(4)

RADC-TR-89-205
Final Technical Report
October 1989

AD-A216 328

DISTRIBUTED SYSTEM INSTRUMENTATION

Carnegie-Mellon University

ITT Research Institute

Z. Segall, D.F. Vrsalovic, J. Kownacki, Byung-Hoon Suh

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS) At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-89-205 has been reviewed and is approved for publication.

APPROVED: Mary L. Denz

MARY L. DENZ
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Igor G. Plonisch

IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-89-205
6a. NAME OF PERFORMING ORGANIZATION Carnegie-Mellon University 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)
6c. ADDRESS (City, State, and ZIP Code) Computer Science Department Pittsburgh PA 15213	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-87-D-0094
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 62702F; PROJECT NO. 5581; TASK NO. QC; WORK UNIT ACCESSION NO. 01
11. TITLE (Include Security Classification) DISTRIBUTED SYSTEM INSTRUMENTATION	
12. PERSONAL AUTHOR(S) Z. Segall, D.F. Vrsalovic, J. Kownacki, Byung-Hoon Suh	
13a. TYPE OF REPORT Final 13b. TIME COVERED FROM Jan 88 TO Oct 88 14. DATE OF REPORT (Year, Month, Day) October 1989 15. PAGE COUNT 144	
16. SUPPLEMENTARY NOTATION This contract is with IIT Research Institute (IITRI). Work was performed by Carnegie-Mellon University under subcontract with IITRI.	
17. COSATI CODES FIELD: 12; GROUP: 07; SUB-GROUP:	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Performance Instrumentation Distributed Systems

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The first part of this report provides the rationale and justification for the design features of PARADISE - PARAllel and Distributed Instrumentation System Environment. The second part describes, in a bottom-up manner, the PARADISE design, and a set of feasibility studies culminating with a sample execution program. PARADISE is a distributed instrumentation system working on DISE under CRONUS and uses a variety of tools and an Integration Platform.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Mary L. Denz	22b. TELEPHONE (Include Area Code) (315) 330-3623 22c. OFFICE SYMBOL RADC (COTD)

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Executive Summary

This document is the Task 0002 Final Technical Report, entitled "Distributed System Instrumentation," under RADC contract F30602-87-D-0094. The report is organized into two parts. The first part provides the rationale and justification for the design features of PARADISE -- PARAllel and Distributed Instrumentation System Environment. The second part describes, in a bottom-up manner, the PARADISE design, and a set of feasibility studies culminating with a sample execution program. PARADISE is a distributed instrumentation system working on DISE under CRONUS and uses a variety of tools and an Integration Platform. In choosing the design approach, special attention has been paid to fulfill the design requirements described in RADC PR. No. B-8-3501, using proved in-practice techniques.

To this end, a number of instrumentation systems have been evaluated. The systems evaluated include: IDT (Honeywell), PIE (CMU), FIAT (CMU) and Para-Sights (Encore). In many ways, PARADISE incorporates the most valuable technology from all of these systems.

PARADISE's main conceptual idea is the relational model, a well known and proven model used in many applications, including databases. Applying this model to an instrumentation system is conducive to high programmability and flexibility, as exemplified by the PIE system. Using a well known query language, the User can ask high level questions about, say, the performance of his program. Then the system will determine the necessary sensors to be enabled/disabled and present the user with a suitable high level response to the query.

Another conceptual idea of the PARADISE design is the incorporation of a set of tools, enabling automated instrumentation, automated experimentation and automated fault injection. The intellectual value resides in their capability of capturing information from the program and/or user and being able to use it for automating the process they are supporting.

Finally, a set of mechanisms for instrumentation and fault injection are described in detail. These provide a highly efficient way to observe and fault inject. Most of these mechanisms have been chosen after a careful feasibility study, and have been proven to be suitable (in terms of performance, intrusiveness and functionality) for CRONUS and the DISE heterogeneous distributed environment.

Part I
Rationale and Justifications for the Design

Table of Contents

List of Tables

1. Paradise Design Justification

This section contains a description of the considerations and justifications involved in the Paradise design.

1.1. Paradise Rationale

In the most general sense, instrumentation is the support for the behavioral and functional observability of a computer system throughout its complete life cycle.

RADC is currently developing and supporting a distributed system test bed, DISE, along with its user level operating system, CRONUS, and an integration platform (IP) for a set of tools. This test bed and its associated tools will be used to develop and evaluate a set of distributed algorithms and applications requiring such features as:

- locally and geographically distributed computing
- parallel computing
- real-time computing
- fault-tolerant and highly dependable computing
- security
- heterogeneous computing
- property validation (predeployment and post-deployment)

In this context instrumentation is perceived as a critical component of the test bed, being geared toward supporting the evaluation and property validation of the above RADC algorithms and application features.

One would successfully argue that the quality of a test bed is directly proportional with the quality of its instrumentation. There are a set of general requirements for such an instrumentation system such as non-intrusiveness, programmability, flexibility, etc. However, all of these requirements have to be judged in the light of their suitability to perform a certain relevant task. Hence, our view of instrumentation for DISE/CRONUS is to suit those requirements to the task at hand. For example, if one needs to measure an object operation with an execution time in the hundreds of milliseconds range, there is little need for zero intrusiveness instrumentation (nearly impossible to achieve). In this case, software implemented instrumentation having a controlled intrusiveness in the range of fractions of a millisecond will be sufficient.

As such, we have identified the performance and dependability evaluation as targets for the design of the Paradise system. Aside from the fact that these two areas of application are central to the DISE/CRONUS test beds, we perceived the need to design a system with realistic expectations and cost.

1.2. Paradise Organization

Paradise system provides support for performance and dependability properties evaluation. As such, it supports the concept of observability through instrumentation for performance evaluation [Gregoretti 86]and of fault-injection for dependability evaluation [Segall 88a], [Segall 88b].

In general, an instrumentation system has three components:

- mechanisms — "how" to observe and/or fault inject
- expertise — "where" and "when" to observe and "where," when and "with what" to fault inject
- automated tools — applying in an automated way the expertise to the mechanisms.

Mechanisms could be further classified into:

- Monitoring mechanisms and policies
- Stimulus mechanisms
- Environment control
- Time management

1.2.1. Monitoring Mechanism and Policies

The monitoring mechanism assumes the existence of a sensor object which provides the following functionalities:

- **Detection** manifests the occurrence of a given event. Detection may be *passive* or *active*. In the former case the event itself triggers the sensor mechanism, in the latter the sensor actively samples the system, looking for the occurrence of an event.
- **Isolation and filtering** determines if the event which has taken place has some significance for the current monitoring policy and should be reported or discarded. Its purpose is to reduce the amount of information derived from sensors; its simplest form is the *enable/disable* (E/D) mechanism. A disabled sensor is transparent to the system in the sense that no information is collected. The enable mechanism may be *static* if the E/D information is contained in the sensor itself and may be modified only by reprogramming it. In a *dynamic* E/D mechanism, the E/D information is external to the sensor and accessible both to it and to an external agent which could change it without reprogramming the sensor.
- **Notification** transfers the occurrence of the event, possibly associated with a timestamp and with other information relevant to the event, to an external environment.

1.2.1.1. Policies

The policy mechanisms deal with the instrumention of each relevant level of the distributed system using the underlying monitoring mechanism.

The levels of observability of a distributed program execution may widely range from microcode instruction execution to the overall distributed program behavior. For each level, one may define a set of objects which are meaningful to that level and a set of events related to those objects. Examples of such multilevel policies are:

- **Hardware Policy (HP).** The objects defined to be visible for instrumentation are those

directly related to the distributed computational process, namely Processors, Memory Units and Communication channels. Examples of events observable at this level are the Cache Hit/Miss ration of a processor, the utilization factor of a memory unit and contention for communication over a common bus. Sensors at this level are not user or application dependent and the policy will be that of a profiler with a rigid and predefined set of observable events.

- **UNIX Kernel Policy (KP).** By the term "kernel" we encompass here the UNIX Operating System functions, including typical kernel mechanisms such as context switching and interrupt handling, as well as policies such as scheduling, memory management,and I/O handling. Examples of visible objects at this level are Processes, Memory Pages, Messages and Ports, I/O Routines and data structures. The KP has a structure similar to that of a profiler. Monitoring functions are predefined and rigidly assigned to sensors embedded in the kernel. Nevertheless, sensors at this level are provided with a limited form of filtering capability in order to be able to focus on a restricted part of the computational status and reduce the amount of data produced.
- **CRONUS Run-Time Support Policy (SP).** At this level, observable objects are instantiations of User level Operating System defined entities. Here, typical significant events are the invocation or the termination of an entity. Such a policy has the structure of a profiler with a more sophisticated filtering mechanism.
- **Application Monitoring Policy (AP).** This is an application level monitor. Sensors placement as well as object and event semantics are User language and application dependent.

1.2.1.2. Integration

The integration among policies has to be organized in such a way that manipulation and retrieval is straightforward and may be tailored by the User or the system to filter only the information required for a specific application.

1.2.1.3. Presentation

The last mechanism element is related to information presentation. This may take a substantially different forms depending upon whether the information is targeted for a human user or for other programs. In the first case, the presentation of the information may take into account various human and application user interface requirements. In the second case, the information format will have to follow the standard software engineering practices for interfacing distributed programs.

1.2.1.4. Paradise Instrumentation Mechanism Considerations

Monitoring mechanisms in Paradise have the following features:

- Detection -- Done by means of software sensors [See Part II, Section 2.3.1]. The reason for choosing software sensors is related to the requirement for portability between heterogeneous hosts. Hardware or hybrid sensors are less intrusive, however, they are not portable and are somewhat harder to manage and interpret.
- Isolation and filtering are done in three ways [See Part II, Section 2.3.2 and 2.3.3]:
  - software enable/disable on a dynamic basis, system wide
  - count/non-count events
  - composite events on a dynamic basis through the mechanism of subordination, local to each object.

The reason for choosing the above features of isolation and filtering is to minimize the amount of information required to travel through the system and, hence, minimize intrusiveness. By the same token, the concept of subordination provides a mechanism to support the programmability requirement through the relational model at the monitoring mechanism level.

- Notification is done through an attachment [See Part II, Section 2.3.4] local to each object. The attachment concept is found to be a suitable way for the flexible and transparent integration of instrumentation with the CRONUS object model. Note that the same concept will work with non-object oriented systems, thus providing portability. To further enhance the filtering capability, an enable/disable notification mechanism for sensor is provided network wide.

There are a number of policies, hence the policy in Paradise is multilevel in nature. It is based on the relational model, which provides at each level the capability to define a set of objects and the relations between them. The choice of the relational model provides for substantial programmability in a well structured system (object and relations) [Segall 83] as well as the availability of well known and understood query languages to support this feature. Alternatives to this design decision are various ad-hoc policies and the Honeywell event-action model [Bhatt 87] . Ad-hoc policies are not conducive to programmability, flexibility and the integration between policy levels. The Honeywell event-action model would fit the Paradise requirements quite nicely. However, due to the fact that the event-action model requires the user to deal with an additional unfamiliar language and that little practical experience is available for this model, we decided in favor of the relational model which does not have the above handicaps.

The proposed initial policies for Paradise are:

- UNIX policy with the following objects:
  - Nodes
  - Processes
  - Communication object
  - Scheduler
  - I/O devices
- CRONUS policy with the following objects:
  - Cronus kernel
  - System managers
  - Object managers
  - Cronus communication
- Application policy with the following components:
  - CRONUS specific
    - Objects
    - Clients
    - Operations
  - Language specific:

- Syntax
- Semantic (some)

- Application specific
  - Data structures
  - Application constructs

Common to all of these policies are the relations of time and usage. Obviously, other relations (simple or derived) could be programmed using the relational model. Example of such relations include tracing and sampling.

The integration and presentation mechanisms in Paradise are also based on the relational model. Aside from the above argument in favor of the relational model, the implementation of this paradigm assumes the existence of a system wide repository for objects and relations. This repository fits nicely into the Integration Platform proposed by Honeywell. Through the Integration Platform, the objects and the relations become accessible to both programs and the User. Should the Integration Platform be an object oriented database, this would provide a positive step toward optimizing the types of accesses the Paradise policy mechanisms would most likely perform.

1.3. Stimulus Mechanism

A stimulus mechanism is correlated to the particular system property to be validated. In the case of dependability evaluation, the stimulus mechanism is fault injection. Fault injection is a controlled corruption of one or more components of the distributed system. For a complete description of the fault-injection validation process, see [Segall 88a], [Segall 88b] .

From the perspective of the instrumentation system, fault injection can be considered as an application of instrumentation and, hence, external to the scope of the instrumentation system. However, for efficiency reasons, fault injection needs to be integrated with the instrumentation system. The description of the integrated fault injection mechanism follows.

The Paradise fault injection mechanism should answer the following questions:

- "how" to fault inject -- use fault injection attachments to corrupt memory contents [See Part II, Section 2.4].
- "when" to fault inject -- synchronized by the Paradise sensors [See Part II, Section 2.2.2 and 2.3.2].
- "where" and "with what" -- expertise is captured by the automated tools [See Part II, Sections 4.1.4, 4.2.4, and 5.1.2].

The Paradise fault injection policies follow the structure of the instrumentation mechanism policies providing a multilevel relational model based on the set of policies at the UNIX level, CRONUS level and Application level. The filtering policies provide for both transient fault and permanent fault features. Furthermore, the integration and presentation mechanism are identical to the instrumentation mechanisms. The fault injection mechanism design is based on experience

gained with the FIAT system, but is far from being identical. The main difference is related to the adaptation of the relational model for monitoring, integration and presentation, as well as a substantial integration with the instrumentation systems.

1.3.0.1. Environment Control

In order to compare and understand two or more executions of an instrumented workload, one has to know about the exact conditions in which the test bed has been used. Furthermore, in preparing the experiments, specific commands have to be available to determine the environment configuration. Accordingly, Paradise provides control of the following:

- Workload control [See Part II, Sections 3.1 and 4.1.1]
- Multilevel instrumentation control [See Part II, Sections 3.2 and 4.1.2]
- Fault injection control [See Part II, Sections 3.3 and 4.1.3]
- Data collection [See Part II, Sections 3.4 and 4.1.4]

1.3.0.2. Time Management

The last set of mechanisms in Paradise deals with time. For instrumentation purposes only, it is assumed that time is available in a consistent way, network wide, and its accuracy is under 1 millisecond. Although we do not provide an explicit design for such a time facility, we suggest that the above goal could be achieved through the addition of a radio receiver to each DISE station to access a world wide clock (used in astronomical observation, etc.). This approach has been already used a number of times with no apparent problems. Alternatively, a software implementation using Lamport's clocks could provide the same functionality with some added intrusiveness.

1.3.1. Automated Tools

The set of automated tools available in Paradise have two main characteristics:

1. Support the relational model exported by the mechanisms.
2. Encapsulate instrumentation and fault injection expertise in an automated way.

There are two such tools in Paradise [See Part II, Section 5] :

PPROC — responsible for syntax/semantic information extraction and implantation of sensors.

POPROC — does the required attribute extraction for fault injection.

The tools extract development time views, such as the syntactic/semantic view of a language program, and then automatically instrument the program for either performance evaluation or dependability evaluation.

The policy definition step could be handled for each level through the use of these tools.

Integration and presentation in Paradise is handled through the relational model and an automated tool which supports it through a graphical multiwindow interface. The tool, PARASCOPE, is somewhat identical to PIESCOPE [See Part II, Section 5.4] in functionality and

design. PIESCOPE and its associated technology has been proven to be quite successful in practice. To our knowledge, PIESCOPE is unique and we do not know of any such tool with similar capabilities. Further discussion on PIESCOPE could be found in [Segall 85], [Gregoretti 86], [Segall 88c].

1.4. Librarians and the Experiment Support Tool

There are two Librarians in Paradise:

- Workload Librarian [See Part II, Section 5.1.1]
- Fault Injection Librarian [See Part II, Section 5.1.2]

Each librarian has particular expertise in the definition, managing and archiving, respectively, program modules and fault classes. Although these tools are not usually found in an instrumentation environment, they are required to support the automated experimentation process in the context of the Honeywell Integrated Platform.

For the same reasons, an Experiment Definition and an added support tool is proposed. Not an instrumentation tool proper, it automates the definition and execution of experiments. Given that the librarians and the experiment definition tools are somewhat optional in an instrumentation environment, but are necessary in an experimentation environment, we suggest that such tools be included in the overall design.

1.5. Other Instrumentation Systems

There have been several other experimental instrumentation systems developed inside and outside CMU. In order place the Paradise design in perspective, we will first give short descriptions for the better known experimental instrumentation systems:

- IDT developed by Honeywell [Bhatt 87]
- PIE developed at CMU [Segall 88c]
- FIAT developed at CMU [Segall 88a]
- Para-Sights developed at Encore Corp [Aral 88].

1.5.1. IDT

Honeywell developed the DSW (Distributed Systems Workbench) in order to facilitate software development of distributed applications in ADA. DSW is composed of several parts:

- Development tools for distributed applications in ADA.
- An environment to assist in performance and other types of assessments.
- An environment to control experimentation.
- Run time support to accelerate experimentation.

The IDT (Instrumented Distributed Testbed) represents one possible hardware base for the

Honeywell's DSW. It supports the "Event-action" model. The IDT consists of two components:

- A high performance communications network with a set of computer nodes. These nodes are running the distributed application under study.
- A reconfigurable instrumentation system which collects data from test runs of the application under study.

The IDT nodes are processors capable of running substantial applications written in ADA. Each node consists of a commercial 32 bit microprocessor with memory and dedicated peripherals including a hard disk and CRT. These nodes are interconnected by a network of 10MHz busses. The interconnection structure is reconfigurable on demand, so various types of networks can be emulated.

1.5.1.1. IDT Instrumentation

The IDT instrumentation functions allow experiment monitoring and controlling. The User can specify the observability of an element as well as the methods for data collection and presentation.

This instrumentation system uses dedicated hardware in order to lower the intrusiveness of instrumentation into the application under test. The dedicated IDT instrumentation resources consist of a dedicated instrumentation processor, memory and specialized hardware for event detection and signalling.

1.5.1.2. The Event - Action Model

The IDT instrumentation functions according to the "Event - Action" model.

From the IDT perspective, events are occurrences that are detectable by instrumentation. For example:

- A packet is sent
- A packet is received
- A communication error occurs
- A statement is executed
- A value is out of limits, etc.

Such events are made visible via special event detection mechanisms. An event which is signalled by such a mechanism is considered to be a simple event. Multiple simple events which have a specific relation can form a composite event. The IDT supports a mapping-table facility which relates events to the actions to be taken upon their occurrence.

The instrumentation actions can be any type of operation(s) which are to be performed upon the detection of a simple or a composite event. Some typical practical examples of instrumentation actions are:

- Counting of different types of events
- Timestamping

- Various relational operations on events
- Various arithmetic operations on events
- Statistical event data processing
- General alteration of the control flow of an experiment in order to provide for the adaptive behavior of the application under test.
- Graphic or alphanumeric displays of the experiment data.

The IDT mapping table provides a many-to-many mapping between events and actions. In addition, the creation of a composite event can be also considered as an action.

1.5.1.3. Experimentation in IDT

The process of experimentation in IDT consists of six distinct phases:

- Experiment definition
- Testbed preparation
- Experiment specification
- Experiment establishment
- Experiment execution
- Post experiment data analysis.

In order to facilitate experimentation in all the phases mentioned, IDT provides the Experimentation Specification Language (ESL). ESL comes with a library of predefined functions that can be used as "actions" during experimentation. The ESL is a high level language. An experiment description in ESL is translated by the ESL translator into event-to-action maps. These maps are loaded into the system during the experiment establishment phase.

The ESL also supports statistical and behavioral performance measurements.

1.5.1.4. Data Presentation in IDT

In addition to the customized actions that can be specified by the User using ESL and the "Event - Action" model, IDT provides a set of prepackaged display functions such as:

- Histogram displays in bar or pie chart form
- On-line time plots
- Scrolling textual displays
- Tabular displays

In addition to data presentation, standard functions to store the experiment data are also provided.

1.5.2. PIE

The PIE project (Performance Efficient Parallel/Distributed Programming and Instrumentation Environment) supports the complete design process from modeling (i.g. prevention), to monitoring (e.g. bottleneck detection), to run-time (e.g. avoidance).

PIE [Segall 85] views parallel processing in the context of "implementation machine" (IM) models. IMs are User templates which provide low level process synchronization and communication details for the programmer. The User can thus concentrate on algorithm design and implementation to a greater degree than previously possible.

The PIE system's approach tends to eliminate performance degradations due to classical structured approaches by introducing "virtual" rather than "ph 'sical" layers. The virtual structure is available during program development time when such abstractions are required to assist in understanding complexity. By run-time, however, the structure has been flattened and removed yielding higher performance parallel programs.

PIE also embraces the concept of "programming for observability" [Gregoretti 86] in which Users make use of visual tools to aid in the development, testing, and, debugging of the application. During development, the PIE system incrementally builds a view of the User program's semantic structure. During testing and debugging, the PIE system allows the User to view the execution of the program (in either an on-line or post-mortem fashion). It is the contention that the extra information gained from the visual displays will assist the User to think more clearly and more in depth about the program's behavior.

The present PIE environment consists of several components:

- PERMOD [Vrsalovic 84] is a modeling tool which provides performance prediction in the early design stages of parallel systems.
- MPC (Multiprocessor C) [Vrsalovic 88] is a C preprocessor that converts special MP (Multi-Processor) language constructs into C program syntax. It implements the "Consistent Abstract Shared Data Type Implementation Machine" (CASDTIM) model. Despite the fact that the target machines can be of different architectures, MPC provides the CASDTIM model to the User via synchronization, and shared memory constructs.
- PIEman implements a relational model for each PIE IM. All PIE tools share data via the relational model.
- PIEmacs is a Gnu-Emacs based editor which extracts development time data about the target program and assists in instrumenting it for the purpose of run-time monitoring.
- PIEscope allows all development and run-time data to be presented to the PIE User in graphical form.
- PIEmon supports the collection and storage of run-time events via the use of sensors.

The following sections discuss only the MPC and PERMOD portions of PIE. However examples given throughout the paper are illustrated by the graphical outputs from PIEscope.

1.5.2.1. PERMOD - Performance Evaluation Model

PERMOD, a model for predicting the performance of algorithms composed of repeated iterations, called *application cycles*, on multiprocessors is derived in [Vrsalovic 84].

The applicable parallel systems consist of a number of processors (N), each having its own local memory. All the processors are connected to a set of global resources via an interconnection structure. Each processor has a variable speed defined as the speed of a referent processor (whose relative speed is 1) multiplied by some factor **p**. The speed of the global resources is defined in the same way by a factor **q**. The interconnection structure allows for the access of any processor to any global resource in FIFO order with a throughput of **r**. Notice that **r** is not exclusively a hardware parameter due to the fact that an application itself has to be able to take advantage of the multiple global access possibility. In order to clarify this statement let us assume for the moment that the interconnection structure is built from **r** parallel buses connecting processors to a set of interleaved memory modules. In such a case there will be **r** parallel global accesses at a time if and only if the application can distribute the data among these modules in such a way that there is no conflict during the access. It will be shown latter that **PERMOD** also gives acceptable results [Vrsalovic 84] for systems where local memories are replaced by caches so long as the hit ratio is high and the portion of code misses are factored into the data access global time.

1.5.2.2. MPC - Multiprocessor C

Section 1.5.2 introduced the concept of the implementation machine, or IM. Unlike the typical virtual machine approach which relies on very generalized, high level interfaces which are reflected in the run-time structure of the code, the *implementation machine* approach translates the User code into target machine code using only low level calls to the run-time system.

MPC is a special preprocessor which translates MP syntax into a C program. It consists of three distinct parts: an analyzer, a constructor, and a target code generator.

The analyzer takes an MPC program as input, which the constructor then converts to a C program. Although the resulting C program may differ from machine to machine, the original MPC program need not be changed. The analyzer also assists in instrumentation of the MPC program so that run-time performance data can be collected. In the present implementation, the target code generator is the C compiler. In the linking stage of the C compiler the User should use the MPC runtime support library.

The MPC language is modeled directly on C, thus allowing parallel processing application programmers to use a language with which they are already familiar. All standard C commands and constructs are recognized by MPC. Identifiers, however, cannot begin with `mp_` or `MP_`, since the constructor uses these as prefixes for internal identifiers. Consequently, virtually any program (noting the above mentioned exception) that compiles under C, will also compile under MPC.

The current version of MPC supports the *Consistent Abstract Shared Data Type Implementation*

Machine or CASDTIM model. Implementation machines differ in the manner in which synchronization and communication are handled. MPC exports CASDTIM to Users via several new constructs that allow for efficient parallel algorithm design, including:

1. **ACTIVITIES**: Sequential units of computation that are spawned and executed in parallel with the creating function.
2. **JOIN AND DETACH STATEMENTS**: Commands that allow activity management.
3. **FRAMES**: An encapsulation of global data and operations on that data. Frames are shared among specified activities and/or C functions and thus represent shared abstract data types.
4. **SYNC AND DSYNC STATEMENTS**: Meta constructs that provide for: synchronization of parallel activities and mutual exclusion for specific parts of data in frames.
5. **TEAMS**: Groups of activities and frames composing a unique subsystem with an associated communication and synchronization structure.
6. **SENSORS**: Locations for collecting information on parallel program execution during run time.

1.5.2.3. PIEMAN

Central to the original PIE thesis is a shared, relational database of information concerning the applications and experiments using those applications. Additionally, the database manager must notify the other components of the system when some shared object is updated so as to maintain system-wide consistency. The current version of the database manager (called PIEman) is implemented on top of the INFORMIX[R] relational database package. Communication is via IPC as implemented in MACH [Rashid 87].

1.5.2.4. PIEMACS

This paragraph describes PIEmacs, the current implementation of the MPOE originally envisioned in the PIE proposal. The MPOE was specified to be an editor[1] with special knowledge of the User programming language. The MPOE uses this special knowledge to create an image in the PIE database (See Section 1.5.2.3) of the interesting constructs in the User's program. The MPOE also maintains a mapping from the objects in the image to their positions in the User's code. This map function enables the User to *navigate* through his application using PIEscope.

The original specification of the MPOE called for a editor that could work with a multitude of languages. The current implementation, however, is just a prototype so the decision was made to support only one language: MPC. The methods used for handling MPC turn out to be well suited for just about any statement-based[2] language.[3]

---

[1]The current version of PIEmacs is written on top of the GNU Emacs editor.

[2]Heretofore, *languages*, when used in the generic sense, refers to statement based languages

[3]LISP mode has also been investigated for PIEmacs but it requires a slightly different approach.

1.5.2.5. PIESCOPE

PIEscope provides a graphical views of the development and execution of a User's MPC program.

PIEscope uses the X-Windows, Version 10, windowing system. X-Windows was chosen as it is becoming the *de facto* window manager in university environments, and at CMU-CSD in particular. The PIEscope receives program updates via messages from the relational database manager, PIEman. Another way to look at PIEscope is as graphical server: any application can be written to take the place of PIEman and can use PIEscope's views without strictly being part of PIE.

Currently, PIEscope understands general **classes** of program constructs, so any programming language which supports these classes can be handled (See Section 1.5.2.4 for details of these classes).

PIEscope provides three development-time views and three execution-time views. The development views are:

roadmap
: a tree-like display of the **definition** structure of the User's program.

use roadmap
: a tree-like display of the **instantiation** and static **invocation** structure of the User's program

sensmap
: similar to the **roadmap** but also includes the User's explicitly-placed sensors. The User uses the sensmap view to enable or disable the sensor firings during the program execution.

The execution-time views are:

barscope
: a bar graph of the execution of the User's program.

animation tree (*NYI*)
: a tree-like display which replays the dynamic **invocations** (and destruction) of the structures in the User's program.

max-animation tree (*NYI*)
: similar to the **animation tree** except that the destructions are not shown, so the User can see the maximum amount of resources used by the program.

Each view has many features for zooming in and filtering the viewed data which are not described here.

1.5.2.6. PIEMON

The PIE performance monitor is a facility for observing computations. It is multi-level, consisting of User, run-time and kernel levels.

A monitor observes and records *events*. An event is an observable, time-stamped object occurring during the execution of a computation; it is the basic unit of information for observability. Events consist of two basic types, *control-driven* and *data-driven*.

- A *control-driven* event is more than just an obfuscating name for a state of a state machine because it not only designates a specific logical point (state) in a computation's

control flow, but includes the time when that state was reached and, occasionally, ancillary computation data requested by a User. Examples of control-driven events are the inception and termination of processes or the start of an iteration of a program loop.

- A *data-driven* event is a time stamped modification of, or demand for, computation data. Data-driven events do not contain direct information about computation states, but enmasse they describe data access patterns. Although inferences can be made about what computation states are possible for a specific data-driven event, they can be made only after comparing the event to where the datum is used in the computation's text and with an analysis of the execution history provided by control-driven events.

