NASA Contractor Report 187451

KNOWLEDGE REPRESENTATION INTO ADA PARALLEL PROCESSING

Tom Masotto
Carol Babikyan
Richard Harper

THE CHARLES STARK DRAPER LABORATORY, INC.
Cambridge, Massachusetts 02139

Contract NAS1-18565
July 1990

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

NASA Contractor Report 187451

KNOWLEDGE REPRESENTATION INTO ADA PARALLEL PROCESSING

Tom Masotto
Carol Babikyan
Richard Harper

THE CHARLES STARK DRAPER LABORATORY, INC.
Cambridge, Massachusetts 02139

Contract NAS1-18565
July 1990

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

TABLE OF CONTENTS

1.0 Introduction

The goal of the Knowledge Representation into Ada Parallel Processing project (KRAPP) is to host and execute an intelligent system on multiple processors of the Fault Tolerant Parallel Processor (FTPP) [Har87]. The methodology that permits the parallelized execution of an intelligent system was developed by Worcester Polytechnic Institute (WPI). It is based on the use of an intelligent scheduling mechanism called an Activation Framework (AF) ([Gre87], [Gre89]). Additionally, it utilizes a suite of Activation Framework Objects (AFOs) to model the intelligence of the desired application. The candidate architecture used in the KRAPP project was the FTPP. It was developed by Charles Stark Draper Laboratory (CSDL) and is capable of providing high throughput while offering extremely high reliability. The intention of the KRAPP project was to use the FTPP to demonstrate that the AF parallelization methodology is feasible, to quantify the gains attainable by parallelizing a candidate application, and to evaluate the performance of the AF and AFOs.

The introductory Sections 1.1 through 1.4 discuss the Activation Framework methodology, the two candidate applications, and the Fault Tolerant Parallel Processor respectively. Section 2 discusses the utilities that were developed and the modifications performed to permit the execution of the AF and the Event Diagnosis Expert on the FTPP. Each utility is outlined and described separately. Section 3 gives an overall view of the CSDL AF-FTPP development system, discussing how each utility outlined in Section 2 fits in and contributes to the system. Section 4 presents the performance metrics attained during the KRAPP analysis, while Section 5 suggests areas for performance improvements. Finally, Section 6 concludes the document with a summary of the primary results and an outline of suggested future work.

1.1 The Activation Framework

As previously mentioned, the Activation Framework methodology was developed by Worcester Polytechnic Institute. The method uses a set of Activation Framework Objects to represent the intelligence of the designated application. Because the AF was developed by WPI and is not CSDL's expertise, only a brief overview of the methodology is presented in this document.

The AF is responsible for the initializing and scheduling the AFOs. During the initialization process, the AF creates a list element for each AFO in the application. This AFO structure is allocated by the AF to record the attributes associated with the AFO, such as its name and importance (priority). The AF initialization process also allocates a list element for each AFO input port. Similarly, this list element is used to store the port's attributes. Further, it records the presence of pending messages and their respective locations. To maintain these lists and the other linked lists inherent in the AF methodology, WPI developed a List Management System (LMS). This LMS provides the low-level functions necessary to create a list header, to allocate or find a list element, and to read or write a data object.

When the AF determines the next AFO to schedule, it considers two criteria: the AFOs' importance and the AFOs' priming conditions. The importance of an AFO is essentially its priority. It is based on the number of pending input messages, is calculated using an application specific (or possibly AFO specific) importance function, and is dynamic. An AFO's priming conditions are the prerequisites that must be met before the AFO can be scheduled, or fired. These conditions are application dependent, and typically they vary from AFO to AFO. The fulfillment of an AFO's priming prerequisites depends on the presence of messages at specific input ports. If any of an AFO's priming conditions has been attained, then it is considered primed. Once primed, an AFO will be scheduled if it has the highest importance of the primed AFOs. In other words, the AF determines which AFOs are primed and, of this subset, executes the one with the highest importance.

A scheduled AFO executes until it is either preempted or finished. An AFO is preempted if a primed AFO exists that has a higher importance. For example, preemption situations can arise when messages are communicated between AFOs. If the executing AFO sends a message, then the associated destination AFO may become primed or its importance may increase. If the message causes the destination AFO to be primed and this AFO's

importance is greater than the executing AFO, then the AFO currently executing is suspended and the other AFO is scheduled.

Preemption situations can also occur when an AFO removes a message from an input port. When a message is retrieved from a port, then importance of the corresponding AFO typically changes. If the message removal causes the executing AFO's importance to decrease and another primed AFO exists that has a higher importance, then the executing AFO is preempted and the other scheduled.

In addition to initializing and scheduling the AFOs, the AF supports the inter-AFO communication. The AF also allows the AFOs to detect the presence of pending messages and to retrieve them from their input ports.

As discussed earlier, the AFOs and their associated attributes (priming conditions, global importance, interconnections, etc.) characterize the intelligence of the application. Representing an intelligent system as a parallel process, however, is a difficult task. An attractive feature of the AF methodology is that these AFOs are automatically generated. Specifically, the AFOs are created by a Rules to AFO Translator. This Translator parses and interprets a set of rules (Horn clauses) which model the application and generates a corresponding suite of AFOs. After their creation, the AFOs can be integrated with the AF and executed. Further, unlike the AFOs which vary from application to application, the AF usually does not have to be changed.

The execution of the AF methodology on a parallel processor is illustrated in Figure 1. Each processor hosts an image of the AF and is responsible for executing a subset of the application's AFOs. The *intra*-processor message communication is supported by the AF. Alternatively, the *inter*-processor data transfer is completed by the AF and an operating system message passing mechanism.

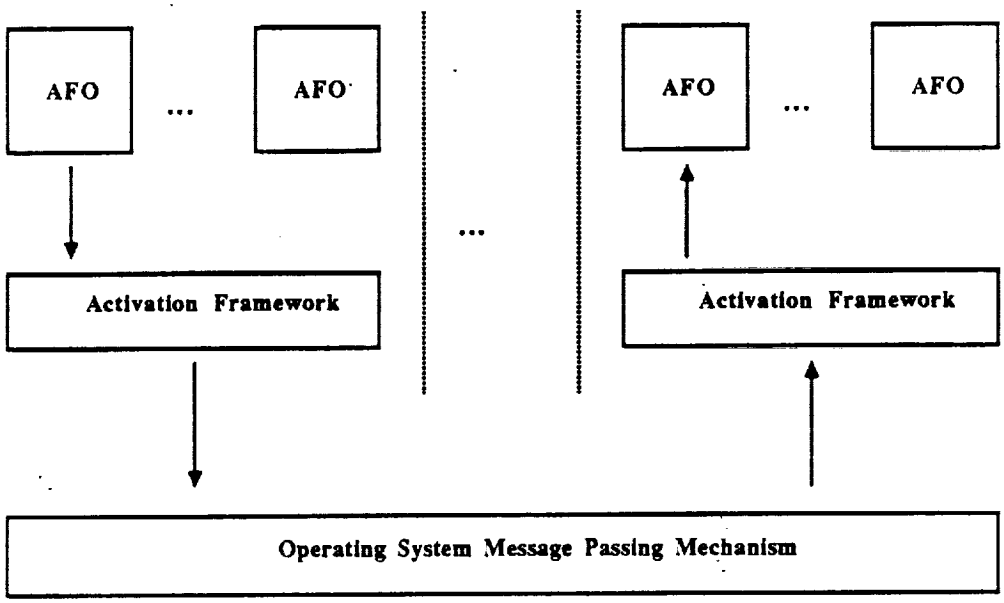

Figure 1 - The Activation Framework and Activation Framework Objects

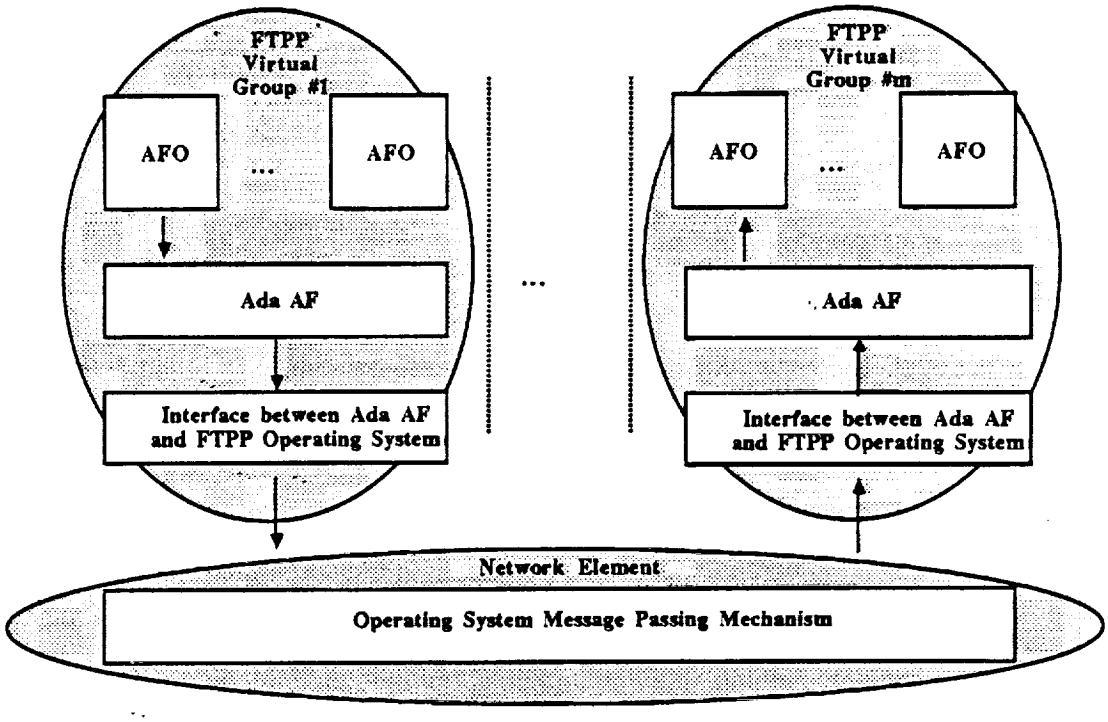

Figure 2 - The AF and AFOs on the FTPP

The implementation of the AF methodology on the FTPP is shown in Figure 2. An FTPP load module is created, and it is downloaded to and executed on each Virtual Group (designated by a Virtual Group Identifier or VID). This load module contains the AF, the suite of AFOs, the FTPP Operating System, and the AF-FTPP Interface. The AF-FTPP Interface, which was developed by Charles Stark Draper Laboratory under the KRAPP project, permits the integration of the AF and AFOs with the FTPP Operating System. In addition, it allows the scheduling of the AFOs as well as redundancy management tasks, inter-VID message communication, the observation of message traffic, and the measurement of the performance of the AF.

Although all VIDs host the entire AFO suite, each VID is only responsible for executing a subset of them. Further, like the *intra*-processor communication previously discussed, the *intra*-VID message transfer is completed by the AF. The *inter*-VID data communication is supported by the AF, AF-FTPP Interface, FTPP Operating System, and FTPP Network Element (NE). The AF-FTPP Interface, FTPP Operating System, and the NE emulate the operating system message passing mechanism.

1.2 The Candidate Application

Advanced military aircraft will host a wide variety of sophisticated avionics subsystems for Terrain Following/Terrain Avoidance, threat avoidance, all-weather and night operations, mission planning and optimization, and weapons delivery. The real-time management of and assimilation of data from these complex functions is expected to seriously overburden an already cognitively stressed aircrew. Consequently, a computational system is needed which will integrate and assess the vast quantity of information emanating from a multisensor navigation suite to produce a meaningful yet compressed set of navigation status and data for presentation to the aircrew.

It is anticipated that these functions will be computationally intensive and in some cases mission-critical. Therefore it is of interest to demonstrate the feasibility of transitioning knowledge-based representations of these intelligent navigational systems to a highly reliable, high-throughput processing system. A successful demonstration of this would result in a technology which can be used to facilitate the development of robust, high-throughput artificial intelligence systems which will be required for programs such as the Pilot's Associate or Space Station Freedom applications. It is the objective of the KRAPP program to develop and demonstrate this technology.

The application selected for the KRAPP program is the Adaptive Tactical Navigator (ATN) ([Ber1], [Ber2], [Gre87]). The purpose of the ATN is to supplant many of the functions of the navigator in the next generation of Air Force attack aircraft. The ATN combines artificial intelligence techniques, knowledge-based systems, and advanced navigation algorithms. With the successful completion of the KRAPP program, it will also have available automated translation of high-level navigation system knowledge, expressed as Horn clause rules, to parallelized Ada code which exhibits computational speedup on a high-throughput high-reliability parallel processor.

The ATN is succinctly described in [Ber2]:

> "The Adaptive Tactical Navigator (ATN) [Ach87] is an intelligent onboard system which utilizes expert navigation sensors and their integration algorithms to provide equipment management and pilot decision-aiding for the multisensor navigation suite of future tactical aircraft. A navigation system has been defined which is representative of the technology which is being developed for operational aircraft in the mid-1990s. This navigation system, which the ATN was designed to manage, includes the following: a strapdown inertial navigation system (INS), Global Positioning System (GPS), Synthetic Aperture Radar (SAR), Doppler Radar, an Electro-Optical system (EO), Sandia Inertial Terrain-Aided Navigation System (SITAN), and a digital moving map display."[Ber2]

The ATN is functionally organized into six expert systems (see Figure 3). The first three manage the equipment suite and the second three perform decision-aiding functions. The descriptions of these functions given below are paraphrased from [Ber1].

Equipment Management Functions:

> The Navigation Source Manager experts use engineering design models to monitor equipment performance and to detect and isolate equipment failures or degradations.
>
> The System Status expert diagnoses system health based on reliability data, recent maintenance patterns, current mission environment, and lower-level diagnoses.

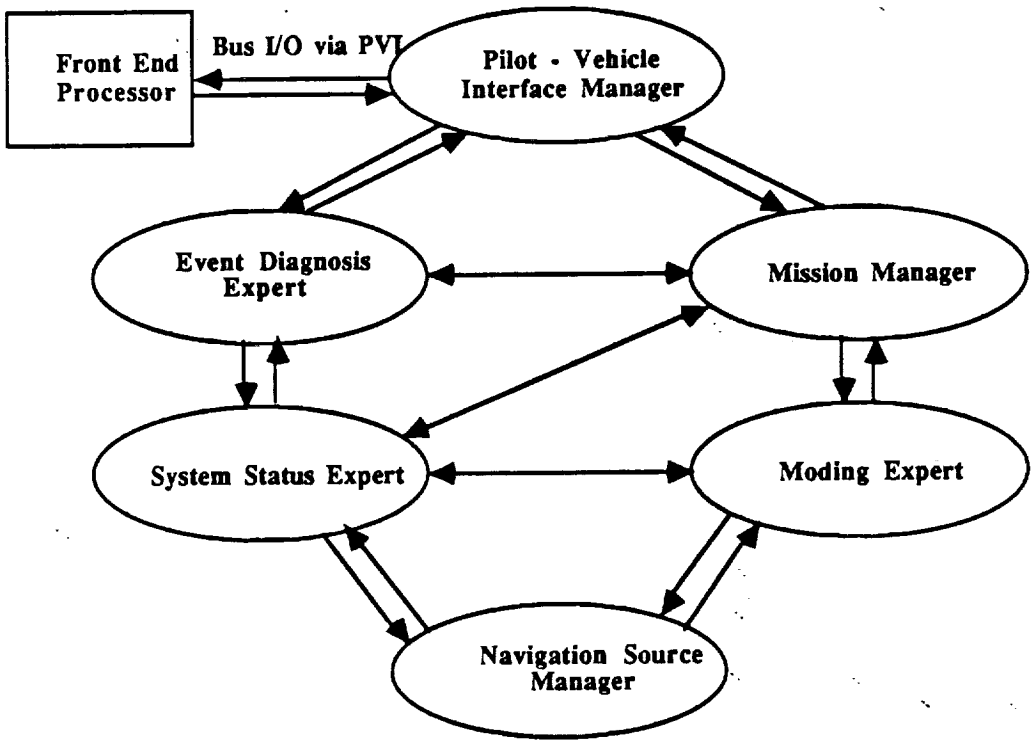

Figure 3 - The Adaptive Tactical Navigator

The Moding expert configures viable component combinations based on current equipment status and determines appropriate handoff strategies for mode changes.

Decision-Aiding Functions:

The Event Diagnosis expert evaluates planned navigation events, such as waypoint encounters and destinations, and diagnoses anomalous or out-of-spec events to support pilot moding decisions.

The Mission Management expert stores mission plan and environment data and determines which available equipment configurations are appropriate for the current and forecast mission situation.

The Pilot-Vehicle Interface Management expert manages communication between the ATN and the pilot in a manner appropriate for the current mission phase.

The KRAPP program uses a portion of the Event Diagnosis expert of the ATN as its demonstration application.

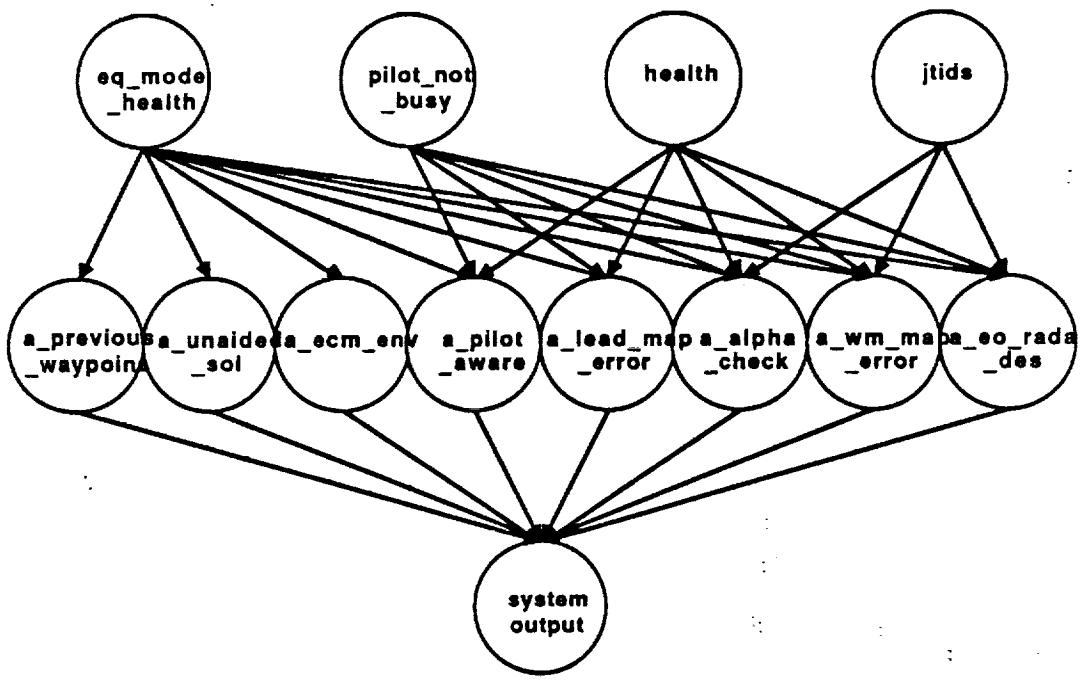

Figure 4 - The Event Diagnosis Expert

The subset of Event Diagnosis Expert employed by KRAPP was characterized using eight Horn clauses, or rules. An AFO was used to represent each rule, and five AFOs were used for the input and output facilities. Accordingly, the AF version of the Expert involves thirteen AFOs. This AFO suite and its associated inter-connectivity are illustrated in Figure 4.

1.3 Another Candidate Application

Although the Event Diagnosis Expert is a suitable rule set to exercise the dynamics of the Activation Framework and its interaction with the FTPP operating system, there were a number of motivating factors toward the selection of a more complex application:

1. The original translators (*Rules to EFG* and *EFG to AFOs*) were riddled with specific references to the Event Diagnosis Expert parameters essentially making the translators specific to this particular application. Consequently, rather than tailoring the translators for each subsequent application, it was

highly desirable to generalize these translators to permit relatively painless implementation of other applications. Furthermore, the generalized translators should handle all input/output connectivity relations. The application of generalized translators to another serious test case ensures not only that complex data dependencies are correctly implemented but also that all old vestiges of the Event Diagnosis Expert are removed.

2. The Event Diagnosis Expert application consisting of only 13 AFOs generated a suite of a mere 13 tasks. In terms of computational load and memory allocation this scenario underutilizes the FTPP capabilities which has 15 available processors for assignment of tasks. A significantly larger task suite would approach the limits of the FTPP's operational activity.
3. In order to develop a sophisticated load balancing algorithm a sufficiently complex data/task dependency graph is necessary to visualize the parameters which should be optimized.

A Real Time Controller was selected as this candidate application to which the AF methodology was applied. Figure 5 depicts the data dependency relations for the Real Time Controller. This task suite consists of 56 tasks, seven of which are input nodes; these tasks with the necessary output task generate a 57 AFO suite.

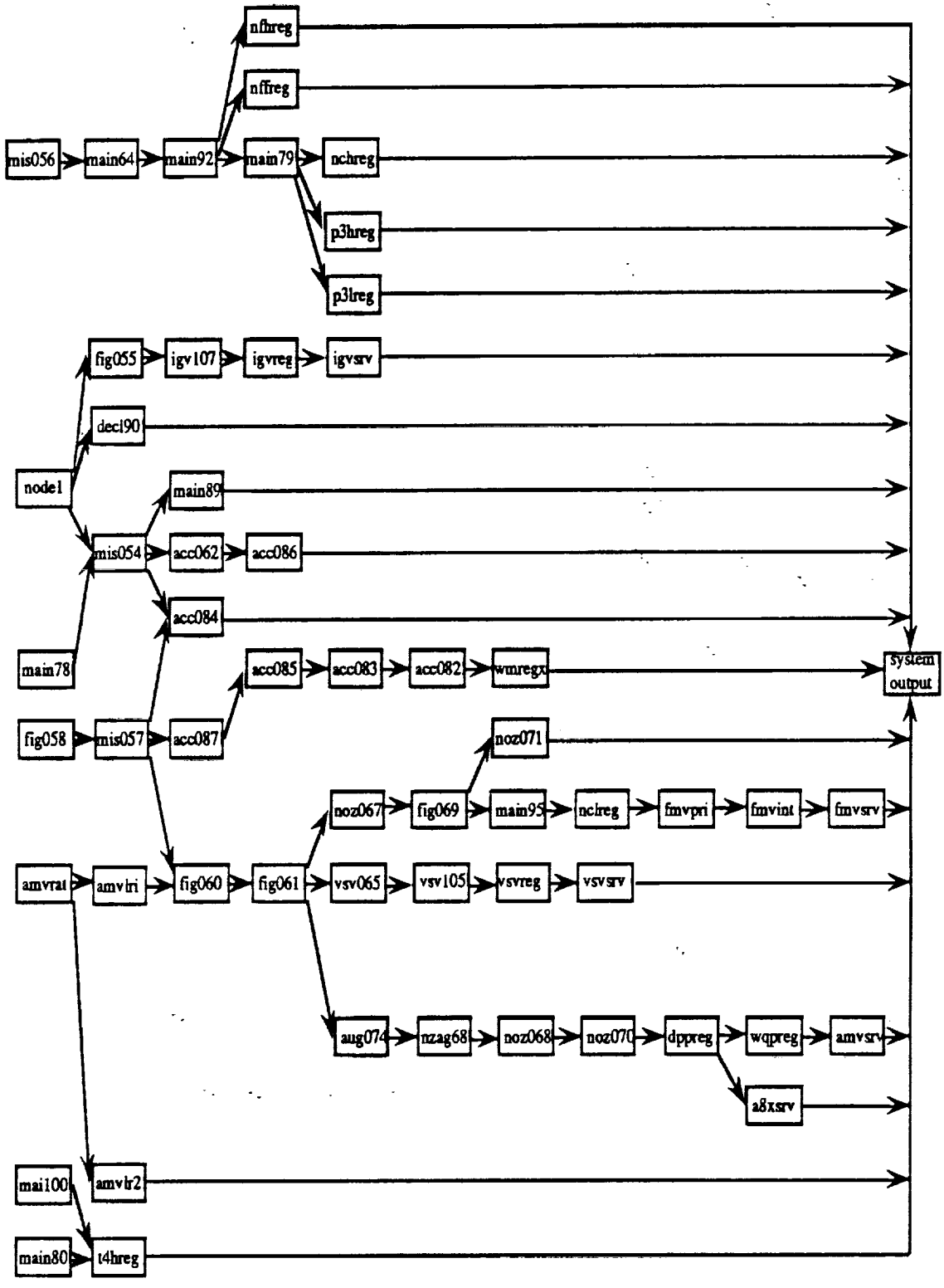

Figure 5 - The Real Time Controller Data Dependencies

1.4 The Fault Tolerant Parallel Processor

1.4.1 Architectural Overview

The basic unit on the FTPP is the *cluster* which contains four network elements and the associated processors (see Figure 6). The *network element* (NE) is a Draper designed component. The four NEs are fully connected and operate in tight synchrony within the network element core to perform message exchanges and to vote message exchanges. Messages entering the NE core are exchanged and voted according to the class parameter of the message. In addition, since messages are addressed using unique identifiers, the operation of the NE is highly contingent upon the system configuration in identifying the physical hardware associated with these source and destination addresses.

In the FTPP prototype cluster 1 (C1), each network element hosts up to four *processing elements* (PE) each of which are standard processors with local memory. The processors currently employed are Motorola 68020 processors; however, this selection is not a design criterion and, in fact, the FTPP is capable of supporting heterogeneous processors. Each processor communicates with the hosting NE via transmit and receive FIFOs which the processors access via a VSB bus.

Each network element and the associated processors comprise a *fault containment region* which satisfies the requirements for fault containment, namely, electrical isolation, physical isolation, independent power and independent clocking.

Virtual groups are logical views of the processing resources capable of accepting work in a parallel processing environment. Using this concept, the physical addresses of the processors as well as the redundancy level of a processing group can be concealed from the view of the programmer. A unique identifier is assigned to each virtual group; this is the virtual group identifier (*VID*). Virtual groups can be composed of any number of processors up to 4 processors; consequently, they may be *simplex*, *duplex*, *triplex* or *quadruplex*. Within a VID each processor is a *channel* (also referred to as a *member*). In the case of a quadruplex, the first channel is designated channel A; the second, channel B; the third, channel C; and the fourth, channel D. Similarly, simplexes have only channel A components. When operating redundantly, each processor within a VID executes a suite of tasks which are functionally congruent with the other members of its VID. For example, in an avionics application each processor of a redundant virtual group would execute the same

navigation task on identical inputs. On the other hand, simplex VIDs are merely individual processors executing tasks with no redundancy. VIDs are comprised of processors each of which must be resident in a different fault containment region in order to satisfy the theoretical requirements of Byzantine resilience. For example, a quadruplex would comprise processors resident on each NE.

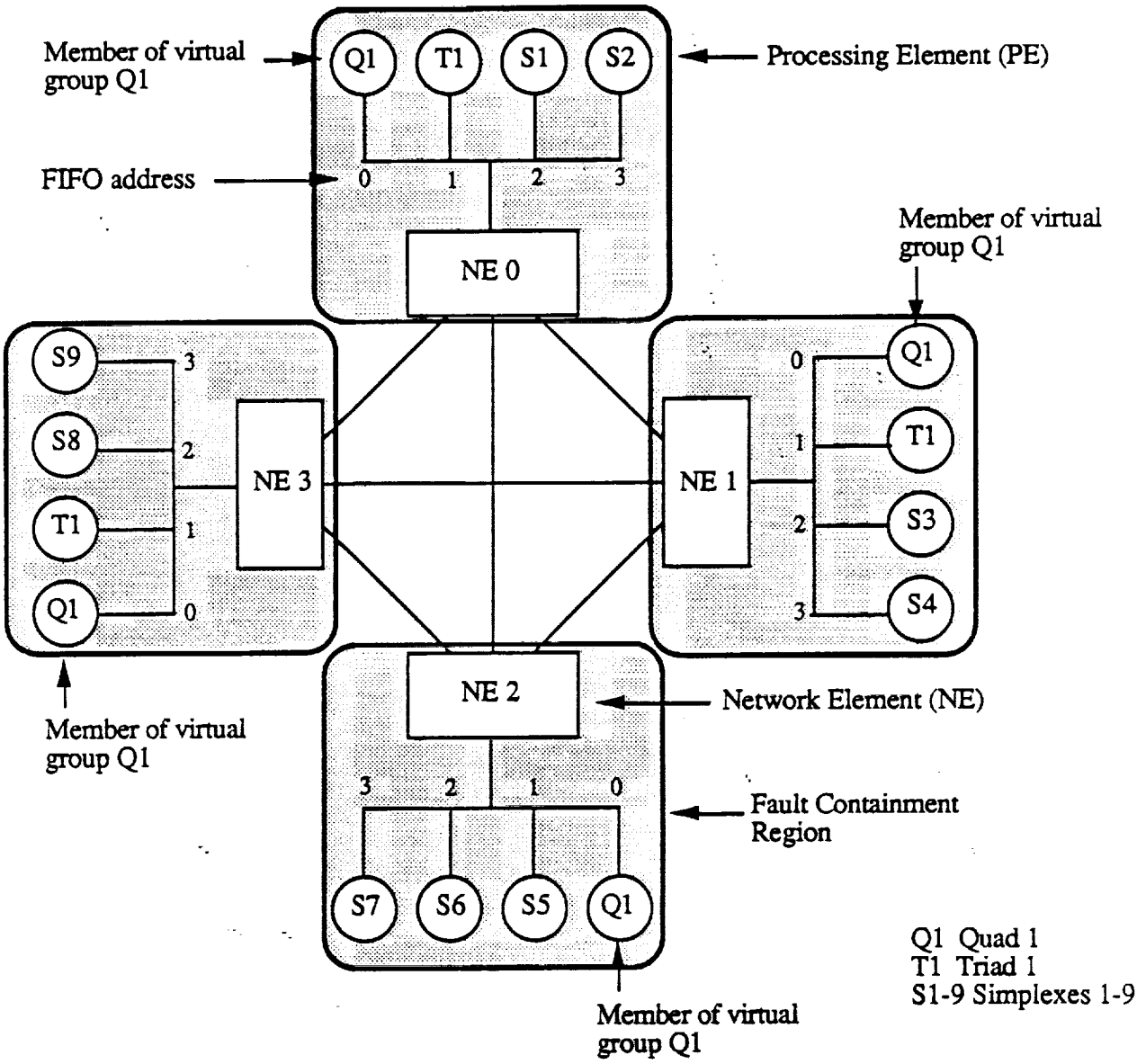

Figure 6 - FTPP Cluster Architecture

Fault tolerance on the FTPP is ensured by grouping 3 or 4 processors into VIDs called *fault masking groups* (FMG). Fault manifestations in a fault masking group can occur without

any degradation in system performance or correctness. Furthermore, these faults can be readily diagnosed.

The system *configuration table* is the mapping of processors to the virtual group identifiers. This mapping identifies the NE hosting the processor as well as the FIFO address through which the processor communicates with the NE. Since all communication within the system is based upon the VID, the system configuration table is resident in the network elements as well as in the processors. Maintenance of this table is provided by a special broadcast message interpreted by both processors and NEs and by adherence to a strict protocol when the system configuration is modified.

Figures 6 and 7 define a sample system configuration consisting of 1 quadruplex, 1 triplex, and 9 simplexes.

	VID	Member	NE id	FIFO id
Q1	12	A	0	0
		B	1	0
		C	2	0
		D	3	0
T1	9	A	1	1
		B	3	1
		C	0	1
S1	1	A	0	2
S2	2	A	0	3
S3	8	A	1	2
S4	4	A	1	3
S5	5	A	2	1
S6	0	A	2	2
S7	3	A	2	3
S8	11	A	3	2
S9	10	A	3	3

Figure 7 - Sample Configuration Table

1.4.2 Communication Mechanisms

Virtual groups communicate via messages which are of 4 basic classes: voted messages, source congruency messages, a synchronization message, and configuration update messages. A *voted message* is one sent by all members of a redundant processing group. This message type is employed only when exact consensus amongst all redundant members is expected. Conversely, a *source congruency message* is originated by a simplex processor or by a member of a redundant processing group requiring a channel-specific exchange of information. The *synchronization message* is employed to synchronize members of a virtual group. The *configuration update message* is initiated by a virtual group to modify the VID/processor mapping resident both in the network elements and in the processors.

Each member of a VID requests a message transmission by sending the message body to its associated transmit FIFO followed by storing the message class in the class FIFO. If a majority of members of a VID request a transmission, the class is voted by the NE core to determine the exchange and voting mechanisms. In addition, the destination VID is voted. Subsequently, the message body is manipulated according to the message *class*.

