUTLINE OF DISSERTATION

To provide a testbed for testing our theories and model analysis, we use a simulator that simulates a BMD (Ballistic Missile Defense) application — a distributed processing system for a defense unit to process the radar returns (input scenario). This BMD application *(Distributed Processing Architecture Design - DPAD)* and its simulator are presented briefly in Appendix A. Our UCLA DPAD Simulator is a modified version of the one originally developed by TRW/Redondo Beach, and we have made several enhancements (see Appendix B) for performance measurements, statistics gathering, data reduction, curve plotting, and reduced simulation time.

Since IMC and AET dramatically impacts the performance of distributed systems, a methodology is proposed to measure and present their values over the time (Chapter 2). IMC for file-update traffic and AET of the DPAD are presented as an example. From the insights obtained with our IMC measurements and other data from the simulation, an analytical model is constructed in Chapter 3 to estimate the IMC in a distributed processing system. This model is then applied to the example DPAD system. The results show that the model provides good estimation.

Next, we present an objective function for task allocation that expresses the processor load of a *bottleneck processor* in terms of IMC and AET (Chapter 4). The basic idea is to find a task-allocation algorithm that assigns the modules to processors such that the load of the bottleneck processor is minimized. To avoid the enumeration of a huge number of possible module assignments, a heuristic algorithm based on the objective function is proposed and shown to generate good assignments. Chapter 5 discusses the relationship between the precedence relation (PR) and task allocation. PR specifies that a program module can not be enabled before all its relevant preceding module(s) finish their execution. Simulation and analytical results indicate that the size ratio of two consecutive modules plays an important role in task allocation. Task response time can be reduced by properly allocating the PR-related modules. An improved heuristic algorithm for task allocation is presented based on PR, IMC, and AET. The algorithm considering PR is applied to both the DPAD system and another distributed system example. It yields better response time than the algorithm which does not consider PR. Finally, the conclusions and areas of future research are given in Chapter 6.

CHAPTER 2

MEASUREMENTS OF IMC

Communication between program modules is called *intermodule communication (IMC)*. When two modules are assigned to different computers, their IMC creates *interprocessor communication (IPC),* thereby requiring both the sending and receiving computers to execute instructions for I/O processing. If the modules reside on the same computer, no IPC is incurred. In general, IPC should be minimized to reduce the CPU cycles spent on I/O, thus improving the response time and throughput [CHU80].

Since the total volume (words) of module-enablement messages is normally much smaller than the file-update messages, we consider only the IMC volume between each pair of modules. Here, we report the IMC measurements and characterize the IMC in terms of how it varies with time, number of objects, task threads, and module pairs.

2.1 MODULE EXECUTION FREQUENCY AND IMC PER EXECUTION

A program module is enabled *every time* a predefined event occurs, e.g. M_{22} is enabled by each radar return while M_7 is enabled whenever both M_5 and M_6 finish their execution. Therefore, a module could execute *multiple* number of times during a time interval. The IMC volume from a sending module M_i to a receiving module M_j, for the time interval from t_h to t_{h+1}, $V_{ij}(t_h,t_{h+1})$, has two components: [1]

1. The number of times the sending module executes during the interval, $N_i(t_h,t_{h+1})$
2. The average message volume sent to module M_j *per execution* of M_i during the interval, $v'_{ij}(t_h,t_{h+1})$.

Thus, we have

$$V_{ij}(t_h,t_{h+1}) = N_i(t_h,t_{h+1}) * v'_{ij}(t_h,t_{h+1})$$

Regarding the number of module executions, Figures 2-1 and 2-2 plot $N_i(t_h,t_{h+1})$ ($t_{h+1}-t_h = 100$ ms) for modules M_1 and M_8, respectively. (The vertical lines indicate 90% confidence intervals which are discussed in Section 2.3.) Both Figures 2-1 and 2-2 were obtained from DPAD simulations of a three-computer distributed system for a scenario with 40 objects. [2] Figure 2-3 presents $N_i(t_h,t_{h+1})$ for all twenty-three modules in the DPAD for the same scenario.

[1] h is simply an integer index; we reserve i and j for indexing other items in this thesis.

[2] IMC does not change significantly with module assignment or the number of computers. See Section 2.9.

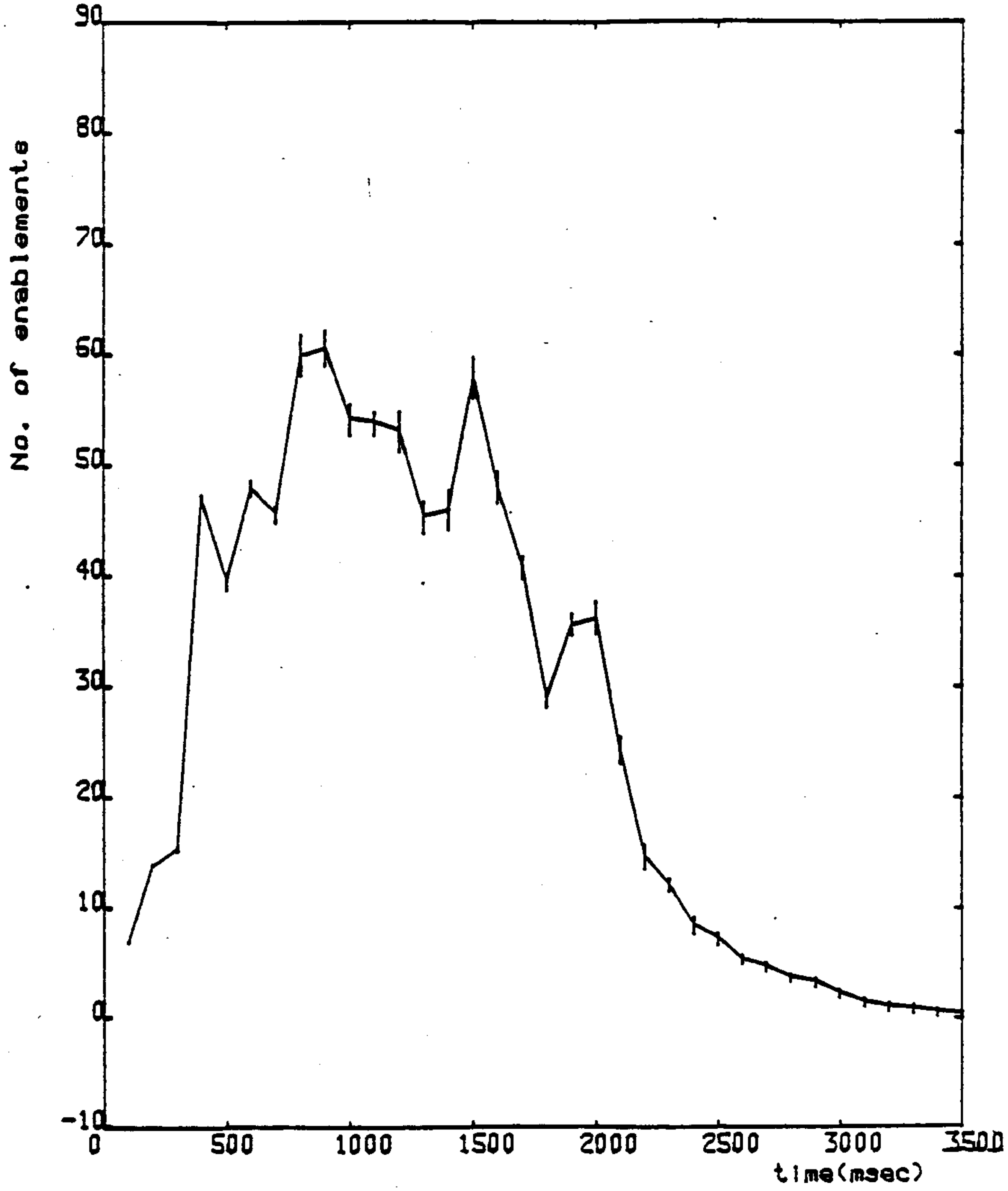


FIGURE 2-1. TASK 01 ENABLEMENTS - 1.5 MIPS, 40 OBJECTS

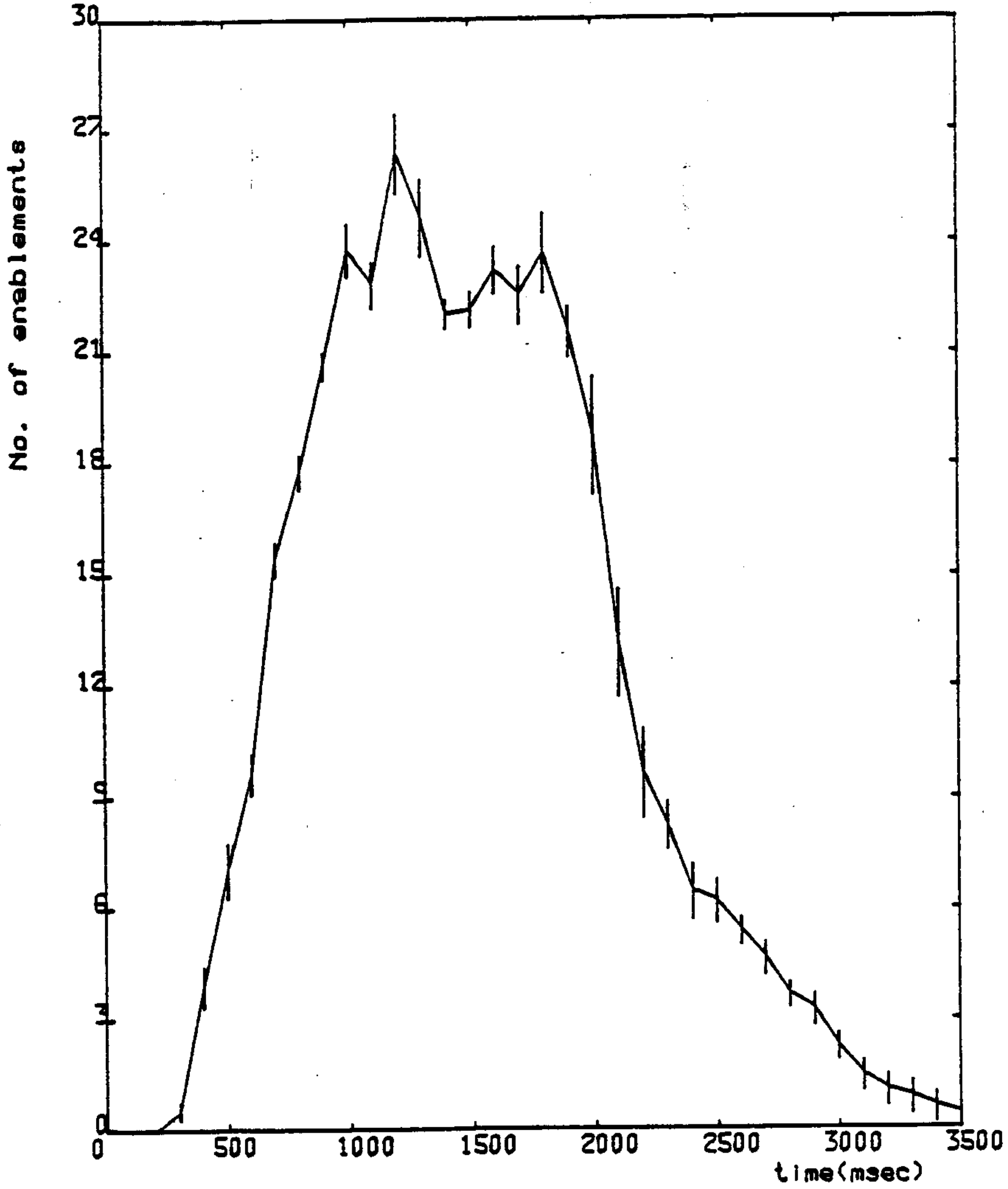


FIGURE 2-2. TASK 08 ENABLEMENTS - 1.5 MIPS, 40 OBJECTS

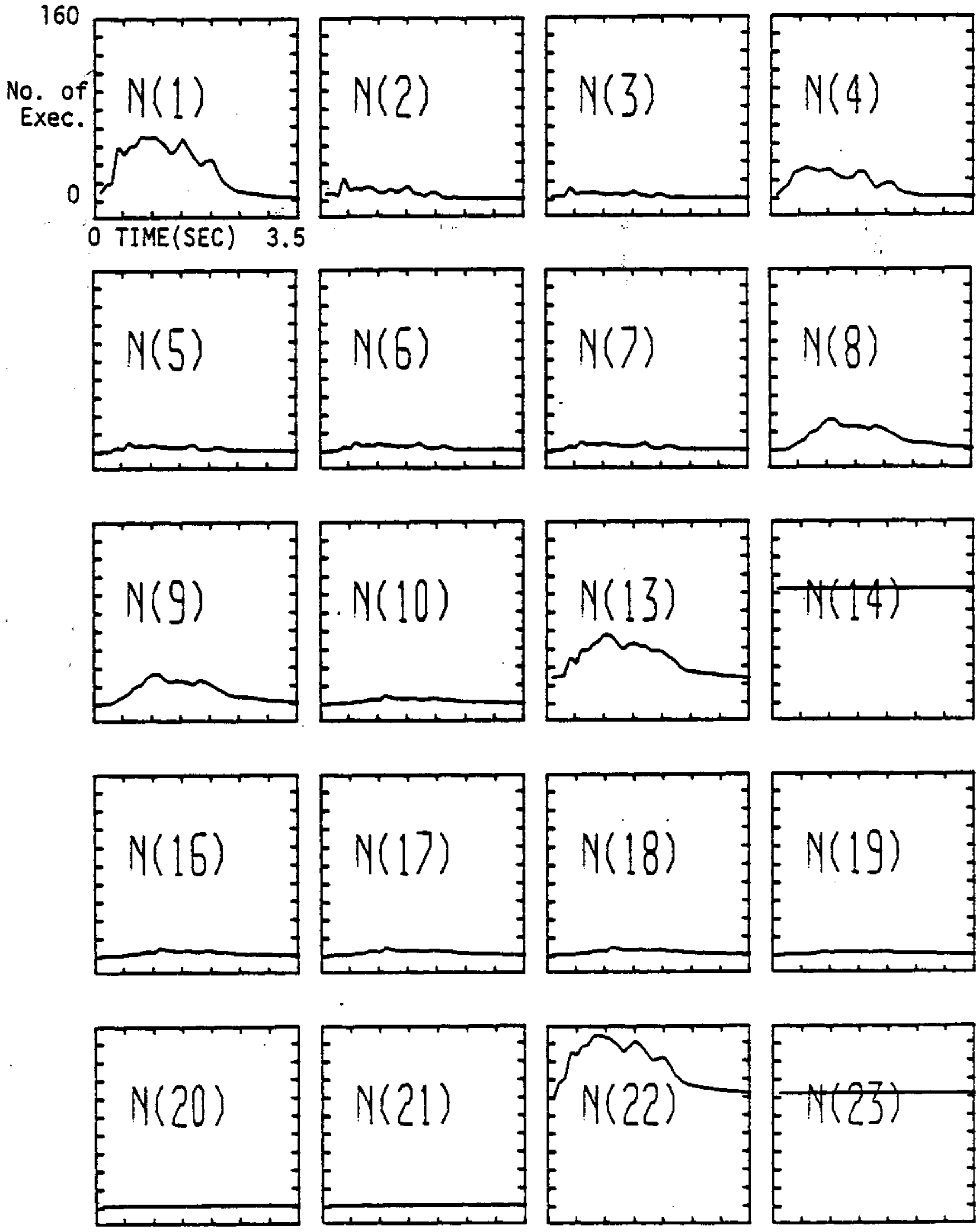


FIG. 2-3 NO. OF MODULE EXECUTIONS

Module M_1 is executed once for each radar return except those SEARCH returns with *no* detection. So, Figure 2-1 reflects offered load: loading increases as more objects are detected and loading decreases when no more object detections occur (about 2 seconds after the first detection). Figure 2-2 shows a similar trend for M_8 but with fewer module executions. This is because only Precision-Tracking radar returns invoke M_8 (see Figures A-4 and A-7) while SEARCH returns with detection, VERIFY returns, Coarse-Tracking returns, and DISCRI (discrimination) returns passed down from M_1 would invoke other modules (M_2, M_4, or the pair M_{16} & M_{17}).

Table 2-1 shows the IMC between the module pairs, measured in words sent *per execution* of the sending module. A blank entry in this matrix means that no message words are sent out from the sending module. In most cases, a module execution results in sending a *constant* number of words to other modules, so $v'_{ij}(t_k,t_{k+1})$ simplifies to v'_{ij}. For example, a module M_2 execution always causes 13 words to be sent to M_3 (10 words for file updates and 3 words for message header). However, there are cases in which v'_{ij} is not a constant (e.g. $v'_{14,23}$). In these cases, the matrix entry has two numbers.

2.2 IMC VS. TIME

In our simulation, we measured IMC in 100 ms intervals for each pair of modules. Figure 2-4 shows the IMC for the file updates sent by module M_8 to M_9. (This result is obtained from the same simulation which generated the

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1																							
2			0/13																				
3													15										
4					0/28	0/28	0/28						0/10										
5							6																
6							6																
7													13										
8						126/149			23/46	126/149			0/10			126/149	23/46		23/46	23/46			
9													13										
10																							
11																							
12																							
13														13/503									
14													6										13/23
15																							
16																		8					
17																		8					
18								0/6											0/28	0/103			
19																				103			
20																					103		
21																							
22	35	35		35				35															0/26
23																							

TABLE 2-1. INTERMODULE COMMUNICATION (WORDS) PER ENABLEMENT

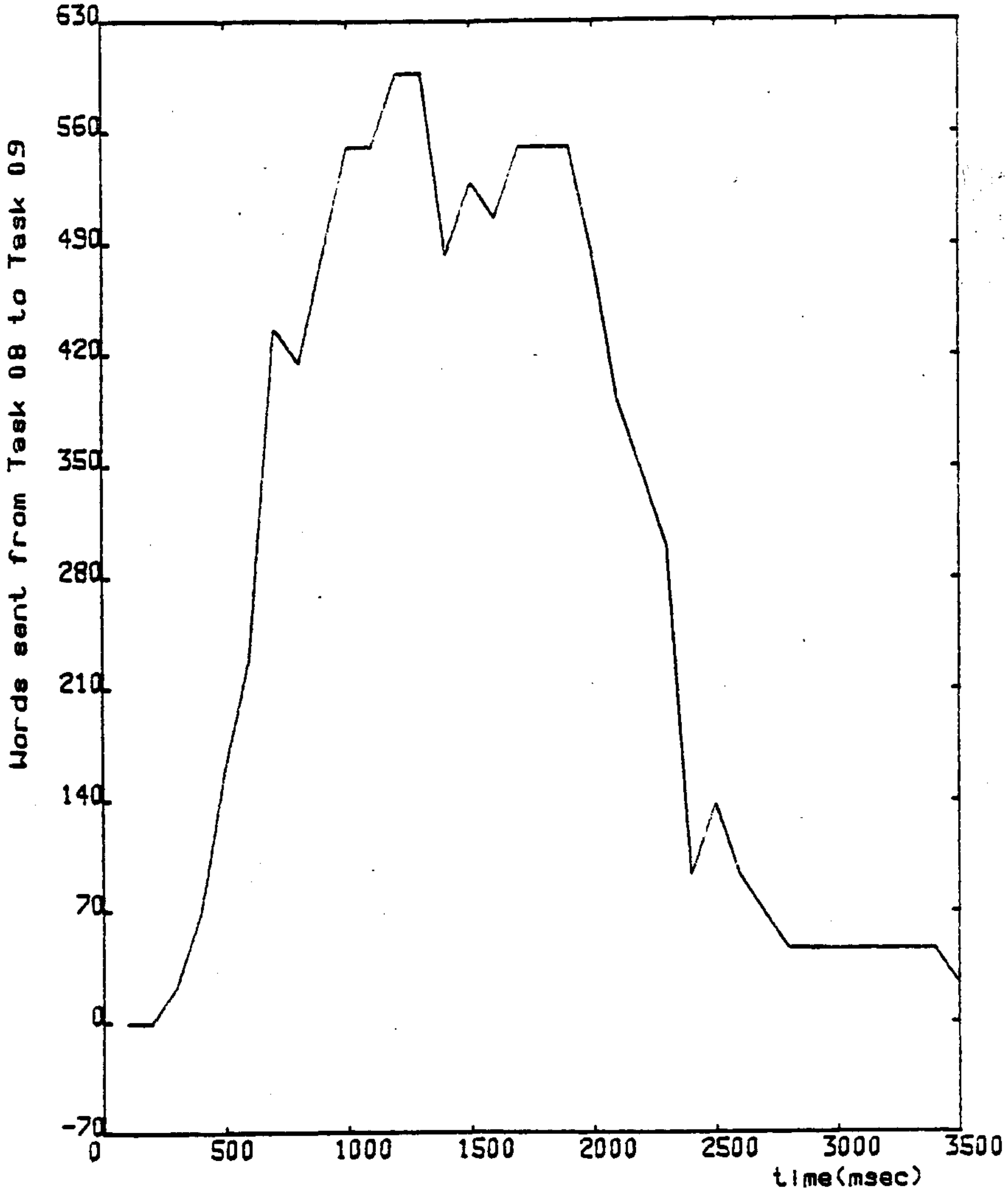


FIGURE 2-4. IMC (08,09) - 1.5 MIPS, 40 OBJECTS (SINGLE RUN)

curves in Figures 2-1 and 2-2.) Note that the activities of M_8 increase from tactical time 0 (when the system is empty) to a peak point at about 2 seconds (when all objects in the scenario move out of the surveillance area). This curve can be divided into three phases:

1. Initial phase. Objects are initially detected and kept in track, and thus the *image arrival rate* is greater than the *image drop rate,* where the image arrival rate is defined as the rate at which images are changed from TRACKI (track-initiate) stage into TRACK (precision tracking) stage, and the image drop rate is defined as the rate at which TRACK images are dropped from further tracking due to image redundancy or image's not being classified as one of the top five threats.

2. Steady phase. After 1.2 seconds, fewer objects are detected (about 2 or 3 every 100 ms) and the image arrival rate is almost equal to the drop rate.

3. Trailing phase. After 2 s, no images arrive but many are dropped.

Two dips appear in the steady phase because the *image drop rate* is slightly higher than the *image arrival rate* during those two short periods. Clearly, the curve shape is highly scenario-dependent.

The curve in Figure 2-4 is similar [1] to the one in Figure 2-2 which shows the number of enablements for M_8, because $v'_{8,9}$ is a constant, 23 words, most of the time (See Figure 2-3).

2.3 EFFECT OF SIMULATOR RANDOMNESS ON IMC

In the UCLA DPAD simulator, random numbers are used in determining events, such as

Cross Traffic Rejection

Impact Point Prediction

Known Object Recognition

Redundant Track Elimination

Passive Object Discrimination

We have enhanced the simulator to permit multiple runs, each with a different starting seed for the random number generator. Figure 2-5 shows five curves as the results of five such *replication runs*.

Since IMC varies with cross traffic rejection, impact point prediction, etc., we are interested in the *mean* IMC and the associated 90% *confidence interval* (C.I.). This confidence-interval approach will be used for all of the measurements in this study, e.g., the number of module enablements

[1] The difference in curve shapes results mainly from the time lapse between the instant that module M_8 is enabled and the instant that this enablement's execution is finished and messages are sent. That is, an enablement may be counted in a 100 ms interval, but its execution is finished and the IMC incurred and counted in the next 100 ms interval.

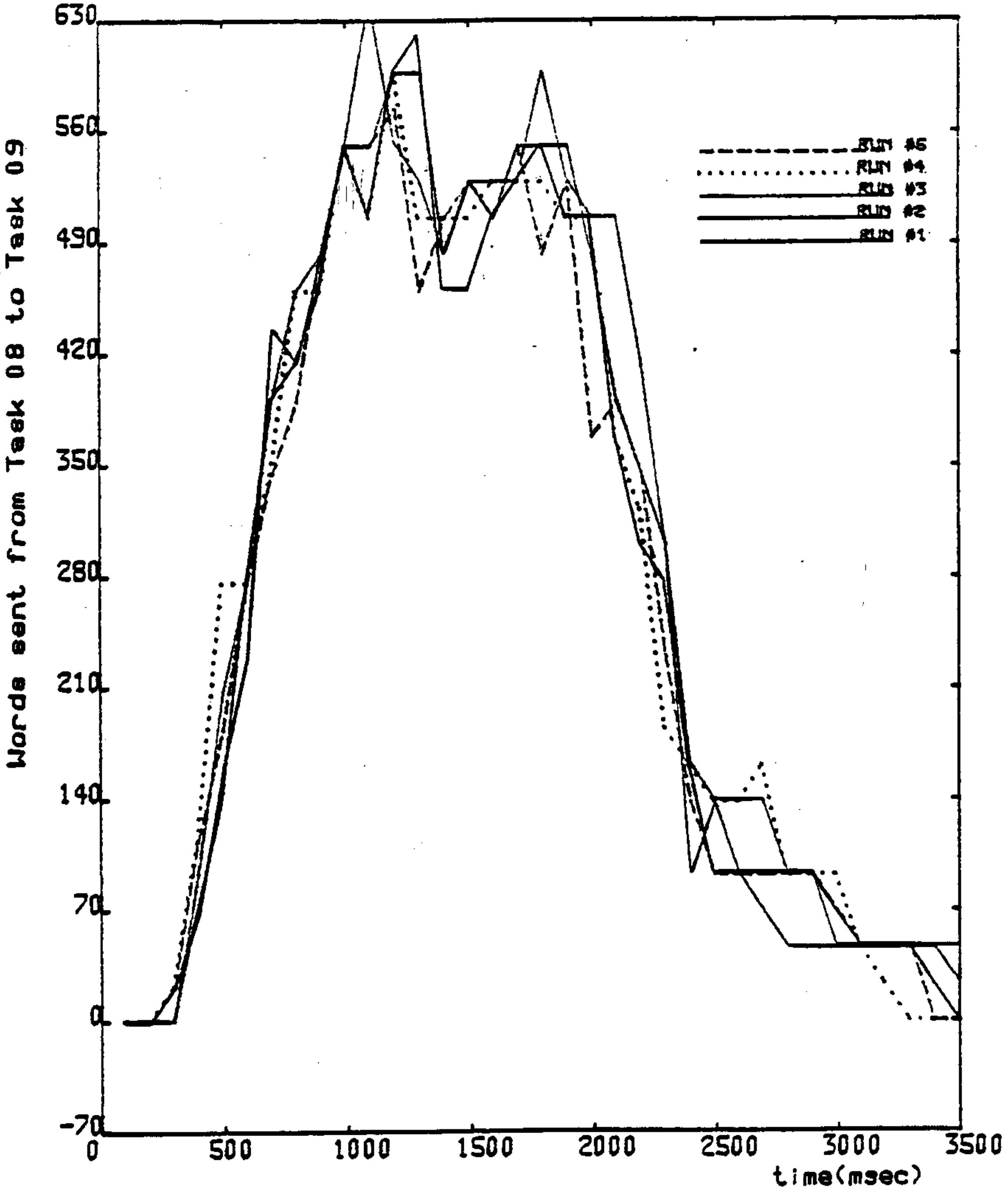


FIGURE 2-5. IMC (08,09) - 1.5 MIPS, 40 OBJECTS (5 RUNS)

presented previously in Section 2.1 and the port-to-port times presented in Section 2.1. Figure 2-6 shows the means and confidence intervals, for the five curves in Figure 2-5.

Many C.I.'s in Figure 2-6 are quite large. Since this mean IMC curve for 5 replication runs is not precise enough, 10 replications were performed as shown in Figure 2-7. Comparing Figure 2-7 with Figure 2-6, we note that the width of the C.I.'s have been reduced almost by 50%. [1]

2.4 IMC FOR VARIOUS TASK THREADS

Where there is an image detected, first there is processing in the Search/Verify (S/V) thread and then in the Coarse-Tracking thread (See Figures A-4 and A-7). For both of these threads, activity drops very quickly when there is no new image arrival, i.e., it is sensitive to image arrivals. That is why the plot of IMC between modules 2 and 3 over time, IMC(02,03), for the S/V thread shows a saw-toothed shape (Figure 2-8).

On the other hand, the activity for Precision-Tracking thread rises steadily (as shown in Figure 2-4) because a detected image is held in this thread until dropped by redundant track elimination or passive object discrimination. At about 1.2 s of the tactical time, activity reaches a constant level. And then, at about 2 s when all objects leave the surveillance area, the

[1] The confidence intervals for 15 replication runs were also derived, but they showed only slight improvement over the 10-run results.

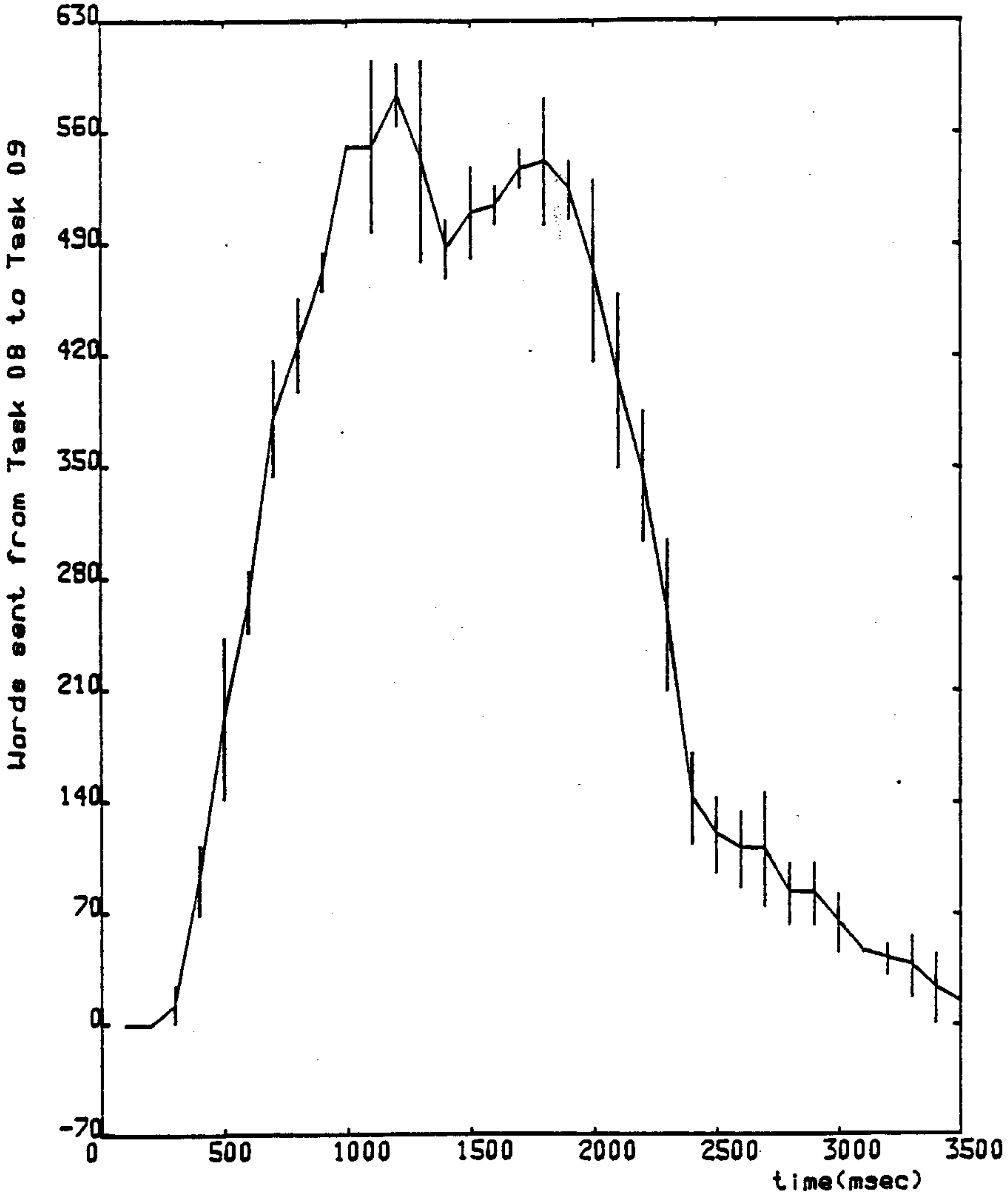


FIGURE 2-6. IMC (08,09) WITH C.I. FROM 5 RUNS - 1.5 MIPS, 40 OBJECTS

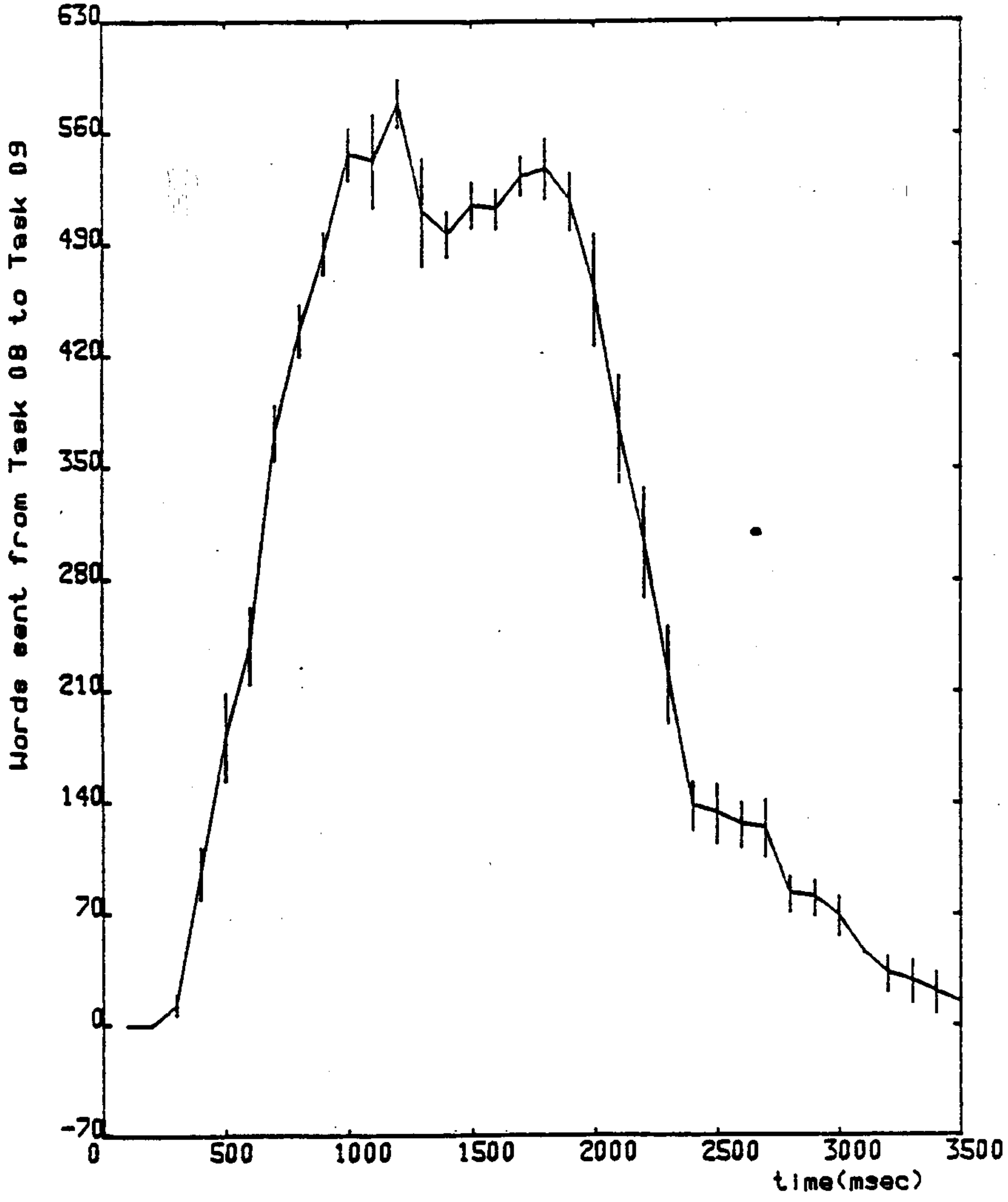


FIGURE 2-7. IMC (08,09) WITH C.I. FROM 10 RUNS - 1.5 MIPS, 40 OBJECTS

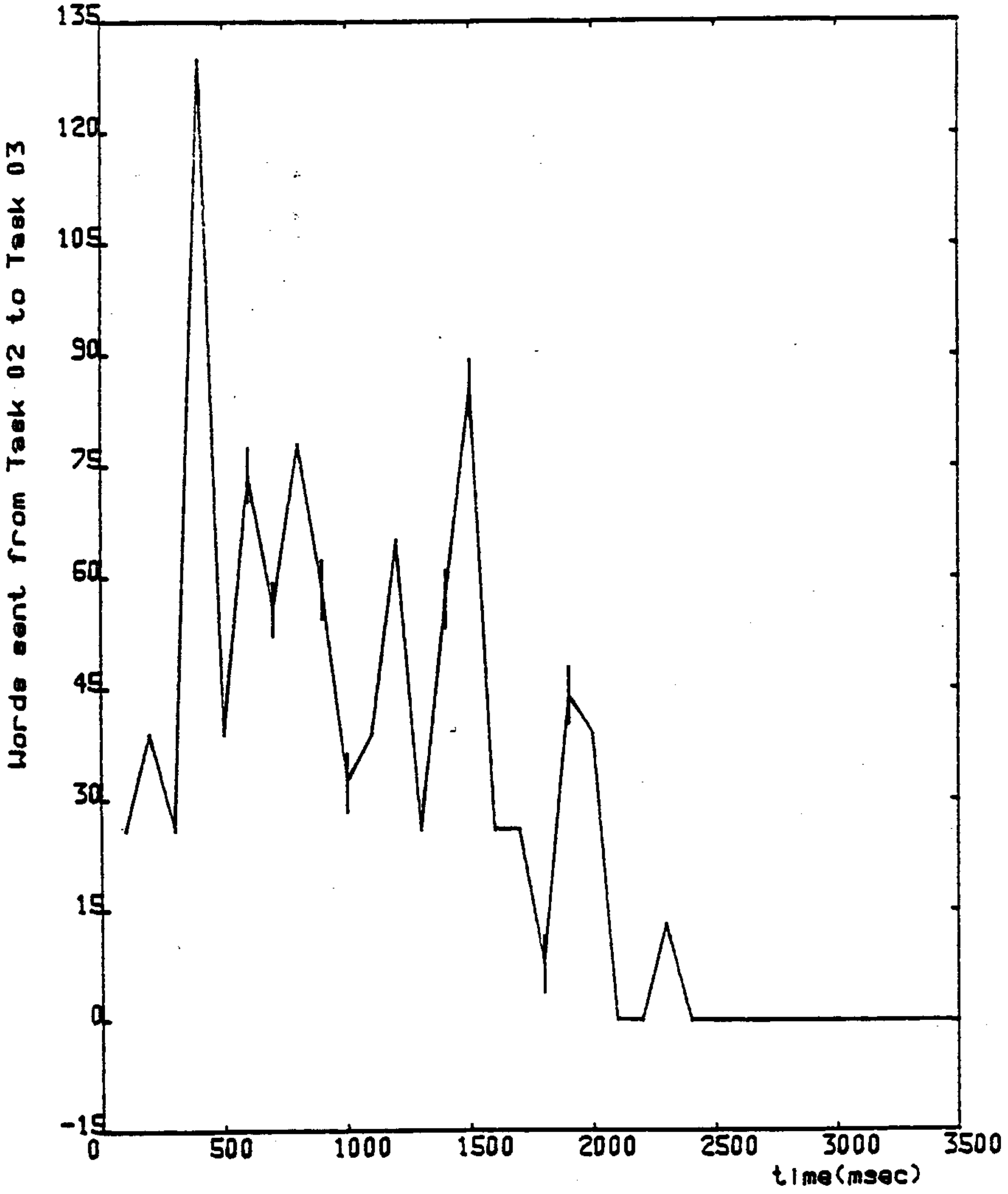


FIGURE 2-8. IMC (02,03) WITH C.I. FROM 10 RUNS - 1.5 MIPS, 40 OBJECTS

activity level enters the trailing phase.

2.5 EFFECT OF NUMBER OF OBJECTS ON IMC

Figure 2-9 shows how the IMC changes with number of objects in the scenario. In general, the increasing number of objects increases IMC [1] because the larger object population results in more activity for each processing thread and, thus, more shared-file updates.

Let us examine the region between 2 and 3 tactical seconds. In our experiment simulating three computers, each running at 1.5 MIPS rate, the 40-object runs generate heavy load on the computers. Thus, one or more of these computers are occasionally saturated during that period, and certain *module enablements are discarded because of the enablement queue overflow.* Therefore, some activities are eliminated undesirably and the measured IMC are smaller than they should be. (The measurement for 40 objects is somewhat inaccurate during that period). On the other hand, the 30, 20, 10, and 0-object runs do not saturate any computer's load. As a result, the IMC for 40 objects are smaller than that for 30 and 20 objects during that period.

Since we already had a feeling about the statistical variation of the IMC for ten replication runs, only sample means of IMC will be presented for the rest of IMC discussions; the confidence intervals will not be shown.

[1] One exception in the simulation is the IMC transferred from module M_{13} to M_{14}. See the last paragraph of this section.

