Toward a Machine Assisted Software Performance Diagnosis Methodology

Anup Mathur and Marc Abrams

TR 93-12

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

April 12, 1993

Toward a Machine Assisted Software Performance Diagnosis Methodology

Anup Mathur and Marc Abrams

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106
{mathur,abrams}@vtopus.cs.vt.edu

April 7, 1993

Abstract

This paper discusses a methodology for diagnosing performance problems for parallel and distributed programs. The methodology is based on the formulation and testing of hypotheses about the cause of performance bottlenecks. The process is illustrated with a case study of an actual problem arising in a parallel discrete event simulation program in which granularity is a primary bottleneck and barrier implementation is a secondary bottleneck. The paper also describes the evolution of Chitra, a software performance measurement and analysis tool whose objective is to automate certain steps in software performance diagnosis.

1 Introduction

Tools that reduce the human time required to diagnose performance problems in parallel and distributed systems are a key to the widespread use of massively parallel systems. Since massively parallel machines increasingly rely on features such as complex memory hierarchies and interconnection networks, the task of tuning the performance of software running on such platforms is also complex. The reasons for performance anomalies can be many. Large scale reasons include partitioning, process to processor mapping, and granularity [21]. Small scale reasons include data structures that causes cache misses and misuse of synchronization primitives. In literature reasons for performance anomalies are called *performance bottlenecks*. We shall use this name to describe them in the rest of this paper.

While several performance tools have become available [7, 9, 11, 13, 17], the general approach to the task of diagnosing performance problems in parallel and distributed programs still remains *ad hoc*. We claim that this is due in part to the fact that no methodology exists that can be used in association with tools to measure and analyze performance. In this paper we introduce a methodology for performance diagnosis (Section 2) in association with a performance measurement and analysis tool called Chitra (see Section 3) [1]. [1]

The task of performance measurement requires data to be collected from instrumented source code. The data is then used by a performance measurement or analysis tool. Today's performance tools, such as execution profilers [13] or performance visualization systems [11, 8] only illustrate what is happening in *particular* program runs. Chitra, in contrast, generates a parameterized empirical model fitting all observed data. A model potentially can explain what happened in all observed runs, as well as predict what happens when the program is executed with different numbers of processes, problem sizes, and computer hardware than that used in the observed runs.

Visualization systems today can represent a limited number of processors on a monitor screen. One trend in visualization research today is just to increase the data bandwidth passed to a

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

human user, for example using sound or virtual reality [11, 12]. However human users have only five senses, which limits their capacity to absorb and interpret data. In contrast parallel computers continue to grow in the number of processors, and hence the volume of trace data available for performance analysis. Inundating a human with massive amounts of data conveyed through sight and sound puts the onus on the human to mentally build complex models to understand such data. Without the ability to reengineer a human, we must turn to what scientists have done for centuries: constructing models, because models reduce the amount of information that a human must interpret.

Chitra takes as input a collection of Program Execution Sequences (PES's) derived from trace files produced by running instrumented source code. A PES records the state of a system as a function of time. Examples of "state" are the memory addresses referenced, the control point of processes, the values of data structures, or a combination of these. Current literature includes several examples of memory-oriented performance tuning [7, 17], code-oriented performance tuning [13], and performance tuning of interconnection networks [18]. Chitra subsumes the function of such tools because of its ability to analyze an arbitrary state representation. Chitra contains a module that generates an empirical model from a set of PES's. However, building a model of the raw PES's typically results in a model with a large, complex state space. Therefore Chitra permits the reduction of the state space size by providing a set of transforms. These transforms map a PES into a simpler PES, leading to a simpler model with a smaller state space size. Each transform has a visual analog, and is selected by a human through the visualization component of Chitra. The transforms in Chitra work with *any* type of software. In fact, Chitra analyzes categorical time series data, and therefore could be used with non-software applications.

Application of Chitra

The evolution of Chitra has proceeded hand-in-hand with its application to actual software problems. Published case studies using Chitra include: the dining philosophers problem [1], a commercial implementation of the TCP/IP protocol for the MS-DOS [1], and a parallel discrete event simulation algorithm called bounded lag [4] (this analysis is presented in Section 2.3 as a

case study.)

Other applications being analyzed include: numerical Algorithms (e.g Gaussian elimination, Matrix multiplication), MPEG player software that retrieves compressed video files from disk [20], the Network Multimedia File System (NMFS) communication protocol [19], and a CICS-based transaction system on a IBM-3090.

The rest of this paper is organized as follows: Section 2 discusses the development of a methodology for performance diagnosis. Section 3 discusses the evolution of the performance measurement and analysis tool Chitra. Finally Section 4 discusses directions for future research.