Message processing within the network element core is handled on a VID-by-VID basis. When a majority of the members of the source VID request transmission of a message, that message is eventually processed and delivered to all members of the destination VID. Consequently, the ordering of messages to the destination VID is preserved, thereby guaranteeing that all members of the destination VID receive messages in the same order.

Redundant members of VIDs execute functionally congruent tasks. Since their sequence of tasks is congruent across all members, messages transmitted during their normal executing cycles will necessarily be equivalent as well. Therefore, the message streams emanating from the different members will be identical at least in the message class when no fault exists. This concept is the basis of functional synchronization which is discussed in a subsequent section.

1.4.2.1 Voted Messages

When the redundant members of a VID transmit a *class 1 message* (voted message) the NEs exchange their copies of the message, create a bitwise voted copy of the message, and compare each copy with the voted copy. This final step generates a vote syndrome which is appended to the message. Network elements which host members of the destination VID deliver the message to the appropriate FIFO; other NEs discard the message.

Figures 8 through 10 depict the transmission of a message x from the triply redundant VID 1 to VID 2 which is also configured as a triplex.

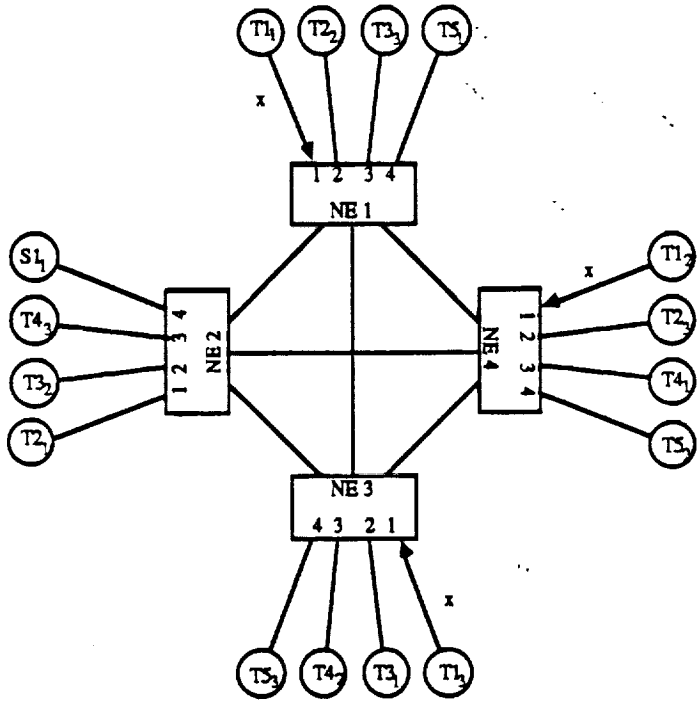

Figure 8 - Transmission of Class 1 Message

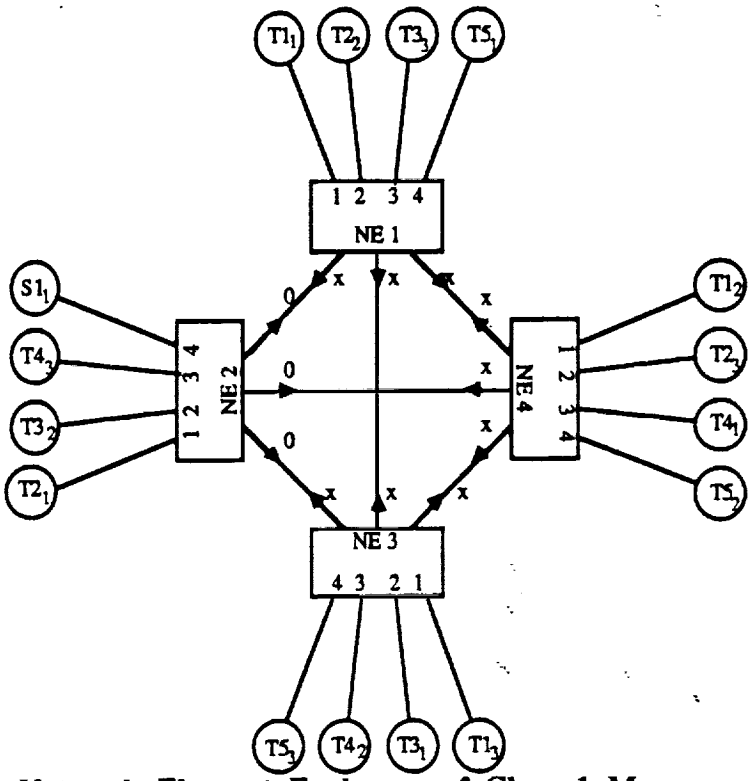

Figure 9 - Network Element Exchange of Class 1 Message

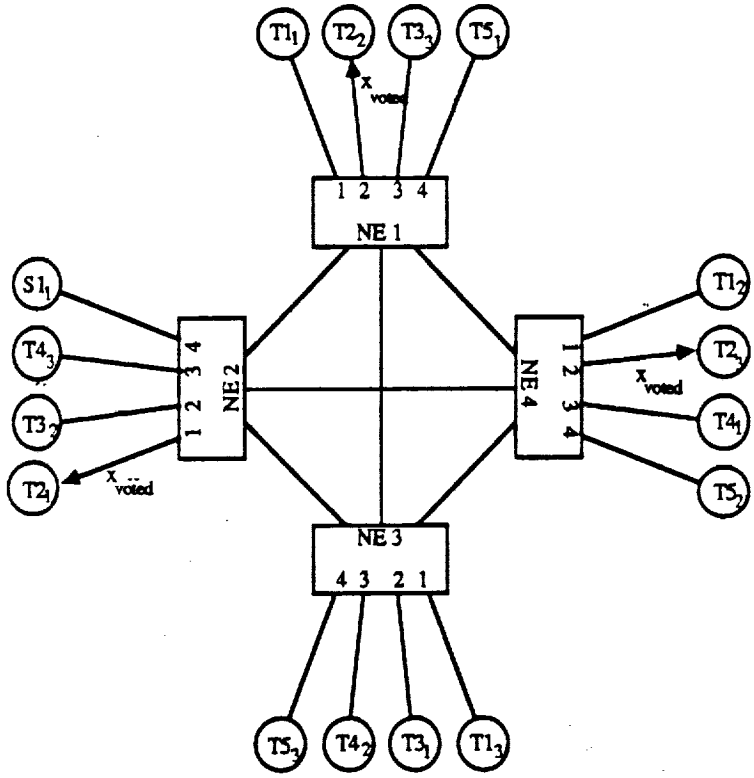

Figure 10 - Delivery of Class 1 Message

1.4.2.2 Source Congruency Messages

Class 2 messages contain channel-specific information such as the value of a processor clock. The network elements perform 2 rounds of exchange of this message, create a bitwise voted copy of the "reflected" copy and compare each "reflected" copy with the voted copy to generate the vote syndrome. Delivery of this message is similar to that of class 1 messages.

Since each NE operates simultaneously on only 1 message, each member of a fault masking group must agree upon which member's channel-specific data are being exchanged. This is achieved by the definition of 4 different class 2 messages. A "class 2 from A" message identifies the VID member A as the source of the message information. However, all members of the VID must participate in the transmission of the "class 2 from A" message. This requirement is necessitated by the fact that the NEs vote the message class from each member of the transmitting VID. If one member transmits a "class 2 from A", the second member sends a "class 2 from B" and the third member sends a "class 2 from C" simultaneously, there will be no consensus on the class of the message. A bitwise voted class is generated and the messages from each member of the sending VID are handled according to this voted class. Therefore, in order to perform an exchange of information where each member receives each other's copy of some information, a series of messages containing this information must be sent by each member of the VID. Each member of a triply redundant VID must sequentially send a "class 2 from A" message, a "class 2 from B" message and a "class 2 from C" message.

Figures 11 through 13 describe the sequence of events in the processing of a class 2 message. The delivery of a class 2 message is identical to the delivery of a class 1 message depicted in Figure 10.

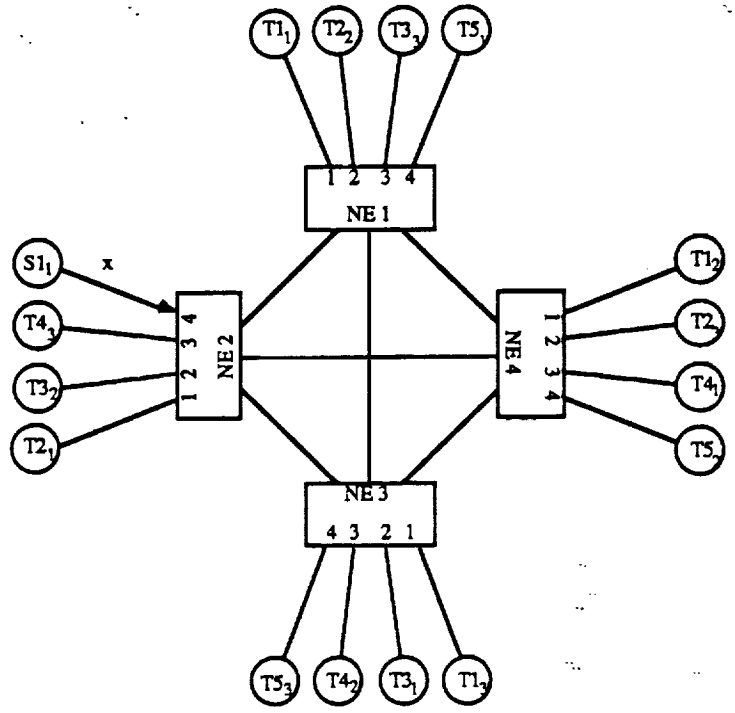

Figure 11 - Transmission of Class 2 Message

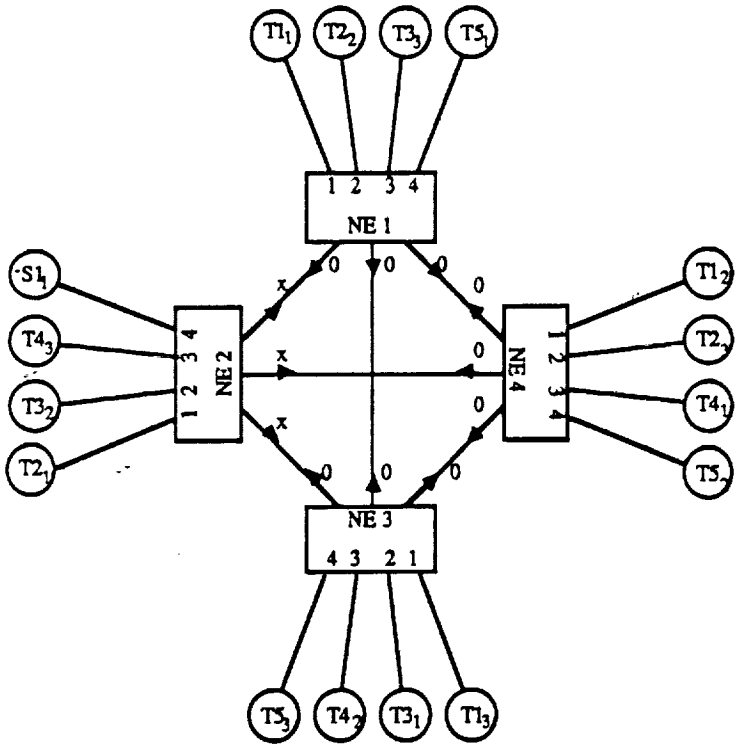

Figure 12 - Network Element Exchange of Class 2 Message

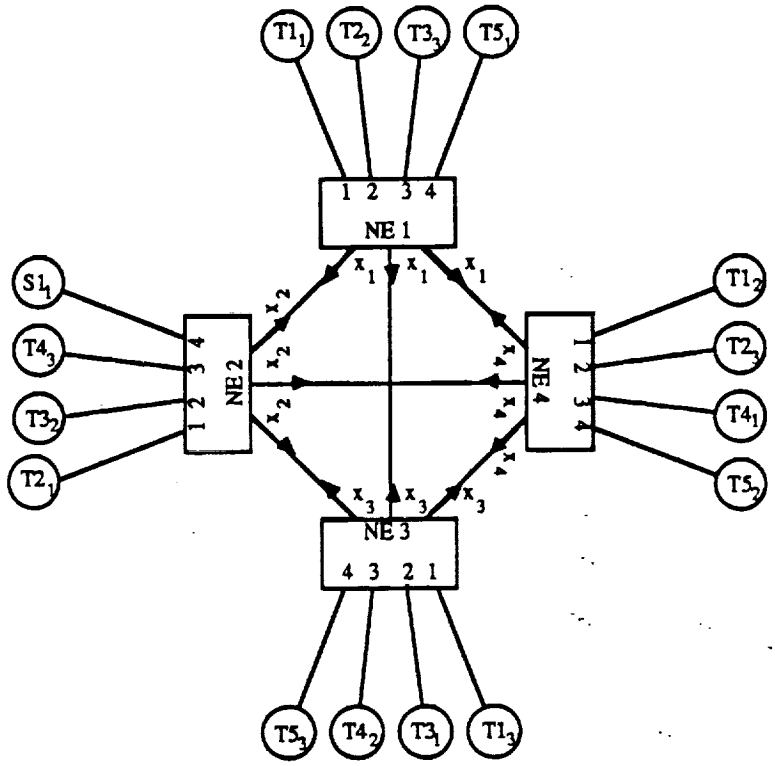

Figure 13 - Network Element Exchange of "Reflected" Class 2 Message

1.4.3 Synchronization

As stated earlier, each member of a VID executes a suite of tasks which are functionally congruent with the other members of the VID. In order to reduce the time skew among the members they must perform some synchronizing act. Since synchronization occurs after some function or sequence of functions has been completed, this methodology has been termed *functional synchronization.* To maintain synchronous operation, members of a VID must periodically synchronize. This sequence of functions between synchronization points is referred to as a *frame* in Figure 14.

The implementation of functional synchronization requires that the sending VID also receive the synchronization message. The process known as *scoop*ing implements this concept (see Figure 15). In essence, a task wishing to perform a functional synchronization sends a scoop message to its own VID. Messages received prior to this scooped message are stamped as readable. Furthermore, due to the fact that the NEs are

tightly synchronized, each member of the synchronizing VID is guaranteed to receive the message within a bounded skew of its other members.

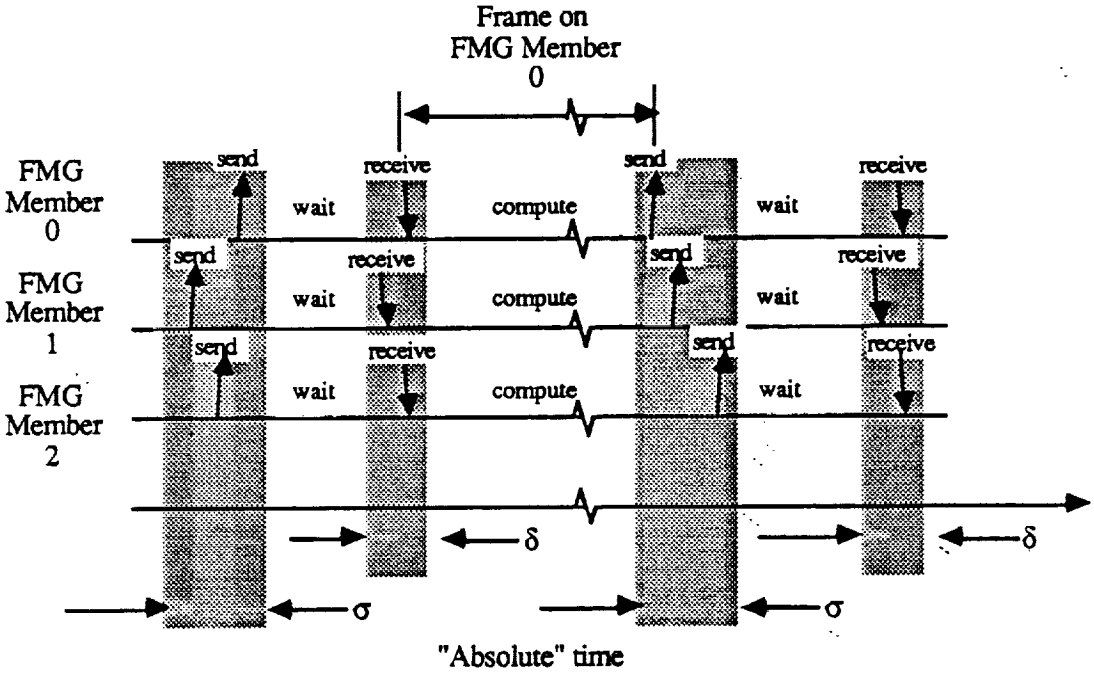

Figure 14 - Functional Synchronization

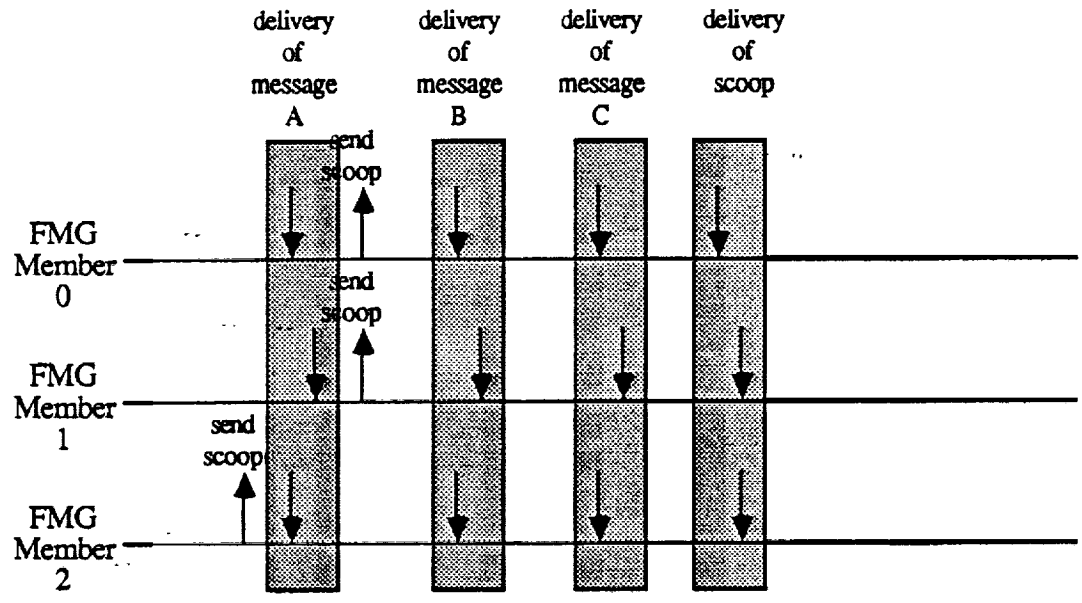

Figure 15 - Scooping a Message

Functional synchronization is implemented by sending a class 0 message via the sync_self primitive or by performing a scp_msg primitive. Both of these scooping mechanisms transmit a special message and block awaiting its return without rescheduling any other tasks. Recognition of this special message is achieved by comparing all incoming messages with the known message. The sync_self primitive sends a class 0 message which has no message body; therefore, only the class of the incoming messages needs be examined. On the other hand, the scp_msg primitive does transmit a complete message requiring inspection of all incoming message bodies until the special message has arrived. Upon resumption of the task calling these primitives, the members of the VID are synchronized. The use of the sync_self mechanism is limited by the fact that the source and destination VIDs must be identical. However, scp_msg messages also may include the set of broadcast messages.

Inter-virtual group synchronization among VIDs is provided simply by sending messages without the tight synchronization of the intra-VID protocols. The time when the message is transmitted by one VID and delivered to another VID is not constrained. In fact, unless the message is broadcast, the sending VID receives no explicit acknowledgement that the message has been delivered. It can only be guaranteed that the message will be delivered in the order sent and that the time of delivery to the redundant recipients will be bounded.

1.4.4 Operating System Functions

The FTPP operating system consists of both asynchronous and synchronous services. The asynchronous services are interrupt driven tasks which include a facility to empty the input FIFOs.

The synchronous services are invoked by the scheduler. The operating system functions include a time function, the front end processor host (FEP), the fault detection and identification function, and the reconfiguration function. Application programs are invoked synchronously concordant with the other operating system functions.

1.4.4.1 Scheduler

The current version of the operating system is a real time system utilizing some of the features of the Versatile Real Time Executive (VRTX) operating system of Hunter & Ready, Inc. VRTX provides the primitives for task suspension, task resumption, and the associated stack manipulations.

The core of the operating system is a non-preemptive scheduler which invokes the other operating system functions and application tasks in a round robin fashion. Tasks (also called services) are scheduled for execution once each scheduling loop or upon receipt of a message for that service; the selection of the scheduling mechanism is determined by the task designer.

Tasks may communicate either via global variables shared with processes in the same VID or via messages to processes executing in other VIDs. Since the scheduler is non-preemptive, suspension of the task is the responsibility of the task itself. Task developers must be judicious in relinquishing control of the processor to prevent starvation of the other tasks.

The fixed time frame typical in real-time systems has been relaxed for the C1 prototype. This departure from normal real-time system implementations results from the requirement to preserve the functional synchronization concept which requires that message streams across all members of a VID be consistent. The transmission of messages from tasks which are executed using time based preemption on non-clock deterministic processors could violate this constraint and result in the transmission of interleaved messages from various tasks.

1.4.4.2 Message Handling

Messages are defined by the application task but are addressed to a VID and a service using the user selected message class. The class of the message is defined by the task to be either a normal message (that is, class 1) or an explicit class 2 message. In the event that the redundancy of the source VID is simplex, normal messages are translated to a class 2 message by the operating system thereby concealing the underlying redundancy level of the task.

Conveyance of a message is initiated by a snd_msg primitive. The operating system divides this message into packets, computes and appends a checksum to the message, and transmits the individual packets via the output FIFOs. The snd_msg function is a blocking invocation which transmits the message immediately.

As indicated earlier, the processing of incoming packets is handled both asynchronously by an interrupt-driven process and synchronously by the scooping mechanisms. In either case the input FIFOs are polled to determine the presence of a packet. Received packets are deposited in a receive packet queue and relevant information regarding the status of the message under construction is saved in the data structures associated with each source VID. Header information regarding the message length is retrieved and subsequent packets are linked in the receive packet queue to complete the task's message.

Completed messages can be read by a task only after a subsequent message has been identified as either a scoop message or a class 0 message at which time a message is stamped as readable. When a message has been stamped and when an inquiring service requests retrieval via the get_msg, the message is reconstructed from the receive packet queue into a contiguous byte stream. The get_msg function is non-blocking and returns the message and a boolean value indicating whether a message is available.

1.4.4.3 Time Keeper

In order to create a uniform clock available to all VIDs within the system, a time keeper function was devised. One virtual group in the system is responsible for the function of reading its local processor clock and broadcasting a consistent, successively increasing time to each VID. In the event that the time keeper is redundant, each member of this VID reads its local clock, exchanges these values with each other using class 2 messages and performs a mean value select to compute the system time which is then broadcast.

1.4.4.4 Front End Processor Host

In addition to other Operating System (OS) functions, one simplex VID has the static role of being the designer's window into the system. The Front End Processor Host (FEP Host) is an OS function which interfaces to a Macintosh computer. The FEP host accepts

messages from the operator via the Macintosh front end, composes messages and transmits these messages to the appropriate VID. Likewise, VIDs may send information to the FEP host for display purposes on the Macintosh.

1.4.4.5 Fault Detection and Identification

One of the cornerstones of the FTPP is its ability to withstand faults. However, to keep the system fully operational faults must be detected and removed as soon as possible. Therefore, the fault detection and identification function (FDI) operates periodically to monitor fault detection mechanisms. Faults which have occurred are remedied in a timely manner to prevent possible system failure.

The major fault detection mechanism is the syndrome information generated by the network elements and appended to each packet transmitted through the network element core. These three bytes of error information are indicative of faults in either the processing elements or network elements. Since the syndrome information is appended to the packet prior to delivery, the FDI function on the recipient VID monitors the fault mechanisms.

The syndrome information is extracted from each packet and buffered as each packet is scooped. Logging the syndromes in this manner preserves the functional synchronization among members of a VID ensuring that the syndrome buffering is consistent. This buffered syndrome information represents the perspective of each member of the VID which is probably different than that data possessed by the other members of the VID. Furthermore, in the case of a Byzantine failure, the other members would have dissimilar information. The members of a VID performing the diagnosis may diagnose different components or may differ in their opinions as to whether a fault even exists. For these reasons, the members of a redundant VID exchange an error vector indicative of whether an error exists. This is accomplished by an exchange of these error vectors via class 2 messages. After this exchange each member of the VID will have a consistent set of error vectors -- one for each member. Subsequently, the VID analyzes the vectors to determine if an error has occurred. In the event that one has occurred, the syndrome exchange phase is invoked to disseminate the detailed diagnosis information. Similarly, a series of class 2 messages are created to exchange detailed syndrome information. When this data has been disseminated, the diagnosis phase is entered.

The repertoire of diagnosable faults is currently limited to those with strong signatures. Not only must all members of the diagnosing VID detect the error but each member of the diagnosing VID must also target a specific component as faulty. The faulty component is identified using the class of the message as well as the system configuration table (in the event of a processing element fault) to target the specific hardware component. Table 1 defines the repertoire of diagnosable faults; the mechanisms are indicated.

Component	Configuration	Syndrome	Message Class
Processor	FMG member	vote	1
NE	-	vote	2
NE	-	synchronization	-

Table 1 - Diagnosable components

Diagnosis of a faulty component may occur by any virtual group, even the faulty VID itself, since a VID is theoretically guaranteed to operate correctly even in the presence of a fault. However, precautions exist to ensure the nonparticipation of the faulty processor in both the error vector and syndrome exchanges.

When a component is diagnosed, a message is broadcast to the reconfiguration service on all VIDs. This permits all sites to be cognizant of the fault and to modify status information to prevent subsequent diagnoses. A processor fault diagnosis will initiate a reconfiguration strategy designed to replace the faulty processor. On the other hand, in the C1 prototype FTPP a network element cannot be replaced; instead syndrome errors emanating from a diagnosed NE are masked, concealing possible faulty processors resident on that NE as well.

1.4.4.6 Reconfiguration

The reconfiguration task performs various functions associated with the creation and dissolution of VIDs of the required redundancy level. Two reconfiguration strategies exist to satisfy different operational requirements. These alternatives are a total reconfiguration and a processor replacement reconfiguration. The total reconfiguration strategy establishes a system configuration with VIDs of the specified redundancy levels. It attempts to satisfy the request by creating redundant groups from simplex processors or by disbanding

redundant groups to satisfy the need for the specified number of simplexes. This strategy is currently initiated by an operator command from the Macintosh front end processor requesting specific numbers of simplexes, duplexes, triplexes and quadruplexes. This strategy could also be invoked automatically during a mission sequence requiring a modification in the redundancy levels for various tasks. Alternatively, the processor replacement strategy is invoked upon receipt of a reconfiguration request triggered by the diagnosis of a faulty processor.

In general, each reconfiguration strategy performs the same general sequence of operations:

1. Contend for reconfiguration authority
2. Virtual group selection
3. Global notification
4. Configuration table updates
5. Synchronization of virtual group
6. Reconfiguration termination.

The reconfiguration authority (RA) is a single VID which directs the reconfiguration operations of deciding which VIDs shall be reconfigured, of deciding which VID shall be the system time keeper, of initiating the reconfiguration message sequence and of accepting acknowledgements. The selection of the RA is implemented by a contention algorithm in which VIDs satisfying certain criteria broadcast a "contend for RA" message. The source VID of the first broadcast message received becomes the RA; subsequent "contend of RA" messages received are discarded.

The RA selects those VIDs to be reconfigured; this is highly dependent upon the reconfiguration strategy invoked. In the case of total reconfiguration, the RA studies the current system configuration, the requested configuration, and charts a plan of attack consisting of creating or disbanding VIDs where necessary. For a processor replacement, the RA merely searches for a simplex VID which resides in a fault containment region different from the other non-faulty members of the compromised VID. If there is no candidate simplex, a message is broadcast to FDI indicating that the VID is unrepairable.

The selection of VIDs to be reconfigured is succeeded by a broadcast global notification message indicating those VIDs chosen. All VIDs must acknowledge this message; in effect, confirming that they will not send any messages to those VIDs undergoing

reconfiguration. This precaution is necessary to avoid the transmission of a message addressed to a VID which may disappear or which may alter its VID/physical hardware mapping. Broadcast messages may still occur during reconfiguration without any restriction. The RA may send messages addressed to those VIDs except during the configuration table update phase.

A well ordered series of configuration update messages are broadcast by the RA to modify the mapping of VID to physical NE/FIFO. This process activates the new virtual groups causing them to enter their synchronization phase. The processor replacement strategy is constrained to maintain the identity of the diagnosed VID.

A VID undergoing reconfiguration synchronizes its members by scooping a special message transmitted to itself. Upon receipt of this message all members operating in unison send an acknowledgement to the RA indicating their successful synchronization. The total reconfiguration strategy creates virtual groups which execute from an initial state; all tasks which had been executing prior to the reconfiguration are terminated and new tasks are created when this new VID is born. Furthermore, since any unread messages were addressed to the "old" VID, the packet queues are flushed. The processor replacement strategy reconfigures 2 VIDs -- a fault masking group in which a member is replaced and a faulty simplex VID. The fault masking group resumes in an *aligned* state; the alignment process copies the memory image from a non-faulty member to the new member so that once the alignment process completes tasks may continue to operate with minimal disruption. Unlike the total reconfiguration strategy, this reconfigured VID is not reinitialized by killing tasks and subsequently recreating them as a "new" VID. Likewise, the packet queues are not flushed; the "new" member of the VID receives a copy of the packet queues by virtue of the alignment process. The faulty simplex resumes in an *initial* state as in the total reconfiguration strategy. As indicated previously, this simplex would normally enter a diagnostic phase to determine its health; however, this is still unimplemented.

Until the RA receives the acknowledgement from the newly formed VIDs, the system is still in an interim state. All other VIDs are unable to communicate directly with those reconfigured VIDs. However upon receipt of the acknowledgements, the RA lifts this restriction by broadcasting a reconfiguration termination message.

2.0 The AF-FTPP Modifications and Utilities

To permit the execution of the Activation Framework on multiple processors of the FTPP, an interface between the AF and the FTPP had to be designed, and the AF, AFOs, and FTPP Operating System had to be modified. This AF-FTPP Interface was developed to allow the integration of the Ada based AF and AFOs with the C based FTPP Operating System. The AF and AFOs were modified to interact with this Interface and thus utilize the FTPP Operating System primitives. Further, the FTPP Operating System was augmented to support the AF methodology and the AF-FTPP Interface. In addition, to allow the execution of applications other than the Event Diagnosis Expert, the Rules to Evidence Flow Graph (EFG) and EFG to Activation Framework Object translators were modified permitting the parsing and interpretation of more general rule sets.

A set of utilities was also developed during the KRAPP project. They were created to ease the evaluation of the parallelized AF and to minimize the cost of the integration. First, a load balancer was designed and implemented. This procedure parses the AF frame file and determines a sub-optimal AFO to VID mapping. Additionally, an automatic load module generator was implemented. This facility uses the load balancer and a set of code generators to create FTPP load modules. Further, to enable message transmission and reception from a remote source, a data insertion and capture facility was designed. Finally, a conversion program (from "VOX" object module format to UNIX System 5 "a.out" format) was written to enable the development of an Ada - C mixed system while utilizing CSDL's currently available resources.

The following sections detail the design and implementation of the aforementioned modifications and utilities.

