Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event, and Frequency Domains

Marc Abrams, Naganand Doraswamy, and Anup Mathur

TR 92-24

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

May 14, 1992

Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event, and Frequency Domains

Marc Abrams
Naganand Doraswamy
Anup Mathur
TR 92-24 (revised)

Virginia Polytechnic Institute and State University
Department of Computer Science
Blacksburg, VA 24061-0106

June 19, 1992

Abstract

Chitra analyzes a program execution sequence (PES) collected during execution of a program and produces a homogeneous, semi-Markov chain model fitting the PES. The PES represents the evolution of a program state vector in time. Therefore Chitra analyzes the time-dependent behavior of a program. The paper describes a set of transforms that map a PES to a simplified PES. Because the transforms are program-independent, Chitra can be used with any program. Chitra provides a visualization of PES's and transforms, to allow a user to visually guide transform selection in an effort to generate a simple yet accurate semi-Markov chain model. The resultant chain can predict performance at program parameters different than those used in the input PES, and the chain structure can diagnose performance problems.

1 Introduction

Formation of models of physical behavior underlies science. Yet our ability to construct behavioral models of parallel and distributed programs is very limited. Fast, simple, and accurate model construction is likely to be central to the success of parallel and distributed software development. For example, models can predict performance metrics; suggest processor mappings; assist in diagnosing software performance problems; and identify parameter values, such as buffer sizes, that optimize performance.

The following definitions are used throughout the paper. A *thread* (or *process*) is a code fragment scheduled by an operating system. A *program state* is an n-tuple providing an instantaneous description of a program. A program state component is a scalar quantity with a finite domain, such as the value of a local or global variable, the virtual address of a data object

that will be referenced in the instruction which will next complete its execution in a thread, the virtual address of the instruction which will next complete its execution in a thread, or a function of these quantities. The decision of what to include in a program state depends on the objective of a performance study. For each execution of a program, there corresponds a *program execution sequence* (PES), representing the sequence and entrance times of program states that a program passes through during execution. Each PES component is an ordered pair whose components are a program state and a non-negative integer entrance time. One instruments a program to collect an estimate of a PES, as described in Section 7. The collected data is an estimate because the presence of instrumentation alters the PES. Associated with a program is a set of *program parameters*, which are data items that may be changed each time a program is run. Program parameters include input data to the program and run-time constants, such as the number of threads.

Example 1 *Dijkstra's dining philosophers problem is used in Section 5 as one case study in this paper. The problem consists of N philosophers, numbered* $0, 1, \ldots, N-1$. *Each philosopher alternately thinks and eats. A philosopher requires utensils to eat, and acquires and later releases utensils serially. The problem 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; therefore there are* N *utensils, and each philosopher requires two utensils to eat. The duration of eating and thinking time for all philosophers is the constant* x.

Each philosopher is represented by an operating system thread (Figure 1). Let utensils be numbered $0, 1, \ldots, N-1$. *Utensils are implemented by an array of locks, L[N]. A thread acquires a utensil by acquiring the corresponding lock. The* Think *and* Eat *operations are implemented as a loop that repeatedly multiplies the same operands;* x *is the number of repetitions. Macros* f *and* g *define the order in which utensils are acquired and released. Orienting philosopher and utensil numbers according to a clock face, even (odd) numbered philosophers acquire and release the clockwise (counterclockwise) utensil first.*

The program state *consists of* N *components, corresponding to* N *threads. Each* program

```
#define f(i) (i is odd) ? L[i-1] : L[i]
#define g(i) (i is odd) ? L[i] : L[(i-1)%N]
extern Spinlock L[N];

while (TRUE) {
        Think;              /*State T*/
        Lock(f(i));         /*State A1*/
        Lock(g(i));         /*State A2*/
        Eat;                /*State E*/
        Unlock(f(i));       /*State R1*/
        Unlock(g(i));       /*State R2*/
        }
```

Figure 1: Code of philosopher i, for $0 \leq i < N$.

state component *denotes which statement in the corresponding thread will next complete its execution, as specified in Figure 1. The six values are thinking, acquiring one utensil, acquiring a second utensil, eating, releasing one utensil, and releasing the second utensil, denoted* T, $A1$, $A2$, E, $R1$, *and* $R2$, *respectively. For example, with* $N = 4$, *in program state* $ETET$ *philosophers zero and two eat while the remainder think. The initial portion of a sample* PES *measured from an implementation of Figure 1 is* ($A2T$,*1550*), (ET,*1560*), ($EA1$,*1570*), ($R1A1$,*1585*), ($R2A1$,*1597*), ($R2A2$,*1601*), ($TA2$,*1607*), (TE,*1621*), ($A1E$,*1633*), ($A1R1$,*1651*), ($A2R1$,*1668*), ($A2R2$,*1669*), ($A2T$,*1685*), (ET,*1686*), ($EA1$,*1695*), ($R1A1$,*1711*), ($R2A2$,*1723*), ($TA2$,*1731*), (TE,*1735*), ($A1E$,*1745*), ($A1R1$,*1760*). *The program has two* parameters*:* N *and* x.

A program state with n components implies a state space size exponential in n. In parallel and distributed programs, the value of n is often proportional to the number of threads, which makes direct state space analysis intractable.

Usually, only a proper subset of the state space and all possible state transitions occur in the set of all possible PES's for a program, called *feasible* program states and transitions. For example, consider a parallel program in which each program state component points to the next instruction to be executed in a corresponding thread. The presence of a critical section precludes, first, occupancy of a program state in which two or more components point to an instruction within the critical section and second, a state transition in which multiple threads simultaneously enter the critical section. The feasible state space size is also reduced if we only wish to characterize the program behavior for certain ranges of program parameter values.

Finally, the feasible state space size is also diminished by functional relationships that may exist among program state components. For example, if a program state represents the values of n global variables (e.g., $n = 2$: number of full, number of empty buffers), and the program maintains an invariant property (e.g., the sum of full and empty buffers is 100), then the state space is limited to program states satisfying the invariant.

Example 2 *In Example 1, a program state in which philosophers that share utensils simultaneously eat (i.e., EE for $N = 2$) is infeasible.*

Success in modeling program behavior hinges on identifying feasible program states and transitions for parameter ranges of interest. One way to discover this subset is to analyze program source code alone. The method works for limited classes of software (see for example [6, 15, 24]).

1.1 Problem Definition

There is an alternative to constructing a program behavior model using only source code. We propose a system that takes as input a PES collected from execution of a program and produces as output an empirical model in the form of a homogeneous semi-Markov chain [8], consisting of a set of chain states, state occupancy time distributions, and state transition probabilities, such that a sample trajectory generated by the empirical model closely matches the input PES. Combining the models of a set of PES's produces the final model of program behavior. This is the purpose of *Chitra*[1]. An advantage of constructing a model from PES's is that it can be applied to any program. The disadvantage is that the model is only guaranteed to fit the observed PES's, and thus successful modeling depends upon observing a sufficient number of PES's.

The remainder of the paper is organized as follows. The next section introduces the program behavior model and analysis methodology underlying Chitra. Section 3 describes the PES transforms provided by Chitra. Section 4 describes the role of visualization in applying transforms. Section 5 contains case studies, applying Chitra to one distributed and one parallel program: a commercial TCP/IP communication protocol product and a dining philosophers program, respectively. A comparison of Chitra to performance visualization tools and issues surrounding

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

collection of PES's are presented in Sections 6 and 7, respectively. Finally Section 8 discusses future directions for further development of Chitra.

2 Chitra

The two key design choices in Chitra are the choice of empirical program behavior model, and the methodology used to construct a model from a set of PES's. The program behavior model is a homogeneous, continuous time semi-Markov chain, discussed in Section 2.1. The methodology to build the model subjects a PES to three forms of analysis, discussed in Section 2.1: conventional data analysis techniques, such as those used in signal processing and in post processing of simulation output; conventional program visualization methods; and novel techniques to graphically edit a program visualization using a pointing device to define patterns and aggregate states.

2.1 Choice of Empirical Program Behavior Model

An "ideal" program model represents each unique *program state* by a *model state*. The model requires a transition function that maps the current model state to a subsequent model state. The transition function must allow deterministic as well as probabilistic transition functions. Deterministic transition functions represent the execution of program statements. For example, 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. The model transition from the state representing the program just before execution of the conditional branch to the state representing the program at the branch target must be deterministic. In contrast, probabilistic transition functions correspond to random events in the environment in which the program is run; examples include asynchronous events (e.g., context switches, page faults, and other interrupt-driven events); contention for resources (e.g., processor cycles, cache lines, memory blocks, communication media); nondeterministic programming language constructs; asynchrony among processor clocks; and completion of program actions that are functions of input data to the program.

No solution to the ideal model exists in the literature. However, combining deterministic and probabilistic transition functions as the ideal model requires has been used in the literature on mapping program source code to a model for certain classes of programs (e.g., see [15, 24]).

Chitra uses as its program behavior model a homogeneous, continuous time semi-Markov chain. A semi-Markov process models general state occupancy time distributions, appropriate for software. At present Chitra uses a homogeneous process, which is sufficient for the programs modeled in Section 5. In theory, software obeys the embedded Markov chain assumption, which requires that the state transitions be based only on the current program state. In practice, one often simplifies the program state to reduce the state space size at the expense of violating the Markov property. Chitra's transforms, discussed next, can alleviate this tradeoff.

3 Transformations and Chain Generation

Upon initiation, Chitra can map the input PES to a semi-Markov chain model. However, this is not normally done, because a sample trajectory generated from this model will not closely match the input PES. To produce a more closely matching model, a user must *transform* the input PES using the transforms described below and then generate a model of the transformed PES. The process of PES transformation and model generation can be iterated to produce a sufficiently good model. Transformations can be undone to allow a user to try alternate sequences of transforms.