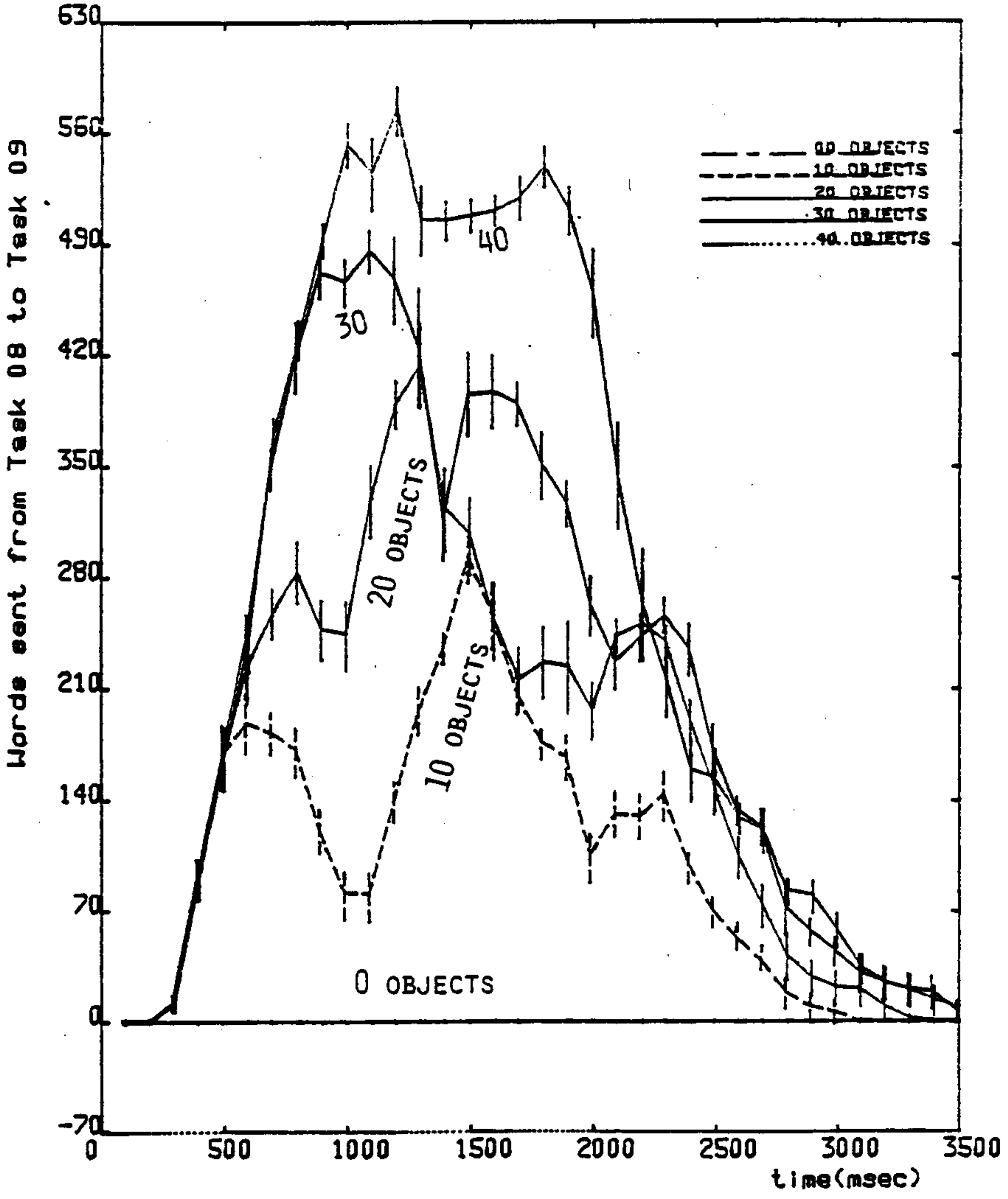


FIGURE 2-9. IMC (08,09) FOR VARIOUS NUMBER OF OBJECTS

Figure 2-10 shows the IMC for another communicating path, from M_8 to M_6. This IMC(08,06) is similar to IMC(08,09) as shown in Figure 2-9, but the ratio of their IMC sizes is about 5:1. We now briefly explain this similarity and the scale difference.

Module M_8 processes a ***precision track*** radar return and updates a record (20 words long) in the OBJSTV file (Object State Vectors). It then distributes that updated record to modules M_6, M_{10}, M_{16}, M_9, M_{17}, M_{19}, and M_{20}. Almost every time it updates OBJSTV, it also updates a record (100 words long) in file COVMTX (Covariance Matrix) and distributes the updated record to Modules M_6, M_{10}, and M_{16}. Therefore,

$$\text{IMC}(08,06) \equiv \text{IMC}(08,10) \equiv \text{IMC}(08,16)$$

and

$$\text{IMC}(08,09) \equiv \text{IMC}(08,17) \equiv \text{IMC}(08,19) \equiv \text{IMC}(08,20)$$

as we have observed; the symbol " $\equiv$" here means that two corresponding figures are identical. (See Figure 2-11 in the next section).

Since those OBJSTV updates distributed to Module M_9 (i.e., IMC(08,09)) are also distributed to M_6, M_{10}, and M_{16}, IMC(08,06) should have been ***6 times*** (100+20 to 20) as large as IMC(08,09). However, there are some occasions when OBJSTV records are updated and no COVMTX update follows. This explains why the scale ratio of the above two figures is about 5:1 instead of 6:1.

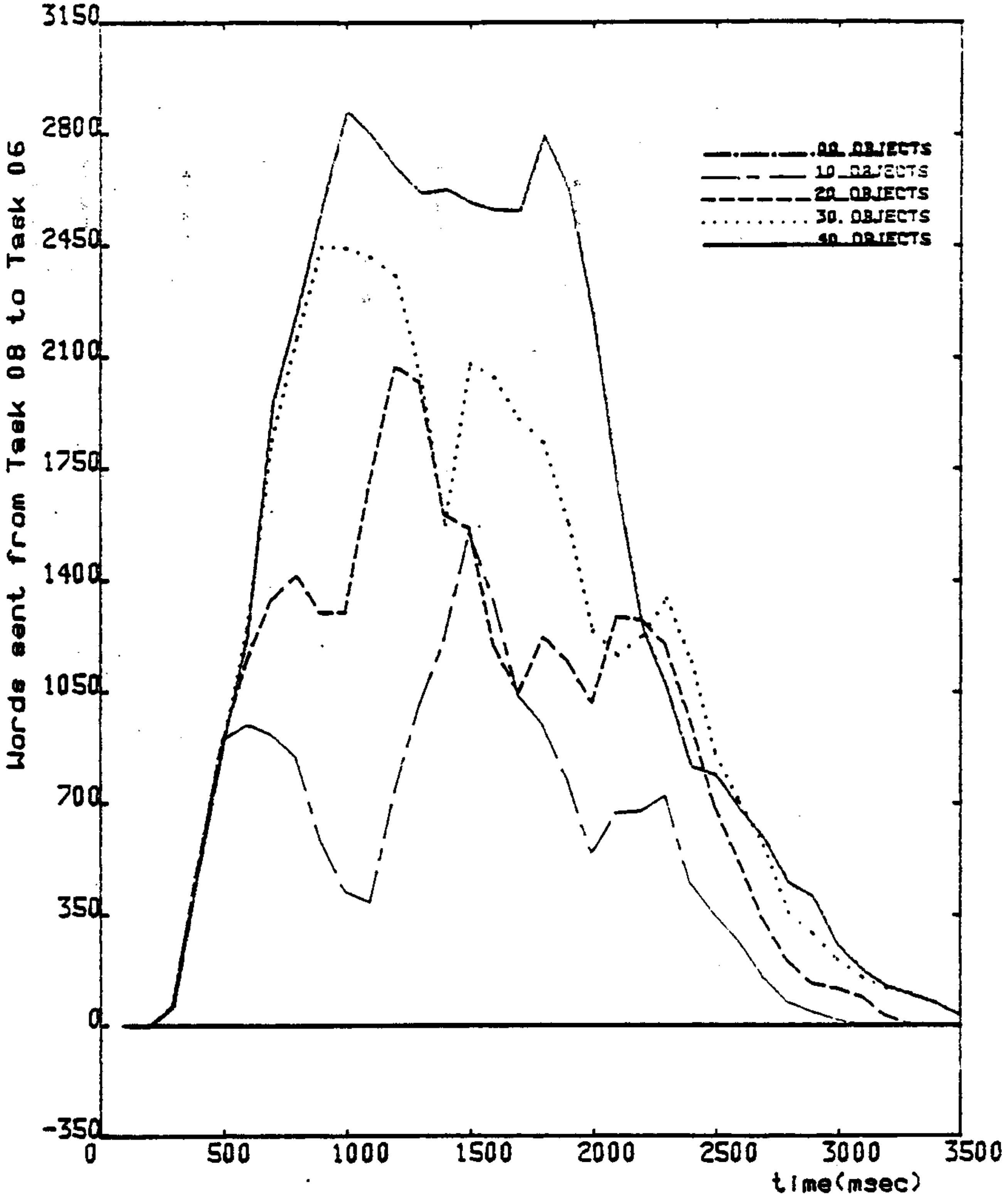


FIGURE 2-10. IMC (08,06) FOR VARIOUS NUMBER OF OBJECTS

Although IMC increases in general with the number of objects, IMC(13,14) is an exception. This IMC remains constant when the number of objects is varied. This is due to the fact that 50 radar scheduling commands (a constant amount of messages) are sent to M_{14} for each module 13 execution, which occurs every 5 ms. This radar scheduling rate does not change with the number of objects.

2.6 DOMINATING IMC

In the last section, we noticed the large difference between IMC(08,06) and IMC(08,09). This motivated us to compare the relative IMC sizes among all module pairs. All those pairs with non-zero IMC are shown in Figure 2-11, where they are all plotted on the same scale to reveal the relative magnitude. These are the *average* IMC obtained from ten replications (see Section 2.3). The IMC_{ij} varies with module pair and time. For example, $IMC_{8,6}$ is quite large during 1 to 2 seconds, $IMC_{6,7}$ is always small, and $IMC_{13,14}$ is always large. This figure discloses those pairs of communicating modules with dominating IMC. The dominating pairs are:

(13,14)

(08,06), (08,10), (08,16)

(22,01), (22,02), (22,04), (22,08)

(14,23)

(14,13)

(08,09), (08,17), (08,19), (08,20)

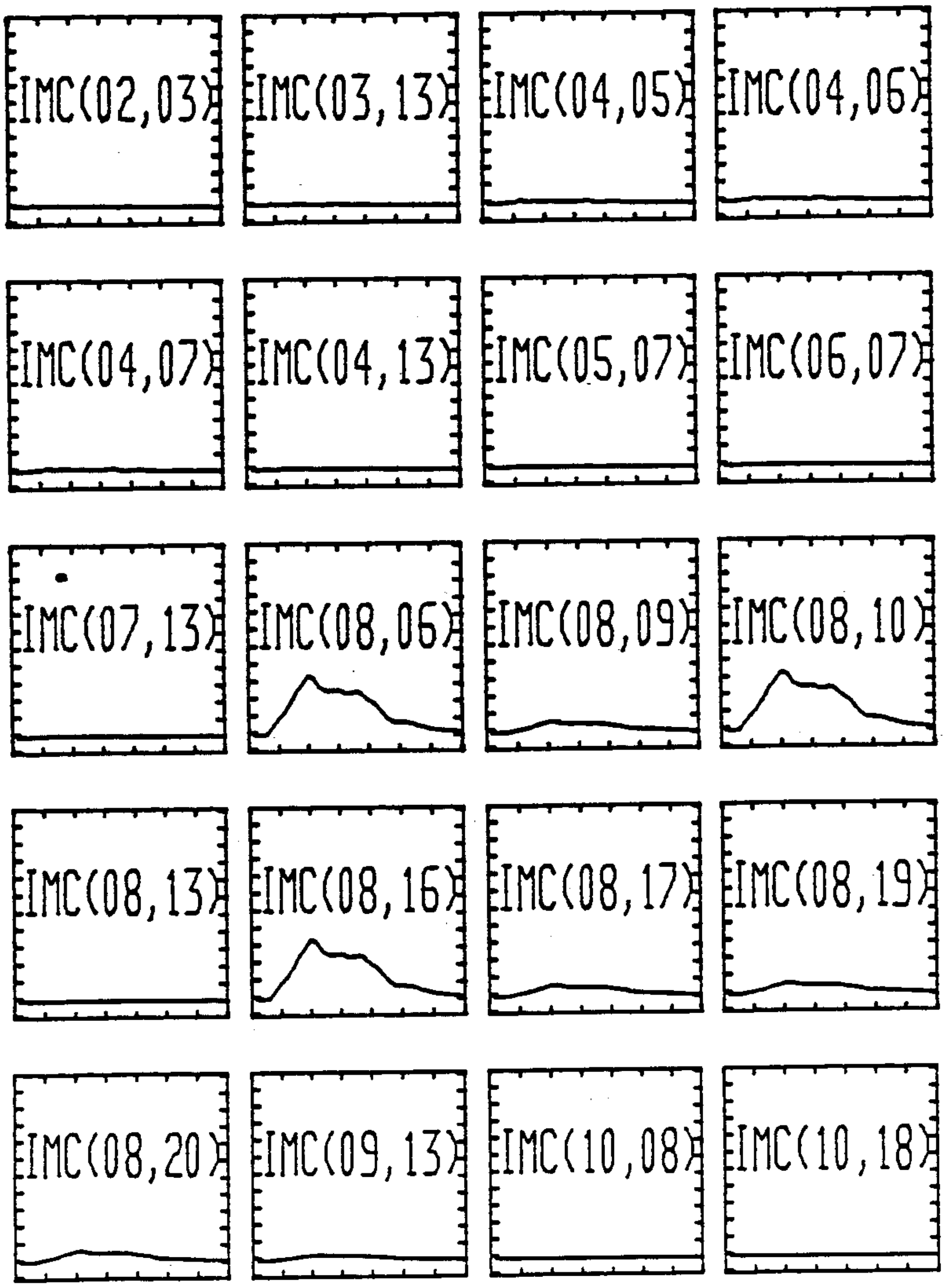


FIG. 2-11. IMC BETWEEN MODULE PAIRS

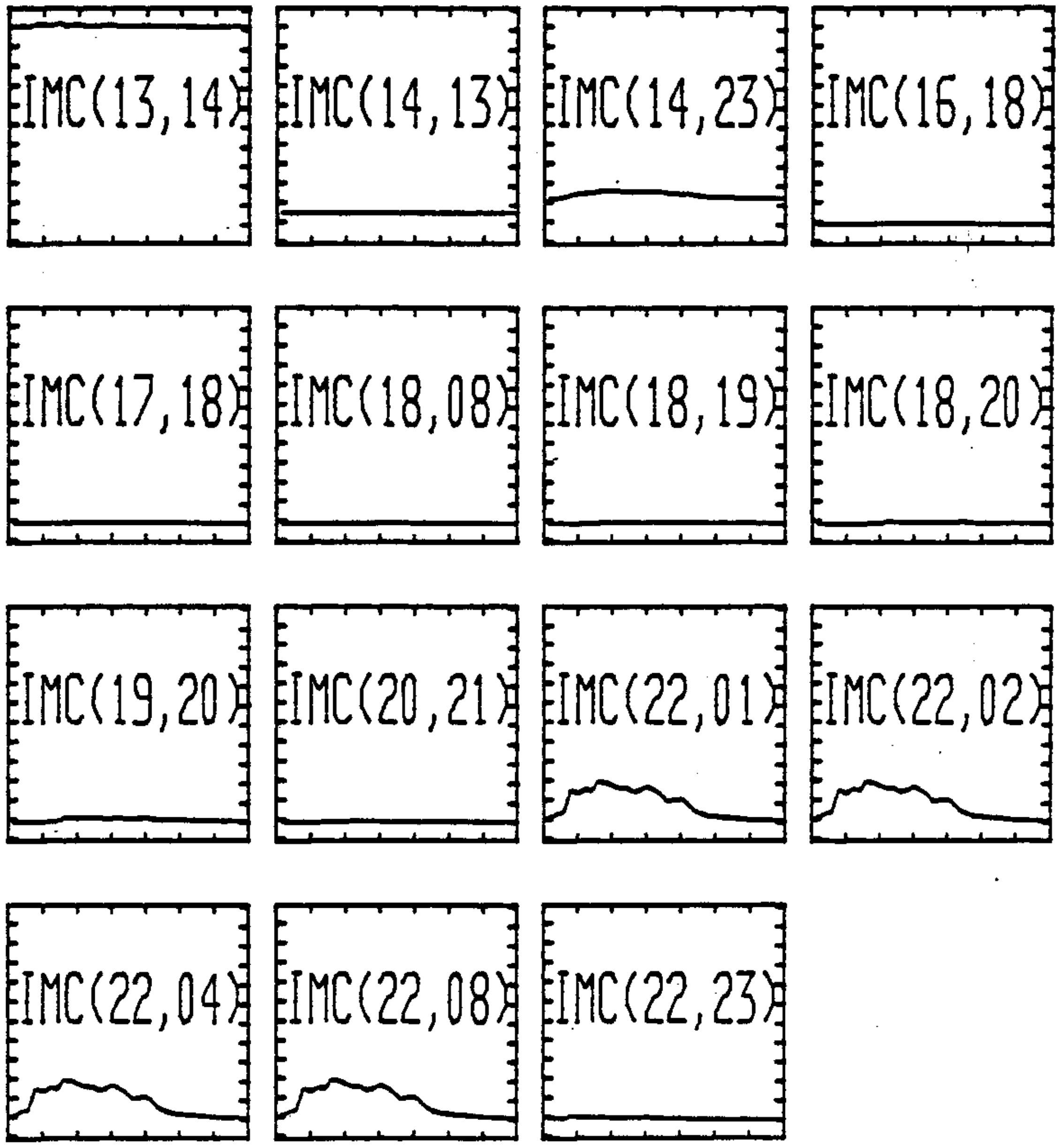


FIG. 2-11. (CONTINUED)

If we enlarge all these small figures, we can see that, just like the IMC(13,14) described before, IMC(14,13) is constant independent of the time. This is again due to the constant rate of radar command scheduling. IMC for all other module pairs varies with the time, (e.g. see the IMC(08,09) and IMC(02,03) shown before).

2.7 ALTERNATIVE REPRESENTATION OF IMC

Since the measured IMC is caused by file updates, there is an alternative way to present the IMC — IMC between a module and a file it writes (updates) or reads. The IMC from a module to a file it writes is referred to as *Module-to-File IMC,* or M-F IMC. The IMC from a file to a module which reads the file is referred to as *File-to-Module IMC,* or F-M IMC.

There is no F-M IMC in the DPAD system because its modules only read local file copies. Figure 2-12 displays the average M-F IMC for the DPAD for the same ten replications that generate Figure 2-11. Each plot V(i,k) gives the IMC size produced by module M_i in order to update file F_k. Note that only twenty-six (26) V(i,k)'s are non-zero and thus shown here. Again, they are all plotted on the same scale to reveal the relative magnitude.

2.8 ACCUMULATIVE EXECUTION TIME

As stated in Section 2.1, a program module M_i might be executed *multiple* number of times during a time interval (t_h, t_{h+1}). For each time of its

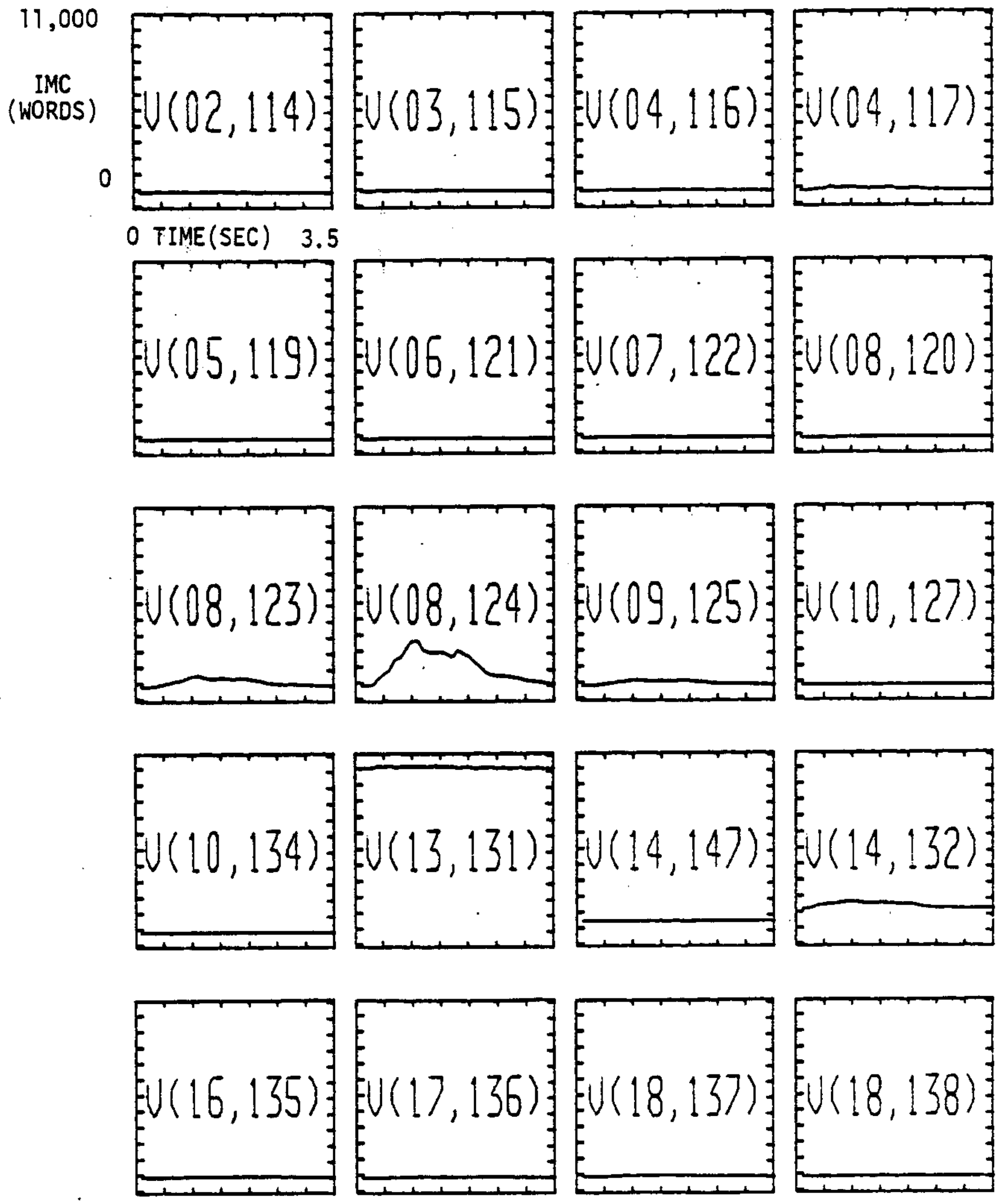


FIG.2-12 MODULE-FILE (M-F) IMC FOR THE DPAD

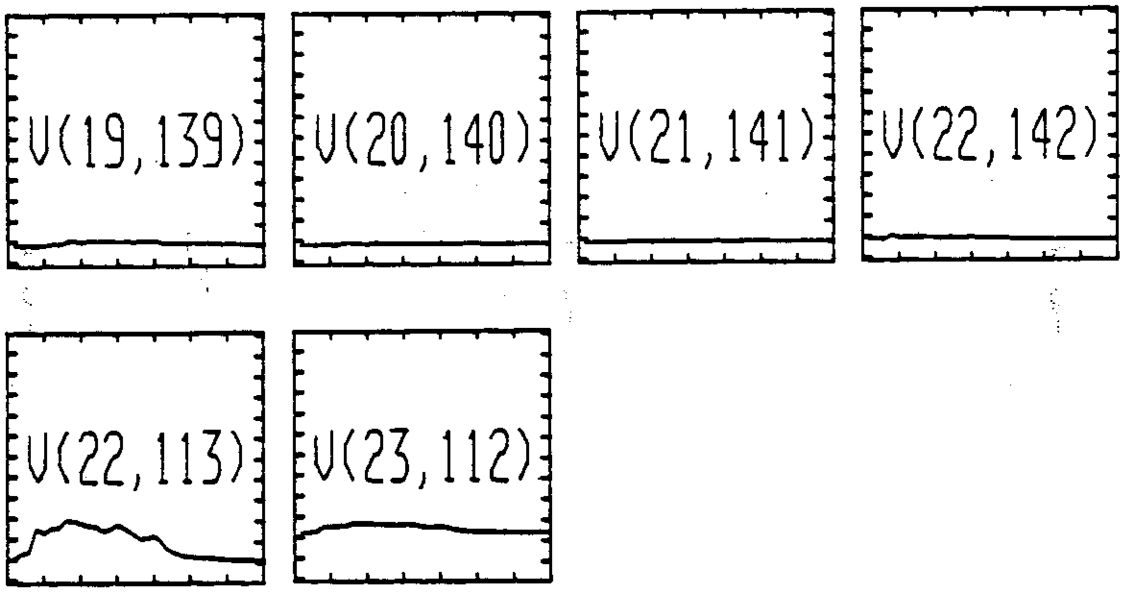


FIG. 2-12. (CONTINUED)

execution, its execution time (ET) contributes to the CPU load. An important piece of information which can be measured from the DPAD simulator is the *accumulative* load caused by M_i due to its multiple times of execution during a time interval. Let us call it *accumulative execution time* (AET) for M_i, denoted as $T_i(t_k,t_{k+1})$. Both the average ET and AET can be expressed in *machine language instructions* (MLI) executed. Although the execution time of one machine-language instruction varies from instruction to instruction, we can find the *mean* instruction execution time given the mix ratios for various different instructions. Our measurement results are given in Figure 2-13 where each plot T(i) shows the module AET $T_i(t_k,t_k+100\,ms)$ for module M_i during all 100 ms intervals.

2.9 EFFECT OF MODULE ASSIGNMENT ON MEASUREMENTS

In general, IMC and AET change with different module assignments. However, our study shows that the number of module executions, IMC, and AET are all almost independent of module assignments if a *fixed* offered load is input to the distributed system. For example, Figure 2-14 shows the number of times module M_8 executes during 100 ms intervals under a scenario with 40 objects in the DPAD simulation; Figure 2-15 displays the AET for M_8 during the intervals; and Figure 2-16 exhibits the IMC from M_{22} to M_4 measured from the DPAD simulation. In each figure, five curves represent five different module assignments. Note that the five curves are so close to each other. This is a very important observation which makes the module

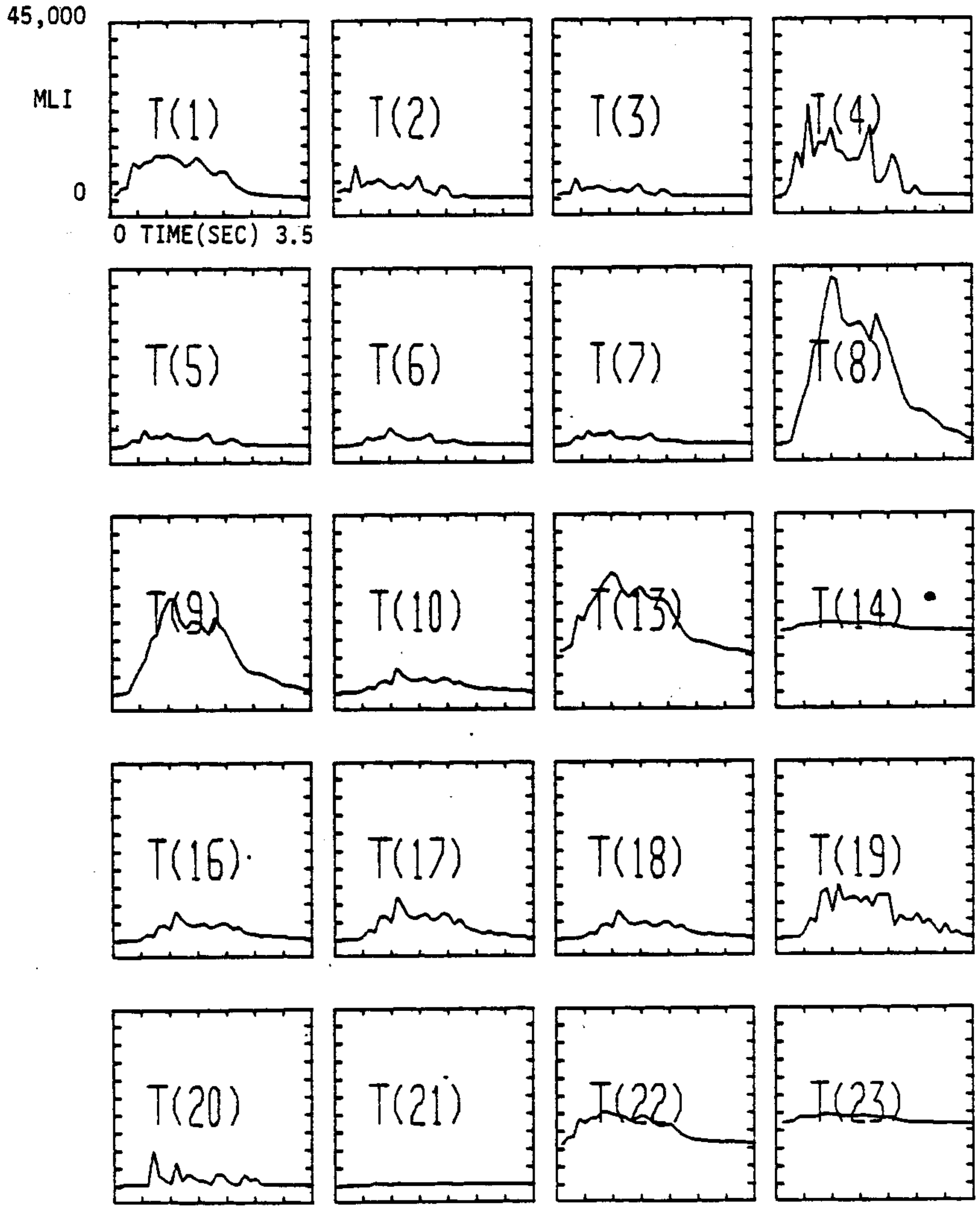


FIG. 2-13 ACCUMULATIVE MODULE EXECUTION TIME

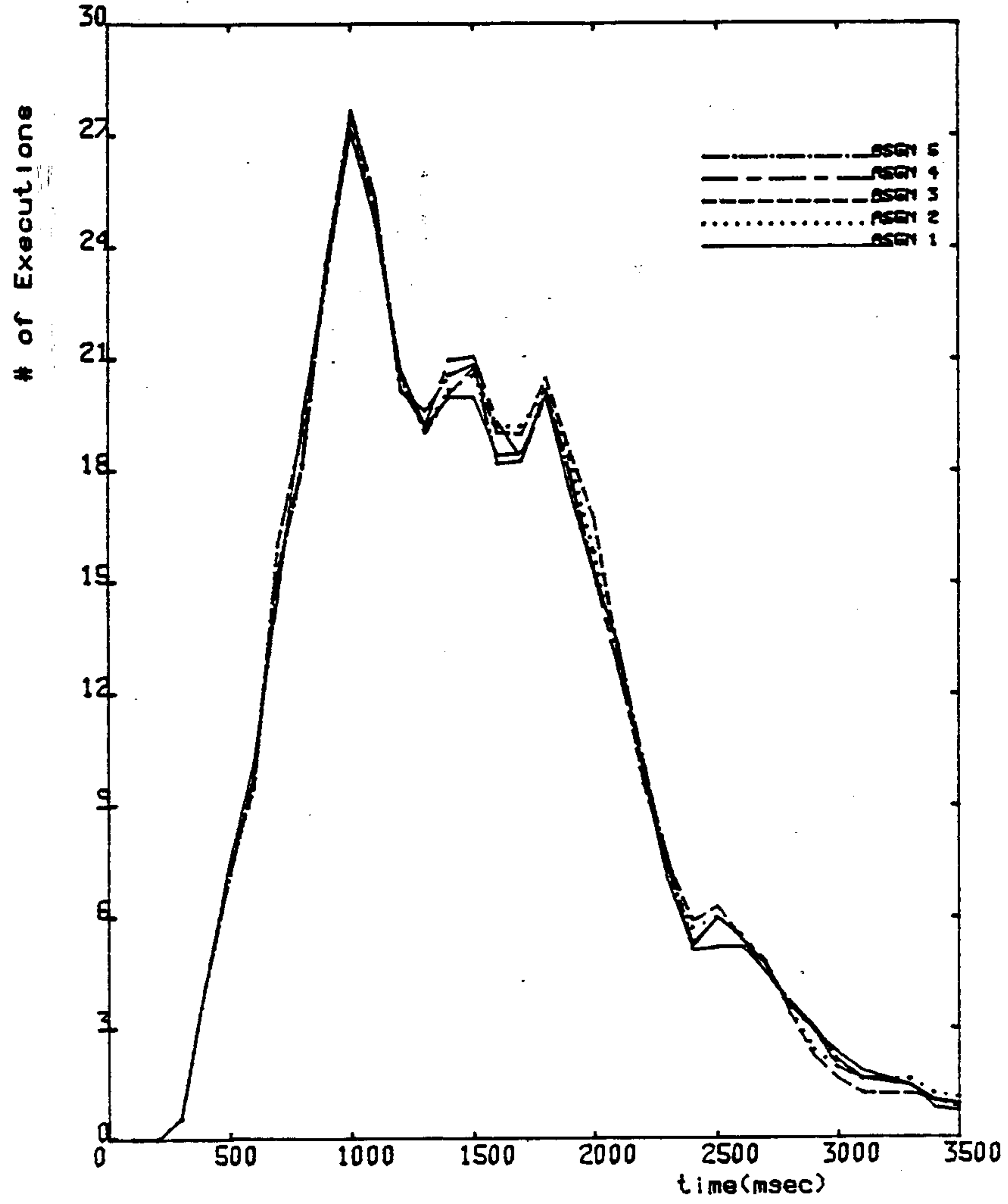


FIG. 2-14 NUMBER OF M_8 ENABLEMENTS FOR VARIUOS MODULE ASSIGNMENTS

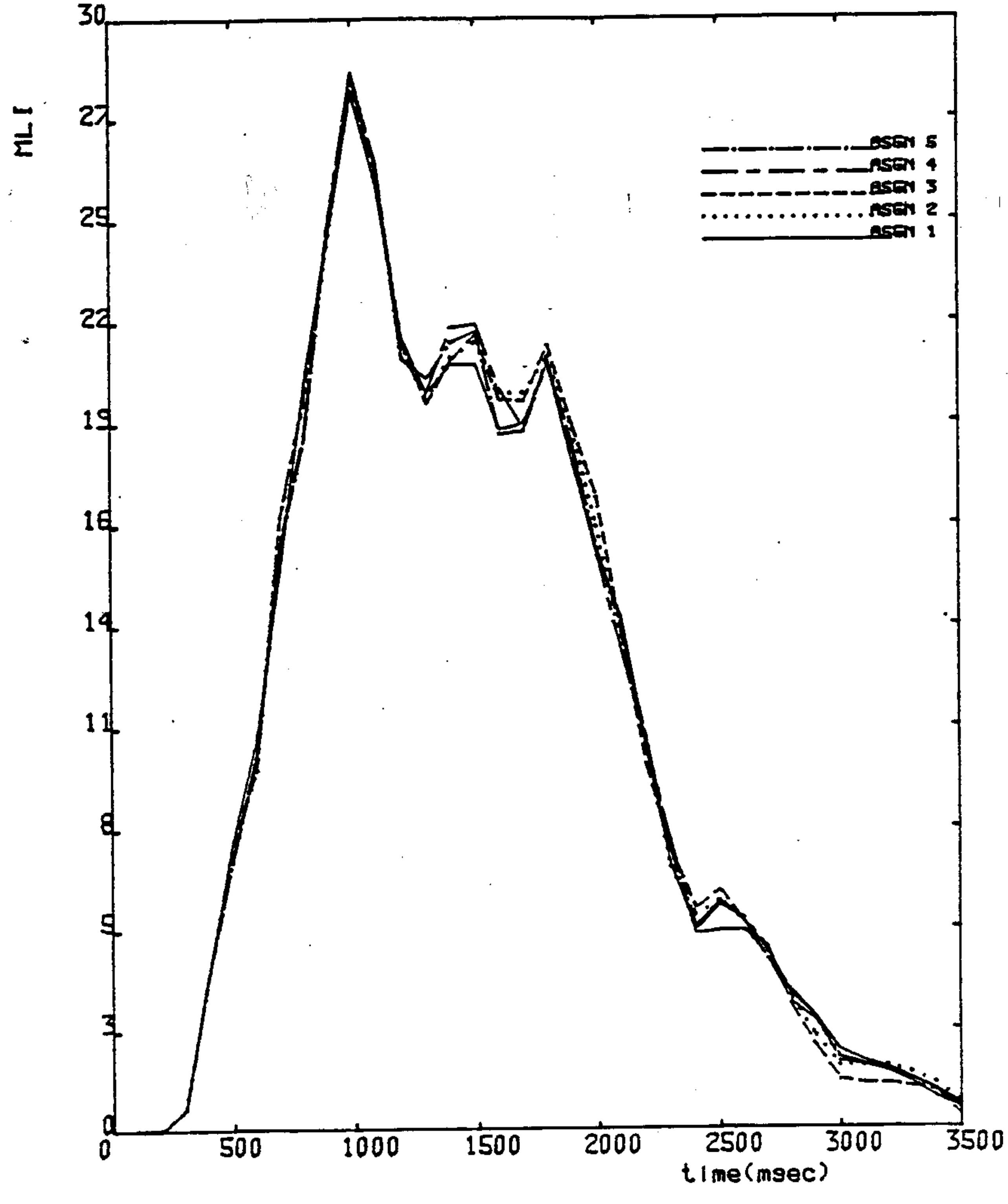


FIG. 2-15. AET OF M_8 FOR VARIOUS MODULE ASSIGNMENTS

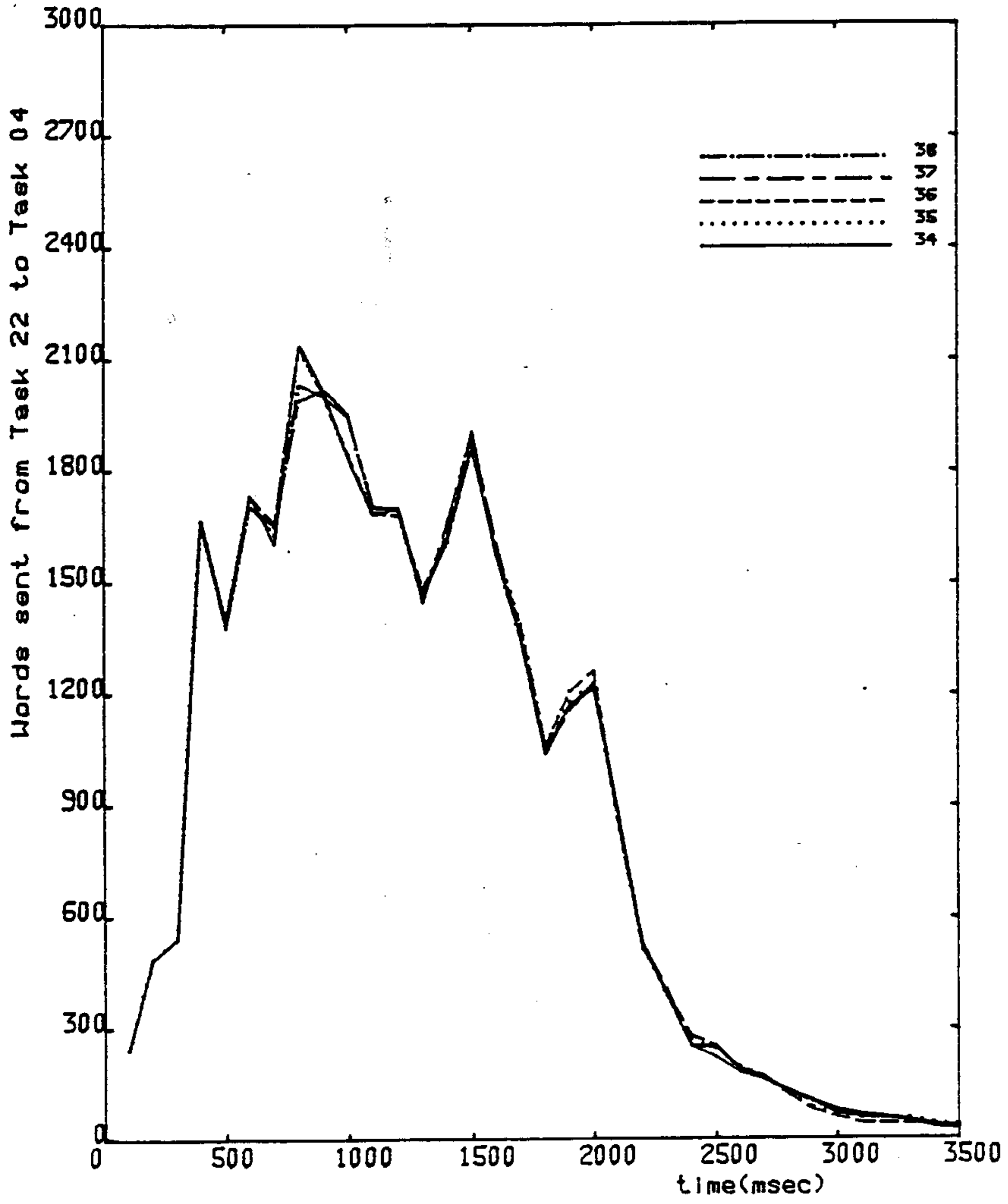


FIG. 2-16. IMC(22,4) FOR VARIOUS ASSIGNMENTS

assignment problem a lot easier.

2.10 CONCLUSIONS

A method for presenting both the IMC and AET has been shown in this chapter. We found that both IMC and AET vary with time, number of objects in the scenario, task threads, and module pairs. However, the measurements indicate that IMC and AET exhibit almost no variation with various module assignments. Both IMC and AET data are important parameters for module assignments.

Using simulation experiments to characterize IMC is quite time-consuming. Thus, we have developed an analytical model in the next chapter to estimate IMC based on the data volume sent per execution instance, branching probabilities for control flow, and number of objects.

CHAPTER 3

ANALYTICAL MODEL FOR IMC ESTIMATION

This chapter presents an analytical model for IMC estimation, which is developed to avoid the time-consuming coding and running of simulation experiments. A distributed application consists of J program modules M_1, M_2, ..., M_J on S computers. Graph models have been developed [BAER68] to study various aspects of computer systems and other more general systems. Figure 3-1 is the same as Figure A-8; it shows a portion of the control-and-data-flow graph of the DPAD system. The granularity of modules is at the subroutine level rather than the instruction level as is being used in data flow machines [TREL82]. A control-and-data-flow graph is a control flow graph superimposed by files and the associated data flow between modules and files. The graph has an entry point and a termination point [CHU80]. Each square box represents a computational module, and each directed *arc* represents the precedence relationship between the two connected modules. The sequence of module execution is referred to as the *control flow*. Module M_j causes M_n to follow it by sending an *enablement* to M_n's computer which is referred to as "M_j enables M_n". *Data flow* is represented by broken lines and consists of reads and writes by modules to the set of K shared data files F_1, F_2, ..., F_K.

