An Introduction to Visualizing Program Execution Dynamics with Chitra

Marc Abrams and Naganand Doraswamy

TR 92-23

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

May 14, 1992

An Introduction to Visualizing Program Execution Dynamics with Chitra

Marc Abrams
Naganand Doraswamy
TR 92-23, May 14, 1992
Virginia Polytechnic Institute and State University
Department of Computer Science
Blacksburg, VA 24061-0106

Abstract

We describe a tool called Chitra, which serves two purposes. First, Chitra visualizes program execution sequences (PES's) collected by monitoring parallel and distributed program execution in the time, event, and frequency domains. Second, Chitra produces an empirical model of a PES. Using a mouse, a user edits a visualization of a PES to transform the PES to a simpler representation in terms of fewer states. The user can interactively define aggregations and filters of program states. Chitra uses the edited PES to construct a homogeneous, semi-Markov stochastic process to fit the PES. A single program state is represented by one or more stochastic process states so that Chitra can embed history into a stochastic process state. This allows the assumption of an embedded Markov chain without a penalty in model accuracy, even though software generally violates the assumption. A case study applies Chitra to predict and evaluate the efficiency and fairness of a resource sharing algorithm at a range of parameter values, given observations of a few parameter values. Inefficient behavior is explained by examining the states that constitute the stochastic process model constructed by Chitra.

1 Introduction

Computer visualization is the process of investigating and modeling a physical system though graphical images. Visualization capitalizes on the perceptive capabilities of human vision, facilitating discoveries about a system that would be unlikely using only textual data or theory.

Software visualization can be used both to understand program behavior and to improve program performance. Understanding program behavior assesses the effect of changes in pro-

gram parameters on the sequence in which operations are performed during program execution. Improving program performance requires identifying program modifications that will improve a performance metric, such as reducing the time required for execution.

Many software visualization tools have been built; their capabilities include program execution animation [3, 15], data structure animation [16, 5], display of interprocess communication [7, 15], program debugging [12, 8, 6], and performance evaluation [13, 10, 11, 6]. Nevertheless, visualization has had limited impact in analyzing software. The explanation for this, we believe, is that visualization dramatically impacts fields where it leads to a discover of unexpected phenomena. For example, strength of materials engineers recently discovered through visualization features of stress wave propagation that are not predicted by solutions to the wave equation. Therefore software visualization must do more than generate interesting pictures. It must ultimately lead the user to construction of an empirical model of behavior, which can serve as a target for theoreticians to derive. That is the objective of *Chitra*[1], the visualization system described here.

1.1 Program Execution Sequences

Chitra uses a *program execution sequence* (PES) as its basic representation of data collected by monitoring program execution. A PES is defined as follows.

A parallel or distributed program consists of a (possibly empty) set of *global variables*, a (possibly empty) set of channels between processes, and a set of *processes*, each of which is associated with a (possibly empty) set of *local variables*. A process (or thread) is a piece of code that is scheduled by an operating system. A *program state* is an instantaneous description of the values of all local and global variables, the sequence of data in each channel, and a pointer to the instruction which will next complete its execution in each process.

[1] Chitra is a sanskrit word for beautiful or pleasing pictures and drawings.

For each execution of a program, there corresponds a *program execution sequence* (PES), which represents the sequence and occupancy times of program states though which a program passes during execution.

In practice program monitors generally have insufficient bandwidth to obtain the entire PES. In addition, for the purpose of program performance analysis, only a subset of each program state is needed. Hence we will assume that the PES output by a program monitor will produce a time dependent sequence, each element of which is a subset of a program state. Henceforth we use the term *program state* to refer to the subset of interest.

For example, a PES for communication protocol software implementing TCP/IP might represent the sequence of advised window sizes for a connection. A PES for a synchronous, parallel discrete event simulation program might contain, as program states, a record of how many processes are waiting for the next synchronization barrier. Each program state in a PES for a program solving Dijkstra's dining philosophers problem might represent whether each philosopher is thinking, waiting for utensils, eating, or releasing utensils.

1.2 Conjecture Underlying Chitra

Each invocation of a synchronization or interprocess communication primitive in a parallel or distributed program introduces a microscopic rule governing the interaction of processes. Our interest is in examining how the set of microscopic rules defined by the programmer together produce a macroscopic behavior. It is then up the the theoretician to derive the macroscopic behavior given a description of the microscopic rules. At the basis of Chitra lies the following conjecture:

> Execution of a parallel or distributed program is governed by the interaction of two behaviors: random and deterministic. Sometimes random behavior dominates,

and sometimes deterministic prevails. Therefore the macroscopic program behavior, expressed as a PES, spans a continuum from purely random to purely deterministic behavior.

Randomness arises from sources such as contention of multiprogrammed processes for resources (e.g., processor cycles, cache lines, memory blocks, communication media), nondeterministic programming language constructs, asynchrony among processor clocks, and interrupts due to asynchronous sources outside the computer system. Determinism arises from activities that require a constant wall clock time and a process synchronization structure that produces feedback. Randomness could prevail in a parallel processor executing several unrelated parallel programs. In contrast, as will be shown in Section 3, in a parallel processor executing a single parallel program whose non-branching activities chiefly require constant time and whose synchronization structure produces feedback, determinism could prevail. We note that the tendency of a PES to be random or deterministic is also dependent on the program inputs.

1.3 Dynamic System View of Software

Much engineering analysis is based on dynamic systems theory. This theory treats a system as a "black box," perhaps characterized by certain parameters, and given observations of certain input signals and their corresponding output signals, provides a means to predict the system response to an arbitrary input signal. Can parallel programs be analyzed using this paradigm? Can any analysis techniques from dynamic systems be directly applied to parallel program analysis?

Chitra attempts to answer both of these questions. Therefore Chitra views a parallel program as a black box. At present Chitra assumes that the system representing a program has parameters that may assume various values, but no input function. The output signal is the PES. Our problem is to predict the output signal (PES) as a function of the parameter values, based on

observation of a finite number of output signals.

There are several motivations for exploring a dynamic system view. First, our conjecture that program behavior, expressed as a PES, spans a continuum from purely deterministic to purely random, has a dual in dynamic systems theory, where a signal may be deterministic (in which case there exists a function that generates the signal) or random (and hence there exists a stochastic process that predicts the signal). Second, the concept of noise in a signal is analogous to random perturbations in the normal PES that a program generates, for example due to the execution of an interrupt handler that suspends a process or servicing of page faults. Third, signal analysis techniques may be applied to the PES to analyze program behavior. In particular, the signal can be viewed not only in the time domain, but also in the frequency domain. Viewing a PES in the frequency domain helps reveal the existence of repeated state transition sequences. Filters in the frequency domain can be defined for PES's; for example periodic clock interrupts can sometimes be filtered out of a PES. Finally, the dynamic system viewpoint may suggest new avenues of research to theoreticians. For example, the superposition property of linear systems aids in their analysis. Can a superposition property be defined for a class of programs, so that we can predict from theory how a PES will be modified by addition of a new synchronization point or process?

1.4 Chitra Versus Other Performance Visualization Tools

Visualization of data from program monitors is not a new idea. Visualization tools suitable for performance evaluation that are listed above include Moviola [6], IPS-2 [13], JED [9], HyperView [10], and an unnamed tool [14]. Moviola displays the time spent in communication as a function of time and the time spent waiting for certain activities to identify bottlenecks. IPS-2 performs critical path analysis to determine the procedures or segments of code that must be

modified to reduce program running time. IPS-2 also provides a variety of performance data about parallel program execution. JED supports event trace management, event trace display, and event query, and allows the user to extend analysis and display functionality. HyperView allows a user to browse through a trace of system and application execution. Sarukkai's tool [14] is the closest to Chitra, in that it also predicts the performance of programs by building a model, although it uses an analytic model, whereas Chitra uses a stochastic process as a model.

Chitra is unique in that it is the only tool to generate an empirical model as the culmination of visualization. To support this objective, Chitra provides several innovations in visual analysis of monitor data. First, Chitra supports three views of a PES: in the time, event, and frequency domain. The frequency domain view is novel. Chitra provides filtering to reduce perturbations in a PES (corresponding to noise in a signal) that reveal patterns that underly a PES; the filtering can be done in the time, event, or frequency domain. Chitra guides the user in defining filters by providing statistical information about the PES displayed. Given our definition of program state, the state space of a program grows exponentially with the number of processes. Therefore Chitra provides several means of aggregating states to reducing the state space to a manageable size. Chitra provides a static aggregation method, in which the user applies knowledge about the program gained without examining any PES, and dynamic aggregation, which the user defines based on examining individual PES's. When the user has filtered and aggregated a PES to reduce it to a sufficiently compact form for the analyst, Chitra generates a discrete state, continuous time semi-Markov process model of the PES.

When we began designing Chitra, we envisioned a visualization tool that presents a PES in the time and event domains, and which allows reduction of a PES through dynamic aggregation. The other features listed in the preceding paragraph were gradually incorporated into Chitra as a result of analyzing many PES's.

The remainder of the paper is organized as follows. The next section describes the features of Chitra. Section 3 contains a case study, applying Chitra to a parallel program. Finally Section 4 discusses directions for further development of Chitra.

2 Visualization with Chitra

Chitra has two objectives:

1. to provide a means to view and transform a PES to reveal any underlying patterns or structure that distinguishs deterministic from random behavior, and
2. to generate an empirical model that fits the set of observed PES's.

First the choice of empirical model is discussed in Section 2.1. Then Section 2.2 describes the views and transformations comprising objective 1. Then we describe in Section 2.3 our approach to objective 2, which is to replace repeatedly each deterministic pattern as well as each random alternation of program states in a PES by an aggregate state, and then model the resulting PES as a stochastic process.

Chitra is invoked with a file name as an argument. This file is called the *input file*, and it describes a PES. In particular it contains a sequence of program states in ascending time stamp order, along with ancillary information such as a character string name identifying each program state and the *time base* of the time stamps. If the time base is microseconds, then a time stamp of 1000 represents a millisecond.

2.1 Choice of Empirical Model

The conjecture from Section 1.2 requires the empirical model of program behavior that Chitra constructs to be capable of modeling both programs in which randomness and in which determinism prevails. The model chosen is a continuous time semi-Markov process with a discrete

state space. At present we use a homogeneous process, which is sufficient for the case study in Section 3.

A stochastic process state is distinct from a program state. Each stochastic process state in the state space is an *aggregate state*. An aggregate state is recursively defined as either (1) a program state, (2) a random selection of two or more aggregate states, or (3) a sequence of two or more aggregate states. This definition is a key to why we can assume that software has an embedded Markov chain. Section 2.3.3 defines how the rates out of a stochastic process state are calculated based on what program states are aggregated to form the stochastic process state.

A semi-Markov process permits general state occupancy times. However it also requires an assumption of an embedded Markov chain. Program codes contain conditional branches whose target is selected based on the value of a complex function of input data to the program and initial values of program variables. In contrast the embedded Markov chain assumption bases all transitions in the stochastic process on a random choice based only on the current state. Can we limit history to the current state?

We argue in the affirmative, but our argument arises because we construct the chain in a manner that implicitly encodes history into each stochastic process state. For example, let $a, b,$ and c denote program states, and let b_0 and b_1 denote stochastic process states. If in a PES under study the only transitions into or out of state b are a, b, a and c, b, c, then program state b can be represented by two stochastic process states, denoted b_0 and b_1. State b_0 always has a as a predecessor and successor, and state b_1 always has c as a predecessor and successor. Therefore b_0 and b_1 implicitly encode their predecessor state. In contrast, in the Markov chain representation that typically appears in the literature, each program state is represented by exactly one stochastic process state, so that the successor to stochastic process state b is randomly chosen to be either a or c rather than being deterministically selected to be the predecessor state

of b.

