Fast forwarding Distributed Programs using Computation Slicing

Abhaya Agarwal, Jitender Sharan and Deepak Gupta
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur, India
{abhayag,jsharan,deepak}@iitk.ac.in

Vijay K. garg
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
garg@ece.utexas.edu

20th July 2004

Abstract

Debugging a distributed program using the traditional debugging cycle is very difficult. First, the nondeterminism inherent in distributed programs makes it hard to reproduce any problems observed in earlier runs. Secondly, it is difficult to efficiently implement data breakpoints where the breaking condition may involve variables from multiple processes. The first problem can be tackled by using a record-replay facility that can record one run of distributed program and can replay it later any number of times. However, most current debuggers for distributed programs implement data breakpoints using single stepping which is extremely inefficient. In this paper, we present an execution replay system for distributed programs with efficient support for data breakpoints. Breakpoint detection is performed off-line on the recorded memory state of program using computation slicing. This reduces the overhead of such breakpoints to the level that they can be extensively used while debugging a distributed program.

1 Introduction

Writing distributed programs is an error prone activity since for writing really effective parallel programs, one has to take in account the problems of data distribution among processes and also computation-communication overlap. Debugging such programs is even tougher since there can be bugs that may surface only for some particular combinations of inputs and ordering of events. Moreover due to the non-deterministic nature of distributed program executions, these bugs cannot be reproduced by simply re-executing the program. A record-replay facility is needed for the purpose.

Breakpoints are commonly used debugging tools that can considerably simplify the debugging process. Breakpoints can be of different types depending on the breaking condition. In

control breakpoints, the break condition is specified in terms of program control abstractions while in data breakpoints, the break condition is specified in terms of memory state of the program. For example, "stop at line 11" is a control breakpoint while "stop when value of `x` becomes greater than 12" is a data breakpoint. In case of distributed programs, breakpoints can either be local or global. In a local breakpoint, the break condition is defined in terms of the memory state of one process whereas in global breakpoints, it is defined in terms of the memory states of multiple processes.

Data breakpoints are very powerful since they let the programmer reason in terms of conditions on data and thus allow him to focus on the logic of the program rather then its control flow. However in distributed programs, there are two problems with data breakpoints. First, the break condition can involve variables from multiple processes. This complicates the problem of detecting the break condition efficiently. But even after are able to detect the break condition, we need a way to stop all the processes. However the delay between the actual occurence of break condition and its detection cannot be avoided. So any online method needs to either resort to the technique of single stepping for breakpoint condition detection or use checkpointing and rollback. Many parallel debuggers use single stepping but it makes the program execution extremely slow and so data break points are rarely used. On the other hand checkpointing and rollback require a lot of book keeping which consumes both disk space and CPU cycles.

In this paper, we describe a debugger for Java MPI programs that efficiently supports both record-replay and data checkpoints. During the record phase, all communication events, and changes to variables of interest are logged. Condition detection is then done (offline) on the recorded run of the program using computation slicing [19]. During replay, the user can request the program execution to be stopped at the next state where the checkpoint condition is satisfied. This is efficiently implemented by stopping the program at the next cut in the computation slice. Our approach of offline breakpoint condition detection using computation slicing does not require any extra work to rollback the computation to the right global state since every process already knows the state where it has to halt. Nor is single stepping required to stop the processes at the right points; program execution can proceed at full speed.

The rest of the paper is organized as follows. In section 2, we discuss some of the related work in the area. The background about distributed systems and computation slicing is given in section 3. Section 4 provides the bird's eye view of the system architecture. Section 5 and 6 present the implementation details and some performance measurements respectively. Finally in section 7, we give summary and discuss possible future work.

2 Related Work

A lot of effort has been put into replay/re-execution of distributed programs by the designers of parallel debugging systems, since replay is necessary for applying the classical cyclic debugging methodology. Replay can be implemented more easily and cleanly for message passing based distributed programs since they do not have any shared memory. All the communication between processes is usually through some underlying library or daemon and hence can be easily logged. So the main focus is on minimizing the information logged and reducing the overhead.

There are two basic approaches for replaying distributed program. The first approach requires the logging of only the racing messages in order to replay the original computation [1]. All the messages in a computation that can get accepted by a certain receive call in some process are said to be racing for that receive. A vector clock is used to discover the messages that are racing. Adaptation of this technique for MPI programs is described in references [2, 4].