Chitra represents a PES as a *Symbol Execution Sequence* (SES). A SES consists of two sequences: a *symbol sequence* and an *occupancy time sequence*. The first occupancy time sequence element corresponds to the first symbol sequence element. An analogous property holds for all other elements; therefore the two sequences have identical length. The domain of symbol sequence elements is a set of symbols, denoted Σ. The set of all program states in the PES input to Chitra is a subset of Σ. The domain of occupancy times is the non-negative integers. Denote a PES as $(s_0, e_0), (s_1, e_1), (s_2, e_2), \ldots, (s_{m-1}, e_{m-1}), (s_m, e_m)$. Upon initiation, Chitra represents the PES by a SES whose symbol sequence is $s_0, s_1, s_2, \ldots, s_{m-1}$ and whose occupancy time sequence is $e_1 - e_0, e_2 - e_1, \ldots, e_m - e_{m-1}$. The SES omits the last PES program state because the state occupancy time cannot be computed.

Example 3 *The symbol sequence representing the PES in Example 1 is* $A2T$, ET, $EA1$, $R1A1$, $R2A1$, $R2A2$, $TA2$, TE, $A1E$, $A1R1$, $A2R1$, $A2R2$, $A2T$, ET, $EA1$, $R1A1$, $R2A2$, $TA2$, TE, $A1E$*. The occupancy time sequence is* $10, 10, 15, 12, 4, 6, 14, 12, 18, 17, 1, 16, 1,9, 16, 12, 8, 4, 10, 15.$

3.1 Transforms

A *transform* maps one symbol and one occupancy time sequence, called the *input* sequences, and denoted I_s and I_o, respectively, to another (*output*) symbol and occupancy time sequence, denoted O_s and O_o, respectively.

Chitra provides four transforms: clipping, aggregation, projection, and filtering. Clipping deletes part of a PES from analysis. Aggregation, projection, and filtering collapse symbols or symbol sequences into *composite symbols*; they differ in the criteria used to select which states are collapsed.

Clipping: Let n_i and n_f denote two integers such that $n_i + n_f$ does not exceed the length of I_s. The clipping transform maps n_i, n_f, I_s, and I_o to O_s and O_o by deleting the initial n_i and final n_f elements of I_s and I_o.

Example 4 *Applying a clipping transform with $n_i = n_f = 8$ to I_s and I_o equal to the sequences given in Example 3 yields $O_s = A1E, A1R1, A2R1, A2R2$ and $O_o = 18, 17, 1, 16$.*

Aggregation: Let σ denote any symbol sequence, and s denote a symbol in Σ; s denotes a composite symbol. The aggregation transform maps σ, s, I_s, and I_o to O_s and O_o as follows. Initially set $O_s = I_s$ and $O_o = I_o$. Next, for each occurrence of σ in O_s, do the following. Let o denote the subsequence in O_o corresponding to the occurrence of σ in O_s. Replace σ in O_s by s. Replace o in O_o by the sum of the elements of o.

Example 5 *Let I_s and I_o be the sequences in Example 3. Replacing the sequence $\sigma = R2A2$, $TA2$, TE by $s = Z$ yields $O_s = A2T$, ET, $EA1$, $R1A1$, $R2A1$, Z, $A1E$, $A1R1$, $A2R1$, $A2R2$, $A2T$, ET, $EA1$, $R1A1$, Z, $A1E$ and $O_o =$ 10, 10, 15, 12, 4, 32, 18, 17, 1, 16, 1, 9, 16, 12, 22, 15.*

Projection: Let S denote any subset of Σ, and s denote a symbol in Σ; s denotes a composite symbol. The projection transform maps S, s, I_s, and I_o to O_s and O_o as follows. Initially set $O_s = I_s$ and $O_o = I_o$. Next, replace each element of O_s that is in S by s. Finally, for each subsequence $s, s, \ldots, s$ in O_s, do the following. Let o denote the subsequence in O_o corresponding

to the occurrence of $s, s, \ldots, s$ in O_s. Replace $s, s, \ldots, s$ in O_s by s. Replace o in O_o by the sum of the elements of o.

Example 6 *In the dining philosophers problem, the state space may be simplified by projecting the two acquire (respectively, release) states, A1 and A2 (R1 and R2), into a single acquire (release) state, represented by symbol A (R). This is accomplished by a set of projection transforms. (One example is* $S = \{A1R1, A2R1, A2R2\}$ *and* $s = AR$.*) The result, applied to Example 3, is* $O_s = AT, ET, EA, RA, TA, TE, AE, AR, AT, ET, EA, RA, TA, TE, AE$ *and* $O_o = 10, 10, 15, 22, 14, 12, 18, 34, 1, 9, 16, 20, 4, 10, 15$.

Filtering: Frequency filter transforms map a parameter p and input sequences I_s and I_o to output sequences O_s and O_o. A symbol in I_s will be *selected by the filter* according to the following rules:

Time domain filtering: Let the sum of all times in I_o be T. For each symbol s, let $h(s)$ denote the sum of each occupancy time in I_o that corresponds to an occurrence of symbol s in I_s. A time domain filter selects all symbols s for which $h(s)/T$ is less than parameter p, which is a fraction.

Event domain filtering: An event domain filter transform selects all symbols that occur less than p number of times in I_s.

A *run* is a subsequence of I_s such that all of the run's symbols are selected by the filter, and the predecessor and successor symbol (i.e., the symbols preceding and following the subsequence) in I_s are not selected by the filter.

A filter transform performs the following mapping. Initially set $O_s = I_s$ and $O_o = I_o$. Let the number of distinct runs in I_s be n_r. Select n_r unique symbols not used in I_s; these are composite symbols. Replace all runs in O_s by a composite symbol such that all runs sharing the same predecessor and successor symbol are replaced by the same composite symbol. Replace all occupancy times in O_o that correspond to runs by the sum of the run occupancy times.

Example 7 *In the SES of Example 3, the fraction* $h(s)/T$ *for* $A1E$ *and* $EA1$ *is 15.7% and 14.7%, respectively. The fraction is less than 14.7% for all other symbols. Therefore applying*

a time domain filter with parameter p = 14.7% *to* I_s *and* I_o *representing Example 3 creates four runs, three of which are distinct, and yields* $O_s = T_1$, $EA1$, T_2, $A1E$, T_3, $EA1$, $T2$, $A1E$ *and* O_o = 20, 15, 48, 18, 44, 16, 34, 15. *Composite symbol* $T1$ *replaces the sequence* $A2T, ET$*; symbol* $T2$ *replaces one occurrence of* $R1A1, R2A1, R2A2, TA2, TE$ *and one occurrence of* $R1A1, R2A2, TA2, TE$. *Composite symbol* $T3$ *replaces* $A1R1$, $A2R1$, $A2R2$, $A2T$, ET.

Subsequent application of an event domain filter with $p = 2$ *yields* $O_s = T4$, $EA1$, $T2$, $A1E$, $T5$, $EA1$, $T2$, $A1E$, *and* O_o = 20, 15, 48, 18, 44, 16, 34, 15. *Composite symbols* $T4$ *and* $T5$ *replace symbols* $T1$ *and* $T3$, *respectively.*

3.2 Chain Generation

Chitra maps a SES to a semi-Markov chain model at the user's request as follows. Let OTHER be a symbol not in Σ. Symbol OTHER is appended to the symbol sequence, and time zero is appended to the occupancy time sequence. Let S denote the set of symbols in the SES; S is also the set of chain states. Chitra sets the sample mean and variance of each chain state to the corresponding statistics for all occupancy times in the SES for the corresponding symbol. Finally, Chitra sets the transition probabilities out of each chain state to the relative frequency of transition from the corresponding SES symbol to all other symbol in the SES.

When Chitra generates a model, it expresses the chain in terms of program states as well as composite symbols, and also provides a definition of each composite symbol as an acyclic semi-Markov chain consisting of program states appearing in the PES input to Chitra.

Example 8 *Generation of a chain from the SES in Example 7 resulting from both filter transforms yields the transition graph in Figure 2.*

4 Visualization

SES's and transforms have textual and visual analogs. Therefore three possible ways to decide which and in what order to apply transforms are (1) under user guidance, based on examination of a textual listing of a SES, (2) under user guidance, based on a visualization of a SES, and (3) automatically based on an algorithm. Chitra uses the second approach over the first because a single bitmapped display screen can visualize a SES requiring many sheets of paper for a textual

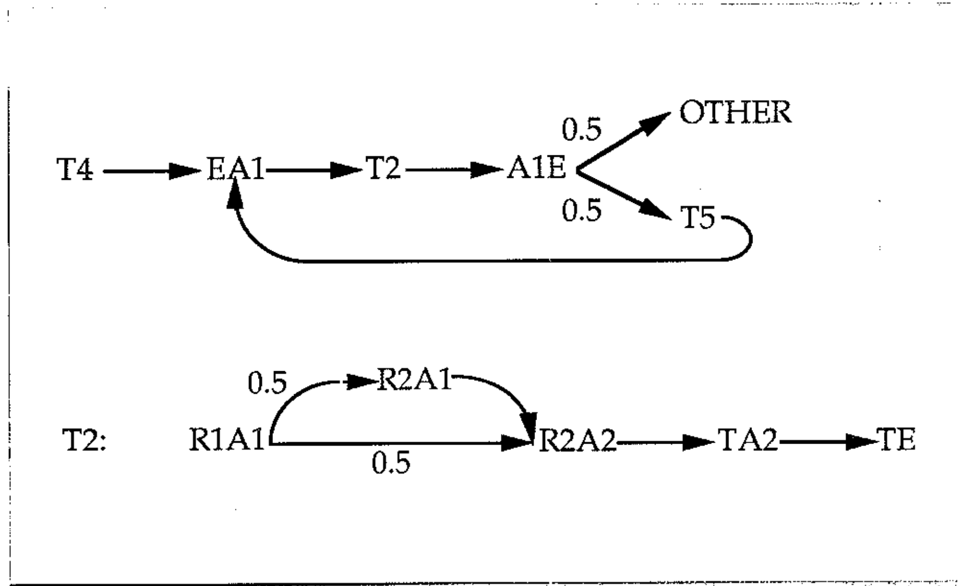


Figure 2: Chain generated by SES of Example 7. Composite symbols $T4$ and $T5$ denote symbol sequences $A2T, ET$, and $A1R1, A2R1, A2R2, A2T, ET$, respectively.

listing. Furthermore, state transitions patterns are easier to detect visually than textually. Approach (3) is left as an open problem, because different orders of transform application produce different semi-Markov chains.