Using Chitra, one can visually identify deterministic state transition sequences that occur throughout a PES; Chitra combines each sequence into an aggregate state, and modifies the PES to use the aggregate states. In addition, if the state sequences between any pair of aggregate states appears random, then Chitra will automatically generate one aggregate state representing an acyclic stochastic process. Each path through the process corresponds to one sequence of states occurring between a pair of aggregates observed in a PES. The final PES one obtains with Chitra is a set of aggregate states that forms a random process. Each aggregate state in turn represents either a random process whose states are aggregates or program states, or a deterministic sequence of aggregate and program states. Thus we have a random process with some deterministic history embedded through mapping of program states to multiple stochastic process states.

2.2 Time, Event, and Frequency Views

2.2.1 Representing a PES in Time, Event, and Frequency Views

Chitra represents a PES in one of the following two dimensional graphs: program state as a function of time, program state as a function of event, and power as a function of frequency. The first two views require mapping the program state space to a single dimension. This mapping is done when Chitra starts execution. Chitra assigns to each program state s in the input file a unique nonnegative integer, denoted $f(s)$, in a contiguous interval. Each integer is the y-coordinate value that represents the corresponding program state. The mapping function used is arbitrary and does not affect the empirical model generated by Chitra. The function essentially collapses a multidimensional space into a single dimension to allow plotting on the y axis. The need to collapse the space does limit Chitra's ability to generate the simplest possible empirical

model, which is a point discussed further in the conclusions.

In time view, the point (x, y) is plotted if and only if program state $f^{-1}(y)$ occurs with time stamp y in the input file. Therefore the units of x axis points is the time base specified in the input file. In event view, the point (x, y) is contained in the plotted function if and only if the x-th program state in the input file is $f^{-1}(y)$, for $x = 0, 1, 2, \ldots$. Therefore in event view, each x axis point corresponds to a program state. Time view conveys the order and occupancy times of states, while event view conveys only the order of states. Event view displays a greater number of states with equal fidelity, compared to time view, but with a tradeoff of a loss of occupancy time information.

For convenience the user can request by a menu that Chitra connect the plotted states using line segments whose endpoints are states. Note that the PES in the input file indicates at what times the program state changes. Hence in time view the line segments are parallel to the x axis because the program state remains constant between transition times. In event view each line segment connects the points representing two adjacent program states. Connecting lines in the event view only provides visual cues that assist in interpreting the function, because non-endpoints of each line segment do not correspond to states.

The frequency view of a PES plots the discrete Fourier transform of all or part of the PES as graphed in either time or event view. If the frequency view corresponds to time view, then the x axis is in units of cycles per second; for event view the x axis is in units of occurrences in the entire log file.

2.2.2 Screen Appearance When Chitra Starts

Upon invoking Chitra, the user sees a menu bar and two windows. The menu bar is used to change the views, invoke transformations described later, generate a model, and perform

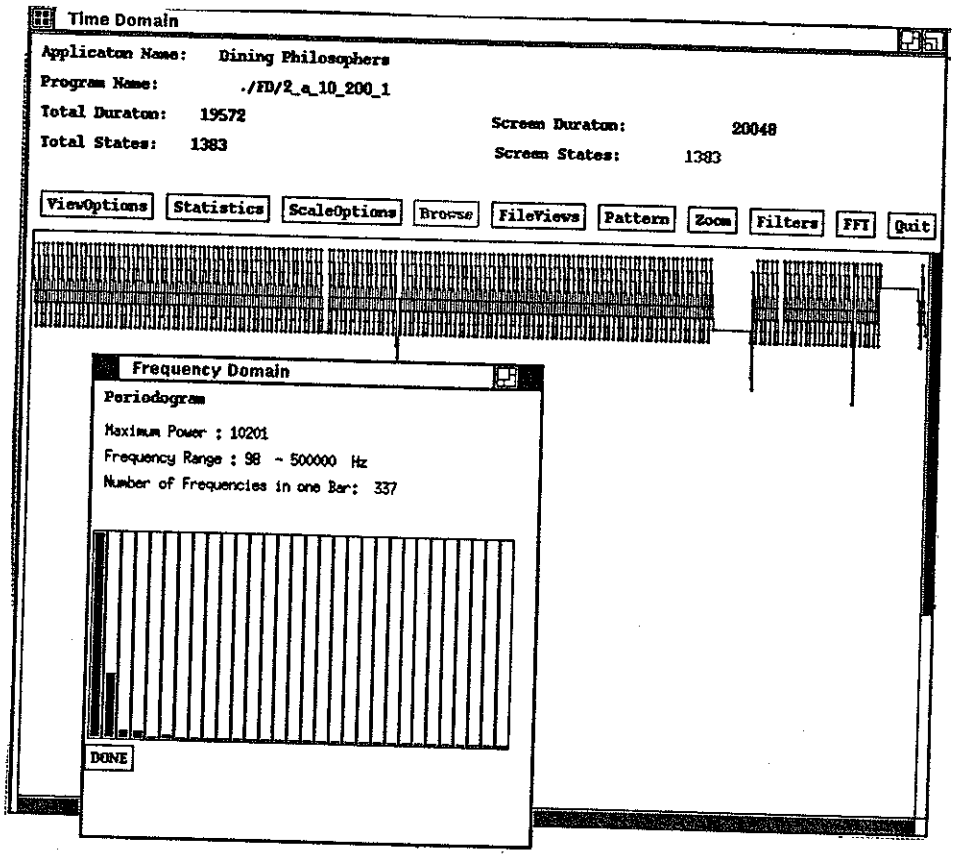


Figure 1: Initial appearance of Chitra, displaying time domain. The PES illustrated corresponds to $N = 2$, $x = 10$ for the dining philosophers program of Section 3.

miscellaneous functions. One window displays the entire PES in either time or event view; the view initially used is set according to a user preferences file, which can be modified by a menu item. The second window displays the frequency domain view of the entire PES. Figure 1 and Figure 2 show the initial display with time and event domains selected as the user preference, respectively.

2.2.3 Features of Time and Event Views

Time and event views represent windows into the PES graph. A user can adjust a slider with a mouse to control the *zoom factor*. As stated earlier, it starts at 100%. This can be reduced to

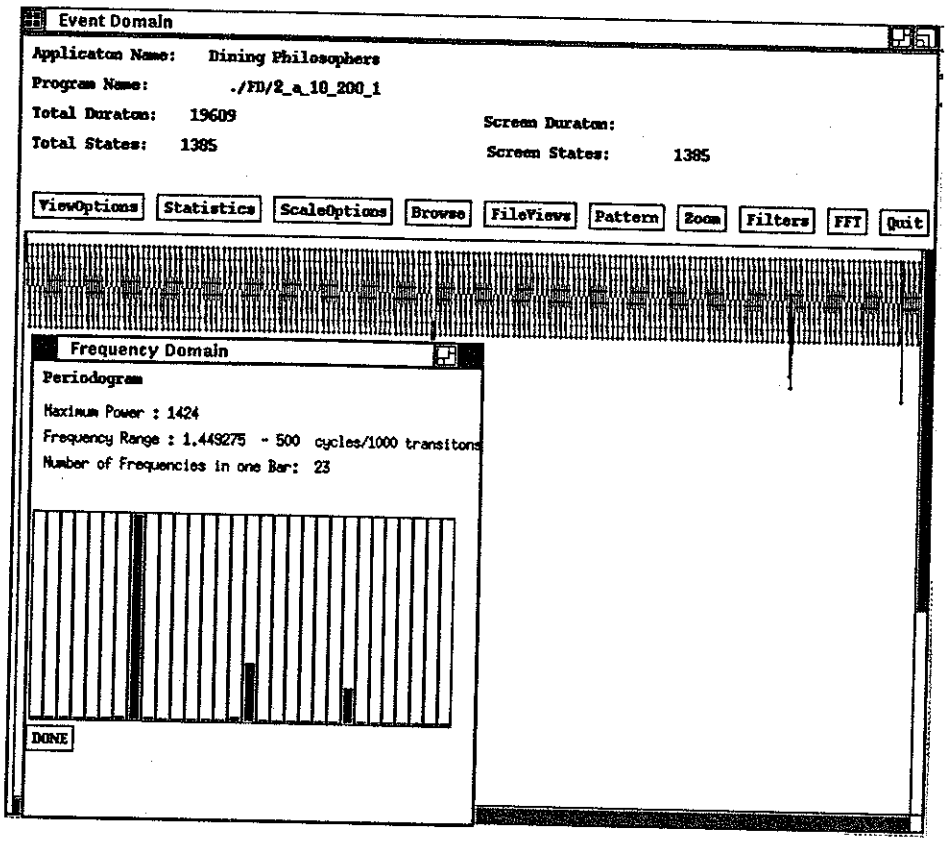


Figure 2: Initial appearance of Chitra, displaying event domain. The PES illustrated corresponds to $N = 2$, $x = 10$ for the dining philosophers program of Section 3.

zoom in on or magnify a portion of the graph. The lower bound on the zoom factor is determined by the virtual address space limit of the machine running Chitra. In an X-windows (X11R4) implementation of Chitra on a DECstation 5000/200 running X windows with 65M of swap space the limit is about 14% for log files of under 20,000 events.

The user can use the mouse with a scrollbar to pan the window over the graph. As the user moves the scrollbar, the time or event window content is continuously updated to smoothly scroll.

The combination of zooming and scrolling gives the user a way to browse through the collected data. Chitra starts by displaying 100% of the graph (Figures 1 and 2). Then the user can zoom in on part of the graph, then smoothly scroll forward or backward, up or down, to look for features of interest, and then zoom out or in.

The user can optionally display the names of each state along the y-axis. If the user selects any state with the mouse in a PES, a window will open that contains that name of the state along with a histogram of the occupancy time of the state for all occurrences in the clipped input log file.

2.2.4 Features of Frequency View

The frequency view has two modes: *auto-update* and *manual update*. In auto-update mode, whenever the user scrolls through a portion of the time or event window, the frequency view automatically computes the discrete Fourier transform of the PES visible in the time or event window when the user releases the mouse button to stop scrolling. Manual update is provided as a practical matter. Computing the transform of the event view window is faster than computing the transform of the time view. This is because all points visible in event view are passed to the FFT algorithm, whereas the user must select a sampling rate (typically one microsecond) for

the time view. An input file representing several seconds of program execution generates several orders of magnitude more points input to the FFT algorithm than inputing the event view data. Hence the user can deactivate auto-update, and the frequency view will only be updated when the user selects the appropriate button.

Frequency view has two uses. First a concentration of energy at particular frequencies means that a state transition is more likely to occur at certain frequencies; this suggests some patterns of behavior may exist. The second use is that the periodogram serves as a "footprint" for a particular observation of program execution. Multiple observations may be quickly compared on the basis of similarity of their periodograms.

For example, the frequency domain views in Figures 1 and 2 show that the PES consists almost exclusively of a repeated pattern of behavior. In Figure 1 the energy is concentrated in the first bin, corresponding to 98-435 Hz, and in Figure 2 the energy is concentrated between 1.4 and 24.4 cycles per 1000 transitions. The remaining bins with non-zero energies represent harmonics of the fundamental frequency.

2.3 Transformations

2.3.1 Clipping

Typically one wishes to analyze only a certain segment of a PES. For example, one might only want to examine the portion of the PES during which all processes are active, thereby eliminating a startup and shutdown interval. Or, a program might pass through phases that can be distinguished based on some visual feature, and the user wants to analyze only a certain phase. In these cases, the user can use the mouse to select a point and then, while pressing a mouse button, move the mouse to another point on the screen. A rubber band rectangle is drawn with opposite vertices at the selected points. The user can then select the CLIP menu

item. Chitra will then function as though the user deleted all program states in the input file *except* those corresponding to points displayed inside the selected rectangle.

2.3.2 Filtering

A filter eliminates certain states in a PES, which can eliminate noise (perturbations). When the user defines and activates a filter, the effect is as though the user deleted all program states in the input file that are selected by the filter. Therefore the filtered states are no longer displayed in time or event view, and the frequency of a filtered state is no longer represented in the frequency view.