2.1 The Rules to EFG and EFG to AFOs Translators

A suite of AFOs is generated from a set of Horn clauses, or rules, that represent the knowledge of an intelligent system. To create the AFOs, the Horn clauses must be parsed, interpreted, and translated. This procedure is completed in two independent steps. First, the Horn clauses are converted to an Evidence Flow Graph (EFG), which is an intermediate set of data structures that characterizes the clauses. Second, the EFG is

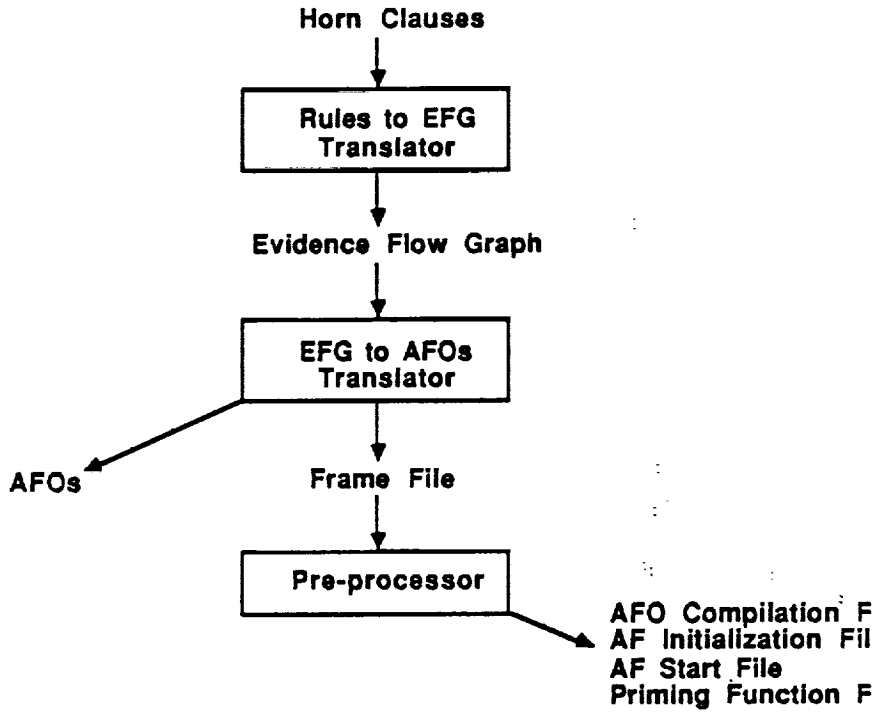

Figure 16 - The Rules to AFOs Translators

translated into AFOs. These two translators are shown in Figure 16. They were designed and implemented in Ada by Worcester Polytechnic Institute under the Knowledge Representation into Ada Methodologies project (KRAM).

The translators developed by WPI had a limitation; the EFG and AFOs, that were generated, were tailored to the Event Diagnosis Expert application. For example, the names of Event Diagnosis AFOs were incorporated into the intelligence of the translators. Furthermore, the connectivity of the application was known apriori and embedded into the translators. Additionally, assumptions were made concerning the data structures being passed between the AFOs, and this information was incorporated into the translation logic.

An extension of Rules to EFG and EFG to AFOs translators, such that they are capable of interpreting and processing a general application, was desirable. Accordingly, under the KRAPP project, these translators were modified. Specifically,

1. The List Management System (LMS), which is the foundation of the translators, was modified to permit its execution on a VAX 8650 using the VAX Ada Development System. Certain Ada constructs, which are not supported by the VAX, are used by LMS. Subsequently, these constructs were replaced.
2. The syntax of the Horn clauses was extended to allow the specification of multiple input and output ports. The translators were modified to interpret this enhanced syntax and automatically generate the corresponding connectivity. This extension enabled the translation (Rules to AFOs) of general purpose applications.
3. The assumptions concerning the data structures passed between the AFOs were relaxed. While still being limited to booleans and integers by the Rules to EFG translator, the EFG to AFOs translator was enhanced to parse and interpret generalized data types. As a result, if the Rules to EFG translator is modified to allow Generalized Object (GO) representation in the EFG, then the GO data types would be maintained and passed to the AFOs by the EFG to AFO translator.
4. The EFG to AFOs translator was modified to restructure the AFOs so that they could be executed on the FTPP.

In addition to the translators, WPI developed a preprocessor that uses the frame file to create "support procedures" for the AF (shown in Figure 16). The frame file is generated by the EFG to AFOs translator, and it characterizes the attributes of the AFO suite (e.g., names, number of inputs and outputs, connectivity, etc.). The AF preprocessor creates three AFO/AF interface files written in Ada and a command file for compiling and linking the AFOs. The AFO/AF interface files are: (1) an Ada package that controls the initialization of the AFOs (*af_start*), (2) a procedure that starts the AF (it is called the *af_run* procedure and is discussed in Section 2.2), and (3) a procedure that permits selection and execution of the AFO priming conditions (*execute*). This preprocessor was also modified by the KRAPP project. These modifications were performed to permit and ease the execution of the AFOs on the FTPP, the compilation of the AFOs on a MicroVAX III, the conversion and translation of the Ada object modules to Heurikon C object modules (discussed in Section 2.5), and the creation of FTPP load modules.

The modifications made to the translators and preprocessor were tested using the Event Diagnosis Expert. Specifically, the EFG and AFOs generated by CSDL were compared to

those created by WPI. Furthermore, the extensions incorporated into these procedures were exercised and verified by developing a set of Horn clauses that represents a data flow diagram for a Real Time controller. The translation process correctly generated a suite of 57 AFOs and the corresponding support files.

2.2 The Interface between the Activation Framework and the FTPP Operating System

The Activation Framework was developed by WPI and is written in Ada. The FTPP Operating System was designed by CSDL and implemented in C. To allow the execution of the AF on the FTPP, an interface between the AF and FTPP Operating System had to be developed. This interface, whose addition caused modifications in both the AF and the FTPP Operating System, is composed of numerous C and Ada procedures.

For clarity, the AF-FTPP Interface is segmented into four sections: (1) scheduling, (2) message transmission, (3) message reception, and (4) performance timing. These sections are depicted in Figures 17 through 20, and in each figure, the work performed by WPI is separated from that completed by CSDL. Albeit the AF-FTPP sections are separated in this discussion, they are closely intertwined.

2.2.1 AFO Scheduling

As mentioned earlier, the AF-FTPP Interface permits the integration of the AF and AFOs with the FTPP Operating System. The scheduling component of this Interface is shown in Figure 17, and it allows the FTPP Operating System to: (1) initialize the AF, (2) invoke the AF scheduler to determine the next AFO to execute, and (3) execute the transfer function of a designated AFO. Furthermore, if the FTPP's redundancy management (RM) processes were incorporated into the AF-FTPP methodology, then the AF-FTPP scheduler would also execute the RM tasks.

With respect to Figure 17, the AF-FTPP scheduler is the process *sched*. To initialize the AF, *sched* calls the procedure *af_run*. As discussed in Section 2.1, the *af_run* procedure is automatically generated by the AF preprocessor, and its content reflects the AFOs involved in the application. More specifically, it is a series of invocations, one set per AFO, which

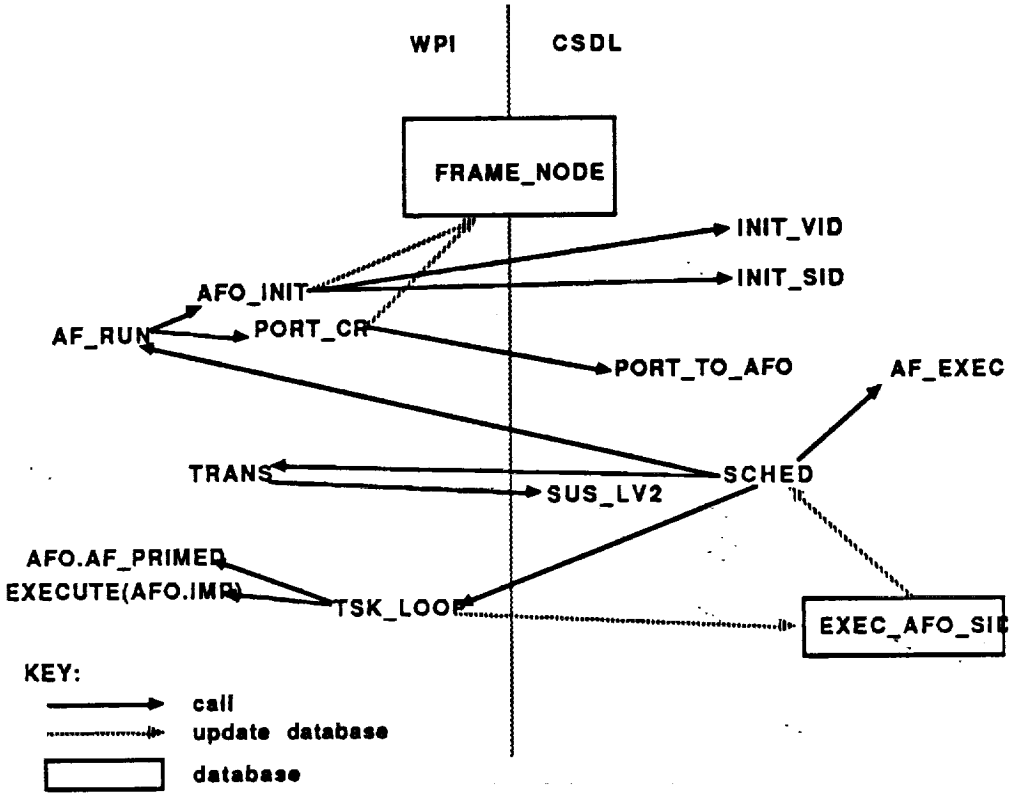

Figure 17 - The AF-FTPP Interface for Scheduling

constructs a database for the AFO suite. This database is called the *Frame_Node*, and it is initialized using WPI's *AFO_Init* and *Port_Cr* procedures. The *AFO_Init* procedure is responsible for allocating and initializing an AFO data structure whereas the *Port_Cr* procedure creates and initializes a port data object. These procedures were augmented by CSDL to permit the use of multiple processors and to integrate the AF and FTPP schedulers. Namely, the *Init_VID* and *Init_SID* functions were developed and incorporated into the *AFO_Init* procedure. The *Init_VID* function is used to return an identifier that indicates the location of the AFO (the VID on which it resides). Further, the *Init_SID* function provides an identifier which is necessary to schedule the AFO (gives the AFO's Service ID or SID). The *Frame_Node* database was also extended to allow the recording of these VID and SID fields. Additionally, the *Port_to_AFO* function was implemented and integrated with the *Port_Cr* procedure. It informs the AF-FTPP Interface of the AFO/port assignments.

Each AFO and its associated ports are initialized on each VID of the FTPP, regardless of the AFO to VID distribution. This is performed to ease and optimize the inter-VID communication. If the AFO initialization process was not the same on each VID, then, when communicating to a remote AFO, the name of the destination port would have to be encoded into the message (the port name is 60 bytes long). Nevertheless, since the initialization process is identical on all of the VIDs, the AFO and port identifiers are global variables. As a result, only the destination identifier is required in the message (port ID is merely 4 bytes long), and thus the overhead due to the communication process is minimized.

After the AF is initialized, the "external input AFOs" are primed by the *AF_Exec* task (*AF_Exec* is invoked from the *af_run* procedure). This function of initially priming the "external input AFOs" has been simulated by a hand-coded Ada procedure (*input_afo*) which sends messages to these external input AFOs. These AFOs distribute the data that is inserted into the application. For the Event Diagnosis Expert (illustrated in Figure 14), the external input AFOs are *eq_mode_health*, *pilot_not_busy*, *health*, and *jtids*. After one or more of these AFOs have been primed, the execution of the Event Diagnosis Expert can begin.

To determine the next (or first) AFO to execute, the *sched* process invokes the AF scheduler which is called *Tsk_Loop*. This procedure locates the primed AFO with the highest importance. To determine if an AFO is primed, the *Tsk_Loop* procedure checks the AFO's *AF_Primed* field (this field is updated during message delivery and retrieval). Further, to calculate an AFO's importance, the *Tsk_Loop* process calls the *execute* procedure with the AFO's Importance field. When the primed AFO with the highest importance is determined, the *Tsk_Loop* procedure updates the *exec_afo_sid* field to reflect the AFO's identity.

Sched uses the *exec_afo_sid* field to execute the corresponding AFO transfer function, *Trans*, which is embedded in a VRTX task. When the *Trans* function completes, it calls the FTPP primitive *sus_lv2* to return control to the *sched* process. After *sched* is resumed, the *AF_Exec* task performs a synchronization thereby scooping all incoming messages as it removes the messages from the FTPP input queue. The *AF_Exec* process retrieves the

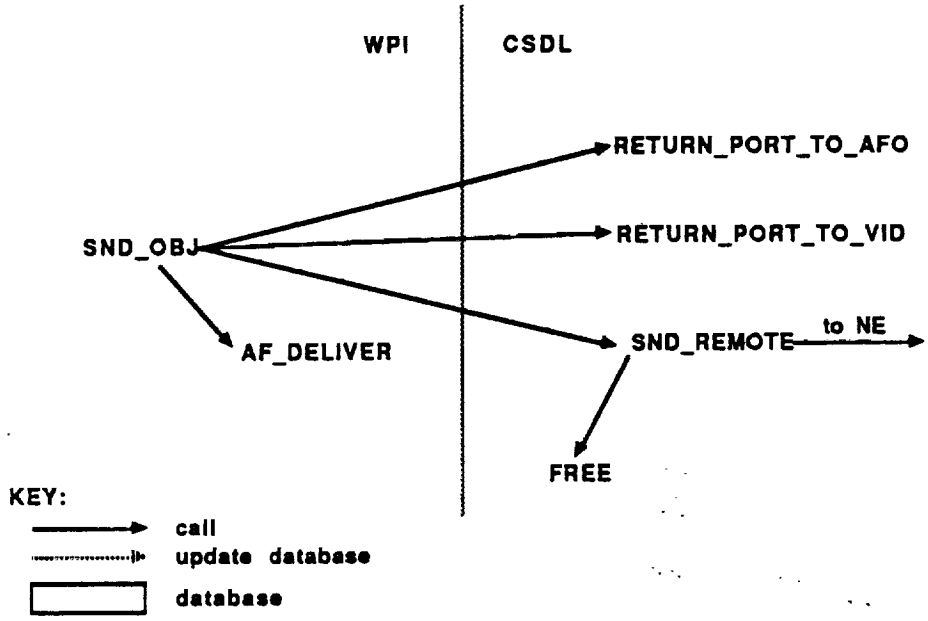

Figure 18 - The AF-FTPP Interface for Sending Messages

messages one at a time and delivers them to their destination AFOs. When the FTPP input queue is empty, the *Tsk_Loop* procedure is called to determine the next AFO to execute. This three step process (determine next AFO, schedule AFO, and deliver remote messages) is repeated until the Event Diagnosis Expert completes.

2.2.2 AFO Message Transmission

The second major component in the AF-FTPP Interface is responsible for message transmission and is illustrated in Figure 18. The AF primitive *Snd_Obj* transfers a specified object to a designated port of an AFO. The *Snd_Obj* procedure determines the location of the destination AFO by invoking the *Return_Port_to_AFO* function. This function returns the AFO that owns the destination port, if that AFO is local to the VID. Conversely, it returns -1, if the destination AFO resides on a remote VID. If the destination AFO is on same VID as the AFO sending the message, then the AF procedure *AF_Deliver* is invoked to complete the communication. Alternatively, if the destination

AFO resides on a remote VID, then the AF-FTPP Interface procedures *Return_Port_to_VID* and *Snd_Remote* are called to perform the message transfer. The *Return_Port_to_VID* function indicates the VID on which the destination port resides. The *Snd_Remote* procedure converts the message from the AF format to the FTPP format and sends it to the destination VID using the FTPP Operating System primitive *snd_msg*. Subsequently, the local memory used by the AF message is deallocated using the *Free* procedure.

2.2.3 AFO Message Reception

The AF-FTPP component responsible for receiving messages from AFOs on remote VIDs is depicted in Figure 19. As mentioned earlier, the *sched* process executes the *AF_Exec* task to deliver remote messages to the local AFOs. When scheduled, the *AF_Exec* process calls the *External_Input* procedure. The *External_Input* procedure queries the VID's input queue to locate a message. If a message exists, then the memory necessary to store it is allocated (via the *Malloc* and *Sbrk* functions), and the message is converted from the FTPP format to the AF format. The message is then delivered to the appropriate port of the destination AFO using the AF primitive *AF_Deliver*. Furthermore, during the delivery process, the *execute* procedure is invoked to determine if this new message causes the destination AFO to be primed.

2.2.4 Performance Timing

The final component of the AF-FTPP Interface is a timing utility, and it is shown in Figure 20. Currently, the only method at CSDL to integrate the AF with the FTPP Operating System requires four compilers, two object module translations, and four loosely connected computers (discussed in detail in Section 3). Consequently, because of the configuration control and continual code modifications that are necessary, the recording of performance measurements using discrete digital outputs and an oscilloscope, while very accurate, is extremely tedious. To facilitate the measurement process, a timing procedure using the local PE clock was developed. This process is non-intrusive, and it permits the measurement of multiple functions at once. This timing mechanism uses two procedures: one to start the recording process and another to stop it.

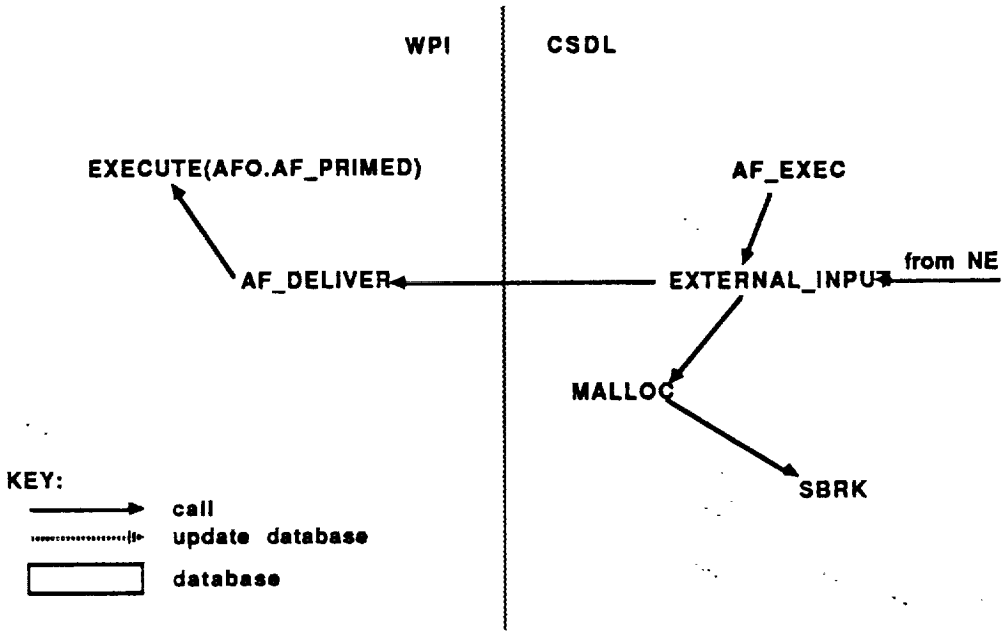

Figure 19 - The AF-FTPP Interface for Receiving Messages

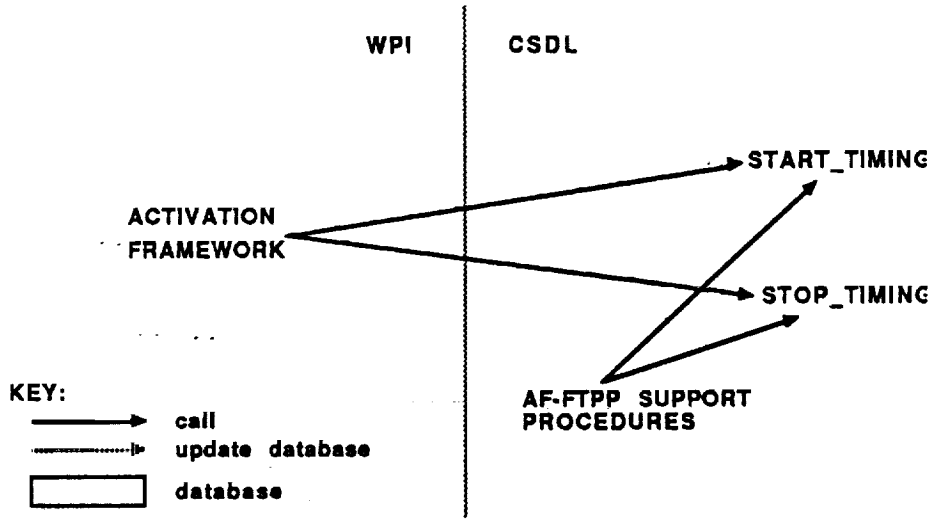

Figure 20 - The AF-FTPP Interface for Timing

The start procedure:

1. Accepts an input parameter to associate the reading with a position in the code.
2. Reads the time which is maintained by the PE.
3. Records the time in an array entry, whose position is indicated by the input parameter.

The stop procedure:

1. Accepts an input parameter to correlate the stop time with the start time.
2. Reads the time.
3. Subtracts the start time from the stop time to calculate the elapsed time.
4. Determines the number of invocations of the start and stop procedures that were performed in this interval, inclusive.
5. Compensates for the timing intrusion by using (4) to adjust the elapsed time.
6. Keeps a cumulative record of the number of timing calls (made with this input parameter) and the elapsed time.

By using this process, any set of independent or nested procedures can be measured at once. Further, the development of this process permitted CSDL to embed the measurement facility in the AF and AFOs. Accordingly, a comprehensive set of performance measurements can be recorded for each execution of the Event Diagnosis Expert without modifying any of the software.

A complete set of process description for the AF-FTPP Interface is presented in Appendix A. Each diagram provides the process' inputs, its outputs, and a functional description.

2.3 The Modification of the Activation Framework

The Activation Framework was designed to execute on a personal computer (PC) using an Ada development system. Accordingly, WPI developed several utilities to enable the AF to operate in a "stand alone" fashion. For example, WPI implemented a scheduler in assembly language to allow the AFOs to be executed on a priority basis. This stand alone Ada AF was given to CSDL to be used for the KRAPP project.

The AF was modified to permit its execution on the FTPP. The execution of the AF on the FTPP is a more complicated process than on a PC, primarily because the AF must be integrated with the FTPP Operating System. For instance, the AF had to be modified to utilize the FTPP scheduler rather than the WPI scheduler. Furthermore, the procedures that supported the AF's stand alone operation had to be removed. In addition, since the Ada compilers used by WPI and CSDL were different, the Ada structures that the CSDL compiler does not support had to be replaced. The dynamic memory allocation schemes used by WPI and CSDL also differed, and consequently, further modifications were necessary.

A detailed listing of the modifications and additions made to the AF are presented in Appendix B.

2.4 The Modification of the Activation Framework Objects

Unlike the AF, the modifications required to allow the execution of the AFOs on the FTPP were few and minor. These modifications were necessary to ensure correct scheduling control flow and to send output messages to the FTPP Operating System. The adjustments performed by CSDL were:

1. A call to the FTPP primitive *sus_lv2* was added to each AFO transfer function to allow the embedding of the function in a VRTX task.
2. The transfer function of each AFO was embedded in an endless loop. This was required to permit multiple invocations of the transfer functions.
3. The *System_Output* AFO was modified to call the AF-FTPP Interface procedure *print_system_output*. Accordingly, the output messages sent by the System_Output AFO could be printed by the FTPP Operating System.

Each modification to the AFOs was completed via the translators. That is, the translators were adapted to generate AFOs with the desired change.

2.5 The "VOX" to "a.out" Translator

As previously mentioned, AF is written in Ada and the FTPP Operating System is written in C. The KRAPP project used a VMS based Verdix Ada Development System to compile the AF, a UNIX based Heurikon C System to compile the FTPP procedures, and the Heurikon System to link the resultant object modules. Because the object modules generated by the Verdix System differed from those created by the Heurikon, a conversion process was required and consequently was developed by CSDL.

This conversion process involves three basic steps: (1) converting the Verdix object module format to an intermediate format, (2) transferring these intermediate modules to the Heurikon System, and (3) converting the intermediate object modules to the Heurikon "a.out" modules. The use of an intermediate file format was required to permit the transfer of the files over an Ethernet connection. Step 1 of the process is performed on a MicroVax III Workstation which hosts the Verdix Ada System, while Step 3 is performed on the Heurikon System.

2.6 The Load Balancer

The optimal mapping of a suite of AFOs to a number of VIDs is a complex time consuming process. Under the KRAPP contract, we developed a sub-optimal load balancer. It was written in C on a Sun Workstation with a UNIX Operating System.

The load balancer determines the AFO to VID mapping by minimizing the inter-VID connectivity. The connectivity was selected as the sole basis of distribution, because it could be extracted from the frame file and the initial development of the load balancer was concerned more with functionality than comprehensiveness.

As illustrated in Figure 21, the input to the load balancer is the frame file and the number of VIDs. The balancer parses the frame file, determines the number of AFOs, and generates a connectivity matrix. The basis of the mapping algorithm is an "assign and evaluate" process. That is, an arbitrary AFO to VID distribution scheme is selected, and the inter-VID connectivity is calculated. The distribution is then changed, and the connectivity is recalculated. This distribution-calculation process is continued for a pre-specified number of iterations. Finally, the mapping that minimizes the connectivity is chosen.

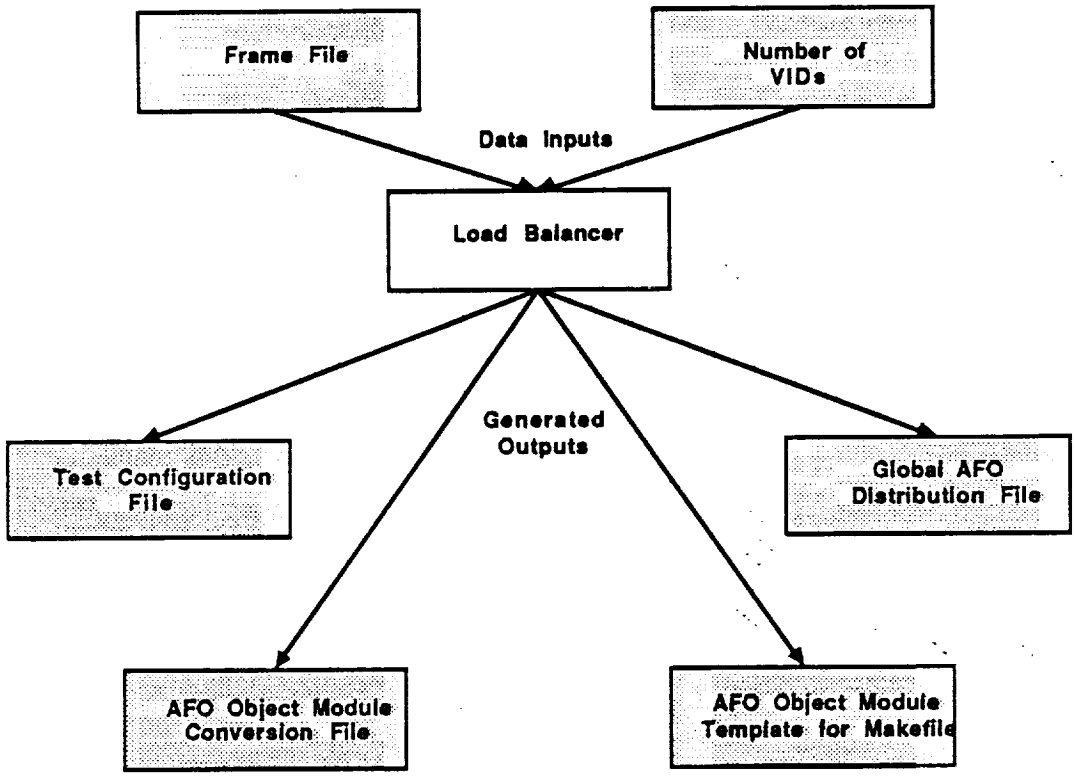

Figure 21- The Load Balancer

The output of the load balancer is four files: a verbose configuration listing, a terse file that depicts the AFO to VID mapping, a file for decoding the intermediate AFO object modules, and a listing of the AFOs that is used to create a Makefile. The configuration listing, called *config.test*, is a readable ASCII file specifying the number of VIDs, AFO's transfer function, relative AFO number, and AFO to VID assignment. The second file, named *global.afos*, is a concise record of the AFO to VID distribution. The latter file is used by the Automatic Load Module Generator to create an FTPP load module.

To ease the third step of the object module conversion process (discussed in Section 2.5), the load balancer generates a UNIX shell script which is the third output file. This script file, called *ud_afos*, is used to convert the intermediate object module format to the a.out format. This program invokes the conversion program for each AFO object module, thus permitting the user to automatically decode the modules rather than perform the process by hand. Similarly, the fourth file, named *afos_for_make*, is created by the load balancer to

facilitate the creation of the Makefile (the Makefile utility is discussed in more detail in Section 2.7).

2.7 The Automatic Load Module Generator

It was desirable to develop a method for automatically generating FTPP load modules, because such a procedure: (1) can be used in conjunction with the load balancer, (2) reduces the probability of mistakes, and (3) provides a friendlier user interface. An Automatic Load Module Generator (ALMG) was written in C. Because of the lack of a unified development system, it consists of two stages, one that resides on a Sun Workstation and the other on the Heurikon System. (Note: this is temporary, resolvable inconvience that resulted because the KRAPP project did not have an Ada compiler for the Sun Workstation; it is not an unavoidable, major drawback of the KRAPP project.)

The FTPP Operating System is composed of many C modules. To record the inter-module dependencies and facilitate the linking process, the UNIX Makefile utility is used. An FTPP programmer uses the Makefile utility to automatically compile C procedures into object modules and link object code into an FTPP load module. Because of the usefulness of this facility, it is a major part of the Automatic Load Module Generator.

Another significant portion of the ALMG is performed by three code generators (illustrated in Figures 22 and 23):

1. The *app_sex.h* code generator uses the *global.afos* file, which is generated by the load balancer, to create a file that specifies the AFOs that are in the application. Only one *app_sex.h* file is required per application.
2. The *lv2_init.c* code generator uses the *global.afos* file to create a function which, when executed, associates the AFO transfer procedures with VRTX tasks. One *lv2_init.c* function is generated per application.
3. The *afo_to_vid.c* code generator uses the *global.afos* file to create a procedure that indicates the AFO to VID mapping. One *afo_to_vid.c* procedure is generated per test.