4.1 Visualizing a SES

Chitra represents a SES as 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 set S of all symbols in a SES to a single dimension. This mapping is done when Chitra starts execution. Chitra assigns to each unique symbol in S a unique nonnegative integer, denoted $f(s)$, in a contiguous interval. Each integer is the y-coordinate value that represents the corresponding symbol. The mapping function used is arbitrary and does not affect the chain generated by Chitra.

In time view the point (x, y) is contained in the function if and only if symbol $f^{-1}(y)$ corresponds to entrance time x in the SES. 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 symbols, while event view conveys only the order of symbols. Event

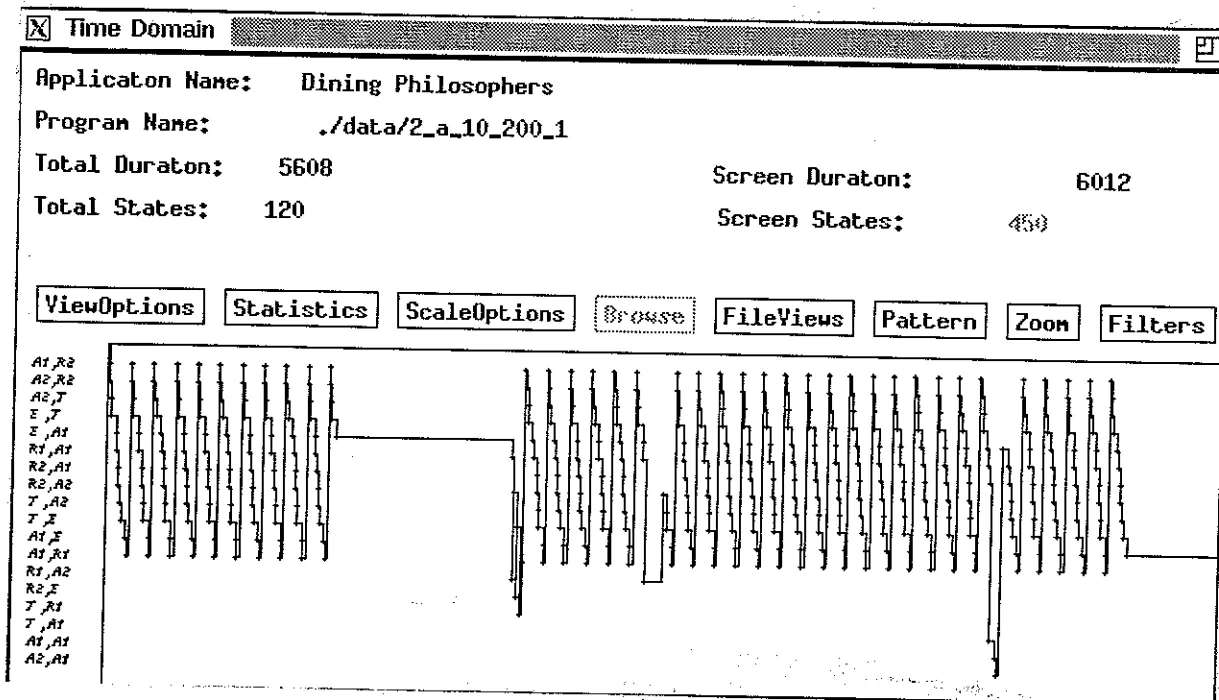


Figure 3: Time domain view of PES for two dining philosophers and eat, think time of 10.

view displays a greater number of symbols with equal fidelity, compared to time view, but with a tradeoff of a loss of occupancy time information. In time and event views graph line segments are parallel to the x axis because the symbol remains constant between transition times.

Example 9 *Figures 3 and 4 illustrate the time and event view of the entire PES to which the PES fragment in Example 1 belongs.*

Frequency view plots the discrete Fourier transform of all or part of the SES currently 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. Frequency view has two uses. First a concentration of energy at particular frequencies implies that a symbol transition in the original PES 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.

Example 10 *Figure 5 contains the frequency domain view corresponding to Figure 4. The peri-*

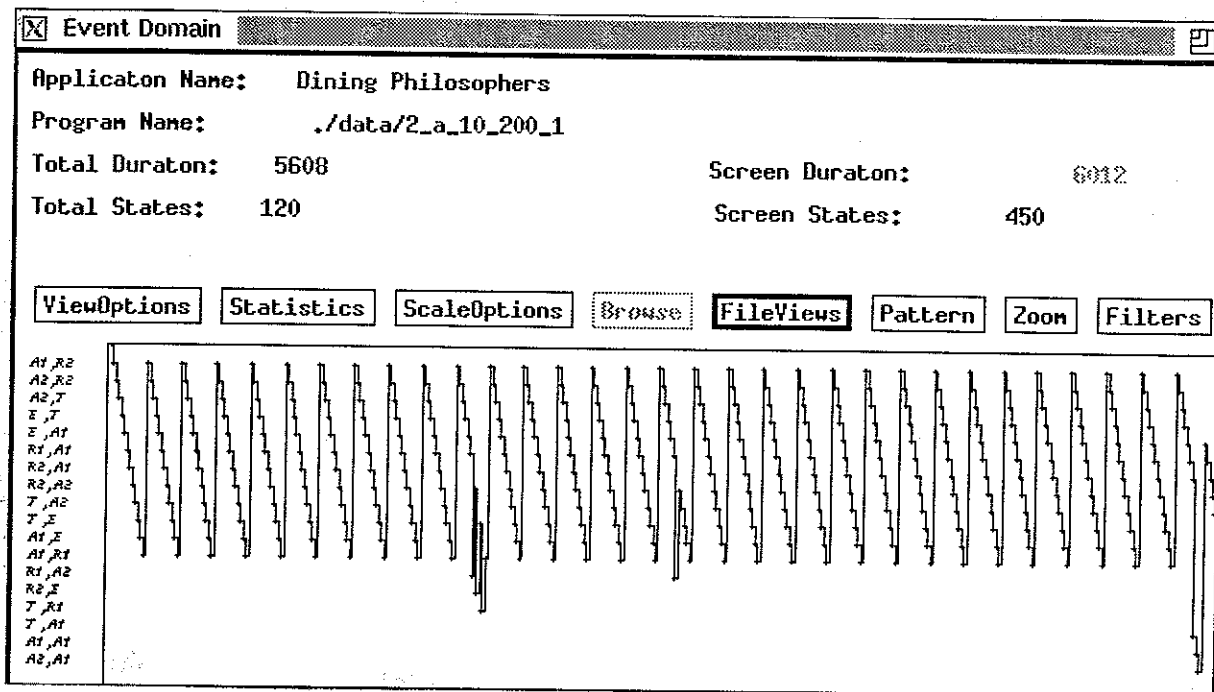


Figure 4: Event domain view of PES for two dining philosophers and eat, think time of 10.

odogram shows that almost all symbol transitions occur with one of five frequencies. Furthermore, the shape of the periodogram indicates that four of the frequencies are harmonics of the lowest frequency. Therefore we expect that the original PES contains predominantly deterministic behavior, as defined in Section 4.3.

Chitra provides various functions to assist with visualization that are typical of other visualization tools. These include zooming to magnify part of a graph, scrollbars to pan a window over a graph, and display of a histogram of symbol occupancy time for any symbol. Further information is contained in [4, 12].

4.2 Visualizing Transformations

Each time a user applies a transform, Chitra modifies the currently displayed SES to display the result of the transform. The user may also undo a transform if the result is unsatisfactory. The visual implication of each transform is discussed below; see [4] for further details.

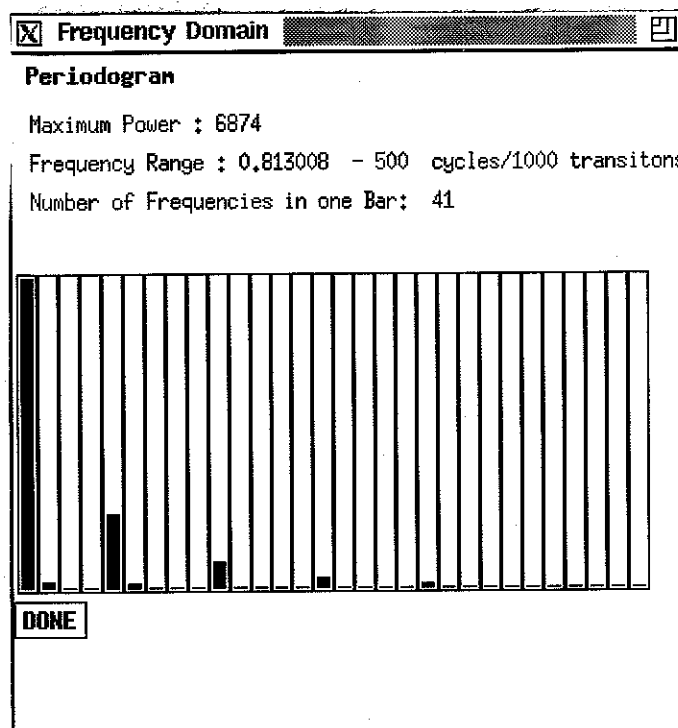


Figure 5: Frequency domain view of PES for two dining philosophers and eat, think time of 10.