The second approach takes advantage of the fact that MPI channels are FIFO. So we only

have to take care of wild card receive operations which do not specify the origin of the message and would accept a message coming from any source. In MPI, this can be achieved by specifying ANY_SOURCE as the source. Similarly there can be receives with ANY_TAG wild card also. To handle wild card receives, the actual sender of the message accepted by every such receive is logged at the record time and at the replay time, a wild card receive is replaced by a point to point receive operation from that very sender. Non-blocking test functions are handled by logging the number of unsuccessful calls to them. References [3, 5, 6] are examples of this strategy. Our strategy for replay is based on reference [6].

The problem of detecting distributed data breakpoints has been extensively studied. The two main problems that one has to deal with here are the detection of global predicates, and halting the computation at the global state satisfying the predicate. Most existing techniques use on-line detection of the breakpoint condition. Miller and Choi [13] used an similar to the distributed snapshot algorithm [11] where one process decides to halt to computation and propagates a halting instruction to others. Manabe and Imase [14] deal with the question of detecting the earliest global state that satisfies the breakpoint condition. Their algorithms depend on the replay and use the trace of earlier execution to poll the processes from which messages are known to be due. Spezialetti [15] proposed a semi-centralized approach based on 'Simultaneous Regions' but these regions do not capture all the global states in distributed computation; so the algorithm is applicable only to stable predicates. Another approach to the problem was proposed by Garg and Waldecker in references [17, 18] where a centralized process checks for the predicate. They also present a distributed version of the algorithm. Drummond and Barbosa [12] presented four distributed algorithms for different classes of predicates. But due to the inherent delay involved in the on-line detection of predicates, they require a checkpointing and rollback mechanism to return the program to the state where breakpoint condition occurred.

The idea of studying the lattice corresponding to the computation poset for predicate detection was first presented in [16]. Later, Garg and Mittal proved that this lattice is distributive and introduced the notion of computation slicing and regular predicates [19]. [20] and [21] present generalization of definition of slicing, algorithms for grafting and algorithms for a much more general class of predicates like Linear, Post-linear predicates etc.

3 Background

In this section, we provide a brief background about the technique of computation slicing.

A distributed system is modeled, as usual, as a set of processes with no shared memory or global clock. Communication among processes is by message passing. The channels between processes are assumed to have infinite capacity and a finite but unpredictable delay.

A computation of such a system can be represented as a set of events that are ordered by Lamport's happened-before relation [9]. An event is either an internal event within a process, or the sending or receipt of a message.

The local state of a process can be characterized by the set of events that had occurred in the process till the state was reached. Since events occurring in a single process are totally ordered, this state can be more concisely represented simply by the last event to occur in the process. A consistent global state (or a consistent global cut) of the system is the set of local states of processes, one for each process, that the system *could* have passed through at some point of time during the computation. It can be easily seen that a consistent cut can be represented by the set of latest events that would have occurred in each process before the corresponding global state was reached. The set of all consistent global cuts of the system for a given computation forms a distributive lattice with respect to the happened-before relation [19].

A *slice* of a computation with respect to a given predicate is the minimal lattice containing all consistent cuts of the computation that satisfy the predicate [19]. Efficient algorithms to compute the slice are known for many classes of predicates, such as, regular predicates, linear predicates, and post-linear predicates [20, 21]. Using the technique of *grafting*, approximate slices for still larger classes of predicates can be efficiently computed [21]. Note that for computing the slice for a given predicate, the computation log must contain all events that affect any variables occurring in the predicte besides all communication events; it is not necessary to log the other internal events in the processes.

Using the slicing technique, it is therefore possible to get the set of all consistent cuts of the system that satisfy the given predicate. Note that the slice corresponding to a given predicate may contain extra cuts that do not satisfy the predicate. Such cuts can be easily removed after the slice computation. Now, given the set of consistent cuts that satisfy the predicate, it is easy to fast-forward the program replay so that execution is stopped only at global states that satisfy the predicate; it only requires that each process is stopped just after its last event specified in the next such cut. In certain cases, it is possible that two cuts satisfying the predicate are directly reachable from the current state (without any intervening states that satisfy the predicate). In such cases, user assistance would be required to select the path that the program execution should take.