Sensors detect the events of a computation and prepare them for retrieval by collection instrumentation. This instrumentation is a software/hardware system which appends an event to the event record of the computation. After an execution terminates, *PIEman* selects and filters the events in the event record using a relational data base and any relevant performance or status goals requested by the User. The relational data base, constructed at development time, contains the static structures of a program as well the semantic and temporal relations between them. The structures contain sensor marks so that events collected during execution can be mapped onto their corresponding computation

Events are observed by a *monitoring* environment which extracts development and run-time information about sequential and parallel structures of a computation, and about its execution. The assemblage of mechanisms and protocols that make up this monitoring environment is called the *monitor*. As discussed earlier, our definition of "monitor" is distinguished from three other common uses of the term by the observational sense which is ascribed to it. Other meanings include (1) synonymy with "operating system," (2) the User interface for an operating system, and (3) a data object responsible for synchronizing and controlling multiple accesses to critical sections of code. Each of these familiar uses ascribes notions of resource management and control to the term "monitor." The monitoring mechanisms consist of *sensors* for marking events, instrumentation for event collection and a relational *model* that ties together the development and execution time data of a computation.

1.5.3. FIAT

FIAT stands for Fault Injection-based Automated Testing environment. Since each of these keywords represents part of the goals and concepts of the system, we will first examine each keyword and its significance.

1.5.3.1. Fault Injection

In reality, in FIAT, we are able to inject both the fault itself as well as its manifestation in terms of, say, triggering the error detection recovery mechanisms (EDRM). This is accomplished through the use of *software* fault injection mechanisms. The concept of fault injection is not new. Organizations committed to the development of highly dependable hardware have, for many years, been using hardware fault injection to evaluate the relative effectiveness of alternative hardware error detection designs.

As hardware error detection mechanisms have been enhanced by this process, solid state circuit design and manufacturing technologies have also been steadily improved, decreasing the rate at which errors due to hardware faults are generated. The combined effect has been to significantly lower the rates at which errors due to hardware faults are generated in systems, and to greatly improve the probability that hardware-caused errors will be detected and compensated for when they do occur.

Unfortunately, comparable progress has not yet been made in understanding, and compensating for, errors caused by primary software faults (software "bugs") and by secondary faults in software caused by undetected hardware errors.

Accordingly, the FIAT environment has been designed for the injection of error patterns into executing software that are representative of errors that are likely to be generated by software and hardware.

The function of this fault injection testing process will be to uncover deficiencies in a system's error detection and recovery mechanism (EDRMs), and to guide tradeoffs between alternative design enhancements, by providing quantitative evaluations of their relative effectiveness.

The FIAT environment provides experimenters with facilities for defining fault classes (relationships between faults, and the error patterns that they cause); for specifying (e.g., relative to the source code of an application) where, when, and for how long errors will strike; and how they will interact with executing object code or data.

The FIAT environment then automatically locates the designated object code or data from its source code designation, and takes action against it.

In its initial version, FIAT software can fault inject User application code and data, and can inject faults into messages (corrupted, lost, delayed), tasks (delayed, abnormal termination), and timers. Subsequent versions will extend these fault injection capabilities into operating systems, and will enable hardware fault injection under FIAT control.

The keyword *testing* refers to both testing of error detection/recovery mechanism as well as quantitative evaluation of the dependability property of the system under test. As mentioned before, the test set for EDRM happened to be a set of fault/errors. This is the set that has to be injected in order to make sure that the EDRM are working properly.

Automated refers to the critical issue of the complexity of the fault injection process. As in testing, the quality of the results will be a function of the capability of the system to inject (test) as many faults as possible per unit of time. This, in fact, means automated support at *development time*, as well as at *run time*, for the fault injection process.

The keyword *environment* refers to the high degree of integration of the FIAT system. The

components of the system, workloads, fault classes, experiments and data analysis (further detailed below), are integrated under one comprehensive User interface supporting the process of preparation, debugging, run time control and data analysis.

To summarize, the goals of the FIAT project are:

- Development of an environment for automated software fault/error injection and error detection/recovery (coverage) analysis.
- Fault-free characterization of the system under investigation (performance and profiling and EDRM testing).
- Relative contrasting of two or more fault-tolerant technique (with a substantial software component) dependability properties.

1.5.3.2. The FIAT Process and Its Abstractions

To achieve the above goals, the following experimental phases are to be applied to the system under test.

- Fault free validation of system: Profile system software components' performance characteristics and collect data.
- Fault free validation of workload: Profile the performance/functionality characteristics of the workload and collect data.
- Fault injection experimentation: Inject faults into profiled workload and collect histories and error records.

To support the fault injection process, the following abstractions are provided:

- Workload
- Fault class
- Experiment
- Data collection/analysis

We now describe each of the above abstractions, as well as the automated tools for manipulating them.

1.5.3.3. Workload (WL)

Users write distributed real-time dependable programs. The FIAT system manipulates *workloads*. A workload is an *observable* set of real-time communicating tasks. A task is an observable (monitorable) scheduled unit of computation communicating through observable communication media named channels.

In order to fault inject a workload at the symbolic level, a number of attributes have to be extracted from the workload. The term attributes refers to the set of symbolic names identifying the tasks in the workload as well as the code and data segments within each task. This analysis is automatically done by a tool known as the *attribute extractor*. The attribute extractor, after analyzing the workload, provides tables (e.g., task tables) known as *domains*. These domains are further used in the process, discussed below, of automatically generating fault lists.

Another aspect of fault injection is how to fault inject a task without breaking the operating system protection mechanism. We solve this issue by linking (at link time) with a number of program attachments which serve as a Trojan horse to fault inject from inside the task on external request.

1.5.3.4. Fault Classes and Fault Lists

A fault class is a template describing a set of workload or system modifications, which are representative of a group of physical/logical faults having common properties. This template is similar to an abstract data type. As in any abstract type, the fault class can be instantiated, meaning that each method is applied to the associated domain and a specific fault (*fault instance*) is generated.

1.5.3.5. Data Collection/Analysis

Data collection (DC) supports two entities: histories and error reports. Histories are records of monitored normal functional and performance events. Error reports are records of exceptions and abnormal events. Data analysis recognizes three hierarchical levels of experiment data processing: run, experiment and multiple experiments. A run is a single fault injection in a specific workload. An experiment is a collection of runs, usually for the same fault class. Multi-experiment analysis refers to the ability to cross section a number of experiments under a specific number of dimensions.

In FIAT two types of data analysis are available: the canned variety and the open variety. The canned data analysis provides a set of predefined functions such as workload profiling and error coverage statistics. The open data analysis is a relational data base query language which enables the Users to define their own analysis goals.

1.5.4. Para-Sights

Para-Sights [Aral 88] is an environment for debugging and profiling of parallel programs. The User can control execution and examine values of program variables by using this environment. Monitoring in Para-Sights is minimally intrusive if a target machine has sufficient parallel resources to spare for monitoring. Some of the main features of Para-Sights are:

- Para-Sight runs as a separate thread of execution in the same address space as the target program.
- Due to address space sharing, Para-Sights can almost nonintrusivelly examine all target variables.
- It provides a dynamic linking facility so various Para-Sights can be linked in on demand without the need for recompilation of a target.
- Para-Sights uses the scan point mechanism, similar to the trace point in common debuggers, in order to pass control and data to an arbitrary instrumentation server called *parasite* running as an independent thread of execution within the same address space.
- Some of the scan points can have the functionality of breakpoints, thus allowing manipulations of the target's control flow.

Para-Sights provides a set of standard User commands. This set is extensible in order to allow for customized commands. Here the basic description of the Para_Sights functionality is discussed.

1.5.4.1. Para-Sights Commands

There are several standard Para-Sights commands:

- run [arg1 arg2 ...]
  - starts procedure *main* in a target. It passes an argc/argv pair to it.
- quit
  - terminates the execution of a target
- hangup
  - sends UNIX SIGHUP signal to threads of execution that are involved in the current execution.
- load [-<port>] <parasite>
  - call to the Para-Sight dynamic loader to load and link in a new code for an independent parallel thread of execution (parasite) to act as a monitoring server. The new thread will have its I/O bound to the Para-Sight console or, if port is different than 0, to a virtual terminal running behind the port.
- dir
  - do list of loaded parasites
- purge <parasite>
  - dynamically remove a parasite
- help [topic]
  - User help for a specific topic

1.5.4.2. Interface to Scan Points and Parasites

Parasite provides an environment for the design of custom parasites and associated Scan Points. There is a set of standard functions which facilitate this process:

- Para_insert(source)
  - inserts a scan point at the source line[4].
- Para_delete(source)
  - deletes a scan point at the source line.
- Para_set(source, trigger_list, quiet)
  - set scan point trigger to a list of names.
- Para_start(name)
  - starts a new thread of execution running the code of the named function.
- Para_display(sp)
  - displays information for a scan point.

[4]Inserts and deletes are atomic operations and can be done while target is actually running.

1.5.4.3. Restrictions of Para_Sights

There are several restrictions on the use of Para-Sights, but most of them have sources in the current implementations. However, there are some, like those connected to the generally nonreentrant UNIX code or libraries, which will require some effort to be fixed. Some of Para-Sight restrictions according to [Aral 88] are:

- The target programs must be compiled with symbol tables included which usually results in larger and nonoptimized code.
- The current directory must contain a symbolic link to a Para-Sight directory.
- The User is responsible to resolve any conflicts in names between Para-Sight and target functions and/or variables.
- Since the target runs as a thread, there are problems with the parallel execution of UNIX kernel and library functions which are mostly nonreentrant.
- Garbage collection of previous broken runs is left to the User.
- Static functions or variables are not visible to the C interpreter.

1.5.4.4. Some Standard Parasites

In addition to supplying a nice interface for building customized parasites, Para-Sight includes three standard parasites. These are:

- File browser
  - is used for the inspection of a text file. It is based on the public domain program less and its primary use is in conjunction with a source level debugger.
- The "C" interpreter
  - its primary function is to allow the User to examine and modify target variables by using the familiar C syntax. The interpreter is a part of Para-Sights but it is loaded and executed on demand.
- Low-Level Debugger
  - is also separately loaded. It provides a function called "low level" which is implemented as a scan point. It is used for:
    - setting breakpoints
    - timestamping
    - counting instructions
    - recording thread ids
    - registering dumps

1.6. Contrasting the Paradise Design with Other Instrumentation Systems

Here we compare the main features of Paradise with the instrumentation systems briefly described in the previous section. Table 1-1 summarizes the findings in the functionality/model dimensions. Table 1-2 contrasts the systems under the properties/user support dimensions.

Next, we will explore some of those dimensions in light of the Paradise features.

1.6.1. Intrusiveness

Paradise uses software instrumentation with dynamic enable/disable to observe and report relevant events. This type of instrumentation imposes some changes in the behavior of the system being observed. On the other hand, Honeywell's IDT uses hardware instrumentation in an attempt to minimize or even to eliminate intrusiveness. Aside from the net advantage of eliminating intrusiveness, hardware instrumentation does have some drawbacks. Some of them are:

- lack of portability
- difficult to program
- difficult to interpret results
- costly
- hard to implement after the fact

One of the main questions is, then, how intrusive is the software instrumentation approach used in Paradise?

There are two types of intrusiveness; namely, the perturbation of absolute execution times of parts of the program observed, and changes in the order of events. While the first type of perturbation can not be completely eliminated, there are two approaches adopted in Paradise to deal with the problem. First, we minimize the perturbation in execution time by choosing an efficient implementation of the instrumentation mechanism. Initial feasibility studies [See Part II, Section 6] have shown the intrusiveness to be under couple of percentage points for a reasonably instrumented application running on top of CRONUS. Second, simple compensation for the cost of the instrumentation is used by PARASCOPE when displaying the behavior of the system under observation. These two techniques compensate, for all practical purposes, the first type of perturbation.

The second type of perturbation, changes in the order of events, is a substantially harder problem. There are no good ways to compensate for this type of perturbation. So, let us ask ourselves the question of how seriously this type of perturbation will influence the system under test.

Distributed systems are known to behave undeterministically with respect to the order of events. Hence, any order of events is possible, including the ones exhibited through the instrumentation perturbation. Then the new question becomes whether all orders of events are equally probable. The answer to this question is still in the research domain. Available results [Segall 88b] seem to point in this direction. Even if the answer to the previous question is negative, one still has to ask when the order of events is important. A naive answer to this question will include at least the issue of the observability of distributed system communication and synchronization.

The truth of the matter is that there are very few well engineered communication and synchronization techniques which depend on the order of events. The real answer to the above question is that the order of events may be important to detect synchronization and communication

bugs. However, in this case, the answer to distributed synchronization and communication debugging is not found in nonintrusive instrumentation, but in *accurate replay* of the run where the error manifested itself Hence, system execution replay is the issue and Paradise makes ample provision for supporting it.

To summarize, after careful exploration of the trade-offs between hardware and software instrumentation, Paradise software instrumentation is implemented with intrusiveness compensation and event replay.

1.6.2. Program Replay in Paradise

As mentioned above, program replay is one of the main mechanisms for distributed system debugging.

Program replay in Paradise is supported by four elements:

- Paradise Preprocessor
- PARASCOPE
- The relational model
- The Integration Platform

The Paradise Preprocessor extracts syntactic and semantic information from the distributed program. This information is organized in the relational form and stored in the Integration Platform.

In addition, the Preprocessor automatically instruments the program. After an execution of a distributed program, the instrumentation mechanism stores the information into the Integration Platform.

Subsequently, PARASCOPE applies relations to both development-time information and run-time information, displaying the execution of the program through flash views of the program text or of a graphical representation of the program entity which generated the current event. Using this approach , unlimited replay of a particular execution could be displayed and analyzed. This feature, inherited from the PIE system, has been proven to be extremely useful.

1.6.3. Programmability and Flexibility

Paradise programmability and flexibility results from the extensive use of relational models and relational query languages at all levels. This allows the user to ask high level questions and let the system figure out the low level details regarding the means to answer the question. In addition, this paradigm captures and stores the expertise required to instrument and to observe the system under test. Ample practical experience with this model [Segall 83], [Segall 85], [Gregoretti 86] enables us to be fairly optimistic on the usability and feasibility of the approach.

1.6.4. Integration with the Honeywell Integration Platform

RADC has commissioned Honeywell to propose a tool integration strategy. The technical approach proposed by Honeywell includes an object oriented data base to act as a repository of data and programs, as well as an integration media among tools. Paradise fully subscribes to this approach and plans to use the Integration Platform for the repository of the development time and run-time information collected. In addition, depending on the performance of the Integration Platform, Paradise may use it for communication among different parts of the system, such as the Preprocessor and PARASCOPE.

Functionality/Model Dimensions					
	IDT (Honeywell)	PIE (CMU)	FIAT (CMU)	Para-Sight (Encore)	Paradise (RADC)
monitoring	hardware	software static	software static	software dynamic	software dynamic
policies	NA	multi-level	two-level	NA	multi-level
integration	event-action	relational	ad hoc	NA	relational
presentation	event-action (graphical)	graphical relational	text relational	NA	graphical relational
Fault Injection	NA	NA	yes	NA	yes
Environment Control	yes	NA	yes	NA	yes
Time	yes	yes	software implementation	NA	TBD
Automated Instrumentation	NA	yes	NA	NA	yes
Automated Fault Injection	NA	NA	yes	NA	yes
Experiment Definition	yes	NA	yes	NA	yes
Overall Model	action event	relational	ad hoc	ad hoc	relational

Table 1-1: Comparison of Functionality/Model Dimensions

Properties/User Support Dimensions					
	IDT (Honeywell	PIE (CMU)	FIAT (CMU)	Para-Sight (Encore)	Paradise (RADC)
programmability	moderate	high	moderate	high	high
intrusiveness	low	moderate	moderate	moderate	moderate to low
portability	low	high	high	moderate	high
flexibility	moderate	high	moderate	high	high
user support	moderate	high	moderate	low	high
experience with the system	low	moderate	moderate	low	NA

Table 1-2: Comparison of Properties/User Support Dimensions

References

[Aral 88] Z. Aral, I. Gertner.
Non-Intrusive and Interactive Profiling in Para-Sight.
Technical Report ETR 88-006, Encore Computer Corporation, July, 1988.

[Bhatt 87] B. Bhatt, W. Heimerdinger.
The Instrumented Distributed Testbed (IDT): A Tool for Prototyping and Evaluating Distributed Systems.
In *AIAA Computers in Aerospace-VI Conference*. October, 1987.

[Gregoretti 86] Francesco Gregoretti, Zary Segall.
Programming for Observability Support in a Parallel Programming Environment.
In *Proceedings of Computer Science Conference*. 1986.

[Rashid 87] R. Rashid, A. Tevanian, M. Young, D. Young, R. Baron, D. Black, W. Bolosky, J. Chew.
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
Technical Report CMU-CS-87-140, Carnegie Mellon University, July, 1987.

[Segall 83] Z. Segall, A. Singh, R. Snodgrass, A. Jones, D. Siewiorek.
An Integrated Instrumentation Environment for Multiprocessors.
IEEE Transactions on Computers c-32(1), January, 1983.

[Segall 85] Zary Segall, Larry Rudolph.
PIE: A Programming and Instrumentation Environment for Parallel Processing.
IEEE Software , November, 1985.

[Segall 88a] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, B. Dancey, A. Robinson, T. Lin.
FIAT -- Fault Injection Based Automated Testing Environment.
In *18th International Symposium on Fault-Tolerant Computing*. 1988.

[Segall 88b] Z. Segall, J. Barton, D. Vrsalovic, D. Siewiorek, R. Dancey, A. Robinson.
Fault Injection Based Automatic Testing: Practice and Examples.
In *8th International Symposium on Avionics, SanDiego*. 1988.

[Segall 88c] Z. Segall, D. Siewiorek, D. Vrsalovic, F. Gregoretti, E. Caplan, C. Fineman, S. Kravitz, T. Lehr, M. Russinovich.
Real Time Status Monitor for Distributed Systems.
April, 1988.
Final Technical Report, Carnegie Mellon University, Computer Science Department.

[Vrsalovic 84] D. Vrsalovic, D. Siewiorek, Z. Segall, E. Schringer.
Performance Prediction and Calibration for a Class of Multiprocessor Systems.
Technical Report, Carnegie Mellon University, August, 1984.

[Vrsalovic 88] D. Vrsalovic, Z. Segall, D. Siewiorek, F. Gregoretti, E. Caplin, C. Fineman, S. Kravitz, T. Lehr, M. Russinovich.
Performance Efficient Parallel Programming in MPC.
Technical Report CMU-CS-88-167, Carnegie Mellon University, July, 1988.

Part II
Design Specifications

Table of Contents

List of Figures

List of Tables

1. Introduction

This document describes a preliminary design for the Paradise system. This design assumes that Paradise is running on the top of DISE in a distributed system consisting of a number of computers connected via a communication network as depicted in Figure 1-1. There is no restriction on the number of nodes or the type of network used. It is also assumed that all the nodes are running the Cronus [Schantz 85, Schantz 86] operating system and that the whole system exports some notion of global time to all of its nodes. In the following text the Paradise nodes are referred to as PNODES.

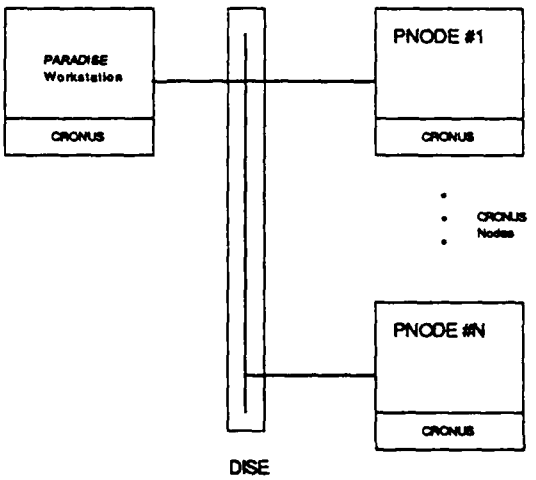

Figure 1-1: Paradise Configuration

At least one of the PNODES in a Paradise system must be able to run a special application which requires specific graphic i/o capabilities (i.e. this node has to be able to run X windows Ver. 10 or higher [Gettys 87, Swick 87]). If a node is running such an application it is referred to as a Paradise workstation, or PWS, in the following text. Note that a PNODE can act as PWS and also run application clients at the same time, but one has to keep in mind the PWS's high resource utilization requirements when doing this.

Figure 1-2. depicts the structure of a single Paradise node. Each node is running a host operating system on top of which resides the Cronus kernel. Paradise introduces four additional functionalities into a system:

- **Workload Control** - implements the mechanisms needed to manage a set of clients distributed in a system. It deals with issues of managing their dynamic behavior: *creation, termination, suspension, continuation, joining* and *signaling*. The workload control mechanisms maintain systemwide unique ids for all the clients. Associated operations are transparent to the locations of their subjects and objects within the Paradise system (i.e. a performer client uses a workload control operation via an unique interface despite the fact that the client this operation refers to may be in a local or some remote node of a system).
- **Monitoring** - implements the mechanisms needed to monitor the execution of a distributed workload and to record the information generated during the execution.

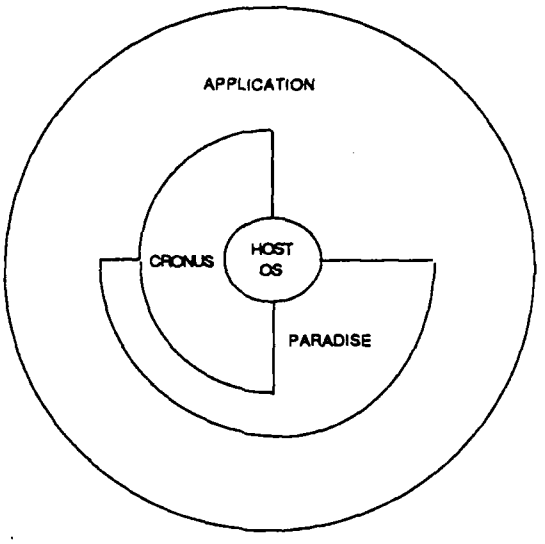

Figure 1-2: Structure of a Paradise Node

These mechanisms deal with the issues of: *identifying* an event, *filtering* out events that are of interest, *isolating* relevant data (like timestamps and various other run-time information) about an event, and *recording* events in a local repository.

- **Fault injection** - implements the mechanisms needed to allow for a corruption of a distributed application. This includes mechanisms for *generation* of special FI triggering events, dynamic *identification* of the FI targets and *administration* of faults.
- **Data collection** - implements the mechanisms needed to retrieve event data from the local repositories and transfer them to a PWS in control of an experiment.

During the execution of any parallel/distributed application each PNODE runs a set of Cronus clients associated with the particular application (these are referred to as Application Clients, or APCs) and a special process called the Paradise Monitor and Controller, or PMC. All four Paradise support mechanisms are distributed between the PMC and APCs in each PNODE. The PMC in each PNODE communicates with the APCs in the node and also with the PWS which is currently in control of the system.

The PWS component associated with the run-time control of an experiment is called the Paradise Experiment Manager, or PEM. PEM consists of the following components:

- WM - Workload Manager
- MM - Monitoring Manager
- FIM - Fault Injection Manager
- DCM - Data Collection Manager

Each component which encapsulates a specific functionality in the PWS has a counterpart in the PMC which is referred to as a *controller*. The following controllers compromise the PMC:

- WC - Workload Controller
- MC - Monitoring Controller

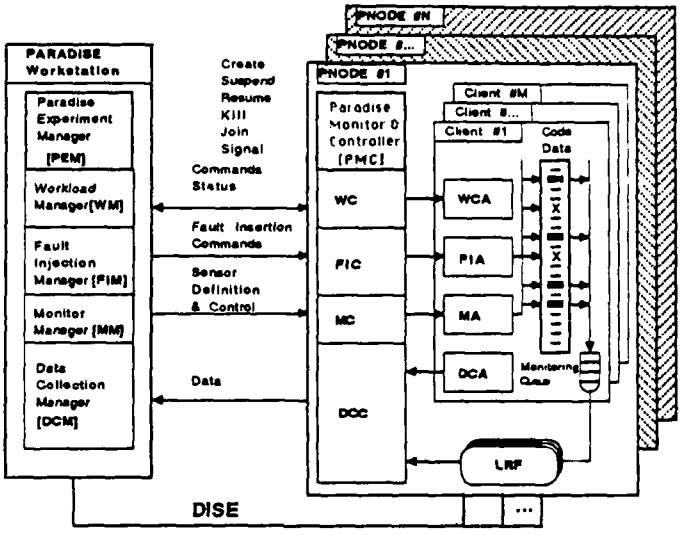

Figure 1-3: Distribution of Paradise Mechanisms

- FIC - Fault Injection Controller
- DCC - Data Collection Controller

Similarly, each part of the PMC has a counterpart in each of the APCs. These are called the APC *attachments*. The following attachments are found in an APC:

- WCA - Workload Control Attachment
- MA - Monitoring Attachment
- FIA - Fault Injection Attachment
- DCA - Data Collection Attachment

Subcomponents of each of the functionalities are distributed and supported in the PWS, PMC, and APC as illustrated in Fig 1-3. The interaction is by means of messages.

After this brief introduction, the following text will discuss the specific design details by using a bottom-up approach. In this manner the four attachments will be discussed first in Chapter 2. Next, the Paradise Monitor and Controller (PMC) design will be discussed in Chapter 3. Chapter 4 will discuss the design of the Paradise Experiment Manager (PEM) which constitutes the Paradise Workstation (PWS) support for experimentation. Finally, Paradise tools that constitute the PWS software are discussed in Chapter 5.

After the preliminary Paradise design is presented, results from a small feasibility study are given in Chapter 6. These results present some basic performance measures and illustrate the more than acceptable low level of Paradise intrusiveness into workload execution. Also, an example application, run and monitored under Cronus and Paradise, can be found in the Appendices.

In the conclusion we present how the Paradise Run-time support mechanisms fit into the complete

Paradise picture.

2. Application Client Attachments

One can envision a distributed application with a number of APCs running in parallel in different nodes of a system. Some of them may even have low enough resource requirements that they can run concurrently in the same node. Cronus supports node independent client code since the object related operations are transparent to the node boundaries.

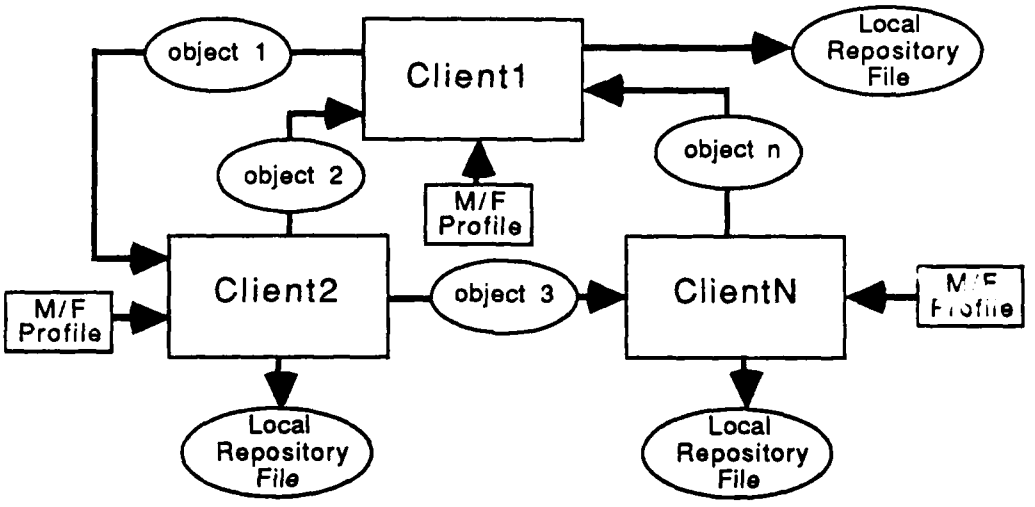

Figure 2-1: Dynamic view of a Typical Paradise Workload

Figure 2-1 depicts a dynamic view on a typical Paradise workload. There is a set of APCs communicating via shared Cronus objects and storing monitoring data into local repositories provided by the Paradise System.[1] There are several operations performed on the control or data flow of an APC which consequently break its abstraction. In the Paradise system, these operations are performed by a set of "Trojan horses" called the APC *attachments*. Due to the fact that Cronus doesn't explicitly support an alternative control flow within an APC in a portable way, Paradise must rely on the available host operating system mechanisms to implement such a support function. This must be done specifically for each of the host operating systems in use. In order to clarify this statement, let us assume for a moment that the host OS is UNIX. In such a situation Paradise can take advantage of UNIX signals to implement alternate control flow mechanisms for the attachments.