Clipping transform: A user specifies a clipping transform in event or time view using a pointing device to select opposite vertices of a bounding box. The graph portion not enclosed by the box will be clipped. Visually, the clipped graph portions are no longer displayed and Chitra functions as though the user deleted all symbols in the SES *except* those corresponding to points displayed inside the bounding box.

Aggregation transform: The user specifies an aggregation transform using the pointing device to select a fragment of a SES by drawing a bounding box around a graph portion. Then Chitra pattern searches the current SES and overlays a rectangle shaded in a color not previously displayed on all graph fragments matching the selected sequence in the entire SES. 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. Application of the transform generates a new composite symbol with a previously unused y coordinate value that replaces colored rectangles.

Projection transform: A user specifies a projection transform textually rather visually.

Filter transform: Chitra assists selection of applying filter transforms as follows. For selecting the time threshold in a time domain filter, Chitra displays a histogram of $h(s)/T$ (defined in Section 3.1) for all symbols in the SES. For selecting the number of occurrences in an event domain filter, Chitra displays a histogram of the random variable representing the total number of occurrences of all symbols in the SES. Each composite symbol created by filter application is assigned a previously unused y coordinate value. The visual effect of applying a filter is to display the output SES, which does not contain points corresponding to filtered symbols, but does contain composite symbols. The use of a filter is illustrated in the case study in Section 5.2.2 (Figures 10 and 11).

4.3 Choice of Transforms

This section discusses the motivation behind the set of transforms included in Chitra.

The clipping transform is useful in eliminating the initial and final (transient) portions of a SES, during which threads are being initiated and terminated. Clipping is also useful when program execution consists of several phases, such that a different homogeneous chain model is appropriate within each phase. In this case application of the clipping transform allows isolation of a certain phase for further analysis.

The choice of filtering and aggregation transforms is motivated by a conjecture that underlies Chitra. A program is characterized as having *predominantly deterministic (random) behavior* if its ideal model (Section 2.1) is most likely to use a deterministic (probabilistic) transition function. The conjecture follows:

> An execution of a parallel or distributed program may sometimes appear predominantly deterministic, sometimes predominantly random, or alternate between deterministic and random behavior. A program with predominantly deterministic behavior may display a pattern of state transitions, such as periodic state transitions.

An objective of Chitra is to investigate this conjecture, and explore the extent to which PES's contain patterns.

Recall from Section 2.1 that an ideal program model allows both deterministic and probabilistic state transition functions. A model of a program displaying predominantly deterministic (random) behavior will most often make deterministic (random) transitions.

The aggregation transform is useful in identifying deterministic behavior and patterns of behavior. If a PES contains a perfectly periodic pattern of state transitions, then aggregating the symbols of the corresponding SES comprising one period produces a semi-Markov chain consisting of a single chain state and a transition from the state back to itself.

Given a PES that alternates between deterministic and random behavior, applying aggregation transforms to the corresponding SES can replace all deterministic subsequences by composite symbols. The resultant SES is then purely random, and therefore accurately modeled by a semi-Markov chain.

The filter transform can eliminate perturbations in a SES. For example, a SES may consist of the superposition of a periodic symbol subsequences and some random subsequences. The filter will remove the random subsequences, revealing the periodic pattern; this is illustrated in Section 5.2.2.

5 Case Studies

The use of Chitra to construct semi-Markov models and diagnose performance problems is illustrated using two examples.

5.1 TCP/IP Communication Protocol

The first example is a commercial implementation of the TCP/IP communication protocols on IEEE 802.3 local area network connected 80386-based hosts. Details of the study are discussed in [5]. The objective of this example is to illustrate the use of Chitra, and to diagnose a performance problem. The protocol exhibits a maximum throughput of about 1 Mbit/sec over a 10 Mbit/sec Ethernet to bulk data transfer applications using the Berkeley socket interface. In contrast, 68020-based hosts using another vendor's TCP/IP implementation achieves about 50% higher throughput. What accounts for the 80386-based host's inability to use no more than 10% of the available network bandwidth? Possible answers include an implementation detail of the protocol software or an inherent problem in the TCP or IP protocols, such as the sliding window.

5.1.1 Software Description

The software studied consists of a user thread that is sending a file to another host using the Berkeley socket interface. The thread makes a sequence of *write* calls, passing 1K bytes on each call. The 1K byte size is selected for study because the maximum throughput occurs at this write size. The socket interface invokes TCP, which contains a 2K byte buffer. TCP copies data from the user thread address space to its own address space, but as part of the user thread. When the TCP thread is scheduled, according to rules specified in the TCP protocol, the TCP thread removes data from the 2K byte buffer to form a segment, which is passed to the IP module. According to the rules specified in the IP protocol, the IP module creates a datagram containing the segment and passes the segment to a device driver. The driver creates a frame containing the datagram and transmits the frame over the network to the receiver.

Asynchronous with the sending of segments, the receiving host will periodically return acknowledgements. An acknowledgement is received by the sending host's device driver, and is then passed to IP and from IP to TCP.

Several subtleties arise in the protocol's functioning that a PES will exhibit. For example, the sending host TCP tries to optimize use of the network by matching segment sizes to the maximum data that will fit into a network frame. Hence the file is passed in 1K units to the socket interface, but the amount of file data in network frames is only loosely related to the 1K byte value. In addition, the rate at which data can be sent is controlled by the amount of free buffer space at the receiver, and the rate at which the user thread can call *write* is controlled by the amount of sending TCP buffer space. Therefore the user thread and sending TCP thread may block due to lack of buffer space.

We select two components for the program state: the segment size, in bytes, that is currently in transmission (or 0 if no segment is in transmission); and the software layer currently in execution: (state mnemonics are listed in parentheses): user thread (USR), copy to TCP buffer (CP), TCP protocol excluding copy to TCP buffer (TCP), IP protocol (IP), device driver (DRV), and idle (IDL). The idle state represents both by the sender waits for an acknowledgement and the interval during which the scheduler runs to context switch threads.

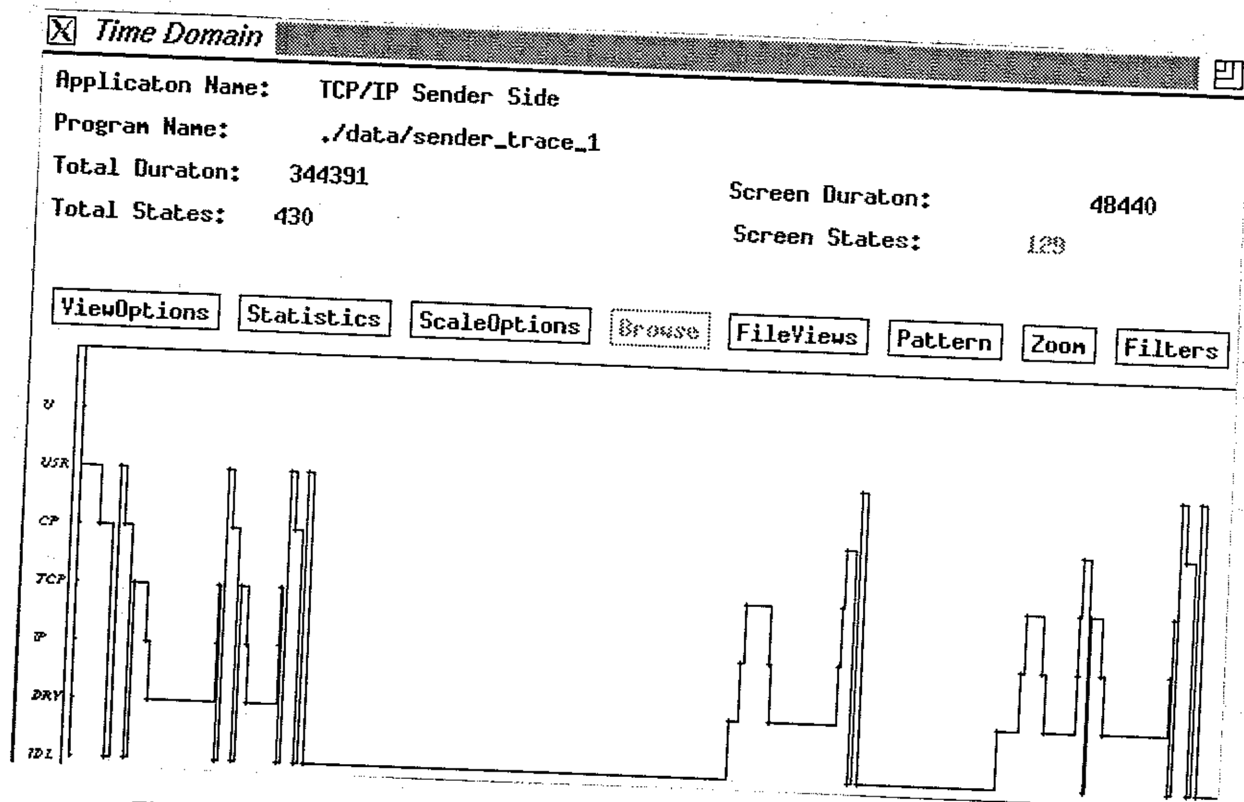


Figure 6: Time view of PES of TCP/IP-based file transfer, which is input to Chitra.

5.1.2 Chitra Analysis

The second program state component alone is visualized in the analysis. Figure 6 shows a 48 msec. portion of a PES. The y axis represents the six program states, and one more state representing an unknown state (U) at the start of the PES. There are five occurrences of the USR state, representing transfer of the first 5K of the file to the socket interface.