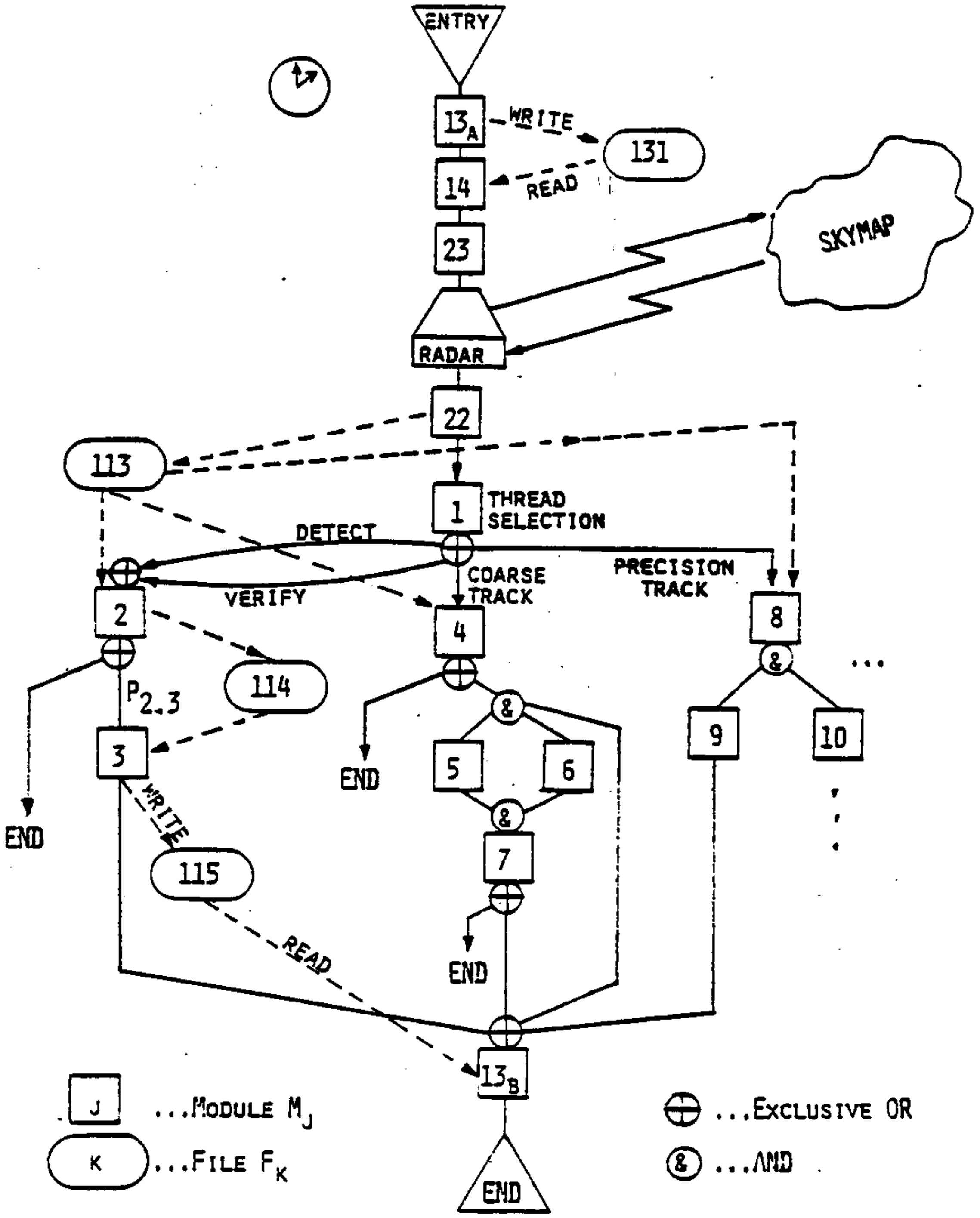


FIG. 3-1. PART OF THE CONTROL-AND-DATA-FLOW GRAPH FOR THE DPAD SYSTEM

Modules communicate through shared files and module enablements; such communication is referred to as *Intermodule communication (IMC).*

Our model for estimating the IMC between each communicating pair of modules is presented in Sections 3.1 and 3.2 where it is assumed that 1) the distributed program has been partitioned, in terms of functions, into a set of modules, 2) the function of each module is fully specified, and 3) the files used by each module are known. Section 3.1 develops a methodology to estimate all key parameters in terms of number of module executions. Section 3.2 shows how to obtain the number of module executions for each module in the control flow. Next, in Section 3.3, this model is applied to the DPAD. The estimated results are compared with those measurement results presented in Chapter 2, as a validation of the IMC and AET model. Finally, the IMC incurred for control functions and system resource utilization in distributed processing systems are discussed.

3.1 ESTIMATION OF KEY PARAMETERS

Three types of IMC exist: Module-to-Module (M-M) for module enablements, Module-to-File (M-F) for file updates (writes), and File-to-Module (F-M) for file reads (see Fig. 3-1). Here we estimate each for all specified time intervals (t_k, t_{k+1})'s. The interval size, $t_{k+1}-t_k$, for the estimation should be properly selected. Too short intervals prevent us from observing the trend of workload changes while too long intervals are not sensitive enough to reveal the dynamic behavior.

For M-M IMC, the number of module M_n enablements by M_j during time interval (t_k,t_{k+1}) is

$$N_j(t_k,t_{k+1})p_{jn}(E;t_k,t_{k+1})$$

where $N_j(t_k,t_{k+1})$ = *number of times module M_j executes* during (t_k,t_{k+1}), referred to as *execution frequency* of M_j

$p_{jn}(E;t_k,t_{k+1})$ = probability that an execution of M_j enables M_n , referred to as *enablement probability.*

Thus, the M-M IMC volume from M_j to M_n is

$$V_{jn}(E;t_k,t_{k+1}) = N_j(t_k,t_{k+1})p_{jn}(E;t_k,t_{k+1})L_{jn}(E;t_k,t_{k+1}) \tag{3-1}$$

where $L_{jn}(E;t_k,t_{k+1})$ = average number of words (enablement-message length) sent from M_j to M_n when M_j enables M_n.

For M-F IMC, the number of updates to F_k by M_j is

$$N_j(t_k,t_{k+1})p_{jk}(W;t_k,t_{k+1})$$

where $p_{jk}(W;t_k,t_{k+1})$ = probability that an execution of M_j updates file F_k . Therefore, the *IMC message volume* (number of words) sent from M_j to F_k is

$$V_{jk}(W;t_k,t_{k+1}) = N_j(t_k,t_{k+1})p_{jk}(W;t_k,t_{k+1})L_{jk}(W;t_k,t_{k+1}) \tag{3-2}$$

where $L_{jk}(W;t_k,t_{k+1})$ = average number of words (record length) written per update.

Similarly, the F-M message volume for read response sent from file F_k to M_j is

$$V_{jk}(R;t_k,t_{k+1}) = N_j(t_k,t_{k+1})p_{jk}(R;t_k,t_{k+1})L_{jk}(R;t_k,t_{k+1}) \tag{3-3}$$

where $p_{jk}(R;t_k,t_{k+1})$ = probability that an execution of M_j reads file F_k , and $L_{jk}(R;t_k,t_{k+1})$ = average number of words M_j reads from F_k .

Finally, the ***accumulative execution time*** (AET) for M_j during (t_k,t_{k+1}) is

$$T_j(t_k,t_{k+1}) = N_j(t_k,t_{k+1})x_j(t_k,t_{k+1}) \tag{3-4}$$

where $x_j(t_k,t_{k+1})$ = average ***execution time*** (ET) of M_j during (t_k,t_{k+1}). This time can be expressed in terms of machine-language instructions (MLI) executed.

Since eqs. (3-1) — (3-4) require values for $N_j(t_k,t_{k+1})$, we show how it can be estimated in the next section.

3.2 ESTIMATION OF $N_j(t_k,t_{k+1})$

To estimate $N_j(t_k,t_{k+1})$, we use only the control flow portion of the graph. There are three basic types of control flows: sequential, conditional branching (Exclusive OR), and parallel (AND). Loop control flow is a variant of conditional branching because the control is conditionally branched back to some previous point. Therefore, in this thesis, we assume control flow graphs with no loops.

Estimation for the execution frequency of module M_j within (t_k,t_{k+1}), $N_j(t_k,t_{k+1})$, requires knowledge of the enablement probabilities $p_{ij}(E;t_k,t_{k+1})$ for all incident arcs $(i\text{-}j)$'s from M_i's into M_j. Enablement probabilities must satisfy the following two conditions:

1. If more than one Exclusive-OR arc leaves M_i, then every such arc $i\text{-}m$ has an enablement probability $p_{im}(E;t_k,t_{k+1})$ such that $\sum_m p_{im}(E;t_k,t_{k+1}) = 1$.

2. If there are more than one AND arcs leaving M_i, each arc $i\text{-}m$ has $p_{im}(E;t_k,t_{k+1}) = 1$. Note that sequential flow is a special case of Exclusive-OR flow (and AND flow) where only one arc leaves M_i and $p_{im}(E;t_k,t_{k+1}) = 1$.

A control flow graph with no loops can be viewed as an acyclic directed graph and hence has a *partial ordering* of modules where the relation "partial order" is denoted as $M_i \prec M_j$ when M_i enables M_j. We can use the *topological sorting algorithm* [KNUT73] to embed the partial order of the J modules into a linear order and rename these modules into M_1, M_2, ..., M_J such that whenever M_i enables M_j, $i < j$. Let the entry node be M_0 and assume the number of entries into the control flow graph, $N_0(t_k,t_{k+1})$, is given. To estimate $N_j(t_k,t_{k+1})$'s, we consider the modules M_j's *in their ordered sequence* $j = 1, 2, ..., J$. Doing so ensures that for each j, $N_i(t_k,t_{k+1})$ is known if M_i enables M_j.

The following estimation rules are used:

1. If M_j has only one incident arc and this arc emits from M_i , then

$$N_j(t_k,t_{k+1}) \approx N_i(t_k-\Delta_{ij},t_{k+1}-\Delta_{ij})p_{ij}(E;t_k-\Delta_{ij},t_{k+1}-\Delta_{ij}) \tag{3-5}$$

where the notation "$\approx$" indicates an approximate estimation, and Δ_{ij} is the average M_i-to-M_j *enablement delay.* An enablement delay is the duration starting from when M_i completes its execution and enables M_j until when M_j completes its execution. This includes the wait time of M_j in scheduling queue plus its own execution time. Normally the module execution time is much smaller than the time interval $(t_{k+1}-t_k)$. (See the discussion on interval size at the beginning of Section 3.1.) If the wait time is also small, then Δ_{ij} is negligible compared to $(t_{k+1}-t_k)$ and eq. (3-5) reduces to

$$N_j(t_k,t_{k+1}) \approx N_i(t_k,t_{k+1})p_{ij}(E;t_k,t_{k+1}) \tag{3-6}$$

For example, if M_2 in Figure 3-1 executes 10 times within a time interval, $p_{2,3}(E) = 0.5$, and $\Delta_{2,3}$ is negligible, then M_3 executes $10 \times 0.5 = 5$ times within that same interval.

2. If multiple arcs $(i-j)$'s enter module M_j in the Exclusive-OR manner, then

$$N_j(t_k,t_{k+1}) \approx \sum_i N_i(t_k-\Delta_{ij},t_{k+1}-\Delta_{ij})p_{ij}(E;t_k-\Delta_{ij},t_{k+1}-\Delta_{ij}) \tag{3-7}$$

Eq. (3-7) *sums over* all the modules M_i which have an outgoing arc connected to module M_j since any of these M_i's might enable M_j. For example, consider M_{13} in Figure 3-1 and assume $\Delta_{i,13}$'s are negligible. Then

$$N_{13}(t_k,t_{k+1}) \approx N_3(t_k,t_{k+1}) + N_7(t_k,t_{k+1})p_{7,13}(E) + N_4(t_k,t_{k+1})p_{4,13}(E)$$
$$+ N_9(t_k,t_{k+1})$$

3. If multiple arcs $(i-j)$'s join at M_j in the AND manner, M_j cannot enter the scheduling queue until all its predecessor modules M_i's have completed their execution. If such a wait time is negligible, then

$$N_j(t_k,t_{k+1}) \approx \underset{i}{any}\, [N_i(t_k,t_{k+1})p_{ij}(E;t_k,t_{k+1})] \tag{3-8}$$

since all these predecessor modules enable M_j the same number of times. For example, for M_7 in Figure 3-1 we have $N_7(t_k,t_{k+1}) \approx N_5(t_k,t_{k+1})\times 1.0 \approx N_6(t_k,t_{k+1})\times 1.0$. If enablement delays are not small, then from the predecessors M_i's we must determine the module M_b that finishes execution last and eq. (3-8) becomes

$$N_j(t_k,t_{k+1}) \approx N_b(t_k-\Delta_{bj},\ t_{k+1}-\Delta_{bj}) \tag{3-9}$$

To compute all the $N_j(t_k,t_{k+1})$'s, we always start from the top of the control flow graph, obtaining them in the topologically sorted sequence. Once we know how to obtain all $N_j(t_k,t_{k+1})$'s, we can use eqs. (3-1) — (3-4) to estimate the desired IMC and AET profiles.

In the estimation of IMC and module AET, some values may be difficult to obtain, e.g., enablement probabilities $p_{ij}(E;t_k,t_{k+1})$'s and average module execution time x_j in eq. (3-4) for some modules. In these cases, a simulation can be used to estimate these values. Since such a simulation is applied to a smaller subsystem, it usually is much simpler than simulating the entire system.

3.3 AN EXAMPLE OF ANALYTICAL IMC ESTIMATION

In this section, we apply the estimation methodology of Sections 3.1 and 3.2 to DPAD Detect/Verify processing for modules M_{22}, M_1, M_2, and M_3. Search/Verify processing determines the presence of potentially threatening objects and consists of: 1) a radar-search return with object detection that causes modules M_{22}, M_1, and M_2 to execute in sequence (i.e., in their topologically sorted order), 2) a radar-verify delay of 9 to 15 ms, and 3) a radar-verify return that causes modules M_{22}, M_1, M_2, and M_3 to execute in sequence. A radar-search return with no detection is simply discarded by M_{22} and so M_1, M_2, and M_3 are not enabled. We shall estimate $N_2(t_k,t_{k+1})$ and the IMC written by M_2 which consists of 13 words to update file F_{114} (i.e., $L_{2,114}(W) = 13$) whenever M_2 enables M_3.

We let $N_j(d;t_k,t_{k+1})$ be the number of module M_j executions for Detect processing, $N_j(v;t_k,t_{k+1})$ be the M_j executions for Verify processing, and $N_j(t_k,t_{k+1})$ be the total M_j executions for both Detect and Verify processing.

Clearly,

$$N_j(t_k,t_{k+1}) = N_j(d;t_k,t_{k+1}) + N_j(v;t_k,t_{k+1})$$

Because each module's execution time is short (less than 1 ms) compared to the measurement interval ($t_{k+1}-t_k = 100$ ms),

$$N_j(d;t_k,t_{k+1}) \approx N_{22}(d;t_k,t_{k+1}) \qquad \text{for } j = 1, 2$$

and

$$N_j(v;t_k,t_{k+1}) \approx N_{22}(v;t_k,t_{k+1}) \qquad \text{for } j = 1, 2, 3$$

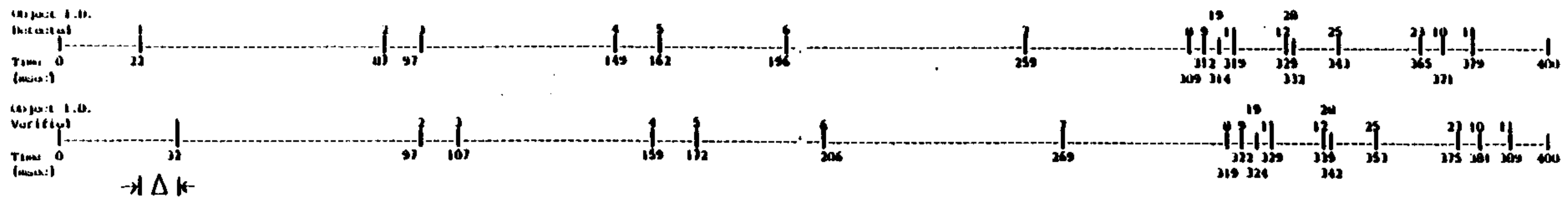
The radar-detection time for each object and, thus, the number of radar detections in 100 ms intervals (i.e., $N_{22}(d;t_k,t_{k+1})$) are computed from the input scenario (the sky map). For simplicity, only the first 400 ms of radar-detect and radar-verify operations are shown in Fig. 3-2(a). The algorithm for computing object detection times is given in Appendix C. The time-shift due to radar-verify delay ($\Delta \approx 10$ ms) is shown in the figure, i.e.,

$$N_{22}(v;t_k,t_{k+1}) \approx N_{22}(d;t_k-10,t_{k+1}-10)$$

Fig. 3-2(b) shows how $N_2(t_k,t_{k+1})$ was estimated. From eq. (3-2) we can obtain $V_{2,114}(W;t_k,t_{k+1})$ as follows

$$V_{2,114}(W;t_k,t_{k+1}) \approx N_2(v;t_k,t_{k+1}) \times 13(words)$$

Figs. 3-3(a) and 3-3(b) plot the estimates for $N_2(t_k,t_{k+1})$ and $V_{2,114}(W;t_k,t_{k+1})$ along with their measured values. Since the average execution time for M_3



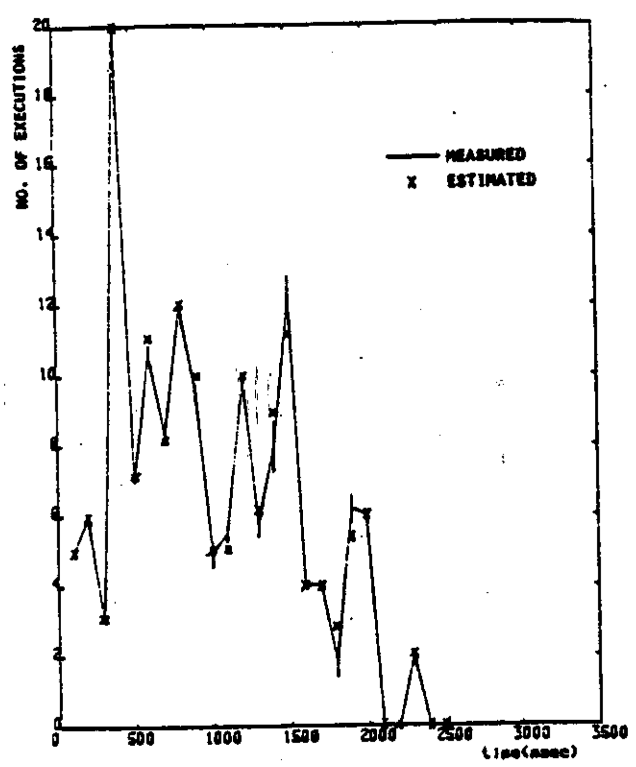
(a) Detection Time and Verification Time of Objects

Time t_h	0	100	200	300	400	500	600	700	800	900					1400					1900					2400					2900
Interval t_{h+1}	100	200	300	400	500	600	700	800	900	1000					1500					2000					2500					3000
$N_{22}(d)$	3	3	1	10	4	5	4	6	5	3	2	5	4	4	5	2	2	2	2	3	0	0	1	0	0	0	0	0	0	0
$N_{22}(v)$	2	3	2	10	3	6	4	6	5	2	3	5	2	5	6	2	2	1	3	3	0	0	1	0	0	0	0	0	0	0
$N_2(t_h, t_{h+1})$	5	6	3	20	7	11	8	12	10	5	5	10	6	9	11	4	4	3	5	6	0	0	2	0	0	0	0	0	0	0

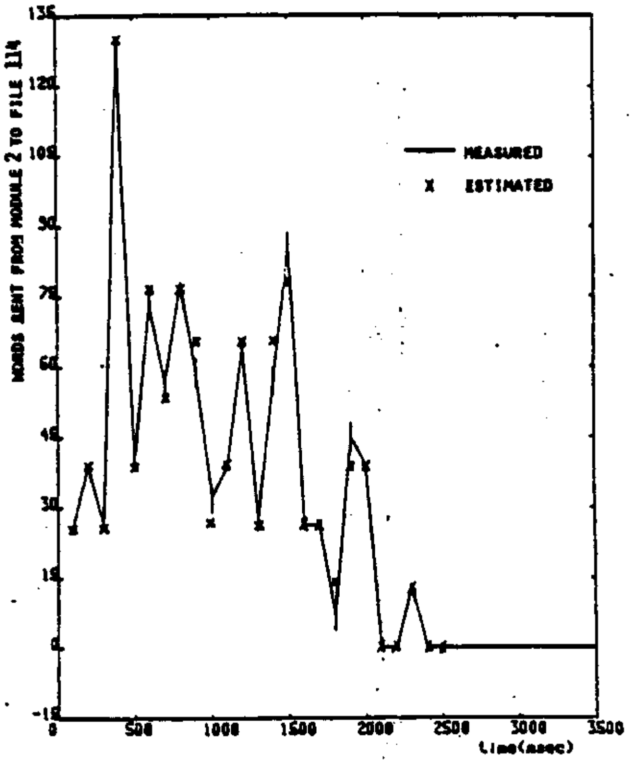
$N_{22}(d)$ = Number of object detections
= Number of M_{22} executions for detection processing
$N_{22}(v)$ = Number of object verifications
= Number of M_{22} executions for verification processing

(b) Number of Objects Detected and Verified During Each Interval

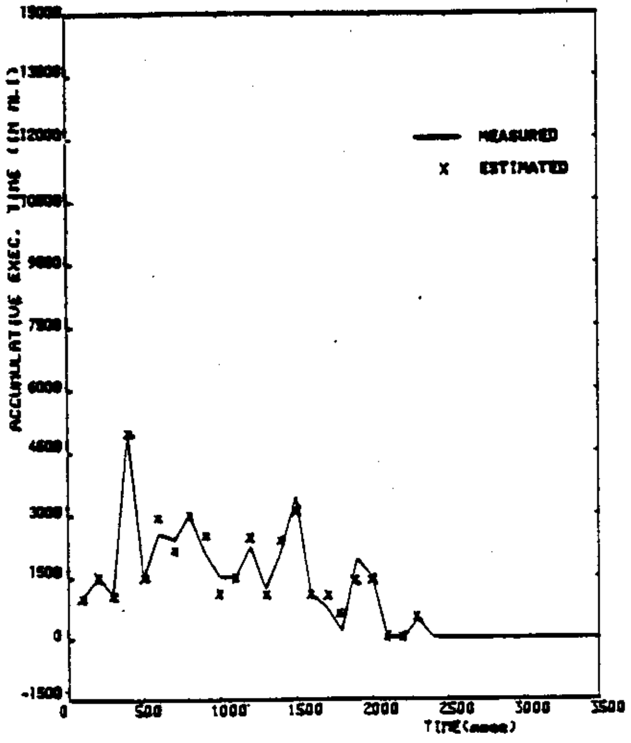
FIG. 3-2. EXECUTION FREQUENCY OF MODULE M_2 DURING EACH INTERVAL



A) No. of Module M_2 executions, $N_2(t_k, t_{k+1})$



B) M-F IMC from M_2 TO F_{114}, $V_{2,114}$,$(\mathrm{W}; t_k, t_{k+1})$



c) Module M_3 accumulative execution time, T_3 (t_k, t_{k+1})

FIG. 3-3. COMPARISON BETWEEN MEASURED AND ESTIMATED VALUES

(i.e., $x_3(t_k,t_{k+1})$) is 500 MLI and $N_3(t_k,t_{k+1})=N_3(v;t_k,t_{k+1})$, the AET for M_3 [i.e., $T_3(t_k,t_{k+1})$] is estimated from eq. (3-4). Both the measured and measured AET are shown in Fig. 3-3(c). All three charts in Figure 3-3 exhibit close correspondence between the estimated and measured values. This gives a validation for the estimation methodology.

3.4 CONTROL IMC

So far in this thesis only ***application IMC*** was studied. However, distributed processing systems require a significant number of control functions to support application file access (e.g., read-requests, lock-requests, and lock-releases) and to maintain the distributed processing system (e.g., status monitoring). These functions are carried out by ***control modules,*** resulting in ***control IMC.***

Control IMC consists of reads and updates (writes) by control modules (e.g., lock managers) to ***control files*** (e.g., lock status files). Once the file access and distributed system maintenance functions are known, control IMC can be ***derived*** from application IMC. For example, in Fig. 3-4, assume that module M_i at computer 1 updates application file F_k at computer 2 for $N_i(t_k,t_{k+1})$ times during (t_k,t_{k+1}). Since every F_k update requires a lock-request and a lock-grant, control module M_A at computer 1 writes $N_i(t_k,t_{k+1})$ lock-request messages to control file F_Y at computer 2 while M_B at computer 2 writes $N_i(t_k,t_{k+1})$ lock-grant messages to F_X at computer 1.

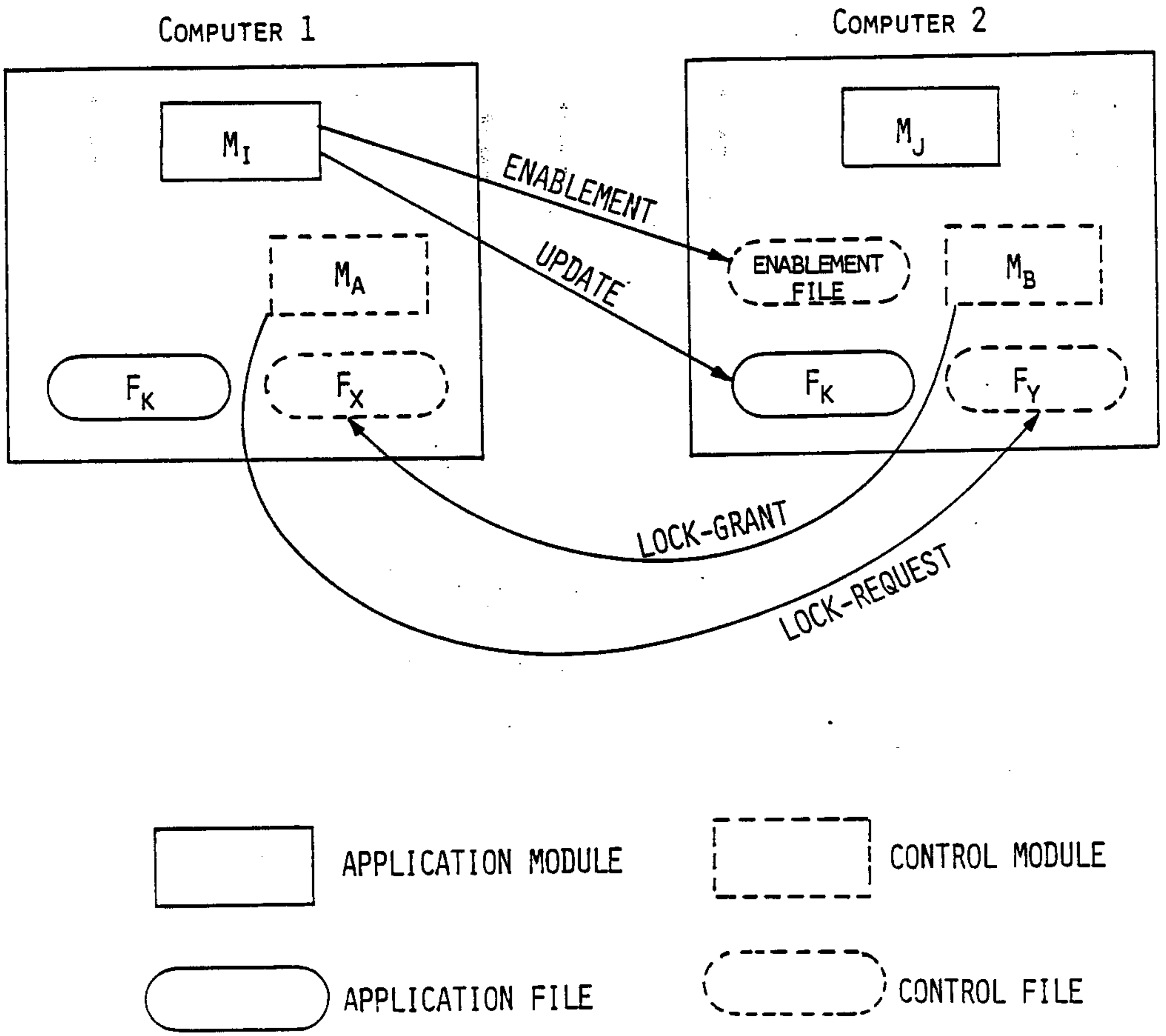


FIG.3-4. IPC IN A DISTRIBUTED PROCESSING SYSTEM

3.5 CALCULATION OF SYSTEM RESOURCE UTILIZATION

The utilization of a system resource (i.e., computer, shared memory, or bus channel) is defined as the fraction of time the resource is busy. Resource utilization is an important measure in distributed processing systems since it is directly related to queueing delays and throughput. Each IPC message incurs a processing cost at the sending computer *and* a processing cost at the receiving computer. It also increases network traffic. Therefore, IPC degrades the performance of distributed processing systems.

The utilization of a computer can be estimated from module AET, the processing cost for sending and receiving IPC messages at this computer site, and operating system overhead. The utilization of a channel can be derived from the channel bandwidth (in words/s) and the IPC volume to be sent on this channel.

CHAPTER 4

MODULE ASSIGNMENT

After discussion of IMC measurement and estimation, we shall now proceed to the study of module assignment problem. Section 4.1 identifies key parameters that should be considered in a module assignment. Section 4.2 describes the concepts of module assignment tree and enumeration procedure, and an objective function based on the concept of *minimum bottleneck* is proposed as a criterion for finding good module assignments. In Section 4.3 the objective function is applied to the DPAD system and is shown to yield good response time. Section 4.4 presents a heuristic algorithm based on that objective function that drastically reduces the computation time in finding good assignments from a huge problem space. This heuristic algorithm is applied to the DPAD (Section 4.5) and has yielded good response time.

4.1 KEY PARAMETERS IN MODULE ASSIGNMENT

The three parameters that play important roles in module assignment are intermodule communication (IMC), accumulative execution time (AET) of each module, and precedence relations (PR) among program modules. Chapter 2 shows that both the number of module executions and the AET are almost independent of module assignments if a *fixed* load is offered to the distributed system.

Our second parameter, IMC, is the communication between program modules through shared files. IMC can also be assumed to be independent of module assignment (see Chapter 2) and a method for estimating both IMC and AET has been proposed in Chapter 3. When a module on a computer writes to or reads from a shared file on *another* computer, such IMC becomes IPC (interprocessor communication) and causes processor overhead. The importance of IPC minimization has been recognized by many researchers [CHU78, GENT78, IRAN82, WU84, CHU84]. A task-allocation algorithm should reduce IPC by assigning a pair of heavily communicating modules to the same computer.

IPC varies with module assignments because the occurrence of IPC between two communicating modules depends on whether these two modules are assigned to different processors. For example, if two modules communicate through a *replicated* shared file and reside on different processors, then the file is replicated on each processor. When a processor updates the file, it updates the copy on its local processor. It then sends the updates to remote processors, resulting in IPC which requires processing load on both the sending and receiving processors. The IPC is eliminated if the two modules are assigned to the same processor since both modules are sharing the same local file copy.

The *processor load* for a computer is the sum of 1) load due to program module execution and 2) load due to IPC. Therefore, both AET and IPC play important roles in module assignment and thus influence task response time. These two components should be normalized and expressed in the same unit. Our approach is to *convert* the number of words transferred due to IPC *into* the number of machine-language instructions (MLI's) spent by the processor in transferring or receiving the IPC. A good module-assignment algorithm should at least follow these two rules:

R1) co-locate M_i and M_j to the same computer if the IMC volume between them is large. This eliminates the large IPC.

R2) Balance the loads on all computers.

For example, Figure 4-1 shows two module assignments for the DPAD System. Module assignment A only considers balancing the module processing load (due to AET) among computers. For example, both modules M_{13} and M_{14} have large accumulative execution time (see Figure 2-13), so they are assigned to different computers. Module assignment B considers both rules specified above. Since there is a large IMC between modules M_{13} and M_{14} (Figure 2-11), they were co-located. Similarly, M_6, M_8, M_{10}, and M_{16} were co-located. Using the UCLA DPAD Simulator, response times (referred to as port-to-port times) were generated for both module assignments. Fig. 4-2 portrays these measurement results (with 90-percent confidence intervals) for the DPAD Detect/Verify processing thread. Fig. 4-3 portrays the port-

ASSIGNMENT	A			B		
COMPUTERS	1	2	3	1	2	3
MODULE #	2	3	1	1	13	3
	5	4	7	2	14	5
	8	6	14	4	23	7
	9	10	16	6		9
	19	13	18	8		17
	20	17	21	10		18
		22	23	16		19
						20
						21
						22
AVERAGE MODULE LOAD	0.218	0.365	0.417	0.205	0.517	0.278

ASSIGNMENT A: BASED ONLY ON BALANCING MODULE LOAD
ASSIGNMENT B: BASED ON BOTH MODULE LOAD AND IMC

Fig. 4-1. TWO MODULE ASSIGNMENTS FOR THE DPAD SYSTEM

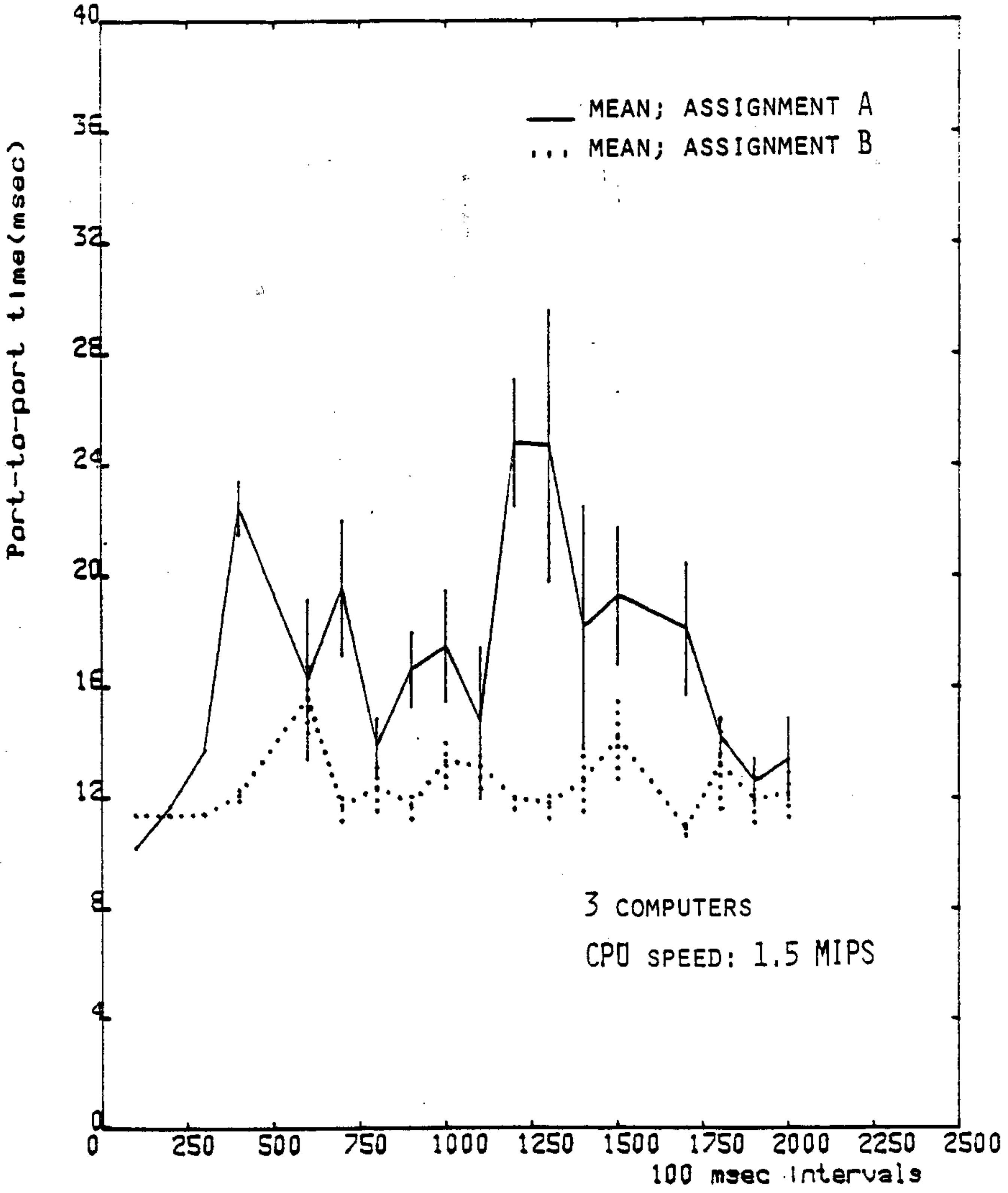


FIG. 4-2. PORT-TO-PORT TIME FOR DETECT/VERIFY THREAD

to-port time for the Precision Track thread. The overall superiority of assignment B clearly demonstrates that IMC plays an important role in module assignment.

4.2 A NEW OBJECTIVE FUNCTION

Let us first define the concept of *module assignment tree*. Consider a software system which has been partitioned into a fixed number of program modules. Given the control and data flow graph, the problem of module assignment is to *assign* the program modules into a number of processors such as to meet the performance requirements. The main design requirement in real-time systems is to meet the response time constraint, or port-to-port (P-T-P) time in BMD terminology. In this dissertation, we concentrate on distributed real-time systems where response time is the main design concern.

Since each module can be assigned to any of the S processors there are S^J different ways to assign J modules to S processors, assuming that each module is assigned to one and only one processor. This can be represented by an assignment tree with S^J leaves, each leaf corresponding to a possible assignment. This tree has J levels, each representing a module. At each non-leaf node there are S downward branches, each representing the choice of a processor to host the particular module. An example with $J = 23$ and $S = 3$ is shown in Fig. 4-4.

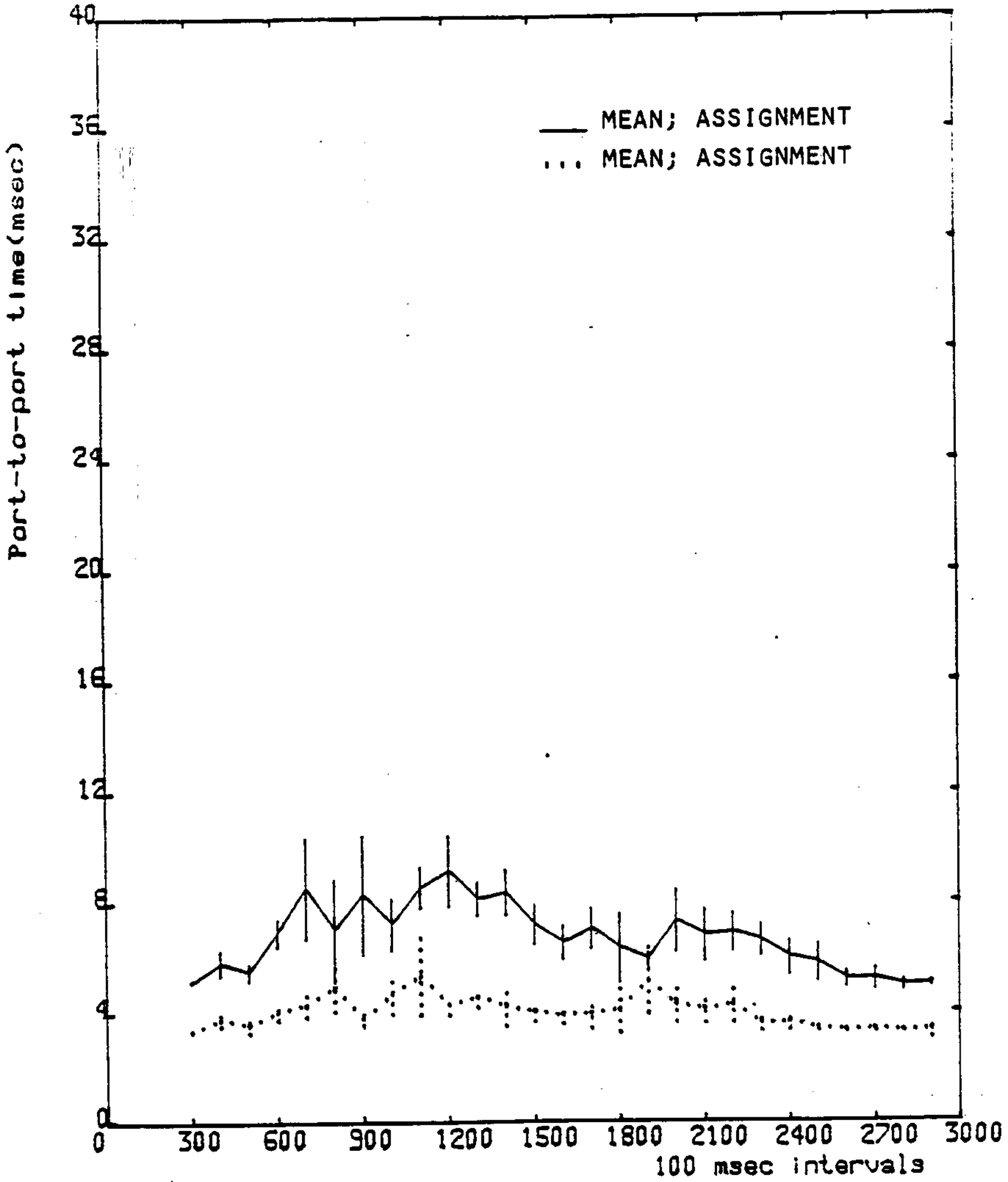
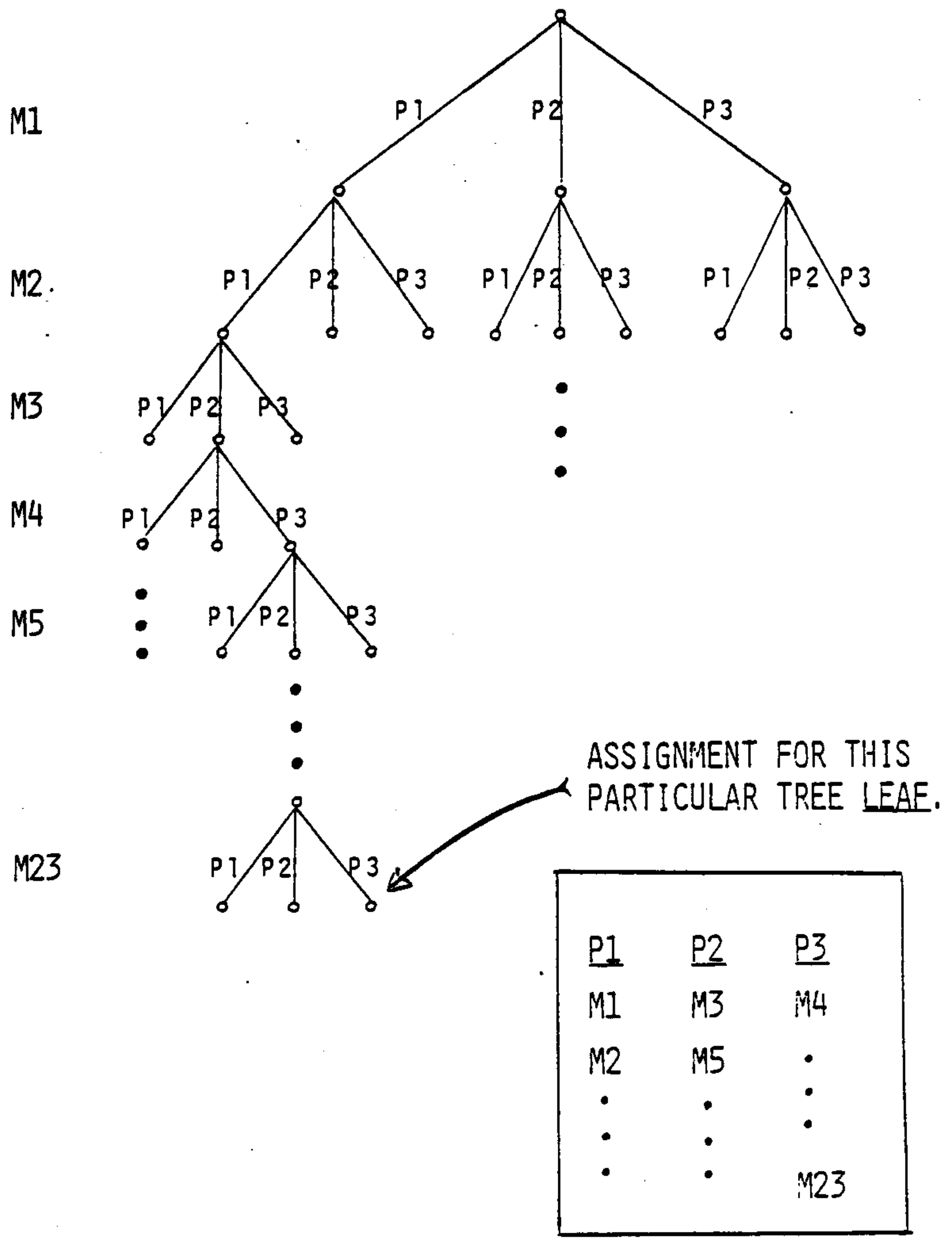


FIG. 4-3. PORT-TO-PORT TIME FOR PRECISION-TRACK THREAD

FIG. 4-4. AN ASSIGNMENT TREE



The module assignment problem is to search through the leaves of the assignment tree to find the particular assignment which yields the *minimum* value of an objective function (or maximum value in some cases, e.g., maximum system throughput.) An *exhaustive search* through all leaves is usually undesirable because of the enormous amount of time involved. For example, when the computation time for one leave requires 250 μs, the enumeration for a tree with 3^{20} leaves takes around 10 days.