A user can define a filter in Chitra based on either the time, event, or frequency domains. An unlimited number of filters can be applied to the input file.

Time domain filter: Consider a random variable representing the total occupancy time of all occurrences of a program state in the input file expressed as a fraction of the total time interval represented by the input file. (For example, if the random variable has value 0.1, then the program spent 10% of the period represented by the PES in each state.) When the user selects a menu item to define a time domain filter, Chitra pops up a window showing a histogram of this random variable. The user then selects a fraction of log file time to serve as a threshold; Chitra then appears to eliminate from the PES all states whose total occupancy time expressed as a fraction of the total time interval represented by the input file is less than this threshold. For example, if the user selects a threshold of 0.3, then only states which each account for at least 30% of the time interval represented by the input file will appear be displayed in the PES.

Event domain filter: Consider a random variable representing the total number of occurrences of a state in the input file. When the user selects a menu item to define an event domain

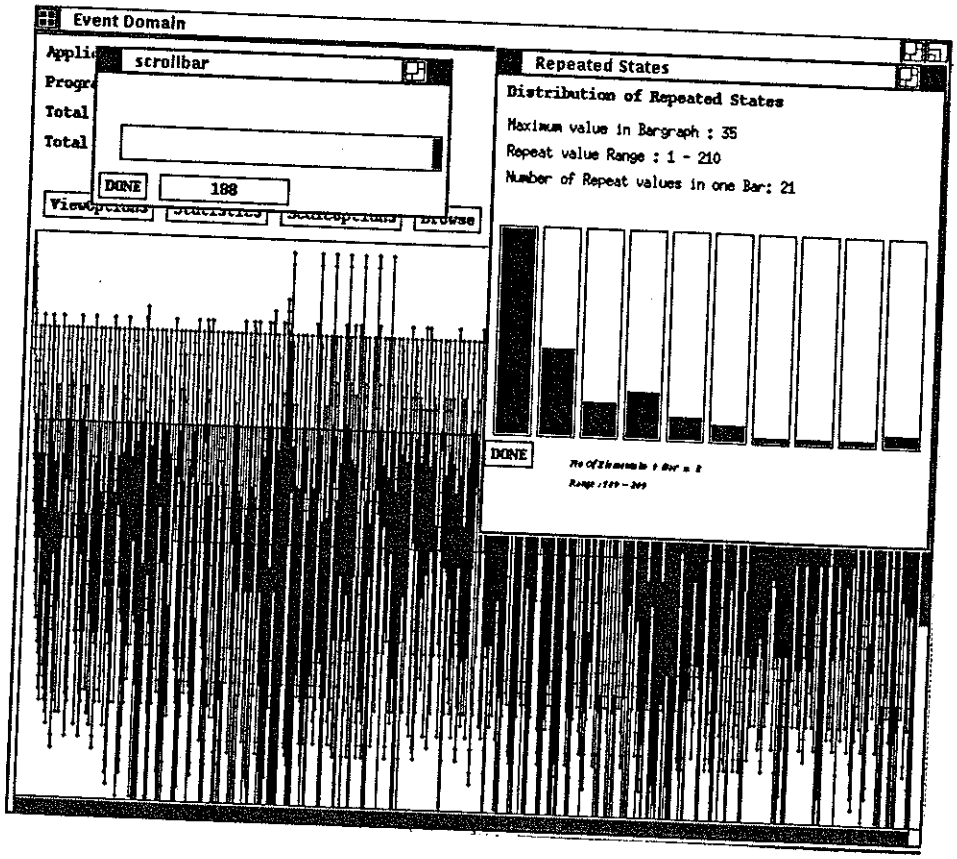


Figure 3: Illustration of event domain filtering (Part 1). Histogram overlaying original, random appearing PES in event view indicates two clusters of states.

filter, Chitra pops up a window showing a histogram of this random variable (Figure 3). The user then selects a number of occurrences to serve as a threshold. Chitra then appears to eliminate from the PES all states whose total number of occurrences is less than this threshold (Figure 4). The graph of Figure 3 appears to be highly irregular. Selecting a threshold of 180 and deactivating the connection of states by line segments turns Figure 3 into the perfectly regular graph of Figure 4. Figure 4 displays purely deterministic behavior underlying Figure 3.

the Figures 3 and 4, the user selects a threshold of 180, after which only states which occur greater than 180 times in the input file will appear be displayed in the PES.

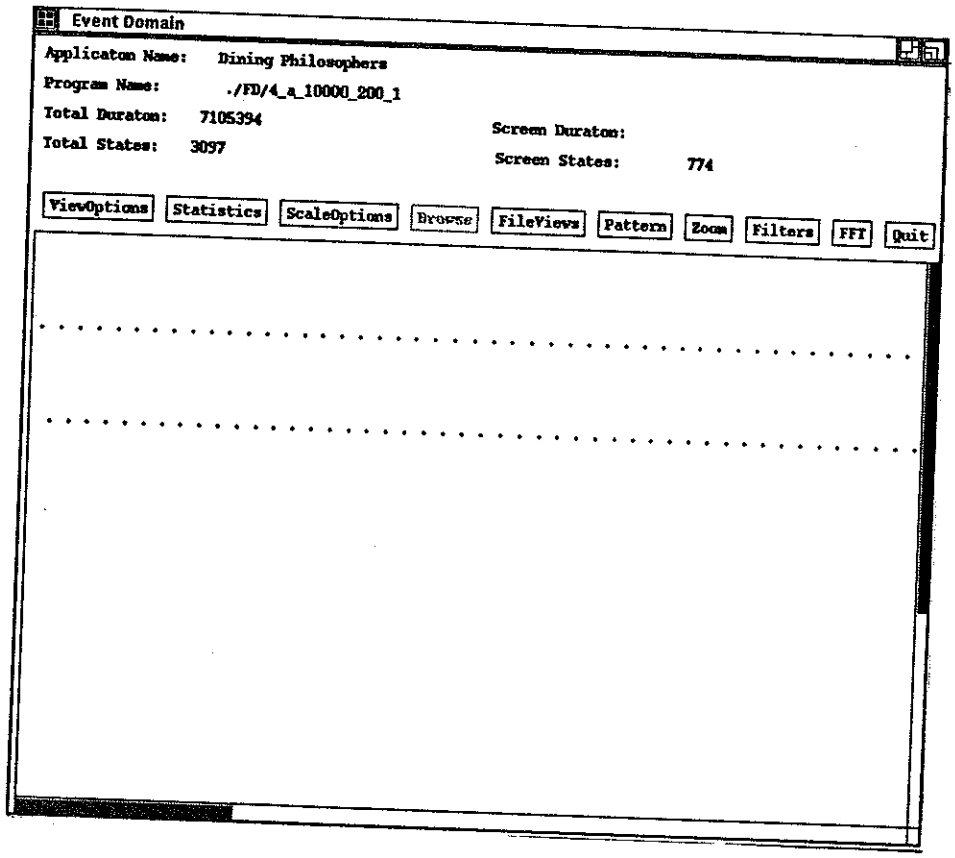


Figure 4: Illustration of event domain filtering (Part 2). Filtering the states occurring greater than 180 times transforms the PES into a completely regular graph.

Frequency domain filter: When a user selects a menu item to define a frequency domain filter, the user specifies an interval of frequencies to filter. Chitra then appears to eliminate from the PES all states whose frequency in within this interval. (This feature is not presently implemented in Chitra, and was not needed for the case study in Section 3, but seems desirable for symmetry with the time and event filters.)

These three filter types are highly effective; they can erase noise to transform an aperiodic PES into a periodic PES, as illustrated in Figures 3 and 4. For example, we are often interested in what the program is doing the majority of the time. Therefore time domain filter can be used to retain only states that amount to, for example, 80% or 90% of the total time interval measured by the clipped log file. As an another example, suppose an interrupt occurs every 10 milliseconds. Then the frequency domain view will show a harmonic at 100 Hz. Filtering out the 100 Hz frequency can eliminate the effect of interrupts.

2.3.3 Aggregation

Aggregation is the process of replacing two or more program states in the input file by a new, aggregate program state. Aggregation is useful to reducing the size of the program state space. There are three forms of aggregation: static, dynamic-sequence, and dynamic-parallel.

Static Aggregation: "Static" means that the user specifies a rule of how to transform the program state space based on knowledge about the program without examining the execution dynamics. Each rule contains a list of program states followed by an aggregate state. Chitra will scan the PES and replace each occurrence of any state in the list by the corresponding aggregate. For example, if the program state comprising a PES is an integer denoting the number of processes that are waiting at a barrier, with state of zero representing the situation

of all processes at the barrier, then two static aggregation rules could define states "At_Barrier" and "Not_At_Barrier," which replace program state zero and all positive integer program states, respectively.

The user must store in a file a set of rules used to statically aggregate. This can be done before initiation of Chitra, or can be created and applied during execution of Chitra. The set of aggregation rule files currently available are read from disk and listed in a Chitra menu.

Dynamic Aggregation: "Dynamic" means that the user identifies an aggregation rule based on examining one or more PES's. The rule may not be one which the user would have thought of by just looking at the source code of the program being analyzed. There are two varieties of dynamic aggregation:

Sequence aggregation: The user specifies an aggregation graphically using the mouse to select a fragment of a PES by drawing a rectangle around a sequence of states in the same manner as described above for clipping. When the mouse button is released, Chitra pattern searches the entire log file for state sequences (but not occupancy times) matching the state sequence in the rectangle. Chitra then overlays a rectangle shaded in a color not previously displayed on all state sequences matching the selected sequence in the entire PES. This shading serves two purposes. First, the user can observe how many times the pattern occurs and hence what percentage of the file is accounted for by the selected pattern. Second, the use of a different color for each pattern gives an indication of the relative frequency and ordering of patterns.

After selecting one or more patterns, the user can select the AGGREGATE menu item, which will replace each pattern by a new aggregate state. The state will be represented by a y-axis coordinate that is not used for any existing state or aggregate state. A PES

containing many repeated patterns will be greatly reduced in size in this manner.

The user can also select an UNAGGREGATE menu item to undo the affect of aggregation. This feature is useful because the user is faced with choices in how to aggregate states, and may want to try one way and then undo the effect. For example, if A, B, C, and D are states and a PES contains the state sequence AABAACAABAAD, then the user might aggregate AA (call the resulting aggregate state Y) and get YBYCYBYD, or the user might aggregate AAB (call the result state Z) and get ZAACZAAD.

The user can repeat the pattern selection and aggregation process an arbitrary number of times. Aggregate states are treated the same as states appearing in the original log file, so that one can aggregate some pattern of aggregate and program states into yet another aggregate states.

Parallel aggregation: For any two states s_0 and s_1 in a PES, a *parallel aggregate state* may be defined. The parallel aggregate state denotes an acyclic stochastic process whose state transition matrix represents all possible paths between s_0 and s_1 that occur in the PES. The occupancy time of the aggregate state is the mean of the sojourn time through each path of the process.

Section 2.3.2 described filters based on the time, event, and frequency domain. When a user activates a filter, filtered states are no longer visible on the screen. Chitra will automatically create a set of one or more parallel aggregate states that incorporates all filtered states; therefore filtered states are represented as aggregate states in the empirical model generated by Chitra.

This type of aggregation is used often in the case study for a PES in which a sequence aggregate state recurs, but each transition out of the state is into one of M (possibly

aggregate) states that seems to be selected at random; following each of these M states is an aggregate state that recurs. A parallel aggregate state represents a set of alternative states, exactly one of which is reached on each return to the state.

2.4 Output Model

The user can select a menu item to generate a disk file of the model which Chitra produces based on the analysis occurring so far during the Chitra session. This output includes:

- a state transition matrix, representing the frequencies of transitions among the program and aggregate states in the final, filtered PES that exists at the end of analysis;
- a definition of the program states comprising each aggregate state; and
- statistics on each program and aggregate state remaining in final, filtered PES.