The first two 1K transfers at the user level occur in immediate succession, which fills the 2K TCP buffer. The next transitions are to states TCP, IP, DRV, and represents a sequence for sending a segment over the network. The other program state component at this point has value 1460; thus the occupancy in state DRV corresponds to transmission of 1460 bytes. Afterwards the function calls return to IP and then TCP. The TCP buffer now has 1460 bytes free, and a transition to the user thread next occurs to copy the third 1K portion of the file to TCP. The sequence TCP, IP, DRV, IP, TCP again occurs to send a 588 byte segment. The remaining PES continues in this manner. The two long occupancies of state IDL arise when the sender is waiting for an acknowledgment before continuing transmission.

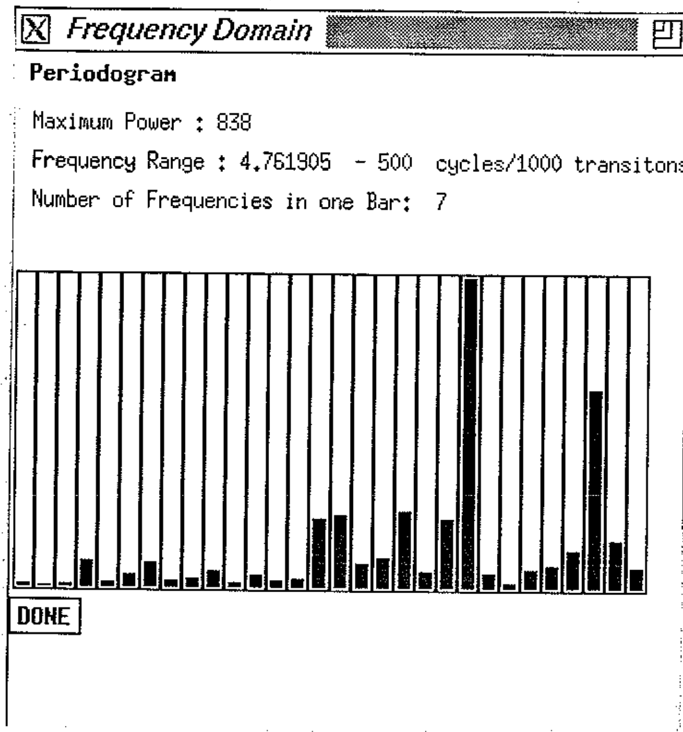


Figure 7: Frequency view of PES in Figure 6.

The next step is to apply transforms to reduce the trace. The frequency view (Figure 7) shows concentration of energy at one frequency, with a reduced concentration at a harmonic of that frequency, implying that a pattern is present in the entire 1/3 second PES. The next step is to use time view (Figure 8) to select a pattern and transform the PES. This step is repeated until the user can recognize no further patterns. Finally, Chitra generates a semi-Markov chain from the resultant PES Figure 9.

5.1.3 Model Properties

The following properties are evident from the model in Figure 9:

1. The protocol moves from IDLE to the segment send sequence 38% of the time, and to the acknowledgement reception sequence 22% of the time. Therefore an acknowledgement is sent by the receiver once for roughly every two frames received. Therefore the overhead of acknowledgements has negligible affect on performance.

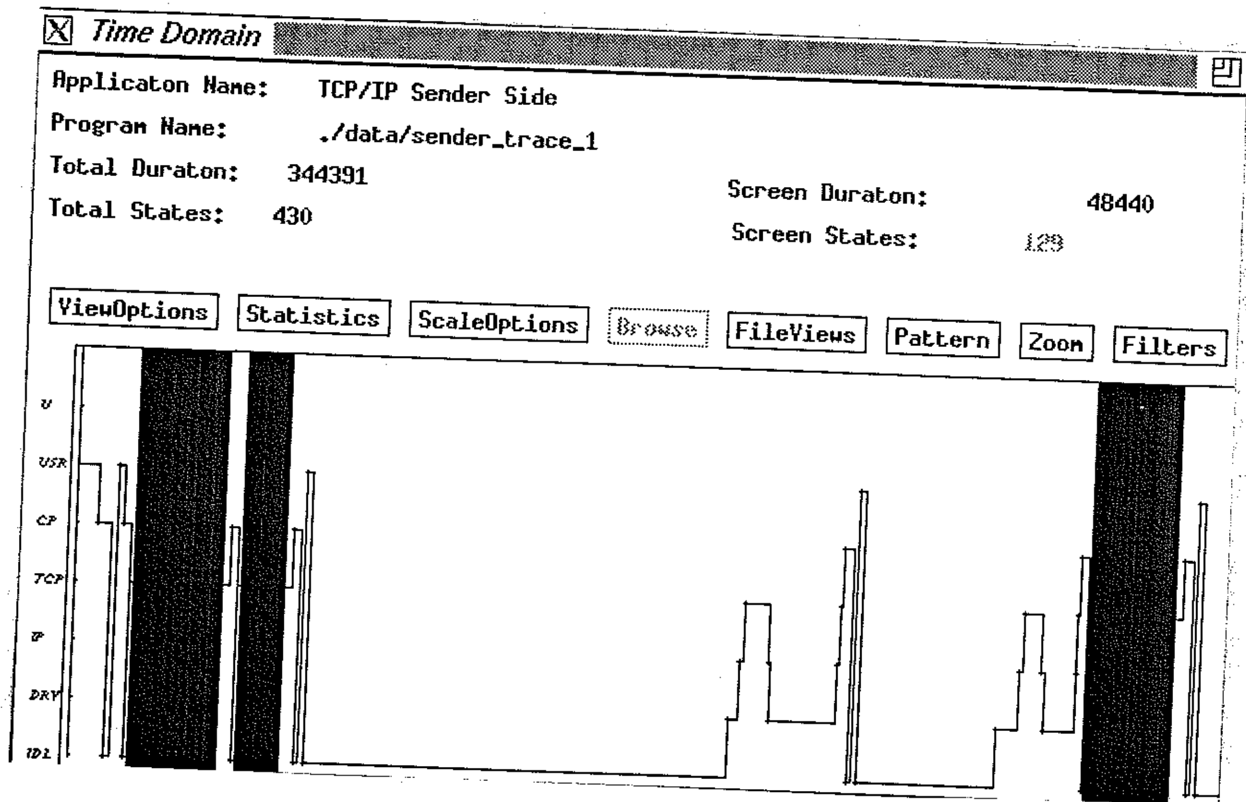
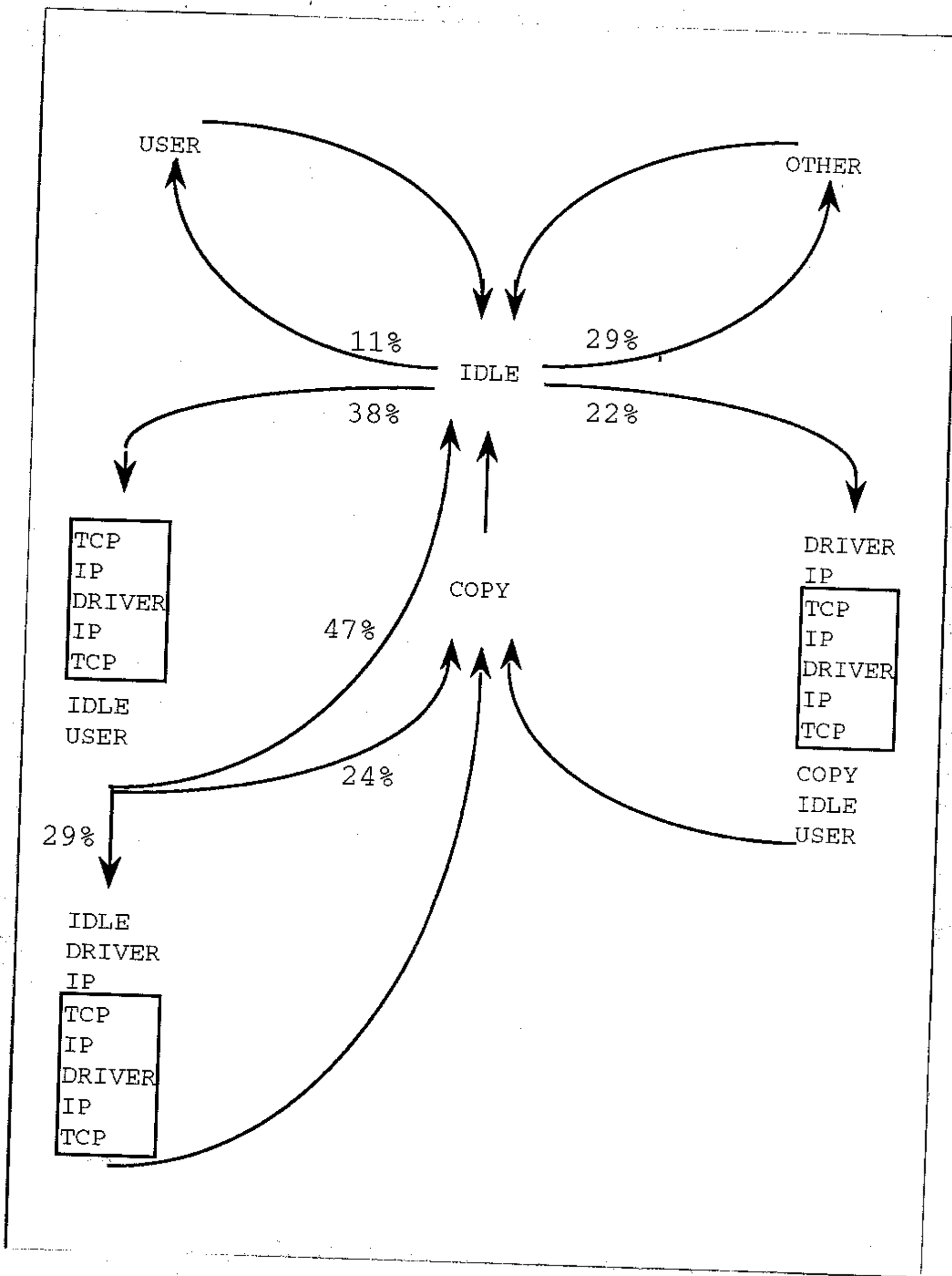


Figure 8: Aggregation transformation through pattern selection in TCP/IP PES.