2 A Methodology for Performance diagnosis

2.1 Motivation

Software performance debugging and tuning is not a first class process in the software life cycle but it still is the most daunting. This is even more so in the case of parallel and distributed software. Whereas several tools have been developed to measure and model parallel and distributed program behavior, there has been little effort directed toward understanding the process of performance debugging. Lehr *et al* [8] describe performance debugging as an iterative task that alternates between measuring and modifying the performance of successive computation prototypes. Several tools in literature [8, 9] have recognized performance debugging as a task. ChaosMON [9], for instance, builds an abstract high level model of an application program that helps the user to determine what to monitor. While PIE, and ChaosMON [8, 9] are steps in the right direction, we claim that a comprehensive methodology is required to address the various steps and issues involved in the task of performance diagnosis. We report here on initial steps to develop such a methodology.

2.2 Performance Diagnosis Methodology

The methodology is based on the construction and testing of hypotheses that explain the reason for performance anomalies. These hypotheses are constructed by the user. Chitra contributes to this process by being the measurement and analytical tool that provides estimates of metrics

chosen by the user. Chitra also provides the user with an empirical model of program behavior.

The following definitions facilitate the discussion of the methodology: First we define a *performance anomaly* as any aspect of program performance which the user finds unsatisfactory. A *performance bottleneck* or simply a *bottleneck* is a cause of a performance anomaly in a program. There may be several bottlenecks that contribute to a performance anomaly. The bottleneck that contributes most significantly to the performance anomaly is known as the *primary performance bottleneck*. Typically once the primary performance bottleneck has been identified and removed, other bottlenecks called *secondary bottlenecks* can be targeted. Hypotheses are defined in terms of bottlenecks. A *strong* hypothesis defines a code segment or memory data structure X as a primary performance bottleneck and asserts that the replacement of X with the code segment or memory data structure Y will improve the performance of the application. A *weak* hypothesis on the other hand is usually not as concrete. It is used to confirm the symptoms of the performance problem. Testing a weak hypothesis may lead to the formulation of a strong hypothesis.

Initially the user is expected to formulate a hypothesis based on the best guess or information available to him or her. This is usually a weak hypothesis. Let H_0 denote this hypothesis. The next step is to select a metric that can be used to test the hypothesis. Let M_0 denote the measure for hypothesis H_0. Instrumentation I_0 is then introduced into the program under analysis (P). P is run with instrumentation I_0 to obtain a trace T_0; in our case the trace is a PES of the program. T_0 is analyzed by a performance tool, Chitra in our case, and an estimate of M_0, denoted $E(M_0)$ is obtained. If H_0 is true, $E(M_0)$ may suggest a bottleneck B. This leads to the formulation of a strong hypothesis H_1. The user then alters the code for P to remove the bottleneck B; let $P^{'}$ denote the modified program. Code $P^{'}$ is then run with instrumentation I_0, and trace $T_0^{'}$ is collected. $T_0^{'}$ is subsequently analyzed by the performance tool to yield the estimate of M_0, $E^{'}(M_0)$. Comparison of $E^{'}(M_0)$ with $E(M_0)$ may lead the user to accept or reject H_1. The performance analysis could possible end at this step. It is usually the case, however, that this process leads to the formulation of more sophisticated hypotheses and the entire cycle is repeated a number of times. An iteratively refined, strong hypothesis, denoted

H_f, finally leads to the discovery of the primary performance bottleneck B_f.

Typically, during the transition from H_0 to H_f, several intermediate hypotheses, H_1 through H_{f-1} are formulated, which can be re-examined after the primary bottleneck B_f has been identified and removed; the program can then be further tuned by identifying and removing secondary bottlenecks.

An important issue pertinent to our methodology is the perturbation in program behavior generated by the insertion of instrumentation to collect performance data. Due to their non-deterministic behavior this problem is of even greater significance for parallel and distributed programs. Maloney [10] likens this situation to the Hiesenberg's uncertainity principle in Physics. Most tools in the literature take one of the following two routes:

1. Instrument the program being analyzed and then alter the performance data collected to compensate for the change in program behavior. To do this we must be able to predict how the instrumentation perturbed the behavior of the program. Maloney *et al* [10] discuss the construction of models for performance perturbation analysis to support this.

2. Instrument the program so that its normal behavior is not significantly altered. To do this we must designate what normal behavior is and what constitutes a significant deviation from it.