For a real-time application, the key performance measure is response time. Normally a computer system is dedicated to a single application. The computer is required to finish a certain predefined processing within a time limit. Merely minimizing IPC alone may not produce a good assignment. In fact, a minimum-IPC assignment will assign all program modules to a single processor which might be saturated, resulting in poor response time. The processor with the heaviest loading in a distributed system is the one that causes the *bottleneck*. For instance, for a system with three processors, an assignment requiring 58%, 60%, and 61% of processor utilizations might have a better response time than a second assignment with 20%, 40% and 90% utilizations. This is mainly due to the fact that the second assignment has a *bottleneck processor* more heavily loaded than the first assignment, and queueing delay is a non-linear function that rises quickly with the level of bottleneck (processor load).

For a given assignment X, the work load $L(s;X)$ on a given Processor s is

$$L(s;X) = \sum_{j=1}^{J} x_{js} T_j + \sum_{\substack{t=1 \\ t\neq s}}^{S} \left[IPC(s,t;X) + IPC(t,s;X) \right] \tag{4-1}$$

The first term is the AET for all modules assigned to Processor s. The second term is IPC overhead which consists of two parts: overhead due to the IPC originated from Processor s to other processors, and incoming messages to Processor s from other processors. For a system whose file-update messages dominate the IPC traffic, we can ignore other types of IPC such as module enablement messages and system control messages. The total overhead due to outgoing IPC at Processor s is

$$\sum_{\substack{t=1 \\ t\neq s}}^{S} IPC(s,t;X) = w \sum_{j=1}^{J} x_{js} \sum_{k=1}^{K} V_{jk} \sum_{\substack{t=1 \\ t\neq s}}^{S} \delta_{kt} \tag{4-2}$$

where K is the number of files used in the distributed system; V_{jk} is the M-F IMC message volume sent from M_j to update the replicated file F_k at a remote Processor t; δ_{kt} indicates whether a replicated copy of F_k resides at Processor t; the term $\sum_{\substack{t=1 \\ t\neq s}}^{S} \delta_{kt}$ indicates the number of remote copies of F_k that must be updated; and w is a weighting constant for converting the message volume into MLI's. For a system with message-broadcasting capability, a file update need only be sent out once; the term $\sum_{\substack{t=1 \\ t\neq s}}^{S} \delta_{kt}$ in eq. (4-2) then reduces to 1.

Similarly, the total overhead at Processor s for incoming IPC from all remote sites is

$$\sum_{\substack{t=1 \\ t \neq s}}^{S} IPC(t,s;X) = w \sum_{\substack{t=1 \\ t \neq s}}^{S} \sum_{j=1}^{J} x_{jt} \sum_{k=1}^{K} V_{jk}\, \delta_{ks} \tag{4-3}$$

Based on the discussion above, we propose to use the work load (in unit of MLI) of the bottleneck processor as the objective function, i.e.,

$$Bottleneck(X) = \max_{s=1}^{S} \left\{ L(s;X) \right\} \tag{4-4}$$

The module assignment problem is to search through the assignment tree to find the assignment that yields the ***minimum bottleneck among all possible assignments*** [CHU84a], i.e.,

$$\min_{X} \left\{ Bottleneck(X) \right\} \tag{4-5}$$

or,

$$\min_{X} \left\{ \max_{s=1}^{S} \left[AET(s) + IPC(s) \right] \right\} \tag{4-6}$$

where AET(s) and IPC(s) are the total module execution time and total IPC overhead incurred at Processor s. Section 4.1 has shown that a good assignment can be obtained from minimizing IPC and balancing processor loads among the set of processors. A minimum-bottleneck assignment generally has low IPC and fairly balanced processor loads because:

1. If the loads were not fairly balanced for an assignment, the bottleneck (highest load among all processors) would be high and this assignment would not be a minimum-bottleneck assignment.

2. If a given assignment had high IPC, the sum of processor loads over all processors would be high and thus, yield high bottleneck.

Eq. (4-6) is different from minimizing the *sum* of processor loads [STON77],

$$\min_{X} \left\{ \sum_{s=1}^{S} \left[AET(s) + IPC(s) \right] \right\} \tag{4-7}$$

An assignment obtained by eq. (4-7) can be quite unbalanced. In a homogeneous system all modules will be assigned to a single processor as stated above. Our minimax principle is also used in [SHEN85] which considers only one-time execution of a task while we accumulate the processing load of *multiple* executions of the task. (Note that each external stimulus cause an execution of the task.)

In the objective function (see eq. (4-1)), AET for a module M_j is represented as a single value T_j. IMC information between a module and a file is also represented as a single value $V_{j,k}$ (eq. (4-2)). However, the measured or estimated value of AET, $T_j(t_k, t_{k+1})$, varies from one interval to another. Since we are concerned with system performance during the peak-load period, we shall use the average AET and IMC during the peak-load period for computation in our objective function.

4.3 PERFORMANCE OF THE PROPOSED OBJECTIVE FUNCTION

In this section we evaluate the performance of the proposed objective function by applying it to the DPAD system. Table 4-1 shows the average AET for the peak-load period for every module in the DPAD experiment where the identified peak-load period is from 1.0 sec to 2.0 sec of mission time. For example, T_8 (=32055 MLI's) for module M_8 is an average of ten measured values $T_8(1.0sec, 1.1sec)$, $T_8(1.1sec, 1.2sec)$, . . . , $T_8(1.9sec, 2.0sec)$. The IMC information was estimated in the same manner. The results are shown in column 3 of Table 4-2. Column 2 shows the file(s) updated by the write module. Column 4 lists all the modules which read the updated file. If a read module for a file and its associated write module are on different processors, both processors would have a copy of the file and IPC occurs for updating the replicated file copy.

A FORTRAN program was developed to compute the proposed objective function for *every* assignment in the DPAD assignment tree. When an assignment (corresponding to a tree leave) yields a bottleneck value smaller than the smallest bottleneck ever obtained so far, that assignment is saved and printed out (see Appendix D). The last ten such assignments were selected (Fig. 4-5). These ten assignments were then simulated with the

TABLE 4-1
ACCUMULATIVE EXECUTION TIME (AET) PER 100 MSEC

(UNIT: MLI)

Module	AET
M1	8865
M2	2700
M3	1590
M4	10410
M5	1860
M6	1950
M7	1680
M8	32055
M9	18600
M10	3360
M11	0
M12	0
M13	25305
M14	16860
M15	0
M16	4170
M17	6240
M18	3975
M19	9705
M20	2010
M21	195
M22	16410
M23	17025

MLI = Machine Language Instructions

EACH AET IS AN AVERAGE ACROSS THE PEAK-LOAD PERIOD, FROM 1.0 SECOND TO 2.0 SECONDS.

TABLE 4-2. FILE UPDATE IMC (in MLI) PER 100 MSEC

Write Module	File Updated	IMC Size	Read Modules
M1	none		
M2	F114	124	M3
M3	F115	144	M13
M4	F116	112	M13
	F117	314	M5, M6, M7
M5	F119	68	M7
M6	F121	68	M7
M7	F122	67	M13
M8	F120	62	M13
	F123	1568	M6, M10, M16, M9, M17, M19, M20
	F124	6387	M6, M10, M16
M9	F125	806	M13
M10	F127	1	M8
	F134	1	M18
(M11)	Module Not Implemented		
(M12)	Module Not Implemented		
M13	F131	30371	M14
M14	F147	1800	M13
	F132	5019	M23
(M15)	Module Not Implemented		
M16	F135	100	M18
M17	F136	100	M18
M18	F137	229	M19
	F138	36	M8
	F139	244	M20
M19	F139	599	M20
M20	F140	62	M21
M21	F141	32	Radar
M22	F142	242	M23
	F113	4593	M1, M2, M4, M8
M23	F112	5112	Radar
Radar	F111	14737	M22

* EACH TRANSFERRED WORD = 3 MLIs

EACH IMC SIZE IS AN AVERAGE ACROSS THE PEAK-LOAD PERIOD, FROM 1.0 SECOND TO 2.0 SECONDS.

DPAD simulator and their performance is compared. Figs. 4-6(a) and (b) show the CPU utilization for Assignments #1 and #2. We note the loads of the 3 processors are quite balanced during the peak-load period between 1.0 sec. and 2.0 sec. Assignments #3 through #10 exhibit similar load-balanced behavior. This coincides with our expectation that our objective function *(the minimum-bottleneck model),* eqs. (4-1) thru (4-6), provides good balanced processor loads. For comparison, Figs. 4-7(a) and (b) show the processor loads for an arbitrary assignment and a knowledge-guessed [1] manually generated assignment [HOLL82]. These two assignments are less load-balanced compared with Assignments #1 and #2, and have much higher bottleneck loads.

Fig. 4-8 shows the Precision-Tracking port-to-port time for Assignments #1 through #9 as well as the arbitrary assignment. The arbitrary assignment has a poor performance because two of its three processors are saturated as was previously shown in Fig. 4-7. Note that the performance difference between a good and a bad assignment can be substantial. Poor assignments yield poor response time.

Fig. 4-9 compares the port-to-port time for the Detect/Verify thread between the knowledge-guessed assignment and Assignments #1 through #9.

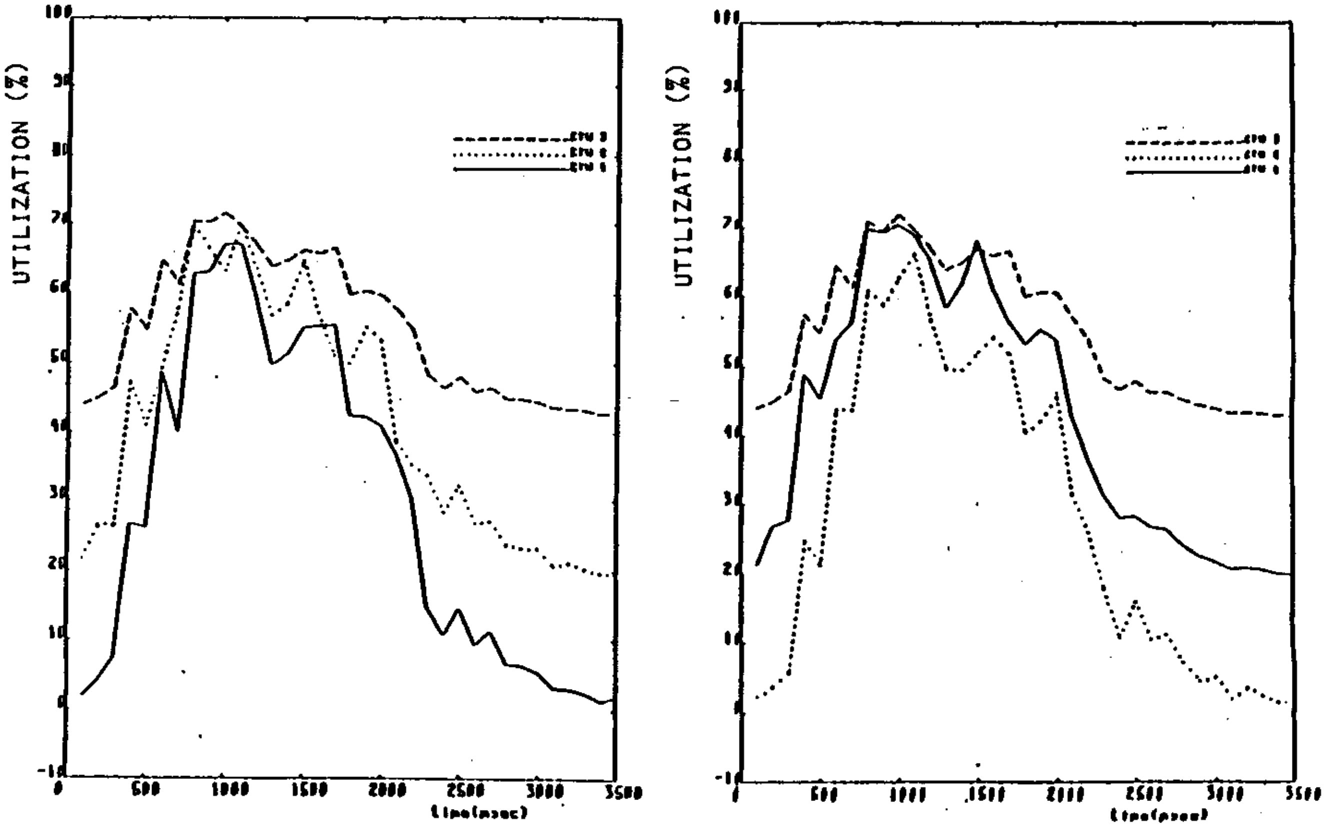
[1] The knowledge-guessed assignment was obtained by a combination of intuitive insight and trial and error. It was one of the best assignments known to the author in terms of port-to-port time for the DPAD example system.

FIG. 4-5. ENUMERATION RESULTS: 10 GOOD ASSIGNMENTS

	ASSIGNMENT	LOAD-1	LOAD-2	LOAD-3	BOTTLENECK	TOTAL LOAD
10th	11111 11121 00330 12322 123	75612	75546	70420	75612	221578
9th	11111 11121 00330 12322 323	75323	75546	70709	75546	221578
8th	11112 11121 00330 13222 223	75413	75352	73643	75413	224408
7th	11112 12131 00220 13231 132	75178	74275	73829	75178	223282
6th	11112 12131 00220 13231 232	75013	74564	73829	75013	223406
5th	11112 32333 00220 33211 112	74414	74275	74023	74414	222712
4th	11112 32333 00220 33211 312	74249	74275	74312	74312	222836
3rd	12123 23212 00330 21322 113	74308	73873	74275	74308	222456
2nd	12123 23212 00330 21322 213	74019	74038	74275	74275	222332
MINIM. BOTTLE-NECK	12213 13121 00330 11322 223	74004	73805	74275	74275	222084

NOTE: 1. LOAD-i IS EACH PROCESSOR'S LOAD PER 100 MSEC (IN UNIT OF MLI).

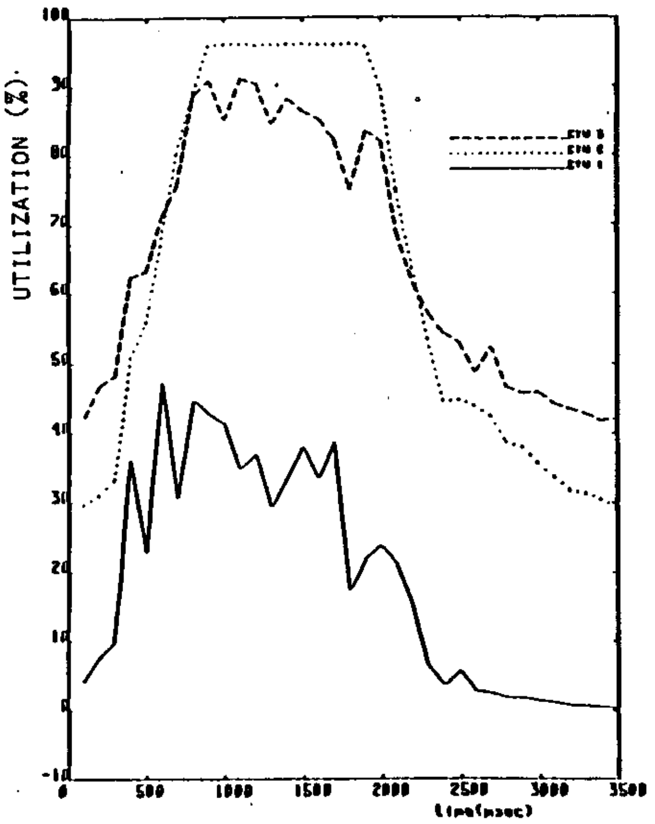
2. AN ASSIGNMENT WITH THE MINIMUM TOTAL LOAD IS NOT THE ASSIGNMENT WITH THE MINIMUM BOTTLENECK.



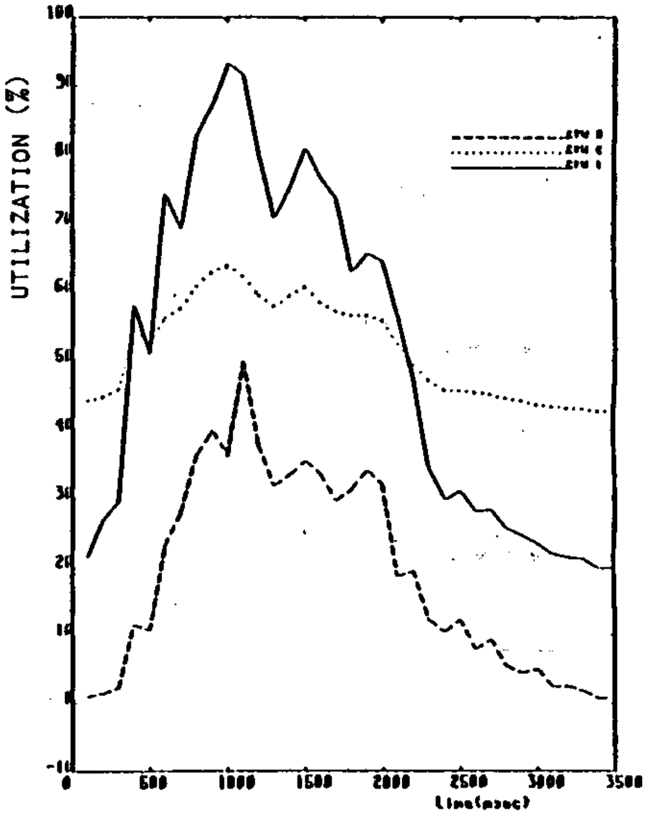
(A) FIRST SELECTED ASSIGNMENT

(B) SECOND SELECTED ASSIGNMENT

FIG. 4-6. CPU UTILIZATION IN THREE PROCESSORS FOR MODULE ASSIGNMENTS SELECTED BY EXHAUSTIVE SEARCH



(A) AN ARBITRARY ASSIGNMENT



(B) A KNOWLEDGE-GUESSED ASSIGNMENT

FIG. 4-7. CPU UTILIZATION IN THREE PROCESSORS FOR TWO OTHER ASSIGNMENTS

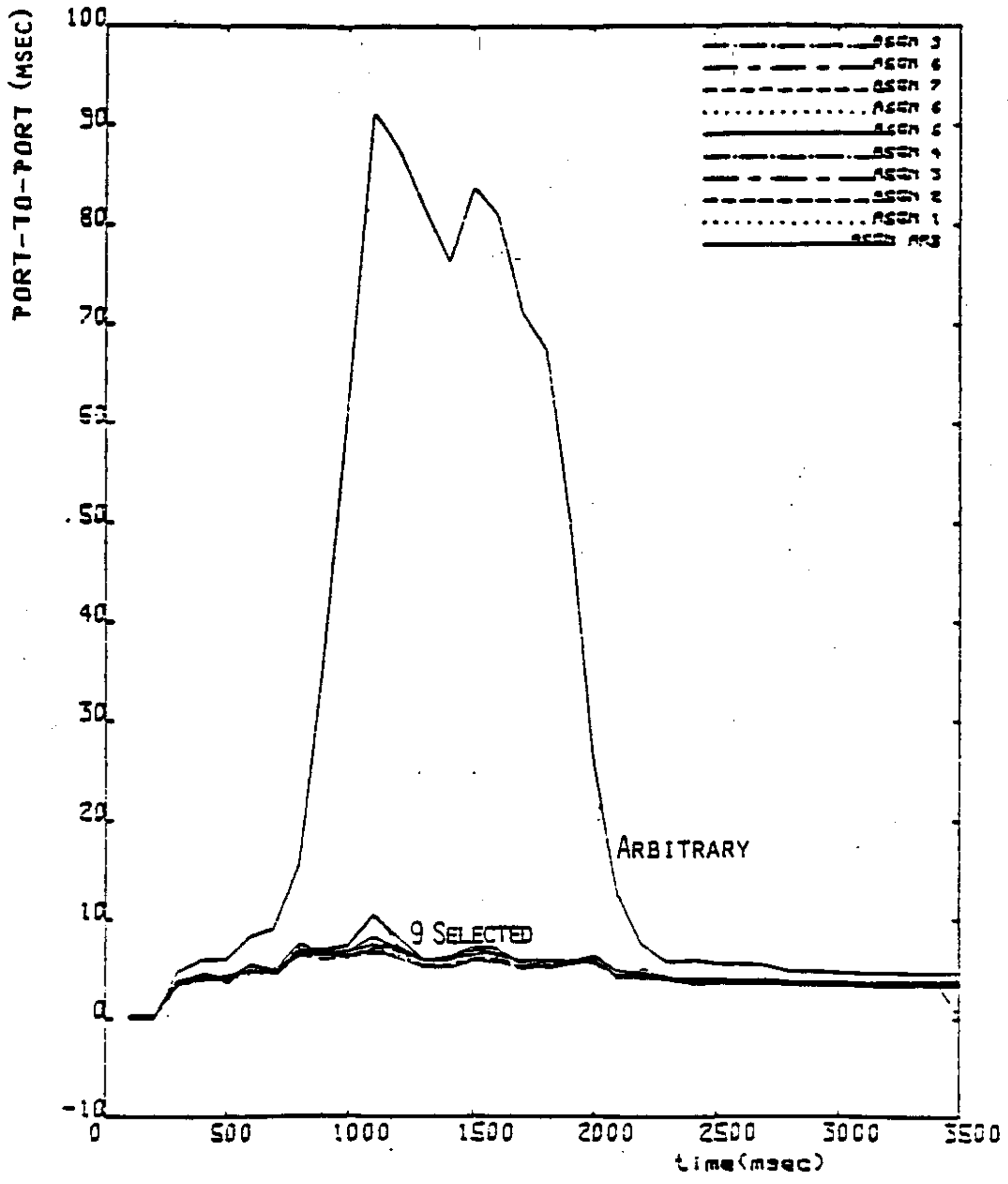


FIG.4-8. PTP TIME FOR THE PRECISION-TRACKING THREAD ---COMPARE AN ARBITRARY ASSIGNMENT AND 9 ASSIGNMENTS SELECTED BY EXHAUSTIVE SEARCH

Figs. 4-10 and 4-11 are for the Track Initiation thread and the Precision Tracking thread. Our experiments confirm that the proposed objective function is able to generate fairly good module assignments.

4.4 HEURISTIC MODULE ASSIGNMENT WITHOUT PRECEDENCE RELATION

An exhaustive search through an entire assignment tree is prohibitively time-consuming. In order to drastically reduce the computation time, we shall develop a heuristic algorithm for selecting good assignments from a huge problem space.

In the following we propose a two-phase heuristic algorithm for module assignment. Let us denote it as Algorithm I-A, for the initial characters of "IMC" and "AET". In order to avoid heavy IPC, Phase I merges heavily communicating modules into groups if the resulting group does not have too large an AET. This phase is a linear-time algorithm, requiring little computation time. Each group is a *set* of modules which will be assigned *as a single unit* to a processor in Phase II. Phases II assigns the module groups to the available processors such that the bottleneck (in the most heavily utilized processor) is minimized. Our algorithm assumes that

1. there are J modules, M_1, M_2, . . ., M_J, and S processors;

2. the average AET, T_i, (over the peak-load period) for each module M_i is given;

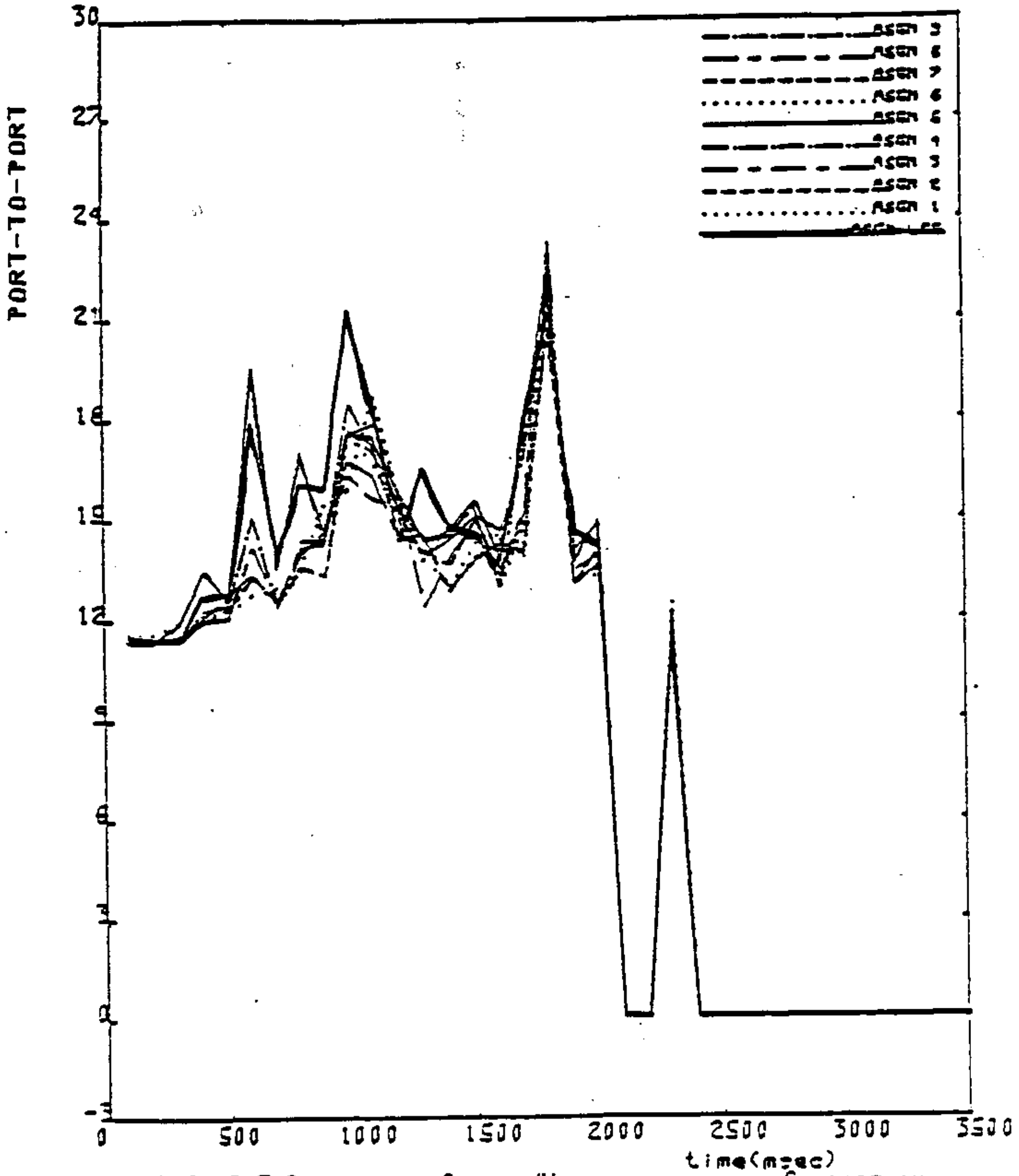


FIG. 4-9. P-T-P TIME FOR SEARCH/VERIFY THREAD -- COMPARE AN KNOWLEDGE-GUESSED ASSIGNMENT AND 9 ASSIGNMENTS SELECTED BY EXHAUSTIVE SEARCH

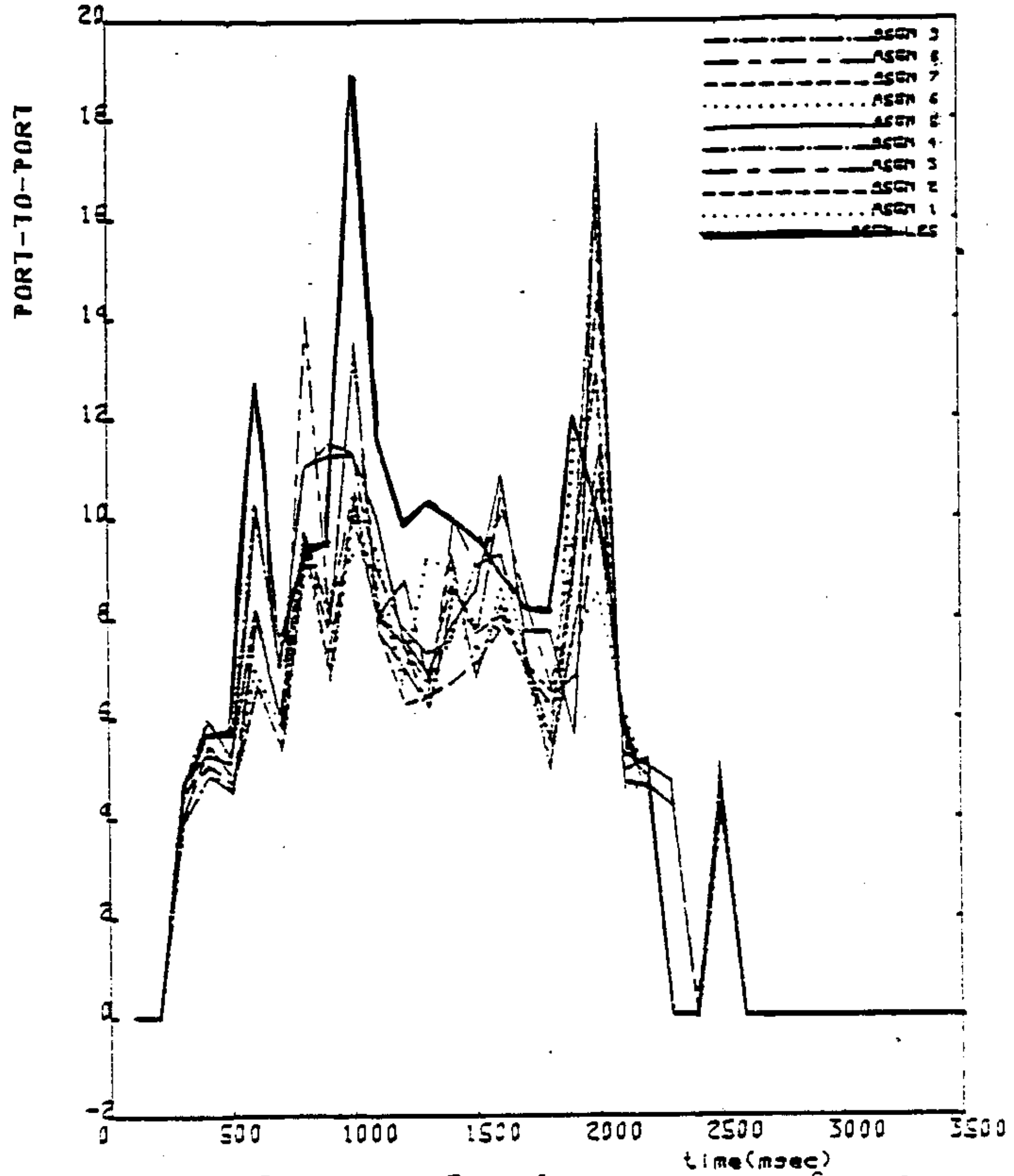


FIG. 4-10. P-T-P TIME FOR TRACK-INITIATE THREAD -- COMPARE A KNOWLEDGE-GUESSED ASSIGNMENT AND 9 ASSIGNMENTS SELECTED BY EXHAUSTIVE SEARCH

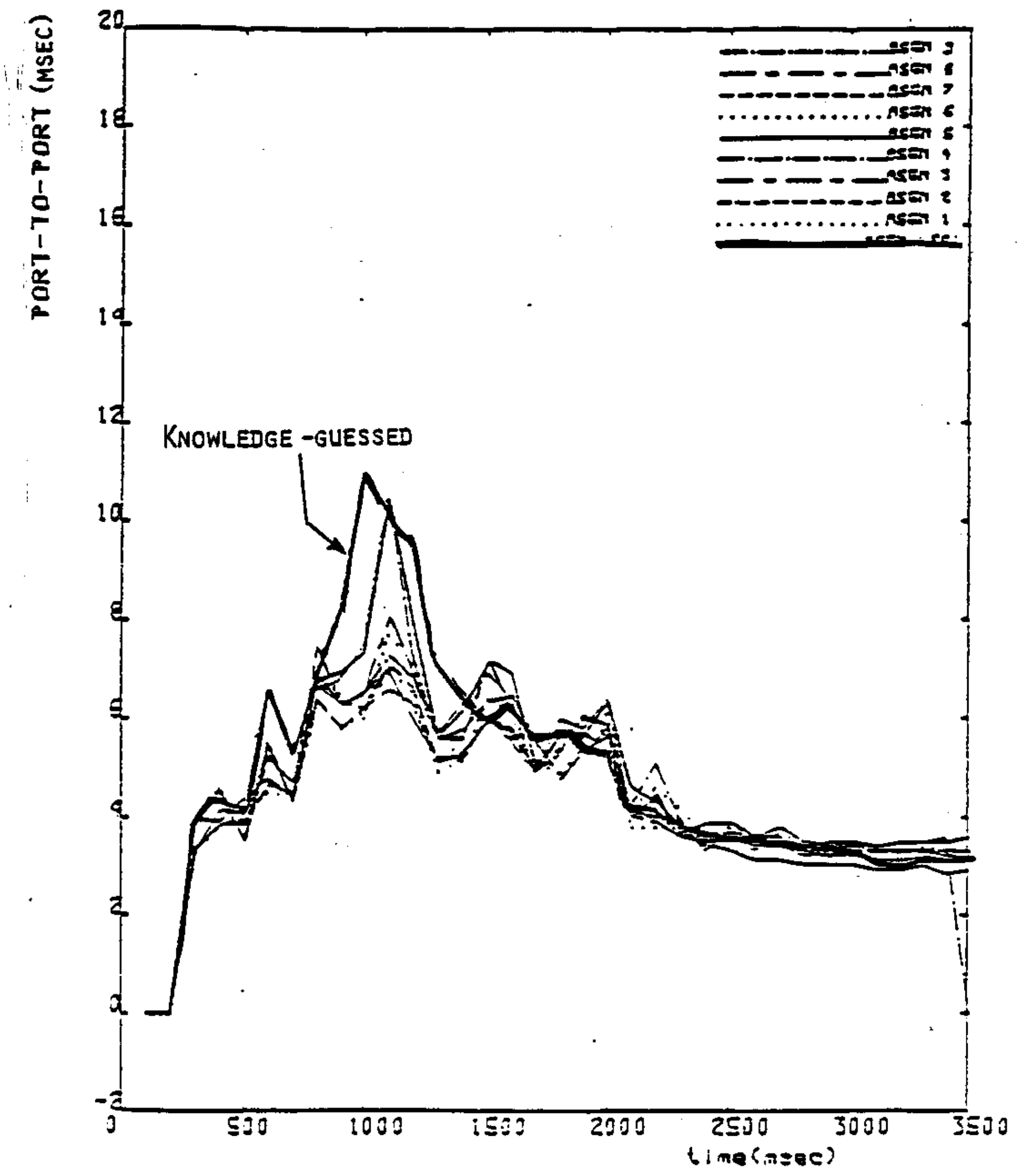


FIG. 4-11. PTP TIME FOR THE PRECISION-TRACKING THREAD --COMPARE THE KNOWLEDGE-GUESSED ASSIGNMENT AND 9 ASSIGNMENTS SELECTED BY EXHAUSTIVE SEARCH

3. the average IMC, $IMC_{i,j}$, (over the peak-load period) between any module pair M_i and M_j is given;

ALGORITHM I-A:

Phase I: Merge modules with large IMC into groups to reduce total system load.

```
1.1 Initially list all module pairs ($M_i$, $M_j$) in the descending order of IMC
    volume.

    Calculate average module size & average processor load:
        $\overline{AET} \leftarrow \sum_{i=1}^{J} T_i / J$
        $\overline{PL} \leftarrow \sum_{i=1}^{J} T_i / S$

    Set threshold values for IMC values & for processor loads:
        $\theta_{IMC} \leftarrow \overline{AET} \times \alpha \%$
        $\theta_{PL} \leftarrow \overline{PL} \times \beta \%$

    Let each program module form a distinct group (a set):
        $G_i \leftarrow \{M_i\} \quad i = 1,\ldots,J$

1.2 If no more pairs exist in the module-pair list
        go to Phase II.
    Pick the next pair of modules, $M_a$ and $M_b$, and delete this pair from
    the list.

1.3 If $IMC_{a,b} < \theta_{IMC}$
        go to Phase II.

1.4 Find the group $G_s$ that contains $M_a$, and the group $G_t$ that contains
    $M_b$ (i.e., $M_a \in G_s$ , $M_b \in G_t$).
    If $s = t$ (i.e., if $M_a$ and $M_b$ are already in the same group)
        go to Step 1.2.

1.5 If $T_s + T_t > \theta_{PL}$
```

```
        go to Step 1.2.

    1.6 Merge the two groups $G_s$ and $G_t$ into a single one:
            $G_s \leftarrow G_s \cup G_t$
            $G_t \leftarrow \emptyset$
            $T_s \leftarrow T_s + T_t$
            $T_t \leftarrow 0$

    1.7 Go to Step 1.2.

Phase II: Assign merged groups to processors to minimize the bottleneck.

    2.1 (We now have $q$ groups, $q < J$, which corresponds to a much smaller
        assignment tree with $S^q$ possible assignments.)
        Perform an exhaustive search through the new assignment tree to
        locate good assignments.

    2.2 Stop.
```

In the following we provide some discussions on Steps 1.3 and 1.5.

Step 1.3: When we reach a pair of modules whose IMC is smaller than the ***IMC threshold*** $\boldsymbol{\theta}_{IMC}$ ($\boldsymbol{\alpha}$ % of $\overline{AET}$), merging them gives little benefit in terms of the IPC saved. The $\boldsymbol{\alpha}$ % should range from 1% to 10%.

Step 1.5: Our assignment algorithm tries to eliminate major IPC. When merging two group into one, we should leave some processing capacity in the resulting new group for accommodating the remaining IPC as well as some other small groups. If two groups were merged and formed a group that is too large, Phase II would not be able to produce a balanced-load assignment. This is the reason why ***processor-load threshold*** $\boldsymbol{\theta}_{PL}$ is based on the parameter $\boldsymbol{\beta}$ % and average processor load $\overline{PL}$.

4.5 APPLICATION OF ALGORITHM I-A TO THE DPAD SYSTEM

Let us now apply the heuristic Algorithm I-A to the DPAD module assignment problem. Table 4-2 shows IMC between a module and the files it accesses. For Phase I of Algorithm I-A, this table is reorganized into Table 4-3 which provides IMC size between module pairs. (Phase II uses Table 4-2). Fig. 4-12 shows the merging process of Phase I where 5% and 75% are used for the α and β respectively. Column 1 lists IMC in the descending order, column 2 displays the modules merged into one group, and column 3 calculates the total AET for all modules within the group.

Since in this example we have 3 processors and 20 modules, the $\overline{PL} = \sum_{i=1}^{23} T_i / 3 = 59555$ MLI and the $\overline{AET} = \sum_{i=1}^{23} T_i / 20 = 8933$ MLI. Phase I finishes when it reaches $IMC_{4,5} = 314$ MLI because 314 is smaller than 5% of $\overline{AET}$. The resultant groups are:

Group	Modules
1	1,2,4,22
2	3
3	5
4	6,8,10,16,20
5	7
6	9
7	13,14
8	17
9	18
10	19
11	21
12	23,(11,13,15)*

TABLE 4-3. FILE UPDATE IMC (in MLI) PER 100 MSEC FOR MODULE PAIRS

Write Module	Files Involved	IMC Size	Read Module
M2	F114	124	M3
M3	F115	144	M13
M4	F117	314	M5
M4	F117	314	M6
M4	F117	314	M7
M4	F116	112	M13
M5	F119	68	M7
M6	F121	68	M7
M7	F122	67	M13
M8	F123, F124	7955	M6
M8	F123	1568	M9
M8	F123, F124	7955	M10
M8	F120	62	M13
M8	F123, F124	7955	M16
M8	F123	1568	M17
M8	F123	1568	M19
M8	F123	1568	M20
M9	F125	806	M13
M10	F127	1	M8
M10	F134	1	M18
M13	F131	30371	M14
M14	F147	1800	M13
M14	F132	5019	M23
M16	F135	100	M18
M17	F136	100	M18
M18	F138	36	M8
M18	F137	229	M19
M18	F139	244	M20
M19	F139	599	M20
M20	F140	62	M21
M21	F141	32	Radar
M22	F113	4593	M1
M22	F113	4593	M2
M22	F113	4593	M4
M22	F113	4593	M8
M22	F142	242	M23
M23	F112	5112	Radar
Radar	F111	14737	M22

• EACH TRANSFERRED WORD = 3 MLIs

$IMC_{a,b}$(MLI)	Modules in the Merged Group	Exec. Time of the Group
IMC(13,14)=30371	13-14	25305+16860=42165
IMC(8, 6)= 7955	6-8	1950+32055=34005
IMC(8,10)= 7955	6-8-10	34005+3360=37365
IMC(8,16)= 7955	6-8-10-16	37365+4170=41535
IMC(14,23)= 5019	Can't group 13-14-23 Otherwise, 42165+10725=52890 > θ_{PL}	
IMC(22, 1)= 4593	1-22	8865+16410=25275
IMC(22, 2)= 4593	1-2-22	25275+2700=27975
IMC(22, 4)= 4593	1-2-4-22	27975+10410=38385
IMC(22, 8)= 4593	Can't group* 1-2-4-6-8-10-16-22 Otherwise, 38385+41535=79920 > θ_{PL}	
IMC(14,13)= 1800	Modules already in same group	
IMC(8, 9)= 1568	Can't group 6-8-9-10-16 Otherwise, 41535+18600=60135 > θ_{PL}	
IMC(8,17)= 1568	Can't group 6-8-10-16-17 Otherwise, 41535+6240=47775 > θ_{PL}	
IMC(8,19)= 1568	Can't group 6-8-10-16-19 Otherwise, 41535+9705=51240 > θ_{PL}	
IMC(8,20)= 1568	6-8-10-16-20	41535+2010=43545
IMC(9,13)= 806	Can't group 9-13-14 Otherwise, 42165+18600=60765 > θ_{PL}	
IMC(19,20)= 599	Can't group 6-8-10-16-19-20 Otherwise, 43545+9705=53250 > θ_{PL}	
IMC(4, 5)= 314		

(Phase I finishes because IMC(4,5)= 314 < θ_{IMC}

* Two large groups would otherwise be merged into one.