3 Case Study

Chitra has been used to analyze a program solving Dijkstra's dining philosophers problem. In the problem, a set of philosophers each forever cycle through four local states: thinking, acquiring utensils, eating, and releasing utensils, denoted T, A, E, and R. Philosophers must contend for utensils. Let philosophers be numbered $0, 1, \ldots, N-1$. A program state is an ordered N-tuple, where the i^{th} component is the local state of philosopher i. For example in program state $ETET$ philosophers zero and two are eating, and philosophers one and three are thinking.

The sharing of utensils is represented by a graph whose vertices each represent a philosopher and whose arcs each represent a utensil. An arc exists between the vertices representing philosophers i and j, for $0 \leq i, j < N$, if and only if philosophers i and j share a utensil. We consider the case where the graph consists of a cycle containing all vertices and the duration of eating

and thinking time for all philosophers is the constant x. The program has two parameters: the number of philosophers, N, and the duration of the eating and thinking times, x.

Case study objective: The *resource acquisition time* for a thread is defined as the time that the thread spends in local state A. The acquisition time is of interest because it is proportional to the time that a thread blocks on a spinlock for a busy utensil. Program execution is defined to be *efficient* if the acquisition time is a small constant at all parameter values, and execution is defined to be *fair* if the acquisition time is equal for all philosophers at any value of x and N.

A program reaches *steady state* if each transition rate in the stochastic process generated by Chitra converges.

Our objective is to formulate an empirical model of the program that will predict the PES for $N = 2, 3$, or 4 and $10^3 \leq x \leq 10^4$, determine if the program reaches steady state, and use the empirical model to predict the resource mean acquisition time of each resource access for each thread, and evaluate whether the program is efficient and fair.

3.1 Program Implementation

The program is implemented using the Presto 0.4 thread [2] package on a Sequent Symmetry multiprocessor. The *Think* and *Eat* operations are implemented as a loop that repeatedly multiplies the same operands; x is the number of repetitions. Let utensils be numbered $0, 1, \ldots, N-1$. Each philosopher is implemented by a non-preemptible Presto thread; the mapping from thread to processor does not change during program execution. Each utensil is implemented by a Presto spinlock; *Utensil[i]* is a spinlock implementing utensil i. A thread busy waits at an unavailable spin lock without relinquishing the processor. Figure 5 contains the code for philosopher i. Primitives *Lock* and *Unlock* are operations on a spinlock. Functions g_0 through g_3 define the order in which utensils are acquired and released. For the results presented in this section,

```
while (TRUE) {
    Think;                  //State T
    Lock(g0(i));            //State A
    Lock(g1(i));            //State A
    Eat;                    //State E
    Unlock(g2(i));          //State R
    Unlock(g3(i));          //State R
    }
```

Figure 5: Code of philosopher i, for $1 \leq i \leq N$.

$$
\begin{aligned}
g_0(i) &= \begin{cases} i-1 & \text{if } i \text{ is odd} \\ i & \text{otherwise} \end{cases} \\
g_1(i) &= \begin{cases} i & \text{if } i \text{ is odd} \\ (i-1) \bmod N & \text{otherwise} \end{cases} \\
g_2(i) &= g_0(i) \\
g_3(i) &= g_1(i)
\end{aligned}
$$

Recalling the graph defined above, the definition of g_0 through g_3 specify that about half the philosophers acquire the clockwise utensil, while the remaining philosophers acquire the counterclockwise utensil. Utensils are released in the order in which they are acquired.

The Sequent provides what appears to a program as a single, global clock with a microsecond resolution in the address space of all processors. The instrumentation of each philosopher thread consists of reading the clock whenever a state transition within the thread occurs; the clock value along with the new state is written to an array local to the thread. The time required for the read and write sequence, when not interrupted, is measured to be four microseconds. Each array fits on a small number of 1024 byte memory blocks within a thread's address space, to minimize the number of page faults. After execution terminates, the thread arrays are written to disk to form the file input to Chitra.

The PES's analyzed in this section were collected when the dining philosophers program

was the only program running on the Sequent. For each data point we execute the program at least three times and observe the PES. In this case study, repeated observations always produce differences in the PES's; these will be evident through the standard deviations of stochastic process state occupancy times that are reported. In all cases we believe the differences are caused by clock interrupts and (very rare) page fault interrupts; this conclusion is justified in Section 3.3. Varying the thread starting order did not alter the stochastic process generated by Chitra.

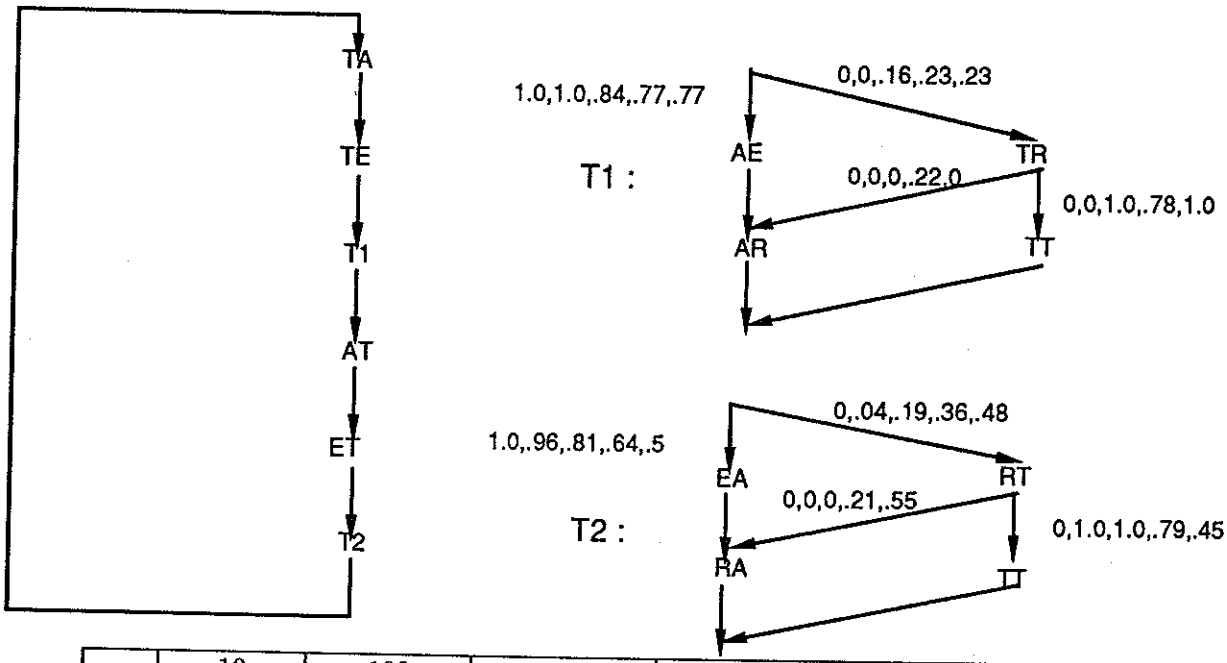
3.2 Analysis of PES's

We examined PES's from dining philosopher programs with $N = 2, 3$, and 4 philosophers for $x = 10, 10^2, 10^3$, and 10^4 microsecond eat and think times. The empirical model generated by Chitra is shown for $N = 2, 3$, and 4 in Figures 6, 7, and 8, respectively. Each figure represents the transition rate matrix of a stochastic process by a graph and a table. The verticies of the directed, weighted graph each correspond to a stochastic process state, and the arcs denote all possible transitions. Each arc label enumerates the mean transition frequency for each value of x observed, listed in ascending order. For example, in Figure 6, upon entry to parallel aggregate state $T1$, state AE is taken 100%, 100%, 84%, 77%, and 77% of the time when $x = 10, 10^2, 10^3, 10^4$, and 10^5. An arc with no weight indicates that the transition was taken 100% of the time for all values of x. The table in each figure contains the sample mean and, in paranetheses, the standard deviation of the observed state occupancy time, in microseconds.

The stochastic processes reveal the following:

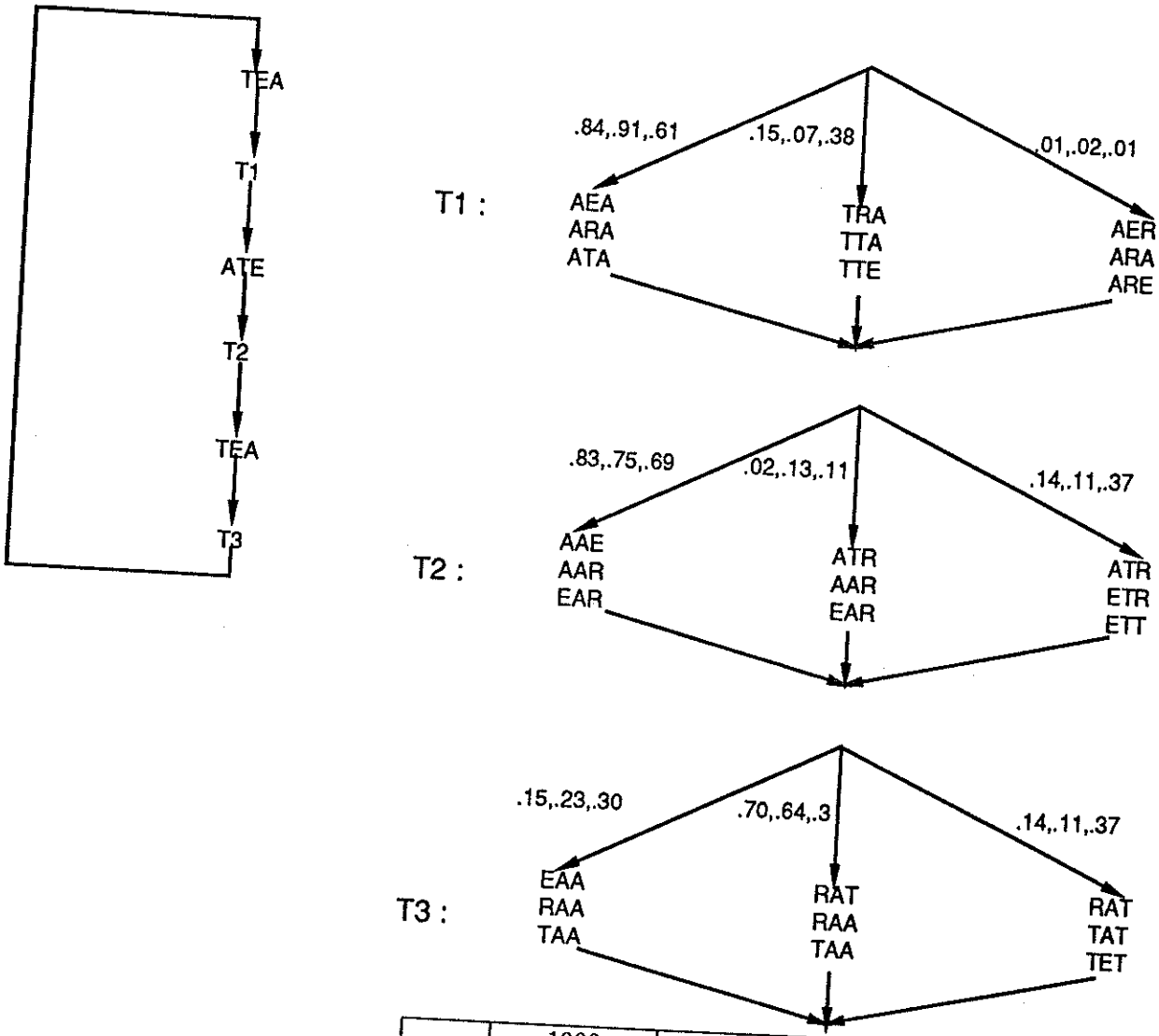
1. The program reaches steady state for all values of x and N investigated.