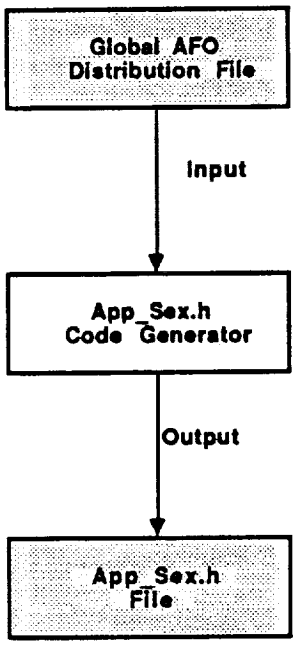

Figure 22 - The App_Sex.h Code Generator

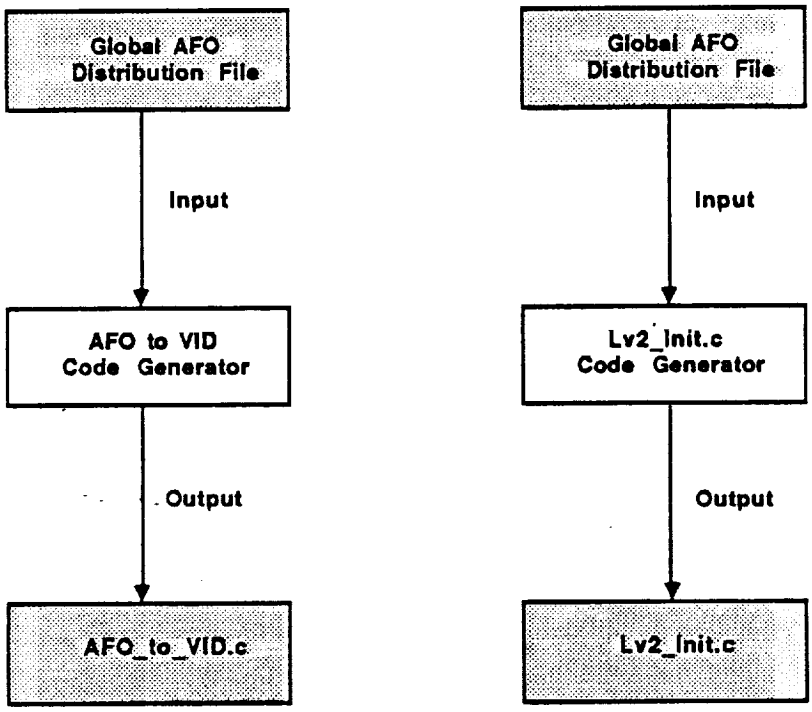

Figure 23 - The AFO_to_VID.c and Lv2_Init.c Code Generators

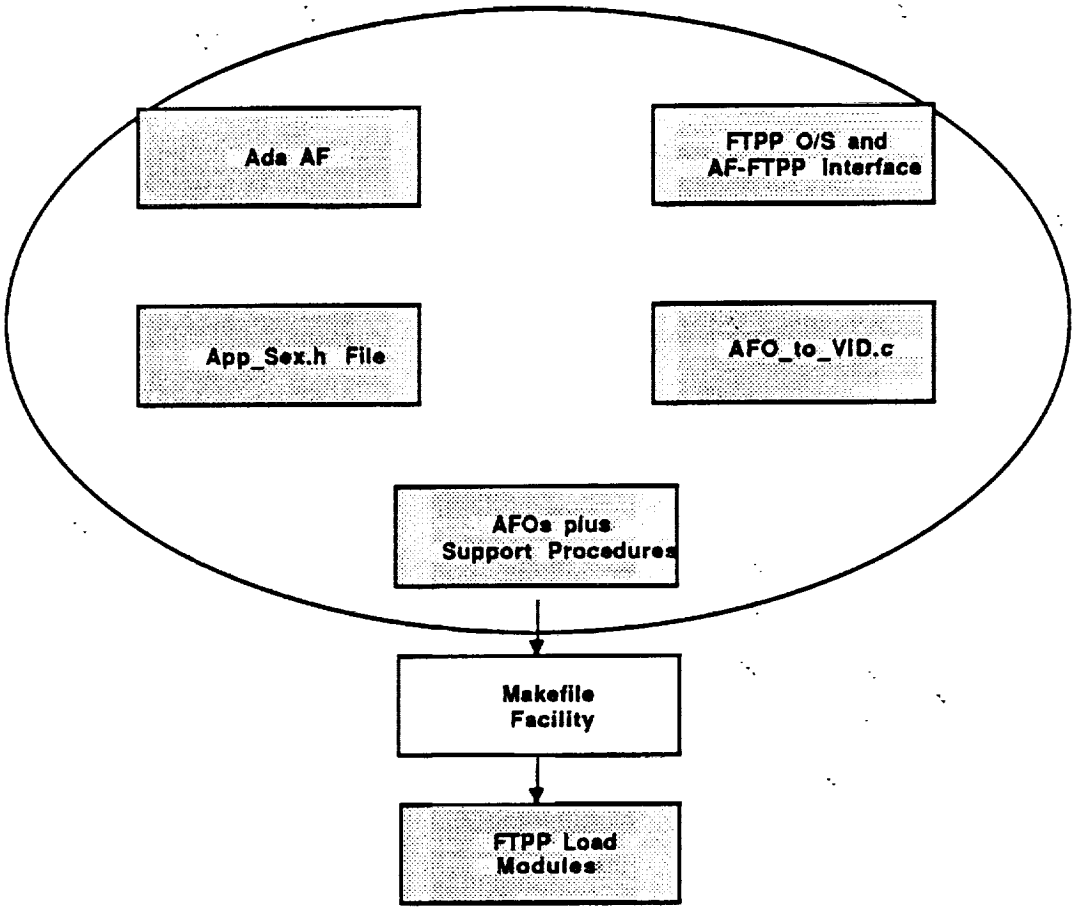

Figure 24 - AF-FTPP Load Module Generation

As stated earlier, the current implementation of the ALMG involves two steps. First, a preprocessing stage, which resides on the Sun Workstation, is executed. This process accepts a frame file and the number of VIDs as input parameters. Subsequently, it invokes the load balancer, the code generators, and then stores the output files in a temporary directory. The second stage involves communication of the output files from the Sun Workstation to the Heurikon System and the creation of an FTPP load module (only one load module is required per test). After the files are stored on the Heurikon, the load module is generated by invoking the Makefile utility (depicted in Figure 24) which links the AF, AFOs, FTPP Operating System, and AF-FTPP Interface into one executable module.

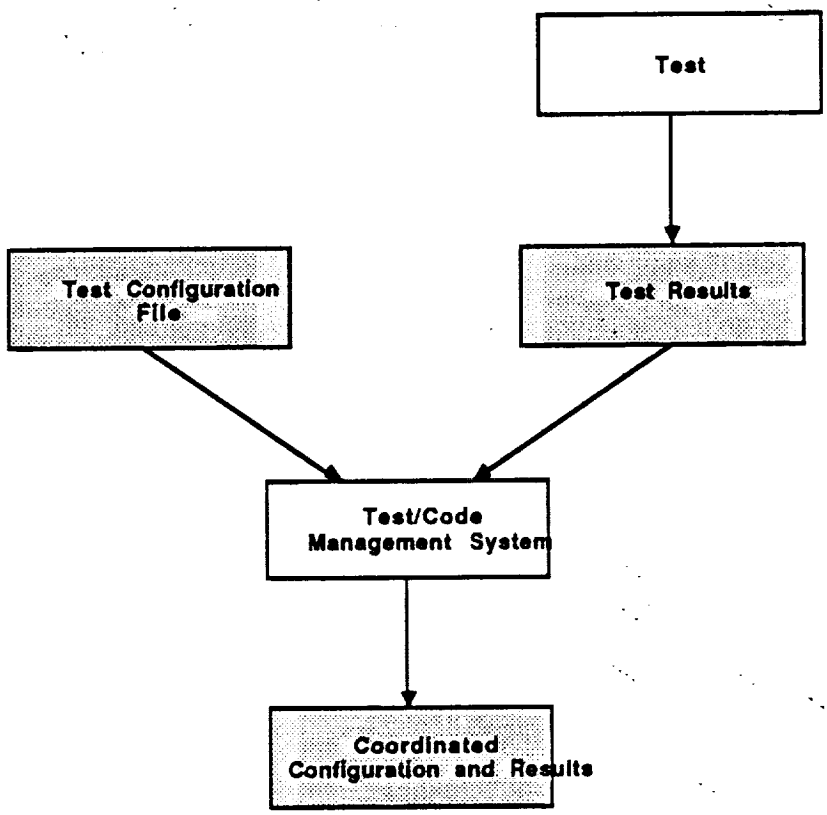

Figure 25 - The Test/Code Management System

After the load module is created, it is downloaded to one or more VIDs. Subsequently, the AF and AFOs are executed on the FTPP, and performance measurements are taken. The metrics, along with AFO to VID configuration, can then be recorded using a test/code management system. An example of such a system is shown in Figure 25.

2.8 Remote Data Insertion and Capture

A method of inserting and capturing messages is necessary for the execution of a real-time complex intelligent system. The method suggested by CSDL utilizes a Sun Workstation as a remote source/destination. In addition, this process uses a Motorola MVME 147 board (30 Mhz. 68030 processor with 4 Mbytes of RAM) as a gateway controller. The gateway software to support the data insertion and capture will reside on the Sun, the 147 board, and a gateway VID.

The data insertion process involves several steps. First, the source messages must be placed in a file on the Sun Workstation. Second, the gateway software, which resides on the Sun, has to be downloaded to the 147 board. Third, the message file is transferred to the 147 board, and the gateway software is started. Last, the AF and the FTPP Operating System are executed allowing the insertion of messages.

The data capture process also involves multiple stages. The messages, whose destination is the Sun Workstation, are tagged, and sent to the gateway VID. The AF-FTPP Interface on the gateway VID, when noticing that the messages are tagged for the Sun, sends them to the 147 board. The gateway software on the 147 board stores the messages in a file. When the run is complete, the message file is uploaded to the Sun for subsequent analysis.

The current design of the message insertion process uses a predefined location on a gateway VID to transfer the messages. The messages are sent one at a time, and a simple handshaking process is used to coordinate the communication. Specifically, the gateway software on the 147 board checks the handshake flag and if it is reset, sends a message and sets the flag. The communication software on the VID polls the flag to detect the presence of a message. If the flag is set, the input message is sent to the destination AFO, and then the flag is reset.

The current design of the data capture procedure also involves a handshaking protocol. The software on the gateway VID checks a handshake flag (a different flag than the one for data insertion) and if it is reset, stores the outgoing message in a reserved memory location on the VID and sets the flag. The 147 board polls the flag and when it is set, reads the message. The 147 board then clears the flag, adds the message to the destination file, and again queries the VID for another message.

The data insertion gateway software for the Sun Workstation and the 147 board has been designed, implemented, and debugged. The corresponding communication software for the FTPP is designed but needs to be implemented. Further, the data capture gateway software has been designed, but it has not been written.

3.0 Development Environment

The aim of the development system is to create an FTPP load module which incorporates the FTPP Operating System, the Activation Framework, the AF-FTPP Interface, and the Activation Framework Objects which represent the application's rule set (that is, Horn clauses). Because of the lack of a single development system which hosts all the software tools necessary to accomplish this task, the creation of an FTPP load module requires the use of four loosely coupled computer systems. Consequently, this overall task has been divided into four phases closely corresponding to the four computer systems involved:

Phase	Computer system
Translation to AFOs	VAX 8650
Compilation of AFOs and AF	MicroVAX III
Load Balancing	SUN 3 workstation
Load Module Creation	Heurikon UNIX

Figure 26 describes diagrammatically the sequence of operations spanning the various computer systems.

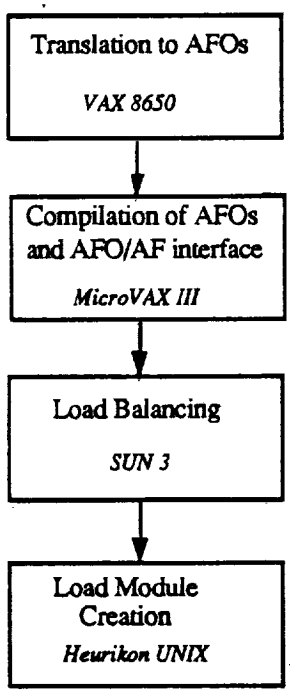

Figure 26 - Development System Overview

3.1 Translation to AFOs

The application is represented by a set of Horn clauses which describe inter-AFO connectivity and data dependencies. The translation of these rules to AFOs is a two-phase process which creates the AFOs and AFO/AF interface modules in the Ada programming language. The two translators involved (*Rules to EFG* and *EFG to AFOs* referred to in section 2.1) are written in Ada as well. These translators therefore must be compiled and executed on a computer system which hosts an Ada compiler and runtime environment. The VAX 8650 system was selected for this phase of development because of the availability of the native VAX Ada Development System.

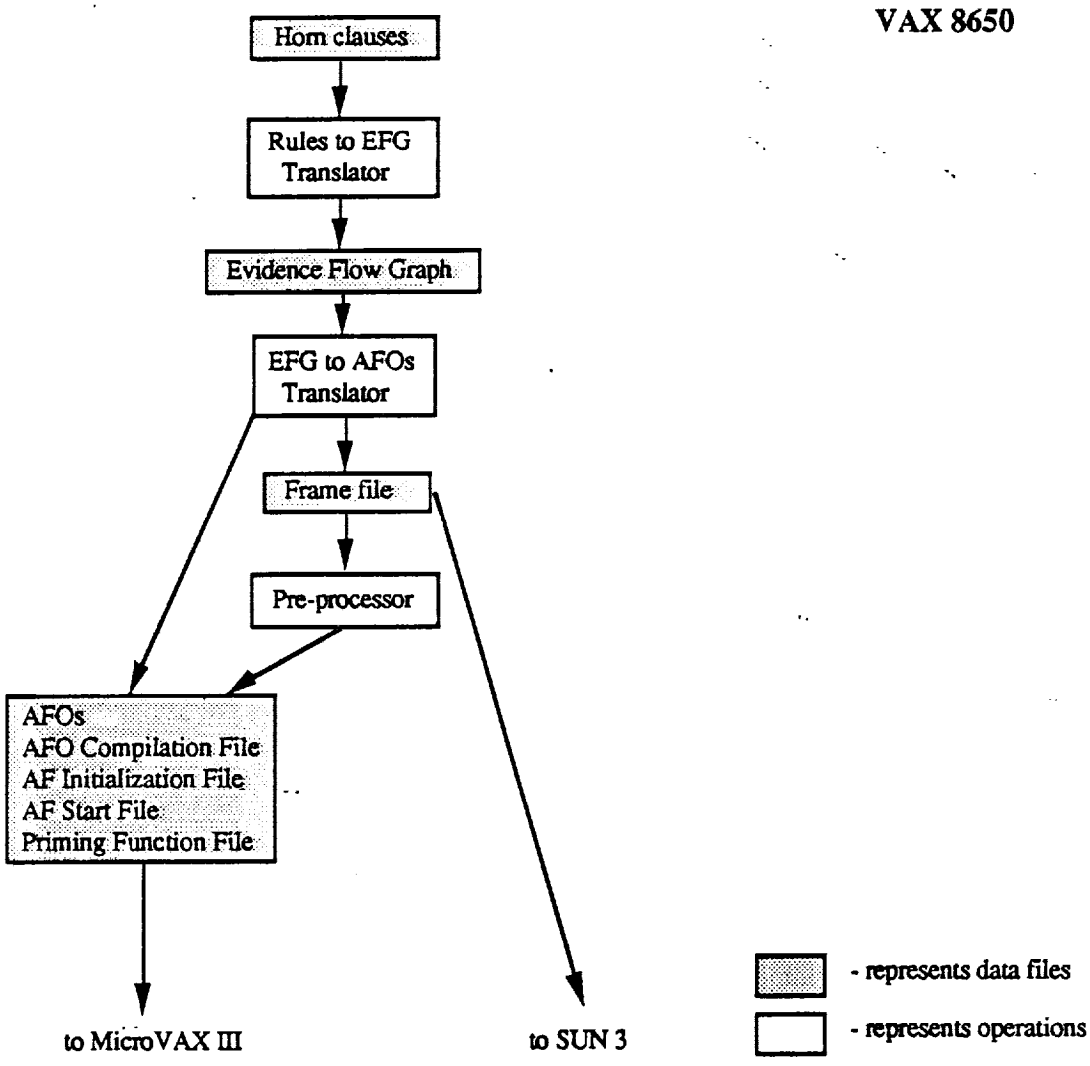

Figure 27 - Translation to AFOs phase

3.2 Compilation of AFOs and AFO/AF Interface Modules

Since the AFOs and the AFO/AF interface modules are targeted for execution on the FTPP (a 68020 based system), an Ada compiler is required which generates object modules suitable for execution on a 68020 processor. However, due to the lack of a cross compiler on the VAX 8650, these modules are transferred to the MicroVAX III. This system has the Verdix Ada Development System which is a cross-compiler targeting the 680x0 class of processors. In addition, to automate the compilation process a command file is also generated in parallel with the generation of the AFOs and AFO/AF interface files during the "translation to AFOs" phase. This command file invokes the Ada compiler for each module and converts the object module to an intermediate format. This format conversion serves the dual purposes of effecting an efficient form for file transfer and of partially translating the object module format from VOX format to UNIX System 5 "a.out" format.

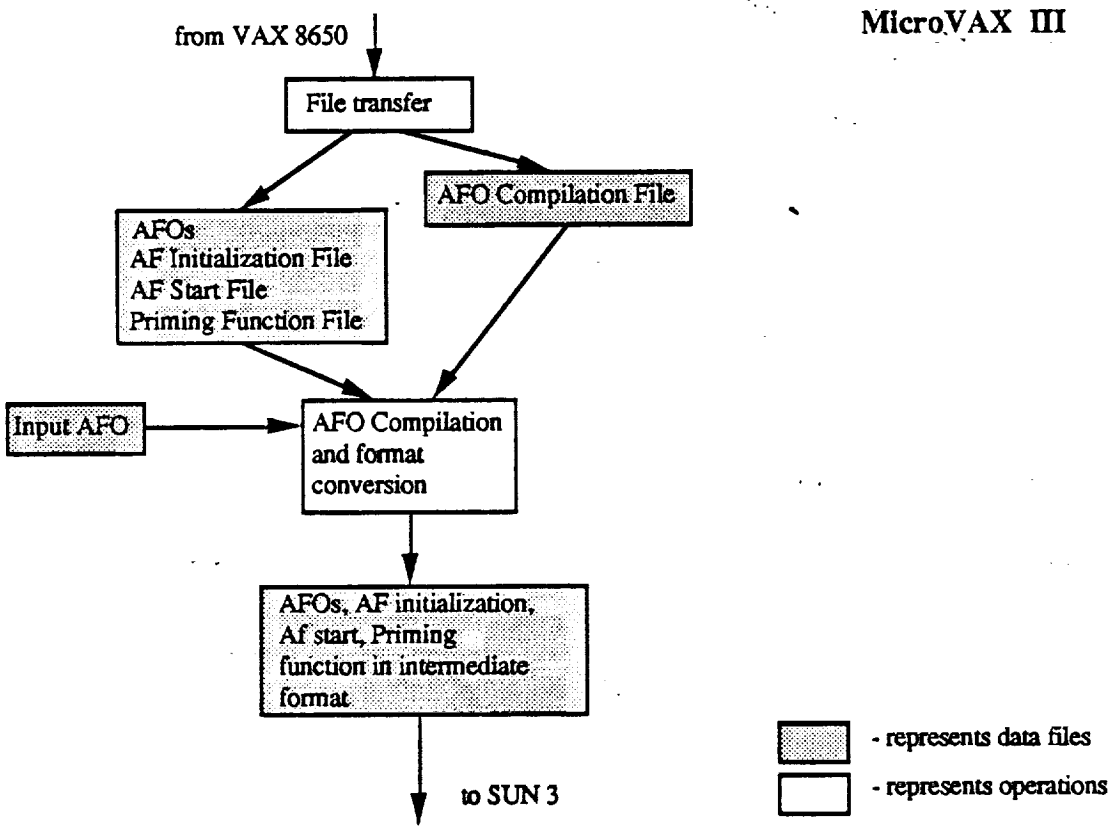

Figure 28 - Compilation of AFOs and AFO/AF interface phase

3.3 Load Balancing

The load balancer has been implemented on the SUN 3 workstation in an effort to eventually unify FTPP development. The frame file generated on the VAX 8650 system is transferred to the SUN 3 for inclusion in the Automatic Load Module Generator (ALMG). The ALMG generates three C source files to allocate AFOs to VIDs, a command file for decoding the intermediate Ada objects, and a makefile template for inclusion of the AFO objects in the load module. These file are transferred to the Heurikon UNIX system. The SUN workstation also operates as the gateway for transferring the encoded AFOs and AFO/AF interface object modules to the Heurikon UNIX system.

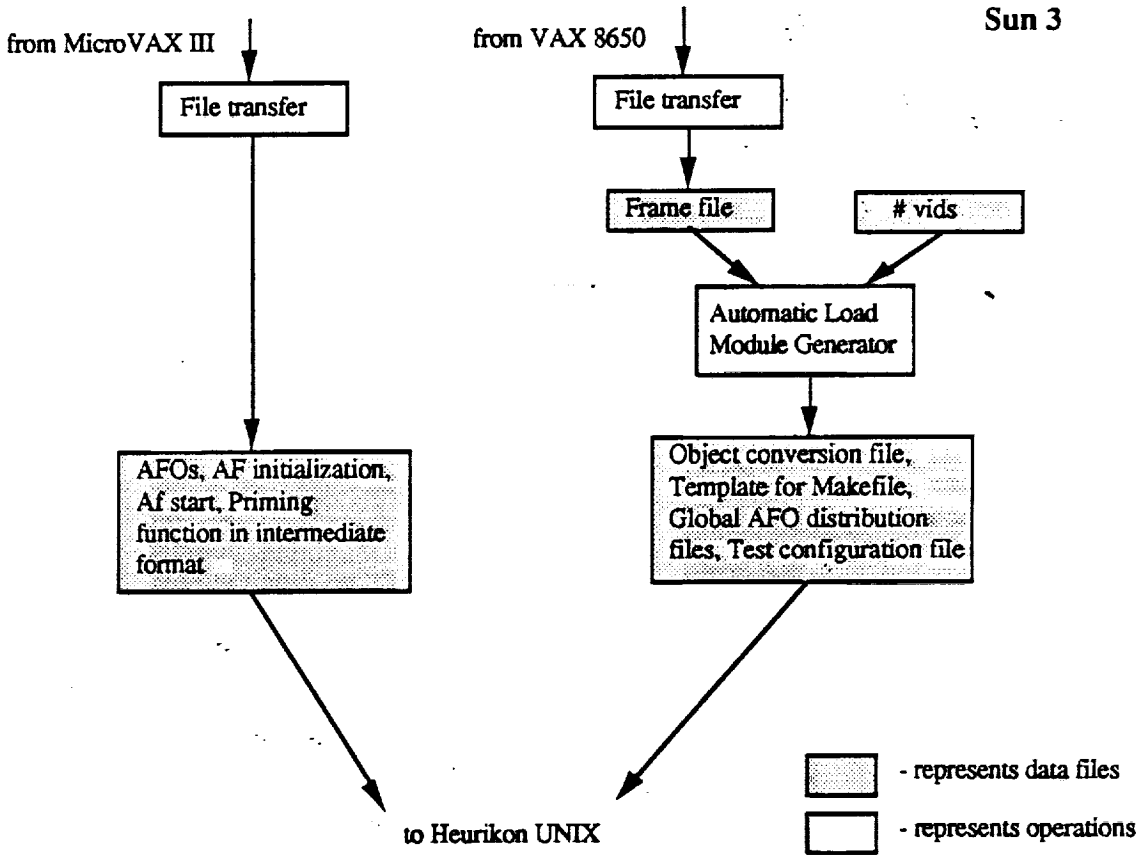

Figure 29 - Load Balancing phase

3.4 Load Module Creation

The Heurikon UNIX C compiler and linker consolidate all the object files to generate an executable load module for the FTPP. However, before actual invocation of the linker, the AFOs and AFO/AF interface object modules must be decoded using the object conversion command file, and the makefile must incorporate the template to include the AFOs and AFO/AF interface object modules in the load module. Execution of the UNIX makefile facility compiles the C modules generated by the load balancer and links the AFOs, AFO/AF interface modules, and the AF object modules with the FTPP operating system object modules to generate the load module.

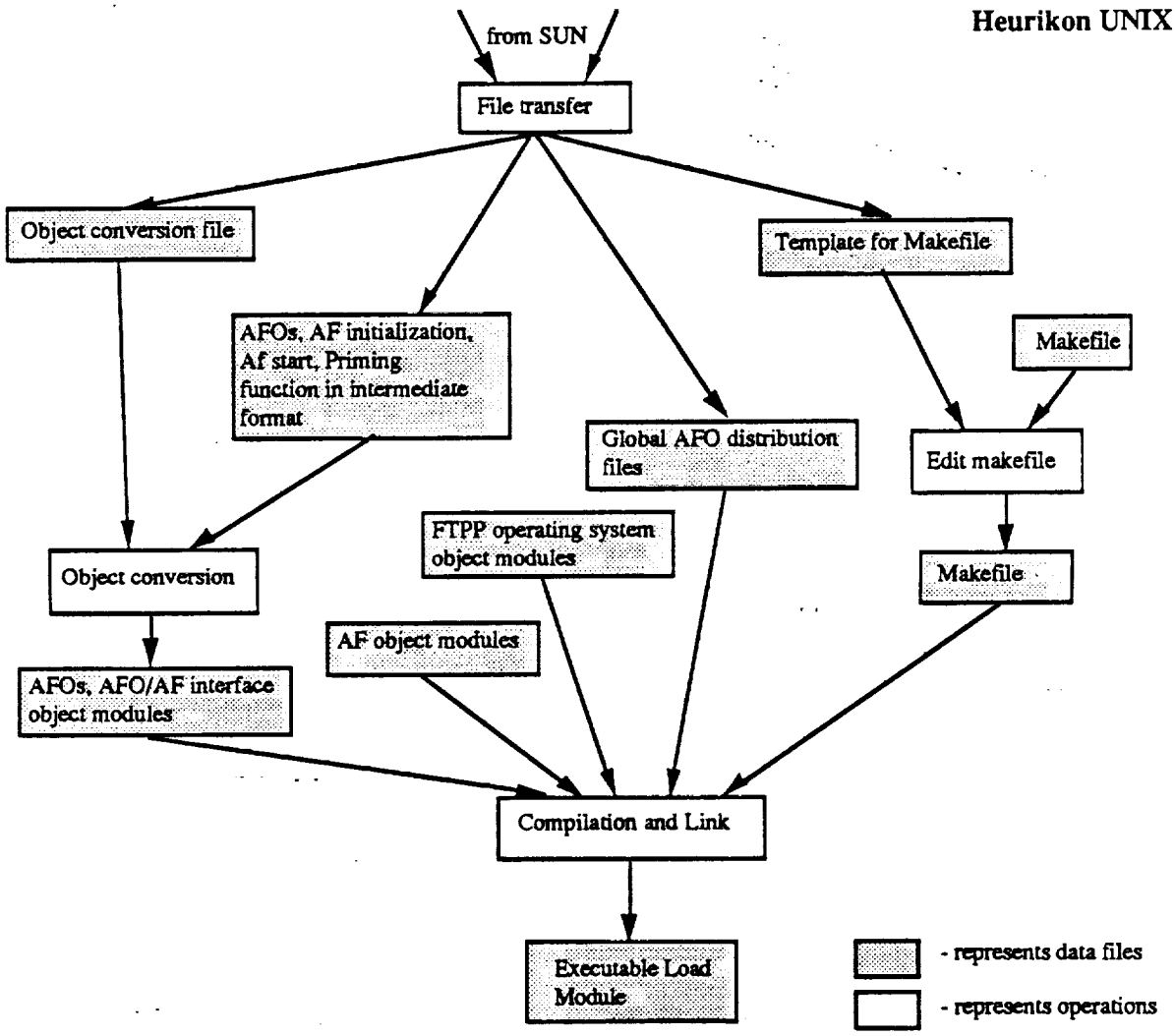

Figure 30 - Load Module Creation phase

4.0 Performance Measurements

The execution of the Event Diagnosis Expert on the FTPP was evaluated for several reasons. First, performance measurements were recorded to determine the speedup that can be attained by utilizing multiple processors. Second, they were taken to identify improvements that will enable more efficient execution of the Activation Framework. Third, performance metrics were used to examine the effect that different load allocations schemes had on the execution of the application. Last, they were employed to determine the effect of different AFO computational loads.

In addition to examining the performance of the Event Diagnosis Expert, we used the AF Methodology and AF-FTPP utilities to execute a data flow structure for a Real Time Controller. Similar to the Event Diagnosis Expert application, the AF implementation of this Controller generates an AFO for each Horn clause. However, the Controller application requires 57 AFOs rather than the mere thirteen AFOs needed by the Expert. As a result, the FTPP is stressed more executing the Real Time Controller AFOs than it is hosting the Event Diagnosis Expert.

4.1 Preliminary Performance Measurements

It was desirable to dissect the AF to discover where the bulk of the processing time was being spent. To accomplish this preliminary task, we used the Network Element Simulator (NESIM) and one to three PEs to measure numerous intervals within the AF, AFOs, and AF-FTPP Interface.

The preliminary performance measurements of the AF and AFOs recorded 38 procedures. The intervals that were examined varied from the low level *malloc* routine (which performs memory allocation) to the total Event Diagnosis Expert's execution time. A sample set of the measurements that were recorded is given in Table 2. For each measurement, four items were obtained: the interval identifier, the number of invocations, the average time required to execute the interval, and the cumulative time (the number of invocations multiplied by the average time per invocation). The interval ID was retained to coordinate the measurement with the procedure being timed. The number of invocations was recorded

Interval ID	Procedure	Num. of Calls	Time/Call (ms.)	Total Time (ms.)
2	malloc	142	0.7	96
12	tsk_loop	20	222	4437
14	afo_init	14	5.7	80
16	port_cr	26	12	311
18	snd_obj	33	6.3	208
38	total execution time	1	7383	7383

Table 2 - Example of Preliminary Measurements

to determine the routines that are called most frequently. Additionally, the time per invocation and cumulative time were measured to quantify the performance and to identify the bottlenecks.

The measurements taken during this evaluation differed from those obtained exercising the actual FTPP hardware (the latter measurements are presented in Section 4.3). These differences resulted primarily because this analysis used the NESIM. The NESIM supports the inter-VID communication via a software program executing under Unix whereas the FTPP uses dedicated hardware. Because the execution of the NESIM program is time sliced by Unix, the inter-VID communication overhead due to the NESIM is significantly larger than that required by the FTPP. Consequently, the time required to execute AF-FTPP processes that involve inter-VID communication was substantially larger using the NESIM than utilizing the FTPP hardware.

Another reason why the performance measurements recorded using the NESIM are different than those measured employing the FTPP hardware is that in the former case, the AF-FTPP Interface was logging debug information during its execution. When evaluating the AF using the hardware, we recognized that this debug information was hindering the system's performance. Consequently, this logging process was removed for all of the AF-FTPP evaluations that employed the FTPP hardware. As a result, the performance measurements recorded utilizing the FTPP are substantially smaller than those taken using the NESIM.

Since the NESIM overhead is significant and debug logging was performed by the AF-FTPP Interface, these times were, and should only be, used to determine where the bulk of the execution time resides and to evaluate the performance gains attained by incorporating enhancements (performance gains are discussed in Section 4.2). These preliminary measurements are valid means for comparing the performance gains, because the inter-VID communication overhead can be determined and subsequently excluded.

As stated earlier, this analysis involved the NESIM and one to three PEs. It was performed to determine which AF, AFO, and AF-FTPP Interface procedures could be improved. A complete list of the resultant measurements is presented along with a description of the procedures instrumented in Appendix C.

4.2 Enhancements in the AF and AFOs

The preliminary performance measurements were analyzed to determine where the majority of the time is being spent. This initial examination addressed the main "bottlenecks" and "time sinks" rather than comprehensively itemize the areas that need optimization.

Three *major* inefficiencies were identified during this evaluation:

1. The *port_num* procedure, which locates a port entry in the port table based on the port name, was needlessly called during each execution of the AFO priming and transfer functions.
2. The AFO priming function unnecessarily checks all of the input ports each time it is called. If one port does not have a message, then the AFO is not primed and the other ports do not have to be checked.
3. The AF spends a large portion of time determining the next AFO to execute.

Each inefficiency was examined, and a corresponding enhancement was designed and implemented. Specifically,

1. The EFG to AFO translator and AF preprocessor were modified to incorporate AFO initialization procedures. During the AF startup process, an initialization procedure is invoked for each AFO in the application. These

procedures use the *port_num* function to determine the appropriate port identifiers (IDs). These port IDs are recorded in variables that are visible to the priming and transfer functions. Since the port ID assignments are performed during the initialization process rather than during the steady-state execution of the AF, the time required to execute the priming and transfer functions is substantially reduced.

2. The EFG to AFO translator was modified to use the Ada "and then" construct in the priming conditions (for AFO priming functions involving multiple conditions). As a result, the number of port checks is minimized.
3. The "polling" scheduler used in the AF was replaced with a message driven scheduler. Consequently, the overhead required to determine the next AFO to execute is greatly reduced.

The performance measurements of the enhanced AF and AFOs used the Network Element Simulator and timed 40 procedures. In addition to the 38 intervals recorded in Section 4.1, this analysis measured the AFO initialization time and the steady-state execution time (see Section 4.3 for the boundaries for these intervals). As before, this examination involved one, two, and three PEs. This analysis was performed to determine the performance gains that were attained by incorporating the previous modifications.

As discussed in Section 4.1, the communication overhead due to the NESIM is significant. Nevertheless, these measurements can be used to approximate the performance gains attained by incorporating our enhancements, because this overhead was determined and extracted. Consequently, after the AF was optimized, its performance increased by 258, 293, and 209 percent for one, two, and three PEs, respectively (illustrated in Figure 31).

Like the preliminary measurements in Section 4.1, a complete list of the measurements recorded during this analysis is presented in Appendix C.