Fig. 4-12. Example of Phase-I Evaluation of the Algorithm

* Modules M_{11}, M_{12}, and M_{13} are not implemented in the DPAD; each has a zero AET, T_i.

We have merged 20 modules into 12 groups, a reduction from 3^{20} possible module assignments to 3^{12} possible group assignments which reduces the algorithm's computation time (CPU time) from 3 days down to less than one minute on a VAX-11/780.

To evaluate the effectiveness of our heuristic algorithm, the best assignment obtained by this algorithm is compared with that by the exhaustive search, as shown in Figs. 4-13 through 4-15. We note that the module assignment generated by the proposed heuristic algorithm provides comparable performance to that from the exhaustive search. We have also used our DPAD simulator to simulate the four assignments (Assignments A-1 through A-4) reported in [MA82]. Fig. 4-16 shows that our heuristic algorithm performs better than that of [MA82]. This is mainly because our algorithm provides better load-balancing than [MA82].

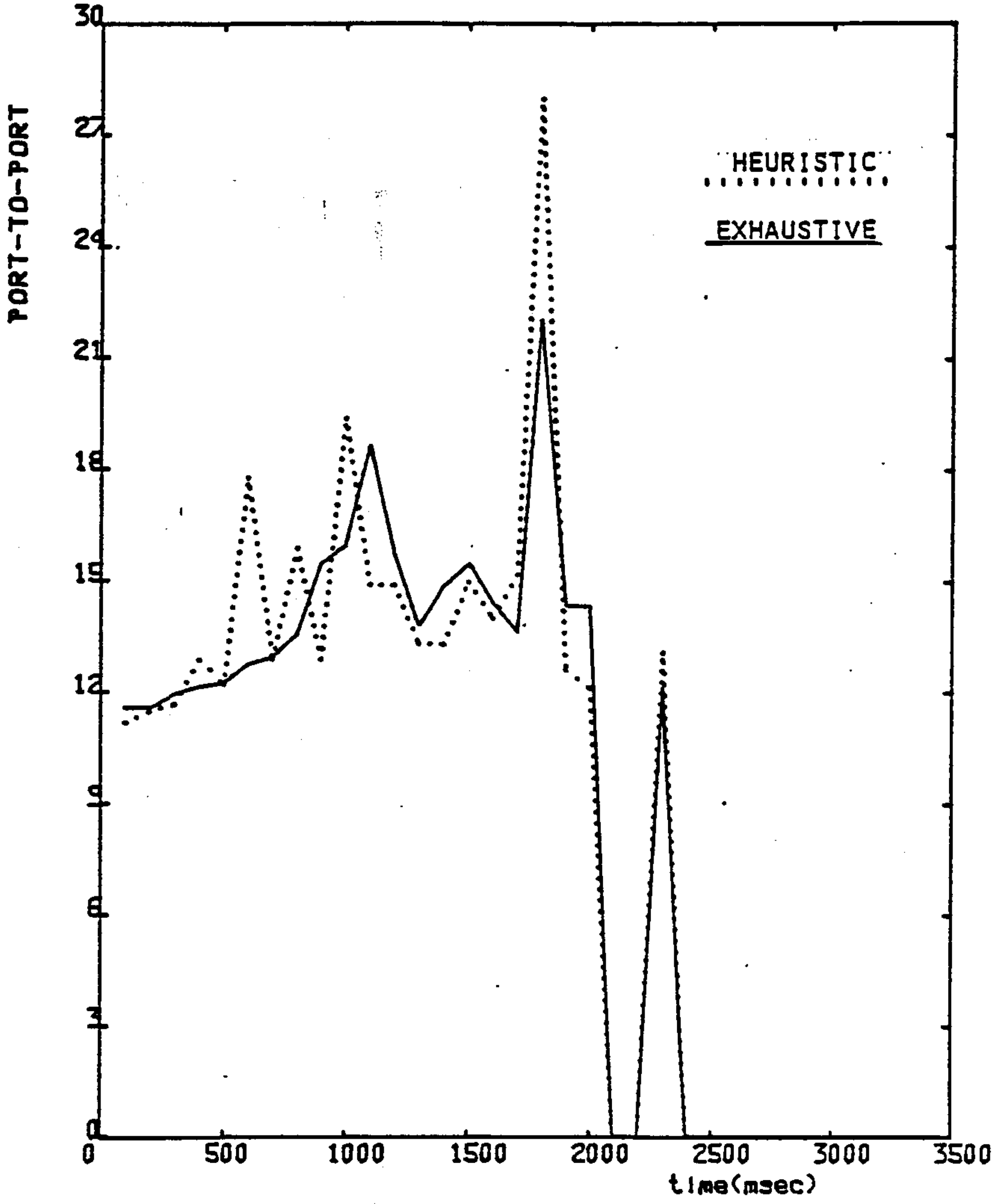


FIG. 4-13.
PTP TIME FOR DETECT/VERIFY THREAD--COMPARE BEST ASSIGNMENT SELECTED BY HEURISTIC ALGORITHM AND BEST ASSIGNMENT SELECTED BY EXHAUSTIVE SEARCH

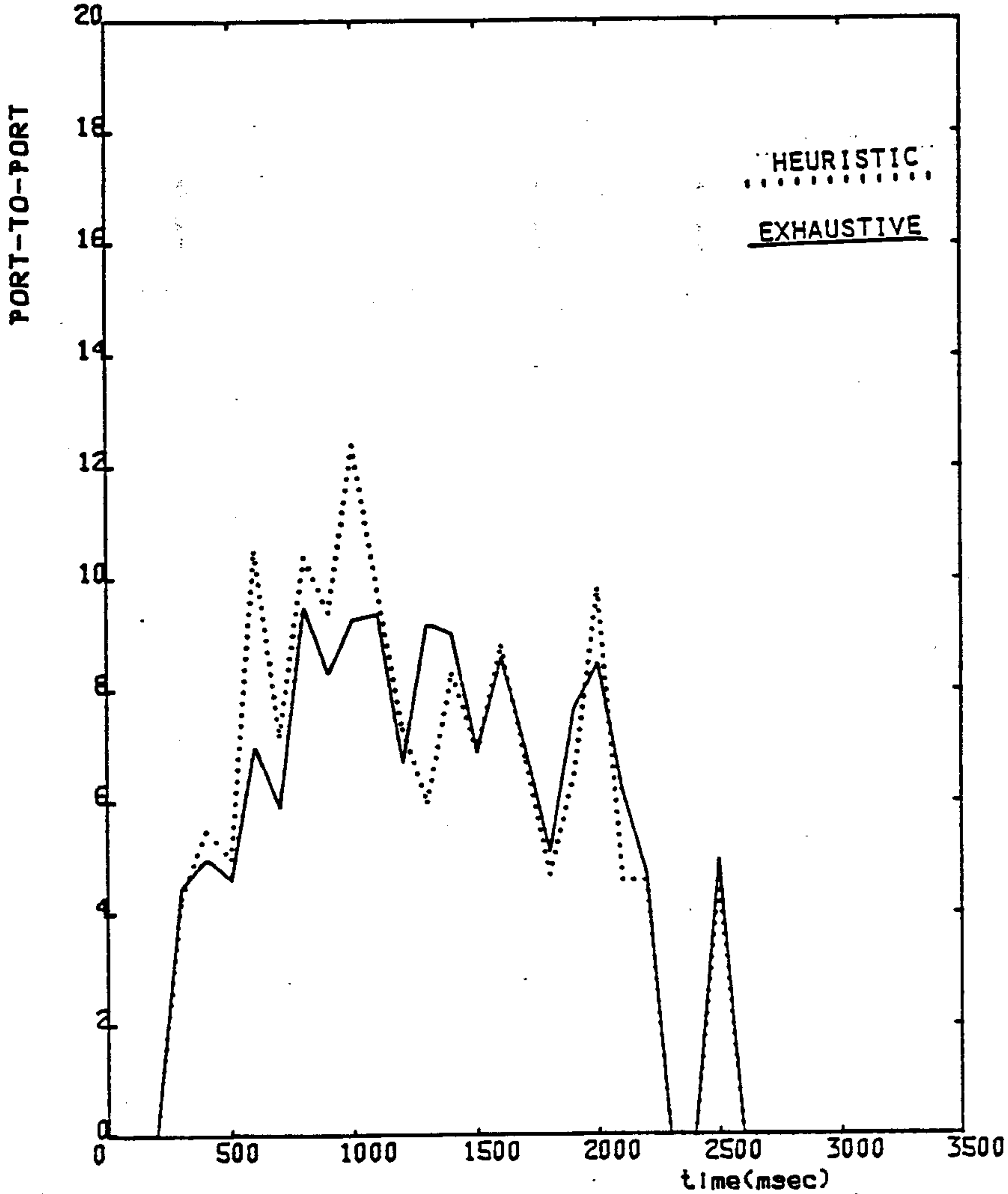


FIG. 4-14. PTP TIME FOR TRACK-INITIATE THREAD --COMPARE BEST ASSIGNMENT SELECTED BY HEURISTIC ALGORITHM AND BEST ASSIGNMENT BY EXHAUSTIVE SEARCH

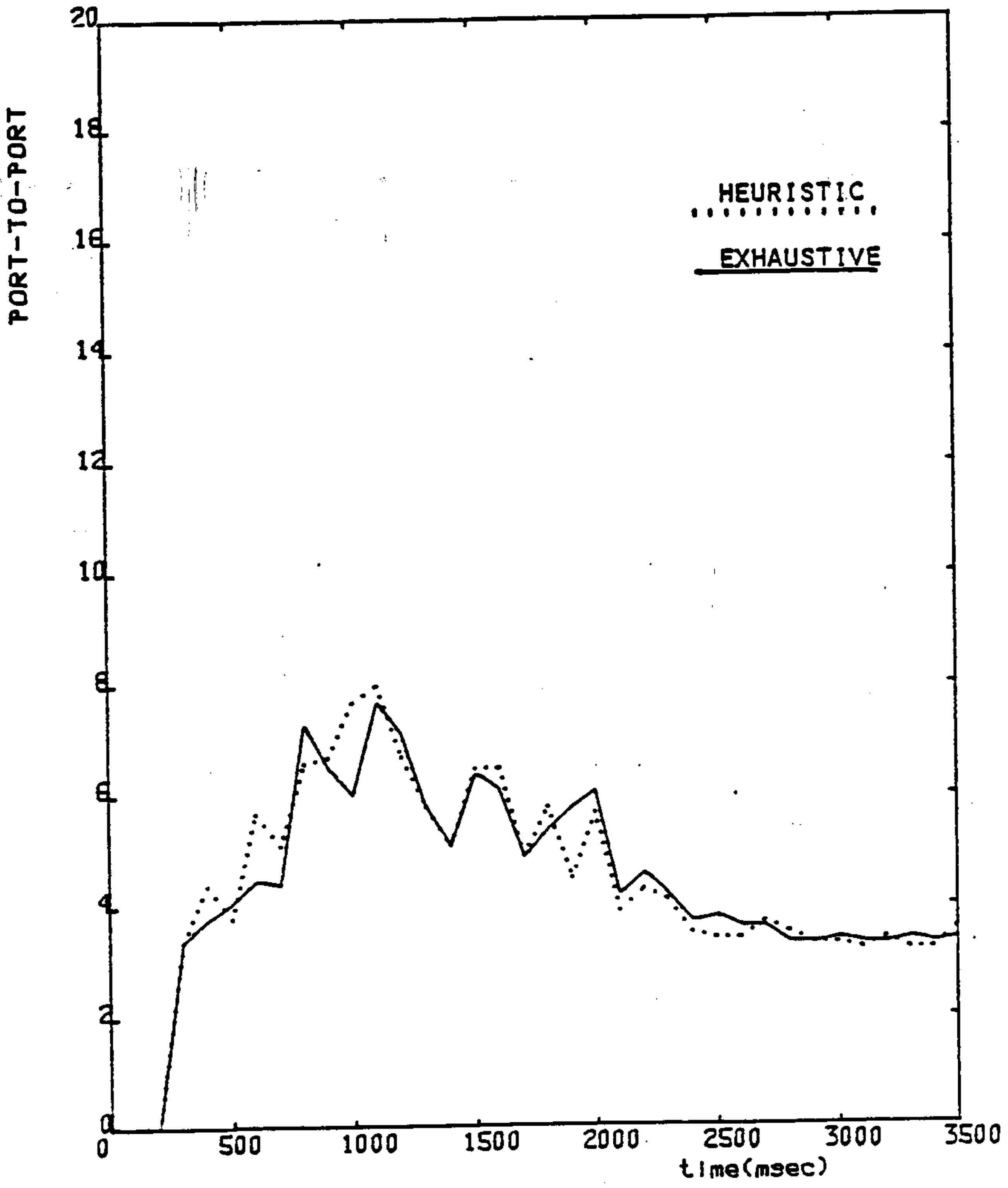


FIG. 4-15.
PTP TIME FOR PRECISION-TRACKING THREAD---COMPARE BEST ASSIGNMENT SELECTED BY HEURISTIC ALGORITHM AND BEST ASSIGNMENT SELECTED BY EXHAUSTIVE SEARCH

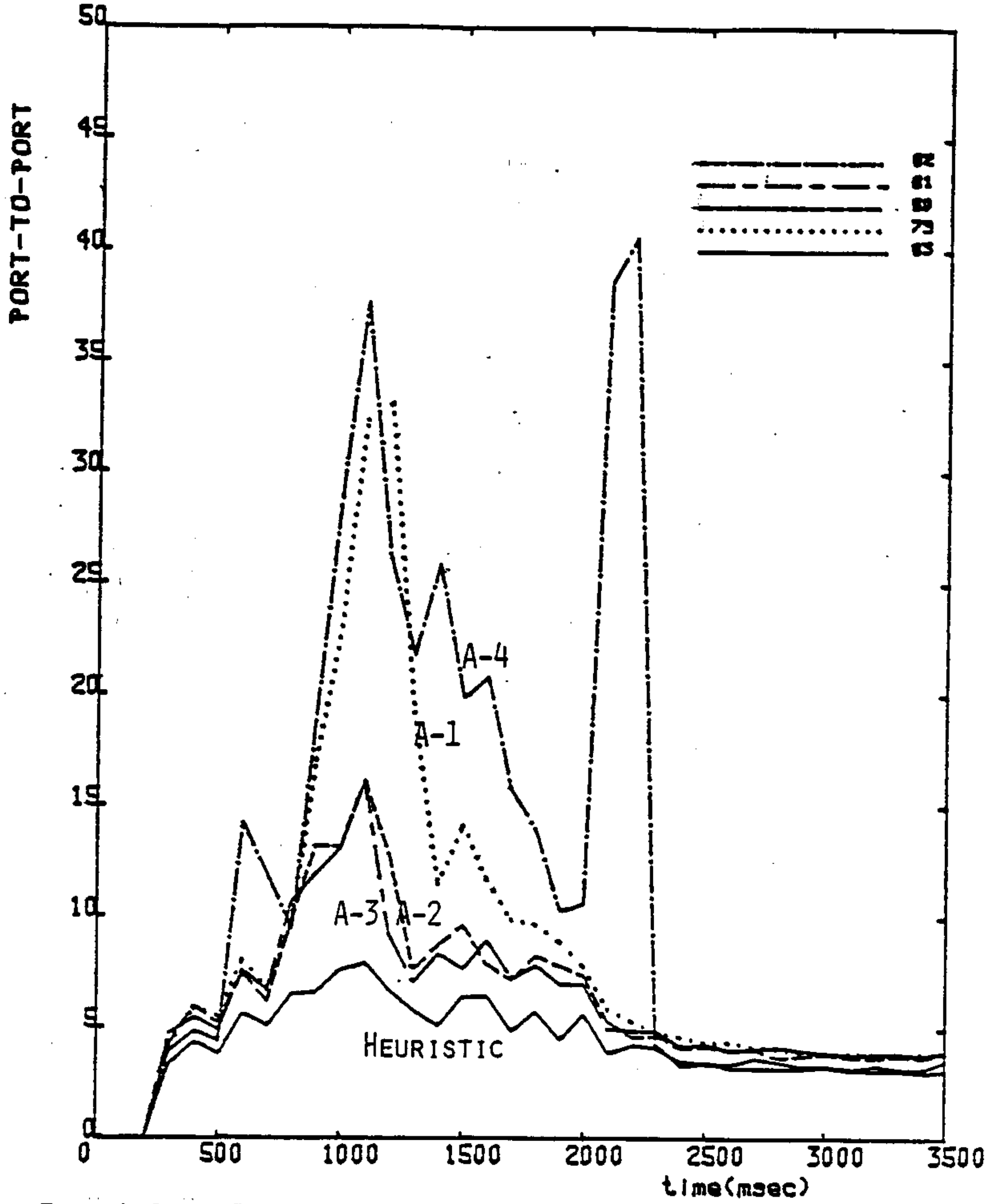


FIG. 4-16. PTP TIME FOR PRECISION-TRACK THREAD -- COMPARE BEST ASSIGNMENT SELECTED BY HEURISTIC ALGORITHM AND MA'S FOUR ASSIGNMENTS

CHAPTER 5

MODULE ASSIGNMENT WITH PRECEDENCE RELATIONSHIP

Algorithm I-A considers only IMC and AET. Another factor that needs to be considered is the precedence relation (PR) among program modules. In this Section 5.1 we describe several experiments on PR and study its impact on module assignment in terms of response time. Simple PR rules are obtained on whether two modules should be co-located on a single processor or separated on different processors. Section 5.2 includes these PR rules in the revised task-allocation algorithm (Algorithm P-I-A). In Section 5.3, Algorithm P-I-A is tested on our DPAD example system and shown to yield only slightly better response time than Algorithm I-A (which does not consider PR), because the size of most DPAD modules are in the same order of magnitude and therefore PR has little effect on the response time. When we apply Algorithm P-I-A to another example system (Section 5.4), significant improvement is obtained over Algorithm I-A.

5.1 PRECEDENCE RELATION EXPERIMENTS

For the first experiment, we compare three assignments of assigning nine modules to three computers (Fig. 5-1). The precedence exists in the control-flow graph from one module to another. Assume the job arrival is a

FIG. 5-1. EXPERIMENT NO. 1
THREE ASSIGNMENTS TO BE COMPARED

ASSIGNMENT #1 (SEQUENTIAL)

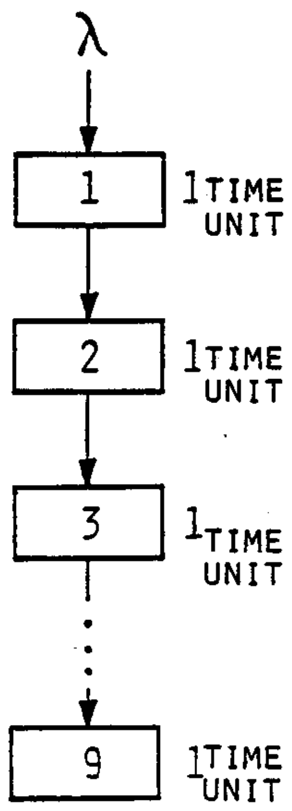
COMPUTER	1	2	3
	1	4	7
MODULES #	2	5	8
	3	6	9

ASSIGNMENT #2 (PIPELINED)

COMPUTER	1	2	3
	1	2	3
MODULES #	4	5	6
	7	8	9

ASSIGNMENT #3 (SKEWED)

COMPUTER	1	2	3
	1	2	3
MODULES #	5	6	4
	9	7	8



Poisson process with arrival rate λ, each job *enables* module M_1 which is placed in the *ready queue* of M_1's residence computer waiting to be processed, and upon the completion of execution a module will enable the execution of the next module in the control-flow graph. The execution time of every module is a *constant* (deterministic service time) and equal to one time unit. To simplify our analysis and to isolate the precedence effect, we further assume there is no IMC between modules and thus no IPC overhead between computers. The three assignments in Fig. 5-1 are simulated with PAWS simulator [BERR82]. The queueing discipline at all computers is FCFS. All three assignments result in balanced loads on the computers. However, simulation results (Fig. 5-2) reveal a significant difference in response time among the three assignments. The pipeline assignment (A1) yields the best response time. (Vertical bars in the figure represent 90%-confidence intervals for each simulation point). Since we assume no IPC overhead and all assignments are load-balanced, the response-time discrepancy is due solely to precedence relation.

In our second experiment, the execution time of each module is changed from a constant to *exponentially distributed* service time with a *mean* of one time unit. All other parameters remain the same as those used in Experiment No. 1. Simulation results (Fig. 5-3) exhibit that the response times for all three assignments are about the same. This is because the three computers form a Jackson network. Each computer can be treated

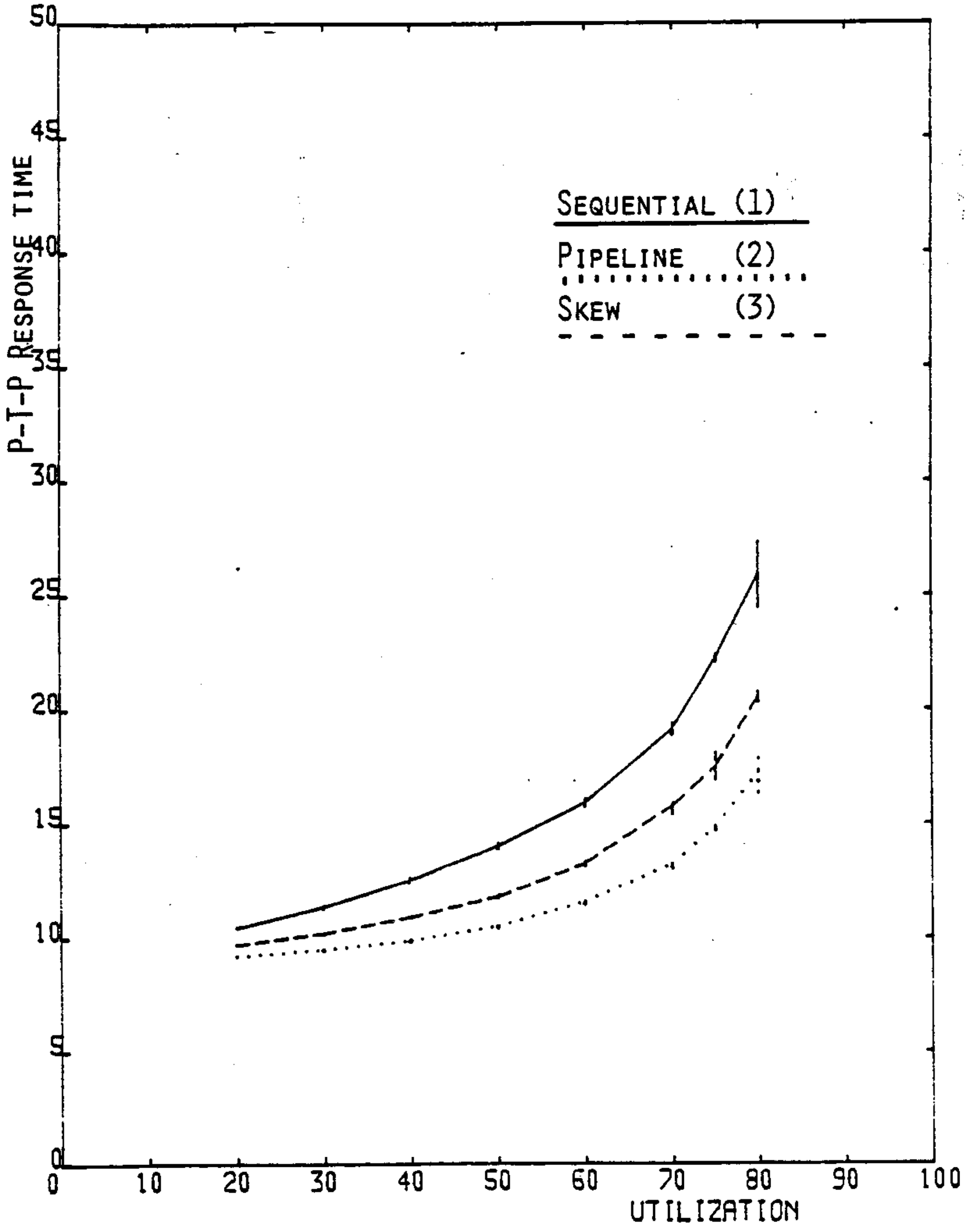


Fig. 5-2. Results of Experiment No.1
(Deterministic Execution time)

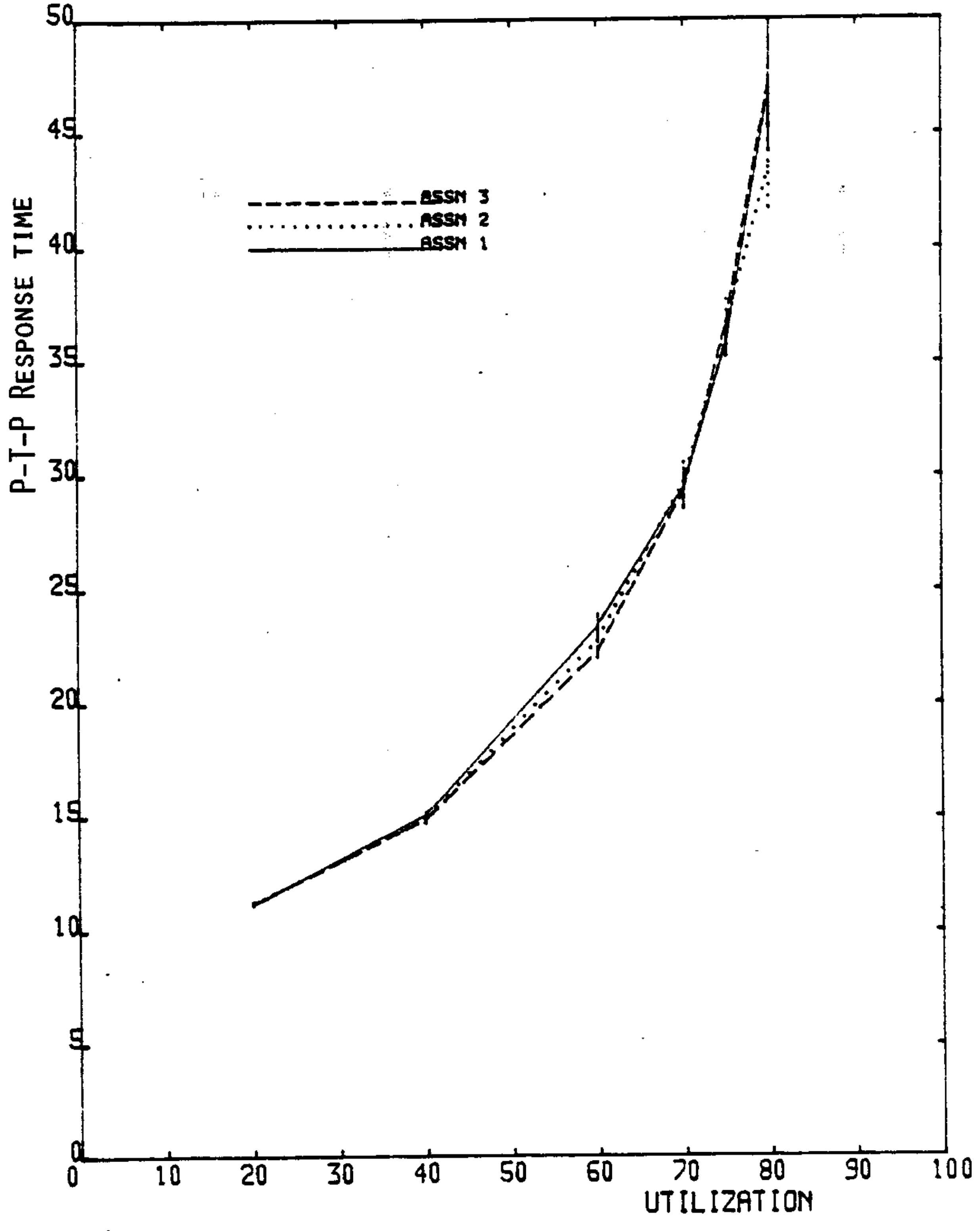
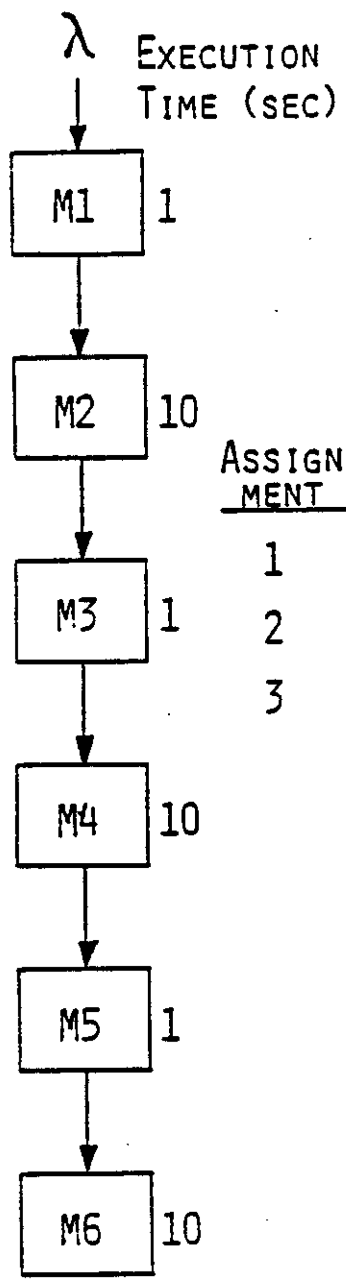


FIG. 5-3. RESULTS OF EXPERIMENT NO. 2
(EXPONENTIAL EXECUTION TIME)

individually as an M/M/1 queue for calculating the queueing wait time for each module, and since all modules have the same execution-time (service-time) distribution and the same arrival rate in this particular case, all load-balanced computers are treated as identical M/M/1 queues, and thus all modules have identical wait time. Experiment No.1 reveals that precedence relationship does have an impact on task response time. Experiment No.2 reveals that module execution-time distributions alter the PR's effect on the response time.

Experiment No.3 is for testing the effect of module size on precedence relationship. Modules have deterministic execution times as shown in Fig. 5-4(a). The three assignments in Fig. 5-4(b) are compared. The results (Fig. 5-5) reveal that assigning two consecutive modules to a same computer will yield a good response time *if the execution time of the second module is much larger than the first one.* We shall call this our PR rule #1. For example, in Assignment #1, M_1 and M_2 are assigned to the same computer. *If the second module is much smaller than the first one, it is better to separate two consecutive modules and assign them on two distinct computers* This is our PR rule #2. In Assignment #1, M_2 and M_3 are assigned on two different computers. Finally, the performance of Assignment #3 lies between Assignments #1 and #2 because Assignment #3 observes PR rule #2 for some module pairs (e.g., separation of M_2 from M_3) and violates PR rule #1 for some module pairs (e.g., separation of M_1 from M_2).

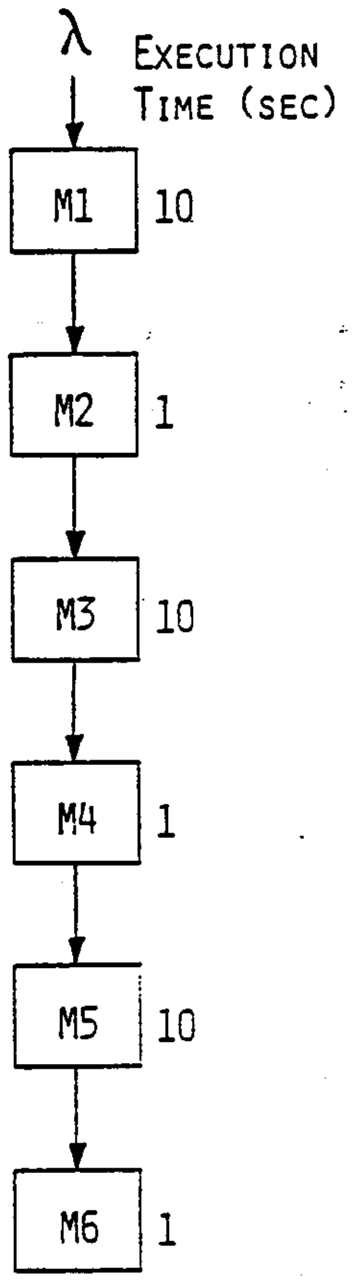


EXPERIMENT No. 3

(A)

ASSIGN-MENT	CPU 1	CPU 2	CPU 3
1	M1,M2	M3,M4	M5,M6
2	M1,M6	M2,M3	M4,M5
3	M1,M4	M2,M5	M3,M6

(B)

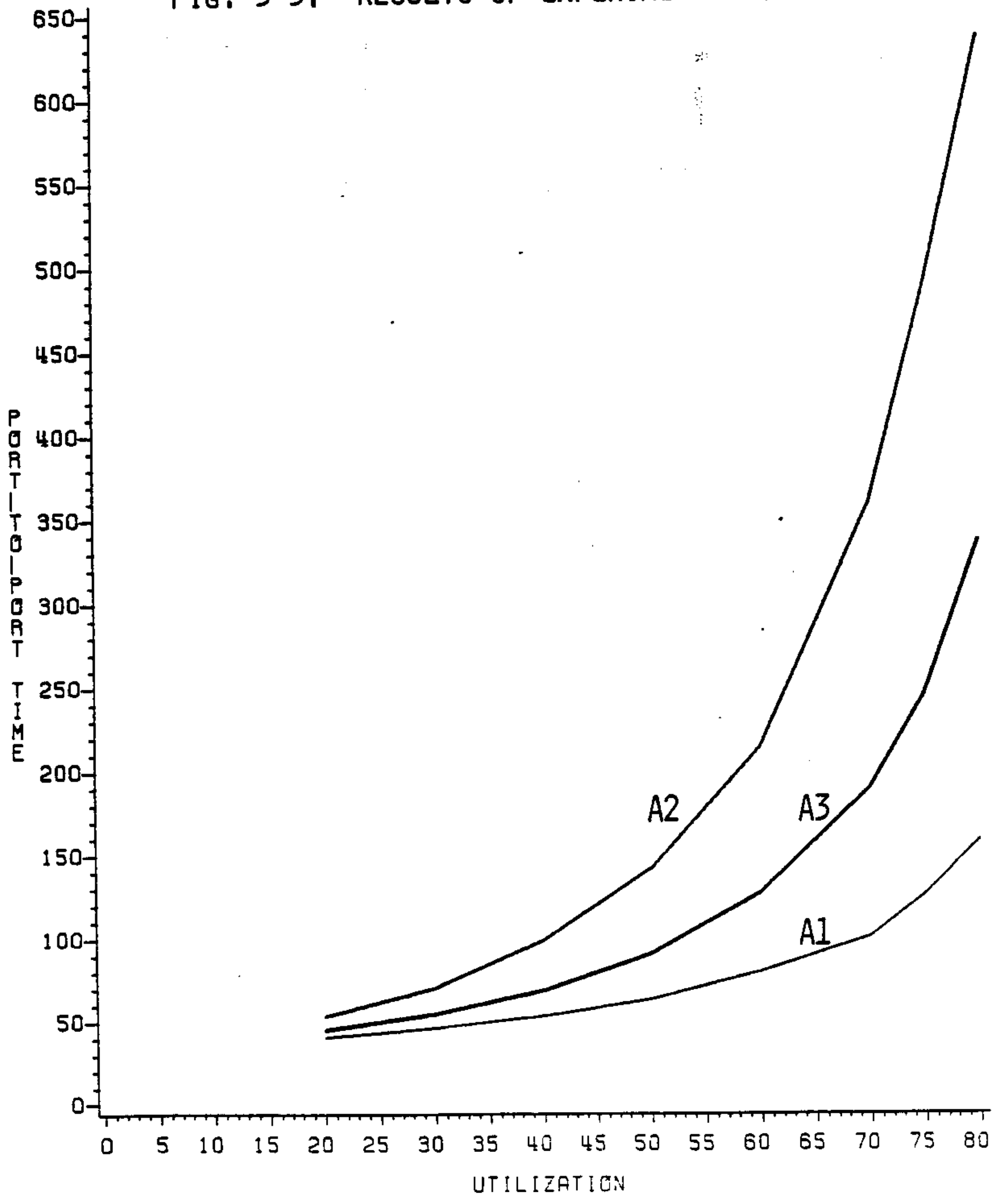


EXPERIMENT No. 4

(C)

FIG. 5-4. EXPERIMENTS No.3 & No.4

FIG. 5-5. RESULTS OF EXPERIMENT NO. 3

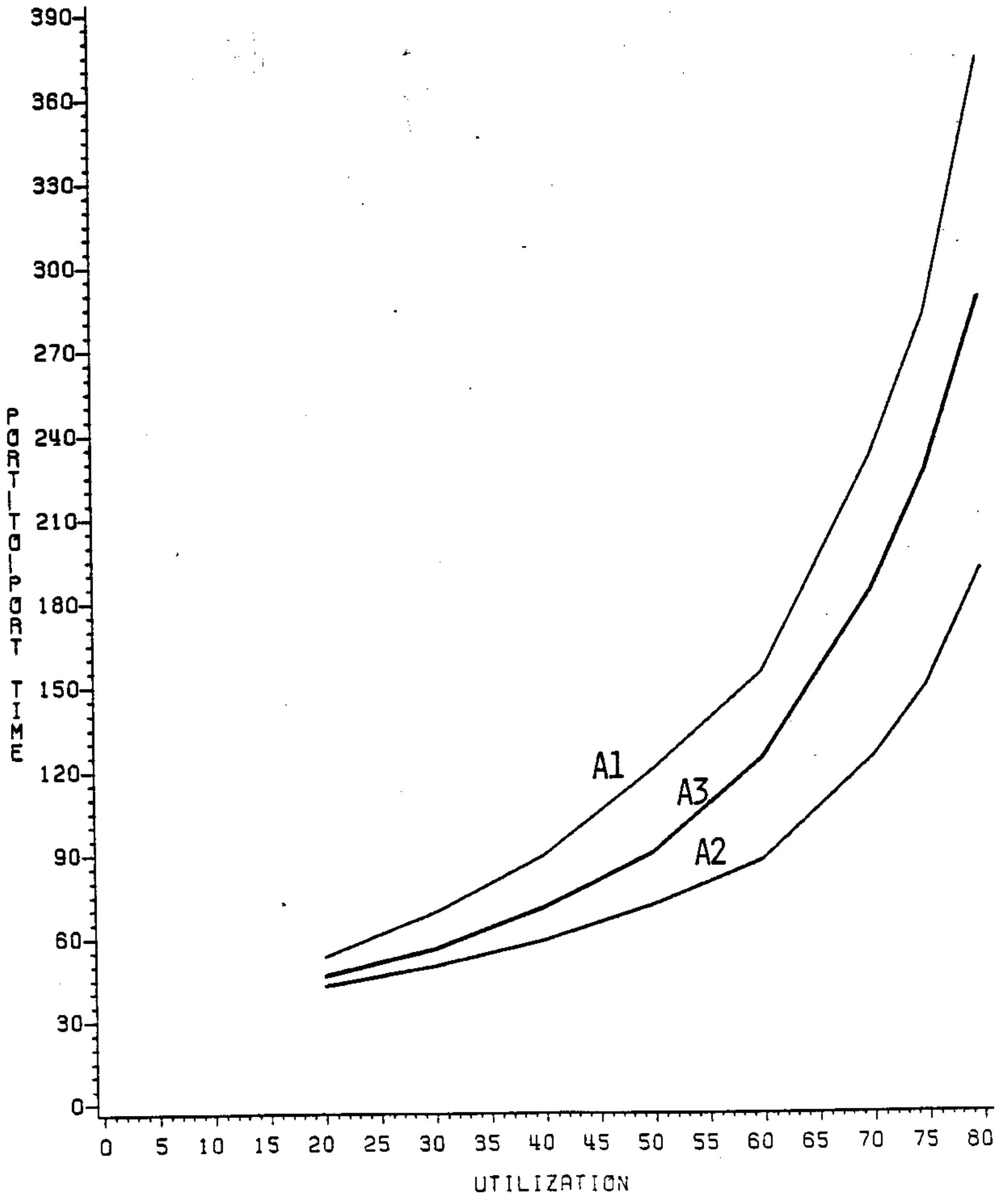


Experiment No. 4 is similar to No. 3 except with different execution times as shown in Fig. 5-4(c). The same three assignments in Fig. 5-4(b) were simulated. Now, Assignment #2 yields the best performance (Fig. 5-6) because it follows the PR rules for most pairs of consecutive modules. Assignment #1 is the worst since it seriously violates the PR rules. We repeat these experiments with exponentially distributed execution times. They yield the same results. The intuitive reasons for the PR rules are as follows:

1. If the arrival process is highly random such as Poisson, there would be periods of bursty arrivals. If the job arrival process is deterministic, the work load is *evenly spread* over the time. As a result, the average queue-length at every computer should be smaller (smaller average module wait-time) than that of a Poisson arrival process. Thus, "let small jobs (modules, in our case) run first, while large jobs wait" yields less average wait-time.

2. For two consecutive modules assigned to a single computer, if the second module is much larger than the first one, the second one will act as *regulator valve* which regulates job flow into the next computer. For instance, in Assignment #1 of Experiment No.3, M_2 at Computer 1 is a large module, therefore the arrivals of M_3 at Computer 2 won't be in bursty mode — instead, arrivals for M_3 are fairly evenly spread over the time, which results in short queue at Computer 2 and thus, short

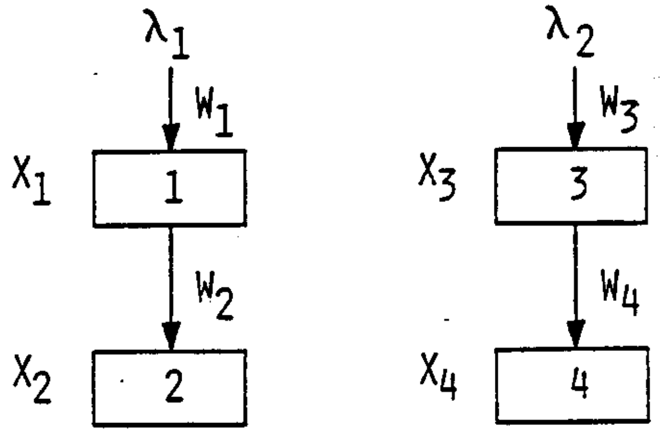
FIG. 5-6. RESULTS OF EXPERIMENT NO. 4



wait time for M_3 and M_4 at Computer 2. In the same manner, M_4 acts as a regulator valve for job flow into Computer 3. On the other hand, the poor response time of Assignment #2 is mainly due to the fact that there is a high possibility that an M_2 arrives while a previous M_2 is still in execution (and more other M_2's might be waiting in the queue), even if M_2 arrival process at Computer 2 is not bursty. That is, there is a high probability to see several M_2's exist, one after another, in the queue. After the first M_2 finishes execution, an M_3 is enabled and placed *behind* all existing M_2's in the queue. This particular M_3 would experience a long wait time because those M_2's in front of it have a large execution time. Later on, we see multiple M_3's *next to each other* which will then quickly finish their execution one after another (because of the small execution time of M_3) and dump multiple M_4 arrivals to Computer 3 *in a bursty mode.*

Having realized that the module-size ratio of consecutive modules influences response time, we should determine whether two consecutive modules M_1 and M_2 (with a module-size ratio x_1/x_2) should be separated or co-located.