2. In each case the steady state behavior, as represented by a stochastic process, consists of a cycle of states. For example the cycle is TA, TE, $T1$, AT, ET, and $T2$ for $N = 2$, where



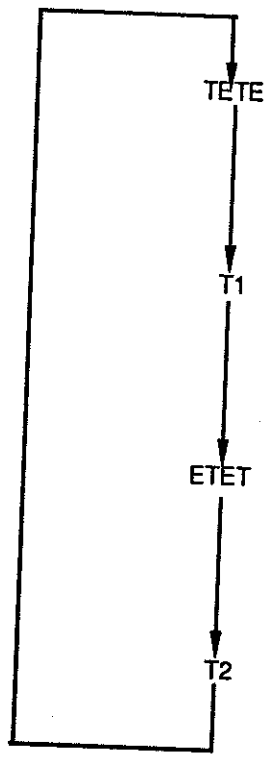
	10	100	1000	10000	100000
TE	21(0.92)	183(13.59)	1807(37.3)	18237(48.98)	182491(32.32)
AE	8(63.43)	9(41.22)	40(76.4)	47(95.79)	47(77.77)
AR	23(4.93)	23(5.02)	22(4.03)	21(3.06)	23(4.71)
AT	3(1.84)	3(2.07)	6(7.77)	9(8.83)	10(9.75)
ET	21(0.9)	183(10.25)	1807(25.84)	18150(1186.58)	181650(10896.05)
EA	8(64.2)	12(66.16)	36(72.0)	158(1404.72)	1348(14109.18)
RA	23((6.58)	23(4.77)	22(5.5)	22(3.54)	20(5.61)
TA	3(1.93)	4(3.76)	7(8.5)	8(7.82)	7(6.56)
TR	0(0)	15(7.28)	18(2.53)	10(7.56)	18(3.05)
TT	0(0)	52(12.06)	50(9.18)	403(2369.5)	2025(17406.8)
RT	0(0)	17(5.29)	18(2.34)	10(8.95)	8(7.35)
T1	0(0)	0(0)	59.74(49.59)	64.44(74.84)	63.89(31.32)
T1	0(0)	0(0)	60.55(51.6)	153.85(1217.72)	833.92 (10949.15)

Figure 6: Empirical model of two philosophers generated by Chitra.



	1000	10000	20000
ATE	1816(22.53)	18239(34.21)	36482(26.08)
T1	50(68.57)	53(69.36)	51(67.55)
EAT	1808(20.04)	18238(37.06)	36476(25.15)
T2	51(68.83)	54(71.02)	73(64.87)
TEA	1810(51.63)	18245(34.74)	36338(1767.08)
T3	55(66.91)	55(65.86)	190(1623.61)

Figure 7: Empirical model of three philosophers generated by Chitra.



	1000	10000	20000
TETE	1762(92.87)	18119(1090.54)	36427(195.57)
T1	144.7(129.4)	199.11(1088.8)	188 (287.07)
ETET	1770(88.56)	18095(1254.86)	36320(1340.36)
T2	133.31(143.53)	279.69(1522.43)	296.03(1660.29)

Figure 8: Empirical model of four philosophers generated by Chitra.

$T1$ and $T2$ are parallel aggregate states for $N = 2$ (Figure 6).

3. The program reaches the same steady state cycle for all values of x for a given value of N.

4. With an even number of philosophers, at steady state the program spends most of its time in a state in which one philosopher is thinking, while the other is eating. With an odd number of philosophers, the program spends most of its time in a state where exactly one thread is eating, one is thinking, and the other is acquiring utensils.

5. A single state transition graph can be used to represent all values of x with a given value of N. Therefore each stochastic process can be used to predict the program behavior at values of x that were not observed.

The desired performance metric of resource acquisition time per thread is plotted in Figures 9 and 10. The graphs were constructed by applying a static filter to all PES's which mapped all program states all program states into just two states in which a particular philosopher either was or was not acquiring utensils.

For $N = 2$, the program execution is not fair because philosopher 1 always waits a constant amount of time, while philosopher 2 waits for a time period that grows with the resource holding time, x. The cases of $N = 3$ and 4 are fair. The program becomes less efficient with a growth in resource holding time for all parameter values. However it is most inefficient for $N = 3$ philosophers. The explanation for this illustrates the value of generating an empirical model for the program: Figure 7 shows that in all stochastic process states, there is never more than one philosopher eating at a time. This explanation would be difficult to deduce without the empirical model.

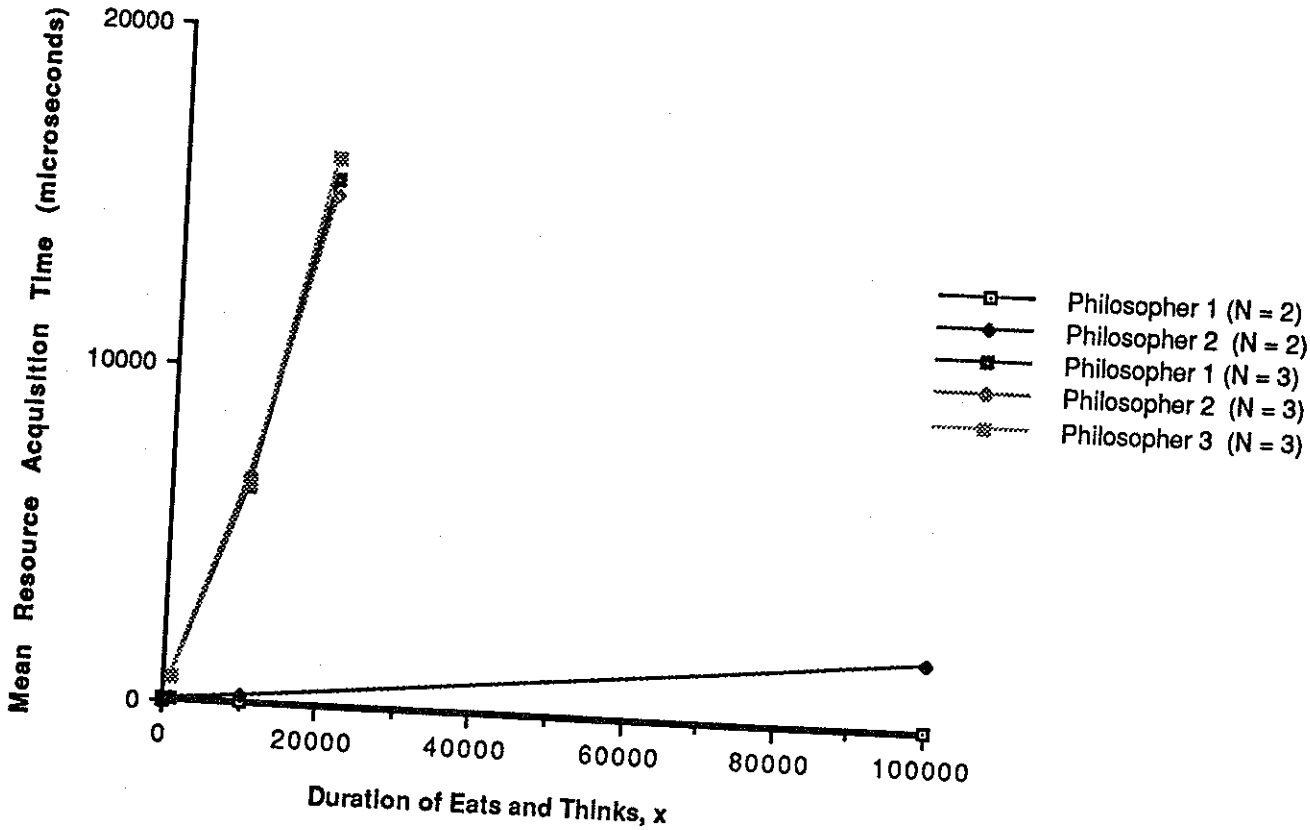


Figure 9: Prediction of resource acquisition time per thread as a function of x, for $N = 2$ and 3, based on empirical models in Figures 6 through 7.

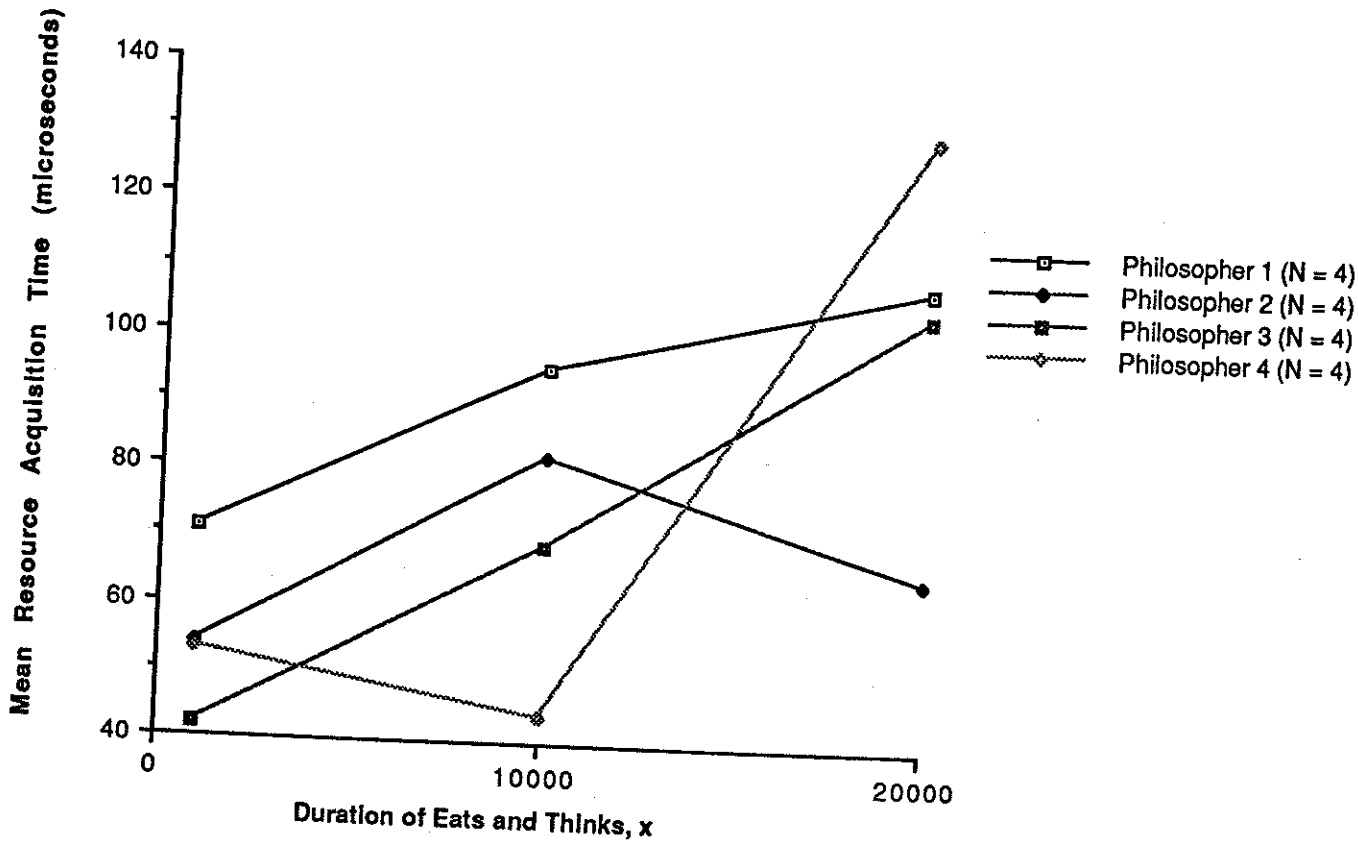


Figure 10: Prediction of resource acquisition time per thread as a function of x, for $N = 4$, based on empirical model in Figures 8.