We choose route 2 for our methodology. We propose using the frequency domain view or periodogram (see Section 3) displayed by Chitra as a footprint of program behavior. The stepwise strategy is outlined below:

1. Some basic non-intrusive instrumentation (henceforth referred to as *base-instrumentation*) is introduced into the program. What constitutes base-instrumentation may differ from program to program. For a typical multithreaded homogeneous parallel application the base-instrumentation could be a trace statement that executes once for every iteration of the main loop of the code being executed by each thread.

2. The trace collected from the base-instrumentation is input to Chitra and a periodogram

of the data is recorded.

3. Next the candidate instrumentation is added to the program (this is in addition to the base-instrumentation).

4. Again the trace collected from the base-instrumentation is input to Chitra and periodogram of the data obtained is compared to the previously recorded periodogram.

5. If the periodograms differ significantly the candidate instrumentation disturbed the program behavior too much. The program should therefore be re-instrumented and steps 2,3, and 4 repeated. If the two periodograms match closely the instrumentation is acceptable.

2.3 Case Study

2.3.1 Problem Definition

The implementation of a parallel simulation algorithm for asynchronous multiple loop networks using Lubachevsky's bounded lag discrete event simulation protocol [4] is the basis of this case study.

As a part of an experimental study [2], the above algorithm was used to simulate a $n \times n$ toroidal network (n being an integer). Each node in the network simulates a server with an infinite capacity input queue. A server is modeled by a Logical Process (LP). All LP's asynchronously execute the simulation algorithm in parallel. The experiment was done using the Sequent Symmetry S81 shared-memory multiprocessor, running Dynix V3.0.18 and the Presto 0.4 thread package [3]. Each Presto [3] thread executed the code for an LP. The parameters for this experiment are the number of threads, denoted P, and the number of nodes in the torus network (n^2) denoted by N.

The benchmarking results from this experiment showed that the running time for the algorithm improved when the number of processors used increased from 2 to 4, but there was no more improvement if the number of processors was increased above 4. This was noted by running the algorithm with 8 processors. It was conjectured that that the reason for this behavior was

that the number of events being processed between barriers 1 and 2 in the algorithm was not dense enough compared to the time the program spent doing barrier synchronization.

2.3.2 Hypothesis 0

The initial hypothesis is a weak hypothesis. It is based on the the conjecture [2] that the duration of the barrier is much longer than the time spent by the program to process events.

Metric Chosen We chose the total time spent by a thread while blocked at the barriers as the metric.

State(s) Chosen The states chosen are *thread blocked at barrier1, thread blocked at barrier2* and *thread blocked at barrier3.*

Instrumentation A local array declared on the stack of each thread records timestamped local states, such as *thread blocked at barrier1, thread blocked at barrier2, thread blocked at barrier3, thread scheduling events,* etc. We used the global microsecond clock on the sequent to get the timestamps. A chosen thread writes the contents of its local array to a disk file before termination. To minimize perturbation to the program behavior due to instrumentation, only one thread collects the trace data. This keeps the calls to the sequent's clock to a minimum. Also collecting the trace data in an array local to the thread avoids reference to shared global data.

Conclusions The output from Chitra, Figure 1, shows that the thread spends most of the time blocked at the barriers (states $br1, br2$, and $br3$). Our hypothesis is therefore confirmed.

2.3.3 Hypothesis 1

The barrier code being used in our case study is an example provided with the Presto thread package [3]. The barrier is implemented by a Master-Slave mechanism based on the use of monitors. Presto offers a C++ class called Monitor that implements a monitor object. Our