Consider the control-flow graph in Fig. 5-7 where all modules have deterministic execution times. Let $x_1 = x_2$, $x_3 = x_4$ (thus, $\frac{x_2}{x_1} = \frac{x_4}{x_3}$), module-size ratio $r_{i,j} = x_j/x_i$, and job arrival rate λ_1 equals to λ_2. Both Assignments #1 and #2 balance the processor loads. We like to determine the benefit or



$X_1 = X_3 \qquad X_2 = X_4 \qquad \lambda_1 = \lambda_2$

ASSIGN-MENT	CPU 1	CPU 2
1	M1,M2	M3,M4
2	M1,M4	M2,M3

FIG. 5-7. EXPERIMENT FOR DERIVING WAIT-TIME RATIO $R_W(A_1/A_2)$ BETWEEN ASSIGNMENTS A_1 AND A_2 , AS A FUNCTION OF THE CORRESPONDING SIZE RATIO X_2/X_1

penalty of assigning M_1 and M_2 to a same computer in terms of response time as a *function* of $r_{1,2}$. That is, we are looking for a threshold value θ such that: if $r_{1,2} > \theta$, M_1 and M_2 should be assigned to a same computer, otherwise they should be assigned to separated computers. More generally, we are looking for a relation (a function) which maps a size ratio into a benefit index (or penalty index if negative value).

Because of the symmetry in this control-flow graph and in loading on both computers, the two threads in the graph have the same response time, which is $w_1 + x_1 + w_2 + x_2$ (or $w_3 + x_3 + w_4 + x_4$), where w_i is the queueing wait-time for module M_i. A model has been developed in [CHU84c] to estimate the wait time w_i for any given module assignment on any control-flow graph. Since x_1, x_2, x_3, and x_4 are constants independent of module assignment, the ***wait-time ratio between two assignments***, $R_w = R(A_1/A_2) = \dfrac{w_1(A_1) + w_2(A_1)}{w_1(A_2) + w_2(A_2)}$, can be used as a measure for the benefit index mentioned above: If $R_w < 1$, then Assignment #1 is better than Assignment #2, i.e., we should assign the consecutive modules M_1 and M_2 to one computer, and the other pair of consecutive modules M_3 and M_4 to another computer. If $R_w > 1$, then Assignment #2 is better than Assignment #1 and consecutive modules should be run on different computers. Fig. 5-8(a) shows the wait-time ratio R_w for various module-size ratio $r_{1,2} = x_2/x_1$. The horizontal axis is the processor utilization $\rho = \rho_1 + \rho_2$ where $\rho_1 = \lambda_1 x_1$ and $\rho_2 = \lambda_1 x_2$ are

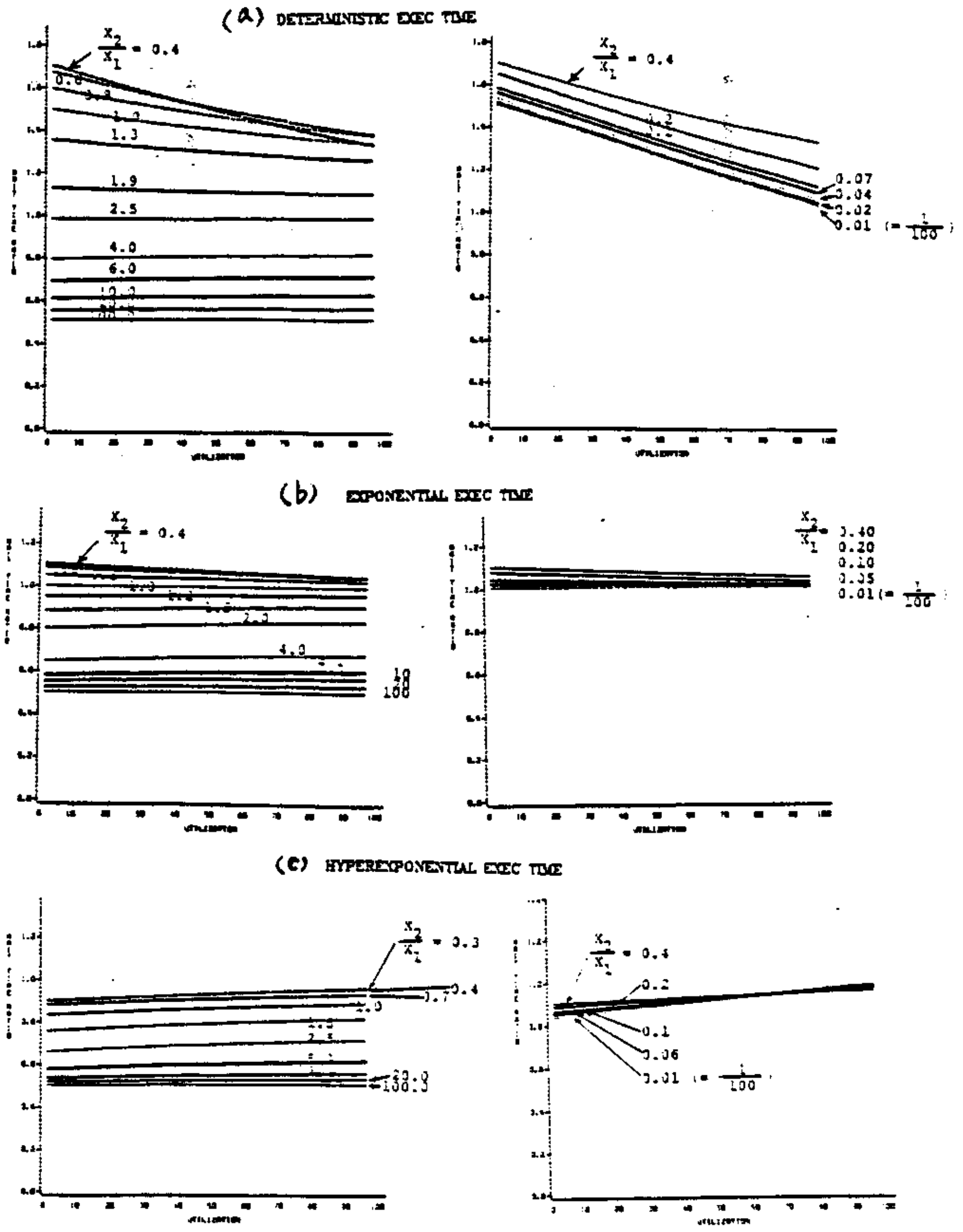
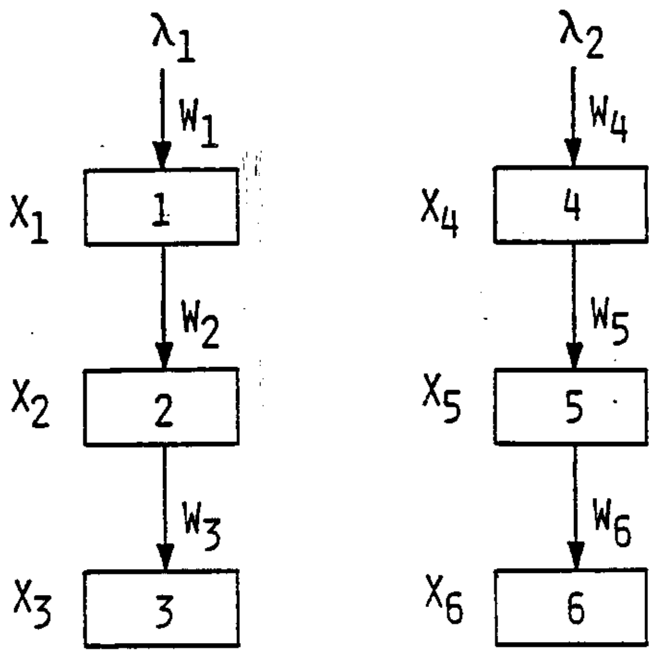


Fig. 5-8. Wait-time ratio between two assignments as a function of program module-size ratio

contributed by the execution of M_1 and M_2, respectively. Note that as $r_{1,2}$ decreases, R_w increases until reaching approximately to 1.7; then it reverses the trend and decreases. R_w only vary slightly with the processor utilization. Figs. 5-8(b) and 5-8(c) are obtained if the execution time of each module is changed from a deterministic value to an exponentially or hyperexponentially distributed random variable. Since the execution times of most programs are more deterministic than exponentially or hyperexponentially distributed, the following discussion will be for deterministic execution time.

We shall now study the execution of *three* consecutive modules (Fig. 5-9). Now the wait-time ratio is $R_w = \frac{w_1(A_1) + w_2(A_1) + w_3(A_1)}{w_1(A_2) + w_2(A_2) + w_3(A_2)}$. Our analysis shows that if the size of M_1 is fixed (thus, $\rho_1 = \lambda_1 x_1$ is fixed), as the ratio of M_3 to M_2 ($r_{2,3} = x_3/x_2 = \rho_3/\rho_2$) decreases, the wait-time ratio R_w increases to a certain point; R_w then reverses the trend and decreases (Fig. 5-10). Likewise, fixing M_3 and varying the size ratio of M_2 to M_1, we observe similar results. These relations between R_w and $r_{i,j}$ are similar to our previous observations for the two-module-thread cases as were shown in Fig. 5-8. Similar relations are exhibited for a control-flow graph consisting of four or five modules in each thread. Finally, if x_1, x_2, and x_3 in Fig. 5-9 are varied simultaneously, the results are shown in a 3-dimension diagram (Fig. 5-11(a)) and the corresponding contour plot when projected on the 2-dimensional plane (Fig. 5-11(b)). Note that when both size ratios $r_{2,3}$ and $r_{1,2}$ are large,



$X_1 = X_4 \qquad X_2 = X_5 \qquad X_3 = X_6 \qquad \lambda_1 = \lambda_2$

Assign-ment	CPU 1	CPU 2
1	M1,M2,M3	M4,M5,M6
2	M1,M5,M3	M4,M2,M6

Fig. 5-9. Experiment on 3 consecutive modules in each control-flow thread

Fig. 5-10. Wait-time ratio as a function of module-size ratio X_3/X_2 (for various processor utilization) for 3-module experiment

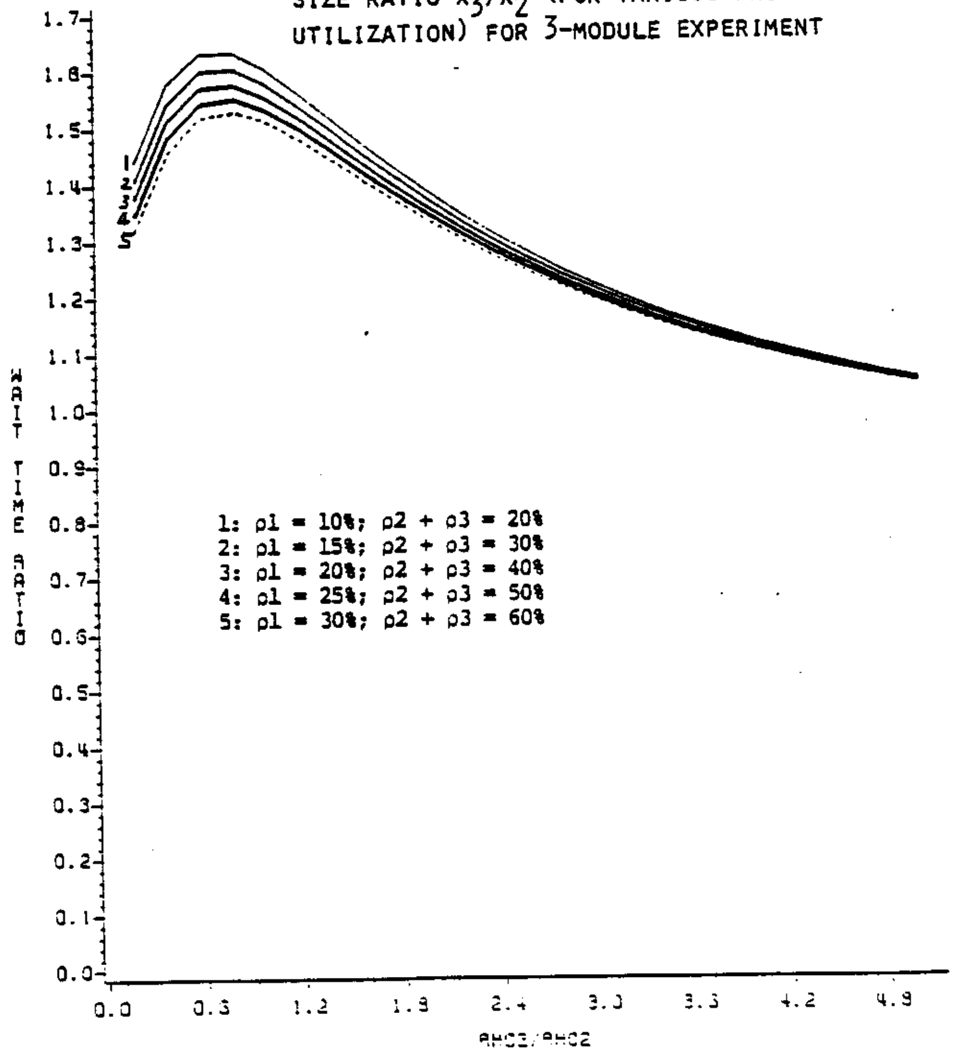


FIG. 5-11(A) 3-D DIAGRAM FOR WAIT-TIME RATIO AS A FUNCTION OF BOTH MODULE-SIZE RATIOS, X_2/X_1 AND X_3/X_2

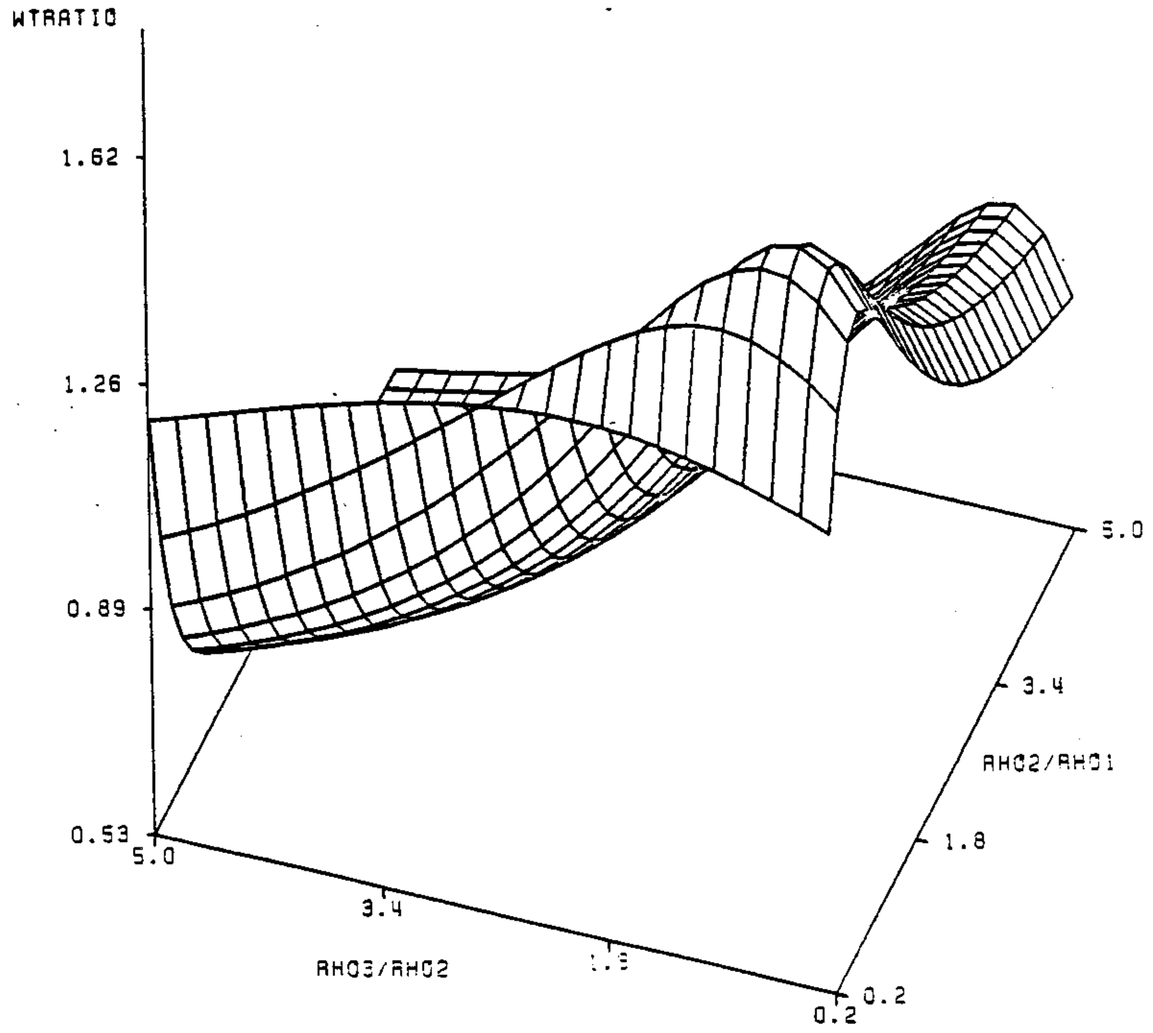
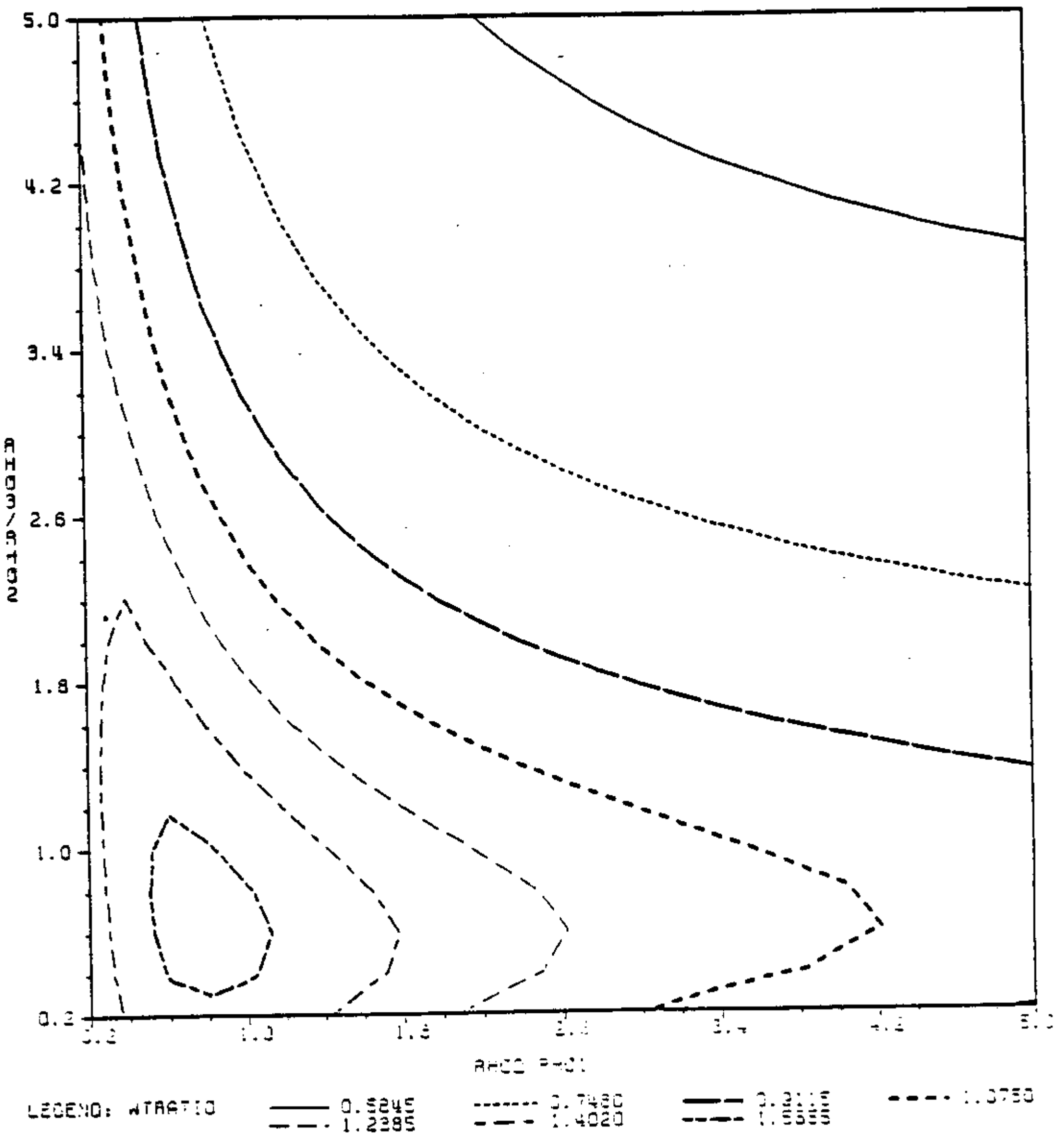


FIG. 5-11(B) CONTOUR PLOT OF THE 3-D DIAGRAM SHOWN IN 5-11(A)



the wait-time ratio R_w is the smallest. Thus, assigning all three consecutive modules to a same computer (i.e., Assignment #1) yields better response time, which is consistent with our previous observation. If one of the ratios $r_{2,3}$ and $r_{1,2}$ is large while the other is small, then the benefit from one module pair is canceled out by the penalty from another pair. As a result, both assignments have similar wait times. If both $r_{2,3}$ and $r_{1,2}$ are small, then Assignment #2 is better.

Our experimental observations reveal that in assigning modules to computers, each pair of consecutive modules in the control-flow graph can be treated independently, and using the PR rules on each individual pair of consecutive modules in task allocation yields good task response time.

5.2 TASK ALLOCATION WITH PRECEDENCE RELATION

We shall now include the PR rules into our task-allocation algorithm. The decision on grouping two consecutive modules or not should base on the two possibly conflicting factors: IMC and PR (i.e., module-size ratio). Therefore, *IMC index* and *PR index* are developed. First let us define (in Step 1.1) the following IMC index and PR index between modules M_i and M_j:

$$\gamma_{IMC}(i,j) = \frac{IMC_{i,j}}{\theta_{IMC}} \qquad i = 1,\ldots,J; \quad j = 1,\ldots,J$$

$$\gamma_{PR}(i,j) = \frac{1 - R_w(r_{i,j})}{I_{PR}} \qquad i = 1,\ldots,J; \quad j = 1,\ldots,J$$

where $\bar{x}_i$ is the average module size of M_i, and R_w is a function of $r_{i,j}$ (see Fig. 5-8). Note that a R_w value on the Y-axis of Fig. 5-8 always lies in the range of [0, 2]. This value should reflect the PR index $\gamma_{PR}(i,j)$ — a positive (negative) R_w should correspond to a positive (negative) $\gamma_{PR}(i,j)$ and prescribes the co-location (separation) of modules M_i and M_j. For simplicity, we divide the range [0, 2] on the Y-axis into N_{PR} equal-size intervals for PR index levels. The interval size is $I_{PR} = 2.0/N_{PR}$. Because R_w equals 1 at the break point between grouping or separating two consecutive modules, the function $(1 - f)/I_{PR}$ gives the PR index $\gamma_{PR}(i,j)$ for any given module-size ratio. For example, if we choose to have 20 PR levels within the range [0, 2], we have an interval size $I_{PR} = 2.0/20 = 0.1$. If f determines the R_w to be 1.4, then γ_{PR} = -4, which opposes the grouping of the modules. To complete our new algorithm, we should replace Step 1.3 of Algorithm I-A with the following:

```
1.3  If $\gamma_{IMC}(i,j) + \gamma_{PR}(i,j) \leq 0$
         go to Phase II.
```

Let us denote this generalized algorithm as Algorithm P-I-A (adding the initial "P" for PR).

There exist three variables in Algorithm P-I-A — α, β, and N_{PR} (or, I_{PR}). For a given distributed system (e.g., the DPAD), if N_{PR} is fixed, then all $\gamma_{PR}(i,j)$ values are uniquely determined. In that case, adjusting the α value will influence the sign (positive or negative) of the sum $\gamma_{IMC}(i,j) + \gamma_{PR}(i,j)$, and thus determine whether M_i and M_j should be co-located on a computer (assuming a fixed β value). If we reduce N_{PR} by half and double the α value, then the minimum-bottleneck assignment generated by Algorithm P-I-A will remains unchanged because both $\gamma_{IMC}(i,j)$ and $\gamma_{PR}(i,j)$ are reduced by half. However, if we reduce N_{PR} while keeping a constant α, the influence of PR is reduced. On the other hand, increasing N_{PR} while keeping a constant α will result in less IMC influence. Table 5-1 contrasts γ_{PR}'s and γ_{IMC}'s for various values of α and N_{PR}. γ_{PR}'s and γ_{IMC}'s in this table have been rounded to the nearest integers. We summarize the heuristic task-allocation algorithm, Algorithm HEU, as follows:

```
Fix the number of PR intervals $N_{PR}$;

Do $\alpha = \alpha_1\%$ to $\alpha_2\%$;
   Do $\beta = \beta_1\%$ to $\beta_2\%$;
      Perform Algorithm P-I-A;
   end;
end;
```

The experimental results on DPAD and two other systems reveal that using $N_{PR} = 20$ and α between 1% and 10% generates good assignments. A good range for β is between 60% and 120%. This is because too small a β would retard proper module grouping while too large a β makes it impossible to balance the loads during Phase II.

		Number of PR intervals, N_{PR}								
M_i	M_j	100	50	40	20	10	8	6	4	2
2	3	-16	-8	-6	-3	-2	-1	-1	-1	0
5	7	-8	-4	-3	-2	-1	-1	0	0	0
6	7	-8	-4	-3	-2	-1	-1	0	0	0
8	9	-8	-4	-3	-2	-1	-1	0	0	0
16	18	-8	-4	-3	-2	-1	-1	0	0	0
17	18	-8	-4	-3	-2	-1	-1	0	0	0
14	23	-19	-9	-7	-4	-2	-1	-1	-1	0
20	21	-8	-4	-3	-2	-1	-1	0	0	0

$\gamma_{PR}(I,J)$

		α%									
M_i	M_j	3	6	9	12	15	18	21	24	27	30
13	14	109	54	36	27	21	18	15	13	12	10
8	6	28	14	9	7	5	4	4	3	3	2
8	10	28	14	9	7	5	4	4	3	3	2
8	16	28	14	9	7	5	4	4	3	3	2
14	23	18	9	6	4	3	3	2	2	2	1
22	1	16	8	5	4	3	2	2	2	1	1
22	2	16	8	5	4	3	2	2	2	1	1
22	4	16	8	5	4	3	2	2	2	1	1
22	8	16	8	5	4	3	2	2	2	1	1
14	13	6	3	2	1	1	1	0	0	0	0
8	9	5	2	1	1	1	0	0	0	0	0
8	17	5	2	1	1	1	0	0	0	0	0
8	19	5	2	1	1	1	0	0	0	0	0
8	20	5	2	1	1	1	0	0	0	0	0
9	13	2	1	0	0	0	0	0	0	0	0
19	20	2	1	0	0	0	0	0	0	0	0
4	5	1	0	0	0	0	0	0	0	0	0
4	6	1	0	0	0	0	0	0	0	0	0
4	7	1	0	0	0	0	0	0	0	0	0
18	20	0	0	0	0	0	0	0	0	0	0
22	23	0	0	0	0	0	0	0	0	0	0
18	19	0	0	0	0	0	0	0	0	0	0
3	13	0	0	0	0	0	0	0	0	0	0
2	3	0	0	0	0	0	0	0	0	0	0
4	13	0	0	0	0	0	0	0	0	0	0
16	18	0	0	0	0	0	0	0	0	0	0
17	18	0	0	0	0	0	0	0	0	0	0
5	7	0	0	0	0	0	0	0	0	0	0
6	7	0	0	0	0	0	0	0	0	0	0
7	13	0	0	0	0	0	0	0	0	0	0
8	13	0	0	0	0	0	0	0	0	0	0
20	21	0	0	0	0	0	0	0	0	0	0
18	8	0	0	0	0	0	0	0	0	0	0
10	8	0	0	0	0	0	0	0	0	0	0
10	18	0	0	0	0	0	0	0	0	0	0

$\gamma_{IMC}(I,J)$

TABLE 5-1. COMPARE γ_{PR} AND γ_{IMC}

5.3 MODULE ASSIGNMENT WITH PR FOR DPAD

Applying Algorithm HEU to the DPAD produces the bottlenecks shown in Fig. 5-12. Simulation reveals that the response-time performance of the assignment with a bottleneck of 74985 MLI (generated by $\alpha = 3\%$ and $\beta = 60\%$) is slightly better than the one with a bottleneck of 74312 MLI (generated by $\alpha = 4\%$ and $\beta = 70\%$). This result shows that a smallest bottleneck does not necessarily yield the *best* response time. However, assignments with close bottleneck values always yield similar response times.

The assignment with a bottleneck of 74985 MLI performs only slightly better than the assignment generated by Algorithm I-A (Fig. 5-13). In fact, using the same parameters $\alpha = 5\%$ and $\beta = 75\%$ as were used in Algorithm I-A in Chapter 4, Algorithm P-I-A will generate exactly the same assignment as Algorithm I-A because of the following reasons. Consider Table 5-1 and imagine a column of γ_{IMC} for $\alpha = 5\%$. A pair of modules recommended to be grouped by Step 1.3 of Algorithm I-A are recommended *the same* by Step 1.3 of Algorithm P-I-A. And a pair not recommended to be grouped by Algorithm I-A are not recommended by Algorithm P-I-A. For instance, both $IMC_{2,3} < \overline{AET} \times 5\%$ according to Algorithm I-A and $\gamma_{IMC}(2,3) + \gamma_{PR}(2,3) = 0 + (-2) < 0$ according to Algorithm P-I-A, which recommends separating M_2 and M_3. On the other hand, both $IMC_{14,23} > \overline{AET} \times 5\%$ and $\gamma_{IMC}(14,23) + \gamma_{PR}(14,23) > 0$, which recommends grouping M_{14} and M_{23}.

20 PR Levels

	1%	2%	3%	4%	5%	6%	7%	8%	9%	10%
60%	77739	77739	74985	75705	75705	75705	75705	75705	75705	75705
70%	79739	79739	76000	74312	74312	74312	74312	74312	74312	74312
80%	79739	79739	76000	74312	74312	74312	74312	74312	74312	74312
90%	80661	80661	76938	76938	76938	76938	76938	76938	76938	76938
100%	79739	79739	76000	74312	74312	74312	74312	74312	79481	79481
110%	79739	79739	76000	74312	74312	74312	74312	74312	79481	79481
120%	80661	80661	76938	76938	76938	76938	76938	76938	79481	79481

FIG. 5-12. MINIMUM BOTTLENECKS FOR DPAD GENERATED BY ALGORITHM HEU.

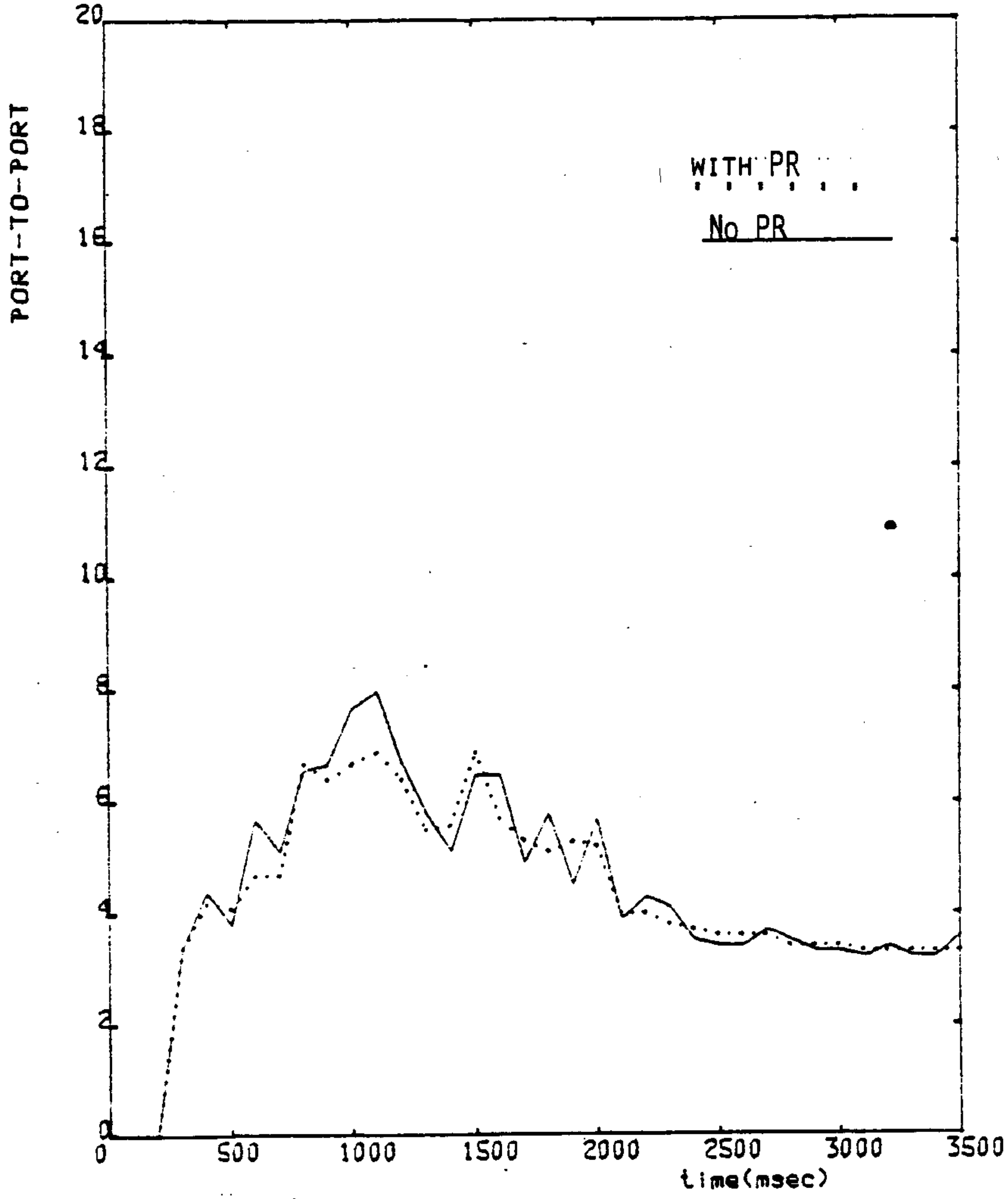


FIG. 5-13. PTP TIME FOR PRECISION-TRACK THREAD -- COMPARE ASSIGNMENTS WITH AND W/O PR

5.4 EXAMPLE OF RESPONSE-TIME IMPROVEMENT BY PR

An example is given in this section which demonstrates that a considerable improvement on response time can be obtained by considering PR in the task allocation. Consider the control-flow graph shown in Fig. 5-14 where each program module has a deterministic execution time of either 100 or 1000 μs, thus the size ratio of every pair of consecutive modules is either 0.1 or 10. According to the PR rules derived in Section 5.1, we should assign M_4 and M_6 on the same computer, and M_9 on a different computer. Using the model of [CHU84b], we can estimate the AET for a specified time interval for each module. In this example let us assume a time interval of 100 job arrivals, the inter-arrival time is exponentially distributed, and *each* arrival invokes the entire control-flow graph once. The estimated AET's are shown in Table 5-2. Let us further assume that the IMC sizes for all communicating module pairs are about equal, either 1400 or 1500 μs as shown in Table 5-3 and Fig. 5-15, so that the IMC plays a less important role than PR. Given these PR, IMC, and AET, the module assignments generated by Algorithms I-A and P-I-A are shown in Fig. 5-16. Both assignments have fairly balanced processor loads with similar bottleneck values. Therefore, if they differ significantly in response time, it is due to the PR consideration. Note that for the assignment generated by Algorithms P-I-A, most module pairs are assigned (either co-located or separated) according to our PR rules instead of by IMC size. For example, the module size ratio $r_{4,6}$ is $x_6/x_4 = 10$, and M_4

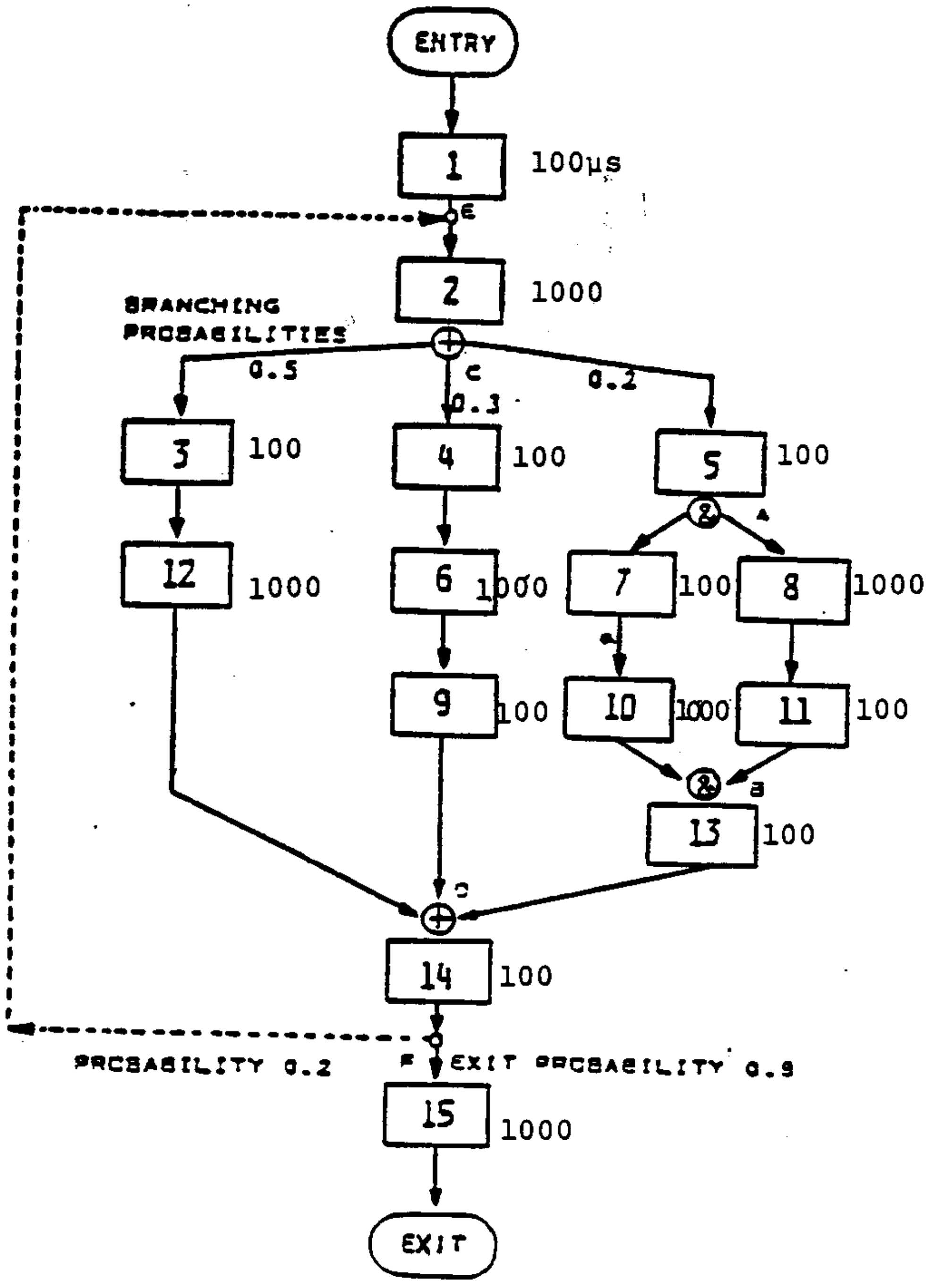


FIG. 5-14.
A SAMPLE TASK CONTROL-FLOW GRAPH

Table 5-2. AET for Modules in Figure 5-14

Module	Exec Time/job arrival x_i X (invocations/arrival)	AET/100 arrivals
1	100 X 1 = 100μs	10,000μs
2	1000 X 1.25 = 1250	125,000
3	100 X 0.625 = 62.5	6,250
4	100 X 0.375 = 37.5	3,750
5	100 X 0.25 = 25	2,500
6	1000 X 0.375 = 375	37,500
7	100 X 0.25 = 25	2,500
8	1000 X 0.25 = 250	25,000
9	100 X 0.375 = 37.5	3,750
10	1000 X 0.25 = 250	25,000
11	100 X 0.25 = 25	2,500
12	1000 X 0.625 = 625	62,500
13	100 X 0.25 = 25	2,500
14	100 X 1.25 = 125	12,500
15	1000 X 1 = 1000	100,000

Table 5-3. IMC List For The System In Figure 5-14

From Module	To Module	IMC/100 Arrival	File-ID
6	9	1500	106
10	13	1500	111
8	11	1500	109
11	13	1500	112
12	14	1500	105
1	2	1400	101
2	3	1400	102
2	4	1400	102
2	5	1400	102
3	12	1400	103
4	6	1400	104
5	7	1400	107
5	8	1400	107
7	10	1400	108
9	14	1400	110
13	14	1400	113
14	15	1400	114

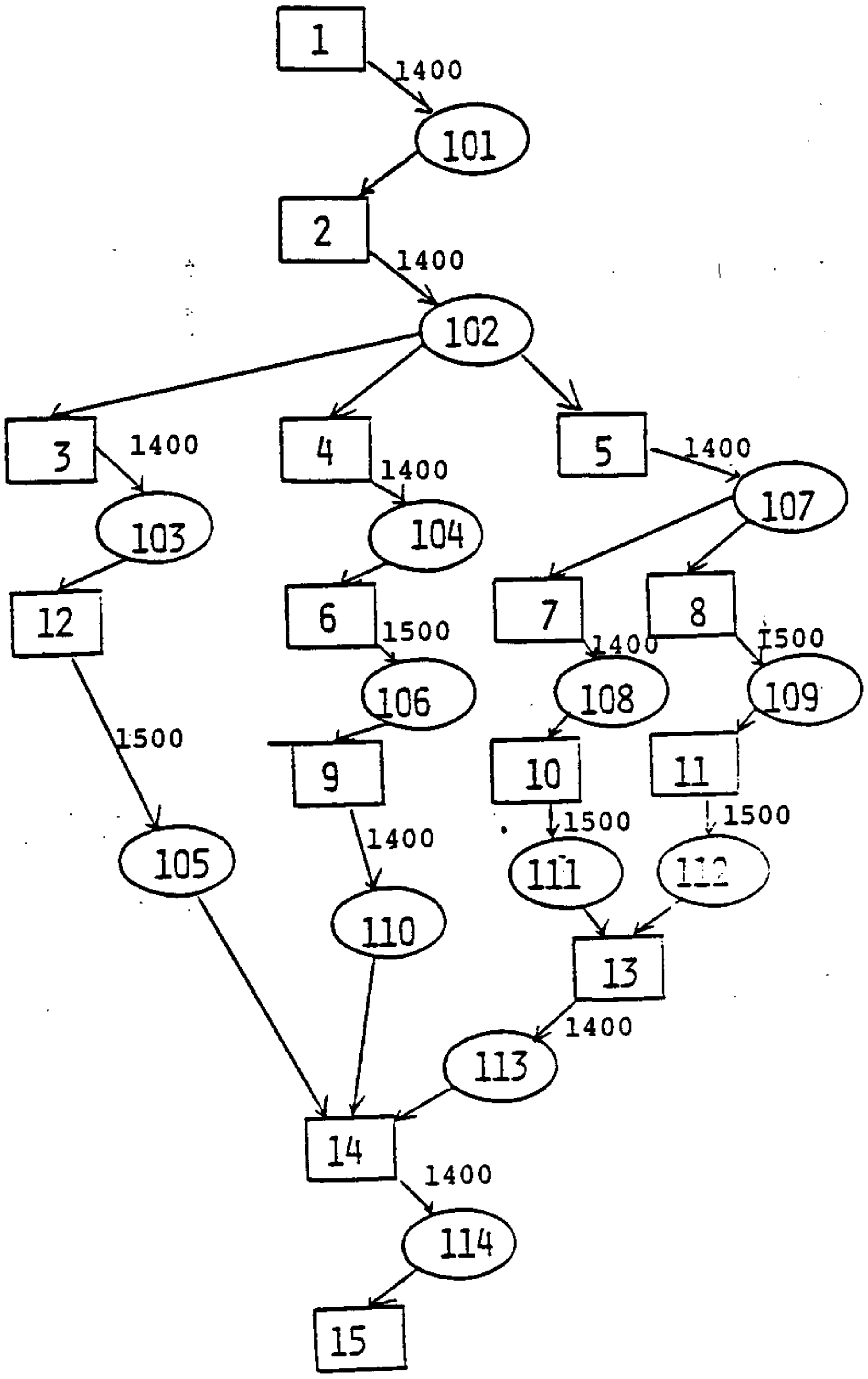


FIG. 5-15. DATA-FLOW GRAPH FOR THE SYSTEM IN FIGURE 5-14

ASSIGNMENT #1
(W/O CONSIDERING P.R.)