3.3 Analysis of Perturbations

The stochastic processes in the previous subsection reveal that, for a given value of N, the program always reaches the same steady state cycle for all observations of any value of x. A *perturbation in a PES* is defined as one of the following:

Type 1: a state transition that is taken with a frequency of less than a certain value,

Type 2: a state occupancy time observation that is separated from the sample mean by more than a certain percentage of the sample mean, or

Type 3: a state whose number of occurrences in a PES is less than a certain percentage of the total number of state occurrences in the PES.

The stochastic processes in Figures 6 through 8 would be simplified by eliminating perturbations from the PES's analyzed. A type 1 perturbation manifests itself through a parallel aggregate state. A type 2 perturbation results in large variances in state occupancy times. A type 3 perturbation manifests itself by increasing the number of states in the stochastic process.

Two questions arise with respect to perturbations. First, what is responsible for perturbations: something external to the program, or are they an inherent characteristic of the program itself? Second, if perturbations are removed from a PES, will the simpler stochastic process that Chitra generates be sufficient to predict program measures of interest? These questions are addressed below.

3.3.1 Source of Perturbations

Perturbations due to Interrupts: The time domain view of all PES's reveals a perturbed state transition in each thread every 15 milliseconds; the perturbation lasts for about 800 mi-

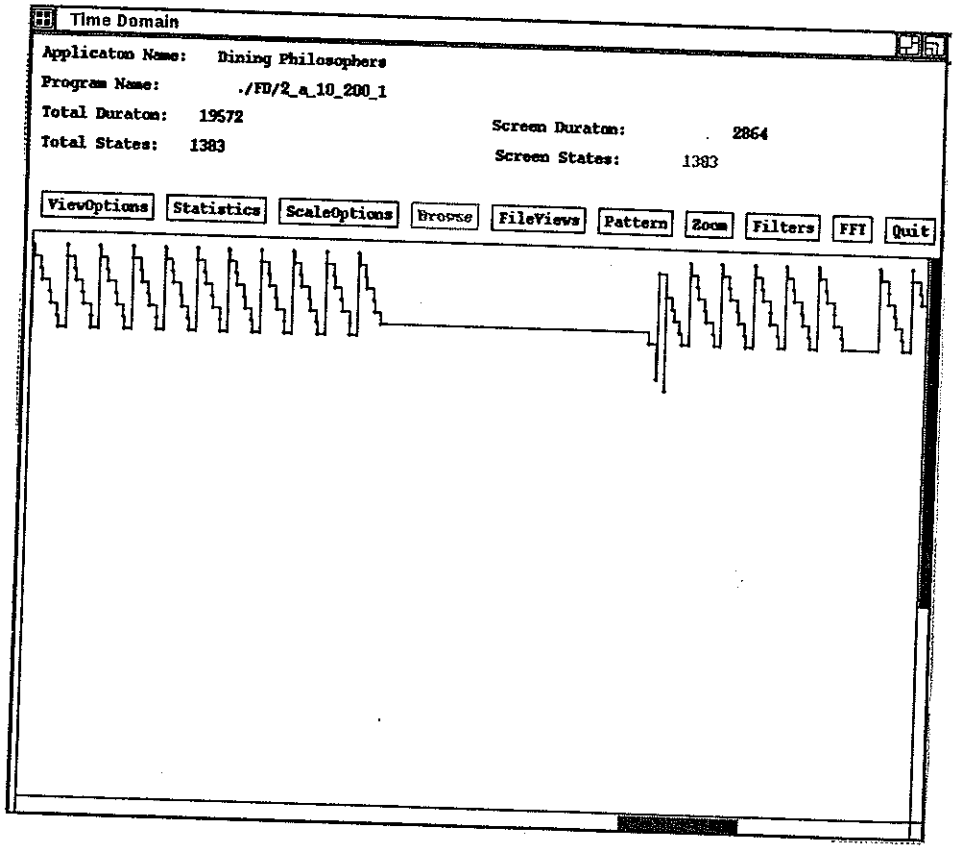


Figure 11: Illustration of perturbations in PES in time view. An interval of 2864 microseconds is represented. The two long horizontal lines are approximately of length 800 and 100 microseconds, and represent perturbations in the normal state transition sequence, probably due to clock and page fault interrupts, respectively.

croseconds (see Figure 11). Perturbations among processors are not synchronized. We attribute these perturbations to the Sequent's clock interrupts.

Measurement shows that the program occasionally experiences zero to two perturbations of duration 100 microseconds between every to clock interrupt perturbations (see Figure 11). There is no pattern to these perturbations. Our hypothesis is that these perturbations correspond to page faults. We confirmed this hypothesis by constructing a test program of a single thread that just executed a loop whose body is a loop of multiplies and records the time required to

execute the inner loop in a large array. Note that the program touched each array element exactly once, so that it traversed a sequence of pages in the virtual address space and hence would periodically produce page faults. The program manifested frequent 100 microsecond perturbations. We then modified the program to eliminate the large array and just keep track of the minimum and maximum loop times. This program displayed virtually no 100 microsecond perturbations; therefore it rarely manifested the interrupt. Running five copies of the program on five processors produced the same results.

An interrupt perturbs either no threads, some threads, or all threads. Generally the design of a parallel program will limit the set of threads affected by the interrupt. Zero threads are affected if only a processor running an idle thread is interrupted, however this is not the case for timer interrupts on the Sequent.

Interrupts can have two domains of influence in the dining philosophers, corresponding to whether the interrupt occurs to a thread that does or does not hold resources. A thread does hold resources in most of the acquire state, all of the eat state, and most of the release state. An interrupt in these states causes two utensils to be "frozen," with neighbors waiting for an extra time due to the interrupt service time. This will make a noticeable perturbation in the observed PES. On the other hand, a thread does not hold resources in the end of release, all of think, and the start of acquire. Here the interrupting thread will not to hold one or more utensils for a period of time that is elongated by the interrupt service time. This can also affect neighboring threads, because they might be able to eat one or more extra times while the interrupted thread is delayed due to the interrupt. This case is particularly sensitive to the relative frequencies of the eat-think cycle and interrupts.

Artificial Perturbations: Another source of perturbations is introduced by the fact that Chitra uses a discrete state space in its empirical model. Suppose that in a certain segment of time, two processes are each traversing through a sequence of L local states that contain no synchronization points. This implies that there are L^2 possible global states representing all possible relative speeds of execution of the two processes. Even if the program execution reaches a cyclic steady state, different PES's will contain different subsets of the L^2 states. Many states will occur with low frequency in the set of all observations and hence be perturbations, even though the actual program execution is highly regular.

As an example, in Figure 7 there are always three successors to each states TEA, ATE, and TEA, and the three successors correspond to a different one of the three threads transiting out of its current state first. For example, in TEA either philosopher 1 finishes thinking first, philosopher 2 finishes eating first, or philosopher 3 finishes acquiring first. This race condition results in the three parallel paths that follow state TEA. This problem causes an exponential growth in the size of the stochastic process which is purely due to the choice of a discrete state space, even though the underlying program may be highly regular.

An open problem is how to better expose a regular underlying program behavior. One possibility is to use a continuous state space visualized in a multidimensional space, which is the solution explored in [1]. The disadvantage of a continuous space is that it needs to be hand tailored to the program under study. Chitra, in contrast, works for any parallel program.

3.3.2 Impact of Eliminating Perturbations

If our hypothesis that the 100 and 800 microsecond perturbations are due to clock and page fault interrupts is correct, then they should be eliminated from PES's because they have nothing to do with the inherent structure of the program under study.

Number of Philosophers, N	Eat and think time, x					
	10	100	1000	10000	20000	100000
2	0.31%	7.86%	9.48%	2.7%	-	8.1%
3	-	-	10.4%	7.64%	7.86%	-
4	-	-	4.6%	6.18%	5.24%	-

Table 1: Thresholds of filtering in the event view in Figures 6 through 8.

The fact that each processor receives clock interrupts independently from other processors implies that the number of perturbations is proportional to the number of processors. Therefore a cyclic steady state behavior in a program such as the dining philosophers tends to become masked by growing numbers of perturbations as the number of processors used increases.

As discussed in Section 2.3, Chitra has the ability to filter type 2 and 3 perturbations from a PES from view, and then, as discussed in section 2.3.3 automatically represent the filtered states by one or more aggregate states. In fact, the stochastic processes in Figures 6 through 8 model PES's with type 3 perturbations excluded at the thresholds shown in Table 1.

4 Conclusions and Future Directions

Visualization provides a graphical window to the execution of a program. Visually-guided construction of an empirical model of program behavior is helpful in understanding the execution behavior and performance of parallel programs. Although gross measures of program behavior can identify the presence of performance problems, they do not explain why the behavior is so. For example, the gross level measure of mean resource acquisition time of each resource access in Figure 9 reveals inefficient performance for N, but the stochastic process in Figure 7 shows that the behavior arises because no more than one philosopher can eat at a time.

Chitra is unique among performance visualization tools in that it is the only tool to generate

an empirical model as the culmination of visualization. Its innovations include display of program behavior in the frequency domain as well as the use of pattern matching and filtering to define aggregate states. Chitra permits aggregation until we get a single aggregate state in stochastic process; therefore one can reduce the model until it is acceptably small for objective of study. Finally, Chitra can generate a realistic semi-Markov model of program behavior, even though software generally does not obey a Markovian property, by mapping individual program states to multiple stochastic process states in a way that embeds history of a state. For example, the program state TT in the dining philosophers problem is represented by two states, TT_0 and TT_1, in the stochastic process of Figure 6. This mapping is determined as a by-product of the visual aggregation done by a user.

Conclusions from the case study in Section 3 are summarized below.

- The dining philosopher program analyzed always reaches a cyclic steady state behavior. One stochastic process can be used to represent the behavior for all observed resource holding times, which allow the model to be used to predict program behavior.
- Perturbations evident in the case study do not appear to be primarily due to inherent properties of the program. Rather, they arise are due to clock interrupts, page faults, and the use of a discrete state space
- As discussed earlier, a discrete state space can represent regular program behavior by what appears to be many perturbed states. An open problem is to find alternate models of program behavior that reveal the regular program behavior.

Besides the dining philosophers program, we have applied Chitra to another problem, a commercial TCP/IP product.[1] The most important future work will be to apply the tool to additional commercial-size software. The empirical model generated by Chitra will probably

need to be changed to a non-homogeneous process, because a program may pass through phases during execution. A non-homogeneous process could capture the behavior of a program as it moves from one state of execution to another.

At present Chitra analyzes a single PES from a single program execution. In the future we would like Chitra to analyze a set of PES's forming an ensemble and representing multiple runs of a program with the same parameter set. This would eliminate some tedious data reduction that was done outside of Chitra in the case study of Section 3 to reduce a minimum of three observations of the program for each parameter value to a single stochastic process.

Ideally, a user could simultaneously display an ensemble of PES's, and define aggregations and filters that are applied to all PES's in the ensemble. Curves representing the periodogram of all log files could be drawn in a single frequency domain view, so that the user can see if the energy distribution of all runs are similar. If they are not similar, then several "modes" are present which must be analyzed separately. One possibility is for Chitra to calculate the mean energy distribution from the distribution of each PES (and use confidence intervals), and then display the inverse transform in the time and event domains; this signal is a stochastic estimate of the PES. From this average signal Chitra could generate the corresponding empirical model, which represents average behavior.

An open problem for theoretical modeling is to explore the superposition of parallel programs. In the case study of Section 3 the dining philosophers program is preempted by clock and page fault interrupts. Can a model be developed to predict how a stochastic process representing interrupts alone be superimposed with a stochastic process representing the parallel program behavior in the absence of interrupts? This is superficially analogous to superposition in linear systems theory.

A design philosophy behind Chitra has been to make the tool as interactive as possible.