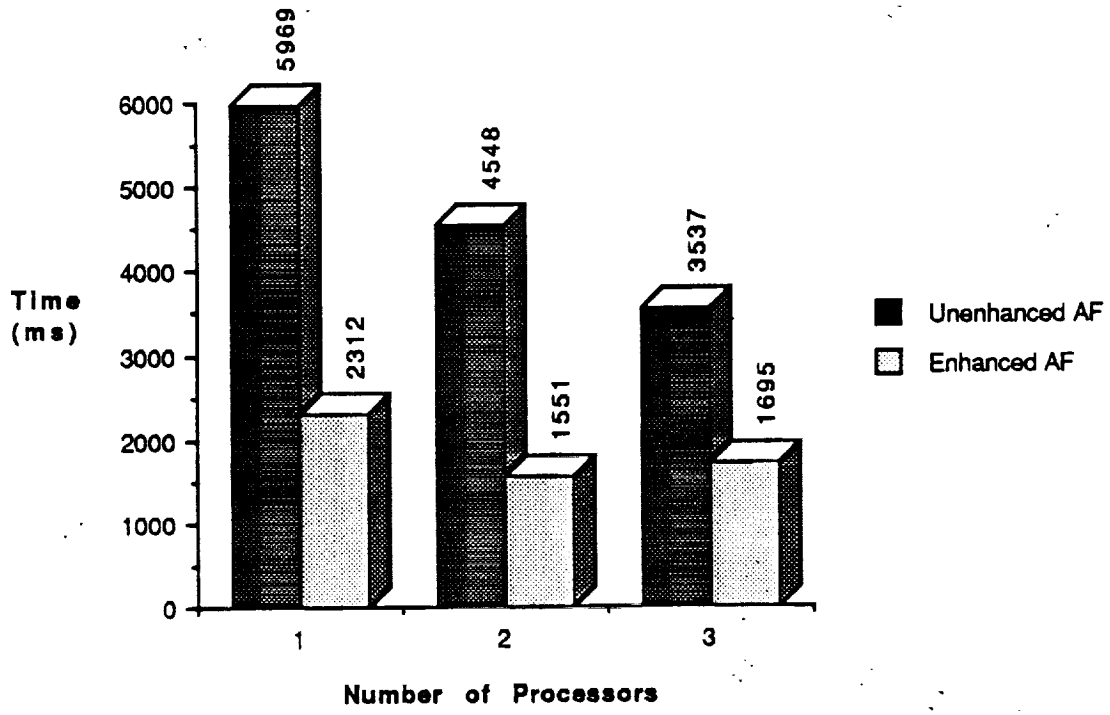

Figure 31 - Execution Times Required by the AF Versions

4.3 Performance of the Event Diagnosis Expert

The Event Diagnosis Expert was executed on one to thirteen simplex VIDs. Three times were recorded, specifically: (1) the time necessary to initialize the Ada AF, AFOs, and AF-FTPP Interface, (2) the time from the completion of the initialization process to the end of the AFOs execution, and (3) the total time required to execute the Event Diagnosis Expert. The first timing essentially measures the overhead required to create the AFOs (the *af_run* procedure) and to send the necessary set of initialization messages. The second timing starts when the scheduler first checks for a primed AFO and concludes when the *System_Output* AFO has successfully processed its last message. This interval is referred to as the steady-state time. Finally, the third measurement is the sum of (1) and (2).

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	775	553	1328
2	766	374	1140
3	766	299	1065
4	766	261	1027
5	654	273	927
6	654	252	906
7	750	224	974
8	766	213	979
9	654	241	895
10	702	233	935
11	638	241	879
12	638	256	894
13	718	218	936

Table 3 - Performance of the Event Diagnosis Expert Utilizing the Load Balancer

The performance metrics obtained during this analysis are presented in Table 3. The AFO to VID mapping for this test was generated using the load balancer. As stated earlier, the load balancer determines the distribution by minimizing the inter-VID connectivity. By examining Table 3, it can be seen that, as expected, the initialization times remain relatively constant. These times fluctuate slightly, because the AFO to VID distributions differ and the timing facility has a granularity of 16 milliseconds. Conversely, the steady-state times noticeably vary since the work is distributed to multiple processors. Further, these times are not monotonic, because the load balancer typically generates a sub-optimal distribution. For instance, the time required to execute the Event Diagnosis Expert increases when shifting from four to five VIDs. This occurs because the load balancer's distribution for four VIDs is better than that for five VIDs.

The speedup attained by distributing the workload is presented in Table 4. These numbers represent the speedup of the steady-state execution time. The AF initialization time was ignored in the speedup calculation, because the steady-state time will typically dominate the execution time in a real application (the initialization process is only performed once

Number of VIDs	Relative Speedup
1	1
2	1.48
3	1.85
4	2.12
5	2.03
6	2.19
7	2.47
8	***2.60***
9	2.29
10	2.37
11	2.29
12	2.16
13	2.56

Table 4 - Steady-State Speedup for the Event Diagnosis Expert

whereas the Event Diagnosis Expert AFOs will be executed many times over the course of a mission). It can be seen in Table 4 that the maximum speedup achieved is 2.60. This speedup results when the load of the system is allocated to 8 VIDs.

The speedup of the Event Diagnosis Expert is graphed in Figure 32. Additionally, in Figure 33, a monotonic logarithmic curve fit is superimposed over the speedup plot to illustrate that the speedup curve approaches a plateau. The speedup eventually stabilizes, because the application's synchronization dependencies counteract the additional throughput capability.

In general, the speedup that was attained is relatively small. Nevertheless, such a speedup was expected after considering the Event Diagnosis application. A minimal speedup was speculated, because it is nearly impossible to achieve a significant speedup when: an application involves such a small number of tasks (13), the computational load of each task is approximately equal to the time required to schedule a task, and one of the primary tasks is a bottleneck (*System_Output* AFO).

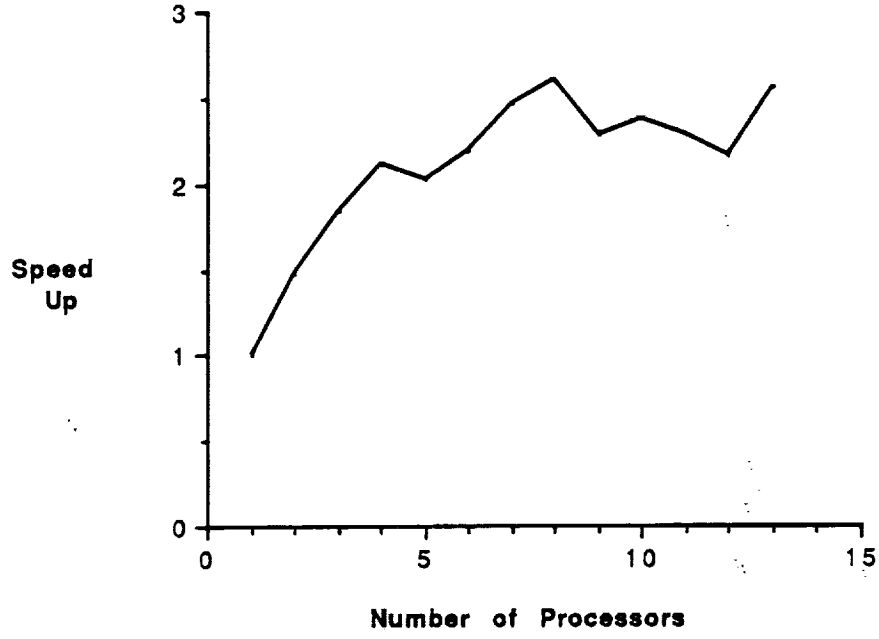

Figure 32 - Steady-State Speedup for the Event Diagnosis Expert

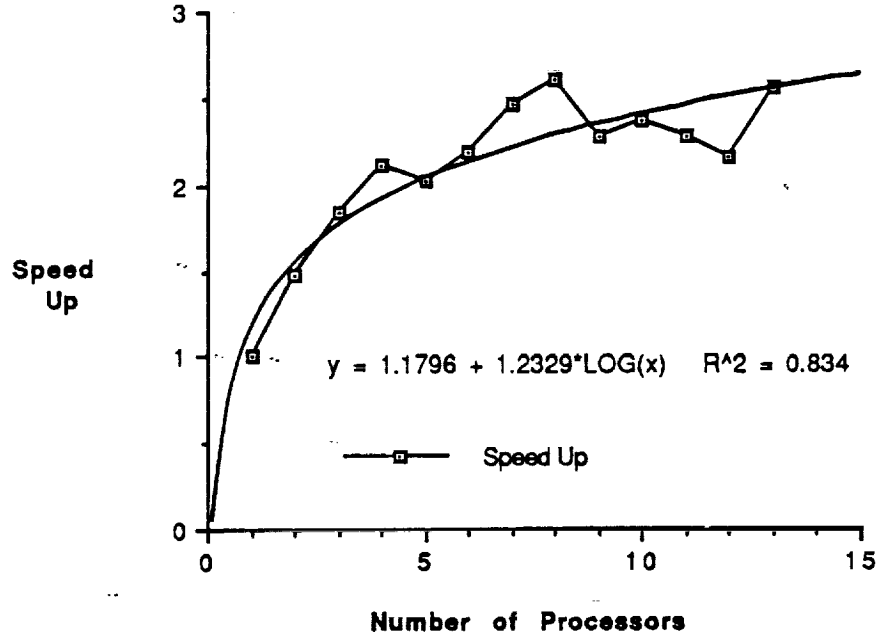

Figure 33 - Steady-State Speedup with Logarithmic Approximation

With respect to the distribution of an intelligent system, speedup is not the only concern. Another important issue is the effect that parallelization has on the output of the system. This issue was examined when hosting the Event Diagnosis Expert on the FTPP. When executed on a single processor, the Expert had a deterministic output. Alternatively, when parallelized, the output became non-deterministic and typically varied with the number of processors used. However, since output ordering constraints were not provided, each output sequence was deemed correct.

4.4 Performance of the Event Diagnosis Expert with Simulated Computational Load

As discussed earlier, the increase in the performance of the Event Diagnosis Expert that was attained by employing multiple processors of the FTPP was unimpressive. We speculated that this resultant speedup would be minimal, partly because the computational load of each AFO is small. Accordingly, we attempted to verify its theory by giving each AFO a uniform computational load.

The 68881 floating point multiplier on each PE was used to simulate an AFO computational load. Specifically, each AFO, except the *System_Output*, was modified to execute a series of floating point instructions prior to its execution. The number of instructions was varied to differ the size of the load. A computational load was not added to the *System_Output* AFO in an attempt to reduce its bottleneck effect (which is currently inherent to the application). The load balancer was used to generate an AFO to VID distribution, and the "loaded" Event Diagnosis Expert was executed on one to thirteen PEs.

The AFOs were given uniform loads of 32 ms., 70 ms., and 150 ms. As speculated, the resultant speedup of the Event Diagnosis Expert improved as the computational load of the AFOs was increased. For example, when the Expert was allocated to ten VIDs, the speedup was 2.37, 3.24, 4.03, and 4.91 for the 0 ms., 32 ms., 70 ms., and 150 ms. loads respectively. A complete list of the resultant performance measurements is provided in Tables 5 through 8. Further, a comparison of the speedup attained through distributing the application is presented in Table 9 and illustrated in Figure 34.

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	775	553	1328
2	766	374	1140
3	766	299	1065
4	766	261	1027
5	654	273	927
6	654	252	906
7	750	224	974
8	766	213	979
9	654	241	895
10	702	233	935
11	638	241	879
12	638	256	894
13	718	218	936

Table 5 - Performance of the Event Diagnosis Expert

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	775	916	1691
2	750	534	1284
3	750	411	1161
4	766	337	1103
5	638	355	993
6	638	321	959
7	750	272	1022
8	750	275	1025
9	654	301	955
10	686	283	969
11	654	275	929
12	654	289	943
13	702	256	958

Table 6 - Performance of the Event Diagnosis Expert Additional Load per AFO = 32 ms.

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	775	1401	2176
2	750	758	1508
3	750	603	1353
4	750	531	1281
5	654	487	1141
6	654	409	1063
7	766	371	1137
8	750	346	1096
9	654	376	1030
10	686	348	1034
11	654	361	1015
12	638	348	986
13	702	297	999

Table 7 - Performance of the Event Diagnosis Expert Additional Load per AFO = 70 ms.

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	775	2377	3152
2	750	1174	1924
3	750	982	1732
4	750	862	1612
5	638	757	1395
6	638	604	1242
7	750	587	1337
8	766	506	1272
9	638	537	1175
10	702	484	1186
11	638	476	1114
12	638	481	1119
13	702	368	1070

Table 8 - Performance of the Event Diagnosis Expert Additional Load per AFO = 150 ms.

Number of VIDs	Speedup No Extra Load	Speedup Extra 32 ms.	Speedup Extra 70 ms.	Speedup Extra 150 ms.
1	1	1	1	1
2	1.48	1.72	1.85	2.0
3	1.85	2.23	2.32	2.42
4	2.12	2.72	2.64	2.76
5	2.03	2.58	2.88	3.14
6	2.19	2.86	3.43	3.94
7	2.47	3.37	3.78	4.05
8	**2.60**	3.33	4.05	4.70
9	2.29	3.04	3.73	4.43
10	2.37	3.24	4.03	4.91
11	2.29	3.33	3.88	4.99
12	2.16	3.17	4.03	4.94
13	2.56	**3.58**	**4.72**	**6.46**

Table 9 - Steady-State Speedup for the Event Diagnosis Expert

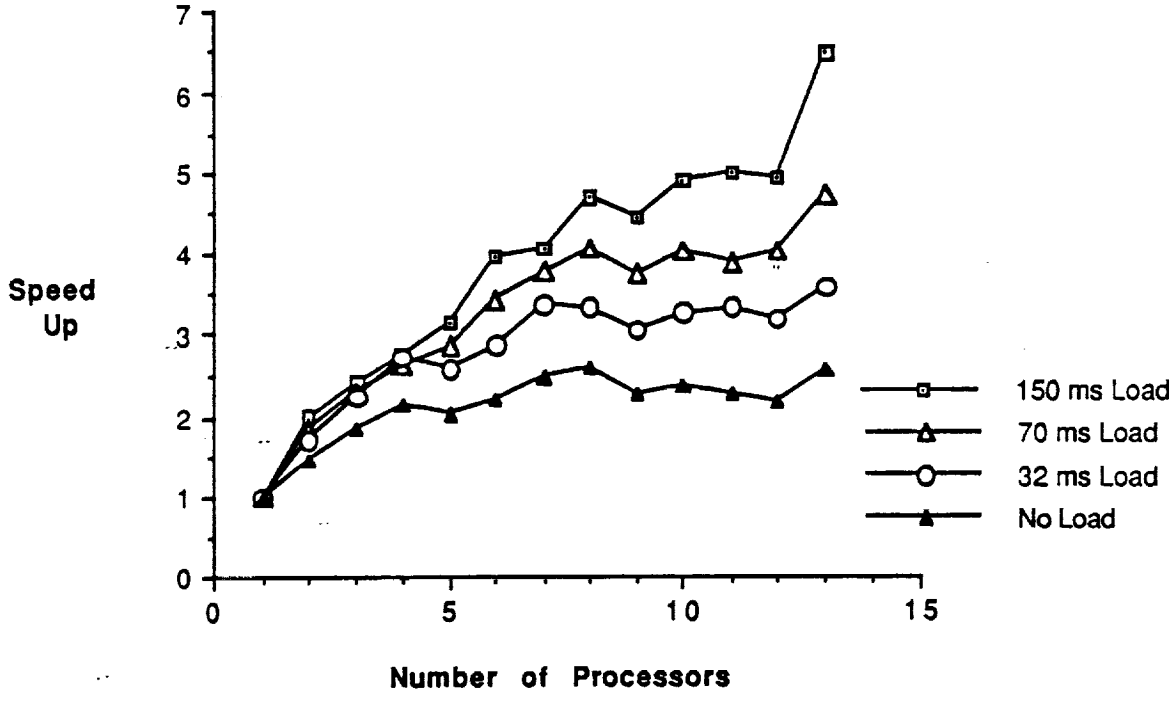

Figure 34 - Speedup Comparison for Differing Loads

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	775	2377	3152
2	750	1174	1924
3	750	982	1732
4	750	862	1612
5	638	757	1395
6	638	604	1242

Table 10 - Performance of the Event Diagnosis Expert Additional Load per AFO = 150 ms. Distributed via Load Balancer

The results of this loading analysis implies that the performance of a parallelized AFO suite will improve if each AFO represents multiple rules rather than only a single rule. Accordingly, it is CSDL's belief that methods by which multiple rules can be grouped and characterized using one AFO should be explored.

4.5 Performance of the Event Diagnosis Expert Using a Hand Generated Distribution

As discussed in Section 2.6, the AFO to VID distribution generated by the load balancer is sub-optimal. This implies that considerations other than connectivity should incorporated into the load allocation algorithm. To support this speculation, a set of mappings were constructed by hand utilizing knowledge of: (1) the connectivity, (2) the Event Diagnosis Expert application, and (3) the implementation of the Activation Framework. The Event Diagnosis Expert was executed on one to six VIDs using these hand coded mappings.

The performance of the Event Diagnosis Expert, augmented to include a 150 ms. simulated load and utilizing the *load balancer*, is shown in Table 10. The 150 ms. load was arbitrarily selected for this illustration. The resultant execution times for this application using the *hand coded* distribution is presented in Table 11, and a comparison of the steady-state speedup is given in Table 12 and Figure 35. It can be seen that, regardless of the number of VIDs, the performance of the Expert using the hand coded distributions is equal to or better than that employing the automated mapping. In conclusion, this evaluation implies that an AFO to processor allocation algorithm based solely minimizing the inter-processor traffic will typically generate a sub-optimal distribution.

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	775	2377	3152
2	750	1174	1924
3	750	846	1596
4	638	731	1369
5	638	544	1182
6	654	527	1181

Table 11 - Performance of the Event Diagnosis Expert Additional Load per AFO = 150 ms. Distributed via Hand Calculation

Number of VIDs	Speedup Load Balancer	Speedup Hand Calculation
1	1	1
2	2.0	2.0
3	2.42	2.81
4	2.76	3.25
5	3.14	4.37
6	3.94	4.51

Table 12 - Load Balancer vs. Hand Calculation Speedup Comparison

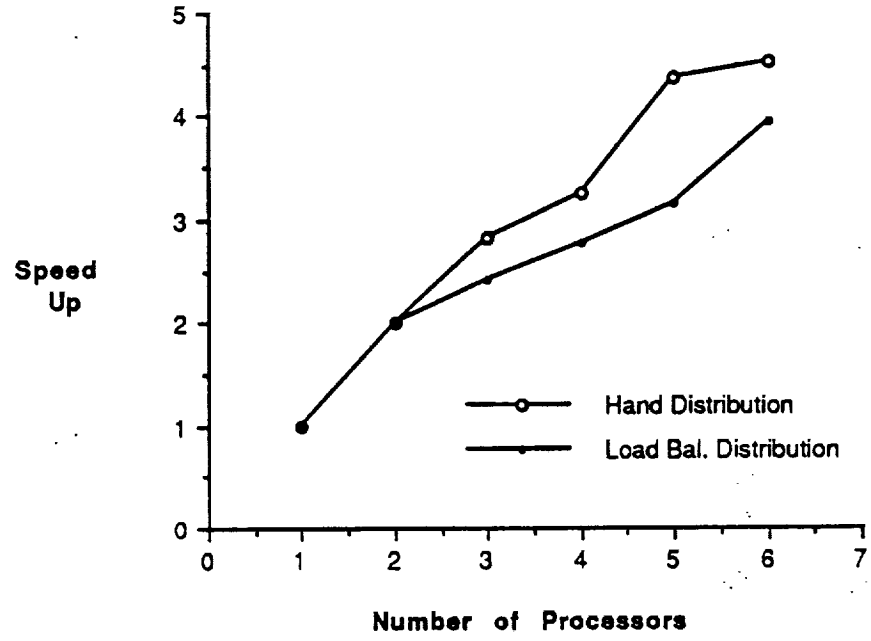

Figure 35 - Load Balancer vs. Hand Distribution Speedup Comparison

4.6 Performance of the Event Diagnosis Expert Using the Dependency Load Balancer

The automatic load balancing algorithm discussed until now is based on minimizing the inter-VID connectivity. As concluded in Section 4.5, this method usually generates sub-optimal AFO to VID allocation. In an attempt to improve the automatic load distribution mechanism, an allocation methodology founded on the AFO priming conditions was designed. To distinguish between these two distribution algorithms in the following discussion, this new mapping utility will be termed the "dependency load balancer", whereas the distribution algorithm described in Section 2.6 will be referred to as the "connectivity load balancer".

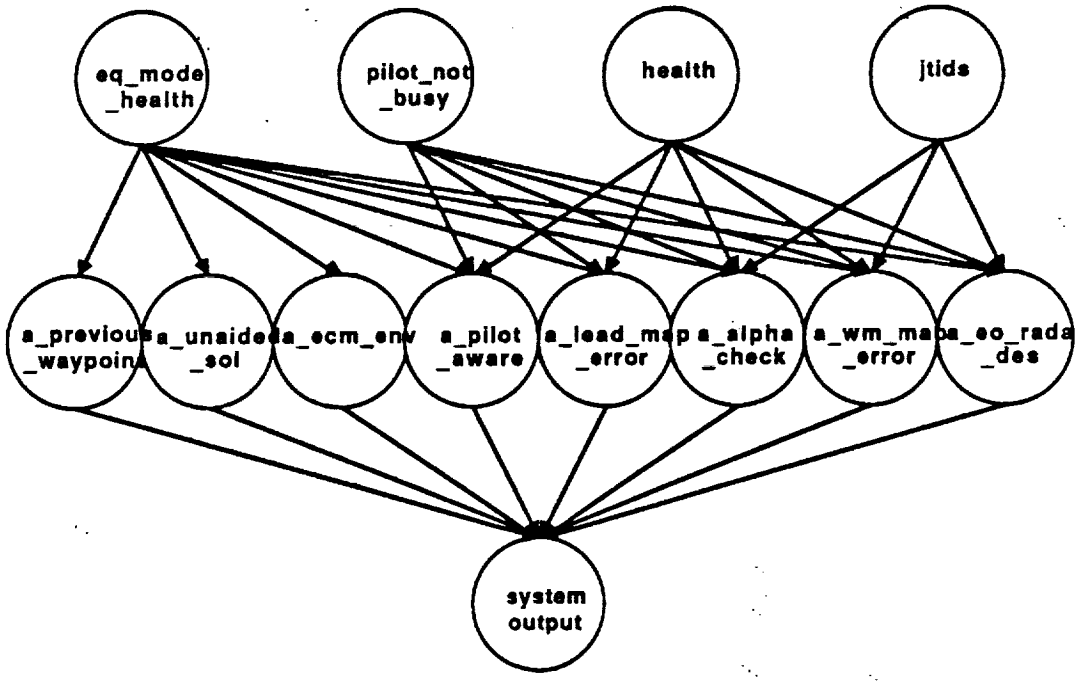

Figure 36 - The Event Diagnosis Expert

One of the limitations of the connectivity load balancer is that it does not consider the AFO priming conditions. Since an AFO's priming conditions must be fulfilled before it can be executed, these prerequisites should be considered when determining an AFO to VID allocation. To examine how useful the priming conditions are for selecting an allocation scheme, we developed the "dependency based" load balancer. This balancer derives its AFO to VID mapping solely from the AFO priming conditions. To give an example of how this method works, we consider the *a_pilot_aware* AFO of the Event Diagnosis Expert (illustrated in Figure 36). This AFO's priming conditions require that, prior to it being scheduled, it receive a message from the following AFOs: *eq_mode_health*, *pilot_not_busy*, and *health*. Accordingly, it can only execute after the latter AFOs have completed. The dependency load balancer uses the AF frame file to determine the AFO priming conditions, and then it establishes an AFO "execution order". An AFO execution order is a chronological/dependency ordering that indicates which AFOs must execute before other AFOs can be scheduled. This ordering is segmented into iterations (or steps), and it allows the load balancer to calculate which AFOs can be executed at a particular time and which AFOs can be scheduled simultaneously.

An outline of aforementioned algorithm is as follows:

1) The frame file is parsed to determine the AFO interconnections.
2) A dependency matrix is constructed to depict the AFO priming conditions.
3) The "external input" AFOs (described in Section 2.2) are located and executed during execution iteration #1.
4) Given that the external input AFOs have completed execution, the dependency matrix is examined to determine the AFOs whose priming conditions have been met. These AFOs are scheduled during execution iteration #2.
5) Given that the external input AFOs and the AFOs of step 4 have completed, the dependency matrix is again parsed to determine the AFOs that can be executed. These AFOs are fired during execution iteration #3.
6) Step 5 is performed until all of the AFOs have completed execution. During each repetition of this step, the AFOs that were executed are recorded. After all AFOs have been fired, the execution order is known. The execution order is designed such that all AFOs in the same execution iteration are capable of being fired *simultaneously*.
7) Each iteration of the execution order is parsed, and all AFOs in that iteration are allocated to different VIDs (if the required number of VIDs is available) to permit their parallel execution.
8) Step 7 is repeated until all AFOs have been assigned of a VID.

To illustrate the algorithm, we examine the Event Diagnosis Expert. Using the dependency algorithm, the following execution order is generated:

Iteration #1 - *eq_mode_health*, *pilot_not_busy*, *health*, *jtids* can be executed, because they are the external input AFOs.

Iteration #2 - *a_previous_waypoint*, *a_unaided_sol*, *a_ecm_env*, *a_pilot_aware*, *a_alpha_check*, *a_lead_map_error*, *a_wm_map_error*, *a_eo_radar_des* can be executed, because their priming conditions have been fulfilled by the execution of the external input AFOs.

Iteration #3 - *system_output* can be executed, because its priming conditions have been fulfilled by the execution of the AFOs in Iteration #2.

Since the AFOs executed during the same iteration can be executed in parallel, they are allocated to different VIDs. For instance, given that four VIDs are available to execute the Event Diagnosis Expert, then the dependency load balancer would generate the following AFO to VID mapping:

VID 1 - *eq_mode_health, a_previous_waypoint, a_alpha_check, system_output.*
VID 2 - *pilot_not_busy, a_unaided_sol, a_lead_map_error.*
VID 3 - *health, a_ecm_env, a_wm_map_error.*
VID 4 - *jtids, a_pilot_aware, a_eo_radar_des.*

As a result of the allocation, each of the four VIDs is completely utilized (if the AFO loads are assumed to be uniform). Specifically,

- *eq_mode_health, pilot_not_busy, health,* and *jtids* will be executed first and simultaneously.
- *a_previous_waypoint, a_unaided_sol, a_ecm_env, a_pilot_aware* will be executed second and simultaneously.
- *a_alpha_check, a_lead_map_error,a_wm_map_error,* and a_eo_radar_des will be executed third and simultaneously.
- *system_output* will be executed last.

The dependency load balancer was employed to generate AFO to VID mappings for the Event Diagnosis Expert for one to thirteen VIDs. Subsequently, the resultant load modules were executed and performance metrics were obtained. These results are presented in Tables 13 and 14. Further, they are illustrated in Figures 37 and 38.

A comparison of the speedup achieved using the connectivity load balancer with that attained utilizing the dependency load balancer is illustrated in Table 15 and Figure 39. As expected the performance of the dependency based algorithm was, in general, better than that of the connectivity method. Accordingly, this analysis indicates that the AFO priming conditions are a more important facto for determining the allocation scheme than the inter-VID connectivity. Nevertheless, both allocations methodologies are sub-optimal. As depicted in Figure 39, neither algorithm was consistently better than other. As a result, it appears that multiple factors should be considered when determining the AFO to VID mapping.

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	775	553	1328
2	763	305	1068
3	763	315	1078
4	748	227	975
5	766	277	1043
6	764	235	999
7	638	246	884
8	766	203	969
9	750	221	971
10	766	219	985
11	638	252	890
12	748	207	955
13	702	230	932

Table 13 - Performance of the Event Diagnosis Expert Utilizing the Dependency Load Balancer

Number of VIDs	Relative Speedup
1	1
2	1.81
3	1.76
4	2.44
5	2.0
6	2.35
7	2.25
8	2.72
9	2.50
10	2.53
11	2.19
12	2.67
13	2.40

Table 14 - Steady-State Speedup for the Event Diagnosis Expert Distribution via Dependency Load Balancer

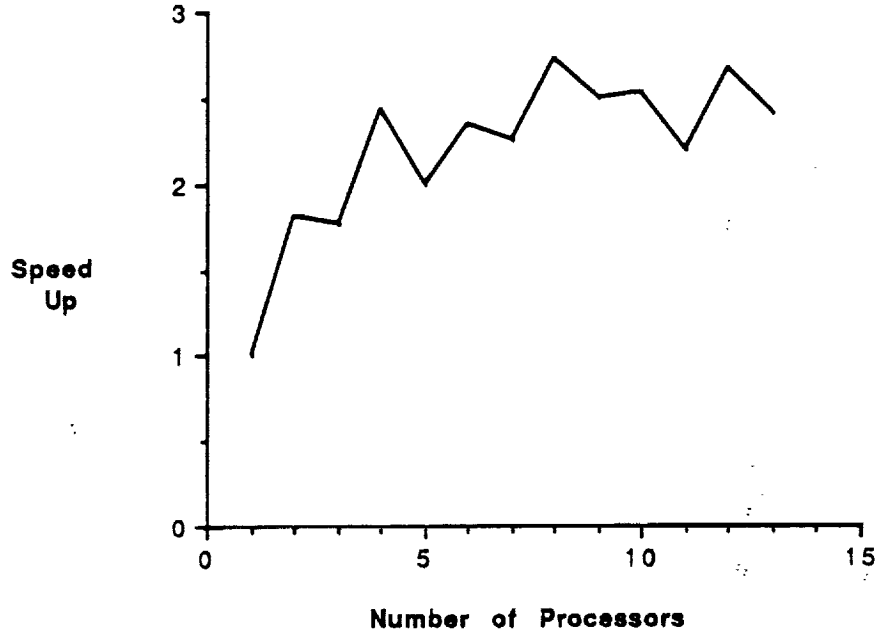

Figure 37 - Steady State Speedup using the Dependency Load Balancer

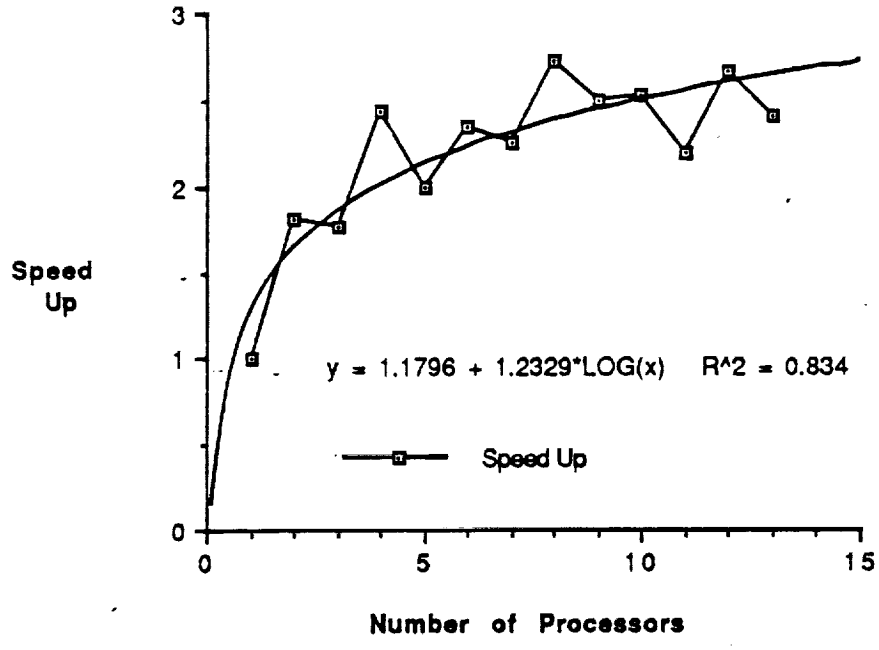

Figure 38 - Steady State Speedup with Logarithmic Approximation

Number of VIDs	Speedup *Connectivity* Load Balancer	Speedup *Dependency* Load Balancer
1	1	1
2	1.48	1.81
3	1.85	1.76
4	2.12	2.44
5	2.03	2.0
6	2.19	2.35
7	2.47	2.25
8	2.60	2.72
9	2.29	2.50
10	2.37	2.53
11	2.29	2.19
12	2.16	2.67
13	2.56	2.40

Table 15 - Connectivity Load Balancer vs. Dependency Load Balancer Speedup Comparison

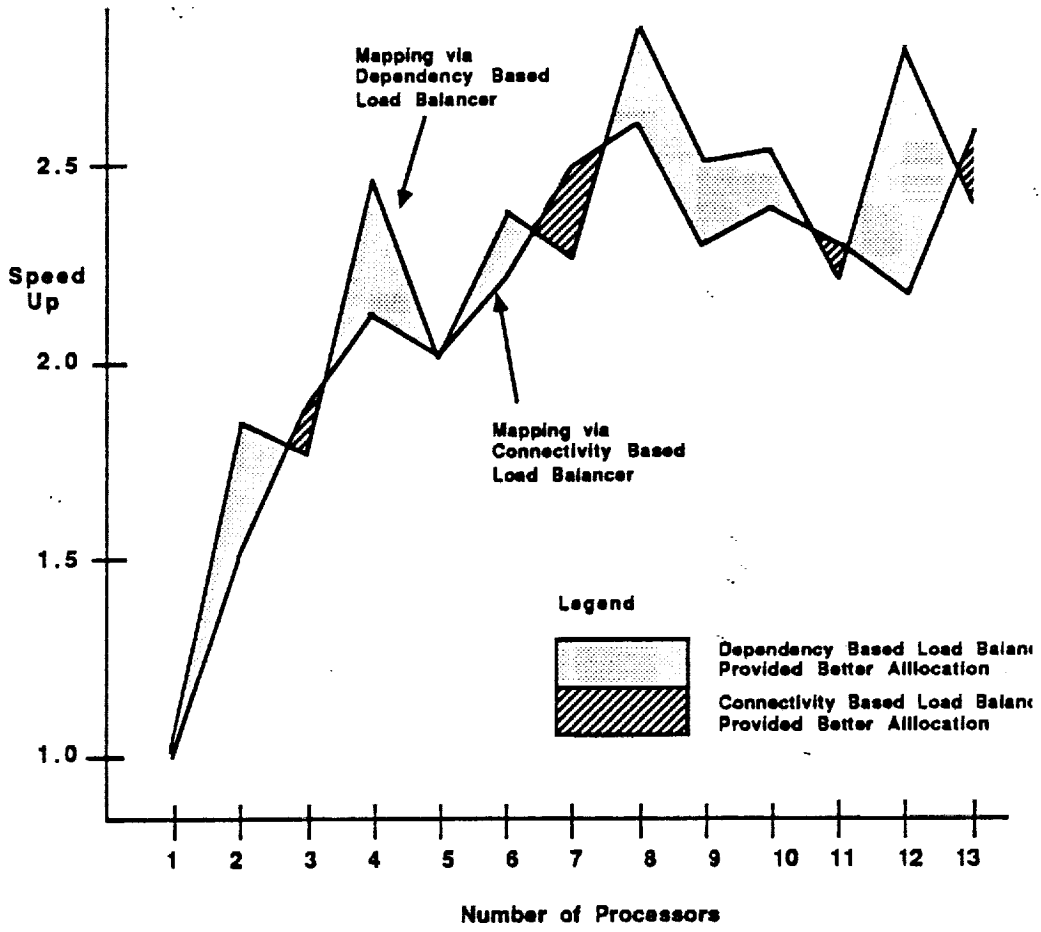

Figure 39 - Connectivity Load Balancer vs. Dependency Load Balancer Speedup Comparison

4.7 Distributing the Work Load of the System Output AFO

The previous analyses have incorporated several allocation schemes: an intuitive hand generated mapping, an automated allocation based on minimizing inter-VID connectivity, and an automated distribution based on maximizing parallelism. However, in each method, the computational load of the System Output AFO was allocated to only one VID (centralized allocation of the Output AFO). The analysis conducted in this Section examines the advantages of distributing the work load of this AFO.