An APC is generated from a User source code in several steps. First it is processed by the Paradise Preprocessor (PPROC), which analyzes the source and instruments it. After this, the resulting code is processed by the Cronus compiler and finally by a C compiler. During the linking process the attachments are added to an APC executable from the Paradise libraries. Figure 2-2

[1]Paradise local repositories are files which will be described in detail in section 4.1.4.

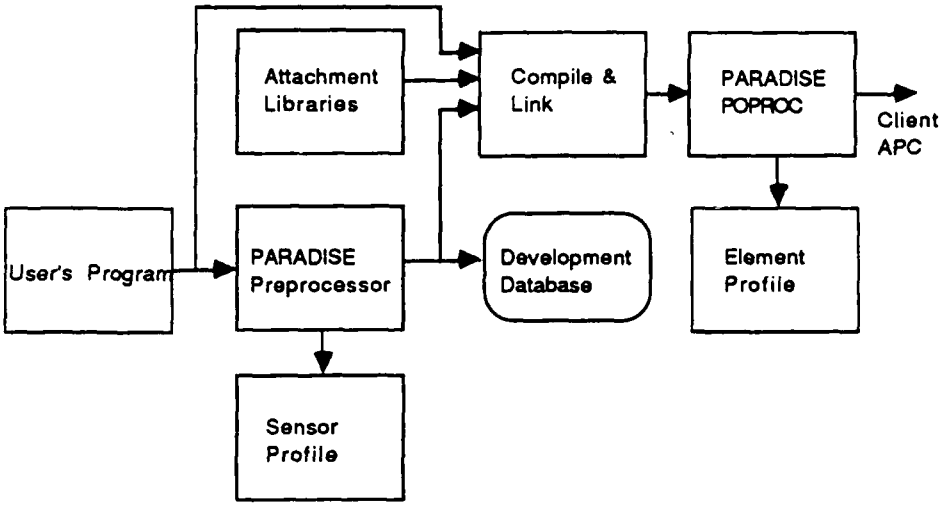

Figure 2-2: Generation of an APC

depicts this process.

2.1. Implementation of Attachments in UNIX

2.1.1. General Structure of an APC

Figure 2-3 depicts the structure of typical APC source code after it has passed through the Paradise Preprocessor PPROC. It consists of the customary User defined C code for a UNIX process with several additional statements inserted by PPROC.

```
....APC Procedures...
......(user code)....
.....................

extern void
P_attachments() ;                    /*(inserted by the PPROC)*/

main(argc,argv)
int argc; char **argv;
{
      P_init(&argc,argv);            /*(inserted by the PPROC)*/

      ..............
       (user code)
       (including Paradise calls)
      ..............

      P_exit();                      /*(inserted by the PPROC)*/
}
```

Figure 2-3: Structure of a Typical APC

There are three basic PPROC inserts in this example[2]; these are indicated by the statement comments.

First, an external procedure, P_attachments, is declared. This procedure defines code to be run on command in order to invoke attachment functions.

Second, immediately after starting, each APC executes the P_init() call in order to:

- Create a communication peer to receive attachment commands (i.e. in UNIX this is done by creating an attachment socket per APC);
- Create an alternative control flow to execute P_attachment functions on demand (i.e. in UNIX this can be done by setting up P_attachments as a signal handler for SIGIO and by setting up the attachment socket to work in the interrupt mode);
- Initialize the attachments;
- Notify the creator (i.e. the local PMC) about its startup and report its UNIX pid, Cronus id and the attachment socket name (or any other internal Paradise ids needed for this matter); An entry is created in the PMC database using this APC information.

Third, prior to exiting, the APC will execute the P_exit() call which will:

- Remove the APC's entry from a local PMC database;
- Perform closing functions for the attachments;[3] Terminate an APC;

2.1.2. The P_attachment() Function

Each of the attachments consists of three functionally distinct parts:

- initialization
- body
- closing

Initialization and closing operations are performed by P_init() and P_exit(), respectively, while the P_attachment() function implements the attachment bodies. Figure 2-4 depicts the structure of the P_attachment function used as the SIGIO signal handler in a UNIX implementation.

It is important to remember that the P_attachments() function is used as an UNIX SIGIO handler and that it will be invoked every time a command message is sent to the APC's attachment socket.[4] The format of attachment messages is defined by the content of the first byte, specifying the attachment to be invoked. The rest of the command message is then analyzed after the specific attachment has been actually invoked. Then the appropriate action is performed. Due to the fact

---

[2] In addition to these inserts, PPROC will insert a number of other sensors into an APC automatically depending upon the code structure; Sensors are described in detail in Section 2.3.

[3] In the case of multiple exits in an APC, PPROC must insert P_exit() calls at all the exits.

[4] This socket is created during P_init() execution.

```
void
P_attachments()
{
   char  buffer[...];

   ............
   receive_command(buffer)        /*command message*/
   ............

   switch(att){
       case WCA:
           ........
           break;
       case MA:
           ........
           break;
       case FIA:
           ........
           break;
       case DCA:
           ........
           break;
       default:
           ........
   }
   send_reply(buffer);            /*reply message*/
   ............
}
```

Figure 2-4: Structure of the P_attachment Function

that the attachment code performing the action is an actual part of an APC,[5]there are no restrictions on the kind of actions an attachment can perform. At the end of an operation a reply message is sent to the requester in order to deliver the result status of an attachment action. The default part of the switch statement in Figure 2-4 provides for associated messages involving error recovery and PMC-APC protocol issues.

There are several commands associated with each attachment. As depicted in figure 2-4 these commands are sent in the form of a message to an attachment.

There are four attachments to an APC as presented in Figure 2-5. The first byte of the message, from a PMC to an attachment, serves as the control code and determines the attachment to be invoked. Paradise attachment control codes are as follows:

- 1 - WCA
- 2 - MA
- 3 - FIA
- 4 - DCA

[5]Please remember that attachments are "trojan horses".

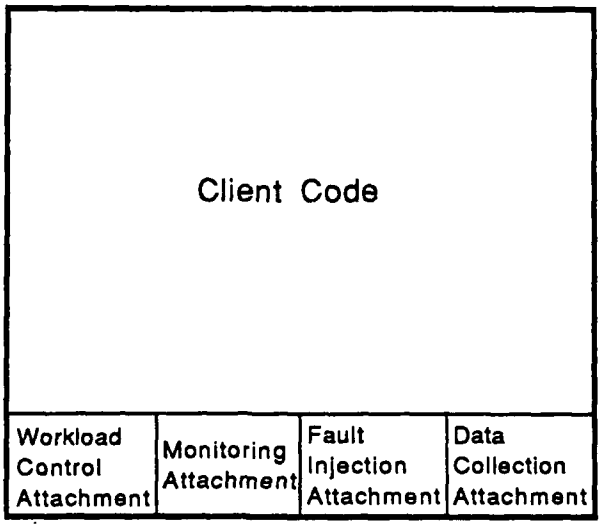

Figure 2-5: APC Structure

The command message formats for different attachments and their particular functions are discussed below.

2.2. Workload Control Attachment

Paradise workload control mechanisms implement a set of functions designed to create, control and synchronize distributed workloads. There are two main groups of workload control mechanisms: *tasking* and *signaling*. In the UNIX implementation of Paradise, both groups use underlying UNIX mechanisms while the attachment code properly deals with the required interfacing details for invoking these mechanisms.

A Paradise environment is a set of string variables that are automatically inherited by all the children of an APC. In this way some global values which are not likely to be changed during an experiment can be shared by all the APCs.

2.2.1. Paradise Environment

WCA supports the notion of the Paradise environment. Each APC inherits the Paradise environment of its creator. A User can control its environment via the following functions:

- P_get_environment(*env_var*) - returns a pointer to the content of the environment variable with the name equal to *env_var*. If *env_var* doesn't exist, it will return NULL.
- P_set_environment(*env_var, val*) - sets the value of *env_var* to the string *val* points to. If the *env_var* doesn't exist, WCA will create one. If val is a NULL pointer, *env_var* will be removed.

The Paradise environment is implemented as an array of characters consisting of records

separated by blanks. Each record represents an environment variable in the following format:

```
%env_var_name={identifier,number}.
```

The maximum size of an environment array is 256 characters.

There are three important predefined environment variables for each APC:

- The first one, called P_queue, defines the size of the monitoring queue.[6]
- The second one, named P_creator, holds the creator's pointer in the Paradise standard notation form. This notation consists of two integers separated by a ':' as the delimiter (i.e. xxx:yyy). The first integer defines the creator's node id, while the second one defines its APC id.
- The last one is P_profile and contains the name of a file holding monitor and fault profiles for an APC.[7]

In each distributed application there will be one initial APC which was started manually from a shell. Such an APC will have father's APC id of -1 and father's node id equal to its own.

2.2.2. Monitor and Fault Profile

Sensor and fault profiles control the functional characteristic of the MA and FIA, respectively. There are actually two lists located in the APC attachment. The first one is called P_MON_PROF and consists of sensor control blocks. The second one is called P_FAULT_PROF and consists of fault control blocks. Upon creation (i.e during execution of P_init), the APC fills these lists from the file pointed to by the Paradise environment variable %P_profile.

```
.................
1 31 10 20 10 2 7 14 1 3
.........................
.........................
@
.........................
3c500 8 05c3214f 1 16
.........................
.........................
```

Figure 2-6: Monitor and Fault Profiles

Figure 2-6 depicts the layout of the sensor and fault profiles. Records are written one after another separated by the CR character. Sensor records are written first and fault records after them. The two groups are separated by the '@' character. The example shows one monitor control block and one fault control block.

The various fields in the example have the following interpretations:[8]

---

[6]Please consult Section 2.3.4 on Notification.

[7]Monitor and fault profiles are initial contents of monitor and fault control blocks in an APC. The specifications for monitor and fault control blocks are given in Sections 2.3 and 2.4.

[8]Please refer to the layouts of the monitor (Section 2.3) and fault (Section 2.4) control blocks.

- SENSOR CONTROL FIELDS:
  - enable [= 1] - sensor will be initially enabled
  - status [= 31] - sensor will use counter, will be repetitive, will enable its subordinates when itself active, will record its event and will apply the faults listed in its control block.
  - prolog_base [= 10] - the first 10 times the sensor fires it will be inactive (i.e. its filtering function will return 0)
  - act_base [= 20] - the next twenty times the sensor fires it will be active (i.e it will:
    - store its event record in the local queue
    - on the first active fire it will enable its subordinates (with id = 7 and id = 14)
    - on the last active fire (i.e. 20th) it will disable the same subordinates
    - on each fire it will apply fault id = 3 until its repetition counts runs to 0
  - epilogue_base [= 10] - the next ten execution times the sensor will be inactive again
  - there are 2 subordinate sensors (with sensor_id = 7 and sensor_id = 14)
  - there is 1 fault target with fault_id = 3
- FAULT CONTROL FIELDS:
  - start [= 3c500] (virtual address)
  - the fault mask is 8 bytes long and its value is: '05c3214f'
  - the function to apply the mask is bitwise logical AND (i.e. 1)
  - repetition count [= 16]

2.2.3. Paradise Signals

The Paradise system implements its own set of signals. There is a set of 32 signals available to the User. They are used within an APC via the following commands:

- P_signal(client_id, signal) - sends a signal to a client specified by a client id in the Paradise standard notation.
- P_handler(signal, handler) - sets up the procedure pointed to by the handler parameter to act as a signal handler for the specified signal.

2.3. Monitoring Attachment

Monitoring attachments consist of mechanisms required to support the monitoring activities in Paradise. There are four phases in the monitoring process:

- Detection
- Filtering
- Isolation
- Notification

2.3.1. Detection

Detection in monitoring can be accomplished in two ways: active (i.e. sampling) or passive (i.e tracing). Paradise predominantly uses the latter technique by means of sensors. Sensors are inserts into the User code by the PPROC which analyzes the code and instrumentscertain predetermined syntactic constructs. In addition to these, the User can directly add additional sensors to the source code. Due to the fact that a sensor is a part of APC's control flow, detection is passive (i.e. a sensor detects an event only at the time that the code segment containing the sensor is executed). On the other hand, in situations where the sampling technique is necessary, the User can write a custom signal handler to perform sampling upon demand.

Figure 2-7a depicts the C source for a typical APC with a main program consisting of one *while* loop. Along with other code, a call to a procedure (i.e. procedure1) is executed inside the loop. As explained before in Section 2.1.1, the P_init() and P_exit() functions from the Paradise run-time library are inserted by the PPROC. PPROC will also automatically insert a number of sensors. Figure 2-7b shows several such automatically inserted sensors. In this example PPROC instrumented the following locations:

- the beginning and the end of main function
- the top and the bottom of the while loop
- the entry and the exit of procedure1 call

There are two important *points to notice at this point*. First, PPROC will automatically assign ids and types to any associated group of sensors. Second, in addition to the regular exits from various syntactic constructs, PPROC will also instrument all irregular exits (i.e. the break from the while loop in Figure 2-7). PPROC will also do the same in the case of *goto* or *exit* statements, but in the latter case it will insert a P_exit() after the added APCEND sensor.

A Paradise sensor is the C macro construct depicted in Figure 2-9. It accepts two input parameters: a type and an id. Each APC gets a P_MON_PROF array (See Figure 2-8) which contains a set of sensor control blocks. The sensor code is executed each time an event is detected. The first step is to check whether or not the enable flag in the sensor control block (associated with the the executed sensor id) is set. If this flag is not set (i.e. the sensor is disabled), APC execution continues as if there were no sensor inserted. As a result, there is very low overhead per disabled sensor.

If a sensor has been enabled, the filtering function will first be performed. Then, depending on the result of filtering, execution will or will not proceed. If the sensor execution proceeds, the isolation phase is executed next. During the isolation phase all necessary data will be extracted from the run-time data structures and then entered into the queue during the notification phase.

```
main(argc,argv)
int argc; char **argv;
{
    P_init(&argc,argv);
    ..................
    ..................
    while(....){
        ..........................
        if(.....) break;
        ..........................
        procedure1(.............);
        ..........................
        ..........................
   }
   P_exit();
}
```

a.

```
main(argc,argv)
int argc; char **argv;
{
    P_init(&argc,argv);
    P_sensor(P_APCSTART,id1);
    ..................
    ..................
    while(....){
        P_sensor(P_LOOPTOP,id2);
        ..........................
        if(.....) {
            P_sensor(P_LOOPBREAK,id2);
            break;
        }
        ..........................
        P_sensor(P_PROCENTRY,id3);
        procedure1(.............);
        P_sensor(P_PROCEXIT,id3);
        ..........................
        ..........................
        P_sensor(P_LOOPBOTTOM,id2);
   }
   P_sensor(P_APCEND,id1);
   P_exit();
}
```

b.

Figure 2-7: A part of an APC without (a.) and with (b.) sensors inserted

```
typedef struct {
        boolean     enable;
        int         status, state;
        int         act_counter, prolog_base, act_base,
                    epilogue_base;
        int         subordinates[MAX_SUBORDINATES];
        int         fi_targets[MAX_FITARGETS];
} P_sensor_ctrl_blok_type;
```

Figure 2-8: Paradise Sensor Control Block

```
#define  P_sensor(TYPE,ID)                    \
         if(P_SEN_CTRL_BLK[ID].enable)        \
            if(P_filter(ID))                  \
               P_notify(P_isolate(TYPE))
```

Figure 2-9: Internals of a Paradise sensor

2.3.2. Filtering

One can assume that filtering is performed in two steps. First, the enable flag is tested and then, if true, the filtering function continues. The exact behavior of the filtering function depends on the status word of the sensor set by the User, and the current internal state set during execution.

The Sensor Status Word contains five bits and is defined as follows:

- 2^0 - COUNTER -Use the counter if set. The counter can be in three states: prologue active and epilogue. Filtering is successful during active count and unsuccessful during other two states. The current state is recorded in "state". Each time the counter reaches zero it is reloaded with the proper base and the state is set accordingly.
- 2^1 - REPETITIVE - If not set, self-disable while entering epilogue state.
- 2^2 - INDIRECT - If set, enable sensors listed as subordinates[9] when entering the active state and disable them when entering the epilogue state.
- 2^3 - RECORD - If set, record event as defined by the type parameter.
- 2^4 - FAULTINJECT - If set, apply the faults specified by the listed fi_targets, (which are indices into the fault profile). Note that the original contents of the targets are saved before fault injection.
- 2^5 - RESTORE - If set, restore corrupted segment to the content it had before the last faultinjection.[10]

2.3.3. Isolation

Several actions are taken during the isolation phase for a Paradise sensor. First, a timestamp is taken[11]; then the performer id, which is the Cronus client id, is retrieved. After that, this data, together with the sensor id and its type, are stored in the sensor record.

In order to make sensors as efficient as possible, the sensor record is produced in a machine dependent and variable format. The Data Collection Controller in the PMC will later transform all monitored data into Cronus canonical types to provide a uniform picture to PWS.

---

[9]Only sensors in the same APC can be subordinates!

[10]Please note that FAULTINJECT and RESTORE are mutually exclusive due to the fact that, if both are applied, no net change will occur.

[11]The Paradise system expects that the local clock is synchronized with all other clocks in the system, or that PWS has accurate data about their current interrelationship. We will discuss later in this text what to do if DISE does not export this notion of global time.

The type of the sensor record is defined by the content of its first byte[12].

2.3.4. Notification

Sensor records created during the isolation process are first stored into a local monitoring queue.This is a simple FIFO queue created during the execution of a P_init() call. The size of this queue is initially defined at APC creation by the Paradise Environment.[13]Later, during execution, this size can be adjusted by a command to the monitoring attachment. When a local monitoring queue fills up, it will be emptied during the next active sensor execution. The content will be written into a local file which is also automatically created during the execution of the P_init() call.[14]

2.3.5. Monitoring Attachment Commands

The attachment commands are sent from the PMC to the APC in the form of a message. The first byte of this message determines the attachment to be invoked. Based on the definition in Section 2.1.2, the code designation for the MA is 2. The second byte of the message defines the command to be executed. The rest of the format is command dependent. All the parameters are given in the local format due to the fact that the conversion is done in the PMC. These are the commands related to the monitoring attachment:

- 1 - Set monitoring queue size. The size is given as an integer and represents the number of event records that can fit in the queue. If the size is made smaller than the number of records currently resident in the queue, it will be flushed prior to resizing.
- 2 - Flush the content of a monitoring queue into a local repository.[15]
- 3 - Enable sensor(sensor_id)
- 4 - Disable sensor(sensor_id)
- 5 - Set sensor status(value, sensor_id)
- 6 - Set base count(value, counter, sensor_id), where the counter is one of the following:
  - prologue
  - active
  - epilogue
- 7 - Add subordinate(subordinate_id, sensor_id)
- 8 - Remove subordinate(subordinate_id, sensor_id)
- 9 - Remove all subordinates(sensor_id)
- 10 - Add fault(fault_id, sensor_id)

---

[12]Please notice that the exact layout of the sensor record will depend not only on the cpu type but also on the host OS. As an example, take the PC-RT which uses three different floating point formats depending on the operating system.

[13]Please refer to Section 2.2 for an exact definition of the Paradise Environment.

[14]Please consult Chapter 2.5 for details on data collection.

[15]The monitoring queue is also flushed during the execution of a P_exit() procedure.

- 11 - Remove fault(fault_id, sensor_id)
- 12 - Remove all faults(sensor_id)

The last byte of the command is a "link" byte. If this byte is different from 0, then the next monitoring command is contained in the same message. In this way several commands can be linked in the same message for efficiency purposes.

2.4. FIA - Fault Injection Attachment

The FIA implements the mechanisms required to perform fault injection in an APC. At the attachment level a fault is represented as a corruption of one or more memory locations.

2.4.1. Fault Control Block

```
typedef structure {
            char    *start;
            int      len;
            char     mask[256];
            char     original[256];
            int      function;
            int      repetition;
}fault_control_block;
```

Figure 2-10: Fault Control Block Layout

Figure 2-10 presents the layout of the fault control block in Paradise. Particular fields in the fault control block have the following meanings:

- start - address of where to start to corrupt memory locations
- len - length of the array to corrupt, in bytes
- mask - pattern mask to be used to corrupt the memory content
- original - original content saved prior to corruption by a fault
- function - function to use to bitwise combine content of the mask and the original memory to generate a new content
- repetition - number of times to apply fault

A fault is executed by an active sensor if its FAULTINJECT flag in the status word is set. Each time after a fault is executed its repetition counter is decremented. When the repetition counter reaches zero, the fault is disabled.

2.4.2. FIA Commands

The attachment commands are sent from the PMC to the APC in the form of a message. The first byte of this message determines the attachment to be invoked. Based on the definition in Section 2.1.2, the code designation for the FIA is 3. The second byte of the message defines the command to

be executed. The rest of the format is command dependent. All the parameters are given in the local format due to the fact that the conversion is done in the PMC.

The following commands are those related to the fault injection attachment:

- 1 - Set start(start, fault_id). Start is given as an address in the virtual space of an APC. The fault id is the index of the relevant fault control block in the fault list.
- 2 - Set mask(size, mask, fault_id). Set the mask size and contents for a fault.
- 3 - Set function(funct, fault_id). Functions are as follows:
  - No action
  - bitwise logical AND
  - bitwise logical OR
  - bitwise logical XOR
  - move mask
  - add character (i.e. the first byte of the mask is used as a character to add to a byte pointed to by start)
  - add integer (i.e. the first four bytes of the mask are treated as an integer to be added to an integer pointed to by start)
  - add float
- 4 - Set repetition count (count, fault_id) sets the repetition count in the fault control record to count.

2.4.3. Higher Level FI mechanisms

In practice there are many situations where an experimenter is interested in upsetting some higher level mechanisms. Typical examples of these are: communication mechanisms, timers etc. All of these high level upsets can be translated into low level memory corruptions provided that the structure of the supporting code is known. As an example, one could envision the emulation of a message drop fault by changing some flag in the communication driver procedure. In order to obtain such results most effectively, such places to corrupt in order to emulate higher level faults in Cronus or host OS code should be carefully selected and preprogrammed.

2.5. Data Collection Attachment

Sensors write their event records into the local monitoring queue. The size of this queue is determined by the Paradise environment variable *P_queue*. Each time this queue fills up, its contents are flushed into a special file. This file, which is created for every APC during the execution of the P_init() call, is assigned a standard name built from the prefix "P_S", the UNIX pid of an APC and the version number concatenated into one string. An example of a monitoring file name would be P_S1235_1. The '1' at the end shows that this file is the first version. Each additional file created after a flush would be correspondingly designated by an incremented version number.

DCA supports a special command which allows for the dynamic retrieval of monitoring data without the need to interrupt the ongoing monitoring process. This command directs that a file P_Sxxxxx_n be closed and that another file P_Sxxxxx_n+1 be opened and submitted to the DCA in an APC. The advantage of this operation is that the APC continues undisturbed in monitoring the execution, while the previously recorded events are available for analysis.

3. PMC - Paradise Monitor and Controller

The PMC is a Paradise server which runs in each PNODE. It is implemented as a Cronus object manager with an associated set of commands. Its functionality is divided among four controllers:

- Workload Controller (WC)
- Monitoring Controller (MC)
- Fault Injection Controller (FIC)
- Data Collection Controller (DCC)

Each of these controllers provides support for a related subset of Paradise User commands utilized in workload APCs. In addition there are several other general functions which are unrelated to any of these controllers.

3.1. WC - Workload Controller

The WC supports commands for the creation and synchronization of APCs. It maintains a list of all active APCs in the local PNODE. Each record in the list is referred to as an **APC control block**.

```
typedef struct{
        UID             ProcID;         /* Cronus UID of process     */
        DATE            Timestamp;      /* Time process was created */
        int             PID;            /* The UNIX process ID       */
        int             group;          /* The group membership      */
        int             Type;           /* Type of creation          */
        int             Status;         /* Status of execution       */
        char            *ProcName;      /* Name of the process       */

} P_APC_control_block;
```

Figure 3-1: Layout of an APC Control Block

Particular fields in the APC control block have the following meaning:

- ProcID is an identifier given to an APC process by Cronus.
- Timestamp is the time of creation.
- PID is an identifier given to the APC by the host OS. For example, in the UNIX implementation of Paradise, this field is identical to the UNIX pid for an APC.
- group is the id of the Paradise group of which the APC is member. (APCs may be associated by means of a User designated assignment to a *group*.) Each APC inherits its group id from its father. It is stored in its Paradise environment variable *P_group*. If this environment variable is nonexistent in an APC's Paradise environment, it is assumed to be 0 by default.
- The variable status reflects the status of an APC.
- Each APC retains the name of its object file in *ProcName*. For UNIX implementations this variable is identical to argv[0] for a particular APC.

The APC id of each APC is defined by its index into the APC list. All WC commands return a value

of P_UNSUCC in the case of an error. When this occurs, a description of the particular error can be found in the global variable *P_error*.

An APC keeps a record of all APCs it created by maintaining the respective pointers to their APC structures. An **APC structure** is defined as follows:

```
typedef struct{
        int     node,group,id;
}P_apc;

#define P_NOAPC   (P_apc *)0
```

The WC commands are described in the text below.

3.1.1. P_create() command

```
P_apc   *
P_create(node, executable, argc, argv, para_env, profile)
int         node;
char        *executable;
int          argc;
char       **argv;
char        *para_env;
char        *profile;
```

Creates a new APC by *starting* the executable binary file pointed to by *executable* with *argc* parameters in *argv*. The new APC inherits the Paradise environment pointed to by *para_env*. Returns a pointer to an APC record containing the node and APC ids, or P_NOAPC in the case of a failure.

3.1.2. P_suspend() command

```
int
P_suspend(apc_ptr)
P_apc  *apc_ptr;
```

Suspends an APC pointed to by *apc_ptr*. Returns P_UNSUCC in the case of an error, P_SUCC otherwise.

3.1.3. P_resume() command

```
int
P_resume(apc_ptr)
P_apc *apc_ptr;
```

Resumes an APC pointed to by *apc_ptr*. Returns P_UNSUCC in the case of an error, SUCC otherwise.

3.1.4. P_kill() command

```
int
P_kill(apc_ptr)
P_apc *apc_ptr;
```

Terminates an APC pointed to by *apc_ptr*. Returns P_UNSUCC in the case of an error, P_SUCC otherwise.

3.1.5. P_join() command

```
int
P_join(apc_ptr)
P_apc *apc_ptr;
```

Attempts to join an APC pointed to by *apc_ptr*. Returns P_UNSUCC in the case of an error, otherwise it returns the value of the P_exit() call in the joined APC. The call is nonblocking and the User is responsible for retries in the case of a failure.

3.1.6. P_signal() command

```
int
P_signal(apc_ptr, signal)
P_apc *apc_ptr;
int    signal;
```

Send the *signal* to the APC pointed to by *apc_ptr*. Returns P_UNSUCC in the case of an error, P_SUCC otherwise.

3.1.7. P_get_apc() command

```
P_apc *
P_get_apc(node, group, name)
int       node;
int       group;
char     *name;
```

Obtains an APC record for an APC in *node*, associated with *group* and having *name*. Returns a pointer the APC record, or P_NOAPC in the case of a failure.

3.2. MC - Monitoring Controller

The MC implements commands needed to control the MA (monitor attachment) of each APC in the PNODE. The available commands are:

3.2.1. P_set_queue() command

```
int
P_set_queue(apc_ptr, size)
P_apc  *apc_ptr;
int     size;
```

Sets the monitoring queue of the APC designated by *apc_ptr* to the specified *size*. Size defines the number of event records that fit in the queue. If the size is made smaller than the number of event records currently resident in the queue, the queue will be flushed prior to resizing. Returns P_UNSUCC if the case of an error,P_SUCC otherwise.

3.2.2. P_flush() command

```
int
P_flush(apc_ptr)
P_apc     *apc_ptr
```

Flushes the content of a monitoring queue in an APC designated by the pointer *apc_ptr*.

3.2.3. P_set_sensor() command

```
int
P_set_sensor(apc_ptr, sens_id, action, value)
P_apc     *apc_ptr;
int        sens_id;
int        action, value;
```

Sets a field of a sensor control block. The block is indexed by a *sens_id* in the APC attachment pointed to by *apc_ptr*. *Action* defines the field to be set to the *value*:

- 1 - enable
- 2 - status
- 3 - prolog_base
- 4 - act_base
- 5 - epilogue_base
- 6 - adds subordinate
- 7 - removes subordinates
- 8 - removes all subordinates
- 9 - adds a fault target
- 10 - removes a fault target
- 11 - removes all fault targets

Returns P_UNSUCC in the case of an error,P_SUCC otherwise.

