NASA Contractor Report 4340

Validation Environment for AIPS/ALS: Implementation and Results

Zary Segall, Daniel Siewiorek,
Eddie Caplan, Alan Chung,
Edward Czeck, and
Dalibor Vrsalovic

GRANT NAGI-190
NOVEMBER 1990

NASA Contractor Report 4340

Validation Environment for AIPS/ALS: Implementation and Results

Zary Segall, Daniel Siewiorek,
Eddie Caplan, Alan Chung,
Edward Czeck, and
Dalibor Vrsalovic
Carnegie-Mellon University
Pittsburgh, Pennsylvania

Prepared for
Langley Research Center
under Grant NAG1-190

National Aeronautics and
Space Administration
Office of Management
Scientific and Technical
Information Division

1990

Contents

1 **Executive Summary** 1

2 **Validation** 3

2.1 System Specification 3

2.2 Validation Methodologies 4

3 **Implementation of a Validation Environment on AIPS** 8

3.1 Architecture of the FIAT Validation Environment 8

3.2 Architecture of the PIE Validation Environment 10

3.2.1 PIESCOPE 12

3.2.2 PIEMON 13

3.2.3 Setting Up an Experiment in PIE 14

3.3 Architecture of the AIPS Validation Environment 20

3.4 PIE Software 23

3.4.1 PIE Software on the AIPS FTP 23

3.4.2 PIE Software on the AIPS Host 25

3.4.3 Workstation Software 26

4 **Experiment Design and Results** 28

4.1 LoopTest 28

4.2 MemTest 30

4.3 ActTest 33

5 Summary 38

References 40

A PIEscope Figures of the Experiments 42

A.1 Low Priority LoopTest 42

A.2 High Priority LoopTest 45

A.3 Low Priority MemTest 49

A.4 High Priority MemTest 53

A.5 Low Priority ActTest 57

A.6 High Priority ActTest 65

1 Executive Summary

This is the final report of work done under contract NAG-1-190. This document presents the work performed in porting the FIAT and PIE[1] validation tools, developed at Carnegie Mellon University, to the AIPS[1] system in the context of the ALS application, as well as an initial *fault-free validation* of the available AIPS system. The PIE components implemented on AIPS provide the monitoring mechanisms required for validation. These mechanisms represent a substantial portion of the FIAT system. Moreover, these are required for the implementation of the FIAT environment on AIPS. Using these components an initial *fault-free validation* of the AIPS system was performed.

This report describes the implementation of the FIAT/PIE system, configured for *fault-free validation* of the AIPS fault tolerant computer system. The PIE components have been modified to support the Ada language. A special purpose AIPS/Ada runtime monitoring and data collection has been implemented. A number of initial Ada programs running on the PIE/AIPS system have been implemented. The instrumentation of the Ada programs was accomplished automatically inside the PIE programming environment. PIE's on-line graphical views show vividly and accurately the performance characteristics of Ada programs, AIPS kernel and the application's interaction with the AIPS kernel. The data collection mechanisms were written in a high-level language, Ada, and provide a high degree of flexibility for implementation under various system conditions.

Beyond the demonstration of the success of the implementation of the FIAT/PIE, we have characterized some of the critical components of the AIPS/Ada kernel. We paid special emphasis to the performance of Ada task management functions, communication, synchronization, and memory management. Given the real-time ALS application requirements we stressed the need for performance predictability. The collected data have pointed out a number of anomalies. Some of these anomalies are due to the Verdix implementation of the Ada runtime library, while others may be related to the implementation of the monitoring tools. Further work is needed to calibrate and fine tune these tools.

The results of this work point in the following directions:

1. The PIE systems provides an automated fault-free validation environment for AIPS. Also, we demonstrated the value of PIE as an architecture independent performance evaluation and program development tool.

2. PIE functionality is required for the FIAT system for *fault-injection based validation* of

[1]FIAT and PIE are described in Section 3.

the AIPS for ALS.

3. With the FIAT/PIE tools in place substantial insights of the performance intricacies of AIPS are available.

4. Initial *fault-free validation* of the AIPS shows a number of anomalies in the critical areas of task management, memory management, communication, synchronization, and runtime overhead. Additional work is needed to eliminate or account for any anomalies in the monitoring itself. Moreover, the discovery of these anomalies demonstrate the benefits of the fault-free validation methodology applied to a system under development.

5. Once fully explained, those anomalies could either become user considerations or could be fixed in future versions of the AIPS. In either case, the system will become substantially more predictable and hence suitable for the real-time requirements of ALS.

6. Due to the concurrent development of AIPS and FIAT/PIE on AIPS, the FIAT/PIE system have not been yet fully exploited for AIPS validation. (It is difficult to validate a system under development – a moving target.)

7. There is a distinct opportunity with the PIE environment on AIPS and also the need for a fault-free validation to be performed on the final version of AIPS. The validation suite for the final version of AIPS must be biased towards the critical (or unknown) portions of the system to avoid uncovering "known" limitations such as the poor memory allocation performance.

8. After the completion of the fault-free validation, we are strongly suggesting the critical need for Fault Injection based Validation of AIPS. Of special concern are the common mode failures (most of which occur in the software component of the system), communication protocols and multiple single mode failures. For this purpose an opportunity exists in using the proven methodology of FIAT/PIE and the availability of these tools on AIPS.

This report is organized as follows. Section 2 motivates the need for a validation environment based on a validation methodology which requires monitoring the system, as well as injecting faults into the system. Section 3 describes the FIAT and PIE environments developed at CMU, and presents the architecture and implementation of the FIAT/PIE environment on the AIPS/ALS system. Section 4 presents experiments and results of the fault-free validation experiments conducted on AIPS. Section 5 summarizes the report. Several appendices contain data results with annotations highlighting relevant abnormalities.

2 Validation

Validation is the process of substantiating, through demonstration, that a given system meets its specification[2, 3]. For highly dependable systems, the specifications contain extreme reliability requirements which necessitate the ability to function under faulty conditions. To demonstrate or validate the system operation, prediction methods must be used to determine the system "operation point" before the system is committed to use. Methods of determining the "operation point", or the nominal behavior, include simulation, modeling, and analysis. Complimentary to these methods are experimental methods, such as program instrumentation and fault injection, which are well suited for areas in which modeling and analysis fail to capture the needed detail.

2.1 System Specification

System specifications can be divided into two domains, with the validation effort directed to demonstrate that both are fulfilled[4]. The first domain includes functionality and the second is the bounds within which correct functioning must occur. Functionality is by far the easier of the two domains to validate; metrics, such as throughput and real-time deadlines, are readily defined. The bounds of correct functioning typically are associated with dependable computing and include metrics such as reliability, maintainability, and fault tolerance.

A functionality requirement includes performance measures where performance is measured in functions per unit time or in the time needed to complete a specific task[5]. The notion of performance exists throughout the digital design hierarchy, from the circuit level (switching times), to the system level (application execution time). With this definition and a validation methodology, a performance evaluation matrix can be created, as depicted in Table 1. The vertical axis is the design hierarchy, while along the horizontal axis are definitions or characterizations of performance. Elements in the matrix are not singular and evaluation measures can overlap. The area of concentration for the measures is dependent upon the needs of the validation in measuring the "operating point" and also the level of instrumentation[2] available in the validation environment.

A typical reliability requirement for a life critical application is 10^{-10} failures per hour. The basis for this failure rate can be justified through the following life-cycle model. Assume a 30 year life, with 8 hours operation per day; this yields approximately 100,000 (10^5) operational

[2]Instrumentation, as used in the FIAT and PIE environment is the process of adding measuring devices or software "hooks" into a workload to monitor its execution and report the occurrence of events.

	Behavior	Throughput	Utilization	Delay
Application	Correct Function in Integrated Environment	Application Task Times (e.g. flight control)	Idle Time	Variation Caused by Shared Data, Increased Load
Executive, Operating System	Correct Operation of Scheduler, Dispatcher, etc.	Operating System Primitive Times	Operating System Frequency of Use	Variation Caused by Hardware and Data Contention
Instruction Set, Hardware	Correct Operation of Interrupts, etc.	Instruction, and Resource Times	Hardware Frequency of Use	Variation Caused by Hardware Contention

Table 1: Performance Evaluation Matrix

hours per unit. If 100,000 (10^5) copies are produced and one failure is acceptable over the life of all copies, then the failure rate must be less than 10^{-10} failures per hour. This translates to one failure per 1 million years per unit or several orders of magnitude greater than the reliability of todays systems. This stringent reliability requirement yields two observations. The first is that non-redundant systems are at least six orders of magnitude less reliable than the goal, necessitating the use of redundancy and its ability to function correctly with faults present. The second is that life testing (monitoring) for confirmation of reliability is impossible, necessitating the need for accelerated testing.

In characterizing and testing any system, it is necessary to apply a sample of the input space into the unit under test. The approach for a fault-tolerant system is the same: in a fault-tolerant system part of the valid input space is faults. Hence faults must be injected to test the system, as are data values. Thus the goal of fault injection is to emulate the behavior of a system with faults present. Once a fault injection methodology is developed, the problem becomes one of testing; namely what set of faults is needed to test and validate the fault tolerant aspects of the system.

2.2 Validation Methodologies

Much work has been done in validation methodologies, especially in aerospace and other life-critical applications. These methodologies include formal proofs, analyses, and tests to assure the system meets its specifications. Although there is no commonly accepted validation methodology, a generalized methodology may be extracted from procedures presented in the literature[4, 6, 7, 8, 9, 10, 11, 12]. The approach is to build confidence in the system by a thorough and systematic methodology of proofs, analyses, and tests. Proofs are formal arguments supported by deductive inferences. Analyses employs models of the system, and testing uses

Development Level	Abstract Design Proofs	$\Longleftrightarrow$ Analyses	Concrete Tests
Architecture	Prove Architecture Against Requirements	Reliability and Error Rate Markov Models	Design Reviews Simulations
Implementation	Prove Implementation Against Architecture	Fault Tree Analysis	Simulations and Emulation of Hardware
Realization	Prove Realization Against Implementation	Failure Modes and Effects Analysis	Support Assumptions from Analyses, Fault Injection

Table 2: Validation Activities throughout the Design Space

statistical inference. These three methods are complementary: proofs and analyses use abstract models of the systems; testing uses the actual system to substantiate the models and results generated in the analyses.

These three processes (proofs, analyses, and testing) are applied throughout the system development as depicted in Table 2. During the architecture development, proofs are generated which specify the conditions necessary to achieve the requirements. Analyses of the architecture include reliability and error rate Markov models, while the testing comprises activities such as high level simulations and design reviews. At the implementation level, the conditions required in the architecture proofs are verified, leading to more conditions for the realization. The analyses includes further refinement of the Markov models developed in the architecture analyses, and in-depth analyses such as Fault-Tree generation. Testing begins to involve concrete methods such as simulation and emulation of the design. In the final level realization, proofs of the hardware and software structure are continued from the implementation level. Analyses includes exhaustive Failure Modes and Effects Analysis, refinement of Fault-Tree analysis, and the inclusion of specific failure rates into the reliability and error rate analyses. Testing at the realization stage measures the assumptions and requirements used in the proofs and analyses. The assumptions involve error rates, fault latency, and coverage, as well as concrete measurements such as throughput, utilization, and error recovery time.

Two possible methods exist for validation of highly reliable systems under faulted conditions. The first method, life testing, monitors actual running systems awaiting the natural occurrence of faults. The behavior of the system, when faults occur, can then be analyzed and used to support validation assumptions or conditions. The second method, fault injection, induces faults into the system and analyzes behavior under these conditions. Life testing offers realism, but due to the current level of component reliability, faults can be expected at a rate of one in 10^3 hours per system. This failure rate is prohibitively slow for the completeness required in

thorough testing. Fault injection speeds up the rate at which synthetic faults occur. The use of synthetic faults is necessary given the large number of fault types, fault locations, and times of occurrence. For example, a small board consisting of 50 packages each with 20 pins, has a fault space of 1000 pin-level faults without considering any time dependencies. Additionally, software faults must be considered as the majority of system complexity moves into software. The software-fault space is also large – consider the amount of code present in even the smallest of operating systems.

Underlying any methodology, there must be a set of guiding philosophies. Over the last decade, CMU has dedicated over 100 man-years of effort in the design, construction, and validation of multiprocessor systems. A partial list of the experimental guidelines developed during the last decade include:

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

With a fault-tolerant, ultra-reliable system other problems arise which make the validation task increasingly difficult. Some of these problems are:

- Life testing is inappropriate, due to large mean-time-to-failure of the system.
- System design complexity makes it difficult to perform failure effect analysis, instrument and measure all relevant parameters, and use exhaustive testing approaches, since there are a large number of states and failure modes possible.
- Large scale integration makes access to control and observation points difficult as well as determining a confidence level for fault coverage.

NASA held several workshops to determine validation procedures. One [11] in particular produced a detailed outline of a validation procedure. The procedure is based on a building block approach. Primitive system activities are characterized first. Once these activities are understood, complex experiments involving the interaction of primitive activities, as well as complex activities built from the basic primitives, may be conducted. This orderly progression insures uniform, thorough coverage and maximizes the ability to locate the cause of unexpected phenomena. The steps in the methodology include:

1. Initial checkout and diagnostics.
2. Programmer's manual validation.
3. Executive routine validation.
4. Multiprocessor interconnect validation.
5. Multiprocessor executive routine validation.
6. Application program verification and performance baseline measurements.
7. Simulation of inaccessible physical failures.
8. Single processor fault injection.
9. Multiprocessor fault injection.
10. Single processor executive failure response characterization.
11. Multiprocessor system executive failure response characterization.
12. Application program verification on multiprocessor system.
13. Multiple application program verification on multiprocessor system.

The first six tasks in the list validate the fault-free baseline functions of the system, items seven through eleven characterize the fault-handling capabilities of the processors, and the last two validate the total integrated environment of the system. This report presents fault-free baseline performance measurements. In general, the methodology follows two parts, a fault-free validation followed by a fault handling part. The importance of the fault-free validation has been shown in previous reports[13, 14], and the methodology presented in this report follows the same line.

3 Implementation of a Validation Environment on AIPS

The validation process is difficult in that it requires observability and controllability of the system under investigation. The observations include measure of performance and behavior while the controllability includes adjusting the workload, as well as injecting faults. To aid in the validation process a validation environment is implemented. The environment combines work done on FIAT and PIE at CMU. This section describes both FIAT and PIE architectures, the combined validation architecture for AIPS, and its implementation[1].

3.1 Architecture of the FIAT Validation Environment

FIAT, Fault Injection-based Automated Testing, is a prototype experimental environment used to explore validation methodologies for fault-tolerant systems. The goals of the FIAT project are to develop the requirements for an automated software-implemented fault injection environment and to gain an understanding of software-implemented fault injection methodologies.

Validation requires the ability to monitor the system under test, the ability to control the system to induce faults and other operating conditions, as well as the ability to repeat tests to identify the source of system deficiencies. These requirements imply a test environment capable of automatically inducing faults and monitoring system behavior. The underlying methodology guiding the FIAT validation process is as follows:

1. Specify a system architecture, including hardware and software. FIAT allows the user to specify the architecture through a combination of emulation, where actual software tasks or hardware components are represented by a software task emulating the actual behavior, or through the use of FIAT software on the actual hardware/software structure.

2. Profile the fault-free behavior of the system to determine a nominal "operation point". Profiling gives general information regarding the execution of the systems, such as task execution order, execution time, memory usage, and possibly bounds for data variables.

3. Select a set of faults and profile the system behavior with these faults present. The fault set is chosen to represent actual faults which the user is interested in studying. These can represent faults occurring in either the hardware or the software of the system and are used to gauge the effect of actual faults on the system.

4. Analyze experimental data and use the results to support validation requirements and other experiment goals. Measures, such as fault latency, error recovery probability, and the like, can be extracted through experimental analyses and used to support validation

requirements such as parameters for Markov models. Given the goals and the desired validation methodology, FIAT was designed to support the following functions.

Architecture Development: The target architecture is divided into portions which are to be emulated and portions which will use the implementation employed by FIAT. This allows the user to design and evaluate a system without customized hardware, software, or a large initial effort. The hardware, software, and communications structure of FIAT is general, so it may emulate a variety of architectures or be applied to an actual implementation. FIAT is oriented towards a message-based, replicated structure, where messages are passed via the FIAT communication channels and the replicated structure is emulated by FIAT hardware and software tasks.

Fault Injection: The goal is to insert data representative of "actual" faults into the system to gain an understanding of system operation under abnormal conditions. The injected faults may be "actual" faults under study or the manifestation of faults - errors. FIAT, through software-implemented fault injection, induces faults or the appearance of faults in a system by modification of the software image or through the execution of special software designed to emulate faults. Software-implemented fault injection was selected for the following reasons:

1. Systems to be validated have a substantial software component. Software fault injection allows penetration into the software portion of the system as well as exploring the interaction of software with hardware.
2. Software-implemented fault injection is less expensive, in terms of time and effort, than hardware-implemented fault injection.
3. Software-implemented fault injection is functionally complementary to hardware-implemented fault injection and does not exclude it.
4. There is a need for a testing methodology to validate software-implemented fault tolerant strategies.

Software-implemented fault injection has its limitations, mainly in its inability to force low-level errors, such as gate output stuck-at faults. However, designers are interested in the behavior of the whole system (hardware and software) rather than the manifestation of individual faults. Furthermore, a large amount of the hardware functionality is visible through software.

Automation and Unity: The quality of experimentation is a function of the fidelity of the fault injection method and of the capability of the system to inject (test) as many faults as possible per unit time. Automation includes both experiment development time and

experiment runtime processes. To be effective, the various components of the system (e.g. workloads, fault classes, experiments and data analysis) must be integrated under one comprehensive environment, which supports the process of preparation, debugging, runtime control, and data analysis.

The FIAT methodology, like the validation methodologies presented earlier, include profiling of the fault-free behavior followed by the fault behavior of the system. The application of the FIAT validation methodology on AIPS includes both parts. The fault-free validation process was initiated by the integration of the PIE environment on AIPS. The integration of PIE on AIPS allows the characterizing the fault-free behavior of the system as well as instrumenting the system for future fault injection work. Within the FIAT methodology, PIE provides the Architecture Development and the Automation and Unity support and PIE is especially useful for the first two steps in the validation methodology.

3.2 Architecture of the PIE Validation Environment

The need for a validation environment stems from the complexity of today's systems and the difficulties which arise in the application of the validation methodology. Moreover, the ability to predict and model the behavior of a system, especially a dependable real-time system such as AIPS, first requires an understanding of the behavior. Two parts to the understanding are typically needed: the first is the understanding used in the design and implementation of the system, while the second is the understanding and modeling of actual implementation to support the design assumptions. The second part requires the observation of the system in its actual environment.

The process of observing a system in its actual environment is the goal of the PIE, Programming and Instrumentation Environment, project at CMU. PIE is a powerful, general purpose tool which supports the monitoring and visualization of programs during execution. This report describes the PIE system as configured for monitoring the AIPS fault-tolerant computer system[1].

The PIE system, depicted in Figure 1, consists of a set of integrated tools for automated performance characterization of a real-time, parallel/distributed, fault-tolerant system. Central to this environment is the concept of performance degradation prevention, detection, and avoidance. Performance degradation prevention is the process of predicting, before completion of the implementation process, the performance of a parallel algorithm on a specific parallel architecture. Performance degradation detection are the set of techniques applied after the cod-

ing process. Performance degradation avoidance is a run time process consisting of dynamically restructuring the application or the system in the presence of predicted or detected performance degradations.

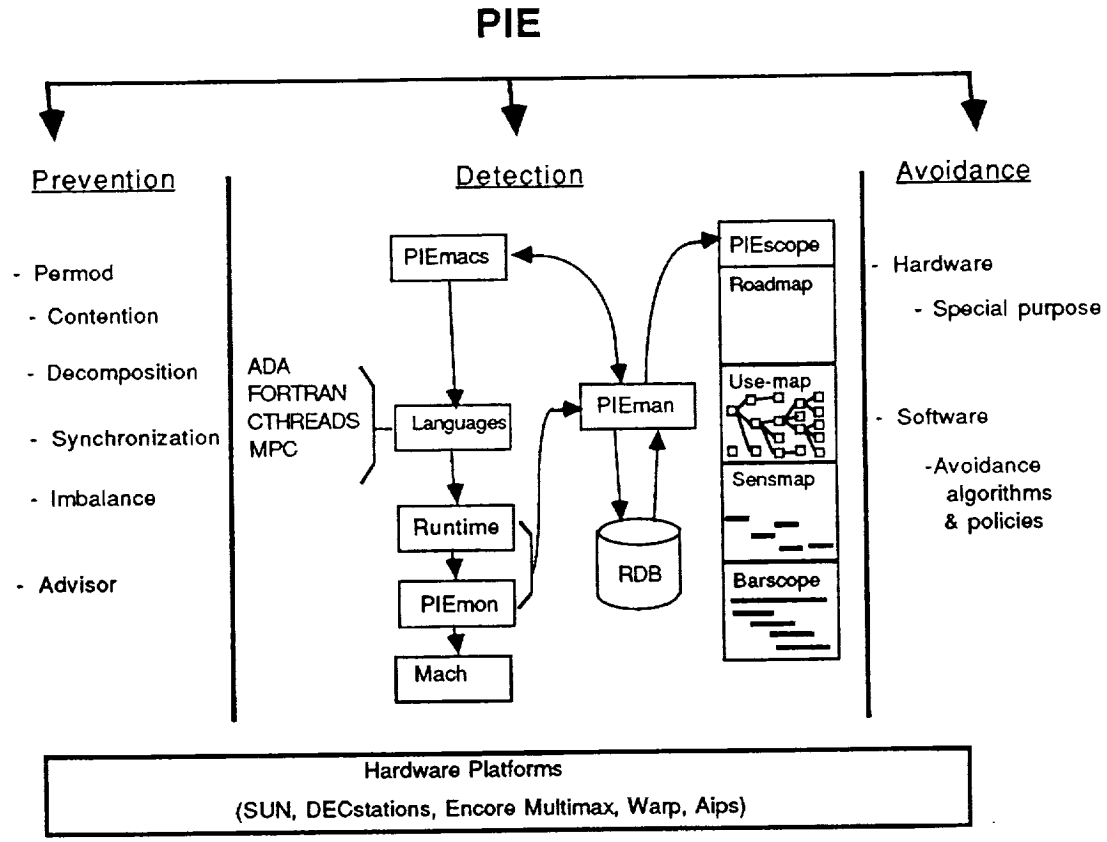


Figure 1: Organization of the PIE System

In this context we will discuss mainly the performance degradation detection, as being the process of fault-free validation. The PIE environment consists of several subsystems for assisting a programmer in developing computations and observing their run-time behavior. These systems are:

PIEmacs: an editor with special features for inserting monitoring hooks into a computation,

PIEmon: a performance and correctness monitoring facility including the AIPS context-switch monitor,

PIEman: a database manager which correlates the text of a computation (development-time information) with information about the execution of the computation (run-time information), and

PIEscope: a graphics package for presenting the run-time and development information to the programmer.

In PIE, a programmer edits a computation using PIEmacs, an extension of Gnu-emacs[3]. PIEmacs automatically marks program constructs for affixing sensors later using a special compiler preprocessor. In addition to the automatically marked constructs, the programmer is permitted to mark a computation in places of special interest. The development-time information which PIEmacs generates as it marks a computation is delivered to PIEman, the PIE database manager. PIEman builds a database from this program development information and later merges this compile-time information with data retrieved at run-time. The run-time information is retrieved by PIEmon, the PIE performance monitor. The information is presented to the programmer via PIEscope, a graphics package which displays several views of the structure of a computation as well as how the structures were executed.

3.2.1 PIESCOPE

This paragraph describes PIEscope, the current graphical user interface for the PIE system. PIEscope provides graphical views of the development and execution of a user's program, using the X-Windows, Version 10, windowing system. PIEscope provides three development-time views and three execution-time views. The development views are:

1. **roadmap:** a tree-like display of the definition structure of the user's Program.
2. **use roadmap:** a tree-like display of the instantiation and static invocation structure of the user's program.
3. **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 firing during the program execution.