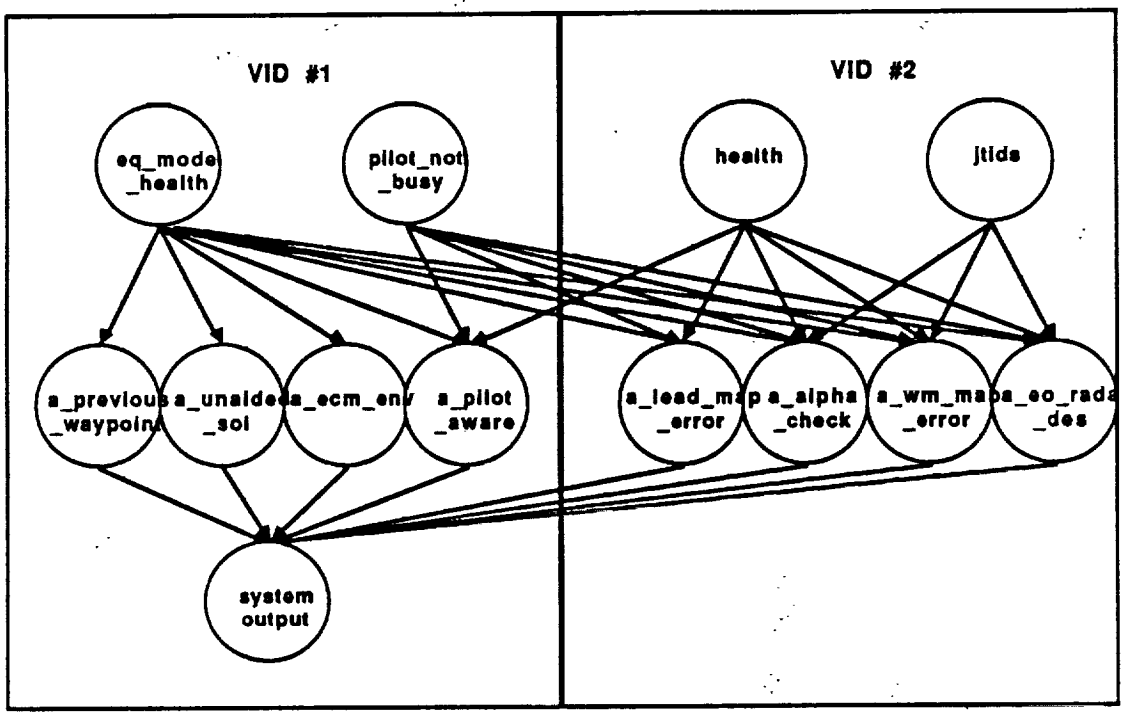

Figure 40 - Centralized Distribution of the System Output AFO

An example of a centralized allocation of the System Output AFO is depicted in Figure 40. The AFOs that comprise the Event Diagnosis Expert are mapped onto two VIDs, and the System Output AFO is assigned to one of the VIDs. The distributed System Output scheme is illustrated in Figures 41 and 42. This method reduces the work load of the System Output AFO by allocating its load over multiple VIDs. For example, when two VIDs are available, two instances of the System Output AFO are employed, one per VID. Each Event Diagnosis Expert AFO sends its output messages to the local instance rather than to a centralized System Output AFO. As a result, the computational load, and accordingly the bottleneck effect, of the System Output AFO is minimized.

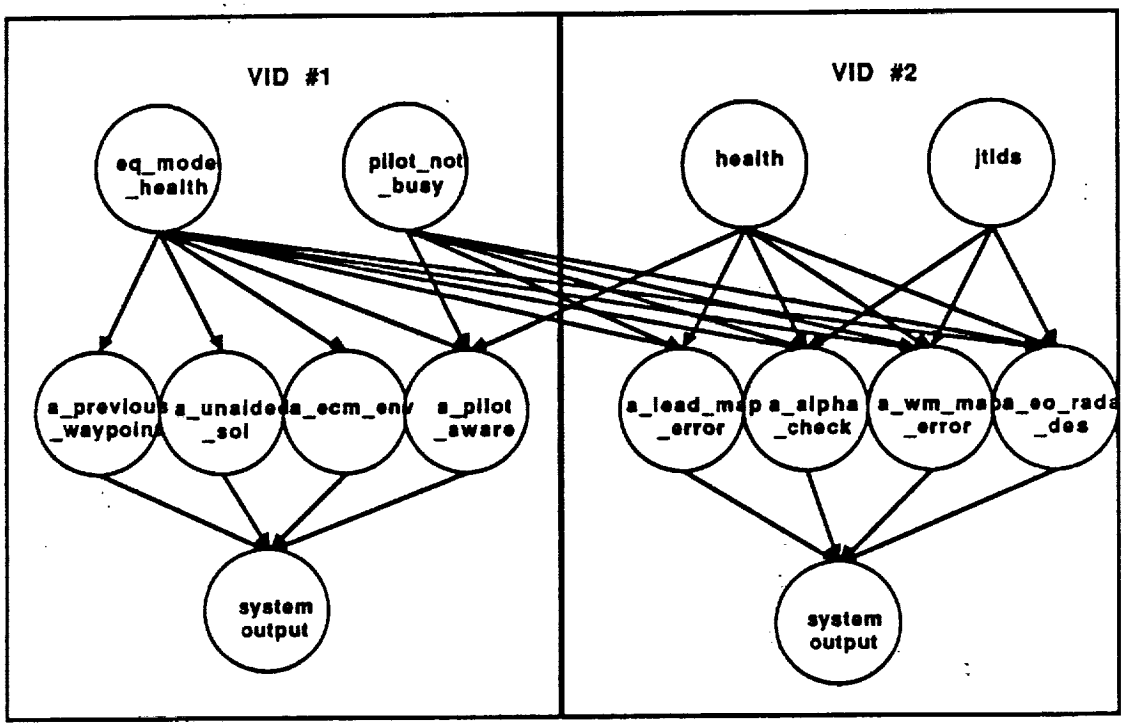

Figure 41 - Distributed Allocation of the System Output AFO - 2 VIDs

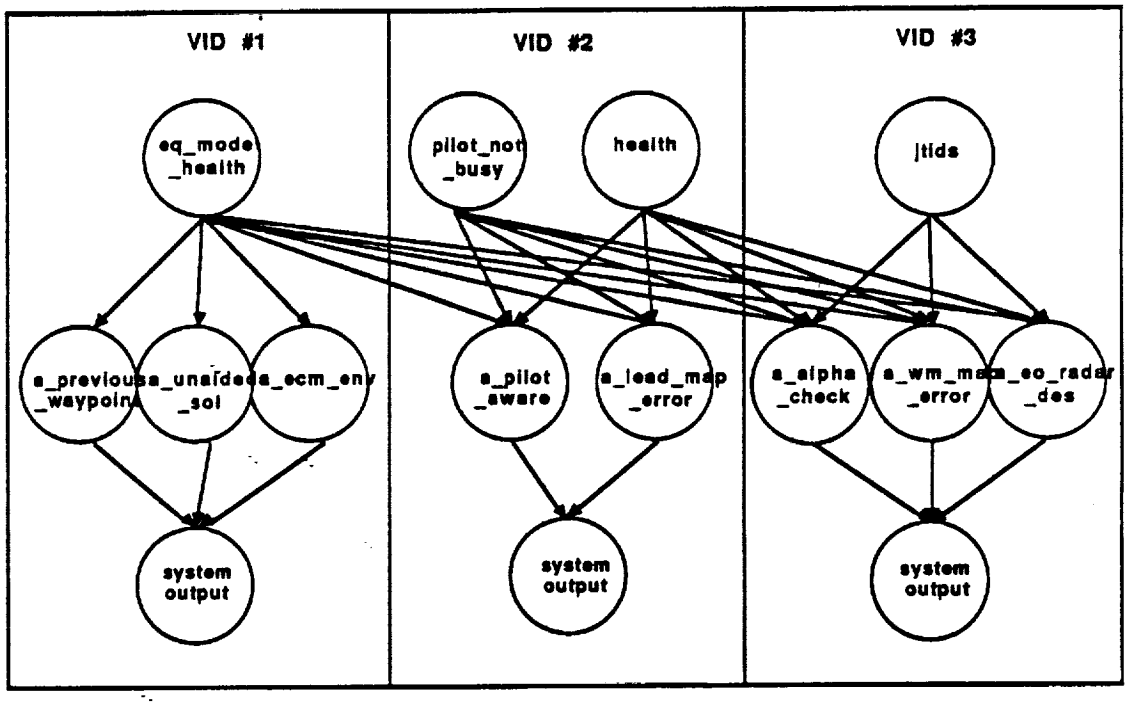

Figure 42 - Distributed Allocation of the System Output AFO - 3 VIDs

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	775	553	1328
2	763	386	1149
3	763	277	1040
4	748	194	942
5	766	218	984
6	764	179	943
7	638	163	801
8	766	163	929
9	750	164	914
10	766	163	929
11	638	153	791
12	748	155	903
13	702	153	855

Table 16 - Performance of the Event Diagnosis Expert Utilizing the Dependency Load Balancer and Distributed System Output

The AF-FTPP Interface and the dependency load balancer were augmented to allocate local instances of the System Output AFO. Further, the dependency load balancer was employed to generate AFO to VID mappings for the Event Diagnosis Expert for one to thirteen VIDs. Subsequently, the resultant load modules were executed and the performance of the application was recorded. These metrics are presented in Tables 16 and 17. Further, the speedup is illustrated in Figure 43.

A comparison of the speedup achieved using the centralized allocation of the System Output AFO with that attained utilizing the distributed method is illustrated in Table 18 and Figure 44. As expected, the performance of the distributed algorithm was significantly better than that of the centralized method. Accordingly, this analysis indicates that the use of a centralized output AFO will constrain the performance of a parallelized application.

Number of VIDs	Relative Speedup
1	1
2	1.43
3	2.0
4	2.85
5	2.54
6	3.09
7	3.39
8	3.39
9	3.37
10	3.39
11	3.61
12	3.57
13	3.61

Table 17 - Steady-State Speedup - Distributed System Output Allocation via Dependency Load Balancer

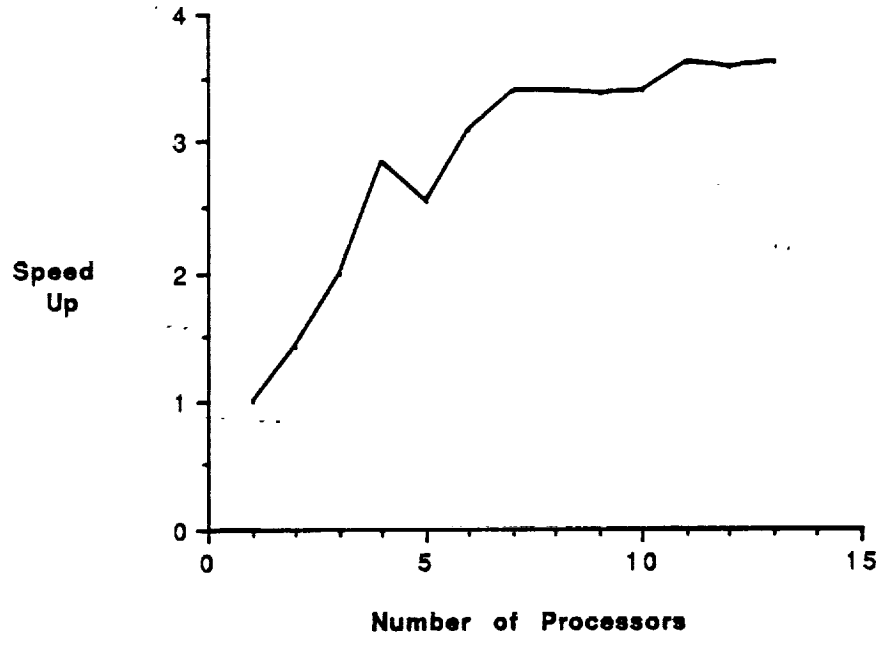

Figure 43 - Steady-State Speedup Using the Dependency Load Balancer and a Distributed Output Process

Number of VIDs	Speedup *Centralized* Output	Speedup *Distributed* Output
1	1	1
2	1.81	1.43
3	1.76	2.0
4	2.44	2.85
5	2.0	2.54
6	2.35	3.09
7	2.25	3.39
8	2.72	3.39
9	2.50	3.37
10	2.53	3.39
11	2.19	3.61
12	2.67	3.57
13	2.40	3.61

Table 18 - Centralized Output vs. Distributed Output Speedup Comparison

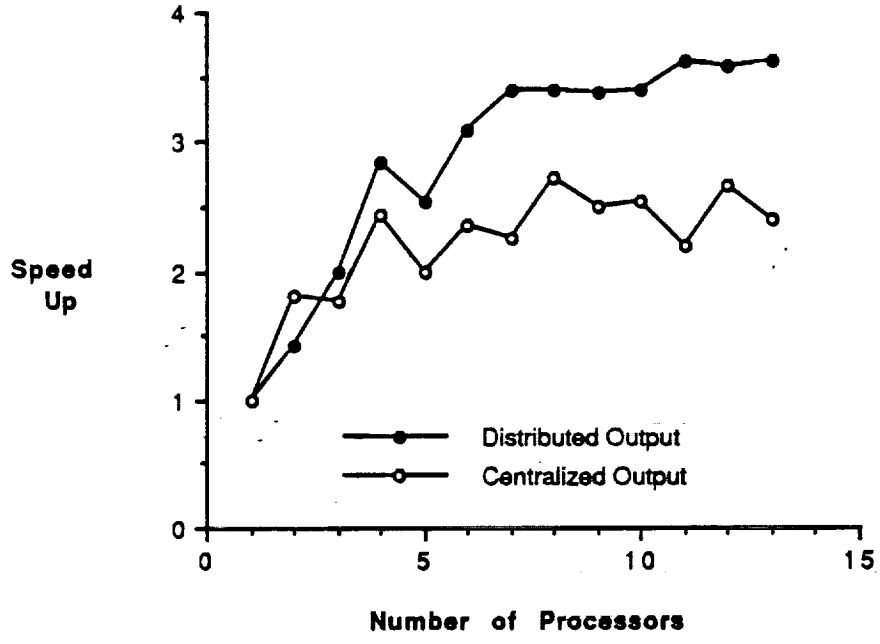

Figure 44 - Centralized Output vs. Distributed Output Speedup

4.8 Performance of the Real-Time Controller

The Real Time Controller task suite was executed on one to 15 simplex VIDs using both the connectivity-based load balancing scheme and the dependency-based load balancing scheme. The seven "external input AFOs" (that is, *mis056*, *node1*, *main78*, *fig058*, *amvrat*, *mail00*, and *main80*) were primed by invoking a procedure which sent messages to each of these AFOs. All terminating nodes transmitted a message to the *system_output* AFO (see Figure 15). Furthermore, to parallel the analysis of the Event Diagnosis Expert the computational load applied to each AFO was varied from a no load state to a uniform 150 ms load. Similarly, three times were recorded: (1) the initialization time, (2) the steady state time, and (3) the total time. Tables 19 through 22 present the results of these measurements.

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	3551	**3916**	**7482**
2	3327	3050	6376
3	3903	2530	6433
4	3919	2365	6283
5	3903	2253	6156
6	3821	1940	5761
7	3903	2038	5940
8	3903	2122	6025
9	3343	2130	5473
10	3327	1917	5244
11	3919	1884	5803
12	3343	1965	5308
13	3327	2015	5341
14	3343	1740	5083
15	3247	**1831**	**5078**

Table 19 - Performance of the Real Time Controller Task Suite Utilizing the Connectivity Load Balancer

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	3551	8524	12074
2	3343	6485	9828
3	3903	3987	7889
4	3919	3520	7422
5	3903	3212	7114
6	3821	2663	6484
7	3903	2681	6582
8	3919	3024	6943
9	3343	2688	6029
10	3327	2461	5786
11	3919	2602	6521
12	3327	2583	5909
13	3343	2718	6059
14	3343	2408	5750
15	3247	2394	5640

Table 20 - Performance of the Real Time Controller Task Suite Utilizing the Connectivity Load Balancer with Computational Load

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	3551	**3916**	**7482**
2	3247	2774	6020
3	3791	2477	6268
4	3727	1665	5391
5	3247	2286	5533
6	3791	2079	5868
7	3727	1558	5284
8	3711	1572	5283
9	3807	1910	5716
10	3791	1849	5638
11	3247	2034	5281
12	3247	2010	5256
13	3807	1822	5628
14	3711	1603	5313
15	3247	**1952**	**5198**

Table 21 - Performance of the Real Time Controller Task Suite Utilizing the Dependency Load Balancer

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	3551	8524	12074
2	3247	4947	8193
3	3791	3951	7741
4	3711	2317	6027
5	3247	3127	6373
6	3791	2801	6591
7	3727	2067	5794
8	3727	1994	5721
9	3807	2380	6187
10	3791	2276	6066
11	3247	2538	5800
12	3247	2456	5701
13	3807	2201	6006
14	3727	2007	5733
15	3247	2349	5595

Table 22 - Performance of the Real Time Controller Task Suite Utilizing the Dependency Load Balancer with Computational Load

The performance analysis for the Real Time Controller closely parallels the evaluation of the Event Diagnosis Expert described in Section 4.3. The initialization times were relatively constant despite the AFO to VID mappings. The computations for speedup were based upon the steady state times and reflect the increase in performance relative to the situation where a single processor executes the entire AFO suite. Maximum speedup values of 2.25 using the connectivity load balancer and of 2.51 using the dependency load balancer were attained when the AFOs (without an additional computational load) were allocated to multiple VIDs. These performance values are rather modest and are comparable to those of the Event Diagnosis Expert. When each AFO was given a constant 150 ms computational load, the speedup characteristics of this task suite improved substantially, reaching maximum speedup values of 3.56 and 4.28 for the connectivity and dependency load balancers, respectively. Tables 23 and 24 represent the speedup for the two load balancing strategies; Figures 45 and 46 graphically depict the speedup. In general, the dependency load balancing strategy provided marginally better AFO to VID allocations than the connectivity load balancing scheme, resulting in greater performance as depicted in Figure 47.

Number of VIDs	Speedup No Extra Load	Speedup Extra 150 ms.
1	1.00	1.00
2	1.28	1.31
3	1.55	2.14
4	1.66	2.42
5	1.74	2.65
6	2.02	3.20
7	1.92	3.18
8	1.85	2.82
9	1.84	3.17
10	2.04	3.46
11	2.08	3.28
12	1.99	3.30
13	1.94	3.14
14	**2.25**	3.54
15	2.13	**3.56**

Table 23 - Steady-State Speedup for the Real Time Controller Task Suite Utilizing the Connectivity Load Balancer

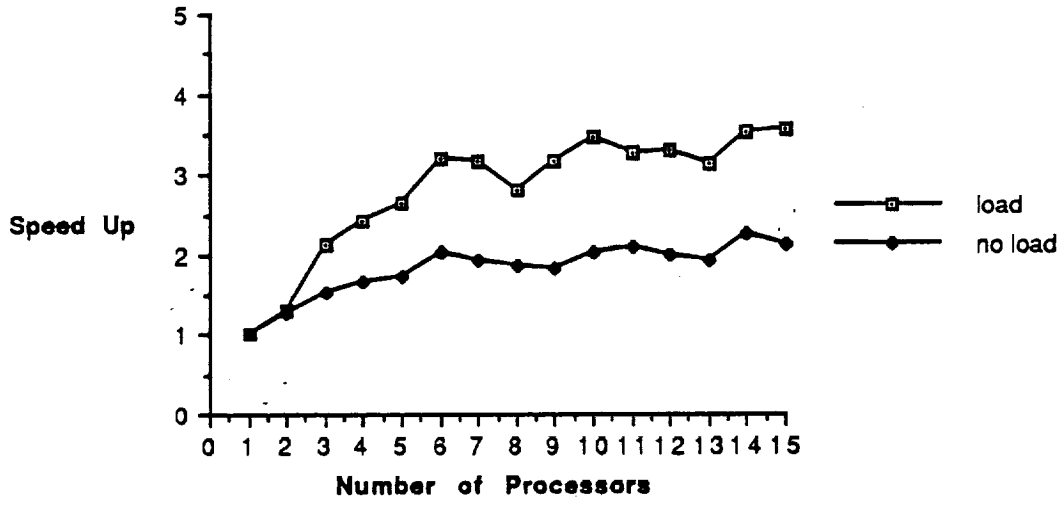

Figure 45. Speedup Comparisons for the Real Time Controller Task Suite Utilizing the Connectivity Load Balancer

Number of VIDs	Speedup No Extra Load	Speedup Extra 150 ms.
1	1.00	1.00
2	1.41	1.72
3	1.58	2.16
4	2.35	3.68
5	1.71	2.73
6	1.88	3.04
7	**2.51**	4.12
8	2.49	**4.28**
9	2.05	3.58
10	2.12	3.74
11	1.92	3.36
12	1.95	3.47
13	2.15	3.87
14	2.44	4.25
15	2.01	3.63

Table 24 - Steady-State Speedup for the Real Time Controller Task Suite Utilizing the Dependency Load Balancer

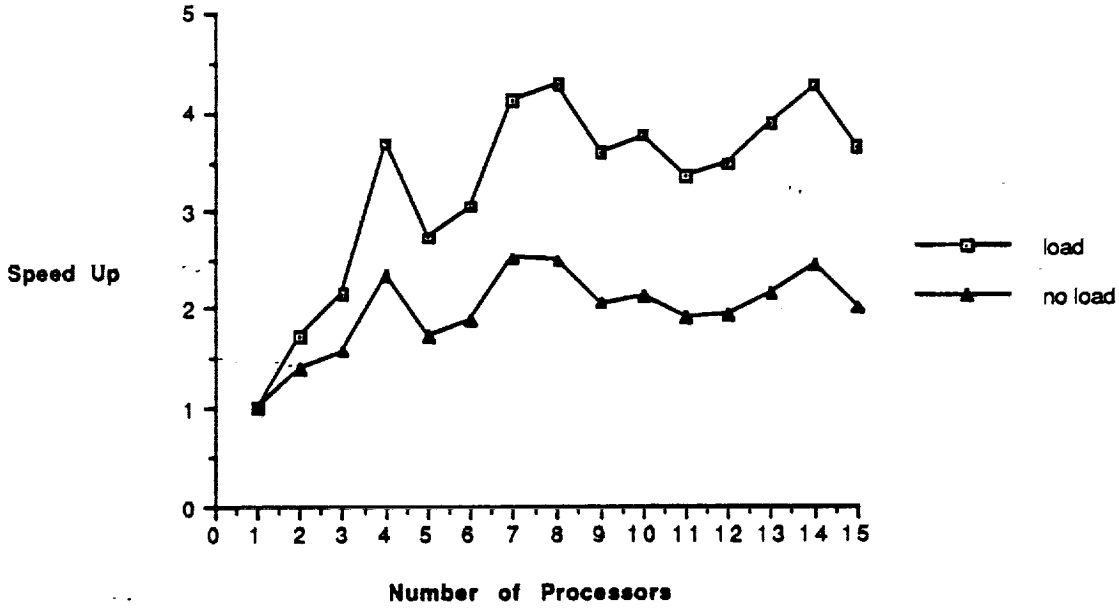

Figure 46. Speedup Comparisons for the Real Time Controller Task Suite Utilizing the Dependency Load Balancer

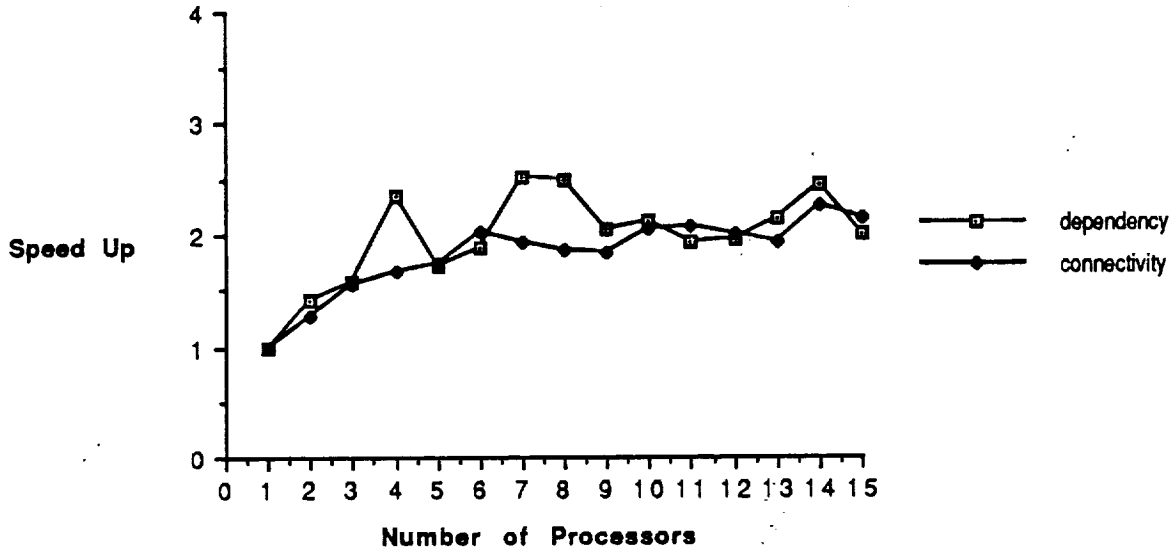

Figure 47 - Connectivity Load Balancer vs. Dependency Load Balancer Speedup Comparison

One of the goals of the application of AF methodology to the Real Time Controller task suite was to stress the computational and memory management resources of the FTPP. In the computations arena, it was desirable to determine how well the FTPP performed when a single VID was assigned the entire computation load as well as when the suite of 15 VIDs shared this load. Not surprisingly, the FTPP survived quite well in both cases. However, it is somewhat astonishing that approximately 7 seconds were required to execute the series of 57 computationally unloaded AFOs when all were allocated to a single VID and that when the AFOs were shared among 15 VIDs the total execution time on a single VID was still a lofty 5 seconds. Of course, in both situations the initialization time was significant. Yet even with initialization time excluded, the steady state times were almost 4 seconds and 2 seconds for the single VID and the 15 VID cases respectively. (Refer to the **bold** values in Tables 19 and 21). In an effort to identify the time sinks, the procedure timings were collected as described previously in Section 4.1. These timings were collected in two cases: (1) all 57 AFOs were assigned to a single VID and (2) the 57 AFOs were distributed among 15 VIDs. Although all procedure timings were measured, some routines were particularly noteworthy because they either required a significant amount of time or were invoked numerous times. Table 25 indicates the procedure times for the single VID case.

Procedure	Num. of Calls	Time/Call (ms)	Total time (ms)
malloc	391	0.53	208
lwrite	261	1.77	464
lread	12375	0.39	4816
lrewrite	718	0.31	224
tsk_loop	75	26.70	2003
port_num	134	16.38	2195
af_go	1	3536	3536
af_exec	75	1.45	109
exec_afo	74	16.74	1239

Table 25 - Procedure timing for the Real Time Controller Suite (1 VID)

When the 57 tasks were allocated among the 15 VIDs, similar data were collected. However, these values were collected on a single VID, specifically the VID that hosted the *system_output* AFO. Table 26 presents these results.

Procedure	Num. of Calls	Time/Call (ms)	Total time (ms)
malloc	345	0.70	240
lwrite	203	2.36	480
lread	8242	0.32	2640
lrewrite	149	0.64	96
tsk_loop	23	22.61	520
port_num	134	15.95	2003
af_go	1	3232	3232
af_exec	236	3.08	728
exec_afo	21	8	168

Table 26 - Procedure timing for the Real Time Controller Suite (15 VIDs)

These results identify areas which are possibly candidates for optimization because of the aggregate amount of time necessary for execution of any particular procedure. A couple of inefficiencies are known:

1. The *tsk_loop* is a procedure which is executed prior to execution of the AFO task and which returns the primed AFO with the largest importance.

However, because the searching mechanism checks the priming conditions of all AFOs even though only a subset of all AFOs is active on any particular VID, *tsk_loop* requires a substantial time expenditure per invocation.

2. Although the execution time for each call to *lread* is not substantial, the number of calls warrants investigations into the efficiency of that routine and into the frequency of its use. It is a List Management System procedure which returns an object associated with a node by scanning a linked list in search of the requested object.
3. The *af_go* procedure initializes the AF, all the AFOs and the port tables. Although it is an initialization function and, consequently, not of primary concern for optimization of steady-state functions, time savings would be reaped in initializing only those AFOs which execute on any particular VID.

The Real Time Controller task suite did exercise the memory management policies in the AF-FTPP system. In fact, the incorporation of this 57 AFO suite abruptly identified the bounds. In the AF-FTPP system there are three memory management strategies corresponding to each memory classification:

1. program code (includes FTPP Operating System, AF, AFOs and global variables)
2. task stack space
3. heap space

The sizes of these memory areas are fixed at link time. The size of the program code is obviously not dynamic, and it resides at an address specified at link time. The task stack area is fixed in size even at the task level; that is, each task's stack is allocated at an address specified at run time. Consequently, the heap size is also fixed but characteristic of heaps, blocks are allocated at run-time upon request. The size of the program code and the task stack space are functions of the number of tasks; the amount of heap space is determined by the residual memory.

The initial attempts to execute the 57 task suite failed because of memory constraints. In one case, the heap space was exhausted even before initialization was complete. This problem was resolved by decreasing the size of each task stack to a mere 2560 bytes. The selection of this value precluded the use of debugging facilities because of the lack of stack space. It is obvious that the task stack space is severely limited in this test case in which

the AFOs perform no real function. In fact, the current system (that is, AF-FTPP Operating System and processor hardware) cannot accept an application consisting of numerous AFOs which require significant allocation of local variables.

4.9 Performance of the Real-Time Controller with Distributed Output

The previous analyses of the real-time controller may be tainted by the fact that a single *system_output* AFO is invoked 18 times on the VID from which the data was collected. This increases the computational burden on this VID in addition to the other AFOs which have been allocated to that particular VID. The timing data includes this additional overhead which certainly is not representative of the majority of VIDs. In order to normalize these factors, the functionality of the *system_output* AFO was distributed among the VIDs such that each VID invokes a local *system_output* rather than sending a message to a remote *system_output* AFO. This concept is described in Section 4.7 in detail.

An experiment using a 15 VID distribution and the dependency load balancer generated a maximum steady-state execution time of 849 ms. This is considerably lower than the 1952 ms (see Table 21) measured when there was a single *system_output* AFO. The computed speedup is 4.61 which is essentially a 130% *increase* in performance. Figure 48 depicts these results. It can be seen that funnelling all output through a single system_output AFO is truly a major bottleneck.