3.3. FIC - Fault Injection Controller

The FIC communicates with the FIA (fault injection attachment) of an APC in order to dynamically change the characteristics of the fault profile in that FIA. The main difference between the MC and FIC lies in the fact that the size of a monitor profile is fixed at compilation time while the size of the fault profile can be changed dynamically during the execution of an experiment. This is done by directing the FIC to add, alter or remove some fault control blocks. (The list of sensor control blocks is static; this means that the monitor profile can only be changed by the MM.)

3.3.1. P_fault_remove() command

```
P_fault_remove(apc_ptr, fault_id)
P_apc     *apc_ptr;
int        fault_id;
```

Removes fault referenced by the *fault_id* from the APC referenced by *apc_ptr*. Returns P_UNSUCC in the case of an error, P_SUCC otherwise.

3.3.2. P_set_fault() command

```
int
P_set_fault(apc_ptr, fault_id, action, value)
P_apc      *apc_ptr;
int         fault_id;
int         action, value;
```

Sets a field of a fault control block indexed by a *fault_id* in an APC attachment pointed to by *apc_ptr* to the value *value*. *Action* defines the field to be set:

- 1 - start
- 2 - len
- 3 - mask[16]
- 4 - original[17]
- 5 - function
- 6 - repetition

Returns P_UNSUCC in the case of an error, P_SUCC otherwise. Note: if the fault profile is not full, a new fault can be added by setting all the elements of an empty fault control block.

---

[16]Value represents a pointer to the mask to be used.

[17]Same as for mask.

3.4. DCC - Data collection controller

In order to minimize execution overhead as much as possible, a decision was made to restrict all filtering to two phase points: the period prior to event notification and/or to the period after the completion of data collection (all data stored in PWS integration platform). Consequently the functionality of the DCC is quite straightforward; there are only two DCC commands and they deal with the LRF (local repository file) of an APC.

3.4.1. P_collect() command

```
int
P_collect(apc_ptr, file)
P_apc      *apc_ptr;
char           *file;
```

This command transfers the content of the LRF of the APC pointed to by *apc_ptr* to the file named *file*.

3.4.2. P_get_vers() command

```
int
P_get_vers(apc_ptr)
P_apc   *apc_ptr;
```

Returns the current version of the APC related LRF, or P_UNSUCC in a case of an error.

3.5. Miscellaneous PMC Functions

Here several functions, which are not directly connected with any of the attachments but deal with general PMC functions, are described.

3.5.1. P_initialize() command

```
int
P_initialize(node)
int        node;
```

Initializes all PMC internal data structures and kills all active APCs that the PMC controls.

3.5.2. P_get_time() command

```
struct P_timeval  *
P_get_time(node)
int        node;
```

Returns the local time of a node.

4. PEM - Paradise Experiment Manager

The Paradise Experiment Manager (PEM) exports a set of abstractions to Users. It also allows Users to apply various operations on these abstractions by means of the mechanisms implemented in the Paradise controllers and the corresponding APC attachments within each PNODE.

4.1. PEM Abstractions

Paradise abstractions, as well as Paradise mechanisms, are grouped into four categories along the lines of associated functionality. They are hierarchically structured and supported by corresponding managers. PEM is at the top of the hierarchy. It supports the abstraction of an **experiment**. PEM supports three basic operations for starting, stopping and saving the results of an experiment in the integration platform which is a Paradise database[18].

In order to start an experiment, a User must provide PEM with the the following types of information:

- description of the workload to be run
- description of all observable points in the workload
- description of all the faults to be injected
- description of the type of data to be collected

Based on these previous statements, one can claim that a Paradise experiment is uniquely defined by four different abstractions, each supported by its own manager:

- Workload
- Monitor profile
- Fault Injection profile
- Experiment Data

4.1.1. WM - Workload Manager Abstractions

The Workload Manager related abstraction is called a **workload**. It is defined as a set of real-time natural or synthetic **clients** (i.e. APCs) communicating via Cronus objects. It is assumed that object operations are implemented by sending and receiving messages. Although these messages are not visible to a Cronus client programmer, Paradise also supports an abstraction called a **link** which is a logical peer for message exchanges in an APC. The purpose of the link is to provide a hook to support the IPC based fault injection in the User's namespace.

A workload is defined by four parameters:

- The first parameter is the name of a special executable APC. This APC is called the **startup application client**, or SAPC. The SAPC is defined by the User and is used to

[18]See Chapter 5 for details.

create the actual clients which constitute a particular workload.

- The second parameter is an (argc, argv) pair to be submitted to the SAPC.
- The third parameter is the Paradise environment to be submitted to the SAPC.
- The fourth parameter is the description of the Paradise virtual machine to run the workload.

4.1.2. Monitoring Manager Abstractions

The Monitoring Manager supports three different abstractions:

- Monitor profile
- Sensors
- Events

A **monitor profile**is a table which specifies the association between various system or User sensors and the ids and types of event records produced. A sensor is an implant placed in an APC by the User or by the Paradise system (i.e. PPROC). For example, in Figure 2-7b, there are two sensors to monitor the entry and the exit of *procedure1*. The monitor profile for this particular example will contain the record that the User known entity called *procedure1* (in the User name space) is associated with two sensors, one of the type P_PROC_ENTRY and another of the type P_PROC_EXIT. It will also keep a record that these two sensors are named id_{xx} (in the run-time name space). The sensors just described are called system sensors due to the fact that they are automatically inserted by the PPROC. However, a User may add any number of his own sensors in order to track values of certain variables, or track executions of arbitrary parts in an APC's control flow.

All types of User sensors, when active, create an event record. This event record has a fixed part and also, in a case of User sensors, a variable one. The fixed part of an event record contains the sensor type and id as well as a timestamp. The content of the variable part of an event record depends upon the type of sensor which created it.

4.1.2.1. User Sensors

There are four different types of User sensors:

- Trace type - is used to notify that execution reached a specified point in an APC control flow. It will create an event with an empty variable part.
- Integer type - is used to notify the User about the value of an integer variable.
- Float type - is used to notify the User about the value of a float variable. The variable part of the event record it generates contains that value.
- String type - is used to notify the User about the content of some string variable. The variable part of the sensor record contains the string in "Pascal notation" (i.e. size,content). The maximal length of such a sting is 255 bytes + 1 byte for length.

A User sensor appears in source code as:

```
P_Usensor(type, id, val_ptr);
```

where *val_ptr* is a pointer to a variable or is P_NULL in the case of a trace type sensor.

4.1.2.2. System Sensors

Paradise system sensors are implanted into an APC source by the PPROC. System sensor event records have only a fixed part which includes an id, type and a timestamp. There are 11 types of system sensors implemented in the Paradise system, but there are no design limitations to increasing this number. A system sensor appears in source code as[19]:

```
P_sensor(type, id);
```

where the *type* can be as follows:

- P_APCSTART - start of an APC
- P_APCEND - end of an APC
- P_LOOPTOP - top of a loop[20]
- P_LOOPBOTTOM - bottom of a loop
- P_LOOPBREAK - break from a loop
- P_LOOPCONT - continue in a loop
- P_LABEL - execution reached a label
- P_PROCENTRY - entry into a procedure
- P_PROCEXIT - exit from a procedure
- P_OBJENTRY - entry into an object operation call
- P_OBJEXIT - exit from an object operation call
- P_OPRBEGIN - manager operation code begin
- P_OPREND - manager operation code end

4.1.3. Fault Injection Manager Abstractions

The FIM supports the following set of abstractions:

- faults
- fault instances
- fault occurrences
- fault classes
- fault profile

A **fault** is an abnormal event in a computer or in the network connecting the computer to the rest

---

[19]Please also see Figure 2-7b and Appendix Figure III.3.1.3

[20]The term *loop* here refers to: **while, for, or do loops.**

of a system. It is represented by a physical/logical **fault instance**. A happening of a fault is a **fault occurrence** defined by a fault instance which occurs in a certain part of a system (i.e. specific PNODE or network) at a certain time.

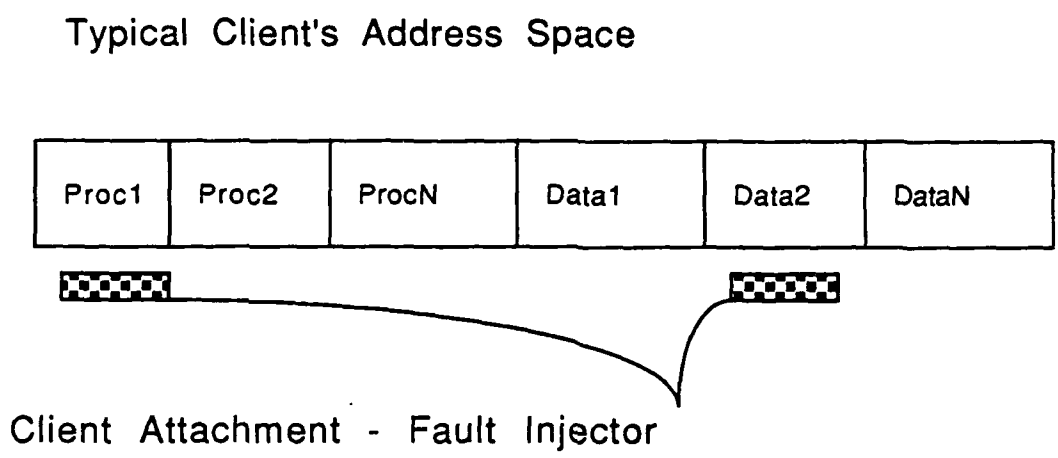

Figure 4-1: Memory Fault Class

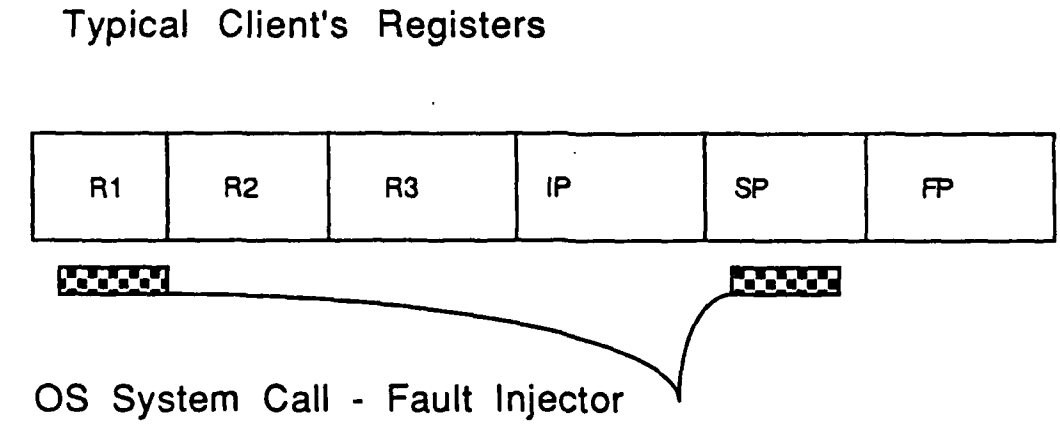

Figure 4-2: Register Fault Class

A **fault class** is a group of fault occurrences with a common denominator. It is a template describing a set of workload/Cronus/HostOS modifications --- actually, *corruptions* --- which are representative of a group of physical and/or logical faults. Figures 4-1 and 4-2 present two examples: memory and register fault classes, respectively.

A **fault profile** is a table which specifies the association between various system or User sensors and the fault occurrences in an APC (See Section 2.2.2).

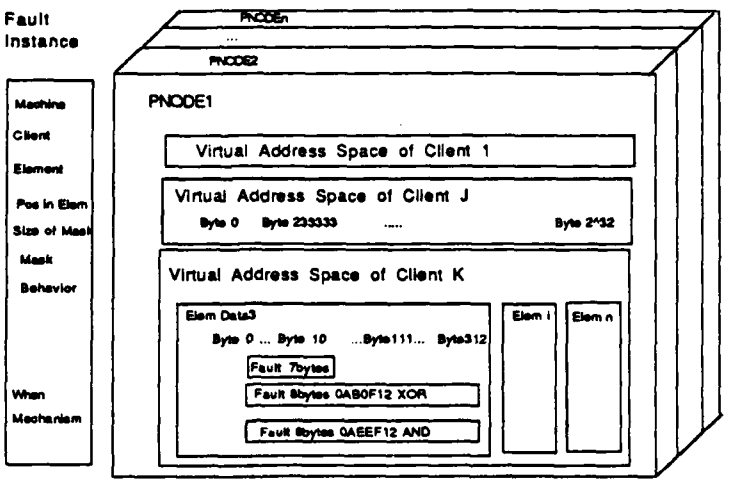

Figure 4-3: Components of a Fault Instance

Figure 4-3 shows the components of a fault instance. It specifies the PNODE in which fault injection is to take place and the APC which is to be affected. It also specifies: an APC element, the position in this element as well as the mask and function to be used.

4.1.4. Data Collection Manager Abstractions

The DCM supports the following abstractions:

- Events
- Local Repository Files (LRFs)

An **event** is a happening of some predefined kind. An event in Paradise can be **observable** or **nonobservable**. Observable events can be **active** or **passive** at some time during the execution of an experiment. In this design the observability of an event can be changed on a static basis (i.e. at compilation time) while the activity of an event can be changed on a dynamic basis (i.e. at execution time).

Active events generate an event record. For performance reasons sensors store event records in a format determined by the machine acting as the execution engine. As a result it is the task of the data collection mechanisms to deliver them to the PWS in the the standard Paradise format.

Event records generated by sensors in an APC are stored in a **local repository file**, or LRF. There is an LRF for each APC in a workload. The data format of the event records in a LRF at this point depends on the CPU and host OS type of the PNODE it resides in. For this reason each LRF carries an identification of these types so that the DCM can correctly interpret the event records. Typically, an APC creates a LRF upon startup and closes it upon exiting. However, there are situations where an APC runs for a long time (or even permanently). In such situations the User can

close the current version of an LRF and redirect future event records to a new version. The old LRF is then available be used for data analysis even though the APC is still executing.

4.2. PEM Commands

The Paradise Experiment Manager (PEM) interprets an experiment script during the execution of an experiment. There are different groups of PEM commands; some of them deal with the control flow of an experiment while others invoke various functions in the PMC controllers located in each PNODE.

4.2.1. General Experiment Manager Commands

All commands in the general PEM group are not specifically associated with any of the PMC controllers but are either executed in the PWS locally or invoke miscellaneous PMC functions.

The PEM supports experiment script string variables with arbitrary names. Due to the fact that the PEM experiment script interpreter is an APC, it consequently supports its own Paradise environment. A workload inherits the environment of the script interpreter. Environment variables are distinguished from other variables by having '%' as the first character in a name. Despite the fact that the script variables are kept in interpreter working memory as strings, arithmetic operators such as: '=', '+', '-', '*' and '/' can be used. If operands are integers in ASCII format, then the result will also be an integer in ASCII format. Otherwise the PEM interpreter will stop and an error will be displayed. The PEM script interpreter also supports a standard set of relational operators such as: '>', '>=', '<=', '<', '==' and '!='. Variables are allocated the first time they appear on the left side of an expression.

The experiment script interpreter supports the notion of a device. A **device** is a Cronus object which is used to transfer data from the interpreter to different tools on a dynamic basis (i.e. for the dynamic graphs and other mostly graphical representations.) Devices are referenced via device script variables which have names that always start with '$'.

4.2.1.1. Experiment_start() command

```
experiment_start(exp_id, workload, fi_prof, data)
```

The experiment_start command begins the execution of an experiment. It initializes all the involved PMCs and transfers the executables specified in the *workload* file to the appropriate PNODEs. It also transfers all the appropriate monitor fault profiles from the PWS to the target PNODEs. In case of an execution error, PEM will stop the execution of an experiment script and return to the menu mode.[21] If the integration platform isn't aware of the experiment id used, it will

[21] This command will also perform timing synchronization if the host OS does not export a synchronized clock to the local network.

create a new entry in the platform's list of the experiments. Data is a name of a file that holds the input data for the experiment. This file will be transferred to the PNODE where the SAPC is to be executed.

4.2.1.2. Experiment_end() Command

```
experiment_end(exp_id)
```

This command terminates an experiment. During this operation it will initialize all PMCs that were involved in the experiment. It will also reboot PNODEs if necessary (i.e. if they were left in a corrupted system state after the experiment termination). After this housekeeping is completed, PEM verifies that all monitored data is collected and stored into the integration platform identified as the experiment data for that particular experiment id.

4.2.1.3. Wait() command

```
wait(time, event)
```

The PEM experiment script interpreter waits for the expiration of the *time* interval or the arrival of the *event* notification, whichever comes first. Using P-NOTME for either the time or event variable forces the interpreter to wait for another alternative.

4.2.1.4. For() - Endfor() command pair

```
for(prologue; condition; adjustment)
.........
.........
     .
     .
.........
endfor()
```

The for-endfor command pair delimits a program loop in a Paradise script. *Prologue*, *condition* and *adjustment* are regular expressions specifying the start and end conditions as well as the adjustment at the end of each iteration. This syntax is similar to the for command in the C language.

4.2.1.5. If() - Elseif() - Endif() commands

```
if(condition)
.............
.............
.............
elseif()
.............
.............
.............
endif()
```

These commands have the same functionality as if-else commands in C. Elseif can be omitted if not needed.

4.2.1.6. Time() command

```
time(node,rtime,ltime);
```

Returns the local time (*rtime*) of a remote *node* and the local time (*ltime*) of the PWS. This command assumes that, in order to obtain time from another PNODE, two messages (to/from)have to be sent. It also assumes that the propagation time for both messages are the same and, therefore, that the remote PNODE reads its time in the middle of the period delimited by sending the request and receiving the answer. During the execution of the time command, the whole process is repeated three times and the mean value is used to produce the final result. The difference between rtime and ltime can be used to modify all data on timing from a remote node.[22]

4.2.1.7. Device() command

```
device(&mydevice)
```

Create a new device and make the variable &mydevice a pointer to a new device.

4.2.1.8. Get_word() command

```
get_word(dev, word)
```

Obtains the next word from the device *dev* and puts it in the variable *word*. A word is defined as a part of a bytestream delimited by one or more blanks at the beginning and at the end.

[22]If global time is not exported by the host OS , this command can be used during lengthy experiments to keep the local clocks synchronized. For an example, see section 6.2.3.

4.2.1.9. Put_word() command

```
Put_word(dev, word)
```

Takes the next word from the variable *word* and puts it on device *dev*.

4.2.2. Workload Manager Commands

The Workload Manager takes care of the creation and termination of a workload. A workload is defined by the executable of its SAPC, its input parameters, and a Paradise environment string.

4.2.2.1. Set_env() - Get_env() commands

```
set_env(var, val)

get_env(var, val)
```

Sets the content of an environment variable *var* to *val*, or gets the content of that variable and stores it into *val*. SAPC inherits the environment.

4.2.2.2. Workload_start() command

```
workload_start(papc, argc, argv, machine)
```

Start a SAPC with argc/argv as the arguments in the root node of the virtual machine defined by *machine* file. It is assumed that, in a typical experiment, the User will run a workload in only a subset of the available PNODES. This subset is described by a list of PNODE names constituting the Paradise virtual machine for the particular experiment. Upon the start of a workload, nodes from the list form a virtual tree, thus allowing for workload commands such as: suspend, resume or kill to take time proportional to log(n) where n represents the number of nodes in the virtual machine.

4.2.2.3. Workload_suspend() - command

```
workload_suspend()
```

Finds all the PNODES that constitute a Paradise virtual machine for a specific workload and suspends all the APCs that are associated with that workload.

4.2.2.4. Workload_resume() - command

```
workload_resume()
```

Finds all the PNODES that constitute a Paradise virtual machine for a specific workload and resumes all the APCs that are associated with that workload.

4.2.2.5. Workload_kill() - command

```
workload_kill()
```

Finds all the PNODES that constitute a Paradise virtual machine for a specific workload and kills all the APCs that are associated with that workload.

4.2.3. Monitoring Manager Commands

The Monitoring Manager deals with the monitor profiles. Each APC reads its own monitor profile at the time of initialization. After that, characteristics of the monitoring system can be changed dynamically from the experiment script. This is done by the change_mon_profile() command.

4.2.3.1. Change_mon_profile() command

```
change_mon_profile(old_profile, new_profile)
```

Both profiles are analyzed and changes are sent to the proper PNODES.

4.2.4. Fault Injection Manager Commands

The FI Manager deals with the fault profiles. Each APC reads its own fault profile at the time of initialization. After that, characteristics of the fault injection systems can be changed dynamically from the experiment script. This is done by the change_fi_profile() command.

4.2.4.1. Change_fi_profile() command

```
change_fi_profile(old_profile, new_profile)
```

Both profiles are analyzed and changes are sent to the proper PNODES.

4.2.5. Data Collection Manager Commands

Data collection mechanisms in Paradise have only one very important task, i.e. to transport event records from LRFs to PWS where they can be merged and stored in the integration platform. For reasons of performance efficiency, underlying data collection mechanisms are highly dependent on the Host OS and the hardware types.

4.2.5.1. Collect() command

```
collect(node, apc)
```

Collects monitoring data from the PNODE *node*. If *apc* is nonempty, the string data will be collected for the APC with the name equal to this string. Otherwise, data for all the APCs in the *node* will be collected and stored in PWS. If *node* is negative, data for all the PNODES in the virtual

machine will be collected[23].

[23]If the collect() command is executed while the workload is still running, new versions of LRFs will be generated to allow for undisturbed monitoring.

5. Paradise Tools

On top of the mechanisms described in the previous text, Paradise supports a set of policies and a related set of automated tools to implement these policies. In order to have a better understanding of how various Paradise tools are used, let us first develop an overview of the control and data flows for the Paradise/Cronus experiment preparation, execution, and data analysis phases. These three phases are depicted in Figure 5-1.

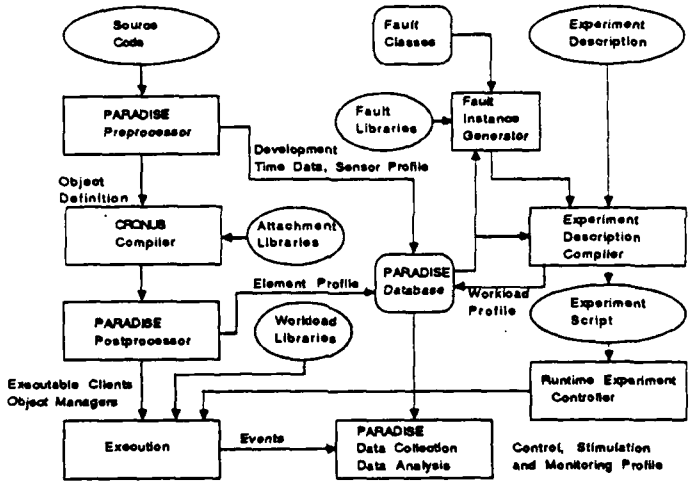

Figure 5-1: Paradise Control and Data Flow

There are three major User inputs to the experimentation process:

- C or Cronus source codes
- Fault classes
- Experiment description

In Paradise, source code represents definitions of either application clients (SAPC, APCs) written in the C language, or the C code used to describe operations of object managers (APMs) prepared in the Cronus standard form. Each source code entity is processed first by the Paradise preprocessor (PPROC) which extracts the *development time information* about the particular APC or APM specified by that source. The sensor profile information is part of this development time information. This data is placed in the Integration Platform for future reference or use by other tools. Using this information, the User instruments the code by selecting the sensors to be installed by means of the Paradise graphical interface PARASCOPE. At this point PPROC instruments the source code by inserting the selected Paradise sensors into it, and generates a Sensor Profile file. These are referred to as *.sp* files.

After the work of PPROC is done, the resulting output is processed by the C compiler. Here the Cronus and Paradise Attachment libraries are linked in.

The object file thus generated is then passed through the Paradise Postprocessor (POPROC) which extracts the element profile information, and generates an Element Profile file and saves the required access information in the Integration Platform. These are referred to as *.ep* files.

Each of the developed APCs and the associated *.sp* and *.ep* are saved in the workload library for use at a later time.[24]

There are two sources of fault classes for an experiment. Either the User defines the experiment specific fault classes or reuses some from the fault library. It is again our experience that the number of practical fault classes (for a particular architecture and workload) is not extensive. Therefore, after the system is used over a period of time, most of the usually required fault classes will have been installed in the fault library.

The Fault Instance Generator (FIG) takes the description of a fault class and produces a number of fault instances to be applied to the particular workload. In order to do this, the FIG has to have knowledge about the element profiles of all the APCs that constitute a workload. FIG also has to have knowledge about their sensor profiles due to the fact that, in Paradise, faults are triggered by the sensors.

The generated fault instances are placed into Fault Lists according to the APMs and APCs affected. The information from the respective Sensor Profile and FaultList is used to generate a Monitor Fault Profile (MFP) for each APM/APC to be observed and/or fault injected in the workload.

Once a workload is generated and all fault instances needed for fault injection are produced, one has to describe the desired experiments to the Paradise system so that the experimentation process can be conducted automatically. An Experiment Definition is compiled by the Experiment Definition Compiler (EDC) to produce an Experiment Script. This Experiment Script is then interpreted by the Runtime Experiment Controller (REC) in order to execute the workload.

The REC issues commands to the PMC in each PNode in order to:

- Start the workload and synchronize its APCs
- Monitor its execution
- Fault inject
- Collect and save and/or present monitoring data
- Terminate the workload

Results extracted from the data collected during the run time of a workload are stored in a relational database which serves as an Integration Platform (IP), as depicted in Figure 5-2. In addition to an available set of preprogrammed interfaces and various development tools, the IP is

[24]Also it is our experience that many of the modules, especially those dealing with i/o or custom measurements, are often reused in practice.

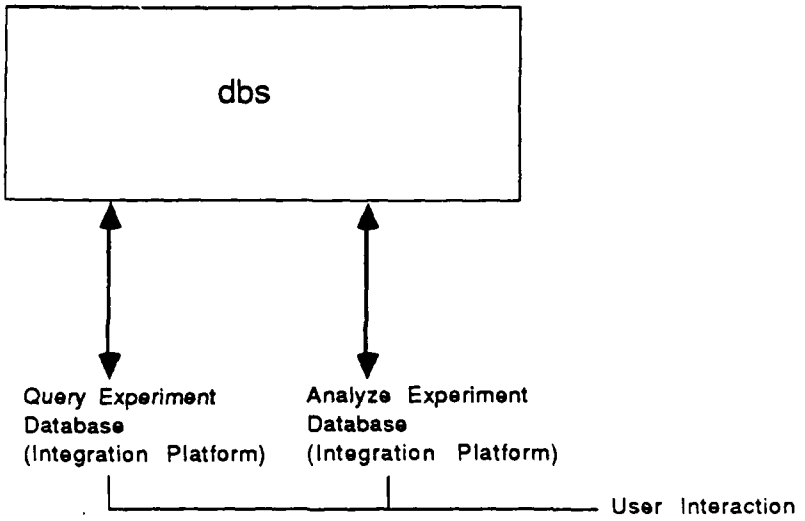

Figure 5-2: Paradise Integration Platform

supplied with a general query interface so Users can also build their own customized tools.

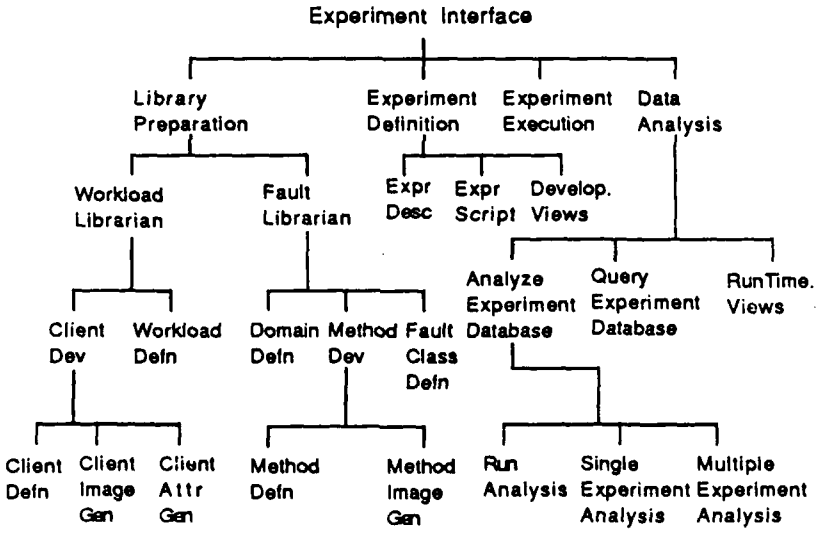

Figure 5-3: Paradise Menu Structure

Due to the fact that multiple passes through the control flow of the experimentation process are possible, Paradise exports a set of menus at the User interface level. The structure of the Paradise menus is shown in Figure 5-3. This arrangement permits the User to conveniently enter or re-enter any of the various development steps simply by traversing the menu tree. The use of menus also enforces Paradise administrative procedures in the experimental environment.