The execution-time views are:

1. **barscope:** a bar graph of the execution of the user's program.
2. **cpu barscope:** a bar graph depicting processor utilization of the user's programs.
3. **animation tree:** a tree-like display which replays the dynamic invocations (and destruction) of the structures in the user's program.
4. **max-animation tree:** 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.

[3] Gnu-emacs is a screen-oriented text editor supported by the *Free Software Foundation*.

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

3.2.2 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 designates a specific logical point (state) in a computation's control flow and includes the time when that state was reached. 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 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 data are 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 system which appends an event to the event record of the computation. After an execution terminates, PIE selects and filters the events in the event record using a relational data base, constructed at development time, containing the static structures of a program as well the semantic and temporal relations between them. The structures contain sensor marks so the 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.

3.2.3 Setting Up an Experiment in PIE

Because PIE is the vehicle for fault-free validation, it is important to be comfortable with frequent references to the environment later in this report. The following brief example of using PIE is condensed from paper in IEEE Computer[15] and should be read if greater familiarity with PIE is desirable.

A Problem Application Assume a user desiring to fault-free validate a matrix multiply application on a 16-processor shared-bus machine. The basic structure of the application consists of passing well-partitioned parts of two matrices to several child tasks. Each process first examines the parts of matrices it is passed and decides whether they are small enough to operate on without partitioning them further and passing them on to its own child process. After making the decision, each process iterates through its matrix parts, multiplying each pair of row and column and writing out the result.

Figure 2 depicts parts of the text of the application via three windows of PIEmacs. The top window in Figure 2 shows a part of the definition of the application's multiplier procedure, `multproc`. It includes a variable declaration of the type multiply, an instance of an *activity* or `act`, as shown in the middle window. Activities or tasks are process-like units of parallel work which, when spawned from the same application, are able to share and operate on global memory. Notice that `multiply` contains a call to `multproc`. `Multproc` implements the basic matrix partitioning and multiplying functions described above. After the value of a matrix element is calculated, it is written out using `put`, shown in the lowest window. It is an instance of an software object called `opr`, used to operate on global memory. Entities of this type may be shared by several activities. The only feature of `put` that ought to be understood here is the `sync`, a *synchronization* function that enforces mutual exclusion on global operations. Here, `sync` ensures that only one result may be written back to global memory at a time.

Figure 3 is an automatically generated PIEscope *roadmap* visualization of the application's principal constructs. The roadmap view is the first step in PIE for bridging the roadmap (Figure 3) to a corresponding textual entry (Figure 2). When a box is touch-selected by a mouse, as is shown by the enlarged border surrounding the box labeled [c] `multproc`, the PIEmacs window automatically moves its cursor to the head of the corresponding textual construct, in this case, a call to the `multproc` procedure as shown in Figure 2.

Having visualized the structures of the application, it is time to gather performance information. PIE can generate performance views such as histograms, but these are ancillary to a more informative format which will be shown shortly. When an application, with a potential

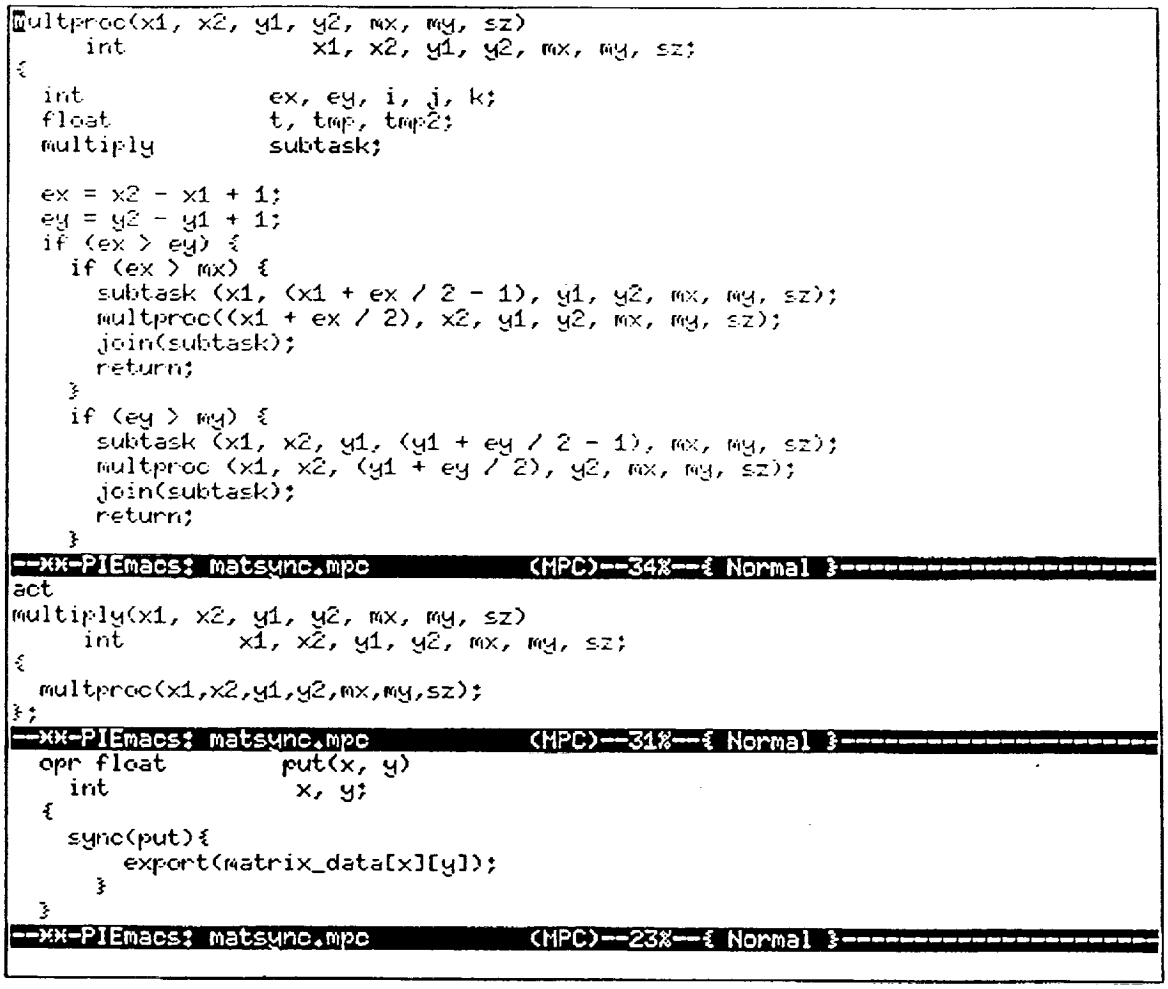


Figure 2: Part of a Matrix Multiply Program Text

parallelism of four, runs only two or three tasks simultaneously, the programmer knows that they should investigate any program construct that might force a multiplier to wait, namely the sync (just discussed) and the join (an example is shown in the top window in Figure 2) which a multiplier executes when it wishes to join its children. To get this information with PIE is simple. Figure 4 shows a number of darkened boxes, [A] multiply, [S] Sync and several cases of [J] Join. The [A] multiply represents the multiplier tasks and [S] Sync is the synchronization function in the put operation discussed earlier. Each [J] Join represents an instance of a join function. The darkening of these boxes indicates that the programmer, using a mouse-click, has selected them to be automatically observed during execution.

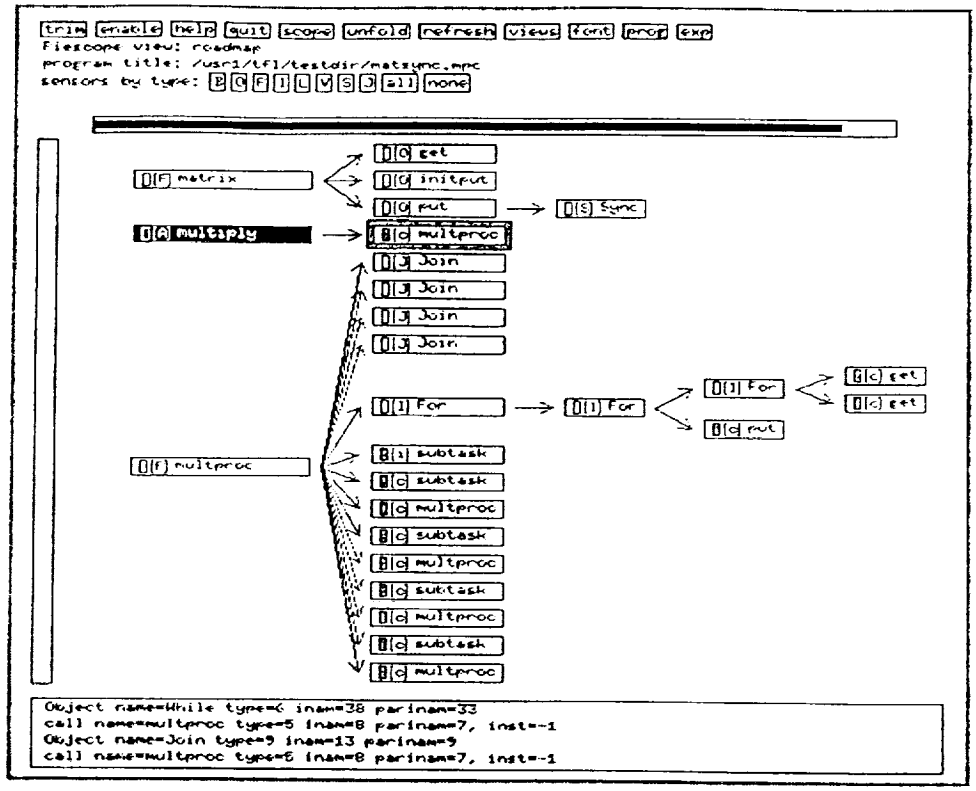


Figure 3: Part of Visual Representation of Computation

Examining the Results PIE's foundations for instrumenting computations includes software event sensors, hardware event sensors, and hybrid event sensors. Currently, however, computations in PIE are instrumented using only software sensors. During run-time PIE ensures that when a selected construct executes, important information is automatically collected about the construct. An example of PIE's principal formats for visualizing performance data is shown in the upper two views of Figure 5. The top view of Figure 5 is called the execution barscope view. Time is measured in seconds (with micro-second resolution) on the horizontal while the tasks of the computation are ordered on the vertical. Although it is possible to show any part of a computation, this particular view shows only the tail end of the execution from about 2.6 to 2.8 seconds.

The execution of each task is depicted by the concatenation and occasional "overlap" of several textured rectangles, each representing a particular episode in the task's history. A rectangle is "in front of" another rectangle if the entity represented by the rectangle in the forefront is contained within the entity represented by the rectangle behind it. In the top view of Figure 5, for example, waits due to a sync show up as dotted rectangles alternating with

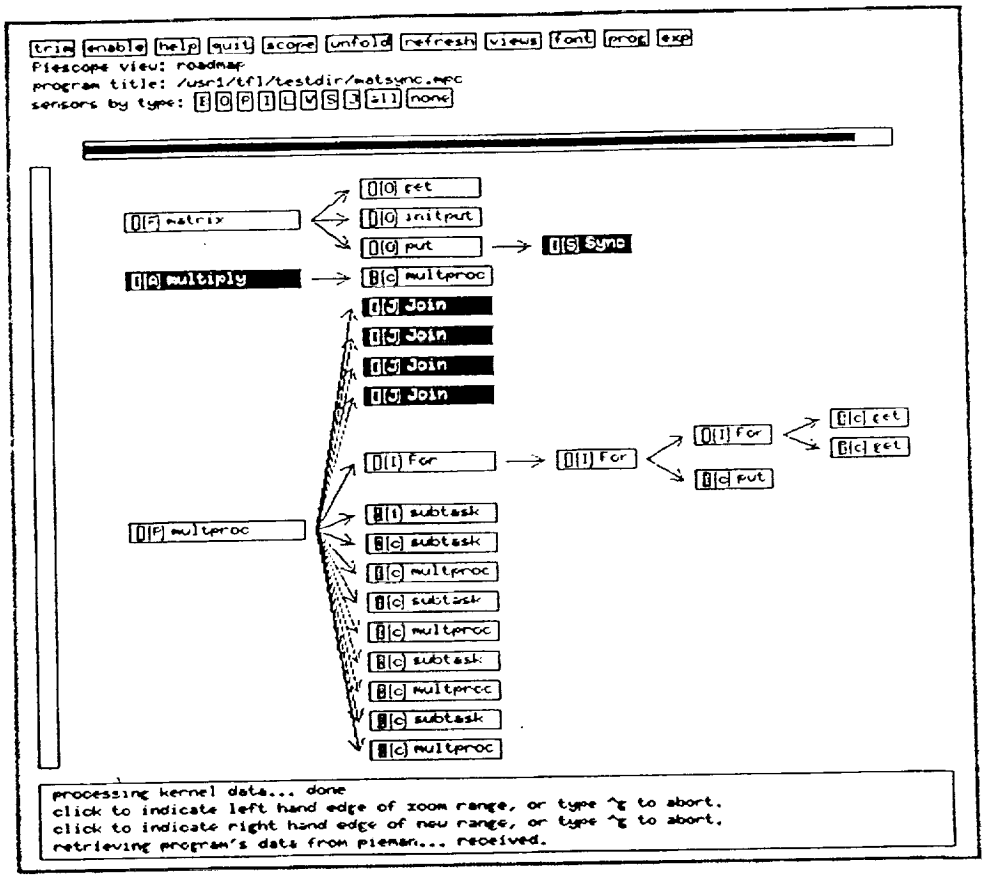
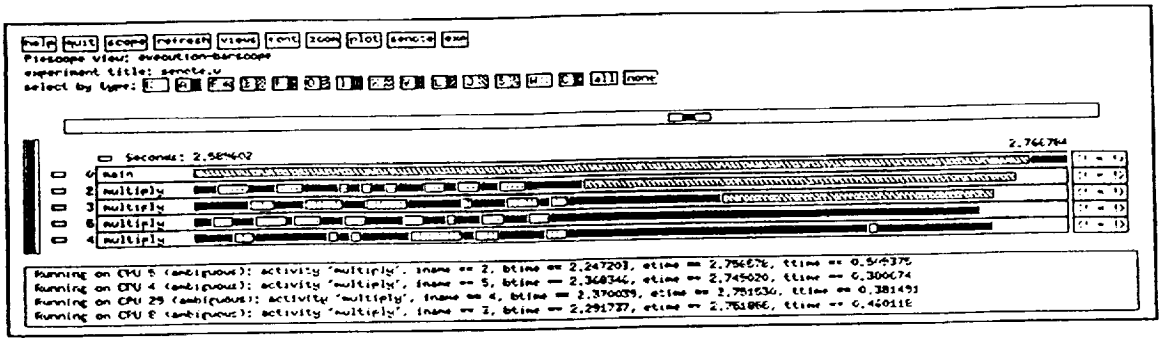


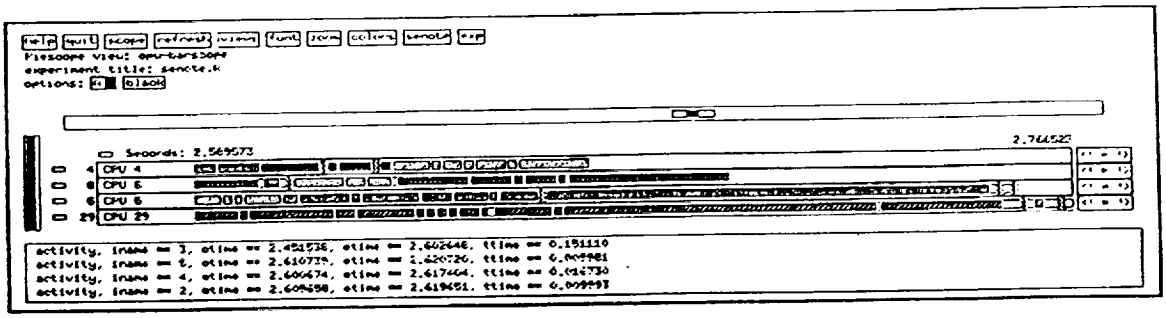
Figure 4: Using the Visual Representation to Enable Sensors

several dark rectangles. The dotted rectangles are actually in front of of a single dark rectangle representing the generic body of the task. The slashed rectangles at the end of tasks zero, two and three are instances of parents waiting to join a child.

The middle view of Figure 5 is called the *cpu barscope* view. It shows the task-to-processor assignment of the computation during the same execution period shown in the top execution barscope view. Time is along the horizontal axis and the machine's processors are ordered on the vertical. Opposite each cpu are alternating sets of textured rectangles representing identifiable tasks. White rectangles are periods when none of the computation's tasks are running on the associated cpu. When any rectangle in either barscope view is selected by the mouse, the cursor in the PIEmacs window is automatically moved to the head of corresponding construct in the program text. In Figure 5 for example, a `sync-wait` rectangle has been clicked on in the execution barscope. The semantic gap is now bridged allowing the programmer to analyze the computation's performance using data automatically projected onto the computation's structures. In addition, the visualization helps the functional gap: the gap between the extent

to which performance monitoring merely reports how computations behave and the extent to which it helps guide users to the source of their problems.





```
  opr float         put(x, y)
    int              x, y;
  {
    sync(put){
        export(matrix_data[x][y]);
      }
  }

  {
  }
} a(SIZE, SIZE)[3];


act
multiply(x1, x2, y1, y2, mx, my, sz)
     int         x1, x2, y1, y2, mx, my, sz;
{
  multproc(x1,x2,y1,y2,mx,my,sz);
};
--xx-PIEmacs: matsync.mpc          (MPC)--20%--( Normal )----------
```

Figure 5: TOP: **Part of a Parallel Execution of the Computation on 16-Processor Machine** MIDDLE: **Corresponding Thread to cpu Assignments** BOTTOM: **A PIE Editor Window**

3.3 Architecture of the AIPS Validation Environment

The implementation of PIE on AIPS is a first step in the implementation of the FIAT environment on AIPS. FIAT requires the instrumentation of the software system to achieve controllability of the fault injection and monitor the activities of the system. This instrumentation is provided by the sensor mechanism used in PIE[16]. Additionally, the prior implementation of PIE on AIPS provided a data collection method to be used with FIAT, and the *Automation and Unity* desired for a validation environment.

The integration of PIE and FIAT is presented in Figure 6. The left half of the figure is the PIE instrumentation, while the right half is FIAT fault generation and experiment development. The joining of the two environments is with the database and the execution unit. The database merges information regarding the workload structure from the PIE side with the fault information from the FIAT side. This joint information is then used for the experiment execution and subsequent data processing.

Figure 7 shows a general overview of the hardware and software configuration of this system and the continued extension for the FIAT requirement for AIPS/ALS. The hardware configuration of the system is extremely simple, because PIE is a software based monitoring system. The hardware components consist of one or more AIPS FTP nodes, the AIPS VAX host, and the PIE workstation. The FTPs are connected to the AIPS VAX via a serial communication link. This link is used to load system and user programs into the FTPs, and to retrieve monitoring data collected during program execution. The connection between the AIPS VAX and the PIE workstation can be implemented in several of ways. If the PIE workstation is to be located near the FTPs and the AIPS VAX, a local area network or a fast serial line can be used. If the PIE workstation is not within physical reach of the AIPS VAX, remote modems have been used successfully to operate the system.

The software components of the PIE AIPS monitoring system can be divided into three groups. The first group is comprised of code for use on the AIPS FTPs, which includes the kernel monitor and the user monitor. The kernel monitor gathers information about context-switches performed by the Ada run-time, while the user monitor collects information about Ada language constructs in the user's program, such as rendezvous, task creations, or simple procedure calls. The second group is the DCL (Digital Command Language), which runs on the AIPS-VAX host and interfaces with the AIPS VIPS debugger program; its function is to read data from the FTP's memory and reformat it for transmission to the PIE workstation. The last group of software components are the PIE tools which reside on the workstation, including PIEmacs, the PIE Instrumentor, and the PIEscope.

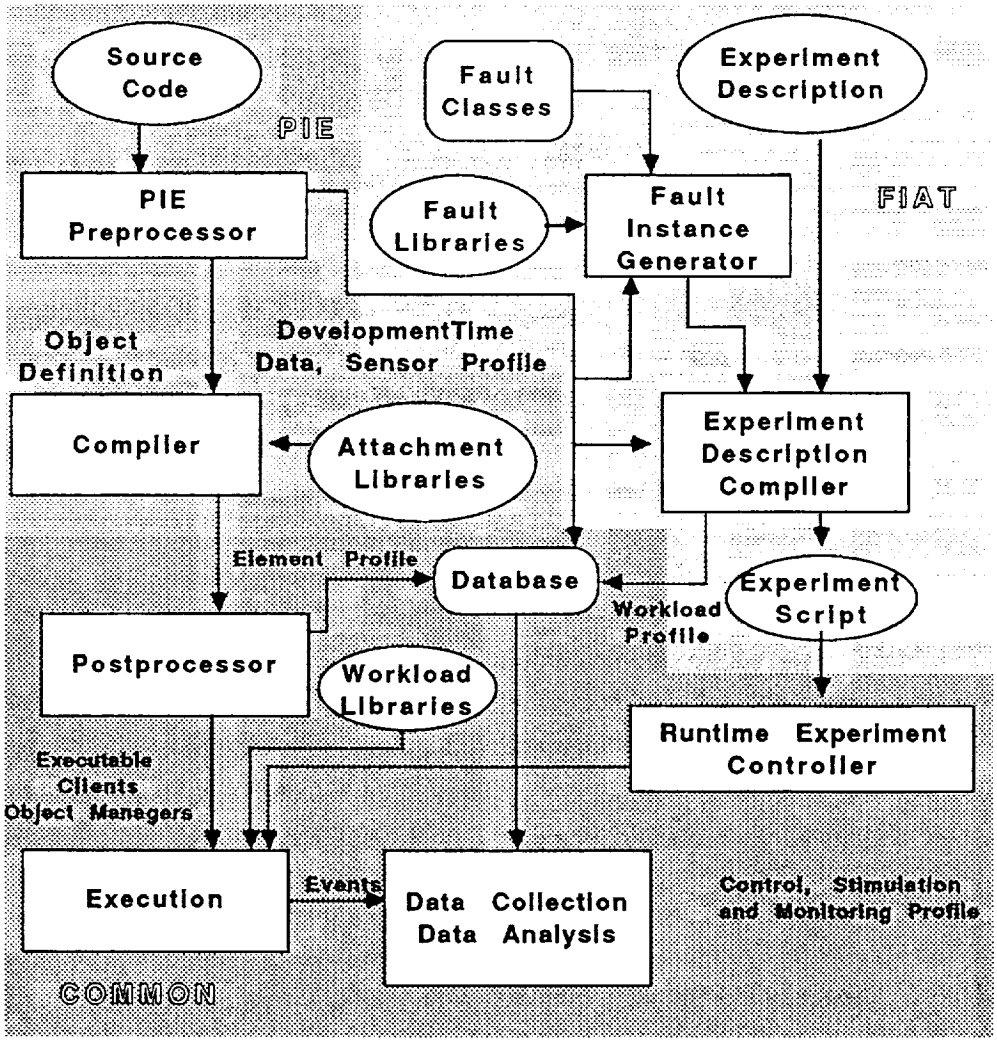


Figure 6: Integration of PIE and FIAT

This report concentrates on initial work to instrument and monitor the functionality of the AIPS fault-tolerant distributed real-time system using PIE, the Programming and Instrumentation Environment. Near-term extensions include exploring the bounds of the correct functioning through fault injection and the integration of the FIAT, Fault Injection and Automated Testing Environment, software.