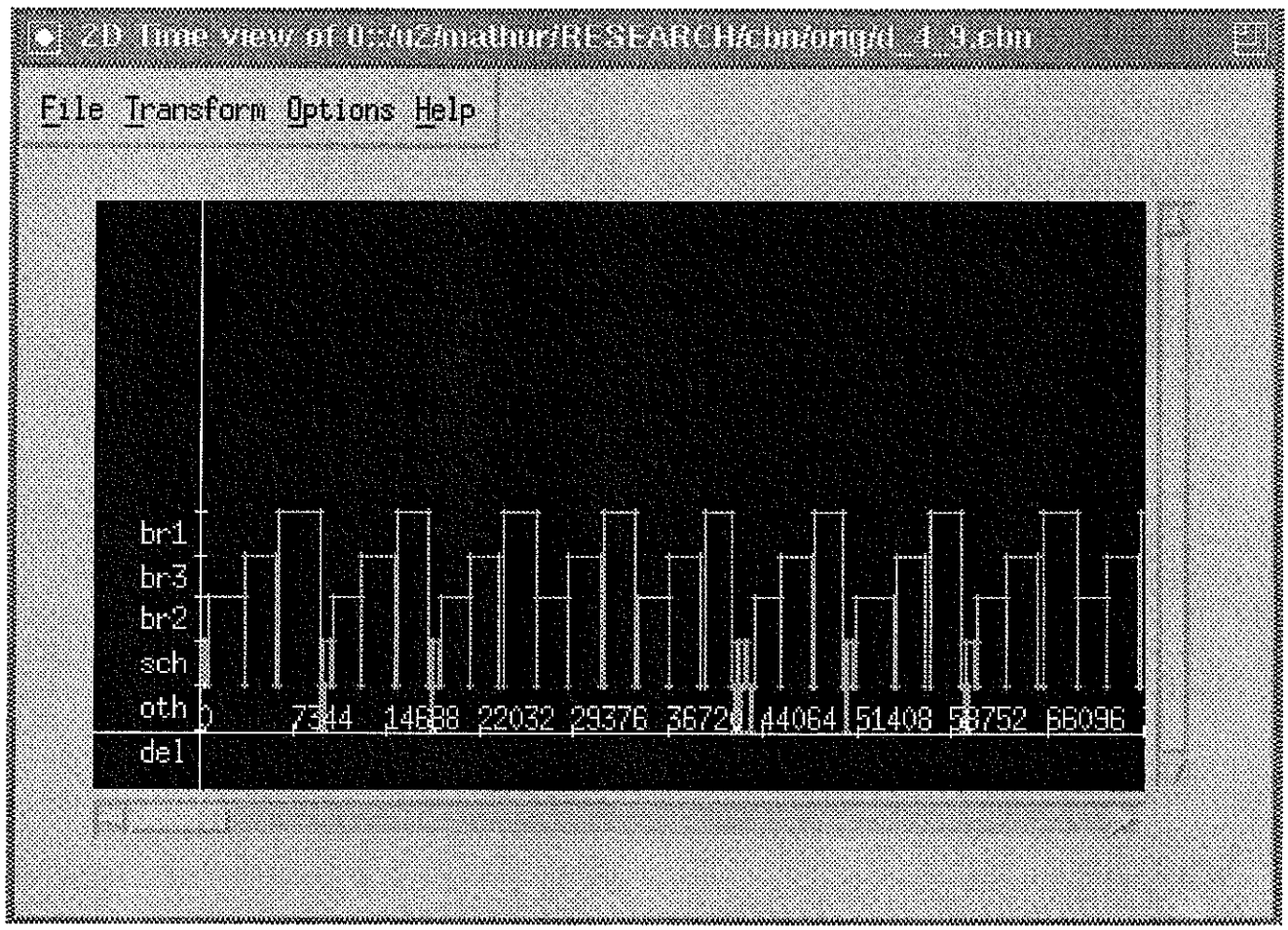


Figure 1: Time Domain View of a Single Thread in the original code.

next hypothesis is of the strong type. It asserts that the threads spend most of the time blocked because the code implementing the barrier is inefficient. In particular the monitor code is hypothesized to have two problems:

- the critical section of the code contains some statements which do not need to be serialized and,
- dynamic allocation of Monitor object(s) is contributing to the bottleneck.

Metric chosen Remains the same as the metric chosen for hypothesis 0.

State(s) Chosen Remains the same as the states chosen for hypothesis 0.

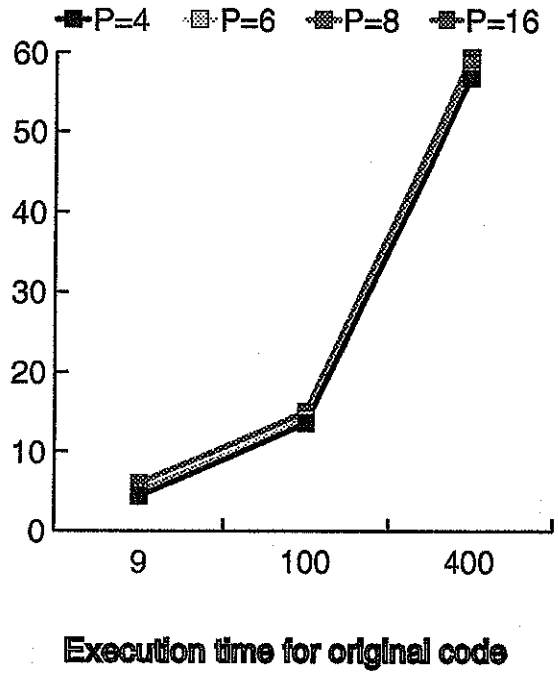
Code Alteration The critical section in the code implementing the barrier is reduced by moving appropriate statements outside it. Also calls to the constructors and destructors to the monitor object are removed; instead monitor objects are created by static declaration.