2. The program spends most of its time (60.4%) in the idle state. This implies that it is not the protocol software, but some other factor that is limiting the throughput of the file transfer. To identify the factor, we look in the chain to see how often deterministic sequences of segment sends of different lengths are done. A single segment transmission followed by a visit to the idle state occurs on 40% (22% + 47% of 38%) of the transitions out of the idle state, while two back-to-back sends before a return to the idle state occurs on only 11% (29% of 38%) of the transitions out of the idle state. No more than two segment sends occur without waiting for an acknowledgement. This suggests that the source of the throughput limitation is that the sliding window mechanism is limiting the number of packets that can be sent before an acknowledgement is received. A modification to parameters associated with the sliding window (e.g., the receiver buffer space that determines the window size) is needed.



State	Occupancy Time Sample Mean, μsec.	Occupancy Time Std. Dev.	Percentage of entire trace
USER	222.3	107.1	2.7%
COPY	287.0	99.5	3.4%
TCP	312.5	330.7	8.0%
IP	84.1	69.6	2.6%
DRIVER	1178.6	859.43	22.9%
IDLE	2504.7	6820.5	60.4%

Figure 9: TCP/IP sending side model output by Chitra. Rectangles enclose state sequence that transmits segments on the network. Unlabeled arcs have transition probability 100%.

5.2 Dining Philosophers

The second example is the dining philosophers program from Example 1 (Figure 1). The example illustrates the use of Chitra to identify and diagnose unfair and inefficient parallel program behavior. Another motivation for this example is that chains similar to Chitra's output have been developed for the problem through simulation and provide a means to check the utility of Chitra.[1]

5.2.1 Software Description

The program is implemented using the Presto 0.4 thread [7] package on a Sequent Symmetry multiprocessor (Figure 1). Each philosopher is implemented by a non-preemptible Presto thread; the mapping from thread to processor does not change during program execution. A thread busy waits at an unavailable lock without relinquishing the processor.

PES's analyzed here were collected with only the dining philosophers program running on the Sequent. The empirical models presented later are based on PES's from at least three executions of the program at each parameter value. Repeated executions always produced differences in the PES's. We attribute the differences to clock interrupts and (very rare) page fault interrupts; this conclusion is justified in [4]. Varying the thread starting order did not alter the stochastic process generated by Chitra.

Study Objective: The study in this section has the following objectives. The *resource acquisition time* for a thread is defined as the time that the thread spends in local states $A1$ and $A2$. The acquisition time is of interest because it is proportional to the time that a thread blocks on a lock 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 objectives are to formulate an empirical model of the program that predicts the PES for $N = 2, 3$, or 4 and $10^3 \leq x \leq 10^4$; determine if the program reaches steady state; 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; and, if excessive acquisition times or unfair behavior is observed, diagnose the origin of the problem using Chitra's output model.

5.2.2 Chitra Analysis

Figure 3 shows the time view of 7 milliseconds of a PES for $N = 2$ and $x = 10$. A pattern of behavior with occasional perturbations is evident in time view. The initial state in the PES segment shown is TE. In the next transition, philosopher 1 begins acquiring its first utensil, and the second state is $A1E$. Then philosopher 2 releases its first utensil, and the third state is $A1R1$. The rest of the PES is interpreted similarly.

There are two types of perturbations, in time and in space. Perturbations in time elongate a state occupancy time. Perturbations in space cause a transition to a state that breaks the normal cycle, such as to the lower six states in the figure (states $R1A2, R2E, TR1, TA1, A1A1$, and $A2A1$).

Figures 10 and 11 illustrate the utility of frequency view. Figure 10 contains a four philosopher PES that appears to have a great deal of irregularity. The frequency domain view shows concentration of energy at two frequencies, harmonics of the lowest frequency, and some noise. This warrants use of filter transforms to attempt to expose an underlying pattern. A single application of an event domain filter removing states that occur less than 180 times (i.e., $p = 180$) yields the perfectly regular PES in Figure 11, consisting of periodic visits to states $TETE$ and $ETET$.

5.2.3 Model Properties

Chitra generates the semi-Markov chains shown in Figures 12, 13 for $N = 2, 3$, and 4, respectively, and x ranging from 10 to 10^6. For simplicity, the chains are expressed using the projection in Example 6, which replaces program state components $A1$ and $A2$ by A and $R1$ and $R2$ by R. Each figure represents the transition rate matrix of a stochastic process as a directed, weighted graph whose vertices each correspond to a stochastic process state and whose 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 12, upon entry to parallel

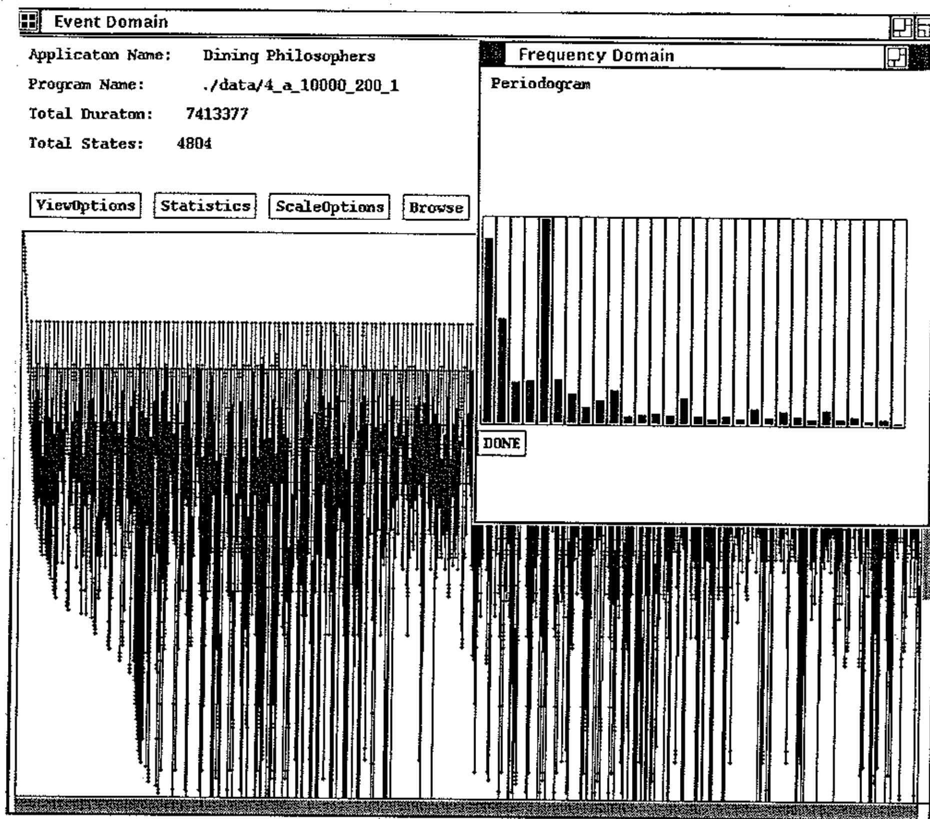


Figure 10: Portion of an original dining philosophers PES $N = 4, x = 10^5$

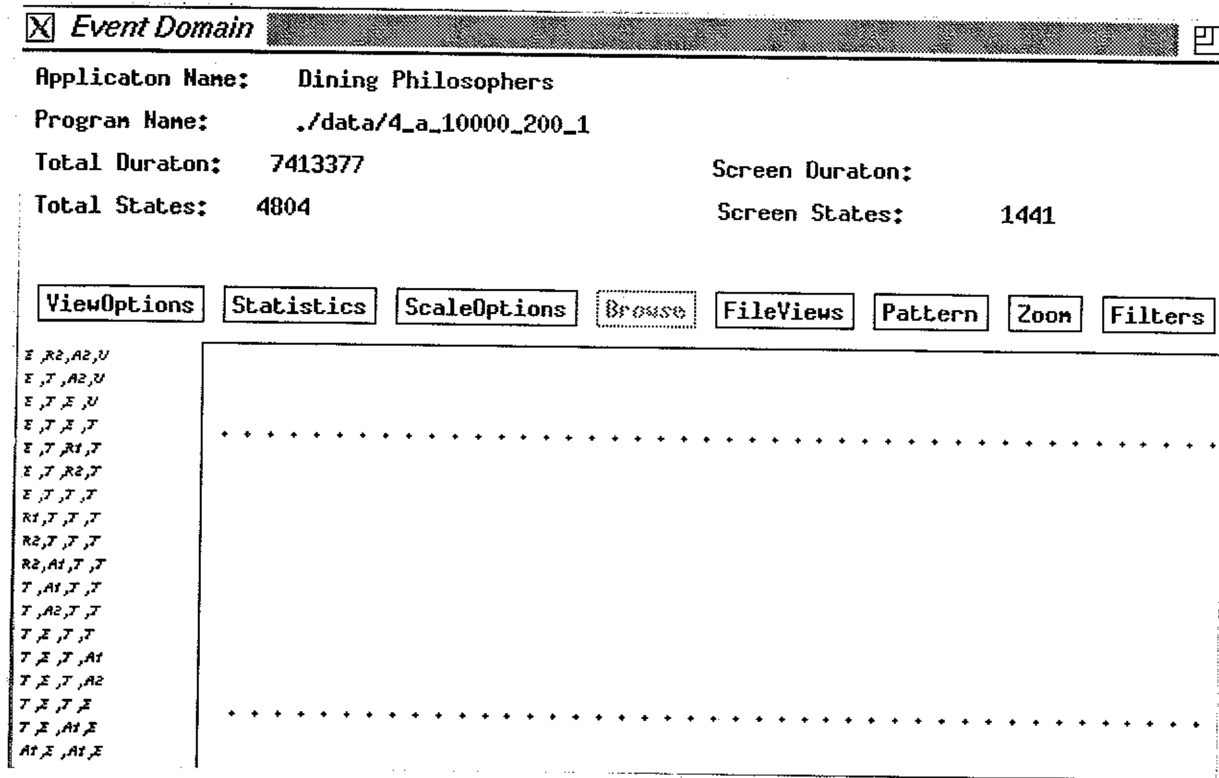


Figure 11: Result of event filter transformation filtering states occurring less than 180 times in the PES of Figure 10.