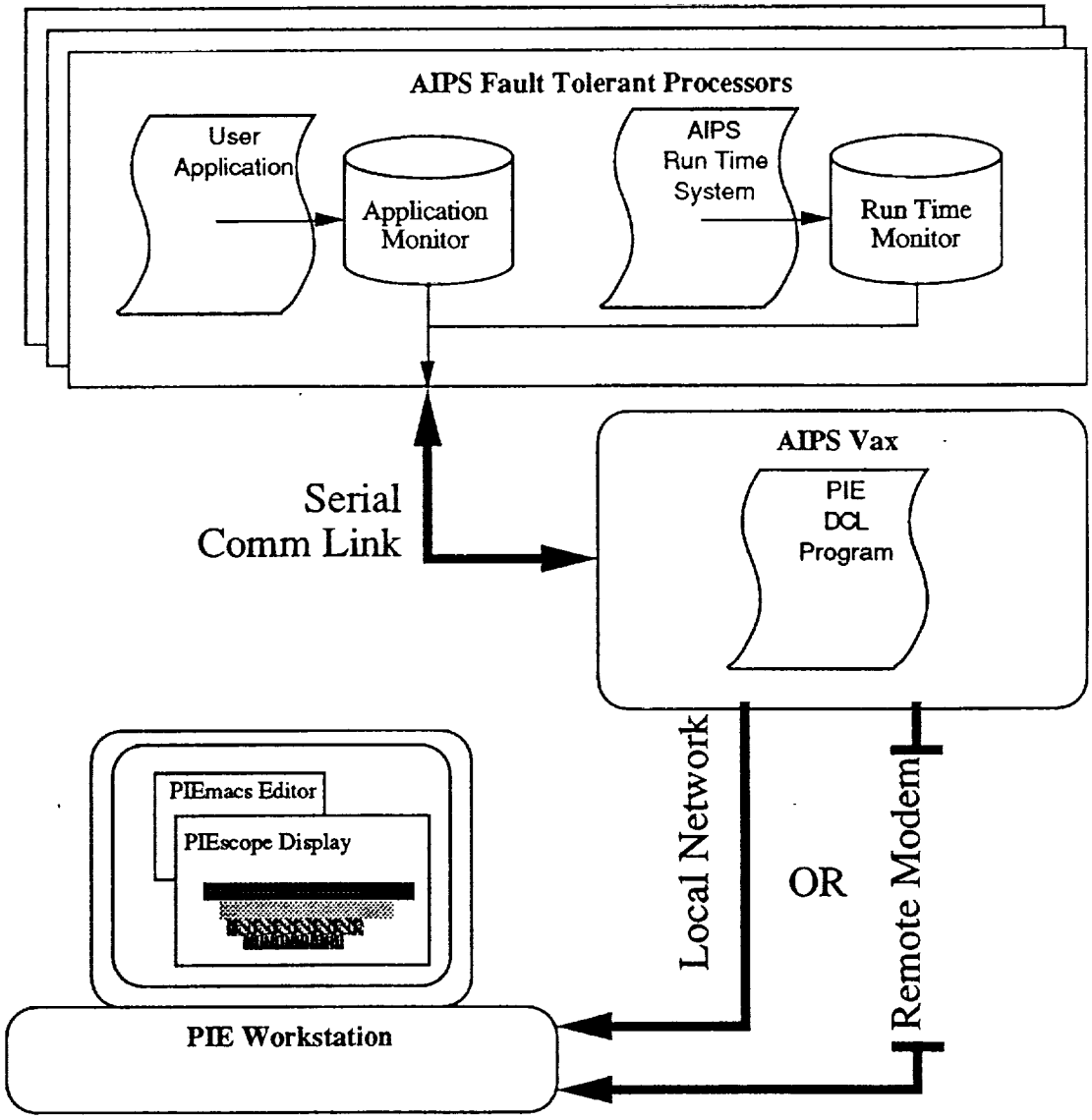


Figure 7: Overview of Configuration for the PIE-AIPS Monitoring System

3.4 PIE Software

In this section, the design and the implementation details of the PIE software system onto the AIPS/FTP are presented.

3.4.1 PIE Software on the AIPS FTP

The PIE monitoring run-time system usually has two main components. The component for generating data is called a sensor, while the other component is the collector which processes and writes the generated data to a file for later analysis. However, the AIPS system used in the laboratory for the development of the prototype AIPS model lacks external data storage. Thus, in this implementation, all data collected are stored in memory and retrieved at a later time by a DCL script executing on the AIPS VAX host. Other possible implementations are discussed below.

The design of the user sensor, which generates system monitoring data, is straightforward. When a user sensor is executed, it records information about the associated Ada language construct and the time at which this event occurred into a memory buffer. However, the laboratory prototype AIPS system has only 2 Megabytes of memory, with only about 50 Kilobytes available for the storage of user sensor data, so priority has to be given to efficient usage of this resource. For this purpose, a simple memory manager is incorporated into the user sensor.

First considered was a memory manager design where each Ada task in the user's program is given an equal share of the 50K of memory and normal data collection for each task continues until the task exhausts its private memory buffer. Clearly, this is not an efficient use of this scarce resource, because different tasks might have different sensor firing rates, and some tasks may not have any sensors. On the other design extreme, each task's sensor data could be written into a single common buffer achieving the most efficient use of memory, but the synchronization overhead incurred to prevent race conditions on sensor operation would make the system useless.

The implementation of the PIE user sensor on AIPS is a compromise, balancing efficiency and speed, and is known as the fast bucket-switching memory manager. The basic idea is every task has a private memory buffer called the bucket. All sensor data from the task is saved by the sensor into this bucket. When the bucket is full, the sensor will check-in the full bucket to a full bucket queue and check-out an empty bucket from an empty bucket queue. The queue used for holding the full buckets is a FIFO queue in order to preserve the temporal ordering of the buckets, while the empty queue is implemented with a LIFO queue for simpler operation. Both queues are controlled by a single lock to prevent any race conditions. The size of the buckets in

this system is kept relatively small, so the memory manager can adopt to the different sensor firing rates of the user tasks. With synchronization occurring only when buckets are switched, the system performance is maintained at an acceptable level.

One issue regarding this bucket-switching system, which needs to be addressed before being used for the monitoring of the AIPS system, is: what does the monitoring system do when the empty buckets have all been exhausted? With only 50K of memory allocated for user sensor buckets, running out of empty buckets during execution is the norm rather than the exception, regardless of the efficiency of memory utilization. A great deal of power and flexibility are built into the mechanism for handling the event when no empty buckets are available. The system offers two options: the monitoring system can lose some sensor data or the system can block further execution using the AIPS halt feature to allow the AIPS VAX host to empty all memory buffers. The user select can switch between the two options during run time under program control. Figure 8 shows the data collection management system.

The second option, halting the system is only intended for use during testing of application programs when loss of monitoring data would make it harder for testing and debugging. For a deliverable FTP control system, other solutions need to be explored, such as (1) losing data, (2) configuring the FTP with external storage, (3) moving the data to another computer that has an external storage device, or (4) storing the data in non-volatile memory. These other solutions are not further explored here.

However, the current implementation of the PIE monitoring does support two different methods for losing data which can be selected to add additional flexibility to the AIPS user sensor. In the first method, all further sensor data will be lost for tasks which cannot obtain an empty bucket, allowing the user to record a complete picture of the beginning of the execution until memory is filled. In the second method, the oldest full bucket is recycled, which allows for a good view of the end of the computation[4]. This method is especially useful for diagnosing conditions inside the AIPS system just prior to some type of failure.

On the kernel sensor side, one 5K buffer is used to store context-switching information. The more complicated mechanism, used to manage the user memory buffers, is not needed here because the kernel is a single task with a fairly steady firing rate of about 40 ms between context switches. When the buffer is full the same options available in the user sensor, dropping data, blocking further execution, off-loading the data, etc., are also available in the kernel sensor.

[4]The last two buckets of every task are protected from recycling to prevent a highly active task from dominating all the memory buckets.

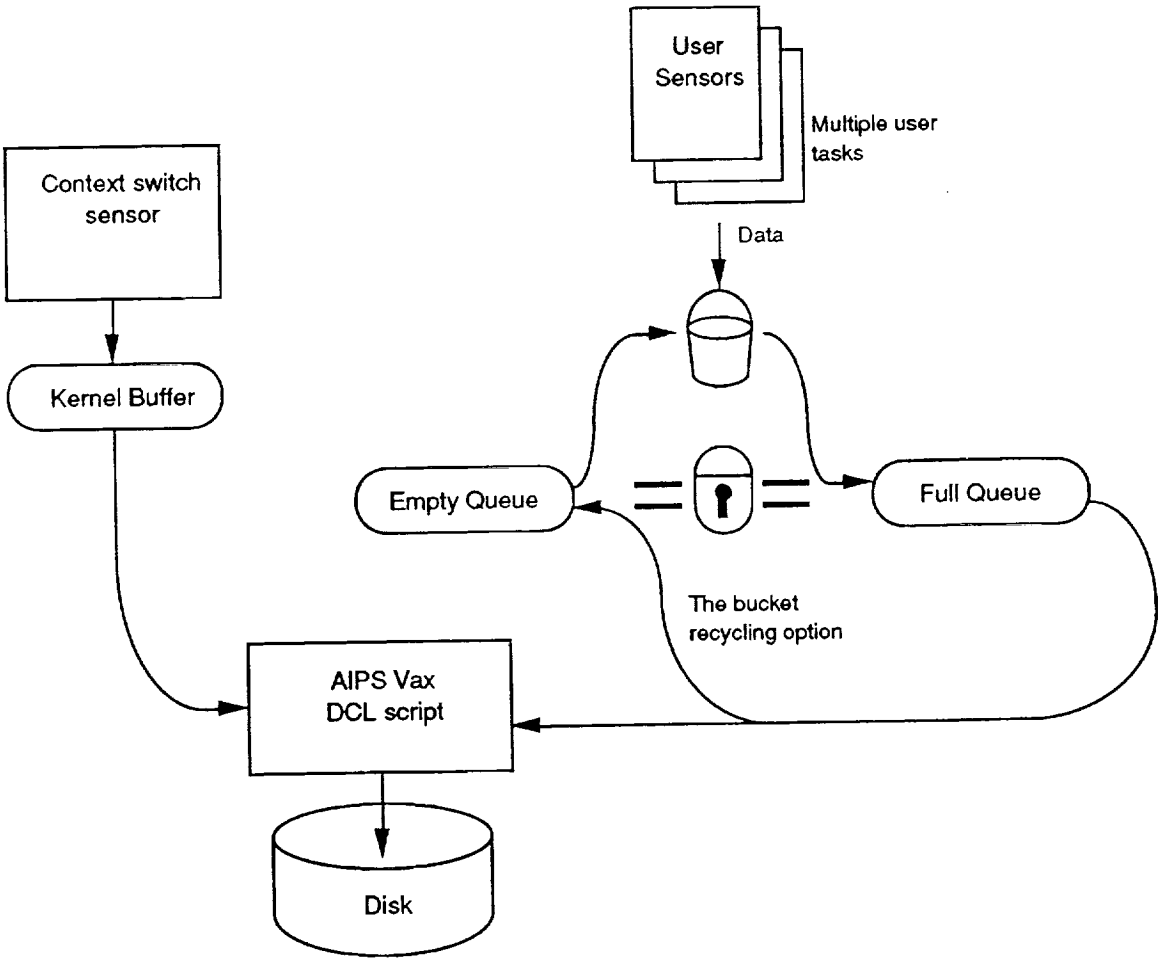


Figure 8: The User and Kernel Sensors on the AIPS FTPs

3.4.2 PIE Software on the AIPS Host

The only PIE software residing on the AIPS VAX host is a DCL script, serving as the data transfer agent between the AIPS FTPs and the PIE workstation. As described above, the data collected by the the monitoring sensors on the FTPs are kept in memory. When the AIPS FTPs are halted[5], either under program control or when execution has been completed, the DCL script will retrieve the sensor data to the VAX host through the use of the memory dump facility of the VIPS debugger. Additionally the DCL performs some data compaction to speed the data transfer to the PIE workstation.

[5]It should be pointed out that *halted* here means that the AIPS monitor is invoked and the real-time clock and two of the three interval timers are stopped. The FTP itself is not halted, and the processor continues to execute instructions. This feature is particular to the laboratory prototype.

The use of the interpreted DCL script language has made the memory retrieval process fairly slow. As part of a future enhancement, the functionality of the VIPS memory dump and other relevant commands could be incorporated into a compiled program dedicated to the PIE memory retrieval task.

3.4.3 Workstation Software

The PIE workstation software is a complete program development environment. The AIPS Ada program which is monitored must first pass through the PIEmacs editor. This modified gnu-emacs program provides a complete editor for program development, plus it also parses and extracts important Ada syntactical information from the user's Ada program. This information will later enable the PIEscope tool to interpret the data collected during execution and present the information clearly. After the program has been developed to the satisfaction of the user, the program is then automatically modified by the PIE Instrumentor. The Instrumentor inserts the software sensors into the proper place in the user's program. This modified source code is ready for compilation on the VAX host and execution on the AIPS FTPs. Finally the data returned from the FTPs is analyzed by the PIEscope and graphical views of the data is presented to the user. It is not within the scope of this report to detail the complete design and implementation of these tools; several references give detailed information[15, 17].

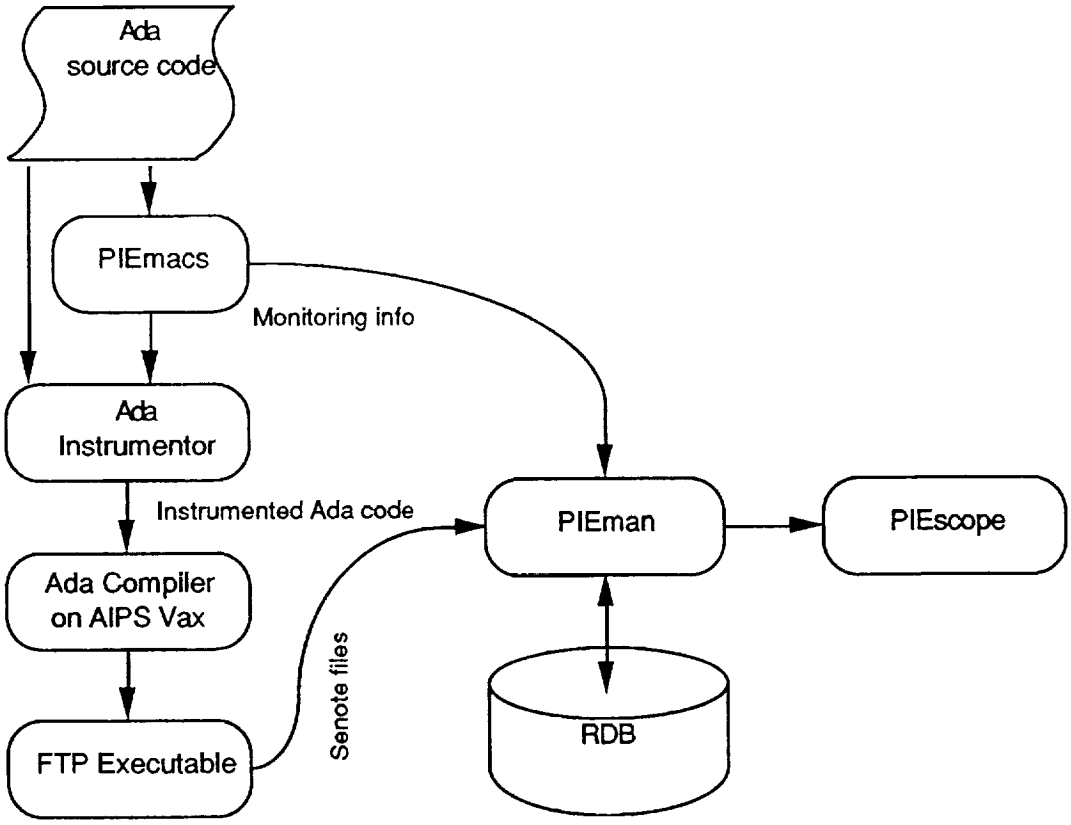


Figure 9: PIE Workstation Software Diagram

4 Experiment Design and Results

This section contains the description and experimental results of three Ada test programs. Each of these test programs are designed to exercise and test one or more functions of the AIPS fault-tolerant run-time or the Ada language run-time. All the experimental data presented below has been verified by repeated trials.

Two versions of each program are presented. The first versions were run at no specified priority, which is priority zero by default. The second versions were run at the highest user priority possible (96). The only higher priority task is the fast FDIR (Fault Detection Isolation and Recovery) routine. The priority-setting mechanisms require that the high-priority tasks be subtasks to the program's main task. Thus, the second versions of the test programs have been rewritten to reflect this necessity. Both versions begin with a delay of 10.0 seconds to allow the AIPS run time to initialize itself. Appendix A provides raw data and pictorial images from the PIEscope package. Data presented in this section are taken from the PIEscope images in the Appendix.

4.1 LoopTest

The LoopTest is conceived to test the AIPS run-time system overhead. The Low priority LoopTest program is simple: consisting of two loops, one inner loop and one outer loop, each iterating a fix number of cycles to act as a synthetic workload. The complete program is listed below.

```
procedure looptest is
  NUM_ILOOPS: constant integer := 1000;
  NUM_OLOOPS: constant integer := 10;

  fred: integer;

begin
  delay 10.0;
  for i in 1..NUM_OLOOPS loop -- outer loop
   for j in 1..NUM_ILOOPS loop -- inner loop
     fred := 3; -- fake work
     fred := fred * 4 + 5;
     fred := fred + j;
   end loop;
  end loop;
end looptest;
```

Using the PIE kernel monitoring feature, all context-switches during the execution of the

test are recorded. The PIE system can identify the task which corresponds to the body of the test program. The context-switching data collected reflects an accurate measurement of the total cpu cycles available to the user's program running at a *non-specified* (that is, zero) priority. Since there are no requests of system services in the program, extra context-switches are not caused by the test program. Hence, the user's program will be time-sliced with all the AIPS priority 0 tasks (self test, the three CRT display tasks, and the MAC display task). By comparing the total time the looptest task is switched-in against the total time of the program execution, one can see the total percentage cpu utilization for the user's program. The data collected for LoopTest show the utilization is only 60.61% under these conditions.

In the high priority version of LoopTest, the low priority test is placed in a subtask set at priority 96. The complete program is listed below.

```
procedure looptest is
  TASK type looptest_task is
    PRAGMA priority(96);
    ENTRY start;
  END looptest_task;

  TYPE looptest_task_ptr is access looptest_task;
  a_loop: looptest_task_ptr;

  TASK body looptest_task is
    NUM_ILOOPS: constant integer := 1000;
    NUM_OLOOPS: constant integer := 10;

    fred: integer;

    BEGIN
      for i in 1..NUM_OLOOPS loop -- outer loop
       for j in 1..NUM_ILOOPS loop -- inner loop
         fred := 3; -- fake work
         fred := fred * 4 + 5;
         fred := fred + j;
       end loop;
      end loop;
    END looptest_task;
BEGIN
  delay 10.0;
  a_loop := new looptest_task;
  a_loop.start;
END looptest;
```

As before, PIE's kernel monitoring feature records context-switches during the execution of the program. However, the high priority results are much different. Now, the looping task runs mainly uninterrupted, except for the steady 40 ms execution rate of the redundancy management task, FDIR, which is run at highest priority. Each context switch takes about 1.5

ms, which is equal to the time for a context switch to the FDIR, the execution of the FDIR, and the context switch back. These interruptions account for a total switched-out time of 9.28 ms, or about 3.86% of the looping tasks computation. That is, the looping tasks' cpu utilization is 96.14%; this is in approximate agreement with LoopTest.

4.2 MemTest

The MemTest is designed to demonstrate the characteristics of the current memory manager in the Ada run-time system. The memory manager is implemented using an unordered linked-list to maintain the free memory blocks. An allocation for a new block is done by searching down the linked-list until either the end of the list or an element with a block of memory greater than the requested allocation is found. The sequence of allocations and deallocations in this test is designed to create a free list with many small memory blocks at the head of the list, before measuring the performance of a large block allocation.

The program first allocates a large block of memory, the size of 300 integers, and deallocates it to obtain a reference time for later comparison. Next the program allocates storage for a single integer and repeats the allocation 300 times. Then the 300 integers are freed. Finally, a large block identical in size to the first large block is allocated and freed. Once again this program is running at the default priority (zero), and has an initial delay of 10 seconds. The program is listed below.

```
with unchecked_deallocation;
procedure nmemtest is
  NUM_ALLOC: constant integer := 300;

  type int_array is array(1..NUM_ALLOC) of integer;
  type int_array_ptr is access int_array;
  type integer_ptr is access integer;

  fred: integer;
  my_int: array(1..NUM_ALLOC) of integer_ptr;
  my_int_array: int_array_ptr;

  procedure free_int
        is new unchecked_deallocation(integer, integer_ptr);
  procedure free_array
        is new unchecked_deallocation(int_array, int_array_ptr);

begin
  delay 10.0;
  for i in 1..1 loop -- allocate and free first big block
    my_int_array := new int_array;
  end loop;
```

```
  free_array(my_int_array);

  for i in 1..3 loop -- allocate lots of little blocks
    my_int(i) := new integer;
  end loop;
  for i in 4..NUM_ALLOC-3 loop
    my_int(i) := new integer;
  end loop;
  for i in NUM_ALLOC-2..NUM_ALLOC loop
    my_int(i) := new integer;
  end loop;
  for i in 1..NUM_ALLOC loop -- free all the little blocks
    free_int(my_int(i));
  end loop;

  for i in 1..1 loop -- allocate the second big block
    my_int_array := new int_array;
                                  -- should take much longer than first
  end loop;
end nmemtest;
```

The data collected from this test show the memory manager takes a constant time to perform an allocation if the memory request can be handled with the first free block in the linked-list. The allocation time of the first large block is identical to the time for allocating the subsequent small blocks, each operation taking about 1.5 ms. However, if the memory manager has to traverse the list to find a large enough block to fulfill the request, the time needed to perform the allocation is dependent on the number of links it needs to travel. In the case of the MemTest program, the number of links is 300, and the time required to allocate the second large block is over 14.5 ms, almost a 10 fold increase, or approximately 43μs per link traversed.

Examining the context switch behavior, we see that no context-switches occurred during the first or last memory allocations, so we can expect that the memory allocation times under high priority to be similar. More generally, the total switched out time was 218.6 ms, or about 49.98% of the execution.

The complete listing of the high-priority version of MemTest is listed below. The task is subtasked to allow for a priority of 96.

```
with unchecked_deallocation;
procedure nmemtest is
  NUM_ALLOC: constant integer := 300;

  type int_array is array(1..NUM_ALLOC) of integer;
  type int_array_ptr is access int_array;
  type integer_ptr is access integer;
  fred: integer;
  my_int: array(1..NUM_ALLOC) of integer_ptr;
```

```
my_int_array: int_array_ptr;

procedure free_int
      is new unchecked_deallocation(integer, integer_ptr);
procedure free_array
      is new unchecked_deallocation(int_array, int_array_ptr);

TASK type nm_task is
  ENTRY start;
  PRAGMA priority(96);
END nm_task;
TYPE nm_ptr is access nm_task;
nm : nm_ptr;

TASK body nm_task is
  begin
    ACCEPT start;
    for i in 1..1 loop -- allocate and free first big block
      my_int_array := new int_array;
    end loop;
    free_array(my_int_array);

    for i in 1..3 loop -- allocate lots of little blocks
      my_int(i) := new integer;
    end loop;
    for i in 4..NUM_ALLOC-3 loop
      my_int(i) := new integer;
    end loop;
    for i in NUM_ALLOC-2..NUM_ALLOC loop
      my_int(i) := new integer;
    end loop;
    for i in 1..NUM_ALLOC loop -- free all the little blocks
      free_int(my_int(i));
    end loop;

    for i in 1..1 loop -- allocate the second big block
      my_int_array := new int_array;
                                -- should take much longer than first
    end loop;
  end nm_task;

begin
  delay 10.0;
  nm := NEW nm_task;
  nm.start;
end nmemtest;
```

As expected, the memory allocation test results are mainly unchanged under high-priority. Here, the initial memory allocation took 1.7 ms, and the final allocation ran for 14.7 ms for an 8.5 increase in time, or again 43μs per link traversed. Further analysis is needed to determine code portions which use this allocation routine, (hence are subject to this behavior) and how this behavior may affect the performance of hard-deadline real-time tasks. The collected data

indicates that the total switched out time was much improved with only 7.7 ms spent switched out or about 3.43% of the total execution.