4 System Overview

We have implemented a prototype debugger for distributed Java programs that use mpiJava [22] for communication. The debugger uses computation slicing to efficiently implement data breakpoints. In a typical debugging session, the user would first specify the breakpoint condition (the predicate for the computation slice), and then run the program in the record mode. Specifying the predicate before recording the program run is necessary so that only the relevant events are logged (those that change any variables occurring in the predicate, and communication events). After the program run completes, the slice of the computation with respect to the specified predicate is automatically computed by the debugger. Now the user can ask for a replay with fast-forwarding. During such a replay, execution stops only at states that satisfy the predicate. If from the current state, there are two or more directly reachable states that satisfy the predicate, the user is asked to select the next state. Whenever execution stops, the user can examine the values of various program variables etc., as in all other debuggers.

Currently the breakpoint condition must be a a disjunction or conjunction of local predicates. Such predicates are guaranteed to be regular. Support for larger classes of predicates can be easily added to the system by changing the slice computation algorithm.

5 Implementation

The overall design of the prototype debugger is shown in figure 1. The debugger has 5 major modules: predicate parser, tracer, slicer, and fast forwarder. In addition the mpiJava library has been modified to enable event logging and replay. In this section, we briefly describe each of these modules.

5.1 Modified mpiJava Library

mpiJava is a Java wrapper library for native MPI implementations. We modified this library to support event logging and replay. Two major modifications were made to the existing mpiJava library. First, it now maintains a vector clock [10] and includes a vector time stamp in every

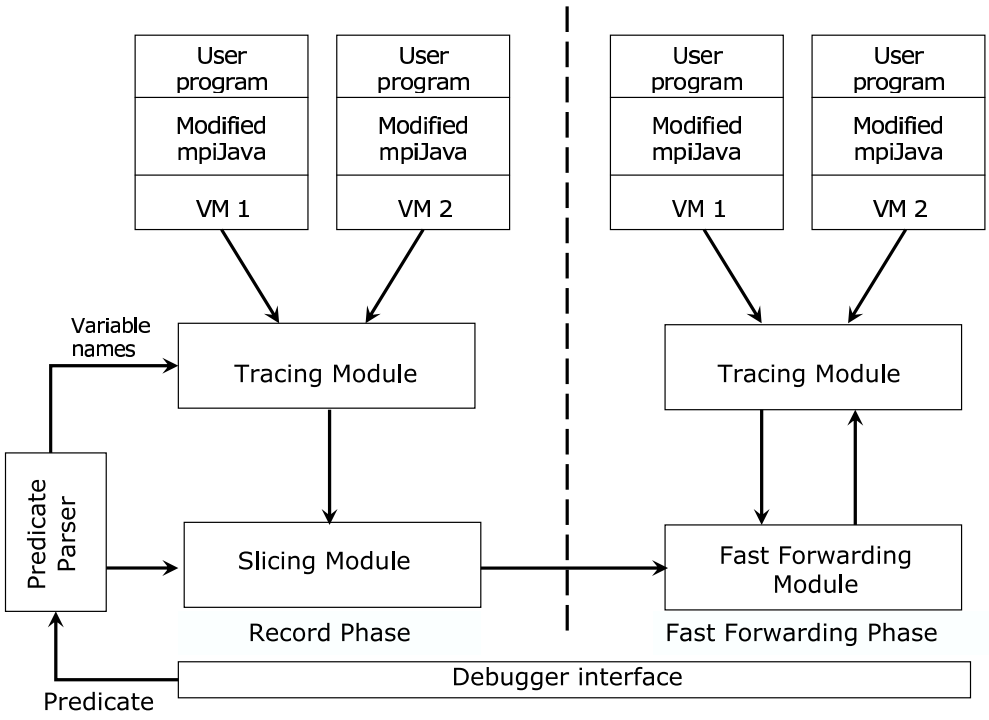

Figure 1: System Architecture

message. This is needed so that the send events corresponding to the message receive events in the log can be determined. Secondly, the modified library can run in two modes: record mode and replay mode. In the record mode, enough information is written to a log so that the communication pattern can be exactly reproduced later. In the replay mode, this log is used to rerun the program with exactly the same message interleaving. Note that this log is different from the computation log maintained for slice computation and fast forwarding.