However, computing the frequency domain view of a PES from the time domain view can require several minutes. At present we can achieve this on a moderately fast RISC architecture color workstation because we analyze a single PES representing no more than a few seconds of program execution. However, analyzing an ensemble PES's, each representing several minutes of program execution will be impossible on a workstation. In the future we would like to run Chitra on a supercomputer and use the workstation as an X terminal.

Another direction for enhancement of Chitra is to explore alternate methods of displaying PES's. Currently Chitra maps the program state space to a two dimensional Cartesian space. One alternate display maps each process to a separate axis in an $N+1$ dimensional Cartesian space, and gives the user the ability to display arbitrary two or three dimensional views into the Cartesian space and to let the user rotate or navigate through the space. Mapping each process to a dimension in Cartesian space has been used to develop algorithms for deadlock detection [5] and to derive the governing equations for some parallel programs [2]. Again a supercomputer would probably be necessary, to support rotations and navigation.

Acknowledgements

Ashok K. Agrawala suggested the use of frequency domain analysis. The Sequent Symmetry at the Argonne National Laboratory was used for the experiments reported.

References

[1] M. Abrams, N. Doraswamy, and A. Mathur, *Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event, and Frequency Domains*, Department of Computer Science, Virginia Tech, TR 92-24, May 1992, submitted for publication.

[2] M. Abrams and A. K. Agrawala, *Geometric Performance Analysis of Mutual Exclusion*, Department of Computer Science, Virginia Tech, TR 90-58, 1990, submitted for publication.

[3] B. N. Bershad, E. D. Lazowska, and H. M. Levy, Presto: A System for Object-Oriented Parallel Programming, TR 87-09-01, Dept. of Computer Science, Univ. or Washington, Spet. 1987.

[4] M. H. Brown, *Algorithm Animation*, MIT Press, Cambridge, MA, Ph.D. thesis, Department of Computer Science, Brown University, 1988.

[5] S. Carson and P. F. Reynolds, "The Geometry of Semaphore Programs," *ACM TOPLAS 9*, No. 1, Jan. 1987.

[6] Jack Dongarra, Orlie Brewer, James Arthur Kohl, and Samuel Fineberg, "A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors," *Journal of Parallel and Distributed Computing 9*, 185-202, 1990.

[7] Robert J. Fowler, Thomas J. LeBlanc and John M. Mellor-Crummey, "An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors," *Proceedings of Workshop on Parallel and Distributed Debugging*, ACM SIGPLAN Notices 24, 1 Jan. 1989.

[8] Alfred A. Hough and Janice E. Cuny, "Belvedere: Prototype of a Pattern-oriented Debugger for Highly Parallel Computation," *Proc. of the 1987 Intl. Conf. on Parallel Processing*, 1987.

[9] Thomas J. LeBlanc, John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers C-36*, No. 4, April 1987.

[10] Allen D. Malony, "JED: Just an Event Display," in *Performance Instrumentation and Visualization*, ed. M. Simmons and R. Koskela, ACM Press, 1989.

[11] Allen D. Malony, *Performance Observability*, Ph.D. thesis, Univ. of Illinois, Aug. 1990.

[12] Allen D. Malony and Daniel A. Reed, *Visualizing Parallel Computer System Performance*, CSRD Tech. Report No. 812, Univ. of Illinois, March 1988.

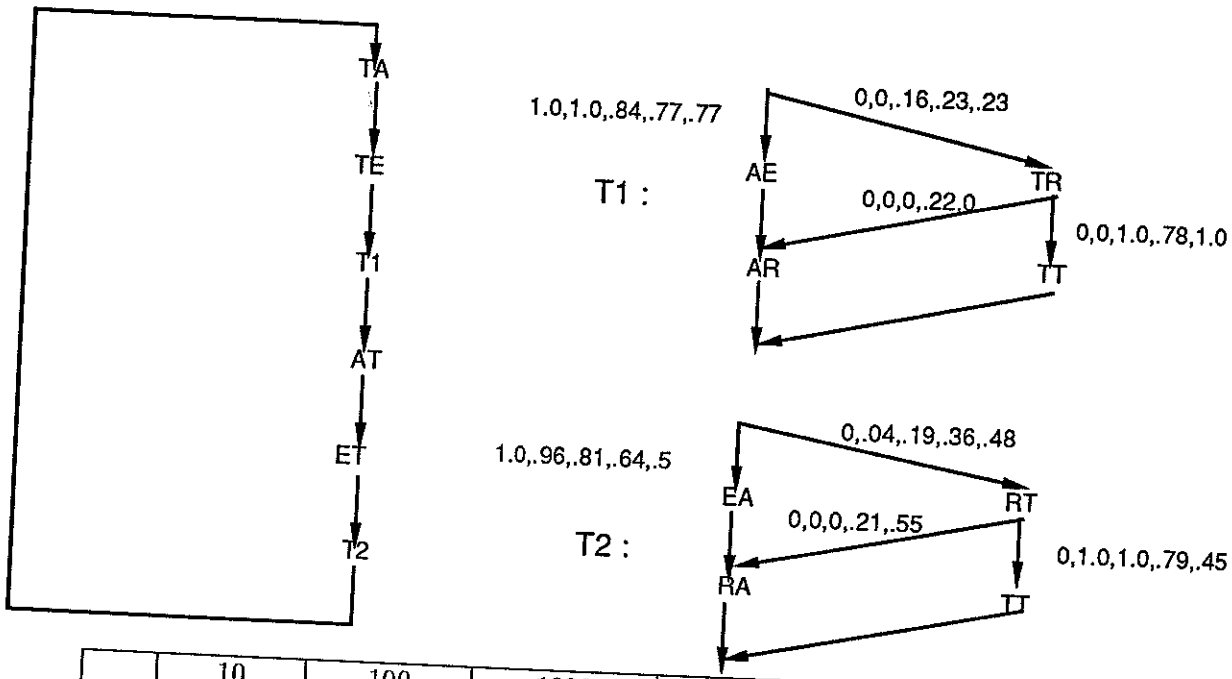
[13] Charles E. McDowell and David P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys 21*, No.4, Dec. 1989.

[14] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," *IEEE Transactions on Parallel and Distributed Systems 1*, No. 2, April 1990.

[15] Sekhar R. Sarukkai. *Performance Visualization and Prediction of Parallel Supercomputer Programs: An Interim Report*, TR 318, Computer Science Dept., Indiana Univ., Nov. 1990.

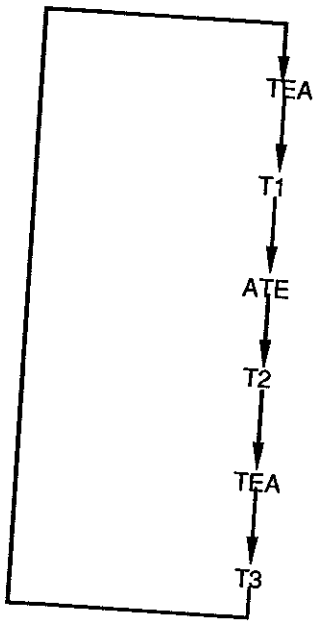
[16] David Socha, Mary L. Bailey, and David Notkin, "Voyeur: Graphical Views of Parallel Programs," *in Proc. of Workshop on Parallel and Distributed Debugging*, ACM SIGPLAN Notices 24, 1, Jan. 1989.

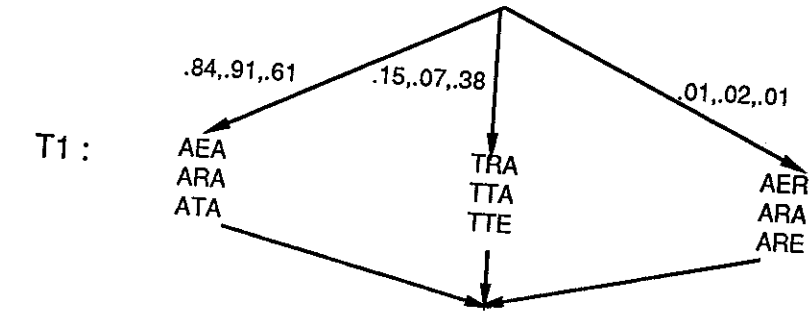
[17] Allan M. Tuchman, Michael W. Berry, "Matrix Visualization in Design of Numerical Algorithms," *ORSA Journal of Computing 2*, No. 1, Winter 1990.

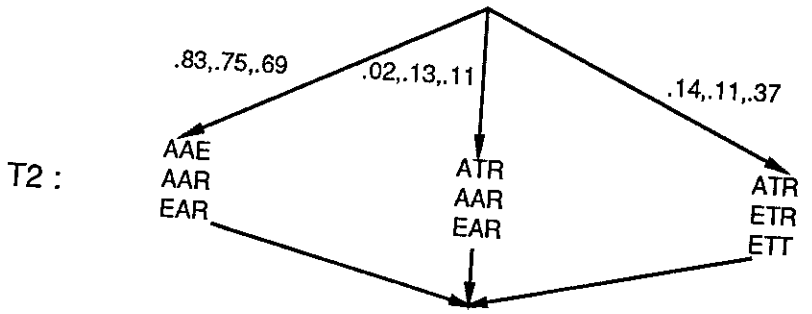


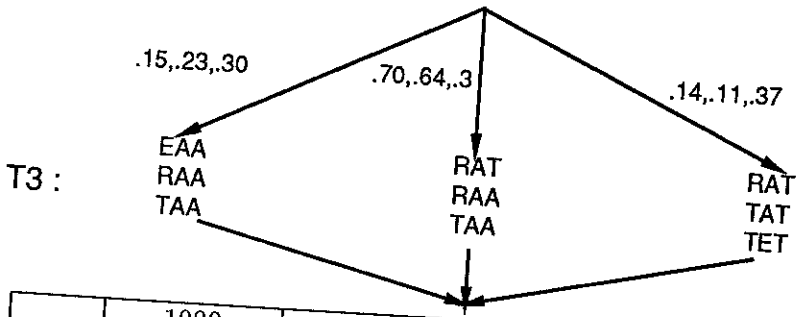
	10	100	1000	10000	100000
TE	21(0.92)	183(13.59)	1807(37.3)	18237(48.98)	182491(32.32)
AE	8(63.43)	9(41.22)	40(76.4)	47(95.79)	47(77.77)
AR	23(4.93)	23(5.02)	22(4.03)	21(3.06)	23(4.71)
AT	3(1.84)	3(2.07)	6(7.77)	9(8.83)	10(9.75)
ET	21(0.9)	183(10.25)	1807(25.84)	18150(1186.58)	181650(10896.05)
EA	8(64.2)	12(66.16)	36(72.0)	158(1404.72)	1348(14109.18)
RA	23((6.58)	23(4.77)	22(5.5)	22(3.54)	20(5.61)
TA	3(1.93)	4(3.76)	7(8.5)	8(7.82)	7(6.56)
TR	0(0)	15(7.28)	18(2.53)	10(7.56)	18(3.05)
TT	0(0)	52(12.06)	50(9.18)	403(2369.5)	2025(17406.8)
RT	0(0)	17(5.29)	18(2.34)	10(8.95)	8(7.35)
T1	0(0)	0(0)	59.74(49.59)	64.44(74.84)	63.89(31.32)
T1	0(0)	0(0)	60.55(51.6)	153.85(1217.72)	833.92 (10949.15)

Figure 6: Empirical model of two philosophers generated by Chitra.