A typical pass through the Paradise experiment control flow is shown in Figure 5-4. The User first develops all the necessary clients. Then the workload is defined and its description stored in the workload library. After this is done, first a fault free experiment is performed and the behavior of

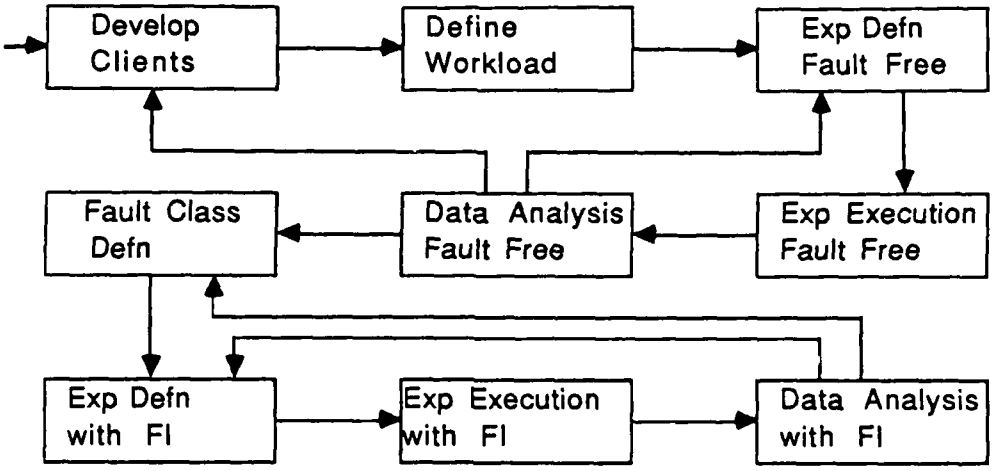

Figure 5-4: Flow of an Experiment

the workload is determined and analyzed. This part of the experiment is important due to the fact that in some cases fault injection will result in reduced system performance rather than an outright corrupted functionality.[25] Typically, after the fault free behavior is carefully analyzed, a new set of experiments involving fault injection is prepared and executed. It is important to note that the User can return from almost any point in the experiment control flow to another point in order to refine or

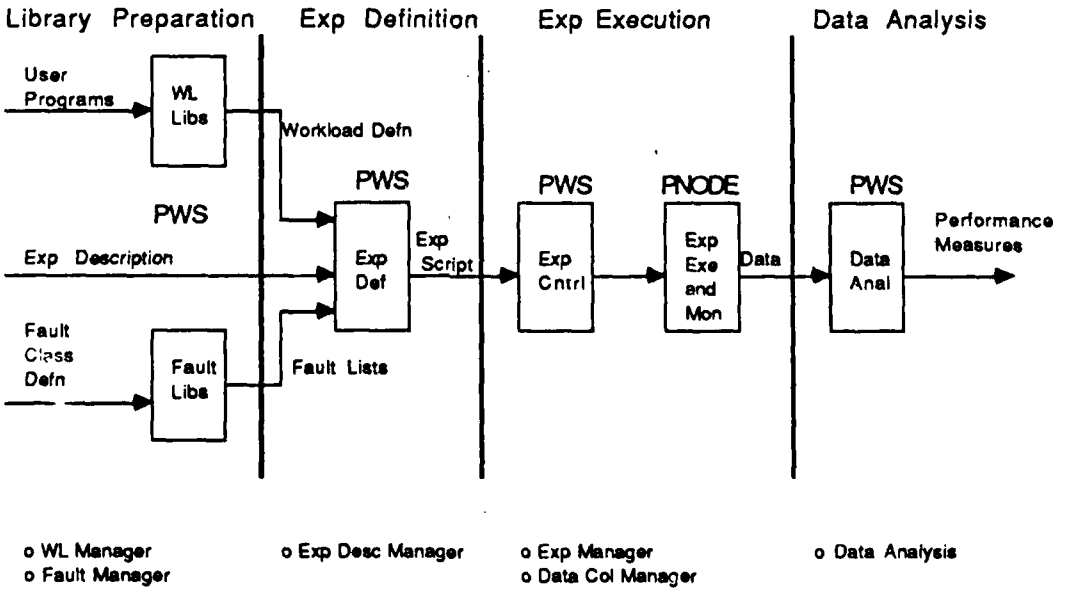

Figure 5-5: Typical Monitoring and Fault Injection Experiment

redefine the experiment.

The control and data flow for a typical monitoring and fault injection experiment is depicted in

[25]The fault free experiment provides the reference basis to detect reduced performance.

Figure 5-5. All four major phases of an experiment are described in this diagram. These are:

- Library preparation
- Experiment definition
- Experiment execution
- Data analysis

Each of these phases and the associated tools are now discussed.

5.1. Library Preparation

As shown in Figure 5-1, there are two important libraries involved in the Paradise experimentation process:

- Workload library
- Fault library

In order to provide for the management of these libraries, the Paradise system includes two Librarians, the Workload Librarian and the Fault Librarian.

5.1.1. Workload Librarian

The Workload Librarian provides tools for the management of the workload library. It supports three main activities:

- APC development
- APM development
- Workload definition development

In addition to an enhanced editor, the Workload Librarian consists of the following tools:

- PPROC
- PARAMAP
- WLGEN
- POPROC

Development of a client begins with the editing of the APC or APM source code. After editing, the User invokes the PPROC tool in order to instrument the particular object described by the edited source code and to extract the sensor profile information.

5.1.1.1. Paradise Preprocessor - PPROC

The Paradise preprocessor reads in a specification of an APC or an APM. It has built in knowledge about the syntax of the input. It also has some limited knowledge about the semantics of the input due to the fact that it recognizes some high level abstractions (i.e. data object operations, etc.). The PPROC can be customized in order to accommodate for various semantic entities dependent upon the specific engineering environment in which it is used. Based on its input, the

PPROC will produce a **.pif* file which contains an internal description of a given APC/APM in Paradise intermediate form. The pointers to these files are stored in the IP and are used by other Paradise tools to increase productivity by avoiding repetitive parsing of the same source.

5.1.1.2. Paradise Roadmap - PARAMAP

The Paradise "roadmap", or PARAMAP, is a member of a set of graphical tools collectively called PARASCOPE. PARAMAP presents the User with a graphical view of the observable structure of an APC/APM. PARAMAP reads a *.pif* file, if it exists, and outputs the structure in the form of an oriented graph. If the *.pif* file doesn't exist for an APC/APM, PARAMAP will invoke PPROC and produce one before generating the graphical representation. The same will happen if an *.pif* file is out of date with respect to its source counterpart. These types of actions are part of the Paradise responsibility to maintain a consistent system.

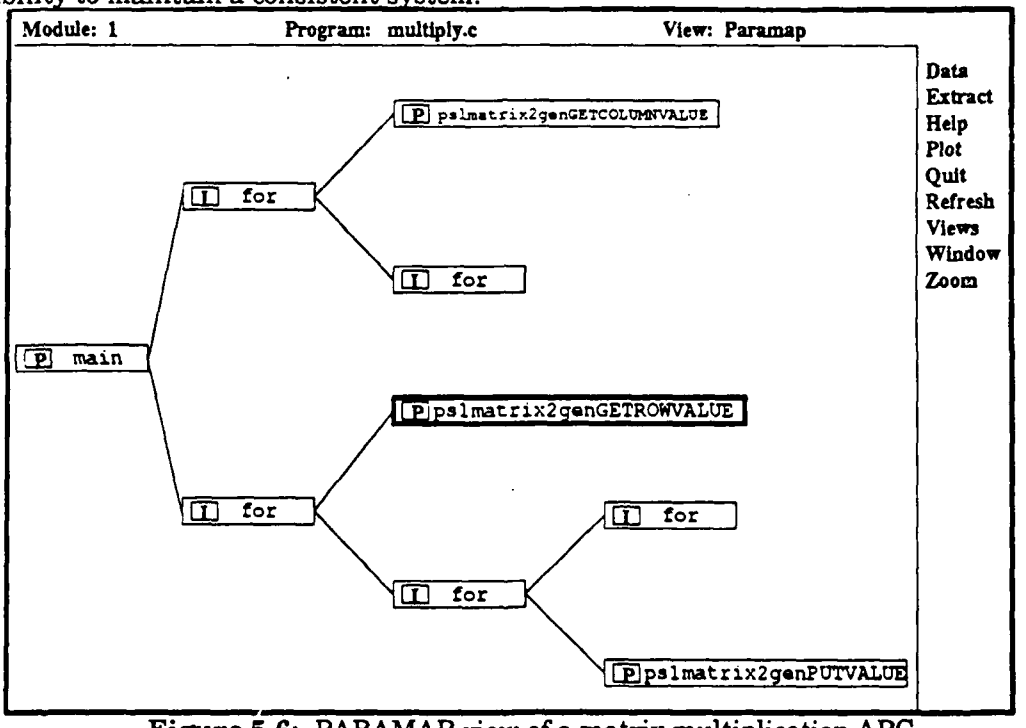

Figure 5-6: PARAMAP view of a matrix multiplication APC

The structure of such a PARAMAP graph, as shown in Figure 5-6, illustrates the nesting of the various syntactic program constructs that form an APC/APM such as for-loops, while-loops, functions, and so on. PARAMAP maintains a reference link with the original source code via the IP so that the User can backtrack and connect various graph nodes with their corresponding source representation, and vice versa. This is done by selecting either construct with a mouse pointer; this causes its counterpart to be highlighted in the other window.[26] There is virtually no limit on a number of text editor and PARAMAP windows that can be simultaneously displayed on a screen, so the User can navigate through several workload APCs/APMs at the same time.

---

[26]As an example, if the User "clicks" in a PARASCOPE graph node representing an data object operation call, the cursor in the text editor window will move to the beginning of the line which contains the source code for that call.

By using the available PARAMAP menus the User can manipulate the graphical representation in order to vary its expressiveness (i.e. the User can trim, zoom, etc). PARAMAP includes commands which can make an individual and/or a subset of existing APC/APM constructs observable. PARAMAP produces a sensor profile which is a list of sensors needed to make observable source level constructs visible.

Upon demand, if any of input specifications (ie. an input source[27] or a *.pif* file) has been changed, PARAMAP produces an output source with the system and User sensors inserted.

In the next step PARAMAP produces a sensor profile for the APC or APM it is processing. A sensor profile is a list of descriptions for each of the sensors manually or automatically inserted into the code. Sensors inserted into the code at various locations in its control flow make these points (i.e. events associated with an execution reaching these points) observable. There is an *id* and a *type* description associated with each sensor entry in the sensor profile. PARAMAP saves the pointer to the sensor profile in the IP so that it can be retrieved and altered if the related APC/APM is revisited at a later time.

PARAMAP outputs are **.sp* and **.c* or **.typedef* file types depending on whether the input file was an APC (**.pc*) or an APM (**.ptypedef*) description, respectively.

5.1.1.3. Paradise Workload Generator - WLGEN

For each workload the User defines a **.wld* file which contains an SAPC and a list of the objects which constitute the workload. The SAPC and each module definition consists of a module name and a list of the PNodes in which this module will be used during the workload execution. Given a module name, all files associated with this name (input, output and intermediates) will have the same name but with different extensions.[28] WLGEN uses a **.wld* file to produce all the objects needed to run the workload, and to transfer and maintain them in all the nodes where they will be needed. This means that at times, especially in a heterogeneous environment, WLGEN will have to start the parallel compilation of sources in different nodes or it will have to distribute copies of an object. In order to produce the objects (**.o* files) for all the APCs/APMs that constitute a workload, WLGEN uses the standard C compilers available in each PNode.

After all the objects have been created, WLGEN will run the Paradise POPROC on them.

---

[27] A **.ptypedef* file in a case of an APM or a **.pc* file in a case of an APC file

[28] For the APC called *"grinder"* the source file will be *grinder.pc*, the Paradise intermediate form will be *grinder.pif*, the sensor profile will be *grinder.sp* and instrumented code will be *grinder.c*.

5.1.1.4. Paradise Postprocessor - POPROC

POPROC is a tool that accepts Paradise APC/APM workload objects and links them into executables. After this, it extracts the element profiles. POPROC deposits these profiles in **.ep* files. An element profile is a list of local data and code elements that constitute an executable. Each of the elements are characterized by its source level name, its virtual address in the executable address space, and its size in bytes. Element profiles are used to generate fault instances in conjunction with fault classes.

5.1.1.5. The Paradise Workload Library

The workload library is organized in a hierarchical manner. Each of the sources, objects, and various profiles or workload description files are stored in separate sublibraries.

Data object development in Cronus yields an executable for the object manager and a library of operation functions to be performed on an object. This library is also managed by the Workload Librarian so that all the locations where an APC references an object can be properly instrumented. The Workload Librarian also maintains a library of workload descriptions (i.e **.wld* files described in Section 5.1.1.3). The Paradise Workload Librarian may be used by the Paradise User to insert/delete various workload components, but its main function is to support the operation of various tools accessing or generating workload components in any of the various development phases.

5.1.2. Fault Librarian

The Fault Librarian maintains three sublibraries:

- fault class library
- fault method library
- fault domain library.

The fault class library holds a set of fault class descriptions, while the fault domain library holds the lists of domains that form a general set of system and User-specified domains. Methods are executable files and are maintained in a similar manner to workload executables.

Note: Please keep in mind that the fault domains, in general, are workload dependent since they depend upon specific PNodes, APMs, APCs, elements, and so on. This information, as described earlier, is maintained by the Workload Librarian. The Fault Librarian has access to these domains also.

There are other types of domains which are not workload specific per se, such as collections of special masks, and the list of possible operations by applying the masks (i.e. AND, OR, MOVE, etc.). The Fault Librarian is used to maintain such domains so that they are available to other tools and multiple Users. These are referred to as *system domains*. Also, the User may wish to define some special domain(s) for a workload. These are referred to as *User-specific domains*. The Fault Librarian supports these types of domains also.

5.1.2.1. Paradise Fault Class Generator - FCGEN

The User generates new fault classes by using the Paradise fault class generator tool FCGEN. FCGEN is based on an advanced editor which knows about the format of the Fault Class Definition file (*.fcd). This file consists of a list of tuples in the form (domain, method) where the second element is an executable file name. A *method* can be any Cronus executable that accepts the list of domain files and a name of an output file as its input parameters. The sole role of a method is to read the domains from a workload description named "workload" and form a new tuple which consists of domain members. When finished, the method appends its result to the content of the submitted output file.

Domains are lists of elements that represent categories in the workload description file. Typical examples are node names, names of APCs/APMs, element profiles or sensor profiles. Element and sensor profiles are further subdivided into categories such as: data, instructions, and the various sensor types. The set of domain types for a workload in the Paradise system is is rather fixed due to the fact that domains are directly related to the internal design of Paradise itself.

Paradise supplies the User with a library of functions to access these domains. Using these library functions, the User can easily prepare methods since the actual names and locations of the domain information is not needed to be known by the User. Since the User works within the context of a certain experiment and a certain workload, all domain information is implicitly named. The Paradise administrative operations are abstracted out. The User needs only to specify the type of domain information, and the library functions perform the actual extraction. Below is a representative list of these functions:

- Node list functions:
  - p_get_node(index,fp) - returns the node id for a node in the position pointed to by *index* in the node list of a *.*wld* file. *fp* is a pointer to an open *.*wld* file.
  - p_get_num_nodes(fp) - returns the number of nodes in the node list of a workload description pointed to by *fp*.
- APM list functions:
  - p_get_apm(index,fp) - returns the APM id for an APM in the position pointed to by *index* in the APM list of a *.*wld* file. *fp* is a pointer to an open *.*wld* file.
  - p_get_num_apms(fp) - returns the number of APMs in the APM list of a workload description pointed to by *fp*.
- APC list functions:
  - p_get_apc(index,fp) - returns an APC id for an APC in position pointed to by *index* in the APM list of a *.*wld* file. *fp* is a pointer to an open *.*wld* file.
  - p_get_num_apcs(fp) - returns the number of APCs in the APC list of a workload description pointed to by *fp*.
- Sensor Profile functions:
  - p_get_sens(index,type,fp) - returns true or false depending if the sensor in position pointed to by *index* in a sensor profile is of the specified type. *fp* is a pointer to an open sensor profile file. *Type* is a pointer to a mask of type flags so the User can select to deal only with a certain subset of all possible sensor types (i.e. the User

can look into loop and procedure call sensors only, etc.)

- p_get_num_sens(fp) - return the number of sensors in the sensor profile pointed to by *fp*.

- Element Profile functions:
  - p_get_elem(index,type,fp) - returns the starting address and the size of an element pointed to by *index* in an element profile. *fp* is a pointer to an open element profile file. *Type* is a pointer to a mask of type flags so the User can select to deal only with a subset of all possible element types (i.e. the User can look into data or code elements only).
  - p_get_num_elem(fp) - returns the number of elements in an element profile pointed to by *fp*.

There is no limitation on the number or type of methods that can be created due to the fact that they are individual programs written by Paradise Users. As experience has shown, a basic set of methods will evolve as more and more experiments are developed on the system. These can be made generally accessible in the form of a library of standard methods.

More so, the User can use already existing methods to form more complex ones. As an example, Paradise will supply methods which select an element from a domain by choosing different random distribution functions. One can easily envision a User combining several such methods to obtain more complex random tuples. The default Paradise method is called *Interactive*. This methods prompts the User to make a choice from a displayed domain list.

5.1.2.2. Paradise Fault List Generator - FLGEN

The role of the Paradise Fault List Generator is to take a fault class description file (*.fcd) and a workload description file (*.*wld*) and produce a Fault List (*.*fl*) file. The User can optionally name the *.*fl* file or the workload id prefix is used as a default (the extension .*fl* remains the same).

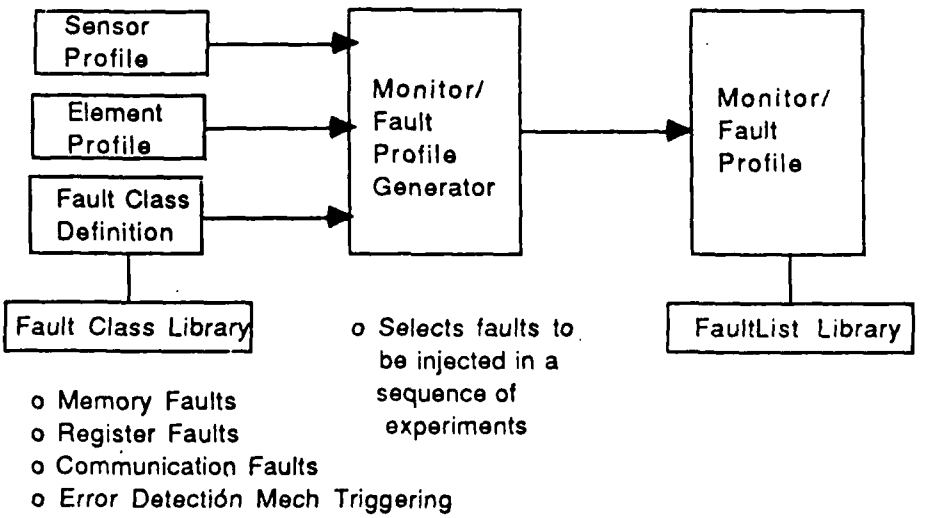

Figure 5-7: Paradise Fault Librarian

The Fault List is a list of fault instances which are used to construct a Monitor Fault Profile (MFP). The process of generating fault lists is depicted in Figure 5-7. They are generated by the Paradise Fault list generator FLGEN.

FLGEN reads a fault class description and uses the workload description in order to fork off the specified methods and produce a list of fault instances.

As part of the Paradise design philosophy, a Fault Free experiment (i.e. monitoring only) is a special case which utilizes only *null* fault lists.

5.1.2.3. The Paradise Fault Library

The Fault Library is organized in a hierarchical manner. Each of the developed fault class definitions, domains, and methods are stored in separate sublibraries.

The Paradise Fault Librarian may be used by the Paradise User to insert/delete fault class definitions and various User defined domains and methods, but its main function is to support the operation of the various tools involved in the production and administration of fault lists.

5.2. Experiment Definition

The User provides Paradise with a high level Experiment Definition which describes the desired control flow of an experiment. The commands discussed in Section 4 are used to compose the definition. As part of this description, the User must also specify the fault class definitions and the workload definitions which have been prepared.

5.2.1. Experiment Definition Compiler - EDC

The Experiment Definition Compiler (EDC) takes these three inputs (as depicted in Figure 5-8) and produces an Experiment Script. The script consists of appropriate PMC, Cronus, and system commands required to set up and execute the experiment. During the process of compilation, the EDC references the relevant workload data in the Integration Platform in order to obtain the monitor and fault profiles and then tailor them according to the needs of the experiment. The workload definition provides the necessary information to retrieve these items. Please note that the monitor/fault profile for each APC/APM includes all of the module sensors, but they are all initially disabled. In this phase, however, the EDC sets up the initial values for the sensor control blocks and also creates sensor enable commands in the script locations where a change of the monitoring characteristics is needed. The same is true for the fault profile which, at the beginning, contains the initial values of the fault control blocks but are not yet linked to triggering sensors. The generated script contains the required commands to create these sensor/fault links.

In the above scenario, the basic MFP for each APC/APM is transferred to the PNode, and the sensor and fault alterations are performed remotely. As a contrast, the example discussed in the Appendix discusses how this is done interactively by the User at the PWS. In this case the MFP is

fully implemented, so no script commands are required.

The creation of an Experiment Script is facilitated with a set of available graphical tools which can be used to pinpoint various parts of the experiment's control and data flow for the purpose of monitoring and/or fault injection.

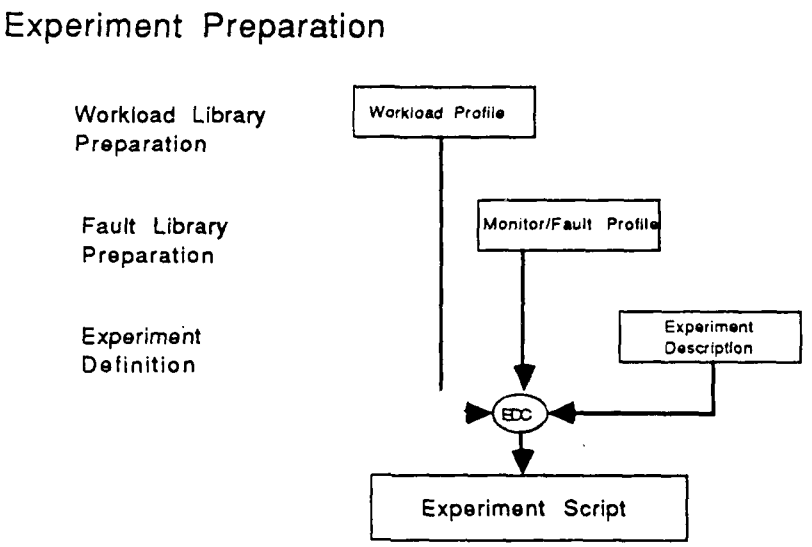

Figure 5-8: Data Involved in Experiment Preparation

5.2.2. Experiment I/O Devices

Paradise will support not only batch but also interactive modes of experimentation. While one can imagine that the User input to a running Paradise Experiment Script will be almost always simple in nature, it will generally be desired to accept, process and present the output data a variety of ways. For this reason Paradise supports the concept of I/O *devices*. A Paradise *device* is a Cronus object which implements five operations:

- open
- read
- write
- control
- close

While the Users can supply their own devices, there will be a set of standard devices available.

For example, one of the standard devices will be a *console* implemented via an X window. There will also be a set of graphical devices presenting the views discussed in Section 5.4 below. A *device* accepts a fixed format of put data in order to produce specified outputs. The Experiment Script interpreter can also read and interpret certain experiment data in order to modify its control flow on the basis of some previous results. In this way it is similar to the Unix Shell since it can perform I/O

and maintain state. A set of devices can be supplied to handle interfacing to the IP and the experiment data files produced by the monitoring facility. To invoke a device, the User specifies a device variable to an *open* call which in turn returns a pointer to a device control block. All successive I/O operations are then performed using that pointer.

5.2.3. Experiment Definition Variables

As mentioned the Experiment Definition is a high level script similar to a shell script in UNIX. This script is interpreted during the experiment execution time. The User can define a set of script variables with arbitrary names. All the variables are strings and contain the information in string form. The User is supplied with a set of three address operators which perform various arithmetic and logical operations on the script variables.

Due to its nature, the Experiment Script in Paradise is used to define the control flow of an experiment, while the actual data processing is performed by the C code modules which implement the device commands.

5.3. Experiment Execution

The execution of an experiment is directed by the interpretation of the experiment script by an experiment controller.

5.3.1. Runtime Experiment Controller - REC

The Runtime Experiment Controller (REC) sends various commands to specific PNODEs in the system. The respective PMCs in these PNODEs receive, interpret and execute these commands. As

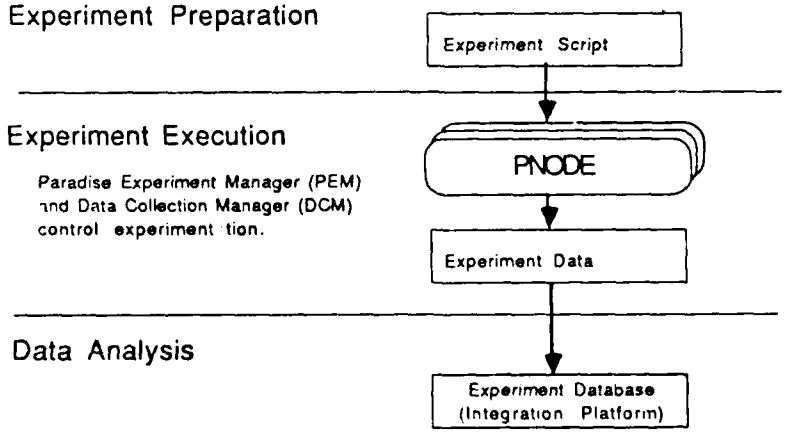

Figure 5-9: Experiment Data Collection

the consequence of these commands, first, the workload SAPC is created and then the actual

workload APCs are created by the SAPC in turn. The SAPC terminates itself at this point. While executing, the APCs interact via data objects. Each time an APC encounters a sensor in its control flow, a filtering function is performed and the sensor is fired or not depending on the current content of its sensor control block. Each time a sensor is fired, a notification flag is tested, and if true, an event record is generated. The event records are collated by the data collection mechanisms and ultimately stored in the IP so that data analysis can be performed. The complete process is depicted in Figure 5-9.

5.4. Data Presentation and Analysis

The function of the data analysis part of the Paradise system is to facilitate the User's understanding of the behavior of the system under test.

Data analysis tools in Paradise can be divided into three groups:

- Preprogrammed experiment database analysis
- Random queries into experiment database
- Specific run time views

Tools from the first group offer a predefined functionality. They access specific experiment data, interpret it, and present the results using one of the standard graphical displays. Here are some examples.

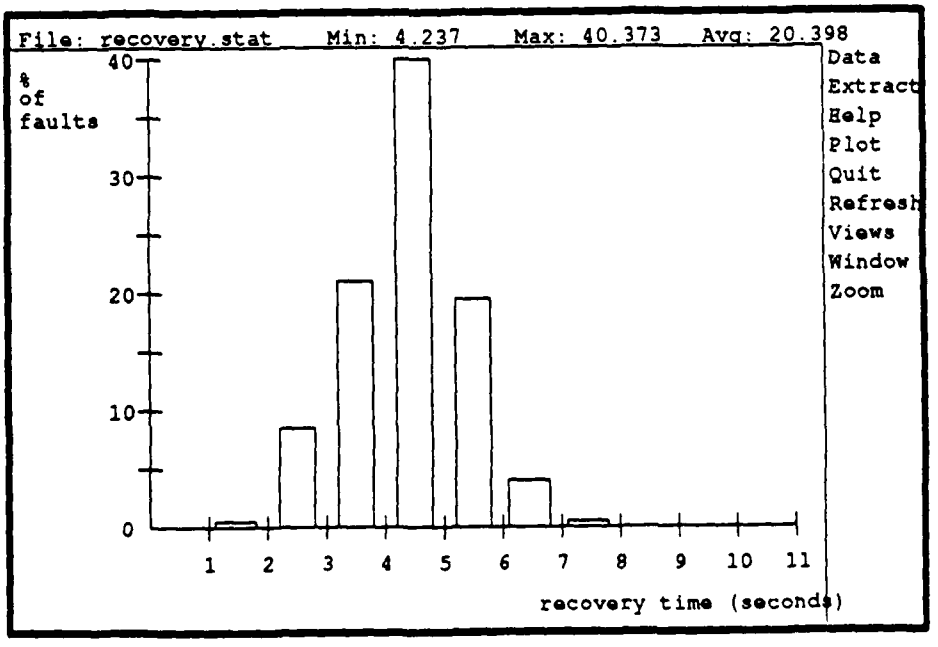

Figure 5-10: Histogram Presentation of the Experiment Data

Figure 5-10 shows the output of a tool which presents data on the recovery statistics from a set of fault injection experiments in the form of a histogram.