CPU1	CPU2	CPU3
1	7	3
2	10	4
9	13	5
	14	6
	15	8
		11
		12

ASSIGNMENT #2
(CONSIDERING P.R.)

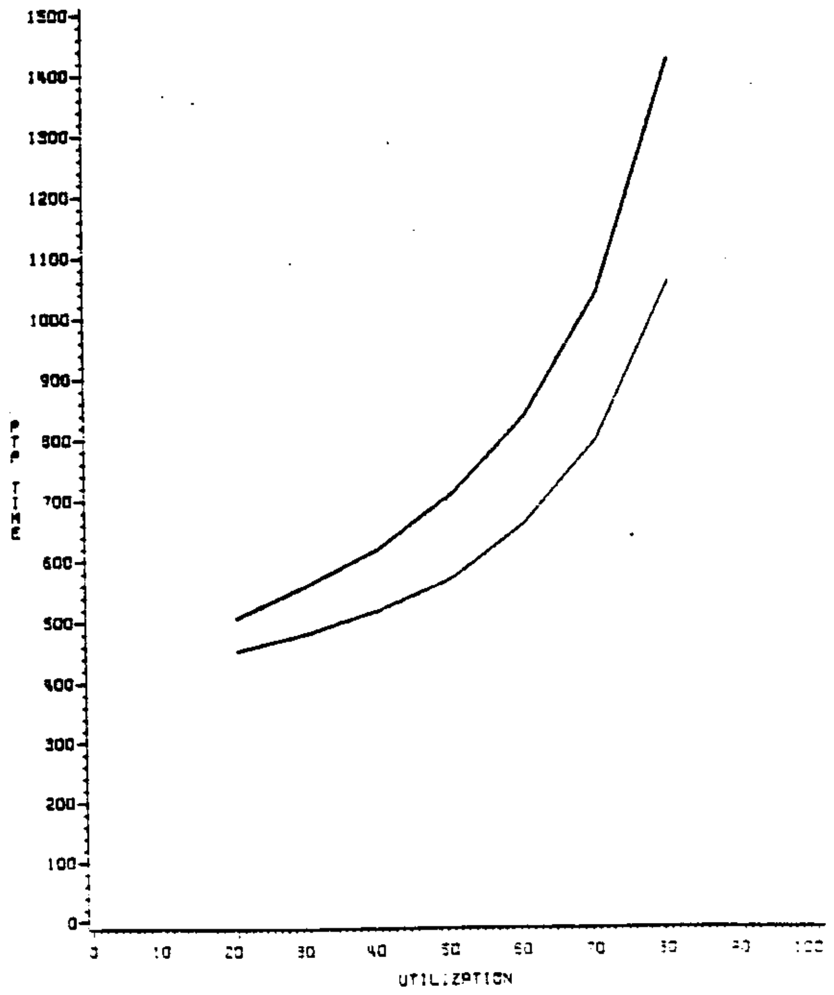
CPU1	CPU2	CPU3
1	6	3
2	9	5
4	15	7
		8
		10
		11
		12
		13
		14

Fig. 5-16. Module Assignments for the System in Fig. 5-14

and M_6 are co-located on computer 3. On the other hand, $r_{8,9} = 0.1$ and M_8 is separated from M_9 although $IMC_{8,9}$ is larger than $IMC_{4,6}$.

These two assignments are simulated via the PAWS simulator. The average response time for each job arrival is measured from when the job arrives at the system until it finishes the execution of M_{15}. Figure 38 compares the response time between the two assignments. Note that Algorithm P-I-A yields better response time than Algorithm I-A, with 10.8% improvement at processor utilization $\rho = 20\%$ and 25.7% improvement at $\rho = 80\%$.

Fig. 5-17. Compare task response time for assignments in Fig. 5-16



CHAPTER 6

CONCLUSIONS AND DISCUSSIONS

6.1 CONCLUSIONS

The three important parameters in task allocation are intermodule communication (IMC), accumulative execution time (AET) of each module, and precedence relations (PR) among program modules. IMC is the communication between program modules through shared files. When a module on a computer writes to or reads from a shared file on *another* computer, it requires extra processing and communication overhead known as IPC (interprocessor communication). Therefore, a task-allocation algorithm should try to minimize IPC by assigning a pair of heavily communicating modules to the same computer. On the other hand, AET always contributes to processor load; its contribution is independent of task allocation. We have proposed a methodology to measure and characterize both IMC and AET.

From the insights obtained with our simulation and the IMC measurements, an analytical model has been constructed to estimate the IMC and AET in a distributed processing system. This model was applied to the DPAD system and it has been shown that the model is able to provide fairly accurate prediction.

An objective function for task allocation that considers both IMC and AET is proposed. It is based on the model of bottleneck processor. An heuristic task-allocation algorithm based on this objective function was also presented to effectively search for good assignments.

The third parameter for task allocation is the ***precedence relationship (PR)*** which specifies that a program module can not be enabled before all its predecessor(s) finish execution. Simulation study and analysis reveal that the module-size ratio of two consecutive modules affects task response time. A set of PR rules are generated to determine if the consecutive modules should be co-located on the same computer. Allocating the modules according to the PR rules yields performance improvement.

An improved heuristic algorithm for task allocation was presented, based on PR, IMC, and AET. The algorithm was applied to the DPAD system and another distributed system example. The results reveal that a module assignment considering PR yields better response time than an assignment without PR consideration.

6.2 FUTURE RESEARCH AREAS

Many related issues in task allocation remain unsolved and need further investigation.

a. Replication of files — A file-replication policy should be developed to decide how many copies of a replicated file are needed and where these copies should reside, for either access speed, fault-tolerance, or reduction of file-update message volume. Data consistency among the copies is a major concern that affects performance in file replication.

b. Replication of program modules — Some modules might be so frequently invoked that their processing requirement cannot be met by a single processor. It is desirable to process identical copies of a given module on multiple computers, each processing a subset of invocations of that module. For that, techniques need be developed to decide a) the needed number of copies for a program module, b) the file structure (centralized, replicated, or partitioned [CHU76]) for the files accessed by a replicated program module, c) the number of copies (and the sites) a file should be replicated and/or partitioned into, and d) the policy for distributing module invocations among all computers which run a copy of the invoked module.

c. Task scheduling policy — Scheduling policy plays an important role in real-time systems. Besides the FCFS discipline, there might be other scheduling policies more suitable for distributed real-time systems. One possibility is to schedule multiple modules and process them as a batch. As a result, some lines of code (e.g., the initializing housekeeping code) can be shared by all modules in the batch. This reduction of overhead should be weighed against the increased complexity in task scheduling.

REFERENCES

ANDE75 G. A. Anderson, "Computer interconnection structure: taxonomy, characteristics and examples," *ACM Computing Surveys*, 7 (1975), pp. 197-213.

ARNO79 R. G. Arnold, R. P. Ramseyer, L. B. Wing, and E. A. Householder, "MMBC architecture," in *Proc. 1st Intl. Conf. on Distributed Computing Systems*, Oct. 1979, pp. 707-724.

BAER68 J. L. Baer, "Graph Models of Computations in Computer Systems," Ph.D dissertation, Report No. 68-46, UCLA-10P14-51, Univ. of California, Los Angeles, 1968.

• BAKE80 C. T. Baker, "Logical distribution of applications and data," *IBM System Journal*, vol. 19, no. 2, pp. 171-192, 1980.

BERN81 Philip A. Bernstein and Nathan Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys*, vol. 13, no. 2, pp. 185-221, June 1981.

BERR82 Robert Berry, K. Mani Chandy, Jay Misra, and Doug Neuse, "PAWS 2.0 — Performance Analyst's Workbench System: User's Manual," Information Research Associates, Austin, Texas, December 1982.

BLAN84 Tom Blank, "A survey of hardware accelerators used in computer-aided design," *IEEE Design & Test of Computers*, vol. 1, no. 3, pp. 21-39, Aug. 1984.

BOKH79 S. H. Bokhari, "Dual processor scheduling with dynamic reassignment," *IEEE Trans. on Software Eng.*, vol. SE-5, no. 4, pp. 341-349, July 1979.

BOKH81 S. H. Bokhari, "On the mapping problem," *IEEE Trans. on Computers*, vol. C-30, no. 3, pp. 207-214, Mar. 1981.

CHU69 Wesley W. Chu, "Optimal file allocation in a multiple computer system," *IEEE Trans. on Computers,* vol. C-18, no. 10, pp. 885-889, Oct. 1969.

CHU76 Wesley W. Chu, "Performance of file directory systems for distributed data bases," in *Proc. AFIPS National Computer Conf.,* vol. 45, pp. 577-587, 1976.

CHU78 Wesley W. Chu, D. Lee, and B. Iffla, "A distributed processing system for naval data communication networks," in *Proc. AFIPS National Computer Conf.,* vol. 47, pp. 783-793, 1978.

CHU80 Wesley W. Chu, Leslie J. Holloway, Min-Tsung Lan, and Kemal Efe, "Task allocation in distributed data processing," *Computer,* vol. 13, no. 11, pp. 57-69, Nov. 1980.

CHU82 W. W. Chu, J. Hellerstein, M. T. Lan, and L. Holloway, "Research on the shared database kernel for the BMD application," Dept. Computer Science, Report # CSD-820430, Univ. of California, Los Angeles, April 30, 1982.

CHU83 W. W. Chu, J. Hellerstein, M. T. Lan, and J. M. An, "Database management algorithms for advanced BMD applications," Dept. Computer Science, Report # UCLA-ENG-83-20 (CSD-830430), Univ. of California, Los Angeles, April 30, 1983.

CHU84a W. W. Chu, J. Hellerstein, M. T. Lan, J. M. An, and K. K. Leung, "Database management algorithms for advanced BMD applications," Dept. Computer Science, Report # UCLA-ENG-84-07 (CSD-840031), Univ. of California, Los Angeles, Apr. 1984.

CHU84b W. W. Chu, M. T. Lan, and J. Hellerstein, "Estimation of intermodule communication (IMC) and its applications in distributed processing systems," *IEEE Trans. on Computers,* vol. C-33, no. 8, pp. 691-699, Aug. 1984.

CHU84c W. W. Chu and K. K. Leung, "Task-response-time model & its applications for real-time distributed processing systems," *5th Real-Time Systems Symposium,* Austin, TX, Dec. 1984.

CHOU82 T. C. K. Chou and J. A. Abraham, "Load balancing in distributed systems," *IEEE Trans. on Software Eng.,* vol. SE-8, no. 4, pp. 401-412, July 1982.

CHOW79 Y. C. Chow and W. H. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Trans. on Computers,* vol. C-28, no. 5, pp. 354-361, May 1979.

DENN68 Denning, P. J., "The working set model for program behavior," *Comm. ACM,* 11, 5, pp. 323-333, May 1968.

DESP78 A. Despain and D. Patterson, "X-tree a structured multiprocessor computer architecture," in *Proc. 5th Symp. on Computer Architecture,* Silver Spring, MD: IEEE Computer Society Press, 1978, pp. 144-151.

EFE82 Kemal Efe, "Heuristic models of task assignment scheduling in distributed systems," *Computer,* vol. 15, no. 6, pp. 50-56, June 1982.

FENG81 Tse-Yung Feng, "A survey of interconnection networks," *Computer,* vol. 14, no. 12, pp. 12-27, Dec. 1981.

FISH73 George S. Fishman, *Concepts and Methods in Discrete Event Digital Simulation.* New York, NY: John Wiley & Sons, Inc., 1973.

GARC78 Hector Garcia-Molina, "Performance comparison of two update algorithms for distributed databases," in *Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks,* Aug. 1978, pp. 108-119.

GENT78 W. M. Gentlemen, "Some complexity results for matrix computations on parallel processors," *J. of ACM,* Jan. 1978, pp. 112-115.

GOKE73 R. Goke and G. J. Lipovski, "Banyan networks for partitioning on multiprocessor systems," *Proc. 1st Symp. on Computer Architecture,* Silver Sprint, MD: IEEE Computer Society Press, 1973, pp. 21-30.

GREE80 M. L. Green, E. Y. S. Lee, S. Majumdar, and D. C. Shannon, "A distributed real-time operating system," in *Proc. Symp. Distributed Data Acquisition, Computing, and Control,* Dec. 1980, pp. 175-184.

GREE80 M. L. Green, E. Y. S. Lee, S. Majumdar, and D. C. Shannon, "Phase III of Distributed Processing Architecture Design (DPAD) program -- the DDP Underlay simulation experiment: tactical applications and d-RTOS models," TRW Defense and Space Systems Group, Special Report 35010-79-A005, May 15, 1980.

GYLY76 V. B. Gylys and J. A Edwards, "Optimal partitioning of workload for distributed systems," in *Proc. COMPCON Fall 76,* Sep. 1976, pp. 353-357.

HAES80 K. Haessig and C. J. Jenny, "Partitioning and allocating computational objects in distributed computing systems," in *Proc. IFIP Congress 1980,* Melbourne, Australia, pp. 593-598.

HARA69 F. Harary, *Graph Theory.* New York, NY: Addison-Wesley, 1969.

HOFF80 R. H. Hoffman, R. W. Smith, and J. T. Ellis, "Simulation software development for the BMDATC DDP underlay experiment," in *Proc. 4th Intl. Computer Software and Applications Conf. (COMPSAC),* Oct. 1980, Chicago, pp. 569-577.

HOLL82 L. J. Holloway, "Task Assignment in a Resource Limited Distributed Processing Environment," Ph.D dissertation, Dept. Computer Science, Univ. of California, Los Angeles, 1982.

IRAN82 K. B. Irani and K.-W. Chen, "Minimization of interprocessor communication for parallel computation," *IEEE Trans. on Computers,* vol. C-31, no. 11, pp. 1067-1075, Nov. 1982.

JENN77 C. J. Jenny, "Process partitioning in distributed systems," in *Proc. NTC 1977,* pp. 31:1-1 — 31:1-10.

KINN79 L. L. Kinney, W. D. Johnson, R. R. Ramseyer, and K. L. Stephens, "Modular missile borne computer hardware modules,", in *Proc. 1st Intl. Conf. Distributed Computing Systems,* Oct. 1979, pp. 736-746.

KNUT73 D. E. Knuth, *The Art of Computer Programming; Vol. 1: Fundamental Algorithms.* Reading, MA: Addison-Wesley, 1973.

LEUN82 K. K. Leung, *Task response-time model,* dissertation in preparation, Dept. Computer Science, University of California, Los Angeles, 1985.

LOCA80 B. N. Locanthi, *The Homogeneous Machine*, Technical Report 3759, Dept. Computer Science, California Institute of Technology, Jan. 1980.

MA82 P. Y. R. Ma, E. Y. S. Lee, and M. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Trans. on Computers*, vol. C-31, no. 1, pp. 41-47, Jan. 1982.

MAHM76 S. Mahmond and J. S. Riordon, "Optimal allocation of resources in distributed information networks," *ACM Trans. on Data Base Systems,* vol. 1, no. 1, pp. 483-497, Mar. 1976.

MALL82 Efrem G. Mallach, "Computer architecture," *Mini-Micro Systems,* Dec. 1982, pp. 246-259.

MARK82 Pauline Markenscoff, "A multiple-processor system for real-time control tasks," in *Proc. 9th Annu. Symp. on Computer Architecture,* Apr. 1982, pp. 274-280.

PRIC81 C. C. Price, "The assignment of computational tasks among processors in a distributed system," in *Proc. Natl. Comput. Conf.*, May 1981, pp. 291-296.

RAMS79 R. R. Ramseyer and R. G. Arnold, "An overview of the MMBC architecture from the requirements and constraints point of view," in *Proc. 1st Intl. Conf. on Distributed Computing Systems,* Oct. 1979, pp. 747-756.

RAO79 G. S. Rao, H. S. Stone and T. C. Hu, "Assignment of tasks in a distributed processing system with limited memory," *IEEE Trans. on Computers,* vol. C-28, no. 4, pp. 291-299, Apr. 1979.

RILE71 W. B. Riley, "Minicomputer networks — A challenge to maxi-computers?" *Electronics,* vol. 44, pp. 56-62, Mar. 29, 1971.

SAUE81a C. H. Sauer and K. M. Chandy, *Computer System Performance Modeling.* Prentice-Hall, Inc., 1981.

SAUE81b C. H. Sauer, E. A. MacNair, and J. F. Kurose, "Computer/communication system modeling with the Research Queueing package, Version 2," *IBM Report No. 128 (39850),* November 2, 1981.

SHEN85 C. C. Shen and W. H. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Trans. on Computers,* vol. C-34, no. 3, pp. 197-203, Mar. 1985.

STON71 H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. on Computers,* vol. C-20, no. 2, pp. 153-161, Feb. 1971.

STON77 H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. on Software Eng.,* vol. SE-3, no. 1, pp. 85-93, Jan. 1977.

STON78a H. S. Stone, "Critical load factors in two-processor distributed systems," *IEEE Trans. on Software Eng.,* vol. SE-4, no. 3, pp. 254-258, May 1978.

STON78b H. S. Stone, and S. H. Bokhari, "Control of distributed processes," *Computer,* Vol. 11, No. 7, pp. 97-106, July 1978.

STON79 Michael Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed INGRES," *IEEE Trans. on Software Eng.,* vol. SE-5, no. 3, pp. 188-194, May 1979.

TREL82 Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins, "Data-driven and demand-driven computer architecture," *ACM Computing Surveys,* vol. 14, no. 1, pp. 93-143, Mar. 1982.

TSUC80 M. Tsuchiya, "Considerations for requirements engineering of distributed processing systems," in *Proc. Symp. Distributed Data Acquisition, Computing, and Control,* Dec. 1980, pp. 61-65.

VICK79 Charlie R. Vick, "A dynamically reconfigurable distributed computing system," Ph.D dissertation, Dept. Electrical Engineering, Auburn Univ., Auburn, Alabama, Dec. 1979.

WU84 William S.-F. Wu, "Minimization of interprocessor communication for parallel computation on an SIMD multicomputer," Ph.D dissertation, Dept. Electrical Engineering, U. of Michigan, Ann Arbor, 1984.

APPENDIX A

DESCRIPTION OF THE TRW DPAD

The TRW Distributed Processing Architecture Design (DPAD) was developed to manage the data processing and radar resources in the BMD Application [GREE80]. The DPAD system is shown in Figure A-1. It consists of four components:

1. the radar interface

2. the interconnection network

3. the distributed Real-Time Operating System (d-RTOS)

4. twenty-three (23) tactical application-program modules

To validate the DPAD concept, TRW produced a simulator consisting of a *fully coded d-RTOS and simulators for the other three components.* Extensive statistical recording and reporting facilities were included in order to assess performance. The original DPAD simulator was developed for the BMDARC testbed and made use of up to eight VAX-11/780 computers interconnected by a PCL 11-B data bus [HOFF80, GREE80]. Later, this simulator was moved to TRW at Redondo Beach, California where it runs as multiple batch jobs on a single VAX-11/780 (under VMS).

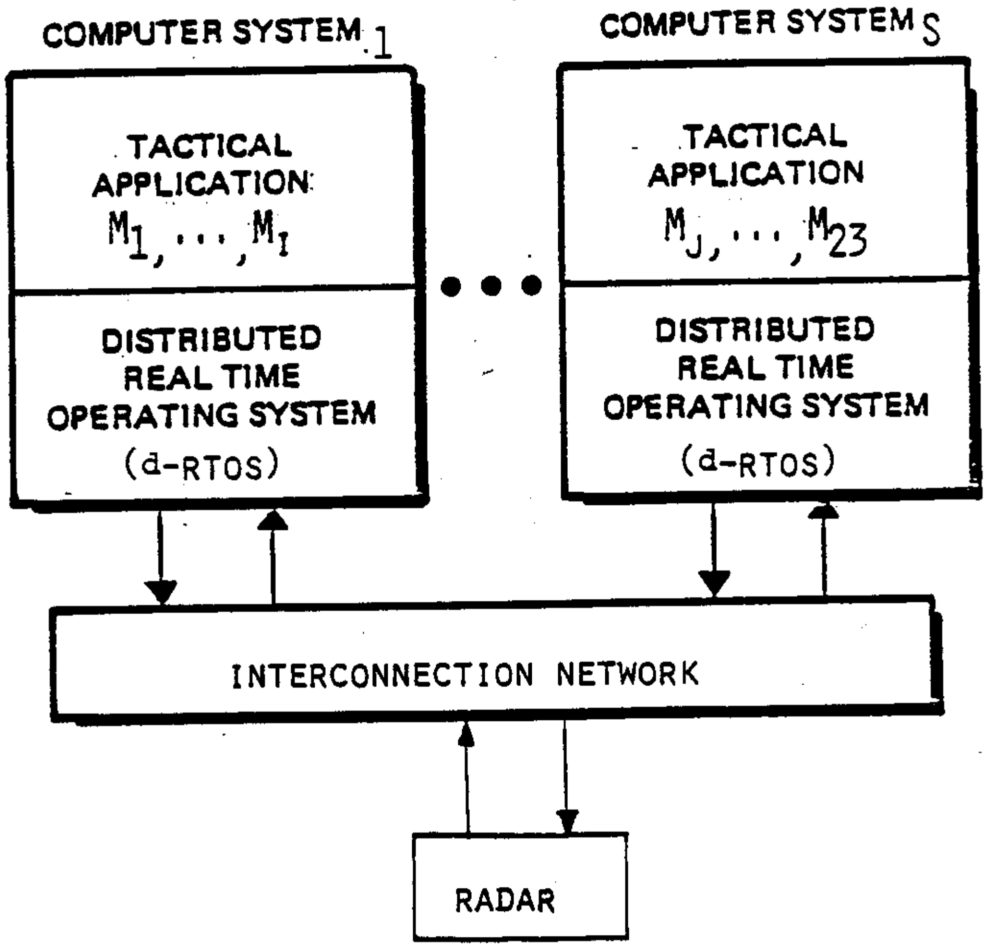


FIG. A-1. THE DISTRIBUTED PROCESSING ARCHITECTURE DESIGN (DPAD) SYSTEM

Residing on each computer is a copy of the d-RTOS. The d-RTOS consists of six cyclically executed tasks (see Figure A-2) responsible for scheduling application modules, shared data management, and resource management. Application modules are scheduled for execution based on their priority and are executed without interrupts. The role of shared database management is to ensure the consistency of the replicated shared data. This is facilitated by the following design constraints:

1. There is only a single writing module for each shared data file.

2. If a module references a shared data file, then a copy of that file is present on the computer to which the module is assigned. Thus every module reads only local file copies and *this incurs no IPC.*

Finally, resource management is concerned with reassigning modules when an overload condition is detected.

Of particular interest to shared database management is the procedure for updating a replicated shared file (see Figure A-3). When a module M_j updates a file F_k, M_j places an entry for F_k in the *Task Completion Queue (TCQ)* of the d-RTOS, indicating the file and record modified. When the d-RTOS task COMPLETE reads this entry from the TCQ, it formats and sends a file-update message (containing the updated record) to all computers which have a copy of that file. *This file-update process causes IPC* which might degrade the system performance. At a receiving computer, the d-RTOS task

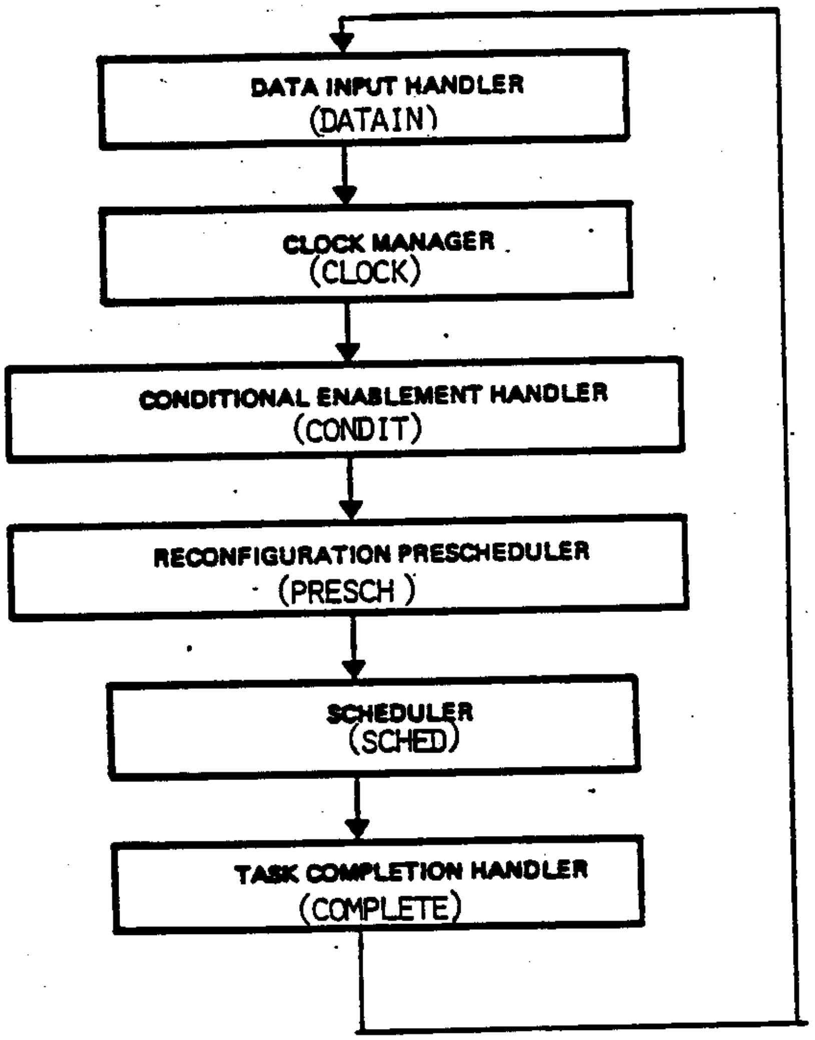


FIG. A-2. THE d-RTOS TASKS

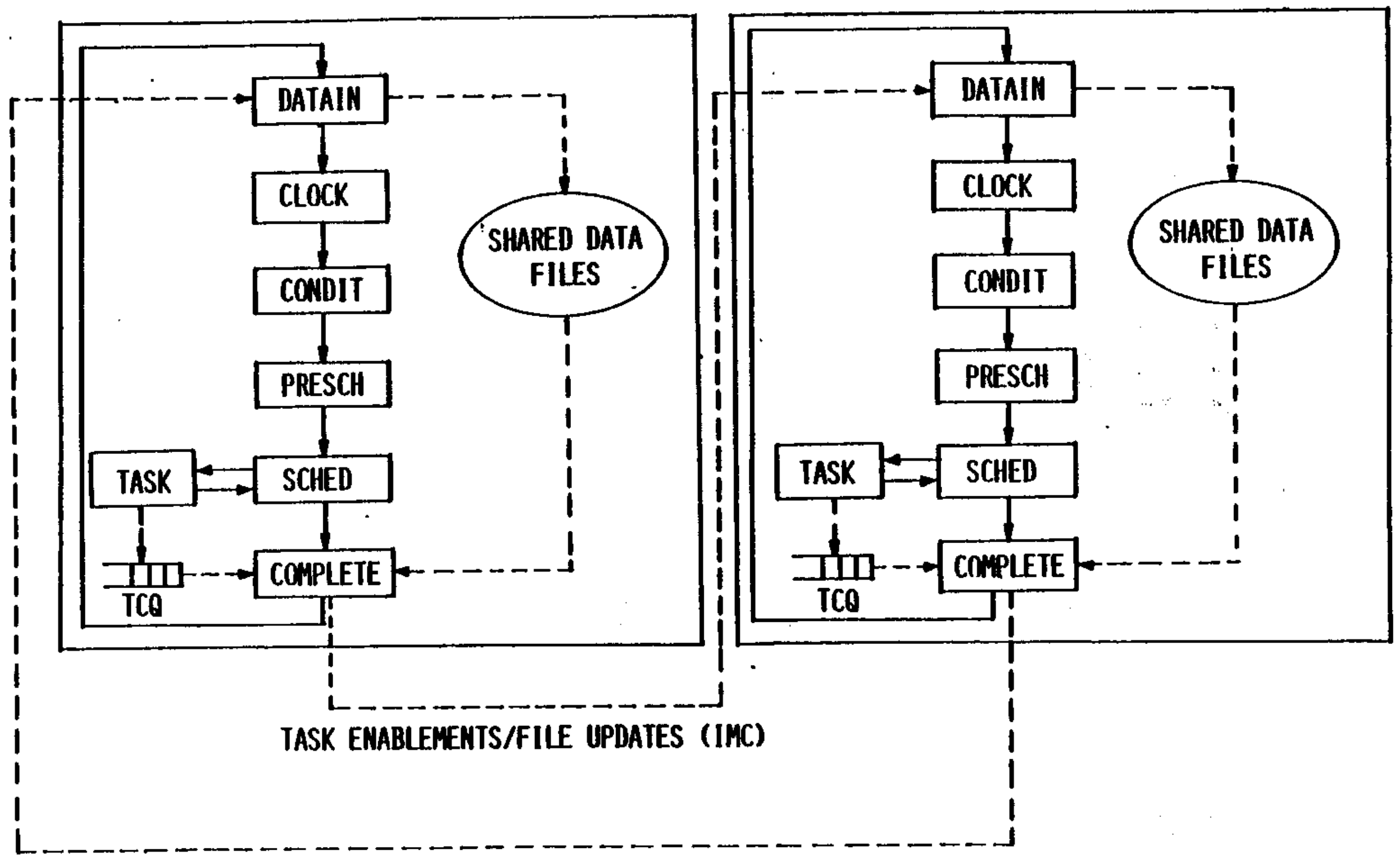


FIGURE A-3. FILE UPDATES IN THE DPAD

DATAIN reads the update message and posts the update. Similarly, when M_j enables M_n , M_j places an entry for M_n in the TCQ. The d-RTOS then sends an enablement message to the computer where M_n is assigned.

Two interconnection networks are considered in the DPAD: a bus and a totally connected point-to-point interconnection. In both cases, the networks are assumed to be error free and infinitely fast, i.e., the network delay is assumed negligible. The TRW DPAD simulates these networks by using the VAX/VMS mailbox communication facility.

The radar interface is simulated by the *System Environment and Threat Simulator (SETS)* which is driven by a predefined detailed *sky map*. Based on the sky map scenario, the SETS reports initial detections and updates the position of objects over time. A verified *radar return* is referred to as an *image*. Since an *object* changes its position, it may have several associated images. For each radar return, the SETS provides the tactical applications with the radar command type as well as the time at which the return was received.

The tactical application modules are driven by the radar. There are twenty-three application modules [1] which are employed in seven *processing threads* (see Figure A-4). For example, the SEARCH/VERIFY thread

[1] However, Modules M_{11}, M_{12}, and M_{15} are not currently implemented in the simulator.

Figure A-4a: DEFINITION OF PROCESSING THREADS

SEARCH/VERIFY.
1) Verifies a radar return
2) Control flow: See Figure A-4b

COARSE TRACKING.
1) Performs cross traffic rejection and known object recognition.
2) Control flow: 4, {5, 6}, 7, 13
({5,6} means that Modules 5 and 6 can be executed in parallel.)

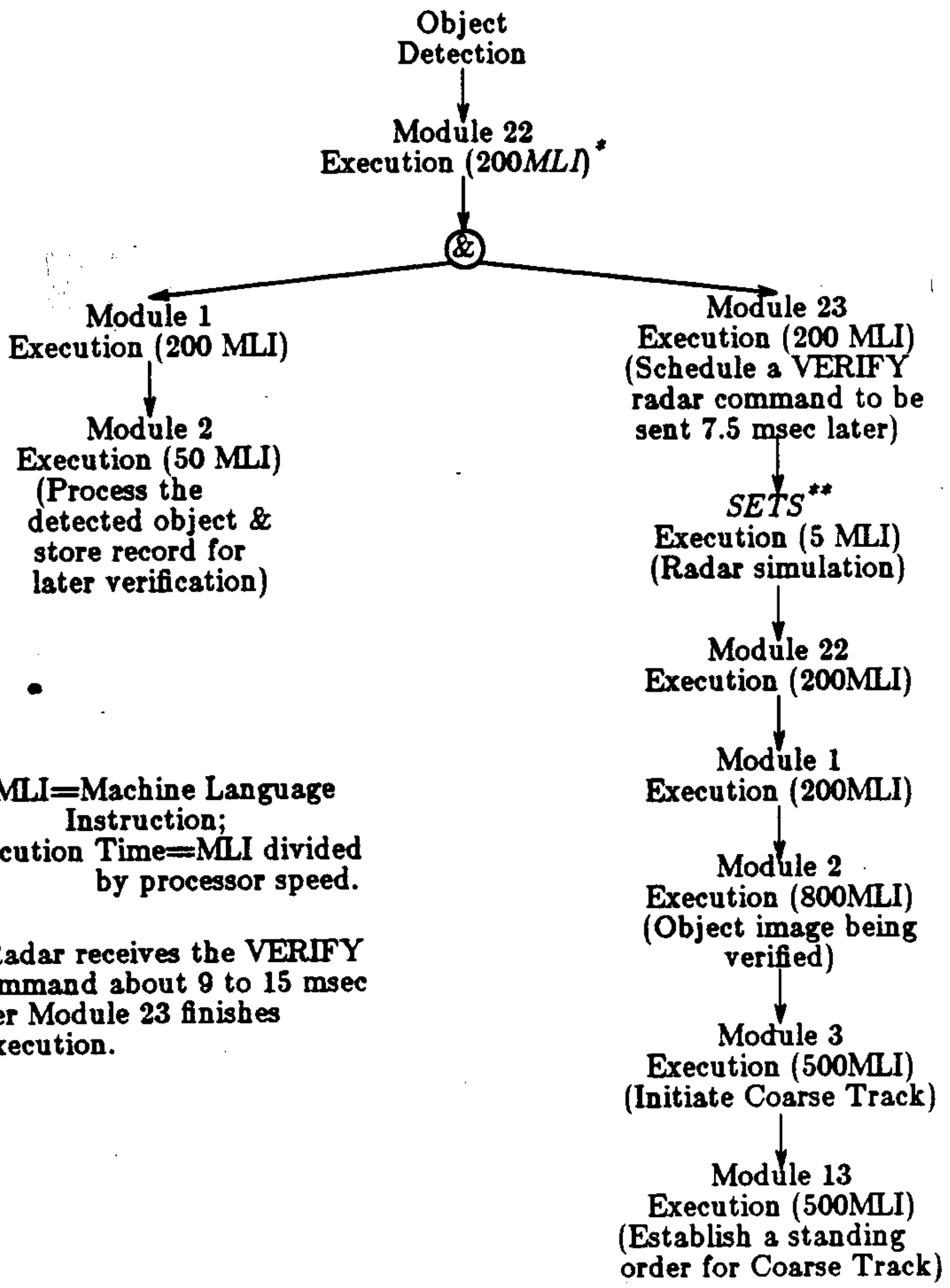
CANCEL COARSE TRACKING.
1) Cancels a standing order for coarse tracking.
2) Control flow: 4, 13,

PRECISION TRACKING.
1) Tracks objects to obtain an accurate estimate of their position.
2) Control flow: 8, 9, 13

REDUNDANT TRACK ELIMINATION.
1) Determines if an image matches an already known object.
2) Control flow: 8, 10

ELIMINATE NONTHREATENING OBJECTS.
1) Determines if an object is threatening.
2) Control flow: 8, {16, 17}, 18

INTERCEPT PLAN.
1) Establishes an intercept plan for a threatening object.
2) Control flow: 8, {16, 17}, 18, 19

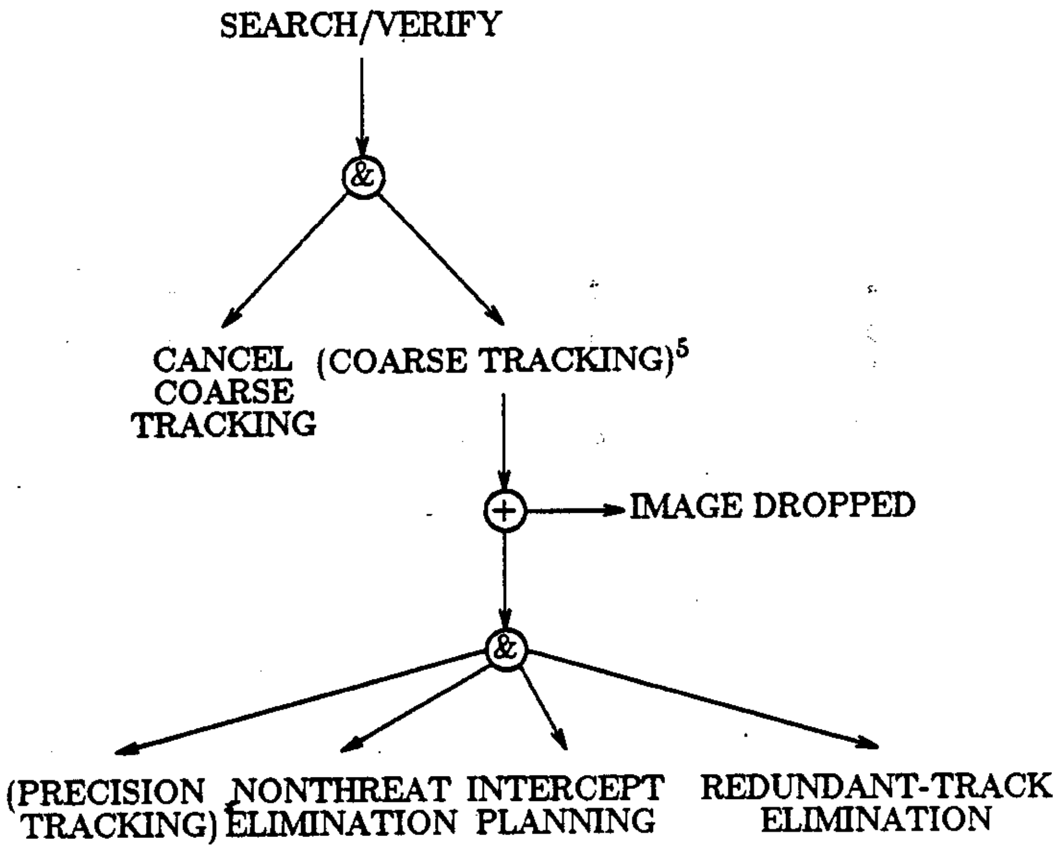

Object Detection
Module 22 Execution (200*MLI*)*
&
Module 1 Execution (200 MLI)
Module 2 Execution (50 MLI) (Process the detected object & store record for later verification)
Module 23 Execution (200 MLI) (Schedule a VERIFY radar command to be sent 7.5 msec later)
*SETS*** Execution (5 MLI) (Radar simulation)
Module 22 Execution (200MLI)
Module 1 Execution (200MLI)
Module 2 Execution (800MLI) (Object image being verified)
Module 3 Execution (500MLI) (Initiate Coarse Track)
Module 13 Execution (500MLI) (Establish a standing order for Coarse Track)

* MLI=Machine Language Instruction;
Execution Time=MLI divided by processor speed.

** Radar receives the VERIFY command about 9 to 15 msec after Module 23 finishes execution.

Figure A-4b. Search/Verify Parallel Activities processes an initial image detection, then schedules another radar pulse to verify the presence of an object. The time from beginning to end of a processing thread is its ***port-to-port (PTP) time.*** The processing threads are applied in a pipelined manner, as depicted in Figure A-5. Communication between modules is accomplished by use of shared files (see Figure A-6). All files in the DPAD are named with an integer number greater than 100. The overall control flow of the modules is shown in Figure A-7.

A portion of the control-and-data-flow was shown in Figure A-8. M_{13} consists of two separate routines: M_{13A} and M_{13B}. M_{13A} is periodically enabled every 1 ms to schedule radar commands: it enables M_{14} which in turn enables M_{23}. All other modules are enabled by the radar returns, directly or indirectly. Returns of different types (e.g. Detect, Verify, Coarse Track, or Precision Track) are processed by different threads of modules; M_1 does the thread selection.