Instrumentation Remains the same as the instrumentation for hypothesis 0.

Conclusions Running the simulation before and after the necessary modifications to the code inside the Monitor were made shows no appreciable change in program behavior. Thus hypothesis 1 is rejected.

2.3.4 Hypothesis 2

Hypothesis 2, is also a strong hypothesis. It hypothesizes that the the amount of work done by each thread is not significant enough to offset the synchronization overhead. Thusly the program needs to be partitioned to increase the granularity of the computation. Specifically, if the number of threads is made equal to the number of processors, so that each thread simulates several nodes of the torus, the duration when a thread is processing events will increase compared to the time it spends blocked at barriers.



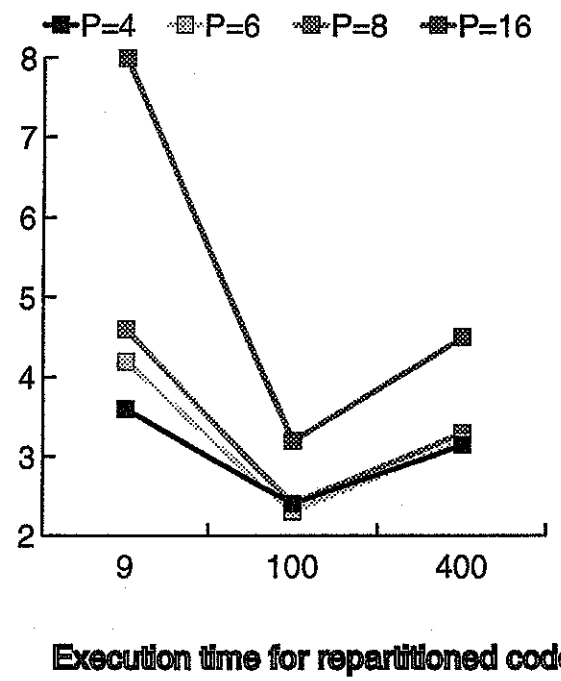


Figure 2: Execution time of the Original vs Repartitioned code, for $P = 4, 6, 8, 16$ and $N = 9, 100, 400$

Metric chosen Same as the metric chosen for hypothesis 0.

State(s) Chosen The states chosen are *thread blocked at barrier1, thread blocked at barrier2, thread blocked at barrier3, thread executing events for the nodes mapped to it,* and *thread executing other code.*

Code Alteration The simulation code is altered so that multiple nodes of the toroidal network map to a single Presto [3] thread. Specifically:

- the number of sequent processors requested to execute the simulation is made equal to the number of Presto threads requested.
- each Presto thread is made to execute the code for several nodes of the toroidal network being simulated.

Instrumentation Similar to the instrumentation for hypothesis 0 and hypothesis 1 a chosen thread is instrumented to collect the trace in a local array.

Conclusions The performance of the simulation shows significant improvement after the repartitioning of the program code as directed by hypothesis 2. This is illustrated by the graphs in Figure 2 that show the execution times of the simulation for different values of P and N. We therefore accept hypothesis 2. Figure 3 shows a parametrized model constructed from the semi-Markov model output from Chitra. The parametrized model depicts how the occupancy times of the states chosen for the testing of hypothesis 2 vary when P varies from 4 to 16 and $N = 400$. The parametrized model reveals that the occupancy time for states $br1$ (thread blocked at barrier1) and $br2$ (thread blocked at barrier2), increases while the occupancy time of the state *env* (thread executing events for the nodes mapped to it) decreases as P is varied from 4 to 16. This behavior indicates the presence of a secondary bottleneck.

2.3.5 Hypothesis 3

We now formulate a weak hypothesis to inspect candidates for the secondary bottleneck. Two candidates bottlenecks are tested:

1. the barrier code itself, and
2. the implicit synchronization due to the locking and unlocking of shared data structures by threads while processing events for the nodes of the torus network.

Metric chosen To compare the two candidate bottlenecks two metrics are chosen, and they are: the time taken by a thread to execute the barrier code, and the time taken by a thread to access data structures shared by all the nodes in the torus.