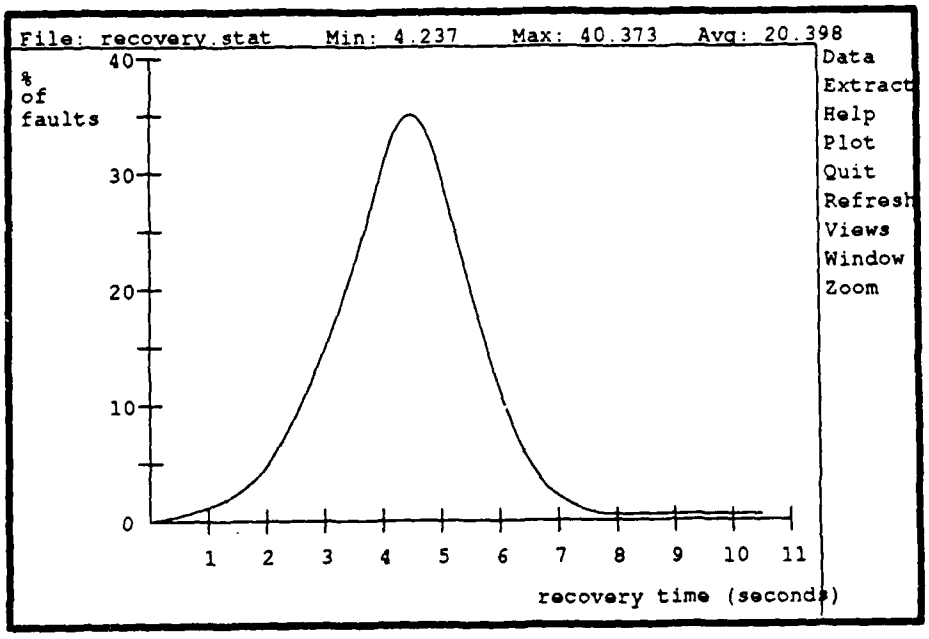

Figure 5-11: Graph Presentation of the Experiment Data

Figure 5-11 shows the same data plotted as a graph. Other standard outputs are: pies, boxes, bar plots, and scatter graphs. The User has the means to change shading and type of lines and symbols. Linear and log scales on graph axis are interchangeable as well as the scales and units.

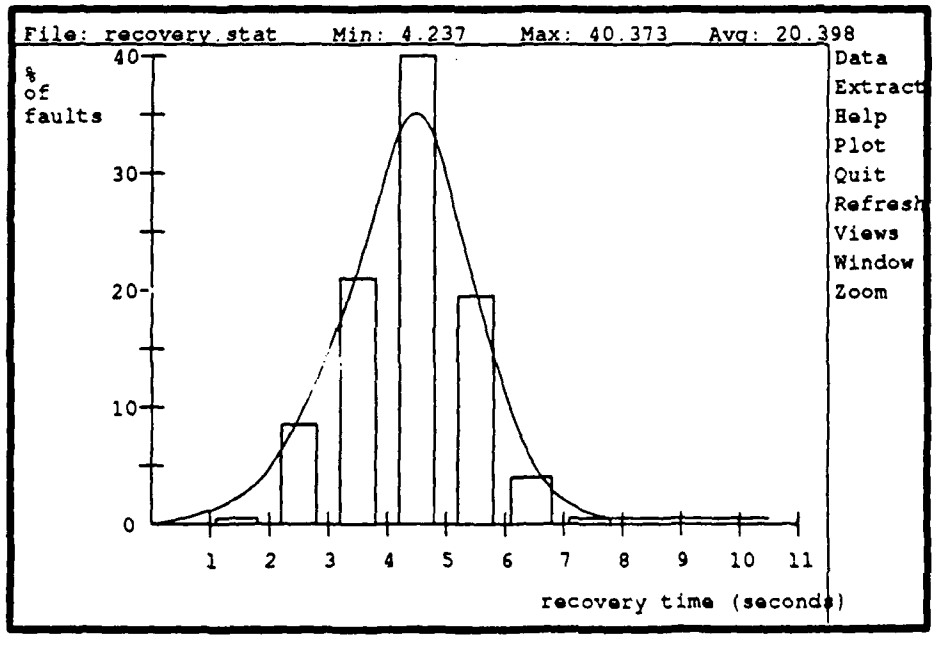

Figure 5-12: Combined Presentation of the Multiple Experiment Data

The User can choose different combinations of presentations as depicted in Figure 5-12. Paradise tools not only provide outputs for the presentation of arithmetic relations but other types of relations

may be presented also.

An example is depicted in figure 5-13. It presents the APC - data object relation for the test application presented in the feasibility study. It shows APCs from three PNODEs using data from an object in PNODE 4. This kind of preprogrammed presentation is called a *dependency graph*. Arrows represent dependencies and boxes represent objects which are related by these dependencies. The User can change the style of objects from boxes to circles. The arrows can be single or double directional, which represent a mutual dependency. In addition to the tools which provide a graphical

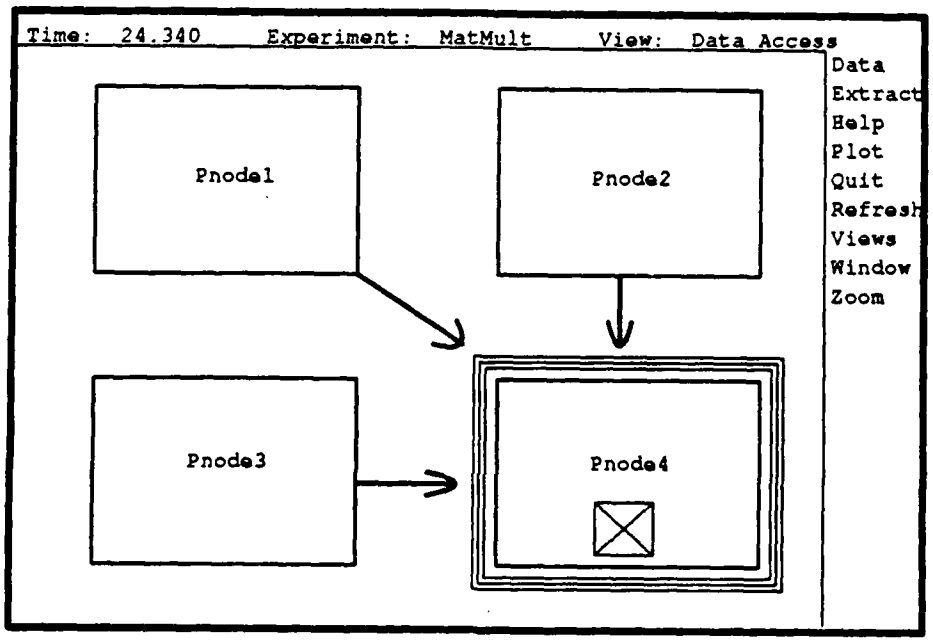

Figure 5-13: Data Access View

output, there are also tools that can produce alphanumeric lists and tables. The common characteristic of all these tools is that the User specifies the data to be extracted from the database and each tool then displays this data in one of several preprogrammed formats.

Due to the fact that all of these tools share the same data set via the Integration Platform, one can easily relate graphical and alphanumeric representations of the same data element by using a mouse and selecting the element in question in one representation. If a tool is interconnected with others then, when an element is selected, it will communicate its identification to the IP so that the other tools currently presenting this element will highlight it. As an example, assume a case where the User has two views of a workload on the screen: one Data Access view as depicted in Figure 5-13 and another alphanumeric view showing the source code for one of the APCs involved. If these views are connected, selecting the arrow in the former case would result in moving the cursor to the exact location of an object operation call in the latter one.

The second group of tools are custom designed tools which Users can build for themselves. Paradise provides a standard interface to the Integration Platform to facilitate the implementation

of such tools. This interface will be in a form of a relational query language. Paradise will also maintain these tools in a library so they can be shared among other Users.

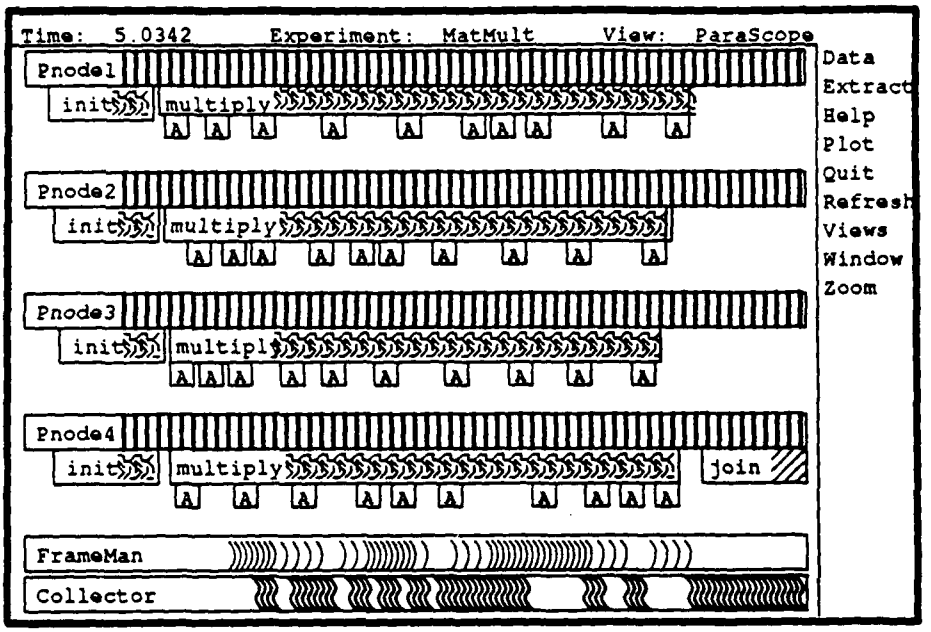

Figure 5-14: PARASCOPE Dynamic View

Up to now the tools described gain their advantage from the fact that all the data and the relations they require are stored and available in the IP. However, there are situations where the User requires a dynamic real time, instead of post mortem, view of specific relations in the experiment. For example, let us assume that the User wishes to monitor a part of an application which is running continuously (i.e. 24 hours a day 356 days a year), and that all that is needed is to obtain an occasional snapshot of the execution. In order to achieve such a mode of operation with a tolerable level of overhead, tools should exist to allow for dynamic retrieval of event records from the data collection mechanism. This data then, together with the relations needed from the IP, can be directly forwarded to one of the preprogrammed presentation tools to achieve an animated dynamic view as the one shown in Figure 5-14. The set of Paradise tools designed to provide the data transfer between the collection mechanisms at the run time and the preprogrammed presentation views is collectively called *PARASCOPE*.

6. Paradise Feasibility Study

6.1. General

A rudimentary environment was implemented in order to simulate certain parts of Paradise. The hardware components of the environment consisted of four Micro VAXs, all running the Cronus kernel and UNIX, and connected via an Ethernet. The four nodes were named pnode1, pnode2, pnode3, and pnode4.

This chapter studies the implementation of a monitoring attachment (MA), a data collection attachment (DCA), a workload manager (WM), and also the issue of global time. Using these experimental and rudimentary implementations, an example APC was developed and instrumented.

6.2. Implementation Experiments

6.2.1. APC Attachment Implementations

This section describes the implementation of the MA and DCA of a Paradise APC. Several methods for collecting and storing sensor data were implemented and then observed by experimentation. Timing measurement data collected from those experiments are presented.

Experimental sensors were placed in a Cronus client program to obtain performance estimates of different sensor data logging techniques. The sensor was placed inside a loop that iterated a number of times specified by the User. The sensor had the following format:

```
pslcollgenSENSOR (Host, Type, StatID, Data)
HOSTNUM     Host;  /* Host sensor manager is on   */
long        Type;  /* Type of sensor              */
long      StatID;  /* Static ID of sensor         */
char      **Data;  /* Optional sensor data        */
```

Additional information that is logged when the sensor is processed include a timestamp, and the UID of the principle owning the client process invoking the sensor.

The different methods of sensor data logging are described below:

Object	Each sensor data entry is maintained as a Cronus object with associated operations.
File	The Cronus object manager is used to provide operations that log the sensor data into a file, not an object.
Independent	Cronus object manager is not used and data is directly written to a file.
Buffered	Sensors are locally buffered before using either the *File* or *Independent* method to store sensor data.

6.2.1.1. Execution Time Measurement Results

The following results were obtained by measuring the absolute execution time for the benchmark client program. Time was measured by inserting the *gettimeofday* command at the beginning and end of the client program. The program was executed on a Micro VAX workstation with a single active User and very low load. All time measurements are in seconds.

For reference, the execution time for the client program without any sensors was approximately 1.19 seconds for 100,000 loops, and approximately 0.01 seconds for 1,000 loops[29]. Without sensors, the program is just an empty loop with no execution parts.

Object

A Cronus manager acts as MA and DCA in the *Object* method. Each sensor in the APC is actually an operation on this manager. Thus, as expected, execution speed is very slow as shown in figure 6-1. The type definition files for this Cronus manager is included in the appendix. With the *Object*

# loops	avg. time	std. dev.	max.	min.	# runs
10	2.29	0.37	3.13	1.89	9
100	21.99	2.71	27.45	20.05	9
1000	214.43	3.40	219.72	210.00	6

Figure 6-1: MA/DCA as Object Manager

method, a file named *objectdb.327*[30] is maintained to keep object information. It was observed that each object entry occupies approximately 500 bytes in the *objectdb.327* file.

File

The *File* method uses a Cronus object manager to write sensor data into a data file. Thus, as in the *Object* method above, each sensor is a manager operation, and each operation results in a write to the data file. The object maintained by the manager contains information relating to the data file.

# loops	avg. time	std. dev.	max.	min.	# runs
100	21.46	1.34	23.55	19.48	9

Figure 6-2: MA/DCA as File Write Manager

Due to the similarity of the measurements of the *File* method and the *Object* method, no further runs were performed. This similarity shows possible large overhead in the object manager itself (or in file writing). To investigate these issues, the next two experiments (*Independent* and *Miscellaneous*)

[29]For 100 loops, it was below 0.01 seconds and was immeasurable with available granularity of *gettimeofday*

[30]327 is the sensor object's unique number. A unique number assigned to each different object type.

were performed. The approximate number of bytes for each sensor entry in the data file was 90 bytes.

Independent

The following data was obtained by writing the sensor as a procedure call to a routine that writes the sensor data to a file. Thus, there is no object manager involved. Once again, the approximate number of bytes per sensor entry in the data file was 90 bytes. The numbers of Figure 6-3 show a rough estimate of pure file write time involved.

# loops	avg. time	std. dev.	max.	min.	# runs
100	8.14	1.48	10.81	6.80	9
1000	67.19	4.18	75.48	64.25	6

Figure 6-3: MA/DCA as Direct File Write

Miscellaneous

This section was included to try to obtain an estimate for the overhead added by using the object manager. Objects were not maintained, instead, the sensors here represent the invoking of an operation that does nothing. Thus, this represents the pure overhead of the object manager. The main reason that the numbers for Figure 6-4 and 6-3 do not add up to those of Figure 6-2 is because, for the *Miscellaneous* method, no data is involved. When operations are called with data, the *psl* routines of Cronus go through data checking and conversion into an allocated Octet Buffer. These checking, conversion, and allocation times are not included in the numbers of Figure 6-4.

# loops	avg. time	std. dev.	max.	min.	# runs
100	7.33	1.03	9.45	6.64	8

Figure 6-4: Manager Overhead

Buffered Sensors

In the *Buffered Sensors* implementation, sensors are cached locally and an operation is called only to empty a full cache (or to flush the cache at end of execution). A Cronus manager writes sensor data into a data file. Thus, the local caching is the MA and the Cronus manager is the DCA. All data of Figure 6-5 is for 100 loops and all times are in seconds[31]. The results shown for each entry represent an average value for 9 runs. The field # *Op. Calls* of Figure 6-5 show the number of operations invoked on the Cronus manager. It is the number of sensors divided by queue size. If the division is

[31] At time of writing, execution speed was significantly improved due to implementation modifications.

Cache size	Avg. Time	Std. Dev.	# Op. Calls
1	20.17	0.5005	100
5	5.49	0.2316	20
10	3.70	0.3517	10
20	2.66	0.0912	5
30	2.44	0.0718	4

Figure 6-5: MA/DCA as Buffered Sensors

not exact, as in the case of cache size 30, one more operation call is need to flush out the remaining sensors at the end of execution. Thus, for a cache size of 30, there are four operation calls. This is why there is a non dramatic speed increase when cache size is increased from 20 to 30 (because while cache size increased by 50%, the number of operations only reduced by 20%). The speed increase achieved with larger queue size levels out due to the fact that the execution time per operation increases with the larger queue size (more data passed).

6.2.2. Workload Controller

The following data is the initial results of a 'controller' that acts as a server to create, kill, and signal processes on Cronus nodes. It is a *very* simple implementation of the WC part of PMC. For this experiment, the basic WC, implemented as a Cronus manager, exists at each node and responds to the User operations on it. The WC has operations to create, kill, and signal processes on its node.

The data shown is for the average time taken to create a process on a node. Other than *fork* and *execl*-ing a process, the manager will keep an object database concerning each process. Each object is of the following structure:

```
UID         ProcID;     /* Cronus UID of process     */
DATE        Timestamp;  /* Time process was created */
int         PID;        /* The UNIX process ID       */
int         group;      /* Group membership          */
int         Type;       /* Type of creation          */
int         Status;     /* Status of execution       */
char        *ProcName;  /* Name of the process       */
```

The example process that was created on each node was a simple program that just has one *printf* statement. A User interface was implemented to trigger operations on the different managers and resided on host Pnode3. The results shown in Figure 6-6 are an average of 10 runs with all units in seconds.

Host Called	Avg. Time	Std. Dev.
pnode1	0.485	0.0251
pnode2	0.526	0.0564
pnode3	0.351	0.0099
pnode4	0.480	0.0189

Figure 6-6: WC Time Measurements

6.2.3. Miscellaneous Measurements

A time manager with a single *GETTIME* operation was set up on each Cronus host (pnode1, pnode2, pnode3, pnode4). These time managers do not actually maintain any objects, but instead, they simply return the local time. The purpose of this experiment was to try and determine the time differences between each host and a standard host. Pnode3 was chosen as the standard host, and thus, a client program was written to run on pnode3 and collect data. The function of the client program is shown below in pseudo code: the *diff_time* for a run of 100 loops were averaged to obtain

```
.....

getstandardtime (begin_time);
getremotetime (remote_time);
getstandardtime (end_time);
average_times (begin_time, end_time, avg_time);
subtract_times (avg_time, remote_time, diff_time);
.....
```

Figure 6-7: Global Time Measurement Pseudo Code

the data. The above method assumes that the *remote_time* returned is obtained at the midpoint between *begin_time* and *end_time*. However, the obtained data indicate that this is not quit true, though very close.

6.2.3.1. Results

Figure 6-8 shows a summary of the collected data. As stated before, each entry is the average of 100 loops, and all times are in seconds. As seen in Figure 6-8, measurements were taken on two separate days, with each day consisting of two measurements. The *Difference* field is the average time difference between standard host and remote host. The *Elapse* field is how long it took to obtain the remote time. The big difference in times for host pnode1 are due to the fact that it was re-booted between the two days. The time differences between the two days for pnode2 and pnode4 show a drift among standard and remote clocks. The results for pnode3 are the case where standard and remote hosts are the same Ideally, the results for pnode3 would be 0, but the non-zero figures are due to our assumption that remote time measurement is taken at the median point.

The difference in time between measurements taken on the same day range from a low of 0.5

Host	Meas. Time	Std. Dev.	Difference	Elapse
pnode1	Aug. 1, 02:00	0.0120	51.6488	
		0.0116	51.6956	
	Aug. 2, 03:00	0.0224	-35.1519	0.3587
		0.0114	-35.1758	
pnode2	Aug. 1, 02:00	0.0188	356.5524	
		0.0501	356.5828	
	Aug. 2, 03:00	0.0281	367.9537	0.3394
		0.0115	367.9713	
pnode3	Aug. 1, 02:00	0.0230	-0.0518	
		0.0126	-0.0358	
	Aug. 2, 03:00	0.0074	-0.0367	0.1830
		0.0101	-0.0382	
pnode4	Aug. 1, 02:00	0.0230	1.1057	
		0.0173	1.1051	
	Aug. 2, 03:00	0.0303	-0.5899	0.3149
		0.0281	-0.5894	

Figure 6-8: Global Time Comparison

millisecond (pnode4 on day 2) to a high of 46.8 millisecond (pnode1 on day 1). Most variations were within 25 milliseconds. This time difference measurement technique is a potential method for obtaining a global standard time (e.g. time on pnode4 is standard an others are adjusted).

6.2.3.2. GetDATE and gettimeofday

Measurements on the execution time differences for Cronus' *GetDATE* and the UNIX *gettimeofday* were also performed and are shown in Figure 6-9.

# loops	gettimeofday	GetDATE
1000	0.49	0.56
10000	5.01	5.66
100000	50.66	57.64

Figure 6-9: GetDATE vs. gettimeofday

6.3. Example Application

6.3.1. Matrix Multiplication

A matrix multiplication was performed in parallel on the four different pnodes. The two multiplicand matrices and the result matrix are maintained as shared data via the use of a Cronus manager. The result matrix is divided into four parts, with the task of calculating each part's values being assigned to the four different nodes.

A starter program is used to either receive the multiplicand matrices as *tty* or *file* input. This program then assigns coordinates for each host and starts the four parallel executions via our preliminary implementation of WC. Within each host, execution time is measured and logged into a file.

Since the Cronus manager of the matrices is on one host machine, and all other hosts must access this manager for *read*s and *write*s, the manager is a potential bottleneck. Thus, a future experiment might be to try replicated objects for the matrices on each host. This report shows the results of having just one matrix manager. The first implementation was tried with single access (i.e. read and write one matrix coordinate at a time). In the second implementation, *read*s are performed in units of a line. Thus, a whole row or column is returned with each *read*, reducing the number of operations that must be called on the matrix manager. The third and fourth implementations were instrumented versions of the matrix multiplier.

6.3.1.1. Results

The results of Figure 6-10 are execution times at each local node for performing its section of a 10 x 10 matrix multiplication. The results for single *read* and *row read* access are shown, followed by instrumented versions. The matrix manager was residing on pnode4.

The *with sensors* entry shows the time measurement results for instrumented code. Two methods for data storage (via a Cronus manager and direct file write) were used. The sensors are locally buffered in a queue size of 30. 18 sensors were inserted at write time (static) which resulted in 442 sensors at run time (dynamic). These 442 sensors resulted in 14,400 bytes of sensor data to be written into the data file. The instrumented code also used *row access*.

The time measurement results are summarized in Figure 6-10. It can be seen that user and system times take up a very small portion of overall execution time. This is due to the waiting periods involved when operations are performed either on the matrix manager or the sensor manager (DCA). With direct file writes as DCA, there is no wait time since no manager operations are involved, thereby resulting in faster execution speeds than the one with sensor managers. The execution times for the direct file write method would decrease with a larger queue size.

The code that runs on each host to calculate a section of the result matrix is included in the appendix. The sensors are of form P_SENSOR (SEN_TYPE, STAT_ID) and can be seen in this code.

Method	Host	Elps. Time	Usr. Time	Sys. Time
Single access	Pnode1	268.48		
	Pnode2	361.75		
	Pnode3	427.09		
	Pnode4	95.96		
Row access	Pnode1	24.75	0.38	0.39
	Pnode2	29.33	0.40	0.41
	Pnode3	19.46	0.38	0.38
	Pnode4	9.22	0.36	0.40
With sensors (Manager as DCA)	Pnode1	35.12	1.49	0.84
	Pnode2	38.26	1.47	1.30
	Pnode3	41.96	1.46	1.02
	Pnode4	14.34	1.41	0.95
With sensors (Direct file write as DCA)	Pnode1	27.01	0.50	0.79
	Pnode2	33.13	0.49	0.99
	Pnode3	24.11	0.48	1.10
	Pnode4	10.22	0.51	0.78

Figure 6-10: Matrix Multiplication Time Measurements

7. Design Conclusions

Now, when all of the Paradise mechanisms and tools have been described bottom up, let us take a top down look on the whole design and try to analyze the various approaches taken. Let us also try to determine how these approaches facilitate the implementation of various policies needed in order to **run, control, monitor, stimulate** and **collect data** from an experiment.

Paradise enforces a hierarchical relational model of the world which consists of objects and their relations. Each of the hierarchical layers supports its own set of objects and the operations on those objects. In order to deal effectively with some layer, the User must have a policy at hand which deals with the specific objects and their relations, and more importantly, knows how to map the objects and relations from the adjacent hierarchical layers into those known to the User at the layer of current interest. In order to clarify the previous statement, let us assume that the policy in question deals with monitoring of a Host OS kernel. Let us further assume that, in a simplified view, objects known to that hierarchical layer are **processes** and **queues** and that the only relations known are relations between these objects (i.e. process can be waiting in some queue or not). In actuality, data collected from an experiment will consist of a set of event records and their absolute time of occurrence.

Unlike other models, Paradise abstracts Users at layers other than the sensor/event layer from having to know anything about the events and their relations during the development, as well as during the data analysis phase. In another words, if somebody wants to observe a process in a Host OS, it is possible to make that process visible without knowing the exact sensor ids that have to be involved in order to make this happen. We will also be able to analyze the behavior of that process without detailed knowledge about the same sensors and the events they produce. It is the designer of the Host OS monitoring policy who makes all the necessary mappings from OS events to objects called processes (and back) available to other Users via the IP.

In order to allow for policies which are independent of underlying mechanisms, Paradise has to adapt the unique interface to all the relations and the objects generated at all the hierarchical layers. It also has to assure that there is a basis for mechanisms which will support mapping from one layer to another if the relations for each layer are known separately. In this design, both goals are achieved by the use of a relational database system acting as an integration platform (IP).

At the paramount of the Paradise design hierarchy is an object called the **experiment**. An experiment is a test of some kind which is run with the help of the Paradise system in order to retrieve specific data about the behavior of the system under test. In Paradise, the User first **prepares** an experiment. Then the User **runs** that experiment. During the experiment run, the User may or may not choose to **interact** with the experiment in question. The User is supposed to **collect data** and **kill** the experiment at the end of its execution. Finally, the User takes the data collected and performs various analysis and/or compares it with the data collected from other

experiments. All of the Users actions can be made automatic via execution of a previously prepared experiment script.

An experiment consists of four distinguished objects: a **workload**, a **monitor profile**, a **fault profile** and **experiment data**. The Paradise User deals with these objects in the various phases of an experiment.

In the present design we chose to associate the two topmost menus in the User interface[32] with the process of experiment preparation. The functions from the first menu (i.e. "**Library Preparation**") deal mostly with the design of a workload by designing the APCs that constitute that workload. They also deal with the design of fault classes for that workload. According to our experience with the similar existing systems, it is most likely that some parts of a workload and/or a fault class will be reused by the same, or by other Users. Due to this fact, we decided to support internal Paradise workload and fault **libraries** in order to facilitate the process of sharing and maintaining of the reusable components. Paradise tools that are used for maintenance of these libraries are collectively called **librarians**.

The menu called "**Experiment Definition**" deals with the definition and the creation of monitoring and fault profiles and the production of an experiment script. In our design we decided to facilitate this process by:

- Creation of the high level commands for the **Paradise Experiment Description Language** (i.e. by adding flexible control flow, synchronization and i/o commands, such as branches and loops, waits etc.)
- providing a **graphical interface** to display development time relations at each level of the hierarchy.

Our design assumes that during the **Experiment Execution** phase an experiment script is interpreted by the Paradise Experiment Controller. It was our decision that this experiment script could allow for not only a **batch** mode of execution but would also allow for some **interaction** by adding **input** and **output** to the set of Paradise Experiment Script Commands. In our view it is very important that not only automatic experimentation, but also the interactive one is possible.[33]

There are two basic types of Paradise Script commands in this design depending upon the place where a particular command is executed. Some of them (like branching, i/o and synchronization related commands) are executed in PWS. The rest of the commands related to the **workload control**, **monitoring**, **fault injection**, and **data collection** are executed by the related controllers in each PNODE.

In this design, PMC is the Paradise peer for all the communication between a PNODE and the

---

[32]Please see Figure 5-3.