Draper Laboratories concur with the assessment of unpredictability and poor performance for the Ada memory allocation routine. They state the cause as an inefficient Verdix implementation, and do not consider it an important problem. Their basis, for the lack of importance, is that "... in most real-time systems, dynamic task (and therefore memory) allocation is not performed during critical portions of the code. Most memory allocations are done during elaboration or during specific initialization modes. If a requirement for dynamic memory allocation were to be specified, the memory allocation routines would have to be modified."[18] With PIE and the fault-free validation methodology uncovering this normal behavior, the usefulness of the methodology in uncovering poor performance is demonstrated and moreover, areas which need further investigation are defined.

4.3 ActTest

The ActTest is designed to test the efficiency and real-time characteristics of the Ada task creation and rendezvous mechanism. The program first creates seven identical tasks. Then the main task will perform a rendezvous with each of the seven tasks. The task finishes and exits after the rendezvous. The main task runs at priority zero, since no priority is specified; all seven of the created sub-tasks have priority of 96. The program again has the 10 second delay built in at the beginning.

The low priority test is

```
PROCEDURE acttest IS
  NUM_TASK : CONSTANT integer := 7;

  TASK TYPE activity IS
    ENTRY start ;
    pragma PRIORITY(96);
  END activity;
  TYPE activity_ptr IS ACCESS activity;

  fred : integer;
  my_act : ARRAY ( 1 .. NUM_TASK ) OF activity_ptr;

  TASK BODY activity IS
    j : integer;
  BEGIN
    ACCEPT start DO
      FOR i IN 0 .. 1000 LOOP -- Do some fake work in rendezvous
        j := i;
      END LOOP;
    END start;
```

```
  END activity;

BEGIN
    DELAY 10.0;
    FOR i IN 1 .. NUM_TASK LOOP  -- allocate tasks
      my_act ( i ) := NEW activity ;
    END LOOP ;

    FOR i IN 0 .. 1000 LOOP  -- dummy delay loop
      fred := i;
    END LOOP;

    FOR i IN 1 .. NUM_TASK LOOP  -- rendezvous with each task
      my_act ( i ).start;
    END LOOP;

    FOR i IN 0 .. 3000 LOOP -- dummy delay loop again
      fred := i;
    END LOOP;
END acttest;
```

The collected data from this test show the impact of one low-priority task. After each subtask is initialized, it takes about 30 ms for the main task to "wake up" from its context switch, reloop and start the creation of the next subtask. It takes even longer for the main task to recover from each rendezvous. Our collected data shows that these long recovery times are not due to the execution of the other subtasks, which remain switched out during this period. However, the long recovery times are attributable to the other system tasks which run at priority zero[6]. These system tasks must be switched in and executed each time the main task is preempted, this is due to the ordering of all equal-priority tasks on the run queue. Note that this behavior is not seen in the reverse when the subtasks need to be switched in, (i.e. as soon as the low-priority main task calls the new statement to create a high-priority subtask, the subtask runs immediately.) A similar behavior occurs when the subtasks are called on to execute their accept statements.

Since each subtask is dependent on the performance of the main task, the total execution time of this test is seriously degraded. Measurements indicate that the total switched-out time was 395.8 ms, or about 50.8% of the execution time; thus the cpu utilization is 49.2%.

However, the recorded results show that both the task creation time and the rendezvous time have an upward trend. The cause of this behavior is not clear from the experimental data, but further experimentation (by Draper) has indicated that it may be due to the sensor monitoring. Further experimentation would be needed to accurately calibrate the influence of the sensors.

[6]The influence of the system threads shows that it might be beneficial to monitor the system tasks so a complete picture of the system's behavior can be seen.

The graphs and the raw numbers for the low priority execution can be seen in Table 3 and Figure 10. Although the high priority numbers are smaller, the trend is similar.

Time for starting a task (milli-seconds)	Time for starting a rendezvous (milli-seconds)
8.085	3.407
10.931	3.976
9.029	4.104
9.227	4.236
9.426	4.372
9.821	4.636

Table 3: Task Starting and Rendezvous Times from ActTest

The complete listing of the the high-priority version of the ActTest follows.

```
PROCEDURE acttest IS
  NUM_TASK : CONSTANT integer := 7;

  TASK TYPE activity IS
    ENTRY start ;
    pragma PRIORITY(96);
  END activity;
  TYPE activity_ptr IS ACCESS activity;

  TASK TYPE enclose_task IS
    ENTRY starttask;
    pragma PRIORITY(96);
  END enclose_task;
  TYPE enclose_ptr is access enclose_task;

  fred : integer;
  my_act : ARRAY ( 1 .. NUM_TASK ) OF activity_ptr;
  enclosure : enclose_ptr;

  TASK BODY activity IS
    j : integer;
  BEGIN
    ACCEPT start DO
      FOR i IN 0 .. 1000 LOOP -- Do some fake work in rendezvous
        j := i;
      END LOOP;
    END start;
  END activity;

  TASK BODY enclose_task IS
  BEGIN
    ACCEPT starttask;

    FOR i IN 1 .. NUM_TASK LOOP   -- allocate tasks
      my_act ( i ) := NEW activity ;
    END LOOP ;
```

```
    FOR i IN 0 .. 1000 LOOP  -- dummy delay loop
      fred := i;
    END LOOP;

    FOR i IN 1 .. NUM_TASK LOOP  -- rendezvous with each task
      my_act ( i ).start;
    END LOOP;
    FOR i IN 0 .. 3000 LOOP -- dummy delay loop again
      fred := i;
    END LOOP;
  END enclose_task;

BEGIN
  delay 10.0;
  enclosure := NEW enclose_task;
  enclosure.starttask;
END acttest;
```

The collected data from this test show that all the negative impact of the low-priority task in the previous example has been eliminated. The main task wakes up from its context switches immediately, and the total switched-out time is only 17.1 ms, or about 3.62%. That is, the cpu utilization is 96.38%, again in agreement with LoopTest and MemTest. However, the upward trend of the task creation and rendezvous times has persisted, indicating that this trend is not priority or context-switching related. This strengthens the belief that the monitoring may be a possible source of these trends.

One intriguing phenomenon in the high-priority version of ActTest is that three of the subtasks do not exit immediately after completing their work. Activities number 3, 5, and 6 (in the accompanying screen snapshots, Figures A.18 through A.23) take one or two context switches before they exit. Strangely, the time spent by these activities, switched in and running, after they have completed their rendezvous is about 2-3 times longer than the activities which do not take a context switch before exiting. We can deduce that the extra execution time is not a result of the task exiting since the context switches occur before the *activity-end* sensor fires. Therefore, suspicion for the source of this extra execution is cast on the context-switch itself or perhaps cast on the expense of recording the context switches' occurrences in the kernel's monitoring mechanism. Further, this may not be an anomaly of neither AIPS nor PIE, but the normal action of an asynchronous, preemptive priority based scheduler. This behavior of the scheduler and how it affects the performance of hard-deadline real-time tasks must be explored in the validation methodology, as listed on page 7, steps 3, 5, and 6.

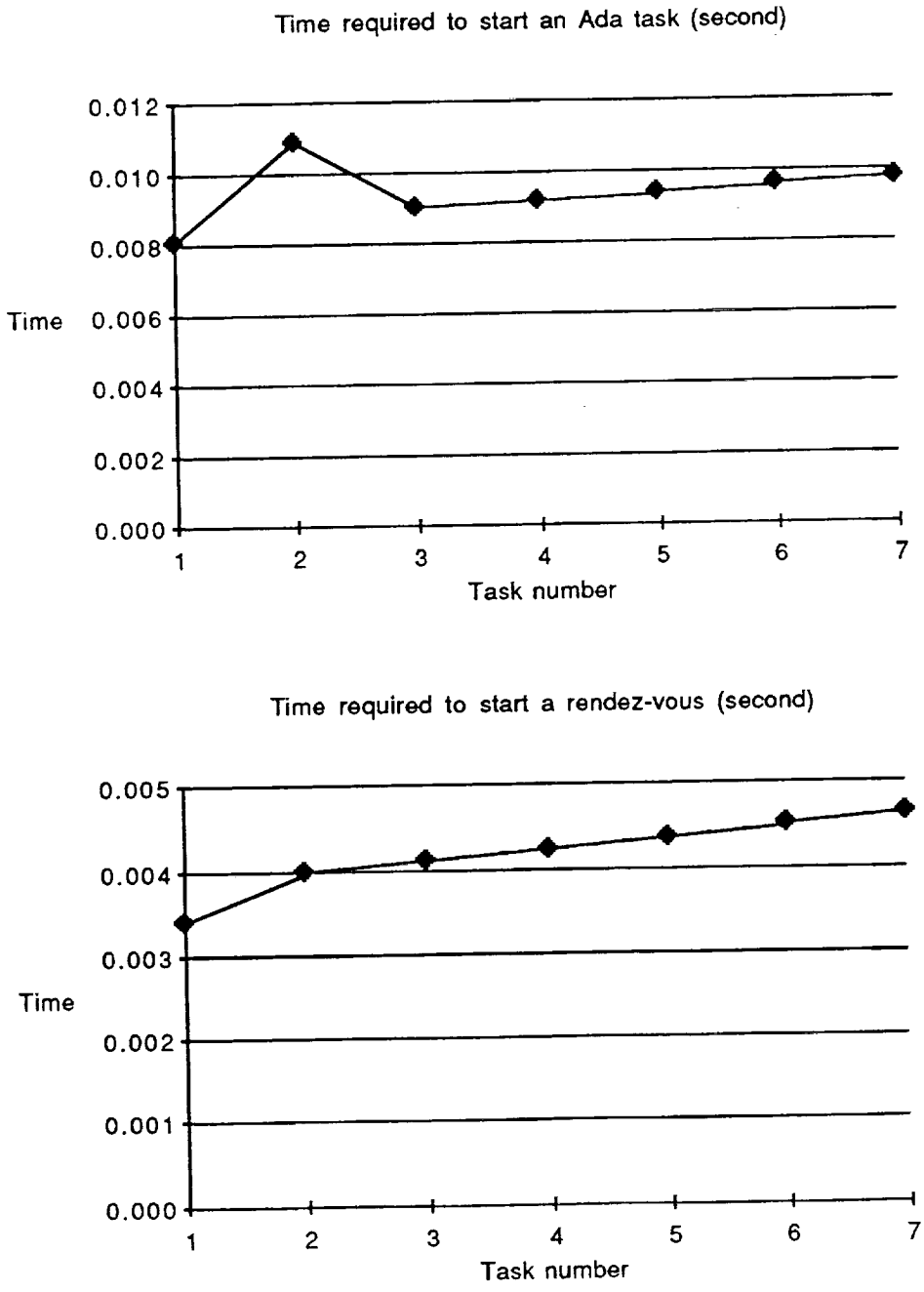


Figure 10: Task Starting and Rendezvous Times for ActTest

5 Summary

This report described the implementation of the PIE system, configured for fault-free validation of the AIPS fault-tolerant computer system. This functionality is required for the implementation of the FIAT environment on AIPS; the PIE components implemented on AIPS represent a substantial portion of the FIAT system. Using these components, a fault-free validation methodology was applied to the AIPS system.

The PIE system has been modified to support the Ada language and a special purpose AIPS/Ada runtime monitoring and data collection implemented. Initially, several Ada programs running on the PIE/AIPS system have been instrumented automatically using the PIE programming environment. PIE's on-line graphical views show vividly and accurately the performance characteristics of the Ada programs, the AIPS kernel and the application's interaction with the AIPS kernel. The data collection mechanisms were written in a high-level language, Ada, and provide a high degree of flexibility for implementation under various system conditions.

Beyond the demonstration of the success of the implementation of the FIAT/PIE, we have characterized some of the critical components of the AIPS/Ada kernel. We paid special emphasis to the performance of the Ada task management functions, communication, synchronization, and memory management. Given the real-time application requirements of ALS, we stressed the need for performance predictability. The collected data have pointed out a number of anomalies. First, the MemTest example indicates the unpredictability of the Ada memory allocator's implementation which uses an unordered linked-list to maintain free memory blocks. Second, the ActTest program shows that the starting and rendezvous times increase linearly with the number of tasks in existence; further investigation is needed to locate the source of this behavior, as it may an artifact of the monitoring itself. Third, the ActTest pointed out a phenomenon wherein context switches seemed to have caused longer execution time. The collected data also show the profound effect of task priority on context switching and runtime overhead.

The Table 4 summarizes the timing results from the experiments. These results point in the following conclusions:

1. The PIE systems provides an automated fault-free validation environment for AIPS. Also we proved the value of PIE as an architecture independent performance evaluation and program development tool.

2. PIE functionality is required for the FIAT system for Fault Injection based Validation of the AIPS for ALS.

Test	total execution	Speedup	total switched out	cpu utilization	first allocation	last allocation
Low Priority LoopTest	449.840	-	177.192	60.61%	-	-
High Priority LoopTest	240.722	46.49%	9.280	96.14%	-	-
Low Priority MemTest	446.441	-	218.662	51.02%	1.543	14.574
High Priority MemTest	227.030	49.15%	7.777	96.57%	1.733	14.763
Low Priority ActTest	777.958	-	395.870	49.11%	-	-
High Priority ActTest	473.086	39.19%	17.108	96.38%	-	-

Table 4: Summary of Results (times are in milli-second)

3. With the FIAT/PIE tools in place, substantial insights into the performance intricacies of AIPS are available.

4. Initial fault-free validation of the AIPS shows a number of anomalies in the critical areas of task management, memory management, communication, synchronization, runtime overhead, and the monitoring itself. Additional work is needed to eliminate or account for any anomalies in the monitoring itself. Moreover, the discovery of these anomalies demonstrate the benefits of the fault-free validation methodology applied to a system under development.

5. Once discovered, those anomalies could either become user considerations or could be fixed in future versions of the AIPS and PIE monitoring. In either case the system will become substantially more predictable and hence suitable for the real-time requirements of the ALS program.

6. Due to the concurrent development of AIPS and FIAT/PIE on AIPS, the FIAT/PIE system have not been yet fully exploited for AIPS validation. (It is difficult to validate a system under development – a moving target.)

7. There is a distinct opportunity with the PIE environment on AIPS and also the need for a fault-free validation to be performed on the final version of AIPS. The validation suite for the final version of AIPS must be biased towards the critical (or unknown) portions of the system to avoid uncovering "known" limitations such as the poor memory allocation performance.

8. After the completion of the fault-free validation, we are strongly suggesting the critical need for *fault injection based validation* of AIPS. Of special concern are the common mode failures (which include most "bugs" in software), communication protocols and multiple single mode failures. For this purpose an opportunity exists in using the proven methodology of FIAT/PIE and the availability of these tools on AIPS.

References

[1] *Advanced Information Processing System (AIPS) Proof-of-Concept System I&E Facility, User's Guide*, Charles Stark Draper Laboratory, 1987.

[2] W. Richards Adrion, Martha A. Branstad, and John C. Cheriavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys*, 14(2):159–192, June 1982.

[3] Algirdas Avižienis and Jean-Claude Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, May 1986.

[4] W.C. Carter. System validation - Putting the pieces together. In *7th AIAA/IEEE Digital Avionics Systems Conference (DASC)*, pages 687–694, 1986.

[5] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell. *Computer Structures: Principles and Examples.* McGraw-Hill Book Company, 1982.

[6] Yves Deswarte, Khadija Alami, and Oliver Tedaldi. Realization, validation and operation of a fault-tolerant multiprocessor: ARMURE. In *16th International Symposium on Fault-Tolerant Computing*, pages 8–13, 1986.

[7] Gary L. Hartmann, Joseph E. Wall, Jr., and Edward R. Rang. Design validation of fly-by-wire flight control systems. In *AGARD Lecture Series No. 143, Fault Tolerant Hardware/Software Architecture for Flight Critical Function*, pages 9.1–9.17. NATO Advisory Group for Aerospace Research and Development, 1985.

[8] H.M. Holt, A.O. Lupton, and D.G. Holden. Flight critical system design guidelines and validation methods. In *AIAA/AHS/ASEE Aircraft Design Systems and Operating Meeting*, 1984. Paper: AIAA-84-2461.

[9] P. Michael Melliar-Smith and Richard L. Schwartz. Formal specification and mechanical verification of SIFT: A fault-tolerant flight control system. *IEEE Transactions on Computers*, C-31(7):616–630, June 1982.

[10] *Validation Methods for Fault-Tolerant Avionics and Control Systems: Working Group Meeting I*, NASA Langley Research Center, March 1979. ORI, Incorporated, Compilers. NASA CP-2114.

[11] *Validation Methods Research for Fault-Tolerant Computer Systems: Preliminary Working Group II Report*, NASA Langley Research Center, September 1979. System and Measurements Division, Research Triangle Institute.

[12] SAE Committee S-18A. Fault/failure analysis for digital systems and equipment. Aerospace Recommended Practice ARP-1834, Society of Automotive Engineers, Warrendale, Pa., August 1986.

[13] Frank Feather, Daniel Siewiorek, and Zary Segall. Fault-free validation of a fault-tolerant multiprocessor: Baseline experiments and workload implementation. NASA CR-178075, Carnegie Mellon Univ., April 1986.

[14] Edward W. Czeck, Frank E. Feather, Ann Marie Grizzaffi, Zary Z. Segall, and Daniel P. Siewiorek. Fault-Free Performance Validation of Fault-Tolerant Multiprocessors. NASA CR-178236, Carnegie Mellon Univ., January 1987.

[15] Ted Lehr, Zary Segall, Dalibor Vrsalovic, Eddie Caplan, Alan L. Chung, and Charles E. Fineman. Visualizing Performance Debugging. *IEEE Computer*, 22(10):38–51, October 1989.

[16] Edward W. Czeck, Zary Z. Segall, and Daniel P. Siewiorek. Predeployment Validation of Fault-Tolerant Systems through Software-Implemented Fault Insertion. NASA CR-4244, Carnegie Mellon Univ., July 1989.

[17] Zary Segall and Larry Rudolph. PIE – A Programming and Instrumentation Environment for Parallel Processing. *IEEE Software*, 2(6):22–37, November 1985.

[18] Roy Whittredge. Draper Laboratories Internal Memo, To: Linda Alger, Subject: Comments on AIPS' Anomalies in CMU Document. May 31, 1990.

A PIEscope Figures of the Experiments

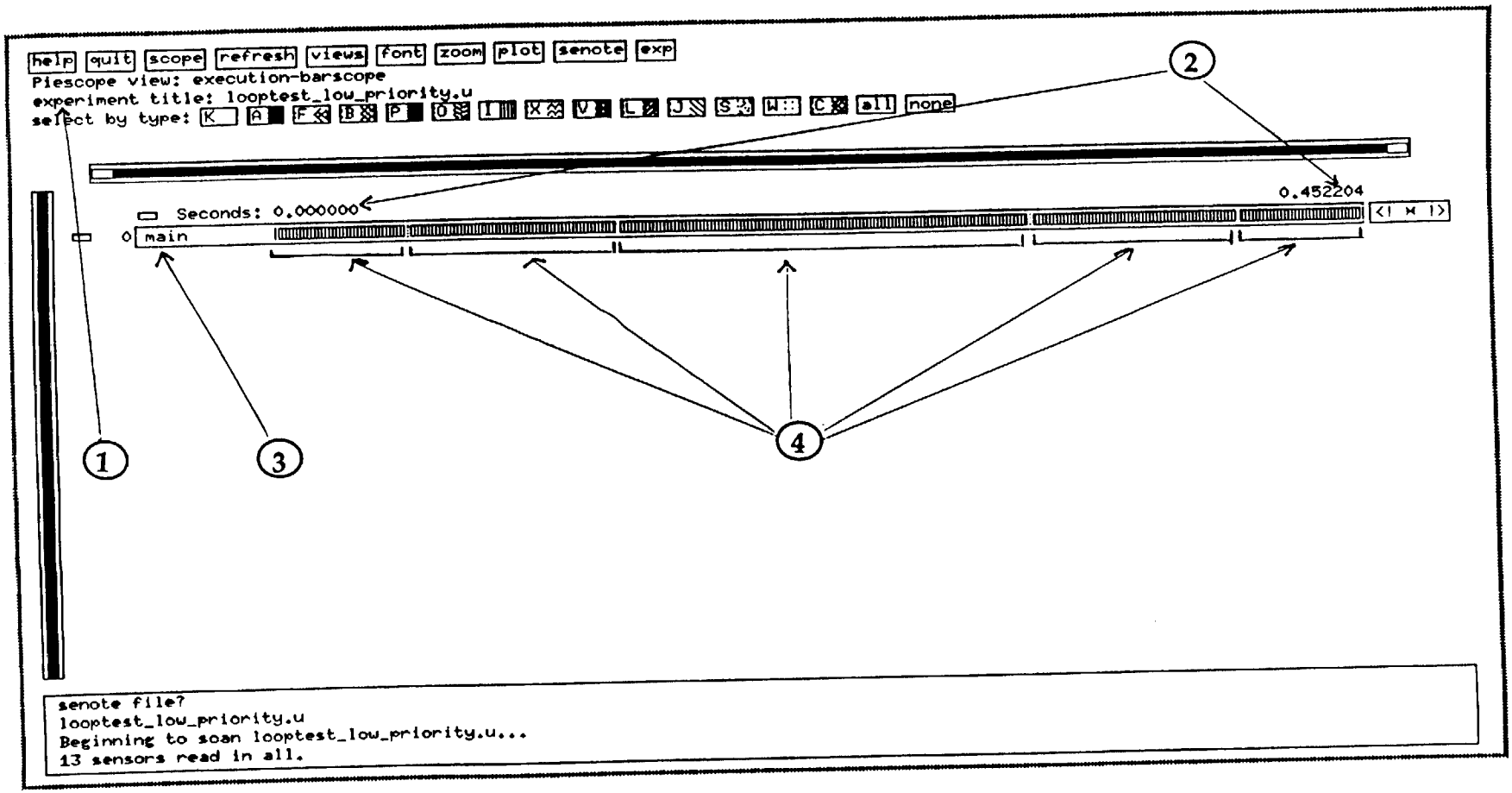
This appendix contains graphical views of the execution of the three test programs, run at both low and high priority. Each figure has been annotated to highlight the area of interest and each set of figures is preceded by a page of explanation. The views were drawn by PIEscope using the X window system.

All of the views follow the same general form: the experiment name is displayed at the top (see notation 1 in Figure A.1); time is displayed on the X-axis (notation 2 in Figure A.1); and each Ada task is displayed as a horizontal bar (notation 3 in Figure A.1) which is colored in from its beginning time to its ending time. Figure A.12 is a clear example of multiple tasks running over different time frames. Different kinds of events that occurred are colored in different patterns. These will each be explained as they are encountered.