(thread)^x = the thread is executed x times,
and the superscript * means 0 or more times

Figure A-5: Sequence of Threads

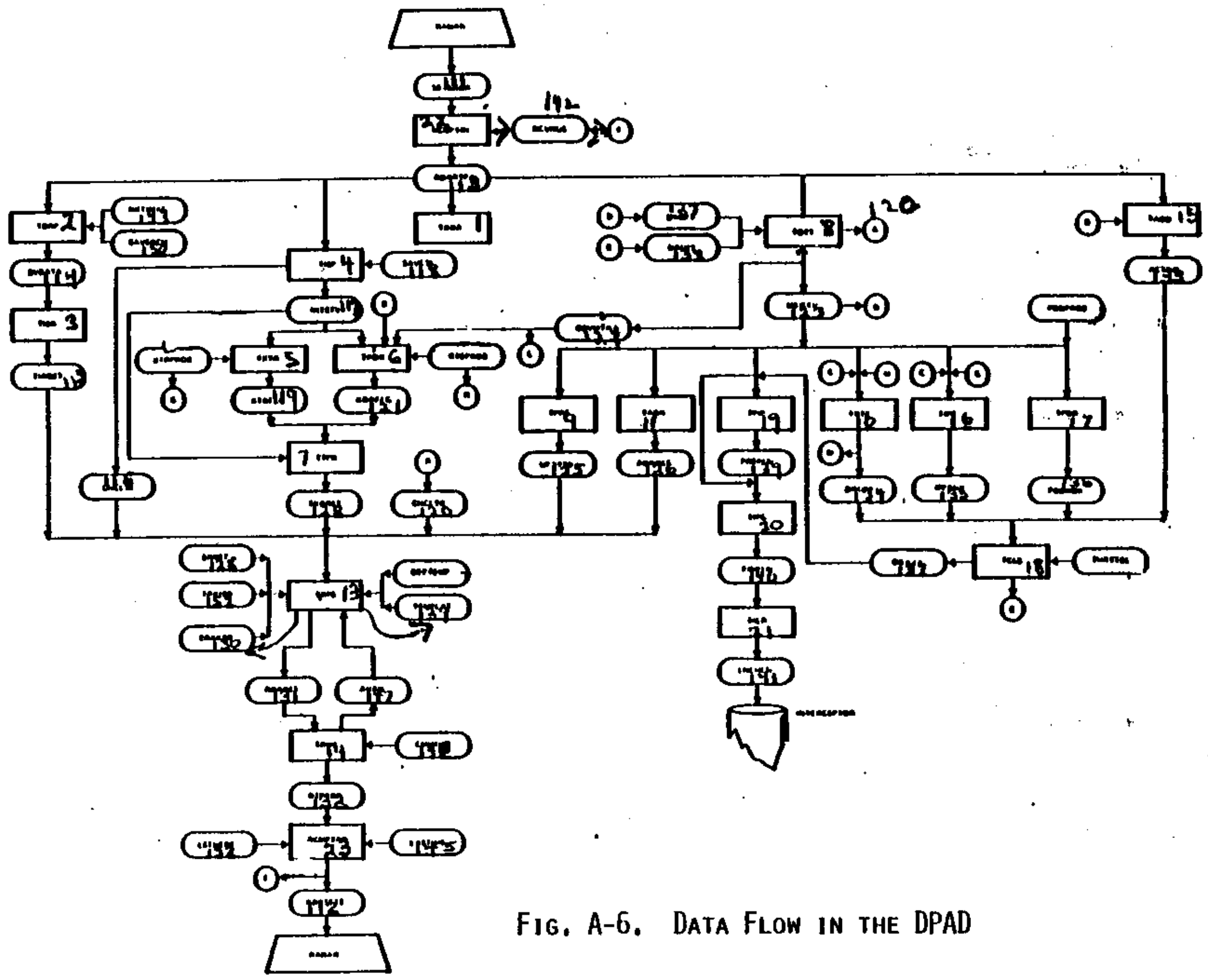


Fig. A-6. Data Flow in the DPAD

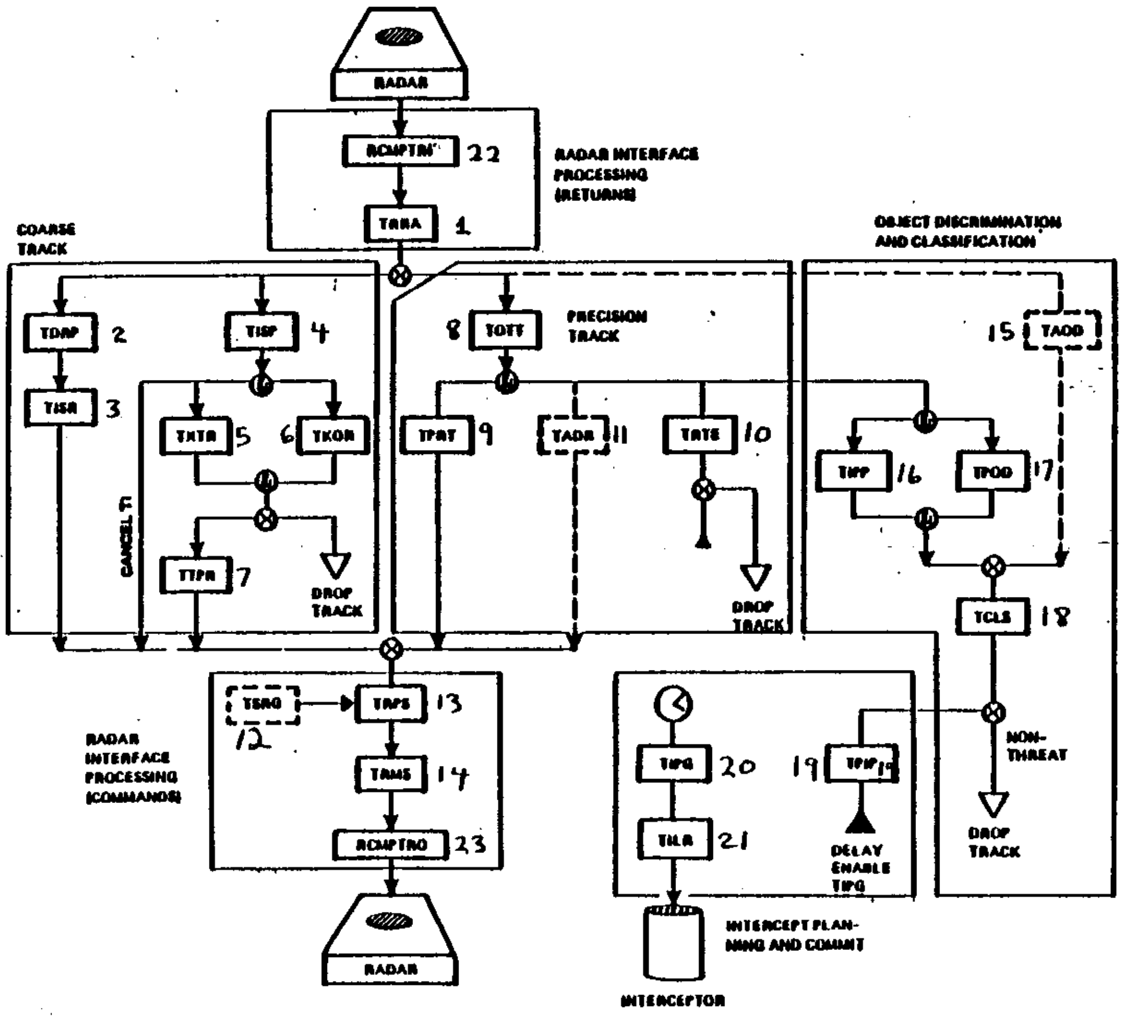


FIGURE A-7. CONTROL FLOW IN THE DPAD

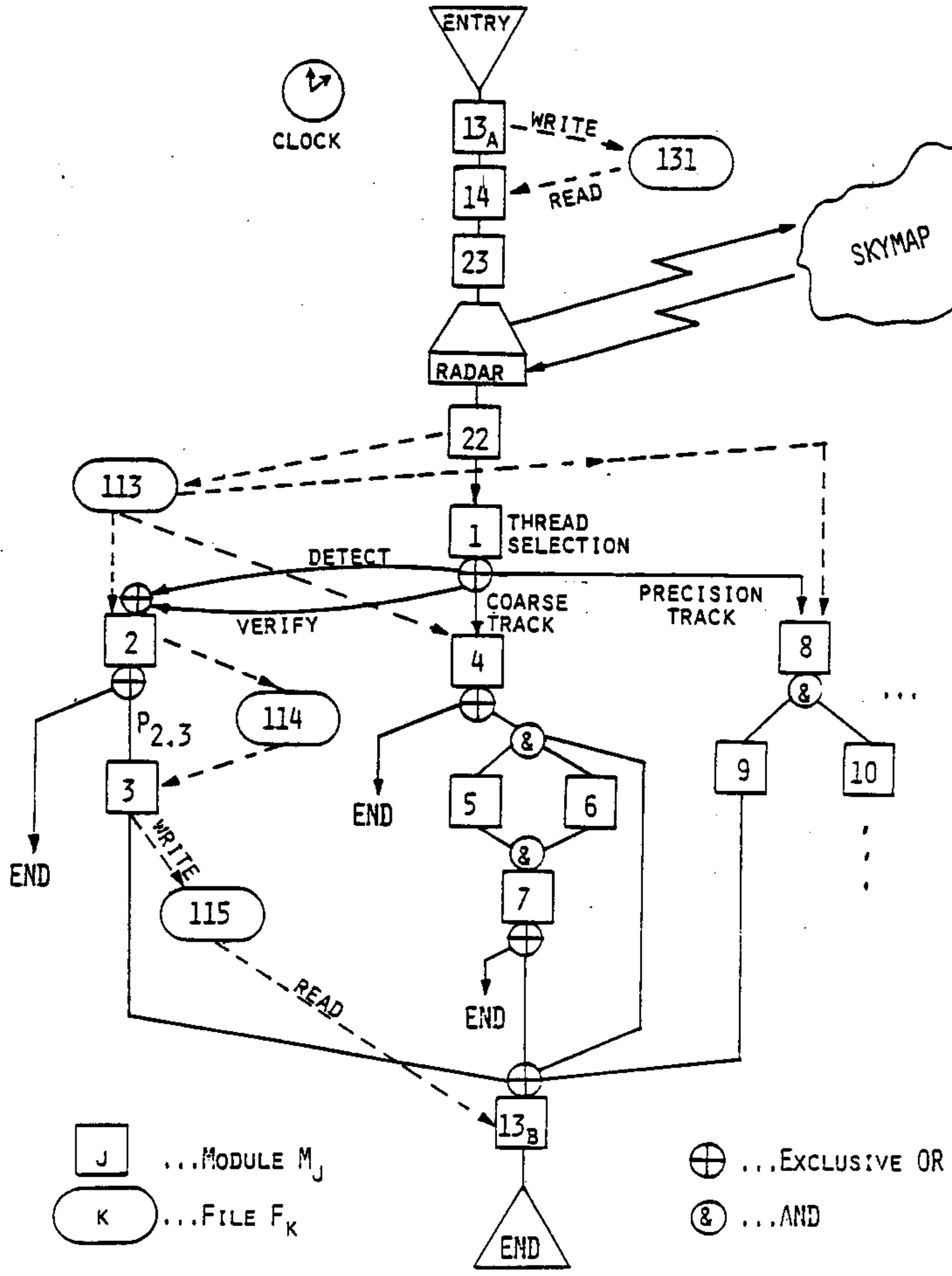


FIG. A-8. PART OF THE CONTROL-AND-DATA-FLOW GRAPH IN DPAD

APPENDIX B

THE UCLA DPAD SIMULATOR

The DPAD simulator at the University of California, Los Angeles (UCLA) is a modified version of the TRW DPAD simulator. In the TRW DPAD simulator, each tactical computer (also referred to as a CPU) in the target network is simulated by a VAX/VMS batch job. The tactical computers (the VMS batch jobs) communicate by using the VMS mailbox system function. The batch jobs must be synchronized so that the simulated CPUs keep roughly the same simulated tactical time. To this end, messages are sent between the batch jobs to synchronize the execution speed of the simulated computers. When a job suspends execution while waiting for a synchronization message, the VMS scheduler selects the next job to run. It can be one of the DPAD simulator jobs or another job unrelated to the simulator. If the next job is a simulator job, the d-RTOS for this job must ensure that its tactical time is the smallest among all the simulator jobs; otherwise, it goes into a *futile loop,* testing and waiting for all other simulator jobs to proceed to larger tactical times. Thus, if n computers are simulated, there will be n-1 batch jobs in a futile loop.

Futile loops made the time to do a simulation quite long and prevented us from making the large number of runs necessary to study the effect of varying the scenario, module assignment, and database kernel design. Thus, the TRW simulator was modified and moved to the UCLA VAX-11/780 (under UNIX). The UCLA version runs 3 to 10 times faster depending on the VAX load, number of objects simulated in the scenario, and number of computers simulated for the target system. [1] The following sections describe the process of converting the multi-batch-job simulator to a faster single-batch-job version, changing it from the VAX/VMS operating system to the VAX/UNIX at UCLA, and providing enhancement to the UCLA version.

B.1 CONVERSION TO SINGLE-BATCH-JOB SIMULATOR

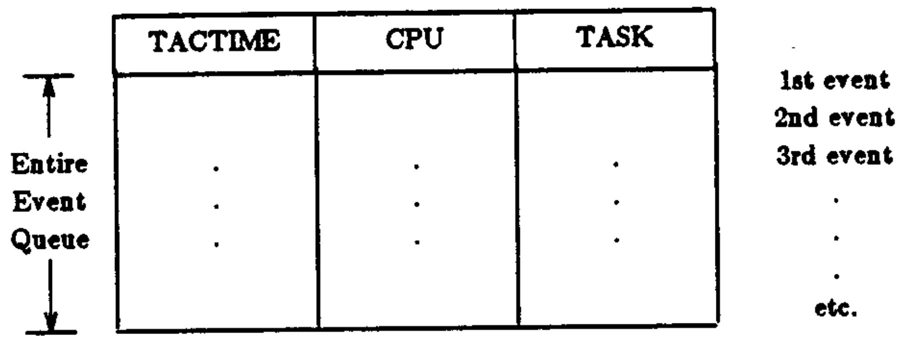
The popular event-scheduling simulation method [FISH73] was adopted to eliminate the futile-loop bottleneck. The modified simulator uses only one batch job no matter how many computers are simulated, (which also reduced the memory space required during a simulation). The techniques used in these modifications on data structures, event scheduling, message communication, and the generation of statistical reports are mentioned below.

a. *Data Structure:* In the TRW simulator, data for a batch job refer to a single computer. For instances, a simulated local tactical time is represented by TACTIME, and a module enablement queue by

[1] More objects or fewer CPUs usually means busy CPUs. And the busier the simulated CPUs, the smaller this speedup factor.

ENABQUE(10,40,4), which means that the queue has 10 priority levels, 40 entries per level, and 4 words per entry. To provide the same type of information for the different computers, the UCLA simulator dimensions most variables and arrays in FORTRAN COMMON statements by computer. For example, ENABQUE(10,40,4) becomes ENABQUE(10,40,4,20) and TACTIME becomes TACTIME(20). Variables and arrays appearing in FORTRAN executable statements are modified accordingly.

b. *Event Scheduling:* To simulate the parallel activities of multiple computers, the UCLA simulator maintains a clock, represented by the aforementioned TACTIME(CPU), for each CPU. Thus, it becomes easy to always schedule the simulated CPU with the smallest value of TACTIME(CPU). The UCLA simulator maintains an *event queue* as follows.



There are as many entries in the event queue as there are computers in the simulated system. Each entry tells when its corresponding computer is to be scheduled for its next activity. Events in the queue are ordered in the *ascending TACTIME* sequence; i.e., the first event has the smallest TACTIME. The simulator always schedules the first event to happen and deletes that event from the queue.

When the current scheduled event finishes all its activities, TACTIME(CPU) is advanced properly. Then, the simulator creates a new event and *inserts* it into the event queue according to the TACTIME sequence. The new event created is for the same CPU to execute the d-RTOS routine specified which is next to the one just finished in the cyclic order. This new event has the following contents:

TACTIME : the tactical time when the current event finishes; also the occurring time for the newly created future event.

CPU : same value as the current event's CPU.

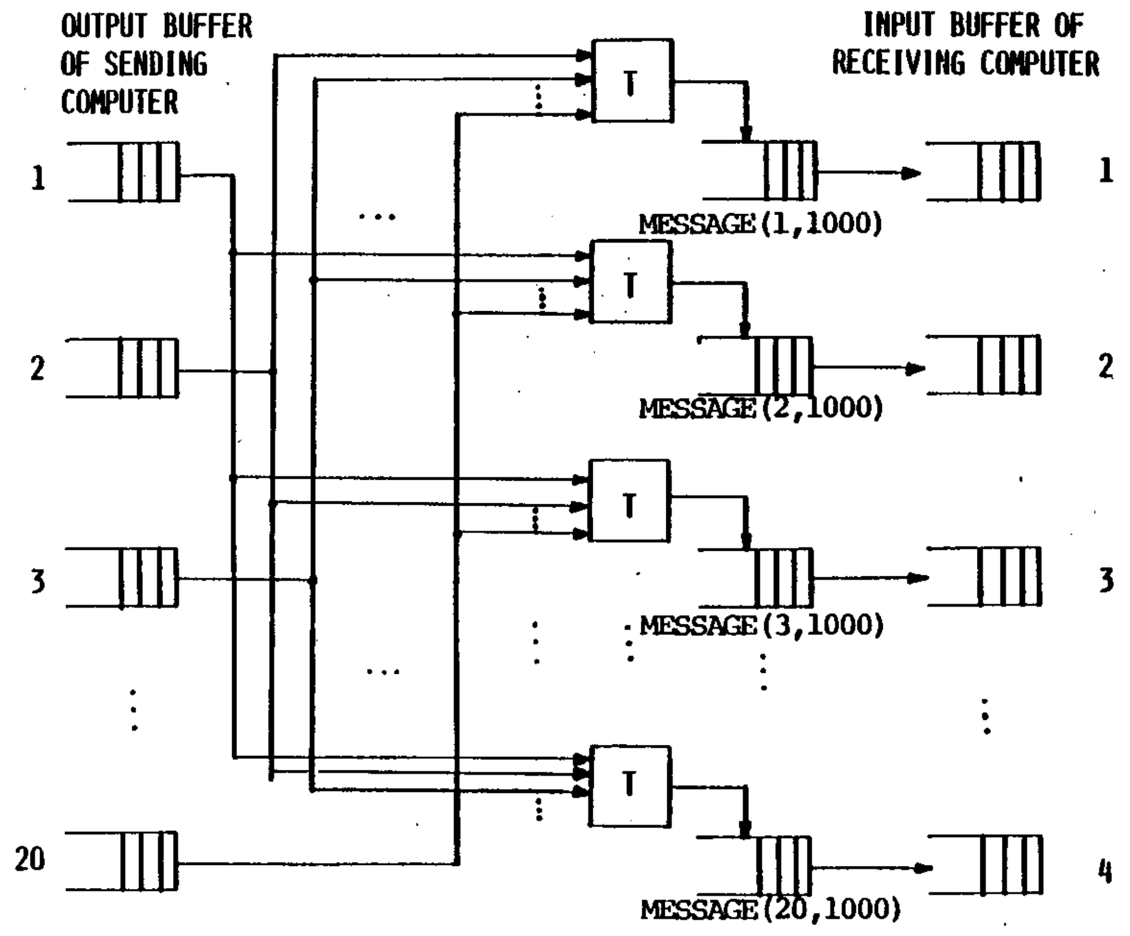
TASK : the next d-RTOS routine to be executed.

To conclude, the UCLA simulator uses only one job to simulate all computers. This greatly reduces the delays due to futile loops.

c. *Message Communication:* Computers can send "module enablement" command messages or shared-data update messages to another computer. The complicated (and thus time-consuming) mailbox function, used for intercomputer communication in the TRW simulator, has been replaced in the UCLA simulator by a simple procedure with 20 message collectors, MESSAGE(20,1000), each collecting messages sent to its associated computer from others. Fig. B-1 shows the role of MESSAGE collectors in the simulator. Note that input and output buffers are FIFO queues while the MESSAGE collectors are not. A message is *inserted* into the MESSAGE collector indicated by the message destination and it is inserted according to its arrival time.

Let us justify the existence of the MESSAGE collectors by an example. Assume CPU i starts an event E1 before CPU j starts E2. E2 will not be scheduled to happen until E1 has been *completely* simulated, although in the target system these two events overlap in time. (Remember that we have only one batch job.) Assume both E1 and E2 generate messages for CPU k at the end of their executions and assume E2 finishes earlier than E1 in the real world. But E2 will not generate messages in the simulation earlier than E1. We must compensate such a distortion and this is done by reordering messages in MESSAGE collector for CPU k, through message insertions according to their arrival times.

d. *Statistical Output Files:* Code was added to the d-RTOS operating system to extract statistical information such as CPU utilization in target computers (named as ZTG files in the simulation), shared-data update message volume from one application module to another (named as ZTU file), and number of times each module is executed



$\boxed{T}$: TIME COMPARATOR

FIGURE B-1. MESSAGE ORDERING MECHANISM IN THE SIMULATOR: MESSAGES ARRIVING AT A COMPUTER ARE REORDERED ACCORDING TO THEIR ARRIVING TIME (named as ZTO files). Two classes of statistical files can be distinguished. When we request for a file of the first class, a file should be generated for each CPU. Since there is a copy of d-RTOS in *each* batch job in the old-version simulator, multiple CPU utilization files were generated, each for one CPU. Because there is only one batch job in the new version, statistical records of a given type (e.g. CPU utilization) are written into the same opened FORTRAN file, with 2 digital characters added in front of each record to indicate the particular CPU this record applies to. After the simulation, the combined file is sorted off-line to recover the individual files for various CPU's.

Let us now consider files of the second class (e.g. ZTO and ZTU files). The old-version simulator generated a ZTO file from each batch job, each file supplying only part of the complete information requested. On the other hand, the new version naturally produces a single ZTO copy for the complete information requested.

B.2 SIMULATOR TRANSFER FROM VMS TO UNIX

Despite the potential portability of FORTRAN IV and the fact that both TRW/Redondo Beach and UCLA use the same VAX-11/780 hardware, problems arose because TRW uses FORTRAN-PLUS under VMS while UCLA employs F77 under UNIX. [1] Two major problems encountered were the number of characters allowed for a variable name and the number of file units allowed for a program.

a. *Variable Name:* FORTRAN-PLUS allows up to 14 characters in a variable name while UNIX only allows 6. Long variable names improve program readability and facilitate enhancements. It required much effort to rename variables while still preserving their uniqueness and their meanings.

b. *File Unit Number:* This problem was worse than the previous one in terms of the effort needed to correct it. UNIX allows only 20 logical file units (units 0 through 19) in a FORTRAN program. The TRW simulator generates more than 30 different statistical report files, each with its own unit number. The technique of "combine and then sort" is used to solve the problem. Frequently requested files are assigned their own file unit between 1 and 19. All other output files use unit 0. File unit numbers in the WRITE statements were changed to 0, and a code was added in front of each output record to indicate the original individual file. Sorting routines were written to sort the output file from unit 0 into separate files.

[1] FORTRAN-PLUS and F77 are different versions of the FORTRAN language.

B.3 ENHANCEMENTS TO SIMULATOR

We have made the following modifications to the simulator to improve its performance and functionality.

a. *Circular Queues:* There are 23 program modules in the TRW DPAD application model. Some files (represented as FORTRAN variable arrays) have as their sole function the information transfer from one program module to another; records of a file are generated by one and used by the other. In the original simulator, the receiving module reset the records to zero (or blank) and then distributes this update so that the same record spaces could be re-used. This technique creates unnecessary IMC. We have changed those modules which communicate in this manner to use *circular queues* instead. Much IMC was eliminated and CPU utilizations dropped in the range of 10 to 30 percents, which depends on the number of receiving modules residing on a particular CPU, frequency of receiving records, and record lengths.

b. *Functions for gathering IMC Statistics:* In order to study the role of inter-module communications in task allocation, we need to know how many messages are generated by what application modules, and to what modules these messages are sent. Special measurement routines have thus been inserted in the simulator to measure the IMC.

c. *Replication Runs:* In order to eliminate the bias resulting from the random number generator, we modified the simulator so that, for each run, it would start with a different initial seed for the generator. A UNIX command procedure was used to repeat the simulation a specified number of times, each time starting with a different seed.

d. *Tool for Data Reduction and Plotting:* To handle the huge amount of output statistical data from simulations, particularly from replicated runs as described above, several data reduction routines and plotter routines were developed.

APPENDIX C

COMPUTING OBJECT DETECTION TIMES

The input scenario is represented by the SKYMAP file (Fig. C-1). START TIME and END TIME are the time instances when an object enters and leaves the sky area covered by the indicated radar BEAM. The following 2 steps calculate the detection times for all objects in the scenario.

a) Perform the following algorithm for each BEAM-ID number:

```
1. Divide BEAM-ID by 15, obtaining the quotient q and the remainder r.

2. If r==0
      then { q <-- q-1;
             r <-- 15; }
   (This BEAM direction is searched during the r-th round of radar search.)

3. Calculate the time instance t in milliseconds
      t <-- 57(r-1)+(q+1)+7

     e.g.  BEAM-ID = 332
              332/15=22 ....2
              57x(2-1)+(22+1)+7 = 87 ms
4. If t < START-TIME for the object to show up within
   the search-range of this beam,
   then { t <-- t+855;
          go to 4; }
5. If t > END-TIME
   then terminate;
   Else
    { Record t as a Detection Time for this BEAM;
      t <-- t+855;
      go to 5; }
```

Note: After performing this algorithm for all the BEAM-ID numbers, we obtain the object detection times as shown in Fig. C-2.

b) List all Detection Times for each object (Fig. C-3).

THE SKYMAP FILE, TTSKYMAP.DAT, CONTAINS 42 OBJECTS

SEQUENCE NUMBER	OBJECT ID	BEAM ID	START TIME	END TIME
1	47	211	0	1000
1	47	212	800	2000
2	69	332	0	800
2	69	375	600	1600
3	56	482	0	400
3	56	528	300	2000
4	80	408	0	1000
5	67	603	0	500
5	67	646	400	1000
6	36	259	0	900
6	36	260	700	2000
7	35	350	0	500
7	35	351	400	1500
7	35	399	1400	2000
8	70	246	0	950
8	70	290	900	1600
8	70	288	1500	2000
9	64	291	0	800
9	64	335	700	1600
9	64	380	1500	2000
10	81	322	0	800
10	81	318	700	1500
11	45	396	0	400
11	45	442	300	1100
11	45	488	1000	2000
12	75	546	0	2000
13	59	205	0	2000
14	57	341	0	800
14	57	386	700	1600
14	57	431	1500	2000
15	58	255	0	800
15	58	300	600	2000
16	46	300	0	800
16	46	346	700	1600
16	46	347	1500	2000
17	65	207	0	1000
17	65	251	800	2000
18	77	329	0	800
18	77	325	700	1600
18	77	366	1500	2000
19	81	321	250	800
20	74	283	0	800
20	74	281	700	1600
20	74	278	1500	2000

Fig. C-1. SKYMAP, Input Scenario File

21	25	265	0	800
21	25	267	700	1600
21	25	270	1500	2000
22	70	291	500	950
23	82	232	0	1000
23	82	229	800	2000
24	78	190	0	400
24	78	189	300	900
24	78	187	800	2000
25	43	756	0	1000
26	61	745	0	1000
27	55	660	0	1100
28	71	591	0	500
28	71	635	400	1000
29	62	608	0	1000
30	33	583	0	800
31	72	552	0	800
32	44	533	0	2000
33	63	474	0	2000
34	68	463	0	2000
35	76	460	0	1500
36	34	447	0	1000
37	73	418	500	1500
37	73	459	1300	2500
38	24	358	500	1000
39	26	176	500	1500
40	37	173	500	2000
41	48	165	500	1500
42	22	130	500	1500
43	0	0	0	0

Fig . C-1 (Con.) SKYMAP, Input Scenario File

THE SKYMAP FILE, TTSKYMAP.DAT, CONTAINS 42 OBJECTS

SEQUENCE NUMBER	OBJECT ID	BEAM ID	START TIME	END TIME	DETECTION TIME
1	47	211	0	1000	22, 877,
1	47	212	800	2000	934,1789
2	69	332	0	800	87
2	69	375	600	1600	830
3	56	482	0	400	97
3	56	528	300	2000	1012,1867
4	80	408	0	1000	149
5	67	603	0	500	162
5	67	646	400	1000	906
6	36	259	0	900	196
6	36	260	700	2000	1108,1963
7	35	350	0	500	259
7	35	351	400	1500	1171
7	35	399	1400	2000	(Not Detected)
8	70	246	0	950	309
8	70	290	900	1600	1110
8	70	288	1500	2000	1851
9	64	291	0	800	312
9	64	335	700	1600	1113
9	64	380	1500	2000	1971
10	81	322	0	800	371
10	81	318	700	1500	998
11	45	396	0	400	319
11	45	442	300	1100	379
11	45	488	1000	2000	1294
12	75	546	0	2000	329, 1184
13	59	205	0	2000	534, 1389
14	57	341	0	800	600
14	57	386	700	1600	1458
14	57	431	1500	2000	(Not Detected)
15	58	255	0	800	(Not Detected)
15	58	300	600	2000	825, 1680
16	46	300	0	800	(Not Detected)
16	46	346	700	1600	886
16	46	347	1500	2000	1798
17	65	207	0	1000	648
17	65	251	800	2000	1449
18	77	329	0	800	770
18	77	325	700	1600	1397
18	77	366	1500	2000	(Not Detected)
19	81	321	250	800	314
20	74	283	0	800	710
20	74	281	700	1600	1451
20	74	278	1500	2000	(Not Detected)

Fig. C-2. Detection Times for Objects by Various BEAMs

21	25	665	0	800	538	
21	25	267	700	1600		1507
21	25	270	1500	2000		1678
22	70	291	500	950		(Not Detected)
23	82	232	0	1000	365	
23	82	229	800	2000		1049, 1904
24	78	190	0	400		(Not Detected)
24	78	189	300	900	476	
24	78	187	800	2000		1217
25	43	756	0	1000	343	
26	61	745	0	1000	570	
27	55	660	0	1100	849	
28	71	591	0	500	332	
28	71	635	400	1000		(Not Detected)
29	62	608	0	1000	447	
30	33	583	0	800	730	
31	72	552	0	800	671	
32	44	533	0	2000	442,	1297
33	63	474	0	2000	495,	1350
34	68	463	0	2000	722,	1577
35	76	460	0	1500	551,	1406
36	34	447	0	1000	664	
37	73	418	500	1500	719	
37	73	459	1300	2500		1349, 2204
38	24	358	500	1000	715	
39	26	176	500	1500	589,	1444
40	37	173	500	2000	1273	
41	48	165	500	1500	817	
42	22	130	500	1500	529,	1384
43	0	0	0	0		

Note: An object with more than one Detection Time means that it is detected multiple times by the radar and redundant images are generated.

Fig. C-2 (Con.) Detection Times for Objects by Various BEAMs

OBJECT NUMBER	DETECTION TIME	OBJECT NUMBER	DETECTION TIME
1	22, 877, 934, 1789	21	538, 1507, 1678
2	87, 830	22	(Not Detected)
3	97, 1012, 1867	23	365, 1049, 1904
4	149	24	476, 1217
5	162, 906	25	343
6	196, 1108, 1963	26	570
7	259, 1171	27	849
8	309, 1110, 1851	28	332
9	312, 1113, 1971	29	447
10	371, 998	30	730
11	319, 379, 1294	31	671
12	329, 1184	32	442, 1297
13	534, 1389	33	495, 1350
14	600, 1458	34	722, 1577
15	825, 1680	35	551, 1406
16	886, 1798	36	664
17	648, 1449	37	719, 1349, 2204
18	770, 1397	38	715
19	314	39	589, 1444
20	710, 1451	40	1273

Fig . C-3. Detection Times for Each Object

APPENDIX D

MODULE-ASSIGNMENT SELECTION PROGRAM

The program selects the module assignments from the enumeration tree according to the objective function. Variable MINMAX keeps the minimum "bottleneck load" evaluated so far in the tree search. It is initialized with a large value, 999999 MLI's. Whenever a module assignment is evaluated to have a bottleneck smaller than MINMAX, the bottleneck value replaces the old MINMAX value and a line is printed to log this particular assignment. Fig. D-1 shows the printout from this program. Progressively better assignments are obtained. The first column displays the module assignment with the minimum bottleneck found so far and the next column shows the associated bottleneck value. Of the 23 numbers shown in the assignment, the j-th number with a value of 1, 2, or 3 means module M_j being assigned to processor 1, 2, or 3 in the particular assignment. Within each row, the bottleneck is the largest of those three load values in columns 3, 4, and 5, each column representing a processor in the distributed system. The rightmost column shows the total load of the 3 processors, i.e., the total system load.

```
Jun 14 17:11 1983  enul.out Page 1
                  MODULE ASSIGNMENT                        BOTTLENECK =           L(1)    L(2)   L(3)   TOTAL
                                                           MAX( L(1),                                   LOAD
                                                           L(2), L(3) )
   1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 1 1 1 1  1 1 1 | minmax = 204846 | Load= 204846       0      0 | sum= 204846
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 1 1 1 1  1 1 2 | minmax = 187970 | Load= 187970   27398      0 | sum= 215368
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 1 1 1 1  1 2 1 | minmax = 178534 | Load= 178534   35982      0 | sum= 214516
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 1 1 1 1  1 2 2 | minmax = 161174 | Load= 161174   62896      0 | sum= 224070
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 1 1 1 1  2 2 2 | minmax = 161009 | Load= 161009   63185      0 | sum= 224194
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 1 1 2 1  1 2 2 | minmax = 153865 | Load= 153865   74997      0 | sum= 228862
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 1 1 2 1  2 2 2 | minmax = 153700 | Load= 153700   75286      0 | sum= 228986
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 1 1 2 2  1 2 2 | minmax = 151562 | Load= 151562   76714      0 | sum= 228276
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 1 1 2 2  2 2 2 | minmax = 151273 | Load= 151273   76879      0 | sum= 228152
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 1 2 2 1  1 2 2 | minmax = 150142 | Load= 150142   79224      0 | sum= 229366
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 1 2 2 1  2 2 2 | minmax = 149977 | Load= 149977   79513      0 | sum= 229490
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 1 2 2 2  1 2 2 | minmax = 147351 | Load= 147351   80453      0 | sum= 227804
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 1 2 2 2  2 2 2 | minmax = 147062 | Load= 147062   80618      0 | sum= 227680
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 2 1 2 2  1 2 2 | minmax = 145422 | Load= 145422   83054      0 | sum= 228476
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 2 1 2 2  2 2 2 | minmax = 145133 | Load= 145133   83219      0 | sum= 228352
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 2 2 2 1  1 2 2 | minmax = 143802 | Load= 143802   85364      0 | sum= 229166
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 2 2 2 1  2 2 2 | minmax = 143637 | Load= 143637   85653      0 | sum= 229290
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 2 2 2 2  1 2 2 | minmax = 141011 | Load= 141011   86593      0 | sum= 227604
at 1 1 1 1 1  1 1 1 1 1  0 0 1 1 0  1 2 2 2 2  2 2 2 | minmax = 140722 | Load= 140722   86758      0 | sum= 227480
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 1 1 1 1  1 2 2 | minmax = 115181 | Load= 115181  101233      0 | sum= 216414
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 1 1 1 1  2 2 2 | minmax = 115016 | Load= 115016  101522      0 | sum= 216538
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 1 1 2 1  1 2 2 | minmax = 113334 | Load= 107872  113334      0 | sum= 221206
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 1 1 2 1  1 2 3 | minmax = 107872 | Load= 107872   96458  27398 | sum= 231728
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 1 1 2 1  2 2 3 | minmax = 107707 | Load= 107707   96747  27398 | sum= 231852
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 1 1 2 2  1 2 3 | minmax = 105569 | Load= 105569   98175  27398 | sum= 231142
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 1 1 2 2  2 2 3 | minmax = 105280 | Load= 105280   98340  27398 | sum= 231018
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 1 2 2 1  1 2 3 | minmax = 104149 | Load= 104149  100685  27398 | sum= 232232
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 1 2 2 1  2 2 3 | minmax = 103984 | Load= 103984  100974  27398 | sum= 232356
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 1 2 2 2  1 2 3 | minmax = 101914 | Load= 101358  101914  27398 | sum= 230670
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 1 2 2 2  1 3 2 | minmax = 101358 | Load= 101358   83292  35982 | sum= 220632
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 1 2 2 2  2 3 2 | minmax = 101069 | Load= 101069   83457  35982 | sum= 220508
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 2 1 2 2  1 3 2 | minmax =  99429 | Load=  99429   85893  35982 | sum= 221304
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 2 1 2 2  2 3 2 | minmax =  99140 | Load=  99140   86058  35982 | sum= 221180
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 2 2 2 1  1 3 2 | minmax =  97809 | Load=  97809   88203  35982 | sum= 221994
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 2 2 2 1  2 3 2 | minmax =  97644 | Load=  97644   88492  35982 | sum= 222118
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 2 2 2 2  1 3 2 | minmax =  95018 | Load=  95018   89432  35982 | sum= 220432
at 1 1 1 1 1  1 1 1 1 1  0 0 2 2 0  1 2 2 2 2  2 3 2 | minmax =  94729 | Load=  94729   89597  35982 | sum= 220308
at 1 1 1 1 1  1 1 1 1 2  0 0 2 2 0  2 2 2 2 2  1 3 3 | minmax =  93775 | Load=  93775   85889  62896 | sum= 242560
at 1 1 1 1 1  1 1 1 1 2  0 0 2 2 0  2 2 2 2 2  2 3 3 | minmax =  93486 | Load=  93486   86054  62896 | sum= 242436
at 1 1 1 1 1  1 1 1 2 1  0 0 2 2 0  1 1 1 2 1  1 3 3 | minmax =  88466 | Load=  88466   78270  62896 | sum= 229632
at 1 1 1 1 1  1 1 1 2 1  0 0 2 2 0  1 1 1 2 1  2 3 3 | minmax =  88301 | Load=  88301   78559  62896 | sum= 229756
at 1 1 1 1 1  1 1 1 2 1  0 0 2 2 0  1 1 1 2 2  1 3 3 | minmax =  86163 | Load=  86163   79987  62896 | sum= 229046
at 1 1 1 1 1  1 1 1 2 1  0 0 2 2 0  1 1 1 2 2  2 3 3 | minmax =  85874 | Load=  85874   80152  62896 | sum= 228922
at 1 1 1 1 1  1 1 1 2 1  0 0 2 2 0  1 1 2 2 1  1 3 3 | minmax =  84743 | Load=  84743   82497  62896 | sum= 230136
at 1 1 1 1 1  1 1 1 2 1  0 0 2 2 0  1 1 2 2 1  2 3 3 | minmax =  84578 | Load=  84578   82786  62896 | sum= 230260
at 1 1 1 1 1  1 1 1 2 1  0 0 2 2 0  1 1 2 2 2  1 3 3 | minmax =  83726 | Load=  81952   83726  62896 | sum= 228574
at 1 1 1 1 1  1 1 1 2 1  0 0 2 2 0  1 1 2 2 3  1 3 3 | minmax =  83520 | Load=  83520   82497  67379 | sum= 233396
at 1 1 1 1 1  1 1 1 2 1  0 0 2 2 0  1 1 2 2 3  2 3 3 | minmax =  83231 | Load=  83231   82786  67379 | sum= 233396
at 1 1 1 1 1  1 1 1 2 1  0 0 2 2 0  1 1 3 2 2  1 3 3 | minmax =  81952 | Load=  81952   79987  67581 | sum= 229520
at 1 1 1 1 1  1 1 1 2 1  0 0 2 2 0  1 1 3 2 2  2 3 3 | minmax =  81663 | Load=  81663   80152  67581 | sum= 229396
```

FIG. D-1. ENUMERATION RESULTS FROM EXHAUSTIVE SEARCH

```
Jun 14 17:11 1983  emul.out Page 2

at 1 1 1 1 1   1 1 1 2 1   0 0 2 2 0   1 2 1 3 2   1 3 3 | minmax =  81591 | Load=  81591  76992  74997 | sum= 233580
at 1 1 1 1 1   1 1 1 2 1   0 0 2 2 0   1 2 1 3 2   2 3 3 | minmax =  81302 | Load=  81302  77157  74997 | sum= 233456
at 1 1 1 1 1   1 1 1 2 1   0 0 2 2 0   1 2 2 3 1   1 3 3 | minmax =  79971 | Load=  79971  78562  74997 | sum= 233530
at 1 1 1 1 1   1 1 1 2 1   0 0 2 2 0   1 2 2 3 1   2 3 3 | minmax =  79806 | Load=  79806  78851  74997 | sum= 233654
at 1 1 1 1 1   1 1 1 2 1   0 0 2 2 0   1 2 2 3 3   1 3 3 | minmax =  78562 | Load=  77180  78562  76714 | sum= 232456
at 1 1 1 1 1   1 1 1 2 1   0 0 2 2 0   1 3 3 2 3   1 3 3 | minmax =  78270 | Load=  77180  78270  77716 | sum= 233166
at 1 1 1 1 1   1 1 1 2 1   0 0 3 3 0   1 2 2 2 1   2 2 3 | minmax =  78238 | Load=  78238  78145  65735 | sum= 222118
at 1 1 1 1 1   1 1 1 2 1   0 0 3 3 0   1 2 2 2 3   1 2 3 | minmax =  77856 | Load=  77180  77856  70218 | sum= 225254
at 1 1 1 1 1   1 1 1 2 1   0 0 3 3 0   1 2 2 3 2   1 2 3 | minmax =  77836 | Load=  77180  70208  77836 | sum= 225224
at 1 1 1 1 1   1 1 1 2 1   0 0 3 3 0   1 2 3 2 2   1 2 3 | minmax =  75612 | Load=  75612  75546  70420 | sum= 221578
at 1 1 1 1 1   1 1 1 2 1   0 0 3 3 0   1 2 3 2 2   3 2 3 | minmax =  75546 | Load=  75323  75546  70709 | sum= 221578
at 1 1 1 1 2   1 1 1 2 1   0 0 3 3 0   1 3 2 2 2   2 2 3 | minmax =  75413 | Load=  75413  75352  73643 | sum= 224408
at 1 1 1 1 2   1 2 1 3 1   0 0 2 2 0   1 3 2 3 1   1 3 2 | minmax =  75178 | Load=  75178  74275  73829 | sum= 223282
at 1 1 1 1 2   1 2 1 3 1   0 0 2 2 0   1 3 2 3 1   2 3 2 | minmax =  75013 | Load=  75013  74564  73829 | sum= 223406
at 1 1 1 1 2   3 2 3 3 3   0 0 2 2 0   3 3 2 1 1   1 1 2 | minmax =  74414 | Load=  74414  74275  74023 | sum= 222712
at 1 1 1 1 2   3 2 3 3 3   0 0 2 2 0   3 3 2 1 1   3 1 2 | minmax =  74312 | Load=  74249  74275  74312 | sum= 222836
at 1 2 1 2 3   2 3 2 1 2   0 0 3 3 0   2 1 3 2 2   1 1 3 | minmax =  74308 | Load=  74308  73873  74275 | sum= 222456
at 1 2 1 2 3   2 3 2 1 2   0 0 3 3 0   2 1 3 2 2   2 1 3 | minmax =  74275 | Load=  74019  74038  74275 | sum= 222332
at 1 2 2 1 3   1 3 1 2 1   0 0 3 3 0   1 1 3 2 2   2 2 3 | minmax =  74275 | Load=  74004  73805  74275 | sum= 222084
```

Fig. D-1. (Continued)

Since it takes several days to enumerate the entire tree, the program is designed to have a checkpoint written out in a temporary output file for every 3^{11} assignments evaluated. When a computer system failure occurs, the program can continue the enumeration from the most recent checkpoint.

From the program output we picked the last ten assignments (shown in Fig.4-20) and simulated each of them with the DPAD simulator. See Section 4.3 for the results.

Although the 10 selected assignments have progressively smaller bottlenecks, they are *not* exactly the 10 assignments in the search tree which have the 10 *smallest* bottlenecks. For example, after the assignment with the bottleneck 74308 MLI was printed on Fig. D-1 (the 3rd line from the bottom), an assignment with a bottleneck 74310 was not printed. Nor was an assignment with a bottleneck 74400. But, both 74310 and 74400 might well be among the 10 smallest bottlenecks. Therefore, there are many more assignments that have better (or comparable) performance than the 10 selected ones.