[33]One can envision more need for the automatic mode during the predeployment testing, while more interactions will be needed during the popstdeployment phase where User needs more direct scrutiny over execution.

rest of the Paradise system. It is very important that the **interface** to this peer support enough **functionality** to allow for the implementation of all the needed mechanisms. It is also important that this interface is **portable** to the variety of architectures. In order to fulfill these requirements, our design defines the PMC as a Cronus object manager. This decision automatically provides for portability. Due to the fact that the basic PEM/PMC structure supports a client-server model, even if we abstract Cronus, it will be easy to implement PEM/PMC with the rudimentary IPC mechanisms outside Cronus if necessary.

All modern operating systems provide, in one way or another, an abstraction of a process. Paradise supports the abstraction of an application client which almost directly maps into a process. However, operating systems differ in the way these processes communicate. Some of them provide mechanisms for shared memory, while others support data exchange via messages. The Paradise design assumes that there is no underlying support for memory sharing, but if certain PNODE architectures support it, related implementations can take advantage of that fact. In another words, the design we present here defines an abstraction of an attachment which is a logical alternate control flow inside an APC servicing the command requests from a PMC. The Interface through which this mechanism is implemented represents a standard remote procedure call (RPC) interface. A very important idea to notice here deals with the fact that PMC as a source of a command and an attachment as the target for that command *could* share the same memory space if possible (i.e. via another PNODE architecture, or another layer like OS).

As an exercise, to prove that the functionality of the designed interfaces suffice for building Paradise policies at various layers, let us compare the mechanisms and their implementations needed to support OS kernel and an APC monitoring policies. It is obvious that in our design one has to insert sensors into the OS kernel as well as into an APC in order to make their objects visible. It is also obvious that such an instrumented entity will have to keep a list of all related sensor control blocks in its global address space. This means that, even an OS kernel which is not a process but a collection of data structures and the procedures to deal with them, will have to have the same type of monitoring and fault profiles as an APC has. Moreover, the same commands will be used to manage these profiles. The only difference will be the implementation which is based, in the case of an OS kernel, on simple procedure calls and not on RPCs. However, for an OS which supports memory sharing, even this slight implementation difference disappears. If one would build a hardware sensor, it also would have to have a block of registers to control the operations, associated with it. There is no reason why a list of such control blocks couldn't be considered as a monitor profile and, consequently, handled in the same way. Exactly the same arguments could be given for the fault profiles and the fault injection policies.

Now that we have given the rationale for the proposed mechanisms and the interfaces that connect them, let us analyze the data on performance obtained from the feasibility study presented in Section 6 of this document. The obtained data can be classified into four different groups:

- Performance data of different notification mechanism implementations

- Performance data of the proposed implementation of a workload controller
- Performance data of two alternative timing mechanisms
- Performance data of a fully monitored test application

Performance analysis of various notification algorithms show that the notification implementation with a local queue and a direct file write operation on overflow gives an order of magnitude better performance than the other implementations measured. This made a clear choice possible in the proposed design. Let us also mention two design features, which, will in most of the practical situations, make the overhead of the event recording even less. Our design offers an interface to control the size of a monitoring queue. In some applications, where only relatively small number of events are recorded, the size of that queue can suffice to hold all the event records generated during the experiment. This means that there will be no overflow and, consequently, no file i/o - which is the most intrusive part of the notification process. A second design feature which reduces a number of flushes to the LRF is the mechanism of subordination which enables that, from the whole class of events, only those which happen in a predetermined order will be recorded, thus reducing the number of event records at the source with virtually no cost in performance.

The workload controller feasibility study shows that average measured APC creation time is 300-500 ms in the present implementation. Based on our experience with other systems we felt that this kind of performance is more than satisfactory.

Timing measurement shows two things. First, that the clocks in the different PNODES drift despite the fact that our local network has standard BSD clock synchronization servers in effect. Second, they show that standard Cronus interface to the time is adding some overhead to the underlying UNIX call it uses (gettimeofday/date ≈ 5.01/5.66) thus we decided to use the UNIX call.

The last group of measurements deal with a fully instrumented distributed matrix multiplication application. Measurements show that:

- 442 event records were generated and collected.
- Performance greatly depends on the way global objects (i.e. matrices) and how client's access to them are organized.
- The predominant part of the execution time is spent waiting for a responses to messages generated inside object operation calls, so the system resource utilization is low (cca. 10 - 20%).
- The overhead generated by the recording of all the 442 event records was less than 10%, even using very short queue length of 30 records.

Taking into consideration that normally a User would like to see about 10% to 20% of the visible events recorded, we expect that in practice, execution time overhead would be around 1% - 2% of the overall experiment execution time.

Appendix I
Sensor Object Manager Specification

The object specification file *collector.typedef* used with the **definetype** command to generate common portions of the sensor object manager:

```
type Collector = 109    /* 109 is a unique type number  */
    is primal           /* make the object primal       */
    abbrev is coll      /* abbreviation for convenience */
    subtype of Object;

cantype ENTRY           /* defining a queue entry       */
      representation is Entry:
      record
          Type:         U16I;
          Timestamp:    U32I;
          Client:       EUID;
          StatID:       U32I;
          DynID:        U32I;
          Status:       U16I;
          Info:          ASC;
      end ENTRY;

  operation SENSOR  /* SENSOR op has 2 parameters     */
      (Type:          U16I;
       StatID:        U16I;
       optional Data: ASC) /* optional data string    */

  operation RETRIEVE ()  /* RETRIEVE op only outputs  */
    returns
      (AnEntry: Entry);

  end type Collector;
```

The manager specification file *collector.mgr* used with the **genmgr** command:

```
manager "The Collector Manager"
    abbrev is coll

type coll
    variable representation is QUEUE
    coll implements all from coll
    obj implements rest
```

Appendix II
Workload Controller

This section includes parts of code that were used to implement the Workload Controller as a Cronus manager.

II.1. Manager Description

The following is the *cont.typedef* file that was used with the Cronus *definetype* command.

```
type PIE_Cont = 227
        is primal
        abbrev is ctrl
        subtype of Object;

cantype PROC_ENTRY
        representation is Proc_Entry:
        record
                Client:         EUID;
                Timestamp:      EDATE;
                PID:            U32I;
                Type:           U32I;
                Status:         U32I;
                ProgName:       ASC;
        end PROC_ENTRY;

generic operation CREATE_PROC(      /* Create a Process      */
        Name:   ASC;
        Args:   array of ASC);

generic operation GETLOG ()         /* Get Log of processes */
        returns (
        List:   array of PROC_ENTRY);

generic operation PSIGNAL (       /* Signal a process      */
        ProcName:       ASC);

end type PIE_Cont;
```

The following is the *cont.mgr* file that was used with the Cronus *genmgr* command to create the common sections of code.

```
manager "The Workload Controller"
        abbrev is ctrl

type ctrl
        variable representation is PROC_ENTRY
        ctrl implements all from ctrl
        obj implements rest
```

II.2. Workload Controller Operation Implementation

The following file includes the implementations of the operations that were defined on the Workload Controller. Non-significant and standard Cronus parts have been omitted.

```
/* $Revision$
**
**
**  Operation code for ctrl manager.
*/
#include "header.h"
#include <sys/types.h>
#include <sys/socket.h>
#ifndef lint
static char RCS[] =
        "$Header$";
#endif  /* lint */

int mgr_sd;

init_mgr(argc, argv)
    int           argc;
    char         *argv[];
{
    struct sockaddr mgr_sock;

    system ("rm -f PIE:MGR");
    if ((mgr_sd = socket (AF_UNIX, SOCK_DGRAM, PF_UNSPEC)) == -1)
    {
        Log (0, "ERROR Could not create manager socket.\n");
        return (ERROR);
    }
    strcpy (mgr_sock.sa_data, "PIE:MGR");
    mgr_sock.sa_family = AF_UNIX;
    if (bind (mgr_sd, &mgr_sock, sizeof (mgr_sock)) == -1)
    {
        Log (0, "ERROR Could not bind name to mgr_sock.\n");
        return (ERROR);
    }
}

ctrl_init()
{
    ....
}

create_database(t)
    TYPE t;
{
    ....
}



ctrlgenCREATEPROC(r, input, output)
    OperationParms *r;
```

```
    reqctrlgenCREATEPROC *input;
    repctrlgenCREATEPROC *output;
{
    long the_PID;
    UID new_UID, *prin_UID;
    Proc_Entry New_entry;
    ObjectDescriptor *objdes;

    GetDATE (&New_message.Timestamp);
    if (input->dimensions.Args != 0)
        input->Args[input->dimensions.Args] = NULL;
    else
    {
        input->Args[0] = (char *)sditto (input->Name);
        input->Args[1] = NULL;
    }
    if ((the_PID = fork()) == 0)
    {
        execv (input->Name, input->Args);
        Log (0, "ERROR execl.\n");
        exit (1);
    }
    if (the_PID == -1L)
    {
        Log(0, "ERROR fork\n");
        return;
    }
                .
                .

        Create Cronus object.
                .
                .

    New_entry.PID = the_PID;
    New_entry.Status = RUNNING;
    New_entry.ProgName = (char *)sditto (input->Name);
                .
                .
          Store object.
}

ctrlgenGETLOG(r, input, output)
    OperationParms *r;
    reqctrlgenGETLOG *input;
    repctrlgenGETLOG *output;
{
    static Proc_Entry *Founds[20];
    Proc_Entry *An_entry;
    ObjectDescriptor *objdes;
    HOSTNUM this_host;
    int i = 0;
    long The_handle;
    UID current_UID;

    for (The_handle = (long)DBNext(CT_PIE_Cont, (long)0, &current_UID);
         ((The_handle ISNT (long)ERROR) AND (The_handle ISNT (long)0));
```

```
        The_handle = (long)DBNext(CT_PIE_Cont, The_handle, &current_UID))
    {
        if (IsGenericUID (&current_UID)) continue;
        if ((objdes = (ObjectDescriptor *)ReadObjectDescriptor (&current_UID))
           == NULL)
        {
            Log (0, "NULL object descriptor.\n");
            continue;
        }
        if ((An_entry=(Proc_Entry *)GetVarData(PROC_ENTRY, objdes)) == NULL)
        {
            Nack (r, lasterror ());
            Log (0, "Invalid [NULL] pointer to data.\n");
            output->valid = FALSE;
            return;
        }

        Founds[i] = (Proc_Entry *)Talloc (sizeof (Proc_Entry));
        CopyUID (&An_entry->Client, &Founds[i]->Client);
        CopyDATE (&An_entry->Timestamp, &Founds[i]->Timestamp);
        Founds[i]->Type = An_entry->Type;
        Founds[i]->PID = An_entry->PID;
        Founds[i]->Status = An_entry->Status;

        Founds[i]->ProgName = (char *)sditto (An_entry->ProgName);

        FreeObjectDescriptor (objdes);
        FreeVarData (An_entry);
        i++;
    }
    if (The_handle == (long)ERROR)
    {
        Nack (r, lasterror ());
        Log (0, "DBNext error.\n");
        return;
    }
    output->dimensions.List = i;
    output->List = Founds;
    output->valid = TRUE;
}


ctrlgenPSIGNAL(r, input, output)
    OperationParms *r;
    reqctrlgenPIESENSOR *input;
    repctrlgenPIESENSOR *output;
{
    struct sockaddr send_sock, recv_sock;
    char send_msg[100], recv_msg[100];
    int send_len, recv_len;

    sprintf (send_msg, "Are you there?");
    send_len = strlen (send_msg);
    sprintf (send_sock.sa_data, "PNODE:%s", input->ProcName);
    send_sock.sa_family = AF_UNIX;
    if (sendto(mgr_sd,send_msg,send_len,0,&send_sock,sizeof(struct sockaddr))
        == -1)
```

```
    {
        Log (0, "ERROR Manager sendto.\n");
        return (ERROR);
    }
    recv_len = sizeof (struct sockaddr);
    if (recvfrom (mgr_sd, recv_msg, 100, 0, &recv_sock, &recv_len) == -1)
    {
        Log (0, "ERROR Manager recvfrom.\n");
        return (ERROR);
    }
}
```

Appendix III
Experimentation Example - Distributed Matrix Multiplication

This appendix describes a simple example of a Paradise experiment. In the example the application used is the distributed matrix multiplication described in Section 6.3.1.

The example is presented in two sections:

- Fault Free Experimention (FFE) - a workload is developed and instrumented, and an experiment is prepared and executed.
- Fault Injection Experimentation (FIE) - using the same workload as the Fault Free case, an experiment is prepared for Fault Injection.

The reason for this approach is to provide a clarity of view by separating issues of concern. The fault free phase focuses on the issues of workload preparation, workload instrumentation, experiment preparation, experiment execution, and data management. This is the usual paradigm with which people are familiar. The fault injection phase focuses on the issues of fault preparation which in itself is a complex topic. This approach actually follows the expected manner of experiment development - first deal with the the deterministic facets of behavior and then introduce the non-deterministic elements. Experience has shown that experimentation usually proceeds in such a graduated manner.

III.1. The Paradise Environment

The purpose of these examples is to comprehensively illustrate the interrelationship of the principal players in experimentation:

- The system User
- The Paradise system
- The workload under test
- The underlying testbed

III.1.1. Experimentation in the Paradise Environment

Before delving into the details of the examples, we first quickly review the major items of experimentation and how they are presented in the examples. The experimentation process includes:

- Workload Preparation
  - Development of the application code
  - Automated instrumentation
  - Sensor profiles
  - Element profiles
  - Monitor Fault Profiles

- Associated data files
- Workload Definition

- Fault Preparation
  - Fault Class Definition
  - Methods and Domains
  - Fault Generation
  - Fault Lists
  - Fault Profiles
- Experiment Preparation
  - Experiment Definition
  - Experiment Script (Compile)
- Experiment Execution
  - Experiment Script (Interpret)
  - Start/Stop experiment
  - User interaction
  - Real-time data presentation
- Data Collection
  - Collect data contained in the Local Repository Files (LRF)
  - Data conversion to Paradise Canonical form
  - Data cataloging and storage
- Data Analysis (Statistical)
  - Specification of analysis tools
  - Development of specialized tools
  - Experimentation reports
- Data Presentation (Graphical)
  - Specification of presentation tools
  - Development of specialized presentation tools

All of the above operations and objects must be managed within the context of a specific experiment. The purpose of the above list is to accent the point that experimentation in itself is a complex undertaking. If one takes into consideration the additional prospect of multiple experiments and multiple Users, the task becomes quite daunting. The purpose of Paradise is to free the User from these issues and allow the User to deal with experiments as entities in their own right.

III.1.2. Working in the Paradise Environment

The User performs all development operations within the Paradise environment. The User interacts with Paradise by means of the Paradise graphical interface which may include an icon/menu package.

The User/Paradise operate within the context of an experiment. Thus, the User enters Paradise at the top level menu and chooses the experiment domain. All of the information concerning the chosen experiment is now available for User review and inspection. The experiment status may be determined as well as the current inventory of components. The use of the menu interface enforces procedural rules which in turn permits Paradise to perform the administrative functions for experiment management.

It is expected that experiments will be at various stages of development at one time or another. Paradise introduces a great degree of flexibility in the manner of preparing and executing experiments. The User may develop workloads, faults, experiment descriptions, etc. at any time or in any order. Data analysis may be performed at a later, convenient time.

As the User develops an experiment (i.e. components), Paradise performs relevant administrative functions which maintain the current status of the Experiment development. When an experiment is executed, the generated data also becomes a part of this administrative domain.

Hidden in this discussion is an extremely important fact. Since Paradise administers the system, it becomes possible to share experimental resources and components among experiments. Thus workloads, fault classes, experiment descriptions may be shared, thereby eliminating unnecessary replication. Version control - the correlation of workloads, faults and experiment data - is automatically performed. This functionality is implemented with the Paradise Librarians.

The important point to keep in mind while reading these examples is that the Paradise system performs the arduous administrative tasks while freeing the User for the more creative facets of experimentation.

III.2. Description of the Example Workload and Experiments

The example workload is a simple variation of the matrix multiplication example discussed in Section 6.3.1. Two multiplicand matrices and the result matrix are maintained as shared data by an Application Manager (APM) residing in an individual PNode. The multiplication task itself is divided into four parallel activities. The result matrix is partitioned into four parts, and the task of calculating the values of each part is assigned to an application client (APC) residing in an individual PNode. Thus, there are a total of five PNodes utilized in this example.

The general scenario of the example is to prepare two experiments, one in which the workload executes in Fault Free mode, and one in which the workload executes with Fault Injection. The first

covers the mechanics of preparing the workload, experiment description, and data management (i.e. collection, analysis, presentation). The second covers the mechanics of generating faults and then, utilizing the previous fault free developments, discusses the manner of performing experiments with fault injection.

Each example is described in terms of seven phases:

- Workload Preparation
- Fault Preparation
- Experiment Description
- Experiment Execution
- Data Collection
- Data Analysis
- Data Presentation

III.3. Fault Free Experimentation

The purpose of this example section is to illustrate the Paradise methodology for preparing a Fault Free experiment. The primary topics discussed are:

- Workload Preparation
- Experiment Description (Section III.3.3)
- Experiment Execution (Section III.3.4)
- Data Collection
- Data Analysis
- Data Presentation

III.3.1. Workload Preparation

There are three main bodies of code for this particular example: SAPC, matrix object manager (APM), and the APC executing in each of four nodes. The code for the three are shown in section III.3.1.4, III.3.1.2, and III.3.1.3 respectively. For the sake of brevity, only the APC code is then followed through the Paradise development process. The others are not presented since the process is similar. The fundamental development steps are:

- Application Code Development (Section III.3.1.1)
- PPROC Pass and Sensor Profile Generation (Section III.3.1.5)
- Compilation and POPROC Pass and Element Profile Generation (Section III.3.1.8)
- Monitor Fault Profile Generation (Section III.3.1.10)

The last step is to create the Workload Definition (See Section III.3.1.11) which describes the topographical specification of the workload. This specifies the PNodes in which the APMs and APCs will execute plus other information for workload startup.

III.3.1.1. Application Code Development

The first step in actually implementing an application in Paradise is to specify the application code. The code for the SAPC and APC for this example is written in the C language while the matrix manager (APM) is specified in the standard Cronus manner.

III.3.1.2. Matrix Object Manager (APM)

The following is the *mat.typedef* file used with the Cronus command *definetype* that specifies the object types in the matrix manager. The C style comments were added for clarity.

```
type PIE_matrix2 = 234
        is primal
        abbrev is matrix2
        subtype of Object;

cantype A_COLUMN
        representation is A_Column:
        record
                Column: array of F32;
        end A_COLUMN;

cantype THE_MATRIX
        representation is The_Matrix:
        record
                Row:    array of A_COLUMN;
        end THE_MATRIX;

generic operation INITIALIZE (          /* Set to 0              */
        rows:   U32I;                   /* Number of rows        */
        columns:U32I);                  /*       and columns     *

generic operation GET_ROW_VALUE (       /* Return specified row */
        row:    U32I;
        which:  U16I)       /* one of 3 matrices: 2 in & 1 out */
        returns (
        Values: array of F32);

generic operation GET_COLUMN_VALUE ( /* Return specified column */
        column: U32I;
        which:  U16I)
        returns (
        Values: array of F32);

generic operation PUT_VALUE (           /* Put value in coordinate */
        row:    U32I;
        column: U32I;
        which:  U16I;
        Value:  F32);

end type PIE_matrix2;
```

Shown below is the *mat.mgr* file that is used with the Cronus *genmgr* command.

```
manager "The Matrix Manager"
```

```
        abbrev is mtrx

type matrix2
        variable representation is THE_MATRIX
        matrix2 implements all from matrix2
        obj implements rest
```

After the *definetype* and *genmgr* Cronus commands are run, several files will have been generated. Of these, the User takes the *mtrxops.skl* file to specify the implementation of the operations in *mtrxops.c*. It is this *mtrxops.c* file that is instrumented in a similar manner as the APC code described in Section III.3.1.6.

III.3.1.3. APC Code

This section shows the matrix multiplication program that is executing in parallel on the four different nodes. Note that no User sensors have been installed because the system sensors to be automatically installed are quite sufficient. As a quick preview, execution profile time measurements are obtained after experiment execution by comparing the timestamps for the P_APCSTART and P_APCEND system sensors which are inserted in the PPROC pass as described in Section III.3.1.6.

```
main (argc, argv)
int argc;
char *argv[];
{
    float B_Columns[MAX_SIZE/4];    /* To store all columns used    */
    float *temp;                    /* ptr to row or column returned
                                       by psl operation routines */
    int x1, x2, y1, y2, sz, i, j, k, len;
    float sum;

    x1 = atoi (argv[1]);
    x2 = atoi (argv[2]);
    y1 = atoi (argv[3]);
    y2 = atoi (argv[4]);
    sz = atoi (argv[5]);

    for (j = y1; j <= y2; j++)  /* Obtain all columns that will be used */
    {
        pslmatrix2genGETCOLUMNVALUE(NULL, j, 2, &temp, &len);
        for (i = 0; i < len; i++)       /* Store in B_columns matrix */
            B_Columns[i][j-y1] = temp[i];
        free (temp);
    }
    for (i = x1; i <= x2; i++)          /* perform multiplication */
    {
        pslmatrix2genGETROWVALUE(NULL, i, 1, &temp, &len); /* Get row */
        for (j = y1; j <= y2; j++)
        {
            sum = 0;
            for (k = 0; k < sz; k++)    /* mult. element by element */
                sum += temp[k] * B_Columns[k][j-y1];
            pslmatrix2genPUTVALUE (NULL, i, j, 3, sum); /* Store result */
```

```
        }
        free (temp);
    }
}
```

III.3.1.4. SAPC Code

The following code is the User interface that will read in the two matrices to be multiplied and then start concurrent execution of the APC on the different nodes by invoking a *creat_proc* operation on the Workload Controller (WC) of each target node. The name of the SAPC is *starter*. Note that intermediate code is not presented since it is unnecessary for the purpose of this description.

```
#include "/usr/cronus/include/cronus.h"
#include <stdio.h>

extern char *sditto();

main (argc, argv)
int argc;
char *argv[];
{
    sz = atoi (argv[1]);
    if ((STRINGtoHOSTNUM ("pnode1", &host1) == ERROR) ||
        (STRINGtoHOSTNUM ("pnode2", &host2) == ERROR) ||
        (STRINGtoHOSTNUM ("pnode3", &host3) == ERROR) ||
        (STRINGtoHOSTNUM ("pnode4", &host4) == ERROR))
    {
        printf ("ERROR in STRINGtoHOSTNUM.\n");
        exit (1);
    }
    pslmatrix2genINITIALIZE (NULL, sz, sz);
                    .
       Read in matrix from file or tty.
                    .
       Initialize proper argument list.
                    .
    pslctrlgenCREATEPROC (&host1, "multme3", args, 6);
                    .
       Initialize proper argument list.
                    .
    pslctrlgenCREATEPROC (&host2, "multme3", args, 6);
                    .
       Initialize proper argument list.
                    .
    pslctrlgenCREATEPROC (&host3, "multme3", args, 6);
                    .
       Initialize proper argument list.
                    .
    pslctrlgenCREATEPROC (&host4, "multme3", args, 6);

    printf ("Use 'showme %d' to see results.\n", sz);
#endif
}
```

III.3.1.5. PPROC Pass

The Paradise PPROC inserts system sensors and extracts the sensor profile along with development time data. This section shows what the code looks like once it passes through the PPROC and has been instrumented. Recall that the two parameters for the *P_SENSOR* are *sensor type* and a Paradise assigned *static id*. Also shown is the sensor profile that is extracted from the APC code by the PPROC. *Once again, note that we are only following through the APC code due to space considerations.*

III.3.1.6. Instrumented APC Code

The following is the APC code of Section III.3.1.3 after instrumentation. In our example, no User sensors were used, but User sensors described in Section 4.1.2.1 could have been inserted when the source code was created.

```
main (argc, argv)
int argc;
char *argv[];
{
    float B_Columns[MAX_SIZE/4];     /* To store all columns used    */
    float *temp;                     /* ptr to row or column returned
                                        by psl operation routines */
    int x1, x2, y1, y2, sz, i, j, k, len;
    float sum;

    P_SENSOR (P_APCSTART, 1);
    x1 = atoi (argv[1]);
    x2 = atoi (argv[2]);
    y1 = atoi (argv[3]);
    y2 = atoi (argv[4]);
    sz = atoi (argv[5]);

    for (j = y1; j <= y2; j++)  /* obtain all columns that will be used */
    {
        P_SENSOR (P_LOOPTOP, 2);
        P_SENSOR (P_OBJENTRY, 3);
        pslmatrix2genGETCOLUMNVALUE(NULL, j, 2, &temp, &len);
        P_SENSOR (P_OBJEXIT, 4);
        for (i = 0; i < len; i++)       /* Store in B_columns matrix */
        {
            P_SENSOR (P_LOOPTOP, 5);
            B_Columns[i][j-y1] = temp[i];
            P_SENSOR (P_LOOPBOTTOM, 6);
        }
        free (temp);
        P_SENSOR (P_LOOPBOTTOM, 7);
    }
    for (i = x1; i <= x2; i++)          /* perform multiplication */
    {
        P_SENSOR (P_LOOPTOP, 8);
        P_SENSOR (P_OBJENTRY, 9);
        pslmatrix2genGETROWVALUE(NULL, i, 1, &temp, &len); /* get a row */
        P_SENSOR (P_OBJEXIT, 10);
```

```
        for (j = y1; j <= y2; j++)
        {
            P_SENSOR (P_LOOPTOP, 11);
            sum = 0;
            for (k = 0; k < sz; k++)     /* mult. element by element */
            {
                P_SENSOR (P_LOOPTOP, 12);
                sum += temp[k] * B_Columns[k][j-y1];
                P_SENSOR (P_LOOPBOTTOM, 13);
            }
            P_SENSOR (P_OBJENTRY, 14);
            pslmatrix2genPUTVALUE (NULL, i, j, 3, sum); /* Store result */
            P_SENSOR (P_OBJEXIT, 15);
            P_SENSOR (P_LOOPBOTTOM, 16);
        }
        free (temp);
        P_SENSOR (P_LOOPBOTTOM, 17);
    }
    P_SENSOR (P_APCEND, 18);
    P_SENSOR_FLUSH ();
}
```

III.3.1.7. Sensor Profile

The following is the Sensor Profile[34]. There is one sensor profile entry for each sensor in the APC code. The line number for the Sensor Profile entries will be the index into the Monitor Control Block within the APC. This line number will also be equal to the static ID associated with each system sensor. Note that all the sensors are initially disabled (i.e. Enable = 0), have no subordinates (i.e. Sub = 0), and have no linked fault targets (i.e. Flt = 0).

Line	Enable	Status	Pro	Act	Epi	Sub	Flt
1	0	31	0	0	0	0	0
1	0	31	0	0	0	0	0
2	0	31	0	0	0	0	0
3	0	31	0	0	0	0	0
4	0	31	0	0	0	0	0
5	0	31	0	0	0	0	0
6	0	31	0	0	0	0	0
7	0	31	0	0	0	0	0
8	0	31	0	0	0	0	0
9	0	31	0	0	0	0	0
10	0	31	0	0	0	0	0
11	0	31	0	0	0	0	0
12	0	31	0	0	0	0	0
13	0	31	0	0	0	0	0
14	0	31	0	0	0	0	0
15	0	31	0	0	0	0	0
16	0	31	0	0	0	0	0
17	0	31	0	0	0	0	0
18	0	31	0	0	0	0	0

Please note that all of the above sensors were automatically inserted into the User code. The user

[34]See section 2.2.2 for details.

now has the option to selectively enable any of the above sensors, which are initialized as disabled.

III.3.1.8. Compilation and POPROC

After all User application code is passed through the Paradise PPROC, it is compiled. The Matrix manager is linked with the *cronus* and *mgr* libraries (supported by Cronus) and the APCs are compiled with the *mtrxpsl.o* and *mtrxcts.o* files that were obtained when the matrix manager was passed through the *genmgr* command described in Appendix section III.3.1.2. The code is compiled with the debugging switch on so that the POPROC can extract the Element Profile.

III.3.1.9. Element Profile

The element profile consists of a list of the starting virtual address and length of all global variables, functions, and procedures. This information will later be used to generate a Fault List for the APC that will be combined with the sensor profile to obtain the Monitor Fault Profile.

Name	Start_address	Length (bytes)
main	0x56	0525
B_columns	0x7fffadcc	50
temp	0x7fffadc8	8
i	0x7ffffc00	4
j	0x7ffffbfc	4
k	0x7ffffbf8	4
y1	0x7ffffc0c	4
y2	0x7ffffc08	4
sz	0x7ffffc04	4
len	0x7ffffbf4	4
x1	0x7ffffc14	4
x2	0x7ffffc10	4
sum	0x7ffffbec	8
argc	0x7ffffaa4	4
argv	0x7ffffab4	8

III.3.1.10. Monitor Fault Profile (MFP) Generation

This is the last step in the application development. In general the information from the APC fault list and the sensor profile are combined to obtain the Monitor Fault Profile.