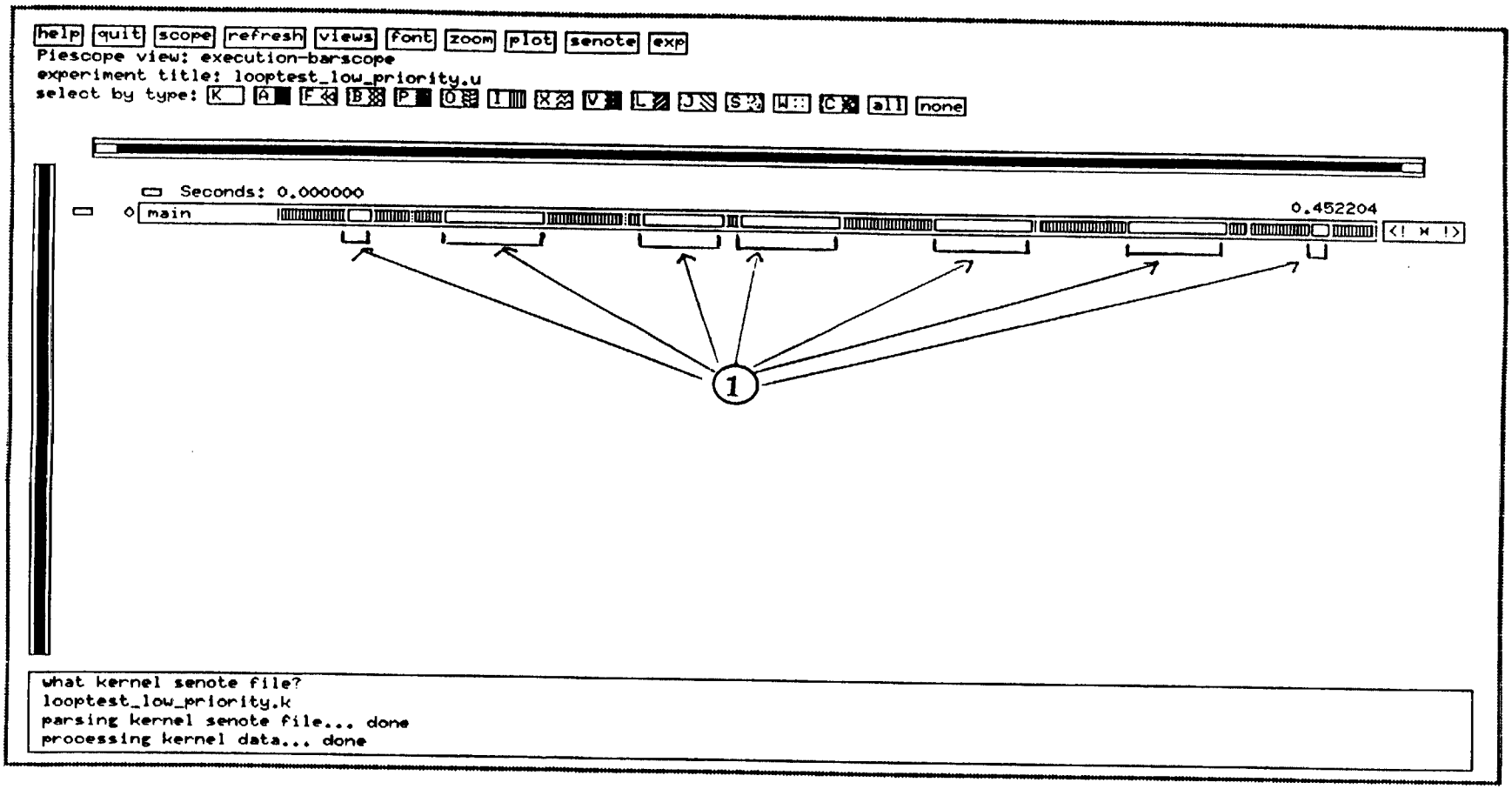
Figures A.17 and A.23 are somewhat different, in that they graphically display the CPU utilization. The notations for those figures will explain the meaning of the pertinent parts of those views.

A.1 Low Priority LoopTest

- Figure A.1, notation 1: The name of the experiment is displayed at the top of the view.
- Figure A.1, notation 2: Time runs along the X-axis, displayed in seconds. Here, we see that the total execution time was 0.452204 seconds.
- Figure A.1, notation 3: Only one Ada task was executed, shown with the name `main`. This task executed completely from beginning to end, thus the entire bar is colored in.
- Figure A.1, notation 4: Each block represents one iteration of the outer loop in LoopTest. Each loop was executed from the beginning to the end of its drawn block.
- Figure A.2, notation 1: This is the same view as in Figure A.1, but now context switch information has been overlaid. Here, a white block indicates that the task was switched out. From this information we determine that the CPU utilization for this experiment is 60.61%.



A.1: Low Priority LoopTest



A.2: View of Low Priority LoopTest Showing Context Switching Information

A.2 High Priority LoopTest

- Figure A.3, notation 1: The scrollbar shows that we are viewing only the end of the experiment. The first ten seconds of the experiment are used in the DELAY statement in LoopTest.

- Figure A.3, notation 2: This experiment now has two tasks. The first is `main` whose only function is to allocate and spawn the subtask that does the real work. This second task `looptest_task` then performs the exact same work as `main` did in the low priority example.

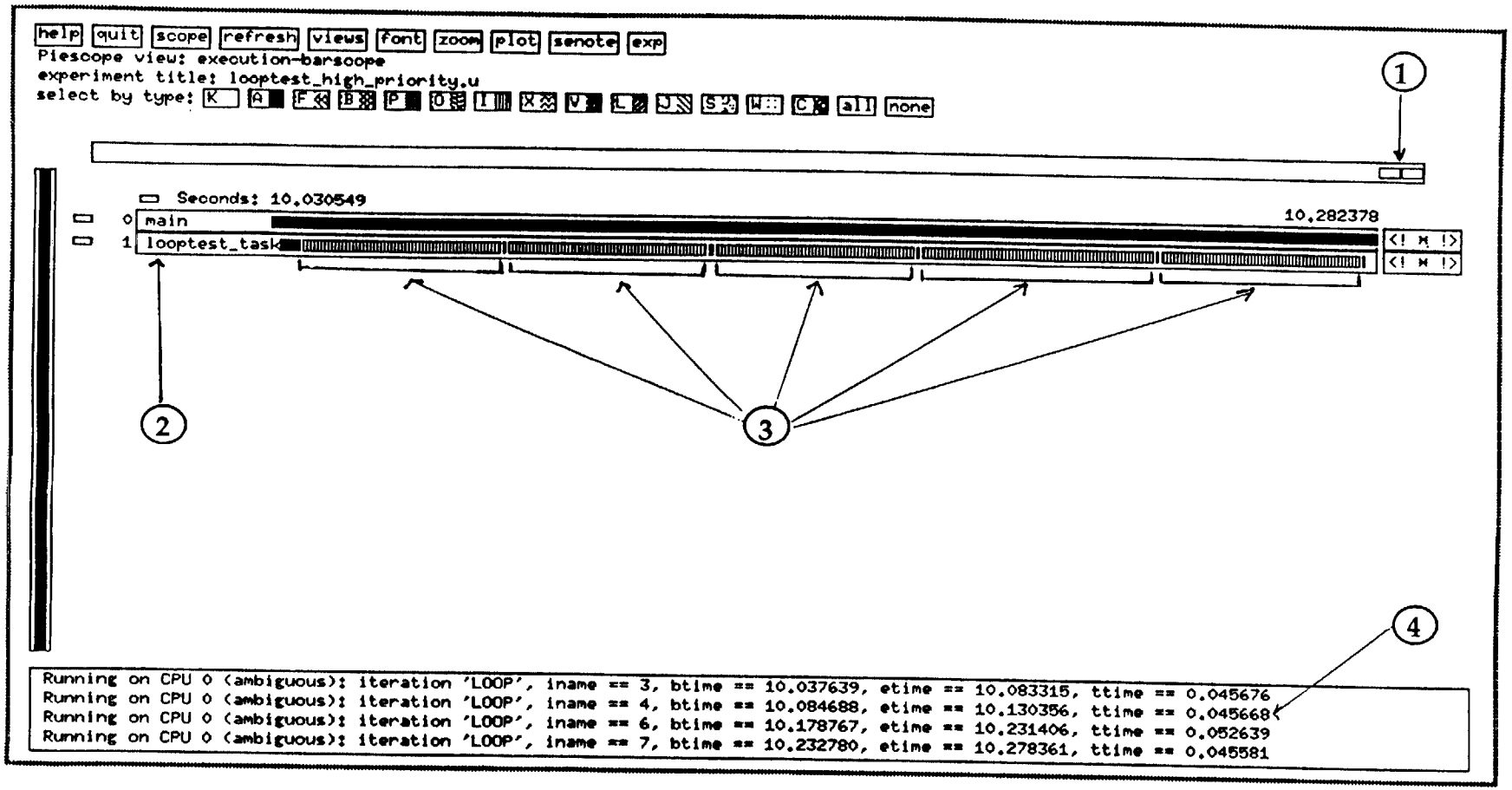
- Figure A.3, notation 3: Each block represents one iteration of the outer loop in LoopTest. Each loop was executed from the beginning to the end of its drawn block.

- Figure A.3, notation 4: When the user clicks the mouse button on each of these blocks then precise information about the execution times of those events is displayed in the window at the bottom of the view.

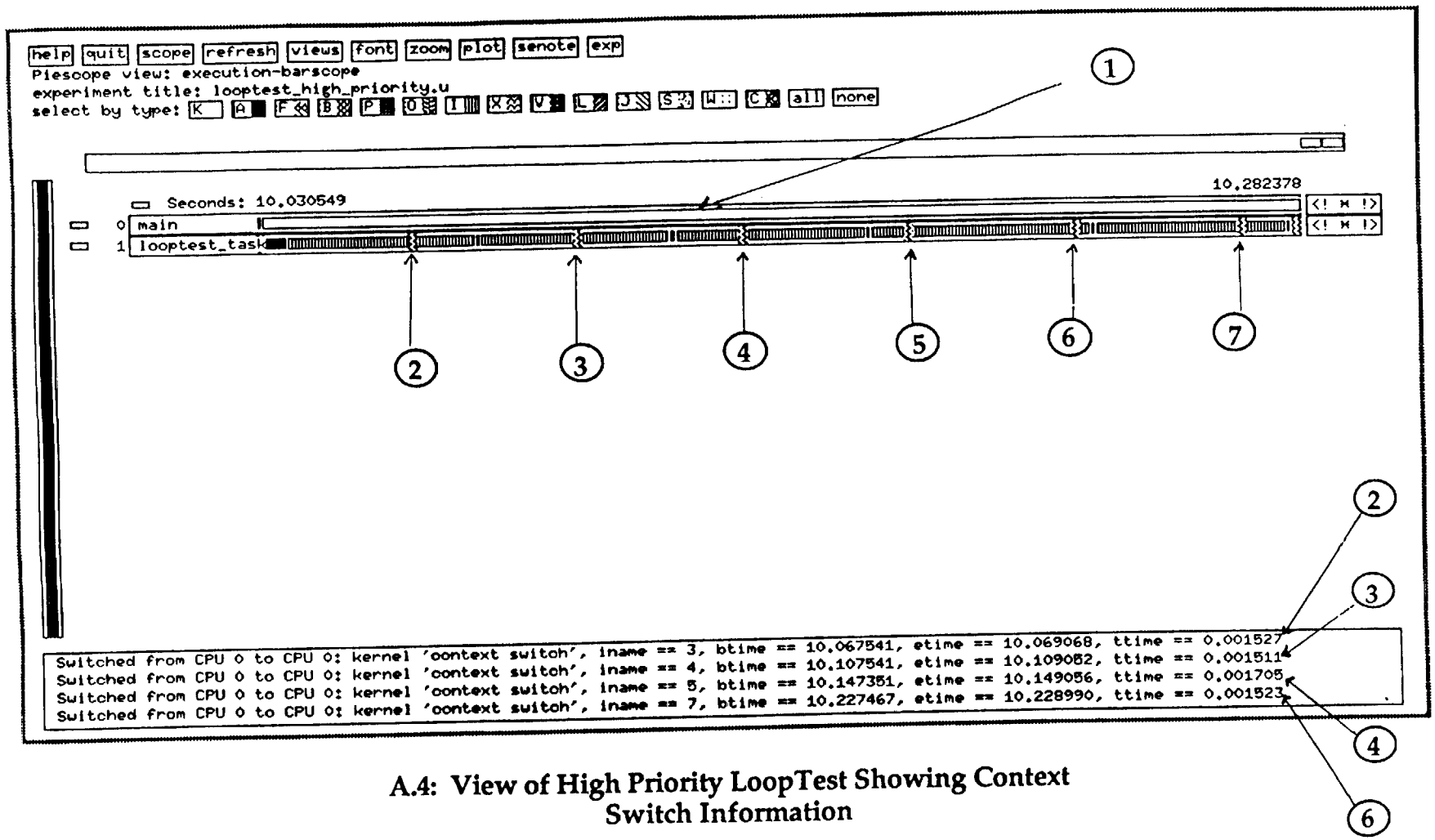
- Figure A.4, notation 1: This is the same view as in Figure A.3, but now context switch information has been overlaid. Here we see that the Ada task `main` was switched out for the entire time that `looptest_task` was doing work.

- Figure A.4, notations 2-7: The *squiggles* indicate that there is not enough screen resolution to display the event. Under each of the *squiggles* is a very short context switch. The precise times for these switches are displayed in the window at the bottom of the view, and have been marked with the corresponding notation numbers. From the context switch information we can calculate that the CPU utilization for this experiment was 96.14%.

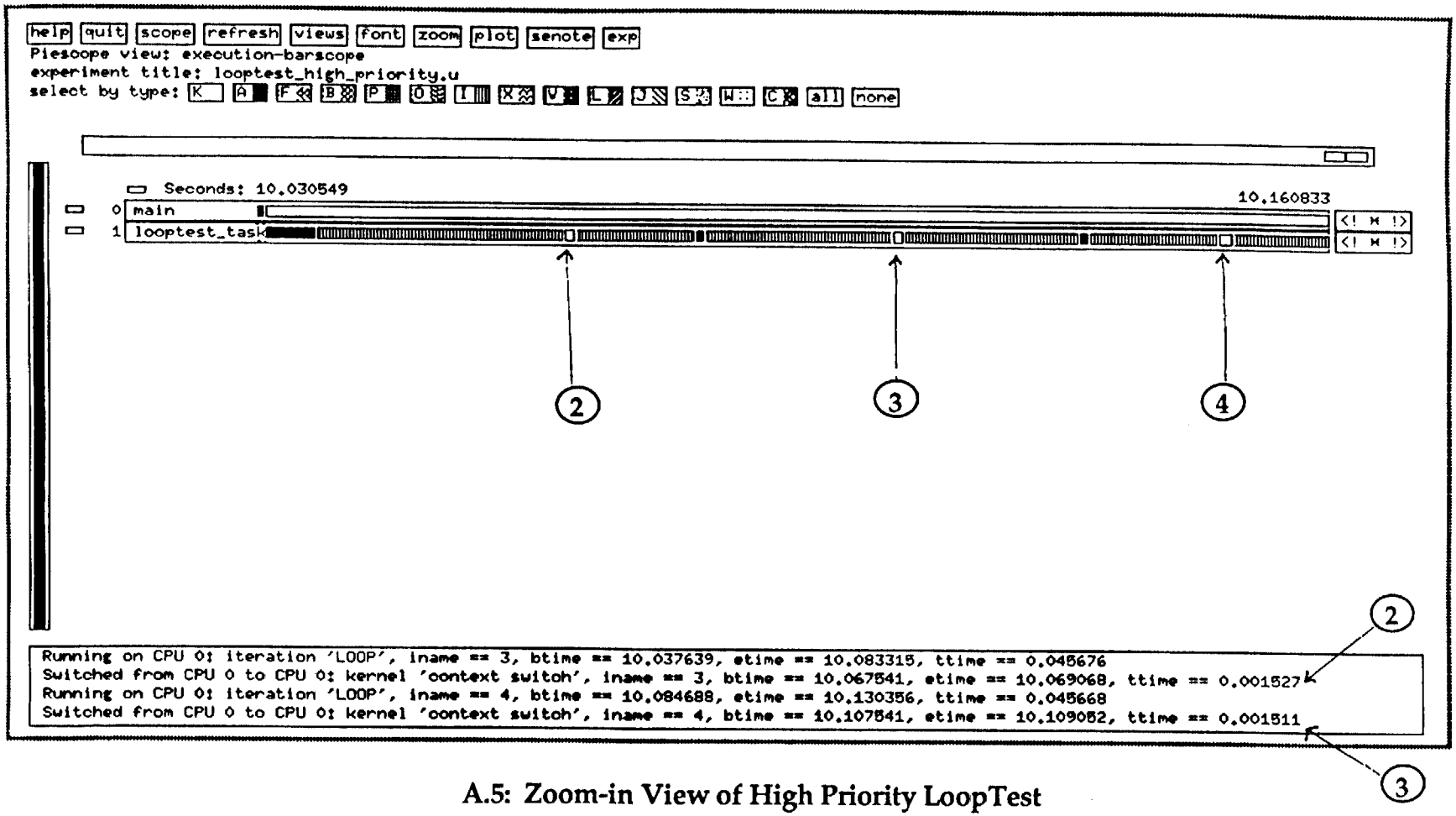
- Figure A.5, notations 2-4: PIEscope allows the user to zoom-in on a particular area of the experiment. This view is the same as Figure A.4, but has been zoomed-in so the context switches for notations 2-4 are now large enough to be visible.



A.3: High Priority LoopTest



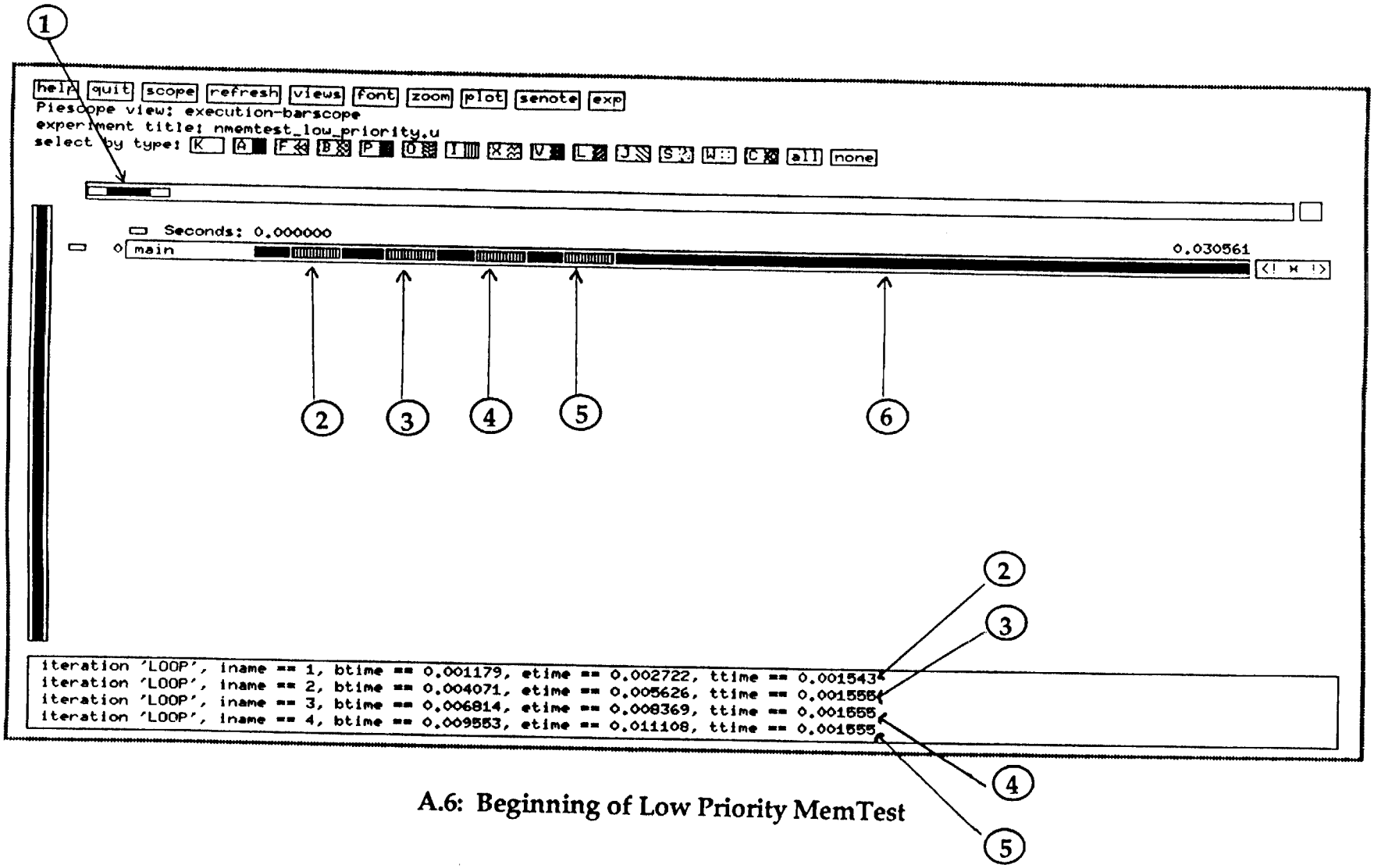
A.4: View of High Priority LoopTest Showing Context Switch Information



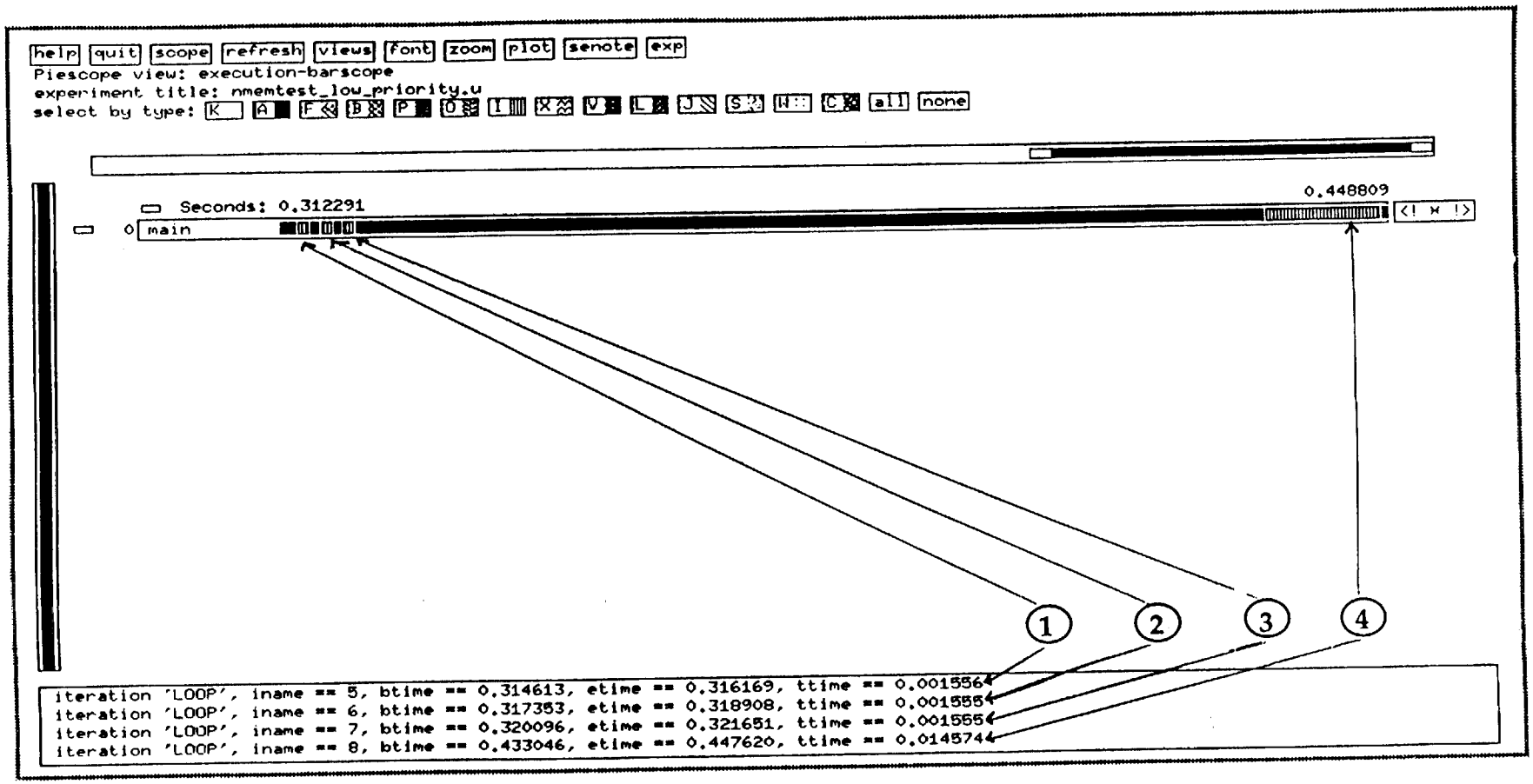
A.5: Zoom-in View of High Priority LoopTest