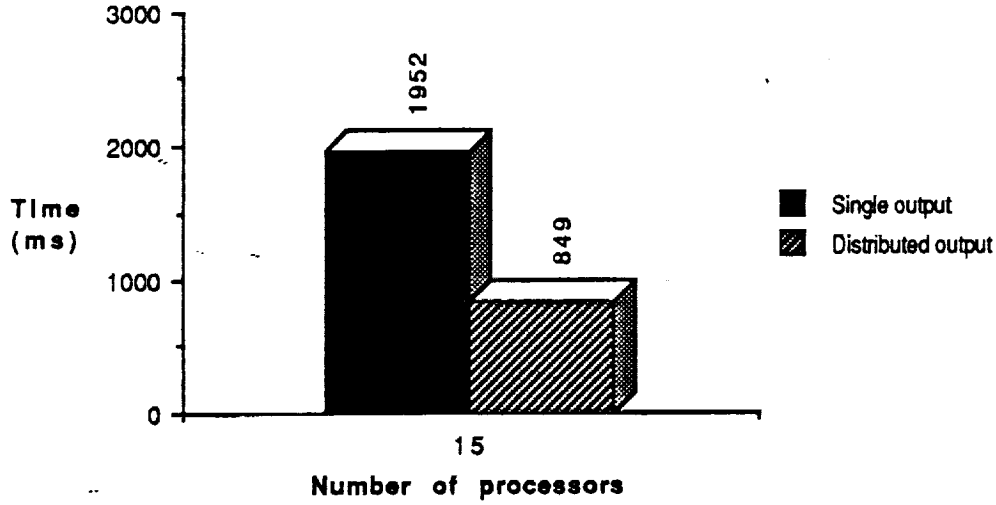

Figure 48 - Execution times of the Real Time Controller Task Suite with Single *system_output* vs. Distributed *system_output*

4.10 Improving the Activation Framework Scheduler

As discussed in Section 2.2, all AFOs are initialized on all VIDs regardless of the AFO to VID distribution. This method was incorporated because at the time that these analyses were performed, only homogeneous load modules could be downloaded to the FTPP. However, this approach is inefficient because the initialization of all AFOs increases the FTPP memory requirements and reduces the available FTPP throughput. The memory requirements are increased, because the initialization of each AFO involves a task creation and the allocation of a corresponding task stack. Since AFOs are initialized on each VID but are invoked on only one, many AFO tasks are needlessly created. Furthermore, the initialization of all AFOs on all VIDs decreases the performance of a parallelized application, because the time required to search the AFO lists increases (since the data structures for both the local and remote AFOs are stored in these lists). For instance, when the AF decides the next AFO to schedule, each AFO is extracted from the AFO list and queried to determine if it is primed. Since this list unnecessarily includes the remote AFOs, the time required to search this list is longer than if the list only contained the local AFOs.

In this analysis, the AF-FTPP scheduler was improved by determining whether or not an AFO resides on the VID before parsing the associated list. It was speculated that this improvement would streamline the scheduling decision and increase the performance of the application.

4.10.1 Impact on the Event Diagnosis Expert

The dependency load balancer was employed to generate AFO to VID mappings for the Event Diagnosis Expert for one to thirteen VIDs. Additionally, the distributed System Output AFO scheme was utilized. Subsequently, the performance metrics were obtained. These measurements are presented in Tables 27 and 28. Further, the speedup is illustrated in Figure 49.

A comparison of the speedup achieved using the unenhanced scheduler with that attained using the improved scheduler is illustrated in Table 29 and Figure 50. As expected, the performance of the Expert improved using the enhanced algorithm.

Number of VIDs	Initialization(ms)	Steady-State(ms)	Total Execution(ms)
1	775	553	1328
2	763	352	1115
3	763	238	1001
4	748	184	932
5	766	194	960
6	764	157	921
7	638	136	774
8	766	136	902
9	750	137	887
10	766	136	902
11	638	146	784
12	748	144	892
13	702	145	847

Table 27 - Performance of the Event Diagnosis Expert Utilizing the Dependency Load Balancer, Distributed System Output, and Improved AF-FTPP Scheduler

Number of VIDs	Relative Speedup
1	1
2	1.57
3	2.32
4	3.01
5	2.85
6	3.52
7	4.07
8	4.07
9	4.04
10	4.07
11	3.79
12	3.84
13	3.81

Table 28 - Steady-State Speedup - Distributed System Output, Dependency Load Balancer, and Improved AF-FTPP Scheduler

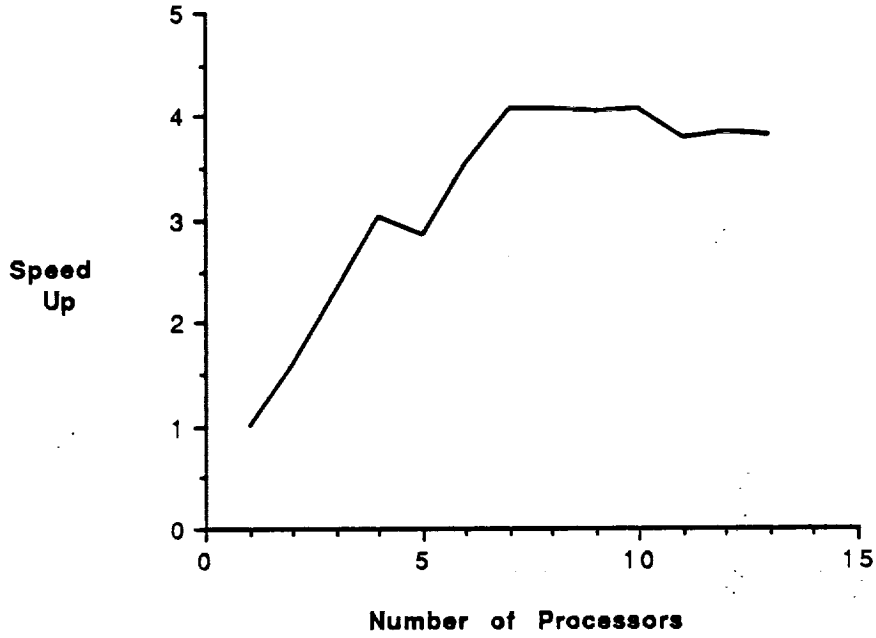

Figure 49 - Steady-State Speedup Using the Dependency Load Balancer, a Distributed Output Process, and an Improved Scheduler

Number of VIDs	Speedup Distributed Output	Speedup Distributed Output & Improved Scheduler
1	1	1
2	1.43	1.57
3	2.0	2.32
4	2.85	3.01
5	2.54	2.85
6	3.09	3.52
7	3.39	4.07
8	3.39	4.07
9	3.37	4.04
10	3.39	4.07
11	3.61	3.79
12	3.57	3.84
13	3.61	3.81

Table 29 - Speedup - Distributed Output vs. Distributed Output with Improved Scheduler

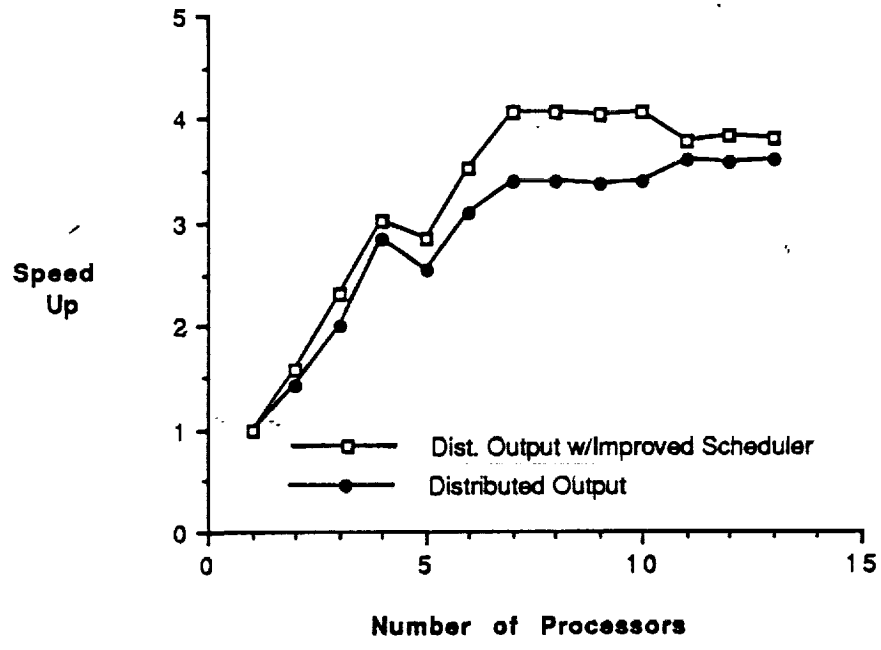

Figure 50 - Distributed Output vs. Distributed Output with Improved Scheduler

4.10.2 Impact on the Real-Time Controller

Since the scheduler improvements were incorporated to eliminate the unnecessary checking the priming conditions of AFOs which are not resident on a particular VID, the Real Time controller application was executed with this optimization. The procedure timings similar to those of Table 26 were collected to identify the performance improvements for the case when the 57 AFOs are distributed among 15 VIDs. These results are presented in Table 30. A comparison of these results with those of Table 26 indicates a significant decrease in the number of invocations of *malloc*, *lwrite*, *lrewrite*, and *tsk_loop*. Conversely, there was a noticeable increase in the number of calls to *af_exec* presumably resulting from the diminished execution time of the scheduling loop.

Procedure	Num. of Calls	Time/Call (ms)	Total time (ms)
malloc	316	0.61	192
lwrite	188	1.87	352
lread	6671	0.58	2512
lrewrite	86	0.56	48
tsk_loop	11	4.36	48
port_num	134	14.28	1987
af_go	1	3232	3232
af_exec	492	2.29	1121
exec_afo	5	12.60	63

Table 30 - Procedure timing for Real-Time Controller Suite with Improved Scheduler (15 VIDs)

In addition, the total steady state execution time was 620 ms (versus 1952 ms in the original case). It should be noted that these timing values reflect not only the improved scheduler but also the effects of the distributed *system_output*. (Refer to Figure 51).

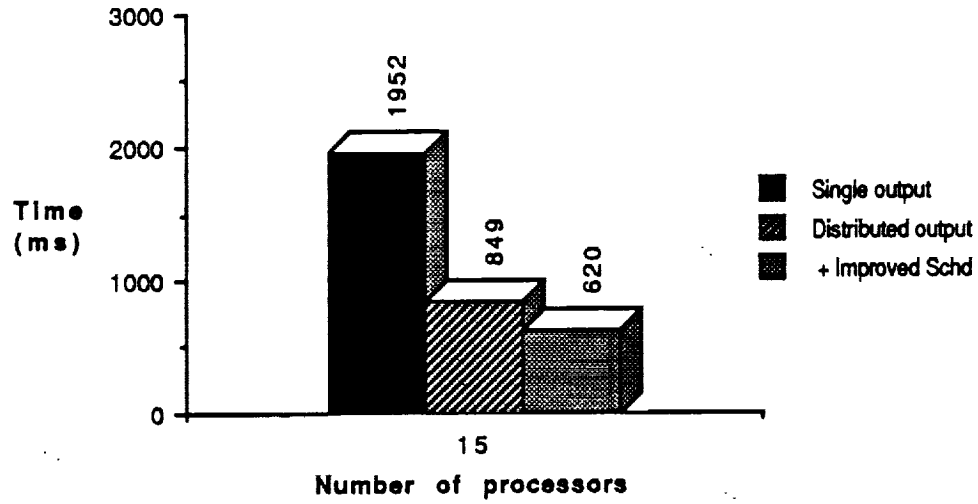

Figure 51 - Execution times of the Real Time Controller Task Suite with Single *system_output* vs. Distributed *system_output* vs. Distributed *system_output* and Improved Scheduler

4.11 Performance Evaluation with Redundancy Management

In the analyses discussed previously the redundancy management capabilities of the FTPP were ignored. Not only were all AFOs allocated to simplex virtual groups but the FTPP's intrinsic redundancy management functions were inactive. Since one of the benefits of use of this computer system is its fault tolerant aspects, it is highly desirable to evaluate the FTPP-AF system with the redundancy managements functions.

Redundancy management requires the invocation of two additional functions. The Reconfiguration function is required to initially configure the simplex processors as fault masking groups which are capable of detecting faults. In addition, the Fault Detection and Isolation function (FDI) must actively monitor the fault detection mechanisms in order to protect the system from failure.

Because of the memory constraints encountered with the Real-Time Controller application and because both of the additional functions have rather ambitious stack requirements, the Event Diagnosis Expert application was chosen to host this series of tests. This selection permits the use of the larger task stack spaces required for the incorporation of these two redundancy management functions.

Since the goal of the test was to examine the impact of the redundancy management upon the steady-state execution of the AFOs and since the reconfiguration is executed only upon request, the reconfiguration of the system into four triplexes was completed prior to the test initiation. On the other hand, FDI invocation is a periodic task which is invoked each scheduling cycle concordant with the AFO scheduling mechanism. Consequently, the timing data reflects only the invocation of the FDI redundancy management function. Moreover, since no fault is present during the execution, the timing data reflects only the minimal overhead for FDI. More aggressive tests were not conducted because the current implementation is structured to execute the AFO suite only once. The AFOs were allocated to these four triplexes by the dependency load balancer. In addition, the *system_output* was distributed and the scheduler improvements discussed in Section 4.10 were in effect.

At the completion of the test each processor computed the steady-state execution time based upon its local clock. Since these clocks are not synchronized, the clock values reported even by members of a VID are not exact. In fact, they frequently differed by 1 or 2 clock ticks (that is, 16 ms or 32 ms). The execution times stated are the averages of the times of the members of the VID with the highest values for execution time.

Since this series of tests introduces the concept of redundant virtual groups, an initial test was conducted to create a baseline for comparison purposes. This initial test reflects the steady-state time for execution of the 13 AFOs on four triplexes without invoking FDI. This time was 207 ms. With execution of FDI periodically, the execution time became 254 ms. The overhead of FDI in this system of 13 computationally void AFOs is 18%.

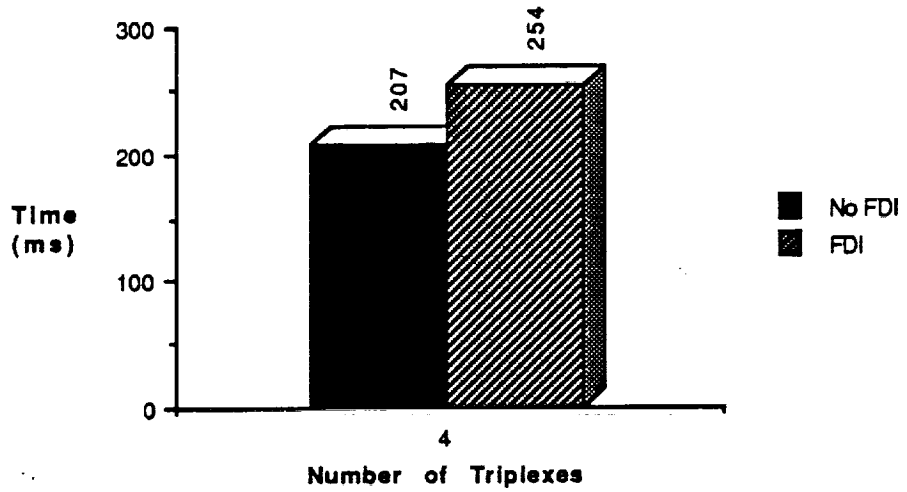

Figure 52 - Execution times of the Event Diagnosis Expert Task Suite with and without FDI (no additional computational load per AFO)

Since the AFO suite is devoid of computational activity, a disproportionate component of the execution time represents the operating system overhead. In an attempt to normalize this factor, a 150 ms computational load was applied to each AFO (excluding the *system_output* AFO). The steady-state execution time of 13 AFOs without FDI was 593 ms; the same test with FDI required 641 ms. FDI now accounts for 7% of the overhead. Furthermore, in both sets of tests FDI required a fixed amount of time (that is, 47 ms).

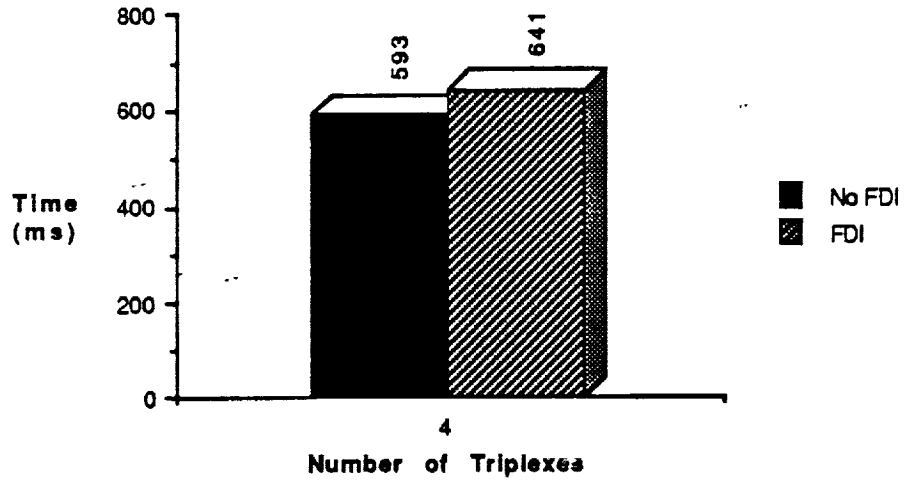

Figure 53 - Execution times of the Event Diagnosis Expert Task Suite with and without FDI (150 ms additional computational load per AFO)

C-2

5.0 Performance Improvements

The performance evaluation of the AF-FTPP system noted some glaring inefficiencies in the AF-FTPP design particularly when the AFOs were distributed among multiple VIDs. Furthermore, other areas which were not obviously inefficient could be optimized to improve performance. The following recommendations would clearly achieve this goal:

1. The method currently employed to parallelize the AFOs across some number of VIDs entails creating identical load modules for each VID and dynamically invoking only those AFOs which have been assigned to any particular VID. However, at initialization in order to create global information concerning the existence and location of each AFO, all AFOs are created as active tasks and initialized on each VID. This parallelization procedure is the source of gross inefficiencies in terms of both computational utilization and memory allocation. A far better approach would entail the activation of only those AFO tasks which actually reside on a VID. This method would reap great savings in memory allocation. The creation of the global AFO information could be achieved in one of two ways: (1) the global information could be amassed statically during the load balancing stage and furnished to the AF during initialization without actual initialization of the non-resident AFOs or (2) a temporary task could be created strictly for the initialization of non-resident AFOs.

2. The Activation Framework is structured upon the List Management System which performs list manipulation functions. Although these mechanisms are extremely versatile and provide dynamicism in terms of creation and deletion of data structures they tend to be very costly in processing time because the searching mechanisms are essentially sequential. Since the AF-FTPP invokes LMS procedures (particularly *lread*) frequently to retrieve global AFO status information, significant benefit would be derived by incorporating a fast searching mechanism such as a hashing function into at least the *lread* procedure. Alternatively, if the characteristics of the desired application can be specified compromises may be agreed upon which maintain some of the important features of the AF while dispensing with some of the more costly features. For example, if the application did not require dynamic instantiation of AFOs many benefits could be evoked. The allocation of AFOs to VIDs

would be established prior to link time with the static global knowledge of AFOs such as port numbers, VID numbers, and service IDs. This improvement decreases initialization time in instantiating AFOs which will not be executed on the particular VID, decreases search times of lists because lists are shorter, and relieves the memory management crunch. On the other hand, implementation of this type of scenario complicates the migration of AFOs to other VIDs. (The entire topic of function migration has yet to be addressed.)

3. It is obvious from the analysis of the Event Diagnosis Expert that both the connectivity based load balancer and the dependency based load balancers are sub-optimal. The connectivity load balancer has been compared directly with a hand optimized distribution (Section 4.5) with results favoring the hand calculated version. Thus far, two metrics have affected the load balancing strategy: (1) the connectivity load balancer attempted to automatically distribute the work load by *minimizing the inter-VID connectivity* and (2) the dependency load balancer aimed at *maximizing parallelism*. A more sophisticated load balancing scheme using both of these parameters would hopefully create a workload distribution which is "more optimal" than either of the strategies developed to date. This certainly would increase the overall performance of the application.

6.0 Conclusion

In brief, the following work was completed during Knowledge Representation into Ada Parallel Processing contract:

1. The Activation Framework and Fault Tolerant Parallel Processor Operating System were adapted to permit the execution of the Event Diagnosis Expert of the Adaptive Tactical Navigator on the FTPP (Refer to Sections 2.2 - 2.4).
2. The AF and Event Diagnosis Expert were analyzed, a number of inefficiencies were identified, and a corresponding set of improvements were incorporated (Sections 4.1 and 4.2).
3. The Event Diagnosis Expert was executed on multiple processors of the FTPP, and the advantages and disadvantages of its parallel execution were examined (Section 4.3).
4. An AFO to VID mapping analysis was completed. In this examination, CSDL compared several mapping schemes. The results of this test were utilized to upgrade the automatic distribution algorithm (Sections 4.5 and 4.6).
5. A computation loading analysis was performed. In this investigation, we used artificial loads to examine the effect that more computationally intensive AFOs would have on the parallel execution of the AF on the FTPP (Sections 4.4 and 4.7).
6. The AF methodology was employed to execute a computationally stressful and memory-intensive AFO suite (that is, the real time controller) on the multiple processors of the FTPP (Sections 4.8 - 4.10).
7. The Redundancy Management capabilities of the FTPP were incorporated into the AF-FTPP Methodology and their overhead was measured (Section 4.11).

With respect to the aforementioned work, the following items are a summary of the results that were attained and conclusions derived during the KRAPP project:

1. The Event Diagnosis Expert can be executed on the FTPP using the AF Methodology.
2. The AF Methodology and AF-FTPP Interface can be enhanced to be more efficient. As detailed in Section 4.2, we improved the performance of the Event Diagnosis Expert by 258, 293, and 209 percent for one, two, and three

processors respectively. Further improvements can be incorporated into the AF Methodology to optimize its sequential and parallel performance.

3. When utilizing the multiple processors of the FTPP and an CSDL developed automatic load balancer, the speed of the Event Diagnosis was increased by a factor of 4.07 (in this case, 7 processors were employed).
4. The AFO to VID mapping investigation indicated that multiple factors should be considered when determining a distribution. Nonetheless, the incorporation of several dissimilar factors, which is common and desirable, makes the automatic load balancing algorithm considerably more complex.
5. The AFO computational loading analysis supported our speculation that the performance of an application using the AF Methodology would improve if the AFOs are more computationally intensive. This conclusion indicates that the parallel execution of the AF Methodology will be more efficient if each AFO represents multiple Horn Clauses rather than merely a single Clause.
6. The impetus to use the AF Methodology to execute a real time controller was to demonstrate that the concept is scaleable and to better stress the capabilities of the FTPP. This test showed that this Methodology can be used to execute a more difficult application on the FTPP (57 AFOs rather than only 13) while still attaining considerable performance speedup (6.32 when using an automatic load balancer and 15 processors).
7. The FTPP's Fault Detection and Isolation process was incorporated in the Event Diagnosis Expert (augmented with a 150 ms. computational load to normalize the test) and decreased the performance of the application by only seven percent.

The aforementioned work, results, and conclusions were discussed in detail in Section 4. Refer to that section if more information is desired.

In addition, Section 5.0 detailed a number of known inefficiencies which should be rectified to increase the performance of the AF-FTPP system. Briefly, these recommendations include:

1. Instantiated AFOs should be initialized only on those VIDs to which they have been allocated.
2. The Activation Framework List Management System procedures should be optimized wherever possible, particularly the *lread* function.

3. A sophisticated load balancing scheme should be developed which maximizes parallel execution of AFOs while minimizing inter-VID connectivity.

Based on the work completed and experience thus attained during the KRAPP project, we believe that completion of the following work will facilitate the integration of future AF based intelligent navigation systems on the FTPP and make the execution of such systems more efficient:

1. As outlined in Section 3.0, the integration of any new application (e.g. Real Time Controller) into the AF-FTPP system involves multiple development systems. This has been a deficiency particularly in terms of designer productivity because of the differences in development system operating systems commands and because of the additional steps required to transfer files and to execute command files. It is *extremely* desirable to develop a single workstation environment for generating AF-FTPP based load modules.

2. The limitations of the FTPP memory management software caused several AF-FTPP implementation and integration problems. In addition to the memory management problems detailed in Section 4.7 incurred while stressing the AF-FTPP system with a 57 task suite, stack space allocations by the Activation Framework during initial integration phase generated task stack overflows. This was remedied by allocating the variables in question in a scratch global memory area. This solution should only be a temporary means to deal with this problem since subsequent versions of the AF would cause significant AF-FTPP integration issues. The development of a more efficient memory management system is essential if more complex intelligent navigation systems are to be executed on the FTPP in real-time.

3. The implementation of the remote data insertion and capture system needs to be completed. In addition, the current design must be enhanced to permit simultaneous insertion and capture of multiple messages.

4. An investigation of methods by which general events can drive the AF-FTPP scheduler should be performed. Such event driven capabilities are necessary if all AF concepts are to be supported.

5. The AF-FTPP Interface needs to be enhanced to enable the use of Generalized Objects. Since the preliminary version of the AF only utilizes boolean and integer data, this Interface currently only supports the transmission and reception of a 16-bit data word.

6. The AF-FTPP Interface must be modified to incorporate a time facility. If such a facility were developed, the AF deadline and time tagging capabilities could be used.

7. The applications addressed until now have been devoid of real computations. Real applications, of course, will require AFOs to perform computations of real variables which will be communicated to other AFOs. Incorporation of procedures into the AFOs has yet to be addressed.

7.0 References

[Ach87] Acharya, N. A., Dowding, J. P., Glasson, D. P., Matchett, G. A., and Pomarede, J. L., "ATN Software Part 1 Specification - Technical Appendices," The Analytical Sciences Corporation, Report No. TR-5344-2, October 1987.

[Bab90] Babikyan, C. A., "The Fault Tolerant Parallel Processor Operating System Concepts and Performance Measurement", CSDL-R-2219, Charles Stark Draper Laboratory, Cambridge, MA 02139, February 1990.

[Ber1] Berning, S., Glasson, D. P., Matchett, G. A., "Functionality and Architectures for an Adaptive Tactical Navigation System," NAECON Proceedings, May 1987.

[Ber2] Berning, S., Glasson, D. P., Pomarede, J. L., "Knowledge Engineering for the Adaptive Tactical Navigator," NAECON Proceedings, May 1988.

[Gre87] Green, P. E., Glasson, D. P., Pomarede, J. L., Acharya, N. A., "Real-Time Artificial Intelligence Issues in the Development of the Adaptive Tactical Navigator," *Proc. Space Operations-Automation and Robotics Workshop*, NASA Johnson Space Center, Houston, Texas, August 1987.

[Gre89] Green, P. and Nainani, K., "Specifications for Activation Framework Procedure Calls and Data Structures", Real-Time Intelligent Systems Corporation, Worcester, MA 01606, October 1989.

[Har87] Harper, R. E., "Critical Issues in Ultra-Reliable Parallel Processing", PhD Thesis, Massachusetts Institute of Technology, June 1987.

A. Appendix A - Software Specifications

Appendix A presents the Software Specifications for the AF-FTPP Interface.

A.1 AF-FTPP Interface File: AFO_to_VID.c

A.1.1

Process Name: Init_AFO_to_VID

Inputs: None

Outputs: AFO_to_VID Table

Reference: KRAPP Final Report, Section 2.7

Notes: None

Description:

This process initializes the AFO_to_VID table. The AFO_to_VID table is used by the AF-FTPP Interface to determine which AFOs are local to the VID. This table is structured as an array. The index into the the array is the AFO Identifier, and the value of the table entry is the corresponding AFO's location (VID number). If an AFO does not exist, the associated table entry is assigned to (-1).

The Init_AFO_to_VID process is created by the Automatic Load Module Generator.

A.1.2

Process Name: Determine_SID_from_AFO

Inputs: AFO Identifier

Outputs: Service Identifier

Reference: KRAPP Final Report, Section 2.2.2

Notes: None

Description:

This function returns the AFO Service ID (SID) to the calling function if the AFO is local to the VID. If the AFO is remote, then a value of (-1) is returned. This process is invoked when creating an AFO structure (to initialize the one of the SID fields of the Frame_Node database) and when sending a message (to determine whether the destination AFO is local or remote).

A.1.3

Process Name: Determine_VID_from_AFO

Inputs: AFO Identifier

Outputs: Virtual Group Identifier

Reference: KRAPP Final Report, Section 2.2.2

Notes: None

Description:

This function returns the VID on which the specified AFO resides. It is invoked when initializing the one of the VID fields of the Frame_Node database. Additionally, it is called by the Return_Port_to_VID procedure to determine the location of a remote AFO port.

A.2 AF-FTPP Interface File: App_Sex.h

A.2.1

Process Name: Application Services that Exist

Inputs: None

Outputs: None

Reference: KRAPP Final Report, Section 2.7

Notes: None

Description:

This process identifies the application services (AFOs) that exist. It is created by the Automatic Load Module Generator and is used to allocate and schedule the AFOs.

A.3 AF-FTPP Interface File: Conv_Msg.c

A.3.1

Process Name: Serialize_Message

Inputs: Activation Framework Message
Destination VID

Outputs: FTPP Message

Reference: KRAPP Final Report, Section 2.2.2

Notes: None

Description:

This process converts an Activation Framework message into an FTPP message. This translation is required to permit inter-VID message communication. The Serialize_Message procedure accepts an AF message as an input parameter, parses it, converts it to the FTPP format, and sends the resultant FTPP message to the destination VID.

A.3.2

Process Name: Deserialize_Message

Inputs: FTPP Message

Outputs: Activation Framework Message

Reference: KRAPP Final Report, Section 2.2.3

Notes: None

Description:

This process converts an FTPP message into an Activation Framework message. The Deserialize_Message procedure accepts a message in the FTPP format, allocates memory for the new AF message, and reformats the FTPP data to obtain the appropriate AF structure. Subsequently, the resultant AF message is delivered to its destination port.

A.3.3

Process Name: External_Input

Inputs: FTPP Message

Outputs: FTPP Message

Reference: KRAPP Final Report, Sections 2.2.1 and 2.2.3

Notes: None

Description:

This process is responsible for retrieving a message received from a remote VID and delivering it to the appropriate destination AFO port. Initially, the External_Input procedure queries the FTPP input queue to determine whether or not a pending message exists. If one or more messages have been received by the VID, then the first one is removed and an AF message is created using the Deserialize_Message process (Process Description A.3.2). This AF message is then sent to its destination port using the Activation Framework procedure AF_Deliver. After the message has been stored in the appropriate AFO port or if no messages are pending in the queue, the External_Input process returns to the calling procedure.

A.3.4

Process Name: Send_Remote

Inputs: Activation Framework Message
Destination VID

Outputs: FTPP Message

Reference: KRAPP Final Report, Section 2.2.2

Notes: None

Description:

This process is responsible for sending an AF message to the appropriate remote VID. The Send_Remote procedure accepts an AF message as an input parameter, converts it to an FTPP message by invoking the Serialize_Message procedure (Process Description A.3.1), and sends the FTPP message to the specified destination VID using the FTPP primitive snd_msg.

A.3.5

Process Name: AF_Exec

Inputs: FTPP Message

Outputs: None

Reference: KRAPP Final Report, Sections 2.2.1 and 2.2.3

Notes: None

Description:

This process controls the retrieval of all messages received from remote VIDs. Each iteration of the AF-FTPP scheduling loop, all pending messages are removed from the FTPP input queue and delivered to their destination AFO ports. This removal and delivery procedure is performed by repeatedly invoking the External_Input process (Process Description A.3.3).

A.4 AF-FTPP Interface File: Lv2_Init.c

A.4.1

Process Name: Lv2s_Init

Inputs: Activation Framework Object Transfer Function Names

Outputs: None

Reference: KRAPP Final Report, Section 2.7

Notes: None

Description:

This process initializes the Versatile Real-Time Executive (VRTX) tasks that are employed by the AF-FTPP methodology to schedule and execute the AFO transfer functions. Similar to the application services file (App_Sex.h - Process Description A.2.1), it is created by the Automatic Load Module Generator.

A.5 AF-FTPP Interface File: Malloc.c

A.5.1

Process Name: Heapinit

Inputs: End_of_Program_Address
Beginning_of_Task_Space_Address

Outputs: None

Reference: KRAPP Final Report, Section 2.3

Notes: None

Description:

The Heapinit process initializes the bounds of the heap. The lower bound of the heap is the end of the memory used by the AF-FTPP load modules. The upper limit is the beginning of the memory utilized for the VRTX task stacks and control blocks.