In this fault free example, however, no fault list has been generated and/or specified, so no fault profile is included in this MFP. Consequently, the only remaining step is for the User to specify the sensors to be enabled in order to make the instrumentation visible. This can be done in one of three ways:

- Interactively, by using the Paradise graphic interface. The User displays the Paradise Paramap and selects the desired sensors either by type or individually using the mouse pointer, for example.
- Dynamically, by issuing P_set_sensor() (see Section 3.2.3) commands to a specific APC. Various sensor parameters (including enable) can be modified this way. This allows

different variations of an experiment to be performed.

- Statically, by associating a type of sensor with a class of faults. This command will be part of an experiment description.

Suppose that the User is interested in profiling the APC execution. By enabling the P_APCSTART (#1) and P_APCEND (#18) sensors, the execution time of the APC can be derived during the Data Analysis phase of the experiment. The *act_base* (i.e Act = 1) for these two sensors is set at 1, thus the sensors will create an event record for only the first firing. The Sensor Profile now looks as follows:

```
Line      |Enable  Status  Pro     Act     Epi     Sub     Flt
  1       | 1      31      0       1       0       0       0
  2       | 0      31      0       0       0       0       0
  3       | 0      31      0       0       0       0       0
  4       | 0      31      0       0       0       0       0
  5       | 0      31      0       0       0       0       0
  6       | 0      31      0       0       0       0       0
  7       | 0      31      0       0       0       0       0
  8       | 0      31      0       0       0       0       0
  9       | 0      31      0       0       0       0       0
 10       | 0      31      0       0       0       0       0
 11       | 0      31      0       0       0       0       0
 12       | 0      31      0       0       0       0       0
 13       | 0      31      0       0       0       0       0
 14       | 0      31      0       0       0       0       0
 15       | 0      31      0       0       0       0       0
 16       | 0      31      0       0       0       0       0
 17       | 0      31      0       0       0       0       0
 18       | 1      31      0       1       0       0       0
@
<null fault profile>
```

As a quick preview, when the APC code is transferred to the specified resident PNode(s), a copy of the MFP accompanies it. (Paradise automatically transfers all related files of an APC to the target PNodes based only the specification of the APC id.) When the APC starts executing, the MFP is used to build the monitor (and fault) control blocks discussed in Section 2.2.2.

III.3.1.11. Workload Definition

The last task of the User would be to describe to Paradise the workload distribution. This might be done in the following possible format:

```
Create (APC, multiply.c, pnode1, /usr/examples/matrix/multiply)
Create (APC, multiply.c, pnode2, /usr/examples/matrix/multiply)
Create (APC, multiply.c, pnode3, /usr/examples/matrix/multiply)
Create (APC, multiply.c, pnode4, /usr/examples/matrix/multiply)
Create (APM, matrix, pnode4, /usr/examples/matrix/mgr/mtrxmgr)
Create (SAPC, starter.c, pnode4, /usr/examples/matrix/starter)
```

The syntax for the *Create* command above is:

```
Create (code type, prog name, target host, target directory)
```

Paradise utilizes this information at experiment execution time. Paradise will obtain all named code from the User (either through interactive input or file input), send this code to the target directory in the target host, and compile the code and store it.

III.3.2. Fault Preparation

Since this is a fault free experiment, there is no need to perform this phase. Note, however, that in a mature Paradise system, certain tools and facilities would be available that would eliminate any requirement for User action here, even in the case of a fault injection experiment. By specifying a Fault Class Definition identifier via the Paradise interface, a certain class of faults would be automatically generated by means appropriate facilities in the Fault Library. (This set of faults is referred to as a Fault List.) This would have to be done before the MFP Generation step discussed above. See the fault injection experiment in Section III.4 for this discussion.

III.3.3. Experiment Description

The purpose of the Experiment Description is to specify the manner in which an experiment is to be conducted. The Experiment Description, for example, can be a text file object which the User creates via Paradise. The User selects the Experiment Description menu and generates the file using one the following possible options:

- copy an existing Experiment Description file
- edit an Experiment Description template file provided by Paradise
- fill in a form provided by Paradise
  (pertinent fields could be pre-filled by Paradise)
- create/edit a new Experiment Description file

The User typically provides the following information in defining an experiment:

- experiment id - provide an identification name for the experiment.
- workload definition - specify the workload to be utilized in the experiment and the distribution topography (See Section III.3.1.11).
- run regime - specify whether the experiment consists of one or multiple runs (i.e. describes experiment iteration).
- time regime - specify the time frames for experimentation runs.
- data collection regime - specify the frequency and time of data collection.
- data analysis regime - specify the type of analysis functions to be applied to the data and which information is to be placed in the IP.
- data presentation - specify the type and form of data processing for graphical and/or textual rendering of experiment data.

The User provides the appropriate experiment control commands and arguments to compose an Experiment Description. See Section 4 for a description of these commands.

As an example, consider one of the simplest forms of experimentation in which a workload is

executed in fault free mode for a specific period of time, and the resulting profile data is presented via Parascope as a bar graph.

The Experiment Description may look as follows:

```
/* Comment:
 This experiment executes the workload fault-free for 3 minutes
 and presents the workload client profile in the form of a bar graph
*/
    P_BEGIN
    experiment_start(FF_EXP1, MatMult, MatData)
    workload_start(starter, "", "", PNode4)
    wait(180sec, P_NOTME)
    workload_kill()
    data_collect()
    data_analyze(profile)
    data_present(profile, bargraph)
    experiment_end()
    P_END
```

The parameters have the following meanings;

- FF_EXP1
  - the Experiment ID is Fault Free Experiment #1.
- MatMult
  - utilize the workload definition for the Matrix Multiply workload
- MatData
  - use the input data file for the workload.
- starter
  - the SAPC for the workload is *starter*.
- "","
  - the command line parameters *argc, argv* are null.
- PNode4
  - the PNode where the SAPC is located.
- 180sec
  - run the workload for three minutes.
- P_NOTME
  - no sensors are used for duration control.
- profile
  - generate the workload profile.
- profile,bargraph
  - display the workload profile using a bargraph format.

Note: the functionality of these commands is described below as part of the Experiment Execution description. The parameter descriptions above illustrate the context of the experiment.

When the User has completed the Experiment Description, Paradise can perform a syntax/semantic check and then store the Experiment Description. As part of these operations, Paradise updates the administrative status information for the experiment.

Sometime before the actual execution of the experiment, the User directs Paradise to compile the Experiment Description and generate an Experiment Script. The Script is simply an ordered set of system level commands that implement the required operations and invocations necessary to perform the actual experiment. The Paradise system automatically produces the required type and number of commands. As part of this process Paradise inserts the necessary parameters, such as PNodes, addresses, flags, etc. If desired, the User can inspect the Script to verify that the Experiment Description was properly specified originally. Paradise stores the Script, and the User can retrieve or use it at any time.

Note: the exact format of these commands is beyond the current scope of this report. The nature and characteristics of these Script commands can be inferred from the description of experiment execution below.

III.3.4. Experiment Execution

At this point the actual execution of the experiment ensues. The User selects the Experiment Execution menu and specifies the Experiment Script. The Runtime Experiment Controller (REC) interprets the Experiment Script and performs the specified operations to implement the experiment. The salient operations, as specified in the Experiment Description of Section III.3.3, are as follows:

- The following operations are performed as a result of the **experiment_start()** command:
  - The experiment id *FF_EXP1* is established as the reference label for Paradise. The resultant data of the experiment is now tagged with this identifier. This is the reference for the Integration Platform.
  - Paradise accesses the Workload Definition and, using the information therein, performs the following operations automatically:
    - Verify the network configuration
    - (Compile modules if required)
    - Distribute executable images to proper PNodes for each APC.
    - Distribute monitor fault profile (MFP) for each APC.
    - Transfer data file to proper PNode
  - At this point the workload is ready to be executed
- The following operations are performed as a result of the **workload_start()** command:
  - The SAPC is created using argc,argv as the command line variables.
  - The SAPC in turn creates the workload entities (APMs, APCs)
  - As each APC starts up:
    - the corresponding MFP file is read
    - the sensor control blocks are built
    - the fault control blocks are built
    - the corresponding LRF file is created

- At this point, the workload is executing
- As each APC executes, the installed sensors are encountered. Depending upon the filtering flags, the enabled sensors fire and event records are generated. *In this example only the entry/exit sensors are enabled.*

- The following operations are performed as a result of the **wait**() command:
  - The workload continues to execute
  - (if any queues fill up, they are flushed to the LRF)
  - There is a pause in the REC for 180 seconds
- The following operations are performed as a result of the **workload_kill**() command:
  - (note: Some APC entities may have self terminated or crashed.)
  - Commands are issued to each APC to terminate.
  - The data queues are flushed to the LRFs.
- The following operations are performed as a result of the **data_collect**() command:
  - The LRF in each PNode is transferred to the PWS.
  - The data files are organized and tabulated as part of the Paradise administrative function.
  - When required, LRfs from heterogeneous PNodes are converted to the Paradise canonical data form.
- The following operations are performed as a result of the **data_analyze**() command:
  - Pertinent information is extracted from the collection of data files and entered into the data base
  - A complex query is performed according to the requirements for generating a workload *profile*.
  - The resulting data is stored in the data base.
- The following operations are performed as a result of the **data_present**() command:
  - A device which implements the *profile* graphic presentation is invoked.
  - The device accesses the data base and, using the argument *bargraph*, generates the window and the display.
  - (At this or a later point in time the User can utilize the display menus to vary the details of the display, e.g. zoom in/out)
- The following operations are performed as a result of the **experiment_end**() command:
  - Reboot crashed PNodes.
  - Re-initialize the PMC in each PNode.
  - Perform associated housekeeping functions.

III.3.5. Data Collection

As the workload executes, event records are deposited in a local APC queue. If there is a queue overflow, the contents are flushed to the LRF.

During data collection the LRF in each PNode is transferred to the PWS. The data files are organized and tabulated as part of the Paradise administrative function. When required, LRFs from

heterogeneous PNodes are converted to the Paradise canonical data form so they present a uniform base of data.

III.3.6. Data Analysis

In this fault free version of the example, the command *data_analyze(profile)* directs Paradise to apply the required tools to analyze the data and to compile an execution profile of the workload. The start and termination times of all APCs are listed.

III.3.7. Data Presentation

In this fault free version of the example, the command *data_present(profile, bargraph)* directs Paradise to apply the required tools to analyze the data and to create a graphical view of the execution profile of the workload. The start and termination times of all APCs are indicated.

III.4. Fault Injection Experimentation

The purpose of this example section is to illustrate the Paradise methodology for preparing an experiment which includes fault injection.

The primary difference between the preparation of a fault free experiment (FFE) and a fault injected experiment (FIE) is found in the MFP used by the APCs in the workload. There are two such significant differences in the MFP: (See Section 2.2.2)

- fault profile entries are included
- sensor/fault links are established

Recall that fault injection is triggered by a sensor firing. (See Section 2.3.2) This statement presumes the following:

- the APC control flow induces the sensor to fire
- the sensor is enabled
- a fault is specified in the fault profile
- the sensor is linked to a fault

The fundamental step then is to generate the faults which constitute the fault profile and to establish the link to the sensor which is supposed to trigger the fault. Note that this is done for each APC that is to be fault injected. The two required steps are referred to as:

- Fault List creation
- Fault Installation

The primary fact to keep in mind is that fault injection experiments proceed exactly as the fault free type except for the additional steps of generating and installing the faults. In fact, experience has shown that this is the usual manner is which Users proceed in the development of an experiment. Also keep in mind that these two steps are performed separately.

III.4.1. Workload Preparation

The workload has been prepared in the fault free phase, and it can be used without change in this fault injection experiment. All of the required code, sensor profiles, element profiles and MFP files are maintained by the Paradise Librarian.

III.4.2. Fault Preparation

The purpose of Fault Preparation is to specify the manner in which faults are to be inserted into a workload during an experiment. Paradise provides a set of tools/facilities with which the User can characterize the faults.

III.4.2.1. Fault Generation Tools

The facilities for generating faults consist of:

- Fault Class Definitions (FCD)
- Domains
- Methods

These are discussed in Section 5.1.2.

Recall that a fault instance can be viewed as an ordered set of specifiers such as: which APC, which element, which sensor, which mask, and so on. A *domain* can be considered simply as a set of alternatives for for each specifier. A *method can be considered* as a selection function over a domain. The method reflects the characteristic of the type of faults that the User wishes to generate.

As discussed in the report, the domains and methods are maintained by the Fault Librarian.

In general, domains are workload specific since they reflect the *workload virtual address*. On the other hand, methods are not workload specific but rather general in nature. In the Paradise system the Users can develop their own methods if so deemed necessary; these are also maintained by the Fault Librarian.

The Fault Class Definition provides a high-level capability by which the User can specify and associate domains and methods that produce the desired faults. Paradise uses the FCD specifications to proceed through the complex process of accessing and selecting the necessary information from the IP in order to produce faults and then integrate them into MFPs of the APCs to be fault injected.

As an example, consider a simple form of experiment fault injection. The following factors characterize the general nature of fault injection.

- Select a set of APC's to be faulted. This set can be all, some, or one of the existing APCs. This set is the *domain* of possible APCs that may be fault injected. Determine the *method* of APC selection from the domain of possibilities.
- Determine where in the APC address space the fault is to be inserted (i.e. corrupt a memory segment of one or more bytes). Determine the how this location is selected.

- Determine when the fault is to be injected. This implies that a sensor(s) be selected since Paradise fault injection is sensor-based.
- Determine the nature of the fault, namely length, mask, etc.

III.4.2.2. Creation of Fault Class Definitions

Now look at these steps in more detail: It is assumed that Paradise maintains a list of the APCs in each workload, this list may be referred to as the *WL_Profile*. This list can be used to constitute a *domain*.

Consider the case in which the User has no specific APC in mind to fault inject, so defers the decision to Paradise. This is done by associating the method *random()* with the domain *wl_APClist*.

Note that the particular workload is known in the current Paradise environment, so the correct wl_APClist is implicitly defined.

Similarly, the User has no specific location with the APC address space in mind, so defers the the decision to Paradise again. This is done by associating the method *random* with the Element Profile of the selected APC.

Since elements have a certain size, the displacement into the element address space can also be randomly chosen. Of course the maximum displacement depends upon the element size, so Paradise provides the required information

Note that the same *random()* method is used above; methods are generalized over the set of possible domains.

Now assume that the User wishes to zero four bytes of the APC address space starting at the virtual address generated by the above steps. The method *zero_mask()* generates a four byte mask. The *null* domain is specified since the method performs no selection function in this case. The method *mask_insert(move)* stipulates that the mask generated is used to directly replace the memory contents.

Lastly, the User wants the fault injection to occur when the APC executes the P_APCSTART sensor after program execution begins. The FCD may look as follows:

```
/*  Comment:
    This FCD selects a single address space location in a workload
    and sets four (4) bytes to zero.

    Domain              Method
*/

    WL_Profile          random()
    elem_profile        random()
    size_profile        random()
    null                mask_insert(move)
    null                zero_mask(4 bytes)
    sens_profile        specific(P_APCSTART)
```

III.4.2.3. Generation of Fault lists

Using Paradise, the User can specify the number of faults to generate given the Fault Class Definition. When they are generated, Paradise places them into respective Fault Lists according to APCs.

The simplest case consists of single workload fault. The User selects the above FCD and specifies that one (1) fault instance is to be generated. The generated fault may look as follows:

```
multme3 j 0 move \00\00\00\00 P_APCSTART
```

The interpretation of the fault is: fault inject APC *multme*, at the location of element *j* at a displacement of *0 bytes*. Replace the contents of these four bytes with zero. Perform the fault injection when the P_LOOPTOP sensor fires.

At this point Paradise creates a fault profile for the APC *multme3* and the fault instance is entered. Subsequently this fault profile will be joined to the sensor profile of the same APC during the generation of the Monitor Fault Profile.

Please keep in mind that the simple example above in no way represents the total scope of fault generation. The number of combinations and variations of Fault Class Definitions, methods, domains (and User intervention) is practically limitless. When the number of faults to be generated is more than one, Paradise creates the required fault profiles and stores the fault instances in the appropriate ones.

Note the capability that Paradise provides. By simply creating the FCD above, the User directs the Paradise system to perform a complex assortment of operations with a diverse set of information. Since Paradise administers/manages all of the above entities, the procedure is performed totally automatically.

Also note that Paradise provides for direct User intervention. The single fault instance above could have been generated directly by the User, and then directed to a Fault List. As Users gain familiarity with Paradise, they will develop the facility to direct and manipulate the system tools in

very specific ways.

III.4.2.4. MFP Generation and Fault Installation

In the last step of application development, the information from the fault list and the sensor profile are combined to obtain the Monitor Fault Profile. This process includes the following steps:

- insert the fault list into the fault profile segment of the MFP (i.e after the "@" marker).
- find the triggering sensor in the sensor profile segment of the MFP.
- enable the sensor if is not already so.
- insert the fault index into the sensor definition.
  (This links the sensor to the fault).

At this point, the only remaining step is for the User to specify the sensors to be enabled in order to make the instrumentation visible. This is done interactively using the Paradise graphic interface. The User displays the Paradise "roadmap" and selects the desired sensors either by type or individually using the mouse pointer. The sensors enabled in the fault free case (#1, #18) are to be utilized again since the objective is to observe any variations in the workload profile. Since it is specified in the fault instance to perform fault injection at the beginning of the multiplication for-loop, Paradise enables the P_LOOPTOP sensor (#2) enables and a link is established to the fault instance. (Note: the fault instance doesn't include the *multme3* field because it is now located in the MFP file for that APC. The *P_LOOPTOP* field is not included also since the link to that sensor is already established. There is one fault (i.e. Flt = 1) associated with sensor #2 which is specified by the fault index 1 (i.e. Ind = 1).

```
Line      |Enable  Status  Pro     Act     Epi     Sub     Flt     Ind
 1        | 1      31      0       1       0       0       0
 2        | 1      31      0       0       0       0       1       1
 3        | 0      31      0       0       0       0       0
 4        | 0      31      0       0       0       0       0
 5        | 0      31      0       0       0       0       0
 6        | 0      31      0       0       0       0       0
 7        | 0      31      0       0       0       0       0
 8        | 0      31      0       0       0       0       0
 9        | 0      31      0       0       0       0       0
10        | 0      31      0       0       0       0       0
11        | 0      31      0       0       0       0       0
12        | 0      31      0       0       0       0       0
13        | 0      31      0       0       0       0       0
14        | 0      31      0       0       0       0       0
15        | 0      31      0       0       0       0       0
16        | 0      31      0       0       0       0       0
17        | 0      31      0       0       0       0       0
18        | 1      31      0       1       0       0       0
@
 1        | j 0 move \00\00\00\00
```

III.4.3. Experiment Description

With respect to the fault free version of the experiment, the new Experiment Definition may be exactly the same except for an altered Experiment ID parameter. An additional command has been included which determines the fault latency time for the workload.

```
/* Comment
 This experiment executes the workload fault-free for 3 minutes
 and presents the client scheduling in the form of a bar graph
*/
    P_BEGIN
    experiment_start(FI_EXP1, MatMult, MatData)
    workload_start(starter, "", "", PNode4)
    wait(180sec, P_NOTME)
    workload_kill()
    data_analyze(profile)
    data_analyze(fault_latency)
    data_present(profile, bargraph)
    experiment_end()
    P_END
```

The parameters now have the following meanings;

- FI_EXP1
  - the Experiment ID is Fault Injection Experiment #1.

  (It was *FF_EXP1, i.e. Fault Free Experiment #1.*)
- MatMult
  - (unchanged) - utilize the workload definition for the Matrix Multiply workload.
- MatData
  - (unchanged) - use the input data file for the workload.
- starter
  - (unchanged) - the SAPC for the workload is *starter*.
- "", ""
  - (unchanged) - the command line parameters *argc, argv* are null.
- PNode4
  - (unchanged) - the PNode where the SAPC is located.
- 180sec
  - (unchanged) - run the workload for three minutes.
- P_NOTME
  - (unchanged) - no sensors are used for duration control.
- profile
  - (unchanged) - generate the workload profile (timeline)
- fault_latency
  - determine when/where fault was detected
- profile,bargraph
  - (unchanged) - display the workload profile using a bargraph format.

III.4.4. Experiment Execution

The experiment is executed in the same manner as the fault free case. The primary difference is that the selected APC will fault inject at startup time when the APC_LOOPTOP sensor is executed. Now that the APC address space has been corrupted, the following situations may arise:

- nothing
  - the corruption caused no change in the APC execution.
- degradation
  - the corruption causes a response change, but no error is detected.
- error
  - the APC execution is altered in some way, and an error is detected.
- crash
  - the APC is killed because of some illegal operation.

Because of the APC error, the following situations may arise in the workload:

- lost message
  - a message was not sent/received.
- time shift message
  - a message was sent early/late.
- corrupted message
  - a message content was altered.

The reason for describing the execution in this manner is to illustrate that fault injection may cause network wide repercussions. With this in mind the User should take steps to properly instrument the workload so as to make these ramifications visible (observable). When done in this manner Paradise can then reconstruct the network wide phenomenon and present it to the User in the form of data and/or graphical representations by means of Parascope.

III.4.5. Data Collection

Data Collection proceeds in the same manner described in the Fault Free case.

III.4.6. Data Analysis

Data Analysis proceeds in the same manner described in the Fault Free case.

In this fault injection case, the User may wish to query some additional information concerning the behavior of the workload as influenced by the fault.

Some of the information that can be provided by the Paradise data analysis tools is:

- List the APCs that failed.
- List the PNodes that crashed.
- List the timelines that were altered.
- List the event records associated with Error Detection mechanisms in the workload.
- List the event records associated with Error Recovery mechanisms in the workload.

- How was the corrupted memory changed? Was a data word altered? Was an instruction word changed?
- If the fault was detected, what was the latency time? Where was the fault detected?

The last situation is specified in the Experiment Description. The command *data_analyze(fault_latency)* specifies that that the appropriate tool(s) be applied to the data such that the desired information is extracted.

III.4.7. Data Presentation

Data Presentation proceeds in the same manner described in the Fault Free case. The workload profile is displayed in bargraph form.

In this fault injection case, the profile display can provide additional types of visual information, such as:

- Highlight the time and location of the fault injection.
- Highlight the APC where the fault is detected
- Highlight failed APCs and PNodes

the User may wish to query some additional information concerning the behavior of the workload as influenced by the fault.

Appendix IV
The Attachments

IV.1. APC Monitoring Attachment

The following code is the MA that was used for APCs. The DCA that was used is shown in a following section.

```
Ret_Info Buffer[30], *Ret_Buffer[30];
int Buffer_count = 0;

P_SENSOR_FLUSH ()
{
    if (Buffer_count != 0)
        pslcollgenSENSOR (NULL, Ret_Buffer, Buffer_count);
}

Send_buff_off ()
{
    pslcollgenSENSOR (NULL, Ret_Buffer, Qsize);
}

int P_SENSOR (sen_type, stat_ID)
int sen_type, stat_ID;
{
    DATE Cur_time;
    int the_size;

    Ret_Buffer[Buffer_count] = &Buffer[Buffer_count];
    GetDATE (&Cur_time);
    CopyDATE (&Cur_time, &Buffer[Buffer_count].Timestamp);
    Buffer[Buffer_count].Sentype = sen_type;
    Buffer[Buffer_count].StatID = stat_ID;
    if (Buffer_count >= (Qsize - 1))
    {
        Buffer_count = 0;
        Send_buff_off ();
    } else
        Buffer_count++;
}
```

IV.2. APC Data Collection Attachment

This section shows the *coll.typedef* and *coll.mgr* files that were used to build the DCA as a Cronus manager. This manager is similar to the one described in appendix section 1 except it accepts a list of sensors (vs. single sensors).

Following is the *coll.typedef* file:

```
type Collector = 627
        abbrev is coll
        subtype of Object;
```

```
cantype RET_INFO
        representation is Ret_Info:
        record
                Timestamp:      EDATE;
                Sentype:        U16I;
                StatID:         U16I;
                Junk:           ASC;
        end RET_INFO;

generic operation SENSOR
        (Info:  array of RET_INFO);

generic operation RETRIEVE ()
        returns
        (AnEntry: array of RET_INFO);

end type Collector;
```

Following is the *coll.mgr* file:

```
manager "The Collector Manager"
        abbrev is coll

type coll
        variable representation is SEN_ENTRY
        coll implements all from coll
        obj implements rest
```

Appendix V
List of Abbreviations

Abbrev.	Description
APC	*Application Client:* A Cronus client running on a PNODE that is associated with the execution of a parallel/distributed application. Each APC has four attachments: WCA, MA, FIA, and DCA.
DCA	*Data Collection Attachment:* A APC attachment that includes commands to allow retrieval of sensor data from the local file without interrupting the ongoing monitoring process.
DCC	*Data Collection Controller:* Implements commands to collect information from or get information on the LRF.
DCM	*Data Collection Manager:* A PEM component that supports event and LRF abstractions.
EDC	*Experiment Definition Compiler:* Given a high level experiment description, a fault description, and a workload description, the EDC compiles an experiment script.
FIA	*Fault Injection Attachment:* Implements mechanisms to perform fault injection in an APC.
FIC	*Fault Injection Controller:* Communicates with the FIA of an APC to dynamically alter the fault profile characteristics.
FIG	*Fault Instance Generator:* Takes the description of a fault class and produces a number of fault instances to be applied to a particular workload.
FIM	*Fault Injection Manager:* A PEM component that deals with fault abstractions.
IP	*Integration Platform:* A Paradise database that contains the monitor profile, fault profile, and runtime data of APCs.
LRF	*Local Repository File:* APC sensor events are recorded in this file. There is one LRF for each Paradise APC in a workload.
MA	*Monitoring Attachment:* An APC attachment that controls the monitoring activities in Paradise. It detects, filters, and stores information from sensors in the APC code. Sensor data is stored into a local queue which empties into a LRF.
MC	*Monitoring Controller:* Implements commands to control the MA of the APC such as setting queue size, and turning sensors on or off.
MM	*Monitoring Manager:* A PEM component that supports monitoring profile, sensor, and event abstractions.
PEM	*Paradise Experiment Manager:* A component of the PWS that is associated with the run-time control of an experiment. The PEM itself consists of four components : WM, MM, FIM, and DCM.

Table V-1: Description of Abbreviations.

Abbrev.	Description
PMC	*Paradise Monitor and Controller:* A special process on each PNODE that communicates with APCs on its node and also with the PWS that is in control. Has four components : WC, MC, FIC, and DCC.
PNODE	*Paradise Node:* A host that is part of the Paradise system. Each PNODE runs the Cronus operating system.
POPROC	*Paradise Postprocessor:* Takes code that has been processed by the Cronus compiler and extracts the fault profile which is then stored in the IP.
PPROC	*Paradise Preprocessor:* Takes User code as input and inserts additional statements so code can be run as a Paradise APC. Also extracts development time data and monitor profile and stores in IP.
PWS	*Paradise Workstation:* A PNODE of Paradise that runs a special application with graphic capabilities that is the User interface. The PWS is in control of the Paradise system.
REC	*Runtime Experiment Controller:* Takes as input an experiment script that has been generated by the *edc* and interprets it.
SAPC	*Startup Application Client:* An APC that creates other clients which constitute a particular workload.
WC	*Workload Controller:* Part of the PMC that implements functions to create and synchronize APCs in the local PNODE.
WCA	*Workload Control Attachment:* As one of the APC attachments, the WCA implements a set of functions to create, control and synchronize distributed workloads.
WM	*Workload Manager:* A component of PEM that communicates with a set of clients called the workload. Communication is done via a Paradise link.

Table V-1, concluded

References

[Gettys 87] Jim Gettys, Ron Newman, Robert W. Scheifler.
xlib - C Language X Interface, Protocol Version 11
Massachusetts Institute of Technology, 1987.

[Schantz 85] Richard E. Schantz, Robert H. Thomas.
CRONUS, A Distributed Operating System: Functional Definition and System Concept.
Technical Report 5879, BBN Laboratories Incorporated, 1985.

[Schantz 86] R. Schantz, K. Schroder, M. Barrow, G. Bono, M. Dean, R. Gurwitz, K. Lam, K. Lebowitz, S. Lipson, P. Neves, R. Sands.
CRONUS, A Distributed Operating System: Cronus DOS Implementation, Final Report.
Technical Report 6183, BBN Laboratories Incorporated, March, 1986.

[Swick 87] Ralph R. Swick, Terry Weissman.
X Toolkit Widgets - C Language X Interface, X Window System
Massachusetts Institute of Technology, 1987.

Index

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.