A.3 Low Priority MemTest

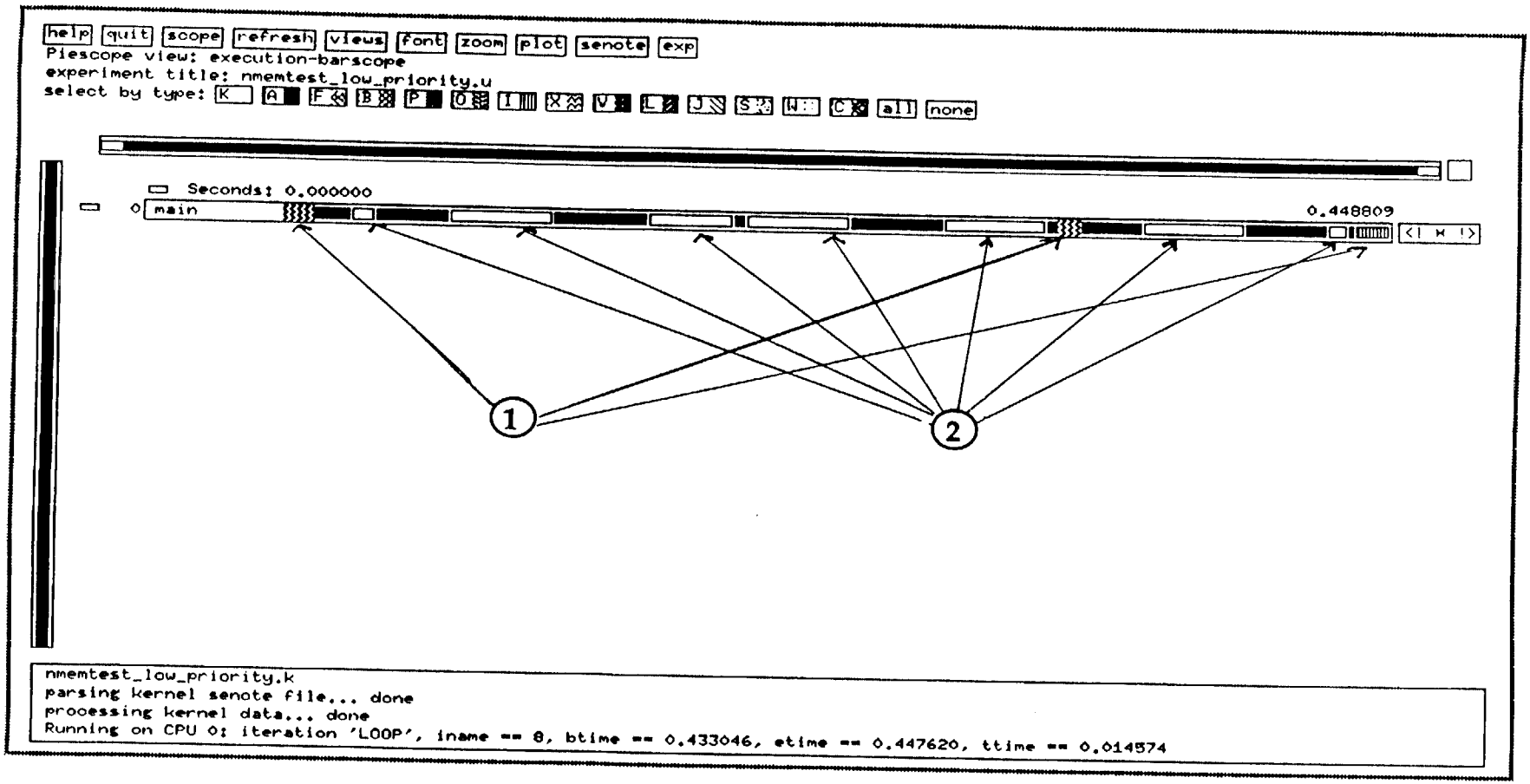
- Figure A.6, notation 1: The scrollbar indicates that we are viewing the early part of the experiment.
- Figure A.6, notation 2: The striped block indicates the period of time when the allocation of the first large block of memory was done. The execution time for that block is displayed in the window at the bottom of the view and is shown to be 0.001543 seconds.
- Figure A.6, notations 3-5: Here, the striped blocks indicate the periods of time when the first three allocations of small blocks of memory were executed. The execution times are displayed at the bottom window as each being 0.001555 seconds.
- Figure A.6, notation 6: The other small allocations were not monitored and therefore do not appear in the view.
- Figure A.7, notation 1-3: The striped blocks indicate the periods of time when the last three allocations of small blocks of memory were executed. Their execution times are displayed as being 0.001556, 0.001555, and 0.001555 seconds, respectively.
- Figure A.7, notation 4: This last block indicates the period of time when the last large allocation of memory was executed. Its execution time is displayed as being 0.014574 seconds. Note that this is nearly ten times longer than the execution time of the first large allocation as seen in Figure A.6, notation 2.
- Figure A.8, notation 1: The blocks which indicate the execution of the first large block and the subsequent small blocks of memory are too small to be displayed with the given screen resolution. The last large allocation of memory was large enough to be displayed.
- Figure A.8, notation 2: This is a view of the entire execution but now context switch information has been overlaid. We can see that large blocks of time during the execution of this experiment were spent switched out. From this data we can calculate that the CPU utilization for this experiment is 51.02%.



A.6: Beginning of Low Priority MemTest



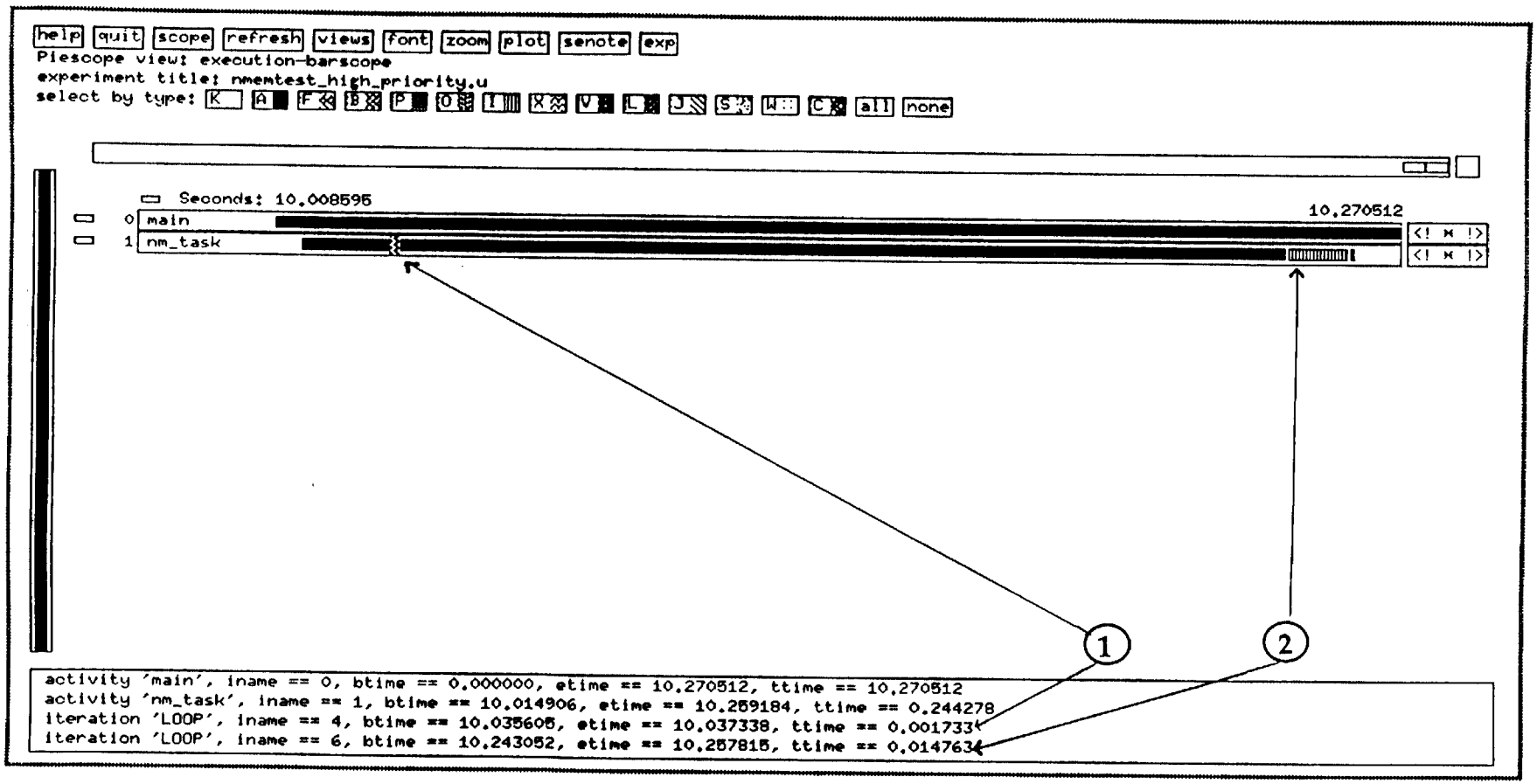
A.7: End of Low Priority MemTest



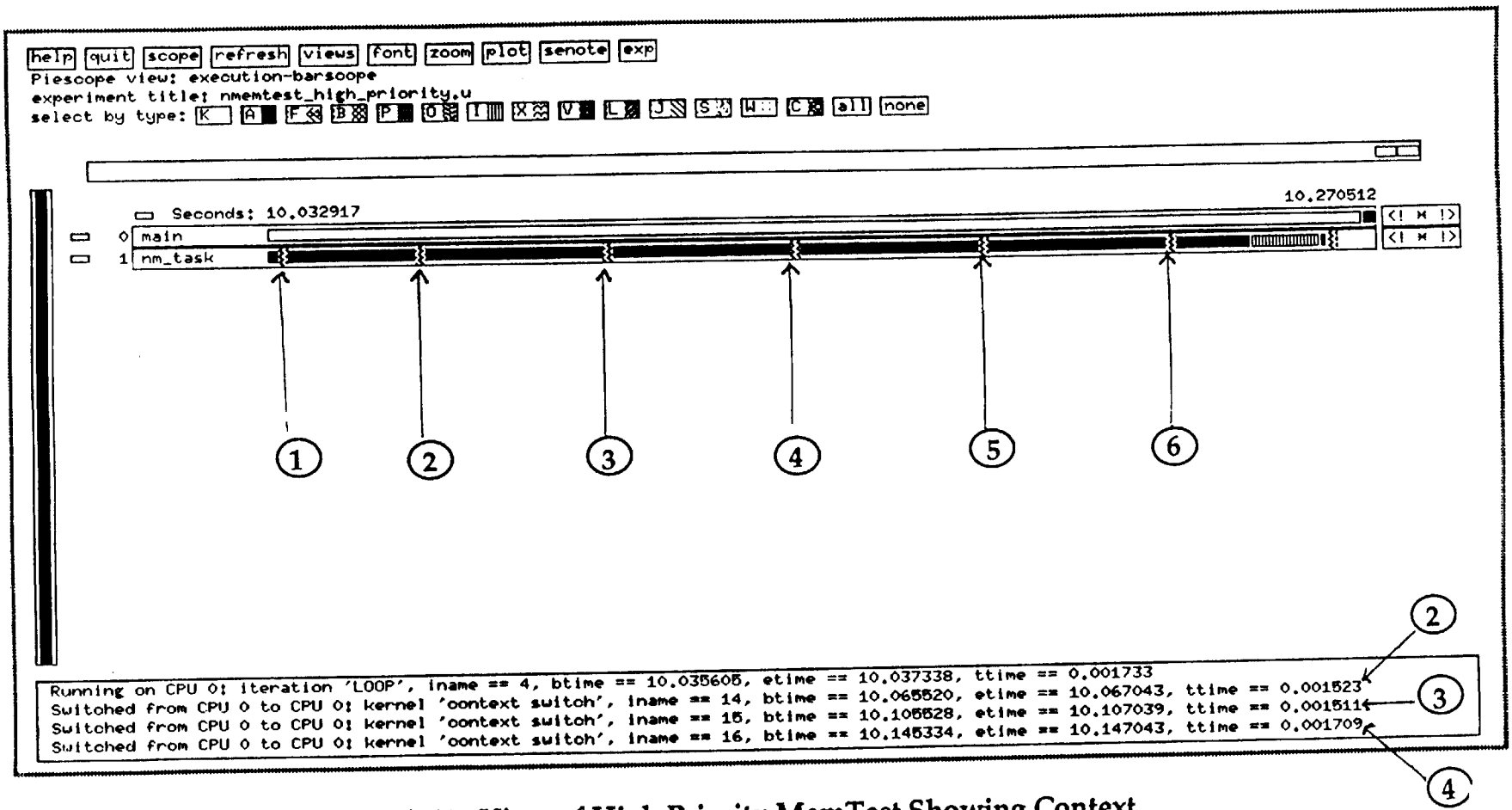
A.8: View of Low Priority MemTest Showing Context Switch Information

A.4 High Priority MemTest

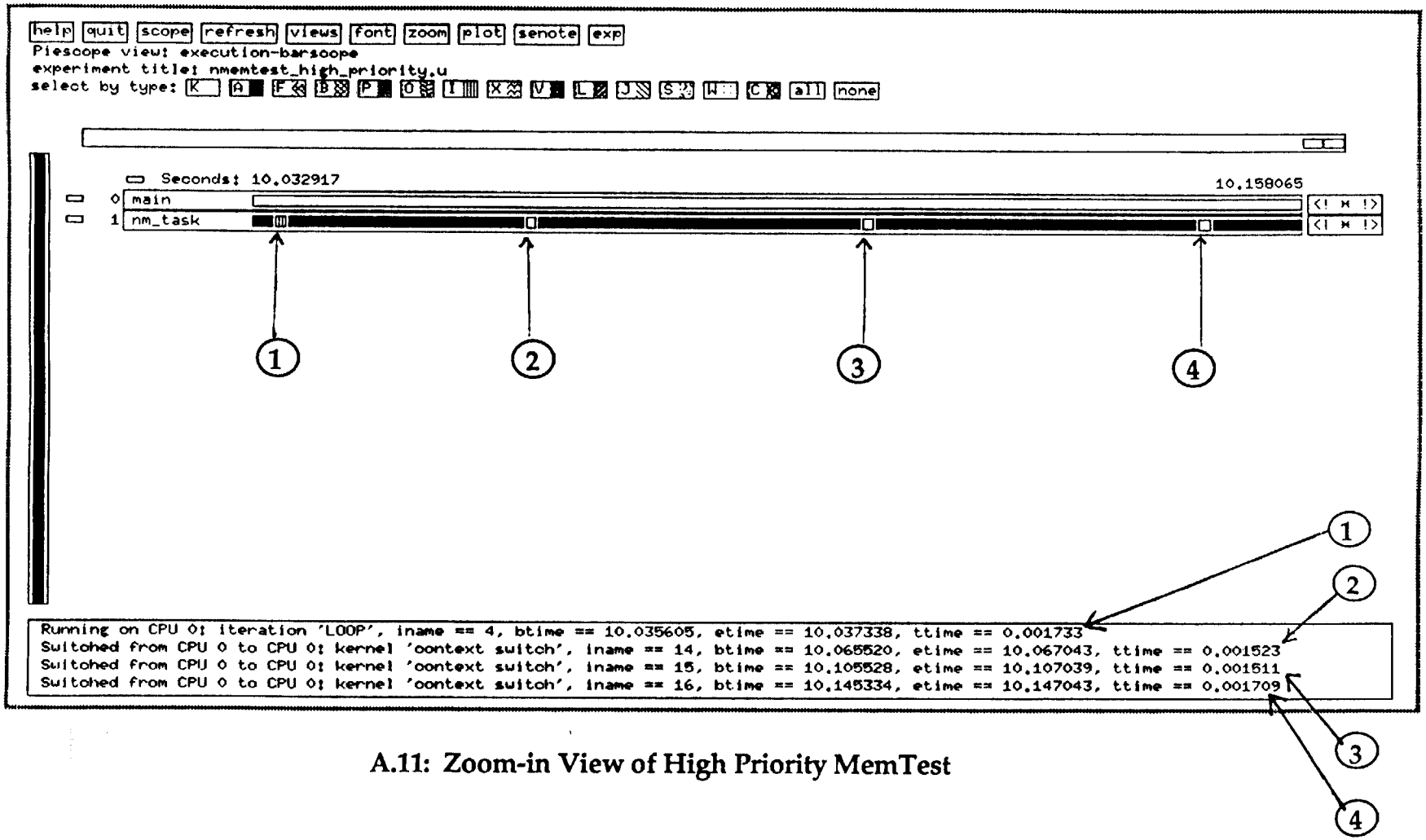
- Figure A.9, notation 1: The first allocation of a large block of memory is too small for the resolution of the screen. The execution time is displayed below as 0.001733 seconds. This is quite close to the execution time when MemTest was run under low priority.
- Figure A.9, notation 2: The last allocation of a large block of memory is displayed, and its execution time is displayed as 0.014763 seconds. This, too, is close to the execution time under low priority.
- Figure A.10, notation 1: The first allocation of a large block of memory is too small.
- Figure A.10, notations 2-6: This is the same view as in Figure A.9, but now context switch information has been overlaid. The Ada task `main` was switched out while `nm_task` did useful work. In `nm_task`, each context switch is too small to be displayed, but switch-out times for 2-4 are displayed in the bottom window as 0.001523, 0.001511, and 0.001709 seconds respectively.
- Figure A.11, notations 1 through 4: This is a zoomed-in view of the first large allocation block and the first three context switches in `nm_task`. The blocks' execution times are displayed in the bottom window.



A.9: High Priority MemTest



A.10: View of High Priority MemTest Showing Context Switch Information



A.11: Zoom-in View of High Priority MemTest

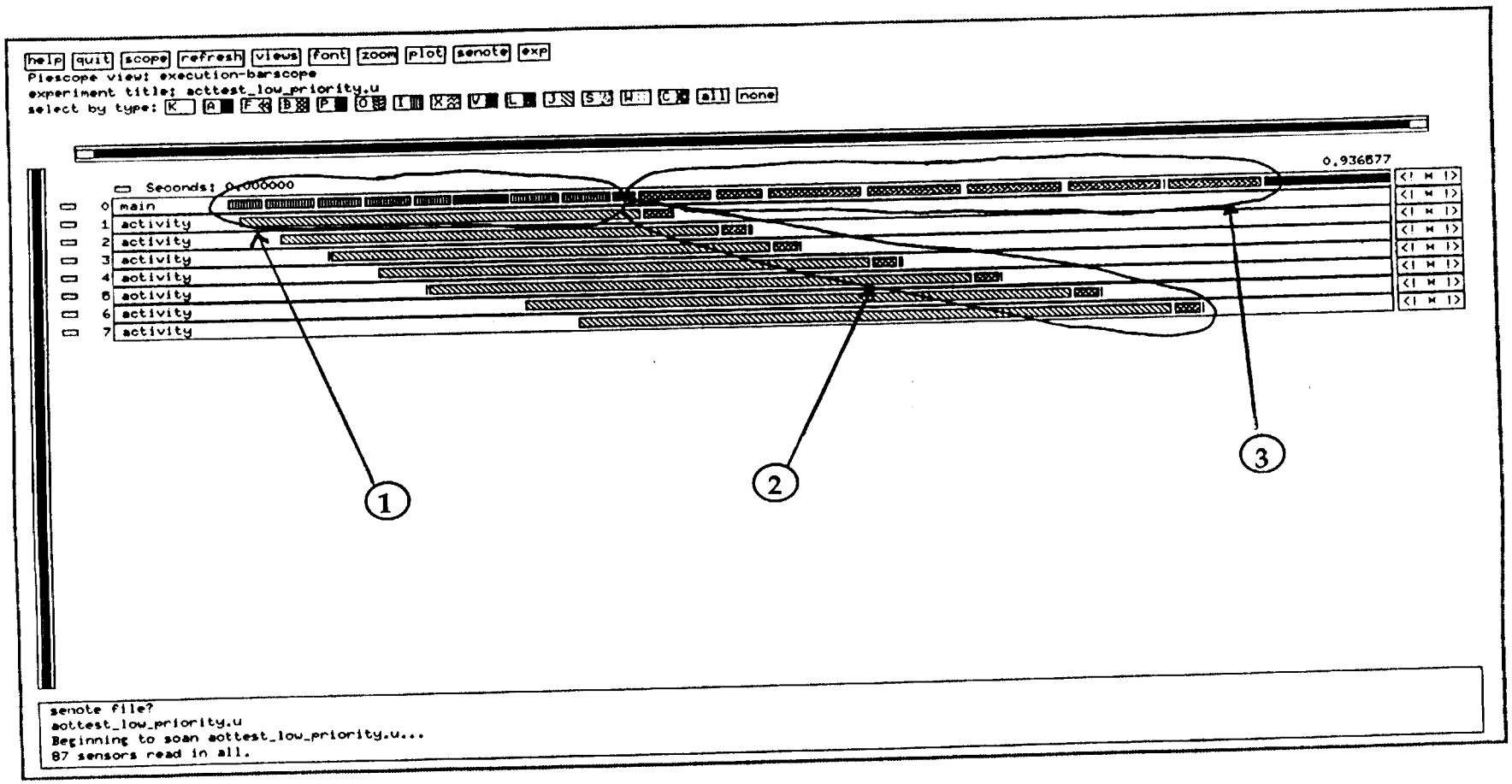
A.5 Low Priority ActTest

- Figure A.12, notation 1: The first seven striped blocks indicate the times when `main` was allocating the seven subtasks.
- Figure A.12, notation 2: The seven cross-hatched blocks (one executed in each subtask) indicate the times when the subtasks were within the ACCEPT statement in ActTest.
- Figure A.12, notation 3: The seven cross-hatched blocks running in `main` indicate the times when `main` was making a call to `my_act.start`, but before the ACCEPT was taken.
- Figure A.13, notation 1: This view shows a zoom-in on the first four task allocations in ActTest. The timings indicated by notation 1 are for the amount of time it took between when the main task called NEW and before the subtask actually began. Note that except for the allocation of the second subtask, each allocation took more time than the previous allocation, increasing in linear fashion.
- Figure A.13, notation 2: The recovery timings indicated by notation 2 are for the time it took between when the subtask was context switched out (after entering its ACCEPT) and before the main task "woke up" from its context switch, which would be the indication to the main task that the subtask had been created and main could proceed.
- Figure A.13, notation 3: The precise times are shown for the events used in the timing calculations in notations 1 and 2.
- Figure A.14, notation 1-3: This view shows a zoom-in for the last three task allocations in ActTest. The timings were calculated the same as in Figure A.13.
- Figure A.14, notation 4: Note the large context switch that occurred in `main` during neither the task allocation nor recovery part of the execution. Here, the main task was context switched out in favor of a non-application job on the system.
- Figure A.15, notation 1: This view shows a zoom-in of the first three rendezvous in ActTest. The timings indicated for notation 1 are for the amount of time it took from the main tasks call to ACCEPT until the ACCEPT as taken in the subtasks. Note that each rendezvous took an linearly increasing amount of time.
- Figure A.15, notation 2: The recovery timings indicated by notation 2 are for the time it took between when the subtask was context switched out (after finishing its ACCEPT and completing its work) and before the `main` task "woke up" from its context switch, which would be the indication to the `main` task that the subtask had finished its rendezvous and main could proceed.