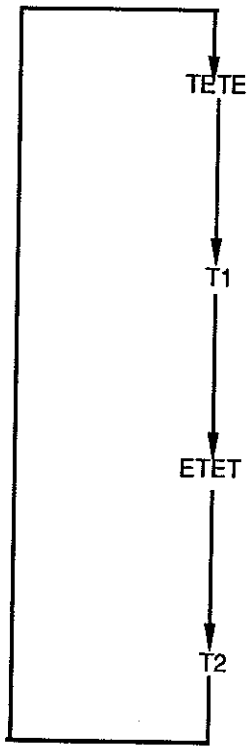






	1000	10000	20000
ATE	1816(22.53)	18239(34.21)	36482(26.08)
T1	50(68.57)	53(69.36)	51(67.55)
EAT	1808(20.04)	18238(37.06)	36476(25.15)
T2	51(68.83)	54(71.02)	73(64.87)
TEA	1810(51.63)	18245(34.74)	36338(1767.08)
T3	55(66.91)	55(65.86)	190(1623.61)

Figure 7: Empirical model of three philosophers generated by Chitra.



	1000	10000	20000
TETE	1762(92.87)	18119(1090.54)	36427(195.57)
T1	144.7(129.4)	199.11(1088.8)	188 (287.07)
ETET	1770(88.56)	18095(1254.86)	36320(1340.36)
T2	133.31(143.53)	279.69(1522.43)	296.03(1660.29)

Figure 8: Empirical model of four philosophers generated by Chitra.

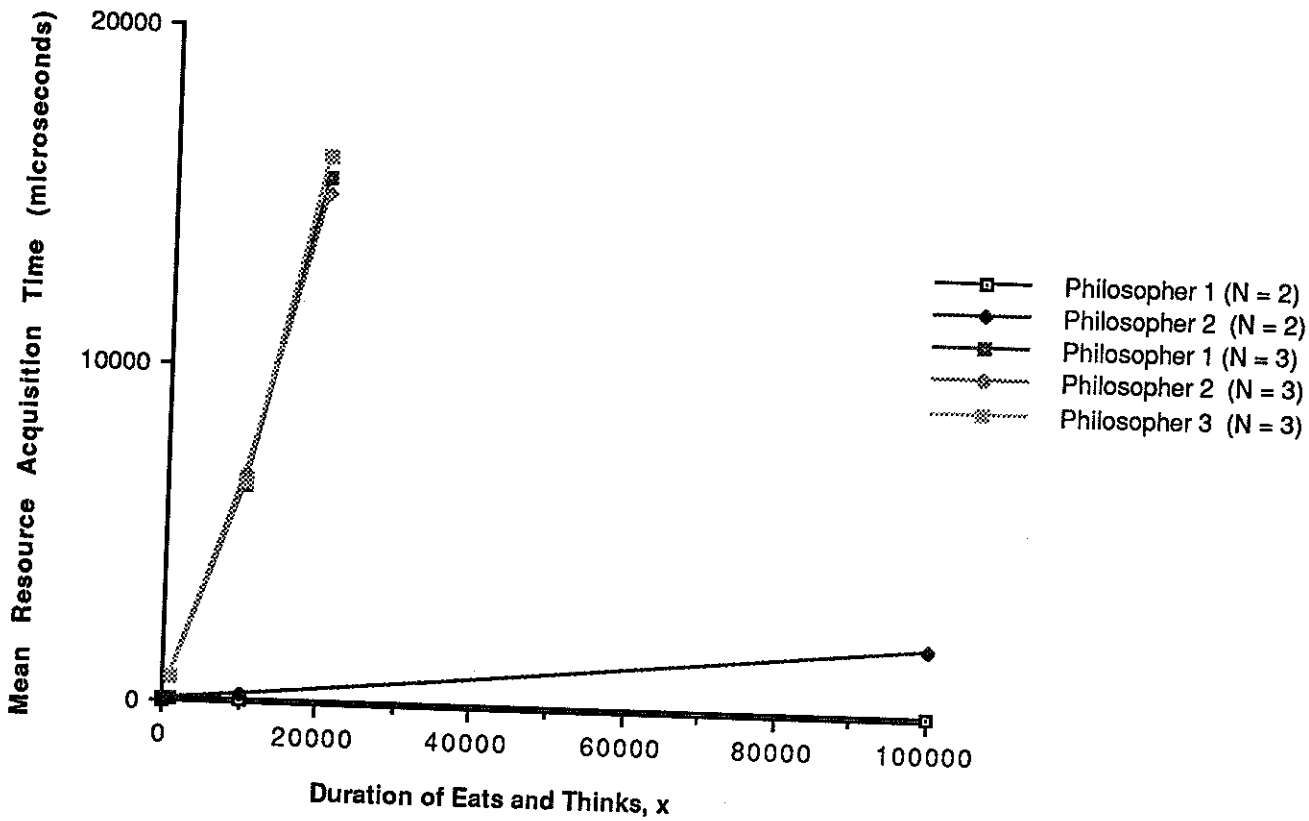


Figure 9: Prediction of resource acquisition time per thread as a function of x, for $N = 2$ and 3, based on empirical models in Figures 6 through 7.

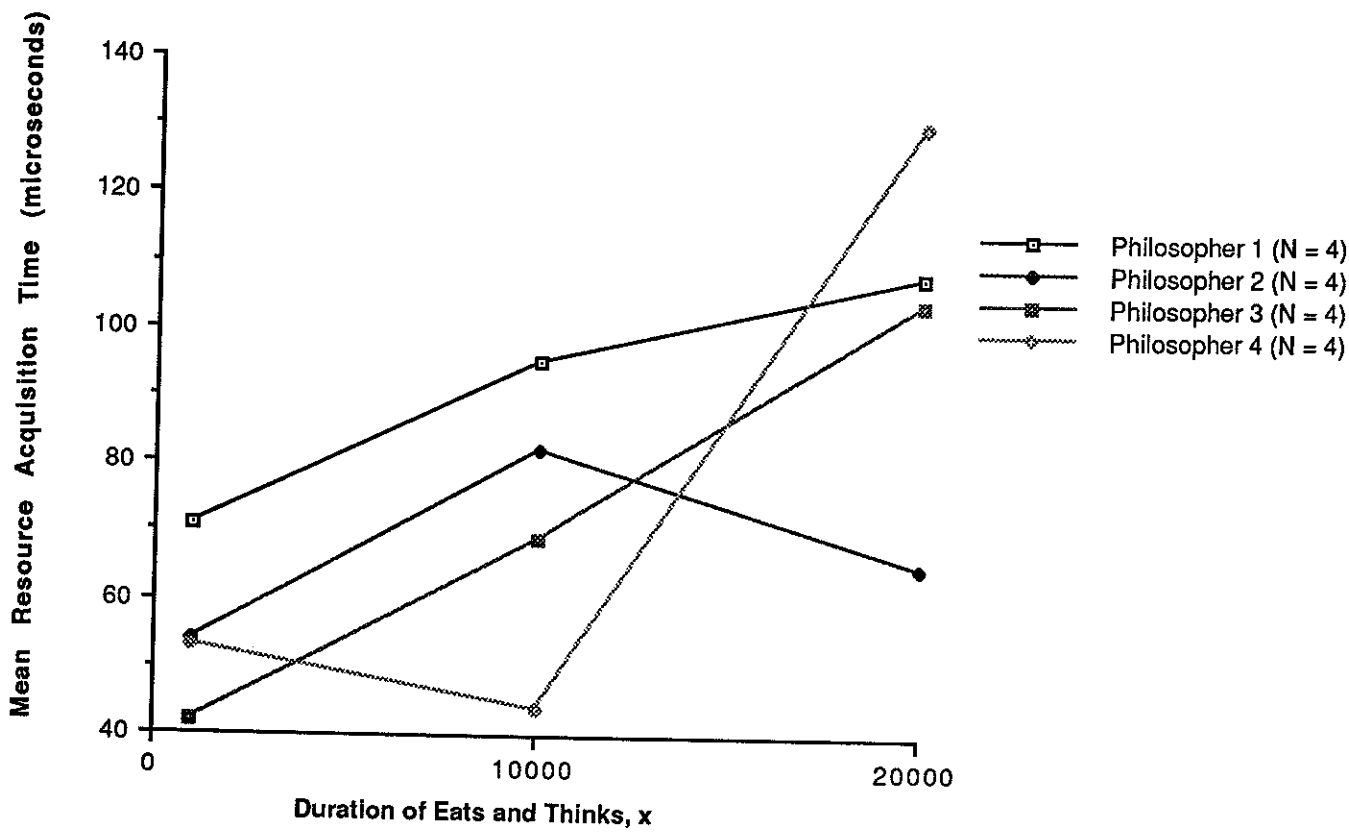


Figure 10: Prediction of resource acquisition time per thread as a function of x, for $N = 4$, based on empirical model in Figures 8.

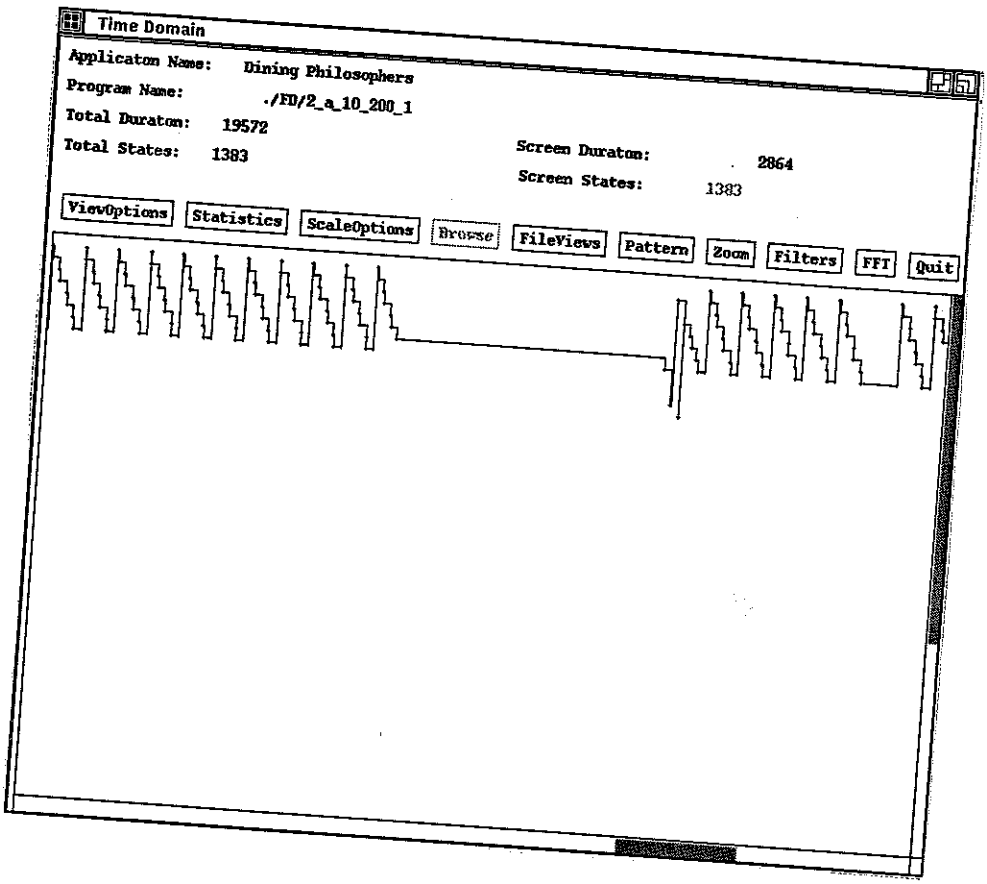


Figure 11: Illustration of perturbations in PES in time view. An interval of 2864 microseconds is represented. The two long horizontal lines are approximately of length 800 and 100 microseconds, and represent perturbations in the normal state transition sequence, probably due to clock and page fault interrupts, respectively.

croseconds (see Figure 11). Perturbations among processors are not synchronized. We attribute these perturbations to the Sequent's clock interrupts.

Measurement shows that the program occasionally experiences zero to two perturbations of duration 100 microseconds between every to clock interrupt perturbations (see Figure 11). There is no pattern to these perturbations. Our hypothesis is that these perturbations correspond to page faults. We confirmed this hypothesis by constructing a test program of a single thread that just executed a loop whose body is a loop of multiplies and records the time required to