State(s) Chosen The states chosen are *thread blocked at barrier1, thread blocked at barrier2, thread blocked at barrier3,* and *thread scheduling events for torus nodes.*

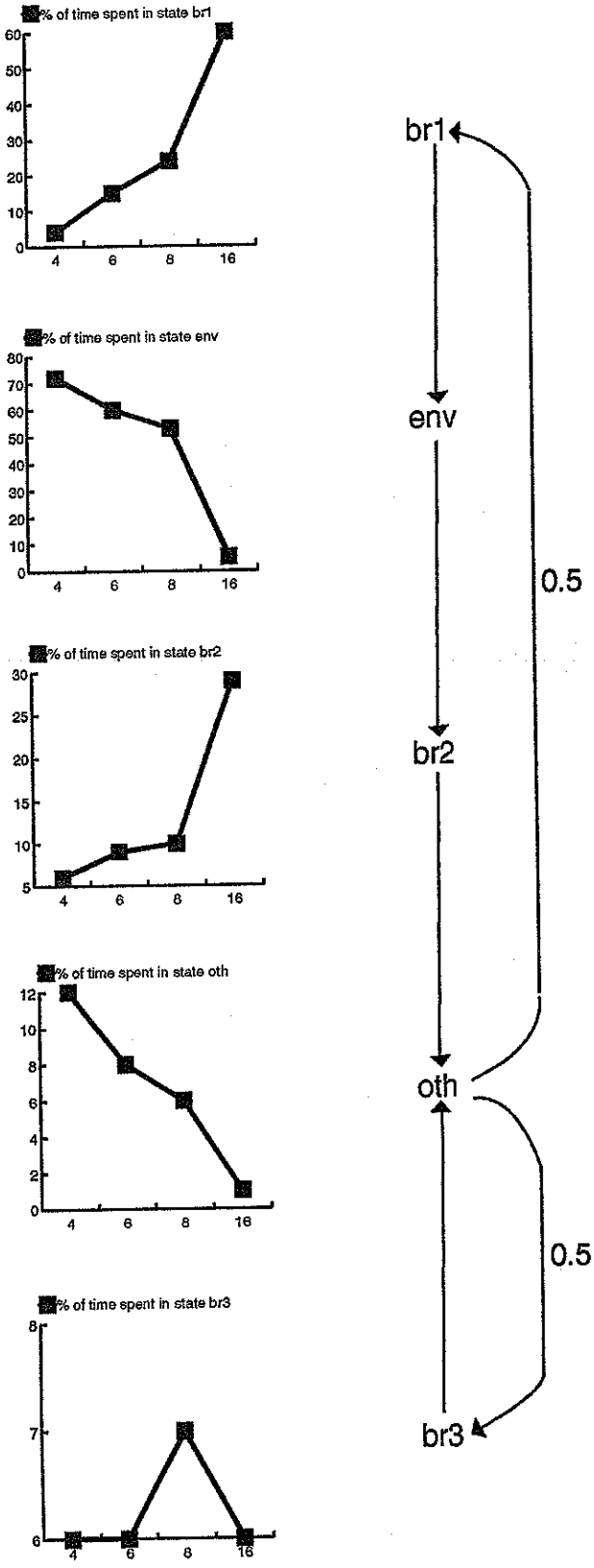


Figure 3: Parametrized model of the Repartitioned code for $P = 4, 6, 8, 16$ and $N = 400$

Instrumentation A copy of the program is instrumented to measure the time spent by the thread blocked at the barriers. The simulation code is disabled to measure the execution time of the barrier code accurately. Another copy of the program is instrumented to measure the time a thread spends scheduling events for nodes in the torus. We measure the scheduling time because this is the part of the program source code where the shared global data structure is accessed.

Conclusions Hypothesis 3 supports candidate bottleneck 1. The parametrized model from Figure 3, and hypothesis 3 suggest the barrier code to be the secondary bottleneck. The logical course of action is to revisit and reformulate a refined version of hypothesis 1. The reformulated version of this strong hypothesis must examine the following assertion: The barrier code which implements an algorithm with complexity $\Omega(N)$ must be replaced by a sub-linear algorithm. In particular the $O(logN)$ barrier algorithm given in [5] should be employed.
The conclusion from this performance diagnosis case study is that the primary bottleneck for the performance problem is the granularity of parallel computation, and the secondary bottleneck is the implementation of the barrier code.

3 Chitra

This section overviews the three successive generations of the Chitra performance analysis system. Our first two generation tools, CHITRA91 and CHITRA92, use a homogeneous semi-Markov process model to describe program behavior. The analysis of CHITRA91 and CHITRA92 assumes that the program under analysis reaches steady state. Planned for the third generation tool CHITRA93 are alternative models of program behavior described in Section 3.3.