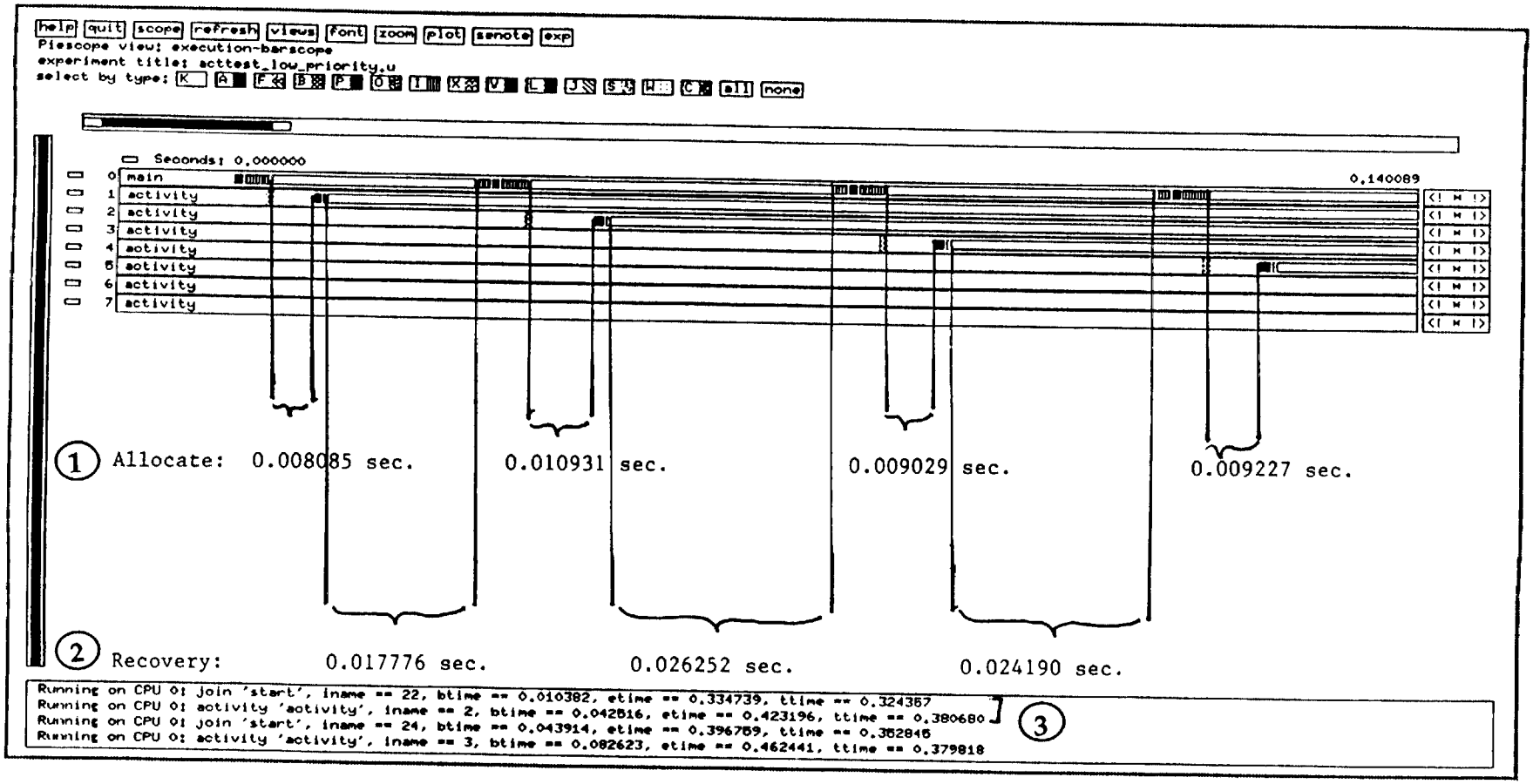
- Figure A.15, notation 3: The precise times are shown for the events used in the timing calculations in notations 1 and 2.

- Figure A.16, notations 1-3: This view shows a zoom-in for the last three task allocations in ActTest. The timings were calculated the same as in Figure A.15.

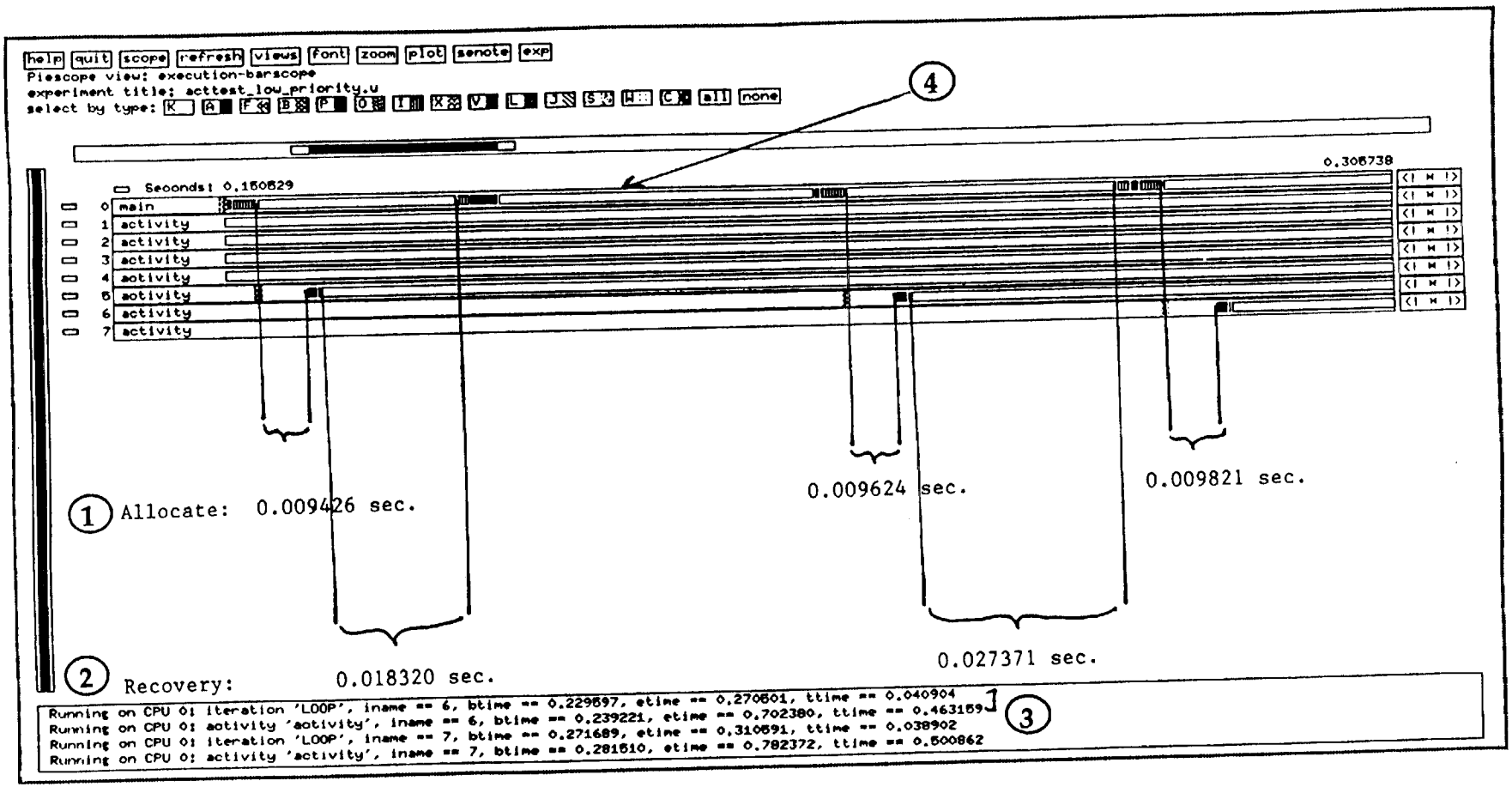
- Figure A.17, notations 1-2: This view is somewhat different than all the previous views. Here, the CPU utilization is being graphically displayed. Time is still on the X-axis in seconds, but here a black area indicates that one of our application's tasks is switched in and and a white area indicates that none of our application's tasks is switched in. In this way it is easy to visualize the overall CPU utilization as it applies to the test application. From this data the calculated CPU utilization was 49.11% for the low priority ActTest.



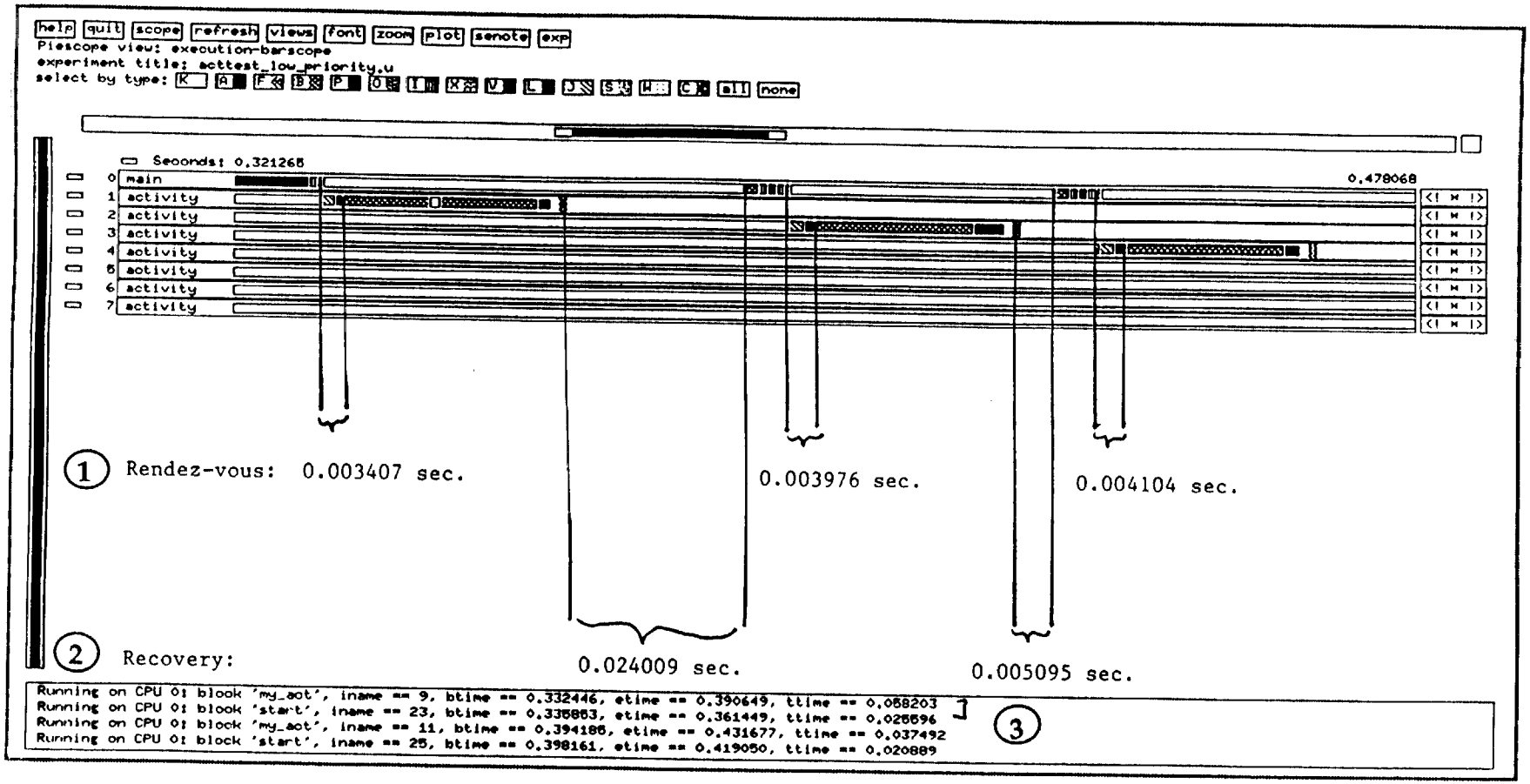
A.12: Low Priority ActTest



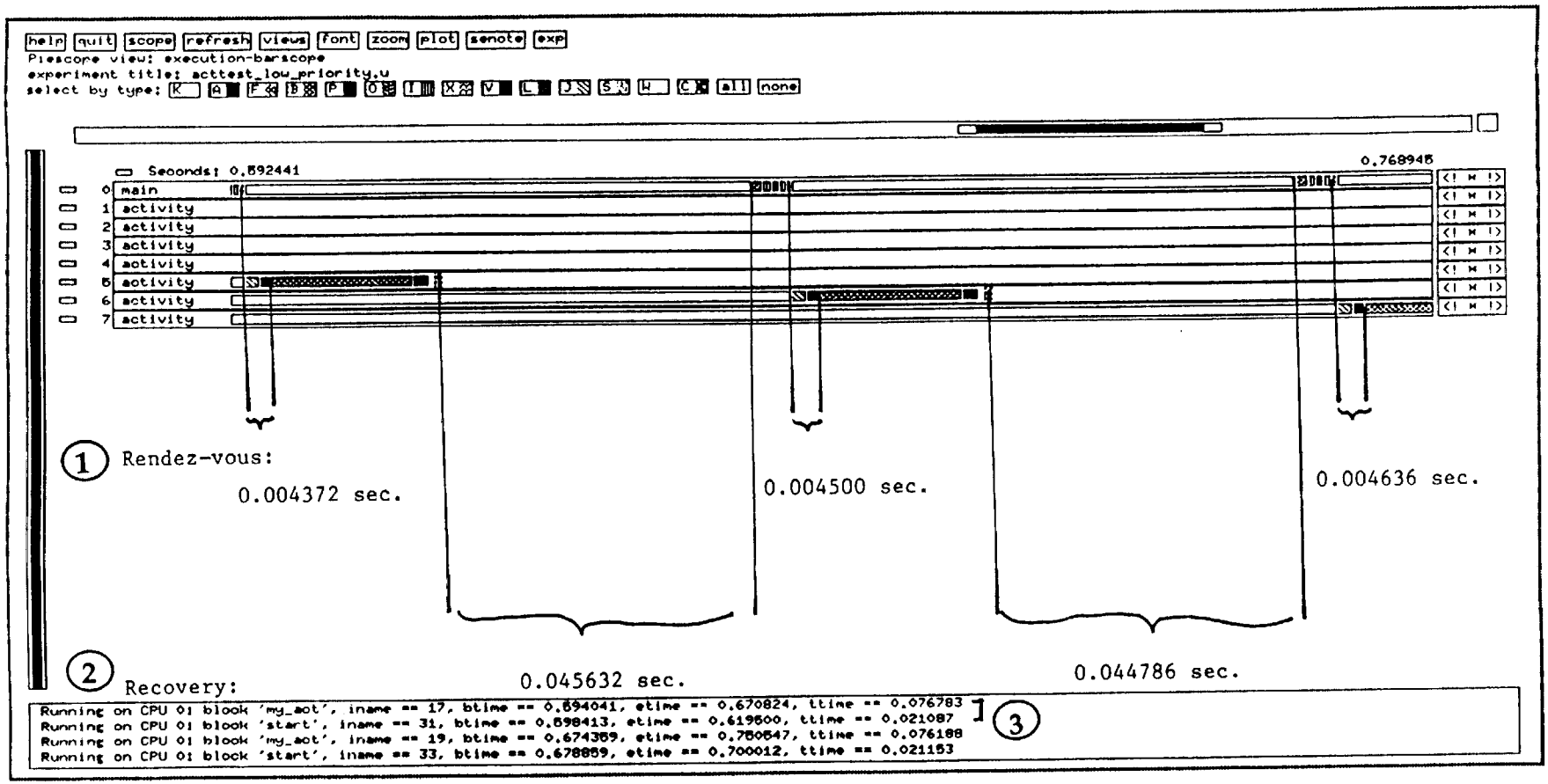
A.13: Zoom-in View of the First Four Task Allocations in the Low Priority ActTest



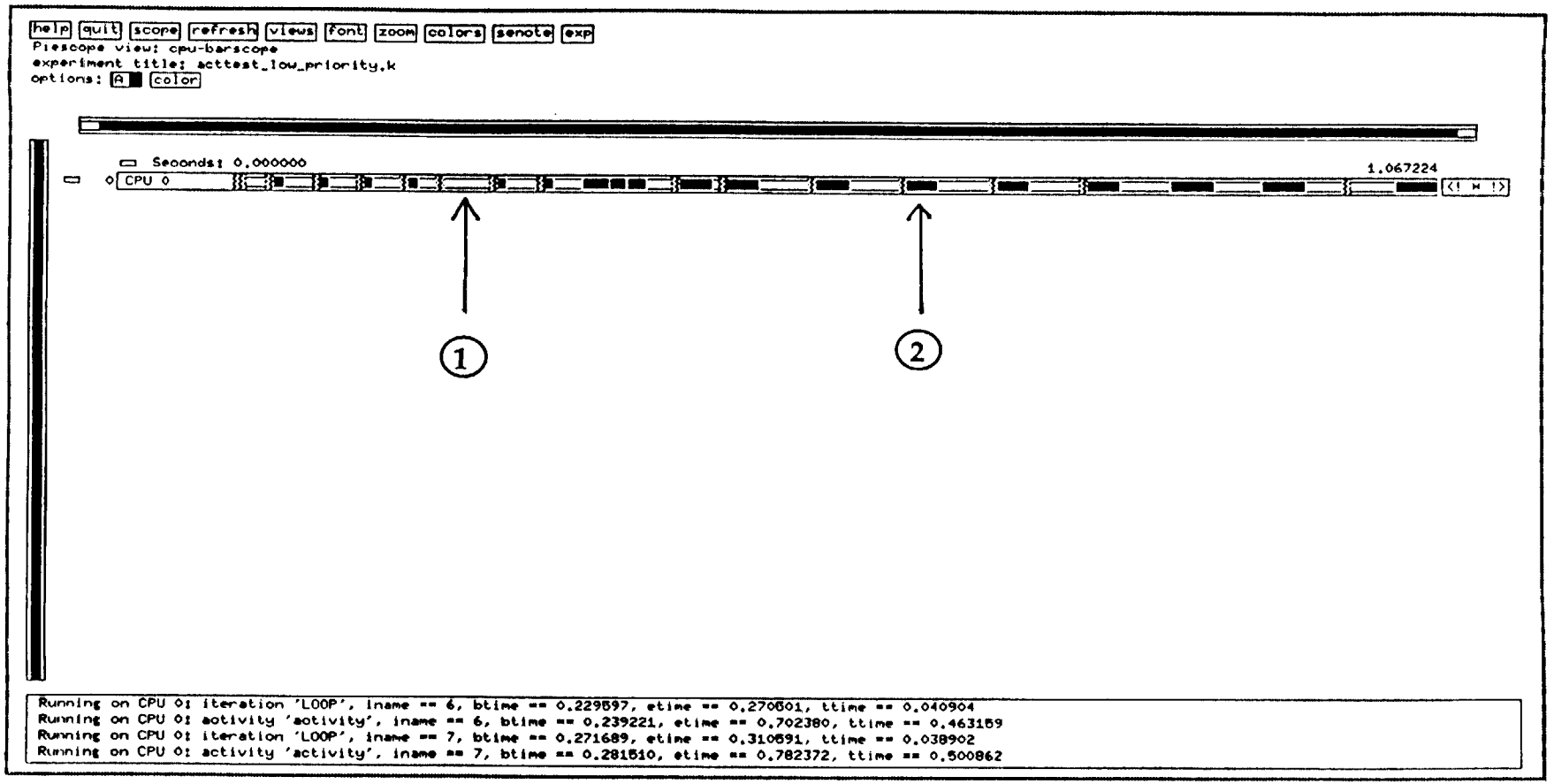
A.14: Zoom-in View of the Last Three Task Allocations in the Low Priority ActTest



A.15: Zoom-in View of the First Three Rendez-vous in the Low Priority ActTest



A.16: Zoom-in View of the Last Three Rendez-vous in the Low Priority ActTest



A.17: CPU View of the Low Priority ActTest

A.6 High Priority ActTest

- Figure A.18, notation 1: The first seven striped blocks indicate the times when main was allocating the seven subtasks.
- Figure A.18, notation 2: The seven cross-hatched blocks (one executed in each subtask) indicate the times when the subtasks were within the ACCEPT statement in ActTest.
- Figure A.18, notation 3: The seven cross-hatched blocks running in main indicate the times when main was making a call to `my_act.start`, but before the ACCEPT was taken.
- Figure A.18, notation 4: Note the unusual behavior of activities 3, 5, and 6 after they have finished their final rendezvous. instead of exiting relatively soon they actually finish quite a bit later. Figures A.21 and A.22 show that these activities took context switches which prevented them from exiting when they normally would have.
- Figure A.19, notation 1: This view shows a zoom-in on the first four task allocations in ActTest. The timings indicated by notation 1 are for the amount of time it took between when the `main` task called NEW and before the subtask actually began. Note that except for the allocation of the second subtask, each allocation took more time than the previous allocation, increasing in linear fashion.
- Figure A.19, notation 2: The recovery timings indicated by notation 2 are for the time it took between when the subtask was context switched out (after entering its ACCEPT) and before the main task "woke up" from its context switch, which would be the indication to the main task that the subtask had been created and main could proceed. Unlike the low priority example, these recovery times are instantaneous.
- Figure A.19, notation 3: The precise times are shown for the events used in the timing calculations in notations 1 and 2.
- Figure A.20, notation 1-3: This view shows a zoom-in for the last three task allocations in ActTest. The timings were calculated the same as in Figure A.19.
- Figure A.21, notation 1: This view shows a *zoom-in* of the first three rendezvous in ActTest. The timings indicated for notation 1 are for the amount of time it took from the main tasks call to ACCEPT until the ACCEPT as taken in the subtasks. Note that each rendezvous took an linearly increasing amount of time. Figure A.21, notation 2: The recovery timing indicated by notation 2 is for the time it took between when the subtask was context switched out (after finishing its ACCEPT and completing its work) and

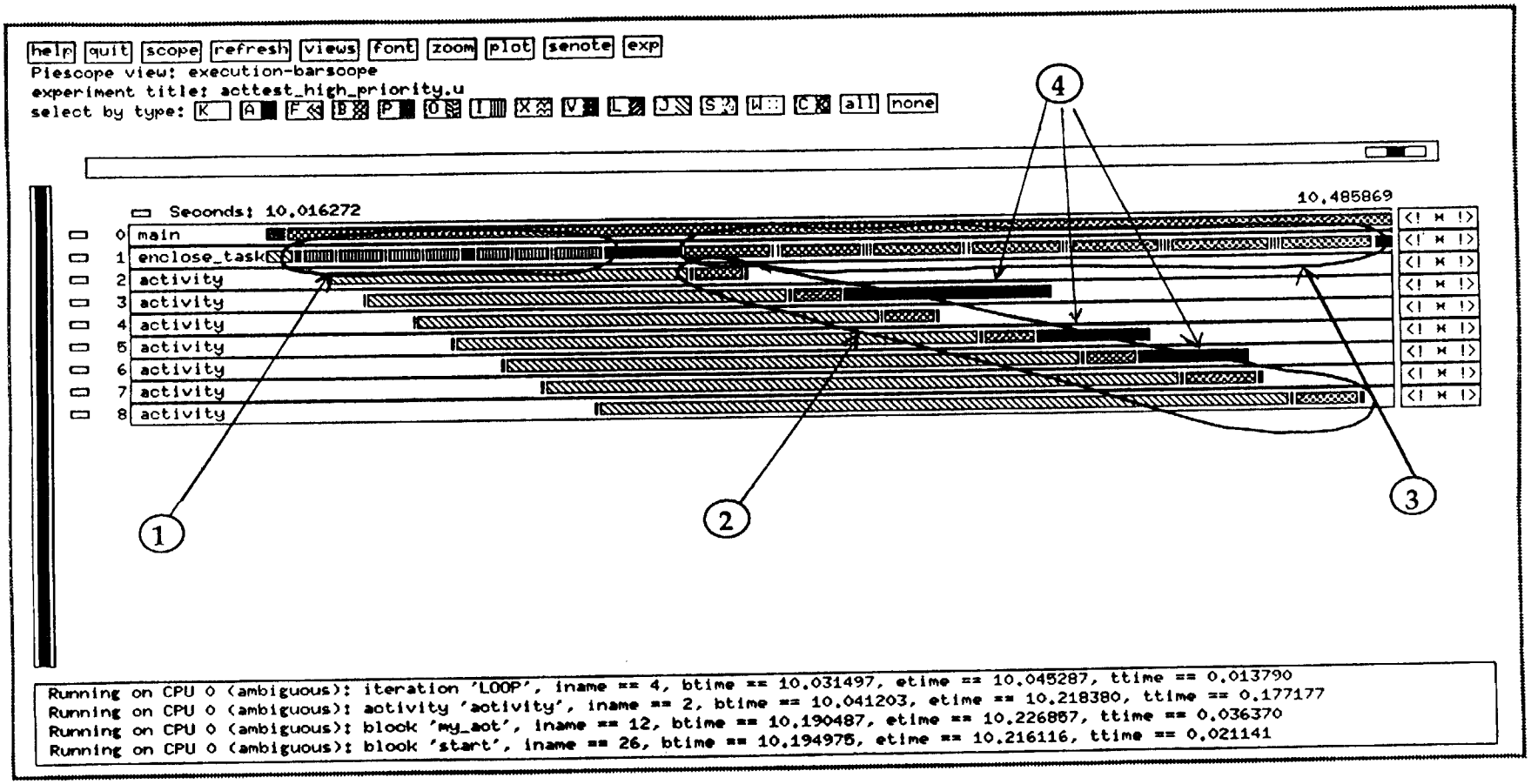
before the main task "woke up" from its context switch, which would be the indication to the main task that the subtask had finished its rendezvous and main could proceed.