This process also initializes the "taken" and "free" lists that are employed for dynamic memory management (see Process Descriptions A.5.3 and A.5.4).

A.5.2

Process Name: SBRK

Inputs: Desired_Number_of_Bytes

Outputs: Address_of_Allocated_Memory

Reference: KRAPP Final Report, Section 2.3

Notes: None

Description:

The SBRK process returns a pointer to an unreserved section of the heap that is equal to the desired number of bytes.

A.5.3

Process Name: Malloc

Inputs: Desired_Number_of_Bytes

Outputs: Address_of_Allocated_Memory

Reference: KRAPP Final Report, Section 2.3

Notes: None

Description:

The Malloc procedure returns a pointer to an unused section of memory that is greater than or equal to the desired number of bytes. This process initially scans the "free list" (memory that has been previously deallocated) for a contiguous block of memory capable of storing the required number of bytes. If none of the deallocated blocks is of sufficient size, then the SBRK procedure is invoked to allocate the necessary memory from the heap. After a section of memory is allocated (from either the heap or the "free" list), the size and address of this space is added to "taken" list to permit its subsequent deallocation. Additionally, a pointer to this section of memory is returned to the calling process.

A.5.4

Process Name: Free

Inputs: Pointer_to_Memory_to_be_Deallocated

Outputs: None

Reference: KRAPP Final Report, Section 2.3

Notes: None

Description:

This process deallocates a section of contiguous memory. Specifically, the block of memory identified by the specified pointer (input parameter) is removed from the "taken" list and inserted into the "free" list to allow its subsequent reallocation.

A.6 AF-FTPP Interface File: Schd.c

A.6.1

Process Name: AF_Swap

Inputs: None

Outputs: None

Reference: KRAPP Final Report, Section 2.2.1

Notes: None

Description:

This process suspends the calling AFO. It is used to return control to the AF-FTPP scheduler after an AFO transfer function has completed its execution.

A.6.2

Process Name: AFO_to_Exec

Inputs: AFO Service Identifier

Outputs: None

Reference: KRAPP Final Report, Section 2.2.1

Notes: None

Description:

The AFO_to_Exec procedure is used to update the Frame_Node database to identify the next AFO that should be executed.

A.6.3

Process Name: Sched

Inputs: Exec_AFO_SID

Outputs: None

Reference: KRAPP Final Report, Section 2.2.1

Notes: None

Description:

The Sched process performs the initialization of the Activation Framework and executes the AF-FTPP scheduling loop. The primary functions of this process are outlined below:

1. Invokes the Init_AFO_to_VID process to initialize the AFO_to_VID table (Process Description A.1.1).
2. Calls the AF_Go procedure to create the Activation Framework Frame_Node database and AFOs.
3. Executes the Lv2_Init process to initialize the VRTX tasks (Process Description A.4.1).
4. Creates a VRTX task for each existing system service and AFO.
5. Invokes the Send_Initial_Message procedure to prime the external AFOs.
6. Calls the AF process tsk_loop to determine the first AFO to execute.
7. Begins the Main Scheduling loop
   a. Executes the AF_Exec task to determine if any input messages have been received from remote VIDs.
   b. Schedules the next AFO to be executed.
   c. Calls the tsk_loop procedure to locate the primed AFO with the highest importance.
   d. If the application has completed, the loop is exited. Otherwise, it repeats (a) through (d).

A detailed discussion of the Sched process is presented in Section 2.2.1.

A.7 AF-FTPP Interface File: Sex.c

A.7.1

Process Name: Init_Sex_Table

Inputs: Service Exists Flags

Outputs: Service Exists Table

Reference: KRAPP Final Report, Section 2.2.1

Notes: None

Description:

This process initializes the Service Exists Table. This table identifies the application and operating system services that will be executing in the load module. This table is used to determine the number of Versatile Real-Time Executive (VRTX) tasks necessary to execute the application AFOs and System tasks. Further, this table is used to associate the VRTX tasks with their corresponding function (for example, an AFO transfer function).

A.7.2

Process Name: Init_Schd_Table

Inputs: Service Exists Flags

Outputs: Scheduling Table

Reference: KRAPP Final Report, Section 2.2.1

Notes: None

Description:

This process initializes the scheduling class field of each existing system service and AFO. The two types of scheduling classes are periodic and on_message_reception.

A.8 AF-FTPP Interface File: Utls.c

A.8.1

Process Name: Test_Done?

Inputs: AFO Completion Count
Maximum Number of Iterations Count

Outputs: Test Done Flag

Reference: KRAPP Final Report, Section 2.2.1

Notes: None

Description:

The Test_Done? process returns a TRUE or FALSE value to the calling process indicating whether or not the application has completed its execution.

A.8.2

Process Name: Print_System_Output

Inputs: AFO Identifier

Outputs: AFOs_Received Table

Reference: KRAPP Final Report, Section 2.4

Notes: None

Description:

The Print_System_Output procedure is called by the System_Output AFO, and it notes that a message has been received by the specified AFO. The process records the AFO's identity in the AFOs_Received table.

A.8.3

Process Name: Record_AFO_Exec

Inputs: AFO Identifier

Outputs: AFO_Has_Executed Table

Reference: KRAPP Final Report, Sections 2.2.4 and 4.7

Notes: None

Description:

The Record_AFO_Exec process is invoked by the AF-FTPP scheduler to indicate that an AFO has been executed. The identity of this AFO is recorded in the AFO_Has_Executed table.

A.8.4

Process Name: Init_SID

Inputs: AFO Identifier

Outputs: AFO Service Identifier

Reference: KRAPP Final Report, Section 2.2.1

Notes: None

Description:

The Init_SID process provides the Service ID of the specified AFO. If the AFO does not reside on the VID, this process returns a (-1) to the calling procedure.

A.8.5

Process Name: Init_VID

Inputs: AFO Identifier

Outputs: Virtual Group Identifier

Reference: KRAPP Final Report, Section 2.2.1

Notes: None

Description:

The Init_VID process returns the VID on which the specified AFO resides.

A.8.7

Process Name: Assign_Port_to_AFO

Inputs: AFO Identifier
Port Identifier

Outputs: Port_to_AFO Table

Reference: KRAPP Final Report, Section 2.2.1

Notes: None

Description:

The Assign_Port_to_AFO updates the Port_to_AFO table to inform the AF-FTPP Interface of the AFO to Port mappings. The Port_to_AFO table is queried by the AF_Deliver and Send_Remote procedures when determining the location (VID) of a port.

A.8.8

Process Name: Return_Port_to_AFO

Inputs: Port Identifier

Outputs: AFO Identifier

Reference: KRAPP Final Report, Section 2.2.2

Notes: None

Description:

If the specified AFO port is local, this process returns the AFO associated with the port. Conversely, if the AFO is remote, then a value of (-1) is returned to the calling function.

A.8.9

Process Name: Return_Port_to_VID

Inputs: Port Identifier

Outputs: Virtual Group Identifier

Reference: KRAPP Final Report, Section 2.2.2

Notes: None

Description:

The Return_Port_to_VID process returns the VID on which the specified port resides.

A.8.10

Process Name: Init_Time_Counters

Inputs: None

Outputs: Data Structures Used for the Interval Timing

Reference: KRAPP Final Report, Section 2.2.4

Notes: None

Description:

This process initializes the data structures that are used by the Start_Timing and Stop_Timing procedures.

A.8.11

Process Name: Start_Timing

Inputs: Interval Identifier

Outputs: Timing Array Entry

Reference: KRAPP Final Report, Section 2.2.4

Notes: None

Description:

The Start_Timing procedure records value of the local processor clock. This process is invoked to mark the beginning of an interval that is being timed.

A.8.12

Process Name: Stop_Timing

Inputs: Interval Identifier

Outputs: Timing Array Entry

Reference: KRAPP Final Report, Section 2.2.4

Notes: None

Description:

The Stop_Timing process is used to mark the end of an interval that is being measured. This procedure reads the current time, determines the elapsed time by subtracting the corresponding start time, adjusts this difference to make the timing facility non-intrusive, and records this "adjusted" interval time for subsequent retrieval.

B. Appendix B - Modification of the Activation Framework

Appendix B discusses the modifications that were made to the Activation Framework to allow its execution on the FTPP.

1. An Interface package written in Ada was incorporated into the AF source code. It was required to permit the invocation of the FTPP C code by the Ada AF procedures.

2. The List Management System was augmented:
   a. The exceptions were removed because the AF-FTPP system does not support them. They were replaced with a corresponding set of error definitions.
   b. The Ada dynamic memory allocation scheme was replaced with the C malloc and free procedures. This modification was performed, because the Ada allocation and deallocation process is not supported by the AF-FTPP system.
   c. Certain data structures were redefined.
      i. The constant *max_size* was decreased from 1024 to 400 to reduce the memory required by the *node* structure.
      ii. Subtypes were removed because the Ada compiler used by CSDL has problems with this structure.
      iii. The *Free_Node* and *Free_Head* procedures were created to allow dynamic deallocation of list elements.

3. The AF package was altered.
   a. The *want_except* constant was initialized to false.
   b. The *pt_len* and *ptb_len* variables were defined as constants. These changes were performed to permit (d).
   c. Subtypes were removed because the Ada compiler used by CSDL has problems with this structure.
   d. The size attributes for the *Frame_Node* record structures were explicitly initialized to ease debugging.
   e. The structure *go_struct* was changed from a variant record structure to a basic record structure. This was performed to facilitate the serialization and deserialization of the AF messages.

f. The type of the *msg_dead* and *msg_sent* fields of the *msg_struct* record were changed from *time* to *integer*. They were altered to simplify the AF-FTPP Interface (currently these fields are not being used by the AF).

g. The order of the *msg_struct* record was changed to make the *go_struct* the last field in the record rather than an intermediate field. This was performed to simplify the Interface by eliminating problems associated with the byte padding of records (a concern when considering the serialization and deserialization of messages).

h. The *af_struct* record was modified to include two additional integer fields, *remote_VID* and *local_SID* which are required for scheduling the AFOs and transmitting messages.

i. The WPI *af_swinit* and *af_swap* procedures were removed because they were no longer necessary.

4. The AF_GLOB package was created and incorporated into the AF source code. It is a temporary scratch pad area that is required because of FTPP memory limitations.

5. The AF_FRAME package was removed for it was not required.

6. The AF_ERR package was modified to remove the raise exception statements and to add code that will alternatively print a set of the corresponding integer identifiers.

7. The PORT_NUM package was augmented.
   a. The local stack variables that were moved to the scratch pad area were removed.
   b. The source code that addressed the aforementioned local stack variables was changed to reference the scratch pad structures.
   c. The call to the *pad* procedure was extraneous and it was removed.
   d. A C *string_compare* procedure was employed rather than the Ada "=" function. This was done because some difficulty with Ada call was encountered during integration.

8. PORT_CR package
    a. The local stack variables that were moved to the scratch pad area were removed.
    b. The source code that addressed the aforementioned local stack variables was changed to reference the scratch pad structures.
    c. The call to the *pad* procedure was removed, because it was extraneous.

9. The ADD2ADDR package was removed, because it was not necessary.

10. The RET_AFO package
    a. The local stack variables that were moved to the scratch pad area were removed.
    b. The source code that addressed the aforementioned local stack variables was changed to reference the scratch pad structures.
    c. The call to the WPI *af_swap* procedure was replaced with an invocation of AF-FTPP Interface *af_swap* procedure.
    d. The logic involving the assignment of the AFO primed field was changed. Rather than always setting the field to false, the AFO priming function was invoked and this field was assigned to the value of the boolean that was returned.

11. The AFO_INIT package
    a. The local stack variables that were moved to the scratch pad area were removed.
    b. The source code that addressed the aforementioned local stack variables was changed to reference the scratch pad structures.
    c. The creation of the standard I/O port was not required and consequently, it was removed.
    d. The call to the *tsk_init* procedure was removed, because it was not necessary.
    e. Calls to the AF-FTPP functions *Init_VID* and *Init_SID* were added to initialize the associated fields of the AF *Frame_Node* database.

12. The AF_DELIVER package was altered.
    a. The local stack variables that were moved to the scratch pad area were removed.
    b. The source code that addressed the aforementioned local stack variables was changed to reference the scratch pad structures.
    c. A call to the AFO priming function was added to support a message driven scheduler.

13. The SND_OBJ package
    a. The local stack variables that were moved to the scratch pad area were removed.
    b. The source code that addressed the aforementioned local stack variables was changed to reference the scratch pad structures.
    c. A call to the *Return_Port_to_AFO* function was added to determine if the destination AFO was local or remote.
    d. If remote, the *Return_Port_to_VID* function was invoked to locate the destination VID and subsequently the *Send_Remote_VID* procedure was called.

14. The GET_OBJ package was augmented.
    a. The local stack variables that were moved to the scratch pad area were removed.
    b. The source code that addressed the aforementioned local stack variables was changed to reference the scratch pad structures.

15. The FR_INIT package was modified to dynamically allocate a frame pointer.

16. The MSG_CHK package
    a. The local stack variables that were moved to the scratch pad area were removed.
    b. The source code that addressed the aforementioned local stack variables was changed to reference the scratch pad structures.

17. The TSK_LOOP package
    a. The local stack variables that were moved to the scratch pad area were removed.
    b. The source code that addressed the aforementioned local stack variables was changed to reference the scratch pad structures.
    c. The main loop was removed. It was no longer necessary, because the corresponding loop control flow was moved into AF-FTPP scheduler.
    d. The references to the *frm_ptr.frm_cafo* field were removed, because they were not required.
    e. The invocation of the WPI *af_swap* procedure was replaced with a call to FTPP interface *afo_to_execute* procedure. This change was required to inform the AF-FTPP interface of the next AFO to execute.
    f. The call to the *tsk_init* procedure was extraneous and therefore removed.
    g. The invocation of the AFO priming function was removed in order to change the polling scheduler to a message driven scheduler.
    h. Invocations to the *Init_SID* procedure were included in the scheduling process to determine if the AFOs were local before searching LMS (performance improvement).

C. Appendix C - Performance Metrics

C.1 Description of the Intervals Measured

Appendix C - Section 1 describes the procedures whose execution times were measured.

1	free	a procedure that releases a specified number of bytes and places this unreserved memory in a "free" list. This unreserved memory can be given to another process that requests the same number of bytes or less.
2,3	malloc	a procedure that requests the allocation of a specified number of bytes of unreserved memory. This memory is allocated from either the heap or the "free" list.
4	lcreat	an LMS procedure that allocates a "header" node for a linked list and returns a pointer to the node.
5	ldel	an LMS procedure that deletes a specified node from a list.
6	lwrite	an LMS procedure that allocates a node element, initializes its data object, and adds it to a specified list.
7	lread	an LMS procedure that returns the data object of a specified node.
8	lrewrite	an LMS procedure that replaces the data object of a specified node with a specified data object.
9	execute	a procedure that invokes the priming function corresponding to a specified AFO.
10	pad	a procedure that pads a specified character string with a specified number of spaces.

11	filter	a procedure that returns the AFO and port names from a character string of the form "AFO/port".
12	tsk_loop	a procedure that returns the primed AFO with the highest importance.
13	ret_afo	a procedure that is invoked when an AFO completes its execution. This procedure determines if the AFO is still primed and suspends to the FTPP scheduler.
14	afo_init	a procedure that initializes an AFO structure.
15	port_num	a procedure that searches the port table for a specified character string and returns its integer port identifier.
16	port_cr	a procedure that initializes a port structure and inserts a corresponding entry into the port table.
17, 35	af_deliver	a procedure that delivers a specified message to the correct port. This procedure also invokes the appropriate priming function to support a message driven scheduler.
18	snd_obj	a procedure that sends a specified message to a specified port. If the destination AFO is local, the AF_Deliver procedure is invoked. If the destination AFO is remote, the message is serialized and sent using the FTPP snd_msg primitive.
19	get_obj	a procedure that removes a message from a specified port.
20	fr_init	a procedure that initializes a frame.
21	msg_chk	a procedure that checks a specified port for the presence of a message.

22	init_afo_to_vid	a procedure that initializes the AFO_to_VID table which depicts the AFO to VID mapping.
23	af_go	a procedure that controls the initialization of the AF and AFOs.
24	init_test_done_flags	a procedure that initializes a set of flags that indicate when the execution of the AFOs is complete.
25	lv2s_init	a procedure that associates each VRTX task with the procedure that should be executed when the task is resumed.
26	send_initial_msg	a procedure that primes the "External Input" Event Diagnosis AFOs by sending a set of messages.
27	af_exec	a task that queries the FTPP input queue to determine if a message has been received from a remote AFO. If one or more messages are present, they are deserialized into AF messages and sent to the appropriate AFO ports using the AF_Deliver procedure.
28	time	a task that requests the time from the NE (removed from the AF-FTPP system).
29	exec_afo	the execution of the primed AFO with the highest importance.
30	send_remote	a procedure that invokes the message serialization procedure and sends the specified message to a remote VID using the FTPP snd_msg primitive.
31	serialize_msg	a procedure that serializes a message in preparation for its transmission to a remote VID.
32	snd_msg	the FTPP primitive for sending a message.

33	sync_self	a procedure that sends a message to itself. It is used to "scoop" all messages to this VID from the Network Element.
34	external_input	a procedure that removes messages from the FTPP input queue using the FTPP get_msg primitive.
36	deserialize_msg	a procedure that converts FTPP messages to AF messages.
37	service creation	a procedure that initializes the VRTX services which exist on the VID.
38	Total time	the length of time required to initialize the AF and AFOs and to execute the application.
39	AF initialization	the length of time required to initialize the AF and AFOs. This time will, in general, remain relatively constant for all distribution strategies, because the process is performed on each VID.
40	AF execution	the length of time from the completion of the send_initial_msg procedure (the end of initialization) to the completion of the application. This time varies based on the number of VIDs involved and AFO to VID mapping.

C.2 Performance Measurements Using the Network Element Simulator

Appendix C - Section 2 presents the preliminary performance measurements. These execution times were recorded when the Event Diagnosis Expert was executed on one, two, and three VIDs in conjunction with the Network Element (NE) Simulator. These times were used, and should *only* be used, to determine where the bulk of the execution time resides and to evaluate the performance gains attained by incorporating enhancements (illustrated in Figure C.1). The performance measurements of the "baseline" AF are presented in Section C.2.1. The times for the "enhanced" AF are outlined in Section C.2.2. The primary points of interest, with regard to a comparison of the performance of the versions of the AF, are printed in **bold** type.

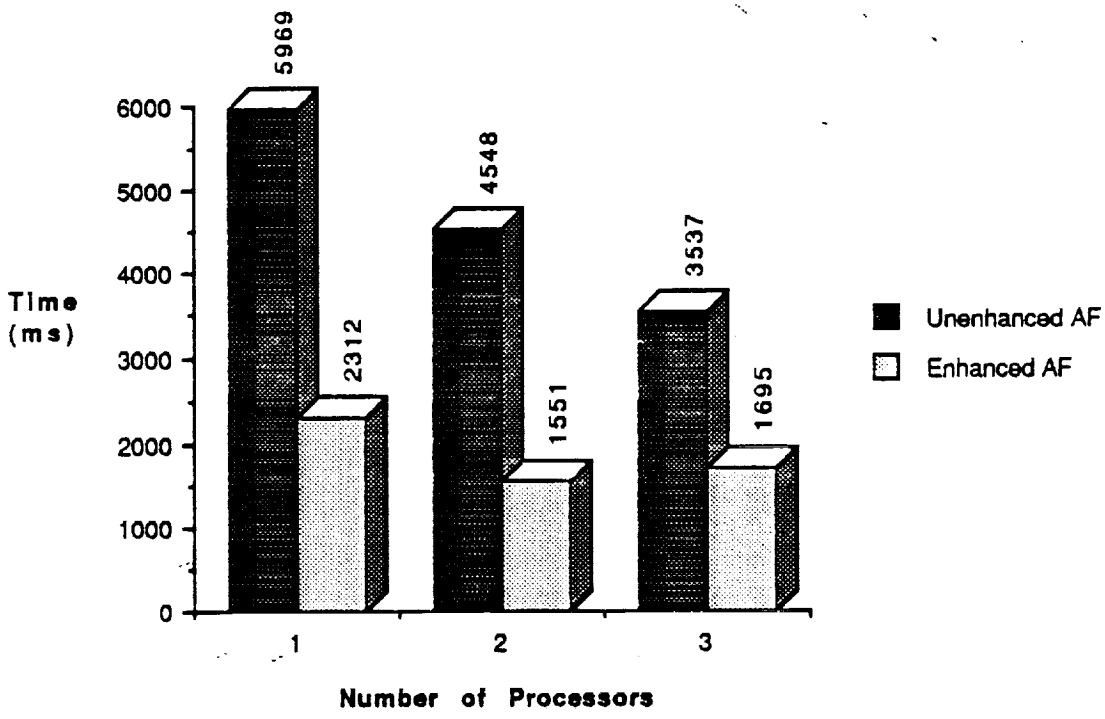

Figure C.1 - Performance of the AF Versions

C.2.1 Performance Measurements Before Enhancements

NUMBER OF VIDS: 1

Position	Procedure	Num. of Calls	Time/Call (ms.)	Total Time (ms.)
1	free	33	0	0
2	malloc - total time	142	0.7	96
3	malloc - free list search	142	0.2	32
4	lcreat	42	0.8	32
5	ldel	33	0	0
6	lwrite	99	1.5	144
7	lread	9605	0.5	4464
8	lrewrite	400	0.3	112
9	execute	246	17.7	4365
10	pad	15	1.1	16
11	filter (not called)	-	-	-
12	tsk_loop	20	222	4437
13	ret_afo (incorrect timing)	-	-	-
14	afo_init	14	5.7	80
15	port_num	541	8	4307
16	port_cr	26	12	311
17	af_deliver	33	5.8	192
18	snd_obj	33	6.3	208
19	get_obj	33	3.3	108
20	fr_init	1	14	14
21	msg_chk (incorrect timing)	-	-	-
22	init_afo_to_vid	1	0	0
23	af_go	1	359	359
24	init_test_done_flags	1	0	0
25	lv2s_init	1	0	0
26	send_initial_msg	1	93	93
27	af_exec	20	24	480
28	time	20	54.4	1088
29	exec_afo	20	46.4	928
30	send_remote	-	-	-
31	serialize_msg	-	-	-
32	snd_msg	-	-	-
33	sync_self (with SIM)	20	16.8	336
34	external_input	-	-	-
35	af_deliver (from ext_input)	-	-	-
36	deserialize_msg	-	-	-
37	service creation	1	48	48
38	time - initial to completion	1	7383	7383

Total time required for execution (without SIM or Time task) = 5969 ms.

NUMBER OF VIDS: 2

Position	Procedure	Num. of Calls	Time/Call (ms.)	Total Time (ms.)
1	free	17	0	0
2	malloc - total time	138	0.2	32
3	malloc - free list search	138	0	0
4	lcreat	42	0	0
5	ldel	17	1.9	32
6	lwrite	83	1.2	96
7	lread	7461	0.4	3216
8	lrewrite	284	0.5	128
9	execute	200	17.4	3471
10	pad	15	1.1	16
11	filter (not called)	-	-	-
12	tsk_loop	15	237	3554
13	ret_afo (incorrect timing)	-	-	-
14	afo_init	14	3.4	48
15	port_num	400	7.8	3126
16	port_cr	26	13.2	343
17	af_deliver	17	3.8	64
18	snd_obj	9	7.1	64
19	get_obj	17	6.4	108
20	fr_init	1	14	14
21	msg_chk (incorrect timing)	-	-	-
22	init_afo_to_vid	1	0	0
23	af_go	1	359	359
24	init_test_done_flags	1	0	0
25	lv2s_init	1	0	0
26	send_initial_msg	1	80	80
27	af_exec	15	38	575
28	time	15	53	800
29	exec_afo	13	29	381
30	send_remote	4	4	16
31	serialize_msg	4	0	0
32	snd_msg	4	4	16
33	sync_self	15	26.7	400
34	external_input (incorrect timing)	-	-	-
35	af_deliver (from ext_input)	12	4	48
36	deserialize_msg	12	1.3	16
37	service creation	1	32	32
38	time - initial to completion	1	5740	5740

Total time required for execution (without SIM or Time task) = 4548 ms.

NUMBER OF VIDS: 3

Position	Procedure	Num. of Calls	Time/Call (ms.)	Total Time (ms.)
1	free	11	0	0
2	malloc - total time	128	0.4	48
3	malloc - free list search	128	0	0
4	lcreat	42	0.4	16
5	ldel	11	0	0
6	lwrite	77	1.2	96
7	lread	5767	0.5	2592
8	lrewrite	212	0.2	48
9	execute	152	17.7	2689
10	pad	15	2.1	32
11	filter (not called)	-	-	-
12	tsk_loop	11	244	2687
13	ret_afo (incorrect timing)	-	-	-
14	afo_init	14	3.4	48
15	port_num	304	8.5	2575
16	port_cr	26	12.6	327
17	af_deliver	11	1.5	16
18	snd_obj	9	3.6	32
19	get_obj	11	1.5	16
20	fr_init	1	14	14
21	msg_chk (incorrect timing)	-	-	-
22	init_afo_to_vid	1	0	0
23	af_go	1	359	359
24	init_test_done_flags	1	0	0
25	lv2s_init	1	0	0
26	send_initial_msg	1	80	80
27	af_exec	11	55.7	613
28	time	11	61.1	672
29	exec_afo	11	28.9	318
30	send_remote	6	0	0
31	serialize_msg	6	0	0
32	snd_msg	6	0	0
33	sync_self	11	46.5	512
34	external_input (incorrect timing)	-	-	-
35	af_deliver (from ext_input)	8	2	16
36	deserialize_msg	8	2	16
37	service creation	1	16	16
38	time - initial to completion	1	4715	4715

Total time required for execution (without SIM or Time task) = 3537 ms.

C.2.2 Performance Measurements After Enhancements

NUMBER OF VIDS: 1

Position	Procedure	Num. of Calls	Time/Call (ms.)	Total Time (ms.)
1	free	33	0	0
2	malloc - total time	142	0.6	80
3	malloc - free list search	142	0	0
4	lcreat	42	0	0
5	ldel	33	0.5	16
6	lwrite	99	1.5	144
7	**lread**	**2162**	**0.5**	**1040**
8	**lrewrite**	**180**	**0.5**	**96**
9	**execute**	**46**	**6**	**275**
10	pad	15	1.1	16
11	filter (not called)	-	-	-
12	**tsk_loop**	**20**	**11.4**	**229**
13	ret_afo	20	5.6	112
14	afo_init	14	2.3	32
15	**port_num**	**59**	**8.5**	**499**
16	port_cr	26	13.2	343
17	af_deliver	33	12.5	414
18	snd_obj	33	15.7	519
19	get_obj	33	5.2	172
20	fr_init	1	14	14
21	msg_chk	83	2.5	208
22	init_afo_to_vid	1	0	0
23	**af_go**	**1**	**751**	**751**
24	init_test_done_flags	1	0	0
25	lv2s_init	1	0	0
26	send_initial_msg	1	122	122
27	af_exec	20	30.4	608
28	time (removed)	-	-	-
29	exec_afo	20	49	980
30	send_remote	-	-	-
31	serialize_msg	-	-	-
32	snd_msg	-	-	-
33	sync_self (with SIM)	20	18.4	368
34	external_input	20	0	0
35	af_deliver (from ext_input)	-	-	-
36	deserialize_msg	-	-	-
37	service creation	1	32	32
38	**time - initial to completion**	**1**	**2670**	**2670**
39	AF initialization (static)	1	903	903
40	AF execution (dynamic)	1	1768	1768

Total time required for execution (without SIM or Time task) = 2312 ms.

258% increase in performance.

NUMBER OF VIDS: 2

Position	Procedure	Num. of Calls	Time/Call (ms.)	Total Time (ms.)
1	free	17	0.9	16
2	malloc - total time	138	0.2	32
3	malloc - free list search	138	0.1	16
4	lcreat	42	0	0
5	ldel	17	0.9	16
6	lwrite	83	1.5	128
7	**lread**	**1772**	**0.5**	**800**
8	**lrewrite**	**92**	**0.3**	**32**
9	**execute**	**27**	**2.8**	**76**
10	pad	15	1.1	16
11	filter (not called)	-	-	-
12	**tsk_loop**	**17**	**10.3**	**175**
13	ret_afo	13	6.2	80
14	afo_init	14	3.4	48
15	**port_num**	**59**	**7.4**	**435**
16	port_cr	26	13.2	343
17	af_deliver	17	10.1	172
18	snd_obj	9	5	45
19	get_obj	17	8.5	144
20	fr_init	1	14	14
21	msg_chk	37	1.7	64
22	init_afo_to_vid	1	0	0
23	**af_go**	**1**	**751**	**751**
24	init_test_done_flags	1	0	0
25	lv2s_init	1	0	0
26	send_initial_msg	1	64	64
27	af_exec	17	62.2	1058
28	time (removed)	-	-	-
29	exec_afo	13	20.5	266
30	send_remote	4	0	0
31	serialize_msg	4	0	0
32	snd_msg	4	0	0
33	sync_self (with SIM)	17	44.2	752
34	external_input	29	5.6	163
35	af_deliver (from ext_input)	12	11.6	139
36	deserialize_msg	12	1.3	16
37	service creation	1	32	32
38	**time - initial to completion**	**1**	**2294**	**2294**
39	AF initialization (static)	1	846	846
40	AF execution (dynamic)	1	1450	1450

Total time required for execution (without SIM or Time task) = 1551 ms.

293% increase in performance.

NUMBER OF VIDS: 3

Position	Procedure	Num. of Calls	Time/Call (ms.)	Total Time (ms.)
1	free	11	0	0
2	malloc - total time	128	0.3	32
3	malloc - free list search	128	0.1	16
4	lcreat	42	0	0
5	ldel	11	0	0
6	lwrite	77	1.9	144
7	**lread**	**1588**	**0.4**	**672**
8	**lrewrite**	**64**	**0.3**	**16**
9	**execute**	**16**	**3**	**48**
10	pad	15	1.1	16
11	filter (not called)	-	-	-
12	**tsk_loop**	**12**	**9.7**	**116**
13	ret_afo	11	5.8	64
14	afo_init	14	3.4	48
15	**port_num**	**59**	**8.5**	**499**
16	port_cr	26	13.8	359
17	af_deliver	11	6.8	75
18	snd_obj	9	5.1	46
19	get_obj	11	4.4	48
20	fr_init	1	14	14
21	msg_chk	16	2	32
22	init_afo_to_vid	1	0	0
23	**af_go**	**1**	**751**	**751**
24	init_test_done_flags	1	0	0
25	lv2s_init	1	0	0
26	send_initial_msg	1	79	79
27	af_exec	25	58.4	1460
28	time (removed)	-	-	-
29	exec_afo	11	29.6	326
30	send_remote	6	0	0
31	serialize_msg	6	0	0
32	snd_msg	6	0	0
33	sync_self (with SIM)	25	41	1024
34	external_input	33	2.6	85
35	af_deliver (from ext_input)	8	7.6	61
36	deserialize_msg	8	2	16
37	service creation	1	16	16
38	**time - initial to completion**	**1**	**2719**	**2719**
39	AF initialization (static)	1	844	844
40	AF execution (dynamic)	1	1875	1875

Total time required for execution (without SIM or Time task) = 1695 ms.

209% increase in performance.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-187451		

4. Title and Subtitle	5. Report Date
Knowledge Representation into Ada Parallel Processing	
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Tom Masotto, Carol Babikyan, and Richard Harper	
	10. Work Unit No. 549-03-31-03

9. Performing Organization Name and Address	11. Contract or Grant No.
The Charles Stark Draper Laboratory, Inc. 555 Technology Square Cambridge, MA 02139	NAS1-18565
	13. Type of Report and Period Covered

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration** Langley Research Center Hampton, VA 23665-5225	Contractor Report
	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Sally C. Johnson
WRDC Technical Monitor: Victor R. Clark
Final Report for Task 10

**U.S. Air Force
Wright Research & Development Center
WPAFB, OH 45433-6523

16. Abstract

The Knowledge Representation into Ada Parallel Processing project is a joint NASA and Air Force funded project to demonstrate the execution of intelligent systems in Ada on the Charles Stark Draper Laboratory's Fault-Tolerant Parallel Processor (FTPP). Two applications were demonstrated--a portion of the Adaptive Tactical Navigator and a Real-Time Controller. Both systems are implemented as Activation Framework Objects on the Activation Framework intelligent scheduling mechanism developed by Worcester Polytechnic Institute. This report details the implementation, results of performance analyses showing speedup due to parallelism and initial efficiency improvements, and suggested further areas for performance improvements.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Parallel Processing Fault Tolerance Computer Architectures Expert Systems Knowledge-Based Systems	Unclassified - Unlimited Star Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified		

NASA FORM 1626 OCT 86