3.1 Chitra91

CHITRA91 is a prototype and is described in [1]. The input to CHITRA91 is an textual trace file that contains the PES [15]. The trace, once loaded, can then be displayed by the CHITRA91 for visual inspection and *visual editing* by the user. Visual editing is the user directed transformation of the PES. Visual editing reduces the size of the state space and helps the user build an accurate

model of the PES. CHITRA91 supports four *transforms* for visual editing of the PES, defined in [1]. *Clipping* is useful in eliminating the initial and final transient portions of the PES. *Aggregation* is useful in capturing deterministic patterns in a PES. CHITRA91 aggregates the states that form a deterministic pattern into one aggregate state. A semi-Markov model thusly formed describes the behavior of the program more accurately. *Projection* is a transform that uses the semantics of states and is specified textually rather than visually. It is also reduces the state space size. Finally, *filtering* is useful in lumping states that occur infrequently into one state, thereby reducing the state space size. The final output from CHITRA91 is a homogeneous semi-Markov model that describes program behavior shown by the PES.

A unique aspect of CHITRA91 is its view of the PES as a *signal* as used in electrical engineering. Operations such as filtering which are often applied to signals are supported by CHITRA91. CHITRA91 provides three *views* of the PES represented as a signal: event domain, time domain, and frequency domain. These views permit the human user to look at the PES as time-series data.

3.2 Chitra92

Application of CHITRA91 to the case-studies in Section 1 prompted the evolution of CHITRA92. Unlike CHITRA91, which allowed only one PES to be viewed at a time, CHITRA92 allows it viewing of multiple PES's at a time. It also allows multiple views of the same PES at a time. CHITRA92 serves as the public release version of our tool. The following features make CHITRA92 [16] a suitable vehicle for many future years of development:

- Modularity: The architecture of CHITRA92 facilitates the extension of Chitra to include alternate modeling techniques, transforms, and views. Several such models have been identified and are listed in Section 3.3.

- Portability: To make the system portable, implementation was done using the widely available OSF/Motif 1.1 toolkit, the MIT Athena toolkit, and the X11 window system.

- Versatility: CHITRA92 can analyze traces from any domain of application as long as the trace input can be converted to the Chitra Specification Language (CSL) format.

3.3 Chitra93

The third generation tool CHITRA93 is under development. CHITRA93 analyzes an ensemble of PES's from the same program simultaneously, thusly the model of program behavior generated by CHITRA93 is more accurate than similar models generated by CHITRA91 and CHITRA92. CHITRA93 also has the ability to build parametrized models of program behavior. Here parameter refers to any or all of the inputs to the program. The approach is rooted in the idea that by actually analyzing the behavior of the program for a set of program parameters we should be able to predict the behavior of the program for a range of program parameter values. For instance, suppose we are interested in analyzing the behavior of a program that has N threads. We could visualize and transform PES's for extreme values of N, say $N = 9$, and $N = 400$, but build a model that would predict the behavior of the program where N ranges from 9 to 400. Figure 3 is an example of a parametrized model where the parameter P varies from 4 to 16. The major goal of CHITRA93 is to add alternate models to describe program behavior. The following new models are being explored:

- Construction of a model of program behavior based on decision tree analysis [22].
- Most stochastic analysis of program behavior assumes that program execution reaches a steady state. CHITRA91 and CHITRA92 make this assumption. This simplifying assumption is usually violated, since most programs are transient and never reach steady state. We are considering a class of models called *quasi-stationary* [6] models to describe such behavior.
- A rule based scheme is being devised to replace deterministic subsequences in a PES by an aggregate state in the PES.
- New views and metrics are being explored to measure the perturbation in program behavior due to instrumentation. By supporting such features CHITRA93 will contribute towards

the automatization of the performance diagnosis methodology described in Section 2.

4 Conclusions

Chitra described in [1] constructs an empirical model of program behavior. We now have the second generation tool CHITRA92 which serves as the production quality version (for instructions to obtain CHITRA92 by ftp send e-mail to chitra@vtopus.cs.vt.edu). The third generation tool CHITRA93 is under development. Chitra, along with the methodology described in this paper is expected to provide a rapid and systematic approach to performance diagnosis. Amongst the foremost of our future goals is to apply the methodology, and Chitra, to parallel and distributed software in several application domains. Section 1 lists applications which have been or are being analyzed. Some other directions which have been targeted for future exploration and research are:

- Enhance Chitra by providing automatic instrumentation of source code as in Pablo [11] and Mtool [17]. A related goal is to develop and provide machine assisted instrumentation reuse techniques. The methodology described in Section 2 is based on the formulation and testing of hypotheses $H_0, H_1, ..., H_f$. After the primary bottleneck has been found, the user may want to revisit some of the hypotheses H_0 through H_{f-1}, tested previously to find a secondary bottleneck. As mentioned in Section 2 each hypothesis H_i, where $i \in \{0, ..., f\}$, is associated with instrumentation I_i. Suppose the user wants to retry a hypothesis H_i which has been tried previously then Chitra could use the known instrumentation I_i to automatically reinstrument the code. This may be easy to do if the code of the program P, under analysis, has changed minimally during the testing of hypotheses H_0 through H_f. To provide similar support if the source code for P has changed significantly is an open problem.