- Figure A.21, notations 3-4: Note that these recovery times probably would have been instantaneous (as in Figure A.19, notation 2) except for the context switches that occurred in activity 3 before it exited. As a result, the recovery was interfered with by activity 3 performing extra execution.

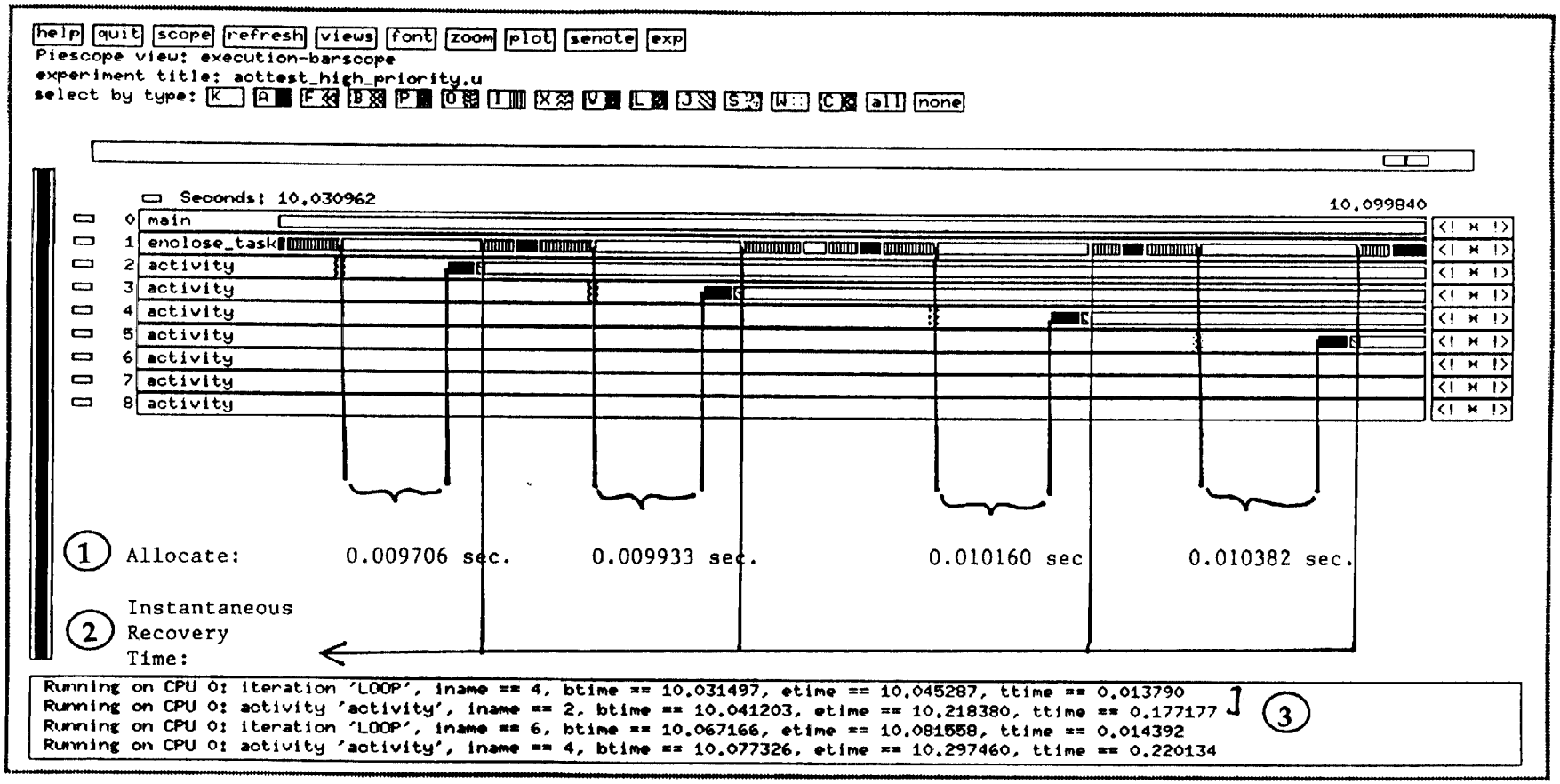
- Figure A.21, notation 5: The precise times are shown for the events used in the timing calculations in notations 1 and 2.

- Figure A.22, notations 1-4: This view shows a *zoom-in* for the last three task allocations in ActTest. The timings were calculated the same as in Figure A.21. Note that the recoveries after activities 6 and 7 finish are interfered with by the extraneous context switching at the end of activities 5 and 6. The recovery after activity 8 is done is again instantaneous.

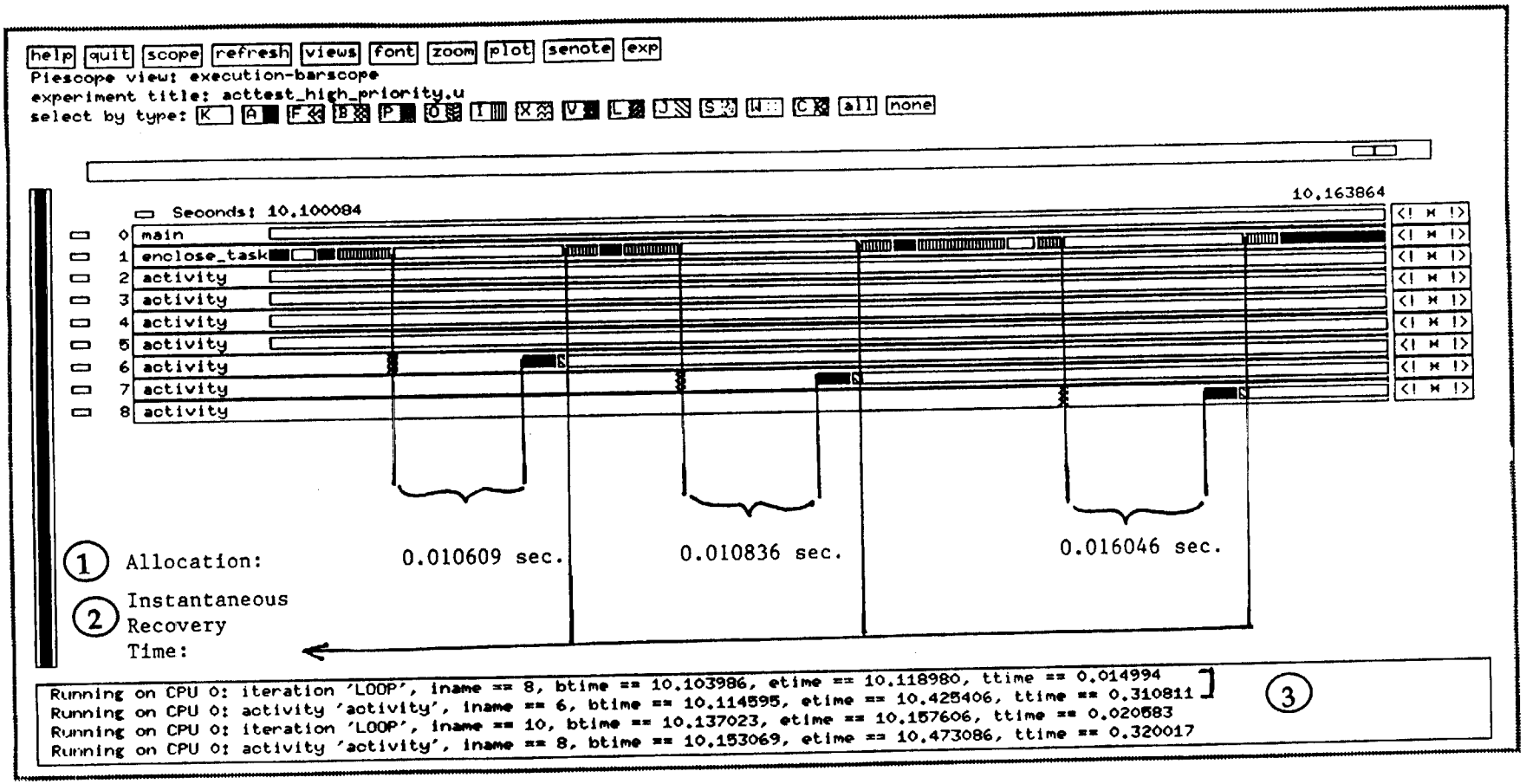
- Figure A.23, notations 1-2: This view shows CPU utilization as in Figure A.17. With far less context switching than in the low priority example, the calculated CPU utilization was 96.38% for the high priority ActTest.



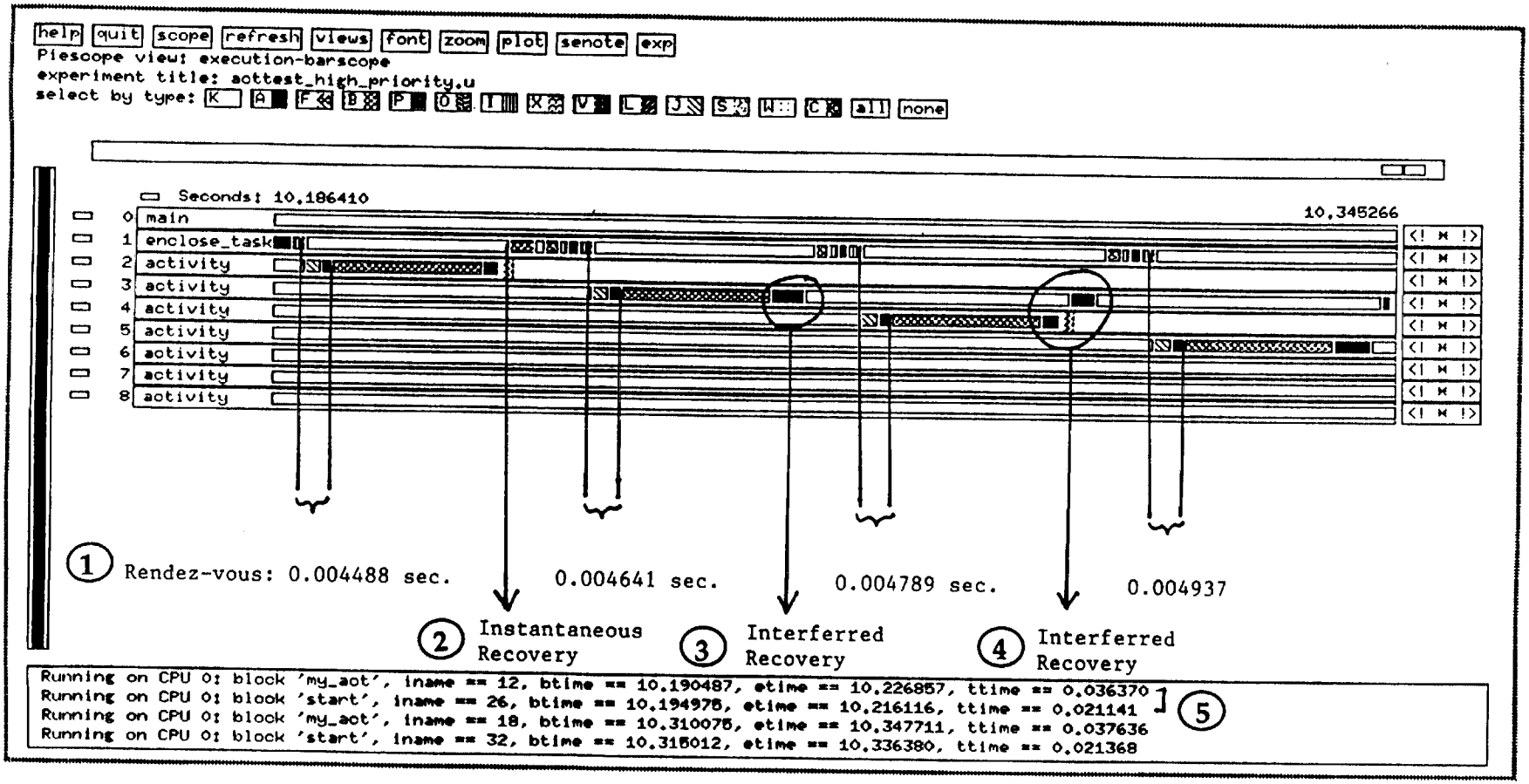
A.18: High Priority ActTest



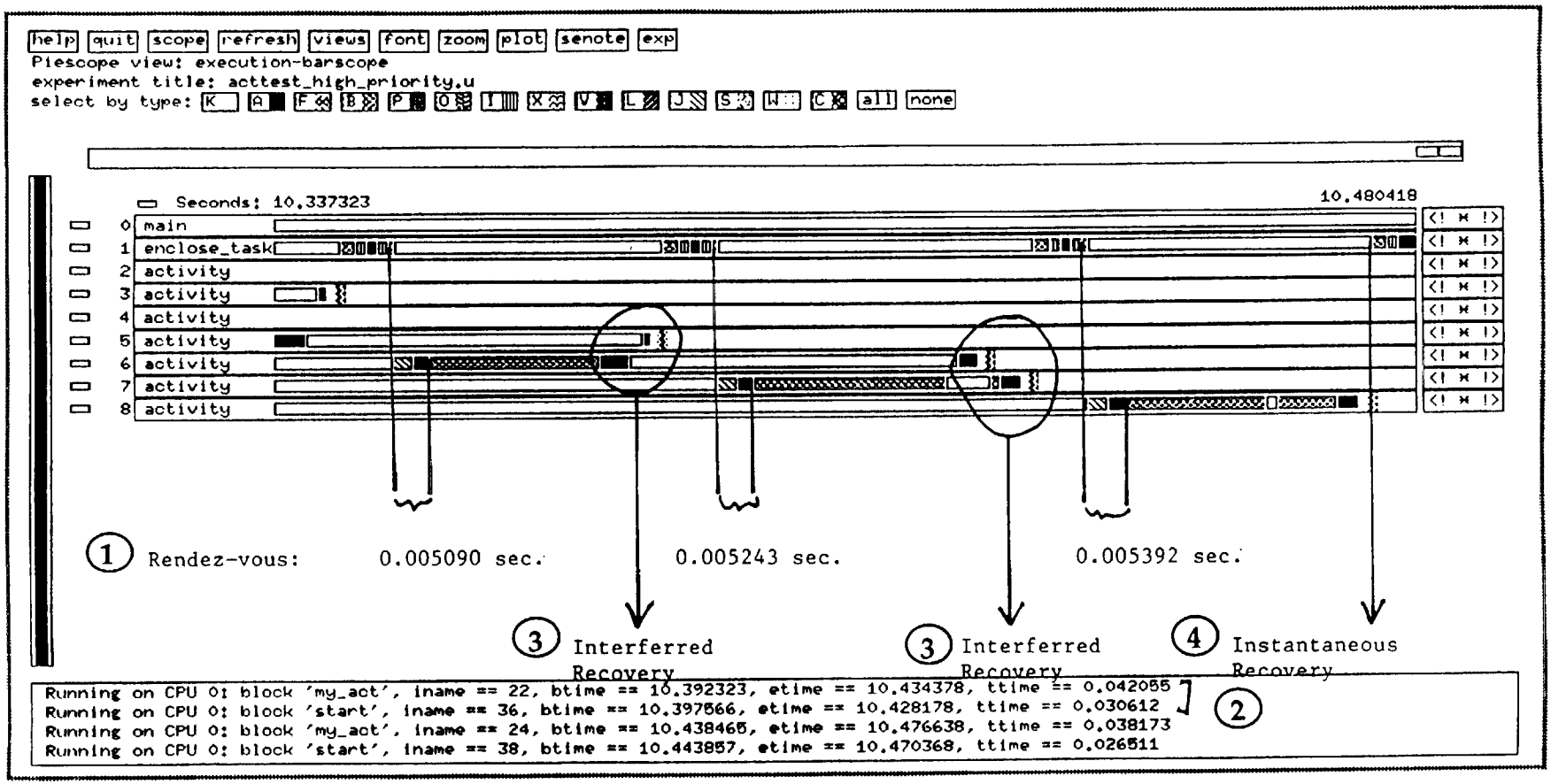
A.19: Zoom-in View of the First Four Task Allocations in the High Priority ActTest



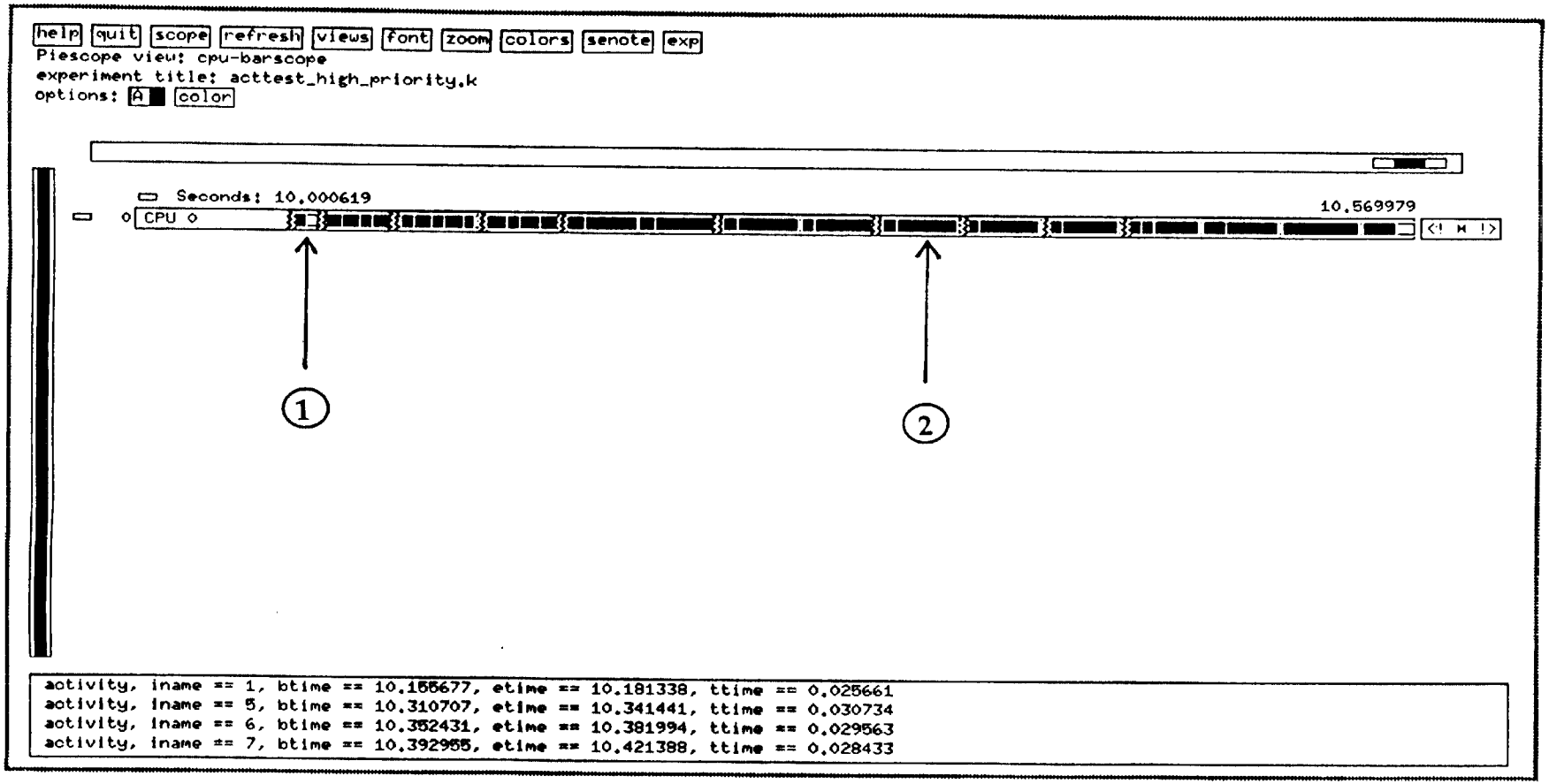
A.20: Zoom-in View of the Last Three Task Allocations in the High Priority ActTest



A.21: Zoom-in View of the First Three Rendez-vous in the High Priority ActTest



A.22: Zoom-in of the Last Three Rendez-vous in the High Priority ActTest



A.23: CPU view of the High Priority ActTest

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-4340		

4. Title and Subtitle
Validation Environment for AIPS/ALS: Implementation and Results

5. Report Date
November 1990

6. Performing Organization Code

7. Author(s)
Zary Segall, Daniel Siewiorek, Eddie Caplan, Alan Chung, Edward Czeck, and Dalibor Vrsalovic

8. Performing Organization Report No.

10. Work Unit No.
506-46-21-05

9. Performing Organization Name and Address
Carnegie-Mellon University
Electrical and Computer Engineering Department
Schenley Park
Pittsburgh, PA 15213

11. Contract or Grant No.
NAG1-190

13. Type of Report and Period Covered
Contractor Report
11/88-11/90

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

14. Sponsoring Agency Code

15. Supplementary Notes
Langley Technical Monitor: Peter A. Padilla
Final Report

16. Abstract

This is the final report of work done under contract NAG-1-190. This document presents the work performed in porting the FIAT and PIE validation tools, developed at Carnegie-Mellon University, to the AIPS system in the context of the ALS application, as well as an initial fault-free validation of the available AIPS system. The PIE components implemented on AIPS provide the monitoring mechanisms required for validation. These mechanisms represent a substantial portion of the FIAT system. Moreover, these are required for the implementation of the FIAT environment on AIPS. Using these components an initial fault-free validation of the AIPS system was performed.

This report describes the implementation of the FIAT/PIE system, configured for fault-free validation of the AIPS fault-tolerant computer system. The PIE components have been modified to support the Ada language. A special purpose AIPS/Ada runtime monitoring and data collection has been implemented. A number of initial Ada programs running on the PIE/AIPS system have been implemented. The instrumentation of the Ada programs was accomplished automatically inside the PIE programming environment. PIE's on-line graphical views show vividly and accurately the performance characteristics of Ada programs, AIPS kernel and the application's interaction with the AIPS kernel. The data collection mechanisms were written in a high-level language, Ada, and provide a high degree of flexibility for implementation under various system conditions.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
AIPS ALS FIAT PIE Fault Tolerance Validation Fault Insertion	Unclassified-Unlimited Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	80	A05

NASA FORM 1626 OCT 86

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171

NASA-Langley, 1990