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 stochastic processes reveal the following:

1. The program reaches steady state for all values of x and N investigated. In addition, for each value of x and N, all PES's reach the same steady state cycle.

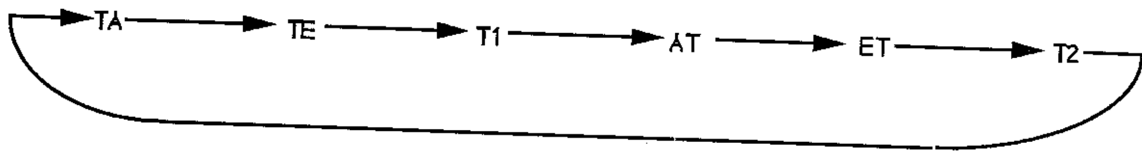
2. A single state transition graph can be used to represent all values of x with a given value of N. Therefore each stochastic process can be used to predict the program behavior at values of x that were not observed.

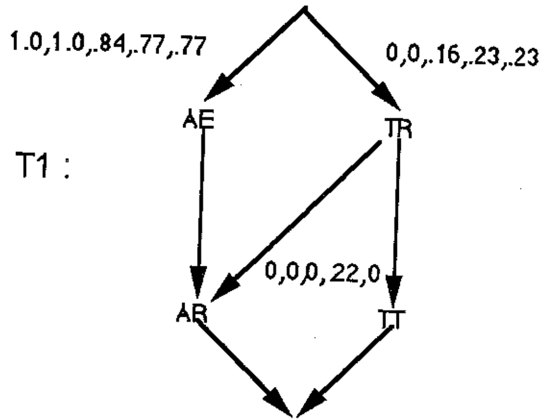
3. In each case the steady state behavior, as represented by a stochastic process, consists of a cycle of states. For example for $N = 2$ the cycle is TA, TE, $T1$, AT, ET, and $T2$, where $T1$ and $T2$ are aggregate symbols (Figure 12).

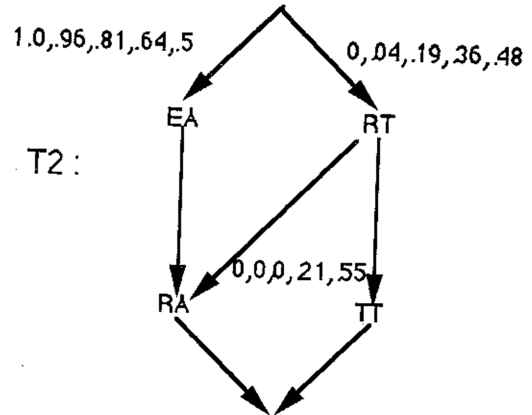
Figure 15 contains the desired performance metric of the sample mean of resource acquisition time per thread using all observations of states $A1$ and $A2$ in all PES's collected for all N. Two performance problems are evident from Figure 15 which can be diagnosed using the semi-Markov chains:

Efficiency: The 3 philosopher case is inefficient, because its resource acquisition time grows rapidly with x. The problem can be diagnosed with the 3 philosopher chain (Figure 13): the program spends virtually all of its time spent in states TEA, ATE, and EAT in which only one philosopher eats at a time. In contrast, in the two and four philosopher cases the program spends most of its time in states TE, ET and $TETE, ETET$, in which half the philosophers always eat (Figures 12,13).

Fairness: For $N = 2$, the program execution is not fair because philosopher 0 always waits a constant amount of time, while philosopher 1 waits for a time period that grows with the resource holding time, x. The origin of the problem is evident from the model in Figure 12. The three states in which philosopher 0 waits (AE, AR, AT) have occupancy times that do not tend to grow with x. In contrast, one of the states in which philosopher 1 waits,

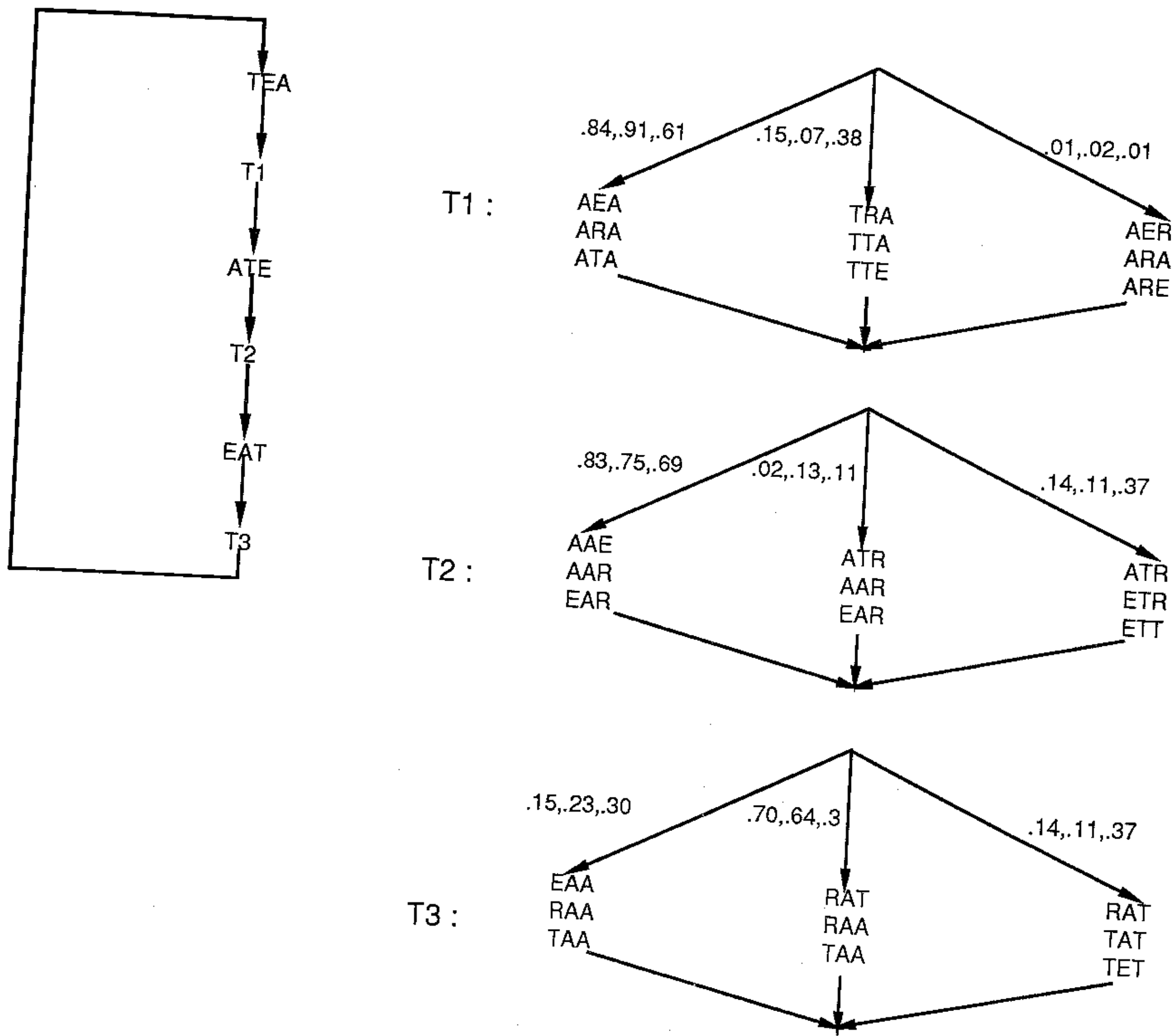






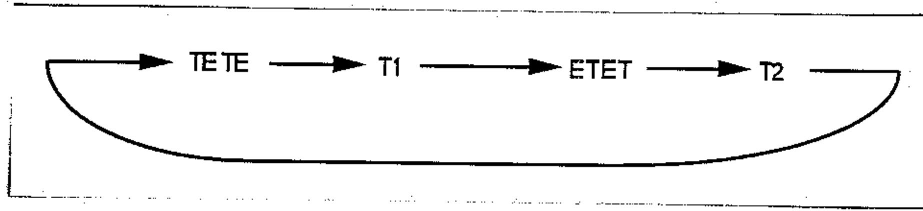
State	*Sample mean (µsec.) and variance of occupancy time*				
	10	100	1000	10000	100000
TE	21(0.92)	183(13.59)	1807(37.3)	18237(48.98)	182491(32.32)
AE	8(63.43)	9(41.22)	40(76.4)	47(95.79)	47(77.77)
AR	23(4.93)	23(5.02)	22(4.03)	21(3.06)	23(4.71)
AT	3(1.84)	3(2.07)	6(7.77)	9(8.83)	10(9.75)
ET	21(0.9)	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 12: Empirical model of two philosophers generated by Chitra. The chain consists of states $TA, TE, T1, AT, ET,$ and $T2$, where $T1$ and $T2$ are aggregate symbols defined as shown in the figure.



State	*Sample mean (μsec.) and variance of occupancy time*		
	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 13: Empirical model of three philosophers generated by Chitra. The chain consists of states $TEA, T1, ATE, T2, EAT$, and $T3$, where $T1, T2$, and $T3$ are aggregate symbols defined as shown in the figure.