- Explore metrics and displays to measure the perturbation in program behavior that arises due to the addition of instrumentation.

References

[1] Abrams, M., Doraswamy, N., and Mathur, A., "Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event, and Frequency Domains," *IEEE Trans. Parallel Distributed Syst.*, vol. 3, no. 6,672-685, November 1992.

[2] Sanjeevan, V., "The Cost of Terminating Parallel Discrete-Event Simulations," *M.S. thesis,* Dept. of Computer Science, Virginia Tech, June 1992.

[3] Bershad, B.N., Lazowska, E.D., and Levy, H.M, "PRESTO: A system for Object-Oriented Parallel Programming," *Technical Report 87-09-01.*, Department of Computer Science, University of Washington, Seattle, Washington, Jan. 1988.

[4] Lubachevsky, B., "Efficient distributed event-driven simulations of multiple loop networks," *Comm. ACM.*, vol. 32, no. 1,111-123, Jan. 1989.

[5] Lubachevsky, B., "Synchronization Barrier and Related Tools for Shared Memory Parallel Programming," *International Journal of Parallel Programming,* vol. 19, no. 3,226-250, July. 1990.

[6] Darroch, J.N., and Seneta, E., "On Quasi-Stationary distributions in absorbing discrete-time finite Markov chains," *J. Appl. Prob.*, 2, 88-100, 1965.

[7] Martonosi, M., Gupta, A., and Anderson, T., "MemSpy: Analyzing Memory System Bottlenecks in Programs," *Performance Evaluation Review,* vol. 20, no. 1, 1-12, 1992.

[8] Lehr, T., Segall, Z., Vrsalovic, D.F., Caplan, E., Chung, A.L., and Fineman, C.E,"Visualizing performance debugging," *IEEE Computer,* vol. 22, no. 10, 38-52, October 1989.

[9] Kilpatrick, C., and Schwan, K, "ChaosMON–Application-Specific Monitoring and Display of Performance Information for Parallel and Distributed Systems," *Performance Evaluation Review,* vol. 26, no. 12, 57-67, December 1991.

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

[11] Reed, D.A., Aydt, R.A., Madhyastha, T.M., Noe, R.J., Shields, K.A., and Schwartz, B.W., "The Pablo Performance Analysis Environment," *Technical Report,* Department of Computer Science, University of Illinois, Urbana, Illinois, Spring 1993.

[12] Francioni, J.M., Albright, L., and Jackson, J.A, "Debugging Parallel Programs Using Sound," *Performance Evaluation Review,* vol. 26, no. 12, 68-75, December 1991.

[13] Graham, S.L., Kessler, P.B., and McKusick, M.K., "An Execution Profiler for Modular Programs," *Software Practice and Experience,* vol. 13, 671-685, August 1983.

[14] Landry, K.D., Cline, G., and Arthur, J.D., "A Distributed Parallel Processing Environment Based upon the Linda Paradigm: A Research Prospectus," *TR-Number TR-92-18,* Dept. of Computer Science, Virginia Tech, Blacksburg, Virginia,1992.

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

[16] Ganugapati, K., "The Design and Implementation of Chitra92, a System to Empirically Model Concurrent Software Performance," *M.S. thesis,* Dept. of Computer Science, Virginia Tech, March 1993.

[17] Goldberg, A.J., and Hennessey, J.L., "Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications," *IEEE Trans. Parallel Distributed Syst.*, vol. 4, no. 1,28-40, January 1993.

[18] Ahluwalia, A.K., and Singhal, M., "Performance Analysis of the Communication Architecture of the Connection Machine," *IEEE Trans. Parallel Distributed Syst.,* vol. 3, no. 6,728-738, November 1992.

[19] Patel, S., Abdulla, G., Abrams, M., and Fox, E., "NMFS: Network Multimedia File System Protocol," *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video.,* 328-334, November 1992 San Diego, California

[20] Rowe, L.A., and Smith, B.C., "A Continuous Media Player," *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video.,* 334-344, November 1992.

[21] Chandy, K.M., and Taylor, S., "AN INTRODUCTION TO PARALLEL PROGRAMMING," *Jones and Bartlett Publishers.,* Boston, 1992.

[22] Biggs, D., De Ville, B., and Suen, E., "A method for choosing multiway partitions for classification and decision trees," *Journal of Applied Statistics.,* vol. 18, no. 1, 49-62, 1991.