State	Sample mean (μsec.) and variance of occupancy time		
	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 14: Empirical model of four philosophers generated by Chitra. States $T1$ and $T2$ are aggregate symbols.

EA, has an occupancy time that grows with x. Also occupancy time of state ET declines with an increase in x. Therefore the program allows some asynchrony in philosopher 0's transition from states E to R and philosopher 1's transition from T to A, while the 0's transition from T to A and 1's from E to R is synchronous.

6 Chitra Versus Performance Visualization Tools

Many software visualization tools have been built; their capabilities include program execution animation [9, 26], data structure animation [27, 11], display of interprocess communication [16, 26], program debugging [22, 17, 14], and performance evaluation [23, 19, 20, 14].

Visualization tools suitable specifically for performance evaluation include Moviola [14], IPS-2 [23], JED [18], HyperView [19], and an unnamed tool [25]. 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. The unnamed and unimplemented tool [25] 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

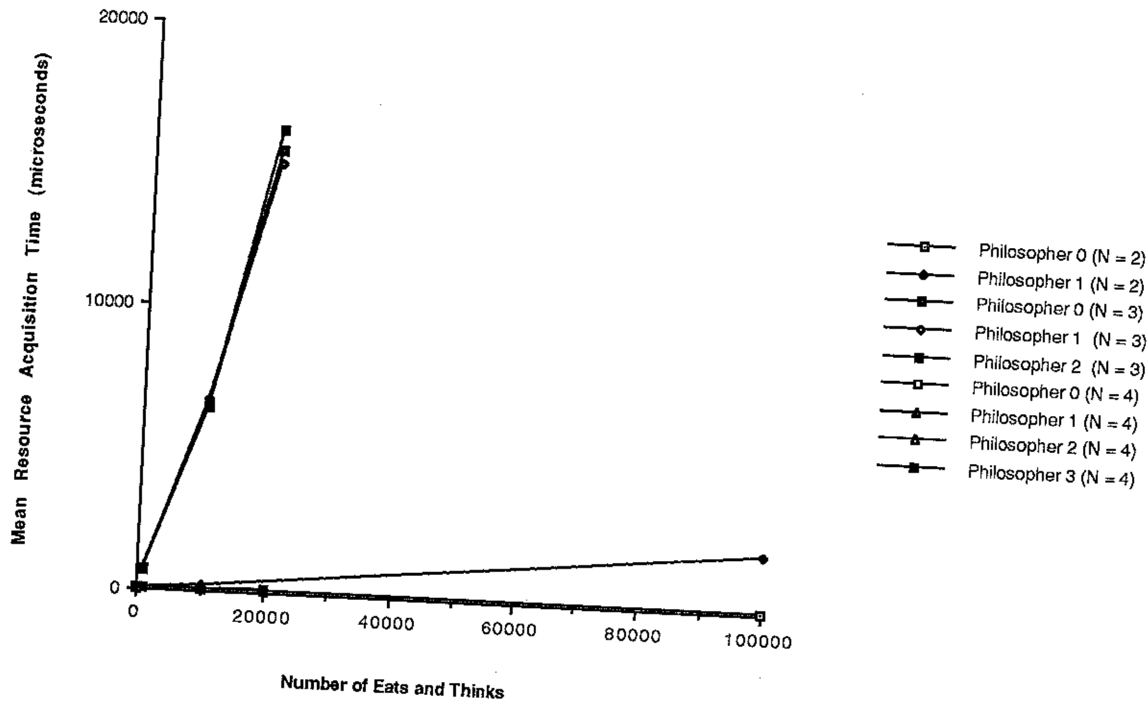


Figure 15: Sample mean of resource acquisition time per thread based on all observed PES's

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. The novel contributions of Chitra are its transformation of PES's, visualization of transformation, frequency domain analysis of software, and construction of a stochastic process from a set of PES's.

7 Collecting PES's

Two assumptions underlying Chitra's analysis of PES's are, first, that a global clock is available to timestamp software events when constructing PES's and, second, that collection of a PES is nonintrusive. The first problem exists in certain systems, including Chitra, while the second problem is universal to all monitoring and performance analysis systems. Some comments on the problems specific to Chitra follow.

PES's of the TCP/IP protocol (Section 5.1), which executes on a network of hosts, are collected using a hardware measurement instrument that is attached to each host and provides

a microsecond resolution global time base, solving the first problem.[2] The instrumented file transfer requires about 3% more time than an uninstrumented file transfer, so that the intrusion is limited.

In the study of Section 5.2, the Sequent multiprocessor provides a global clock with a microsecond resolution in the address space of all processors, obviating the first problem. With respect to the second problem, measurement intrusiveness is minimized through the following steps. 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.

For platforms without a global time base we timestamp events using local clocks, and after program execution completes we use linear regression to estimate the offset and drift of the clocks as described in [13]. This method works when the error of the estimates is small compared to the time between timestamped events. This is the case for the dining philosophers on a network of workstations; however it is not a general solution. We are currently experimenting with methods of aligning multiple trace files with local time stamps using Chitra's visualization to improve the offset and drift estimates.

8 Conclusions and Future Work

Visualization has had limited impact in analyzing software. The explanation for this, we believe, is that visualization has a dramatic impact on fields where it leads to a discovery 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. This is the objective of Chitra.

Chitra constructs an empirical model of program behavior through visually-guided appli-

cation of a sequence of transformations. Chitra provides a single, program-independent set of transforms that are demonstrated on two disparate examples in the paper. The motivation for constructing the semi-Markov chain model produced by Chitra is that gross measures of program behavior can identify the presence of performance problems, although they do not explain why the behavior is so. Section 5 illustrates the use of the chain for performance problem diagnosis. The performance problems could be diagnosed without Chitra, but our contention is that Chitra provides the diagnosis more rapidly.

The most important future work is to apply the tool to additional commercial software. Work is in progress on analyzing multiple hosts running TCP/IP, parallel discrete event simulation programs, and a commercial transaction processing system.

This paper describes Chitra91. Lessons from the case studies in Section 5 are being incorporated into Chitra92. New features include the following:

- Ability to simultaneously analyze a set of PES's forming an ensemble; for example a transform will be applied to all ensemble elements. 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.

- Ability to automatically merge a set of empirical models representing program behavior at different parameter values by fitting functions to formulate empirical formulas for occupancy times and transition probabilities.

- Ability to display different views of a PES in different screens, and to extend the set of transforms to operate on components of the state vector rather than on entire states. This

will help with the state space explosion problem.

- Automatic construction of performance metric graphs (e.g., Figure 15) based on an ensemble of PES's.
- Addition of standard time series analysis techniques.
- Application of Chitra to PES's representing memory reference traces, to permit both data and code oriented analysis, as defined in [21].

Two open problems are to provide means of analyzing PES's collected without a global time base, and to investigate use of a non-homogeneous process as Chitra's empirical model.

Acknowledgements

Qizhong Chen provided the TCP/IP PES analyzed in Section 5.1. Ashok K. Agrawala suggested the use of frequency domain analysis. The Sequent Symmetry at the Argonne National Laboratory was used for some experiments. Krishna Ganugupati, C. R. Chandrasekar, Jay Wang, and Alan Batongbacal contributed several suggestions for the manuscript. The referees made many suggestions improving the manuscript readability.

References

[1] M. Abrams and A. K. Agrawala, "Performance Study of Distributed Resource Sharing Algorithms," *IEEE Dist. Processing Technical Committee Newsletter*, Vol 7, No. 3, Nov. 1985, 18-26.

[2] M. Abrams, *Design of a Measurement Instrument for Distributed Systems,* Research Report RZ1639, IBM Research Division, Zurich Research Laboratory, 1987.

[3] 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.

[4] M. Abrams and N. Doraswamy, *An Introduction to Visualizing Program Execution Dynamics with Chitra,* Department of Computer Science, Virginia Tech, TR 92-23, May 1992.

[5] M. Abrams and Q. Chen, *A New View on What Limits TCP/IP Throughput in Local Area Networks,* Department of Computer Science, Virginia Tech, TR 91-23, 1991, submitted for publication.

[6] G. Balbo, G. Chiola, S. C. Bruell, and P. Chen, "An Example of Modeling and Evaluation of a Concurrent Program Using Colored Stochastic Petri Nets: Lamport's Fast Mutual Exclusion Algorithm," *IEEE Trans. on Parallel and Distributed Systems 3*, March 1992, 221-240.

[7] 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.

[8] U. N. Bhat, *Elements of Applied Stochastic Processes*, 2nd. ed, New York: John Wiley, 1984, pp. 290-294.

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

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

[11] 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.

[12] N. Doraswamy, *Chitra: A Visualization System to Analyze the Dynamics of Parallel Programs*, M.S. thesis, Dept. of Computer Science, Virginia Tech, Dec. 1991.

[13] A. Duda, G. Harrus, Y. Haddad, G. Bernard, "Estimating Global Time in Distributed Systems," *Proc. 7th Int. Conf. on Distributed Computing Systems*, Berlin, Sept. 1987, 299-306.

[14] 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.

[15] M. A. Holliday and M. K. Vernon, "A Generalized Timed Petri Net Model for Performance Analysis," in *Proc. Int. Workshop on Timed Petri Nets*, July 1985.

[16] 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.

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

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

[19] Allen D. Malony, *Performance Observability*, Ph.D. thesis, Computer Science Dept., Univ. of Illinois, Aug. 1990.

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

[21] Margaret Martonosi, Anoop Gupta, and Thomas Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs," *Proc. ACM SIGMETRICS*, Newport, RI, June 1992, 1-12.

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

[23] 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.

[24] B. Plateau, "On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms," *Proc. SIGMETRICS*, Austin, Aug. 1985, 147-154.

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

[26] 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.

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