Every message in MPI has a *tag*. All MPI channels are guaranteed to be FIFO for all messages with the same *tag*. However, a message may overtake another message that has a different tag. A receive call has to specify the sender from whom the message is expected, and the tag of the expected message. The sender can also be specified as `ANY_SOURCE`, indicating that message from any sender can be accepted. Similarly, the tag may be specified as `ANY_TAG`, which means that a message with any tag can be accepted. To handle correct replaying of programs containing receive calls with sender `ANY_SOURCE`, or tag `ANY_TAG`, our version of the mpiJava library logs the source and tag of the message actually received during the record phase. During replay, this information is used to wait for the appropriate message. In the more common case where the sender and the tag are explicitly specified in the receive call, no information needs to be logged in order to reproduce the program execution. This is because, while replaying such a receive call, the message to be accepted can be uniquely determined. This is the first outstanding message from the specified sender with the specified tag.

MPI also permits non-blocking receive operations. Such operations return a *request object* that can either be tested for completion of the communication or for waiting till the message arrives. To faithfully replay the execution of programs that use non-blocking operations, the number of times that the test fails should be the same in the recorded and replay runs of the program. To ensure this, we log the number of unsuccessful tests on a request object. During replay, the test function returns failure for this many attempts before succeeding. This strategy

for handling non-blocking receives is similar to the one in reference [6].

5.2 Predicate Parser

This module parses the predicate given by user and converts it into an internal format to be used by slicing module. It also generates a list of variables that need to be tracked at each process. We only trace those variables that are required to evaluate the given predicate.

Currently there are a few limitations on the kinds of variables that can be used in breakpoint conditions. Any variable occurring in the breakpoint condition must be of a primitive type. In addition, the variable must be specifiable as a Java expression containing only the . (dot) operator, where the first component of the expression is a class name, and the second component is a static member of this class. Such a specification uniquely identifies the variable within the JVM. Note that this restriction does not allow use of those variables which are only reachable through some local variable of a method. This restriction is due to the fact that the JPDA (Java Platform Debugger Architecture) [23] that we use for tracking variables does not support tracking of local variables.

The variables occurring in the breakpoint condition are actually specified as:
<class name>.<static variable name>{.<variable name>}*.<primitive variable>

Essentially the syntax is same as for the Java expressions specified above. For example, `Hello.ch.x` . Note that in this example, `ch` is a static member of the class `Hello`.

The current implementation restricts the breakpoint conditions to be conjunction or disjunction of local predicates. The breakpoint condition format is as follows:

<process no>:{<local predicate$_1$>}<seperator><process no>:{<local predicate$_2$>}...

Here separator can be either 'AND' or 'OR'. The condition should not contain both conjunction and disjunction. The individual local predicates can be any valid boolean Java expressions and can use any arithmetic, logical and relational operators of Java. Variables occurring in the expression are assumed to be on the preceding process number. An example of a valid predicate is `1:{Hello.ch.x>Hello.ch.y} AND 2:{Hello.ch.y>2}.`

In principal, any predicate from the categories that have an efficient slicing algorithm (such as regular predicates, linear predicates, post-linear predicates etc.) can be used as the breakpoint condition. There is, however, no efficient way to determine whether a given predicate belongs to one of these categories. Hence we restrict the predicates to be conjunction or disjunction of local predicates since such predicates are known to be regular. Another option could have been to let the user specify the category of predicate and use the appropriate slicing algorithm.

5.3 Tracer

The tracer module traces the processes running at various nodes using the JPDA (Java Platform Debugger Architecture) API [23]. JPDA provides all the primitives that are required to write a debugger. It supports several modes of debugging like debugger starting the target process, or the target process contacting the already running debugger process, or the debugger process contacting the already running target process. In our system, first a debugger process is started to which all the debugee processes attach themselves. The debugger process creates one thread for every process that connects to it.

The module runs in two modes. In tracing mode, event information is received from the debugee processes, and is stored in internal data structures. It should be noted that there may be many variables, change in whose values can lead to a change in the value of the variables present in the break point condition. For example, a change in the value of the variable `Hello.ch` potentially changes the value of `Hello.ch.x` as well. Any changes in all

such variables are tracked. All communication between the processes is also logged. At the end of the run of the user program, the log is passed on to the slicing module.

In fast forwarding mode, the tracer module gets the information about the global cuts where the program has to stop from the fast forwarding module. It halts the program at those cuts and passes on the control to the fast forwarding module for debugging purposes.

5.4 Slicing Module

This module gets the computation in the form of time stamped events and communication logs from the tracing module and computes the slice with respect to given predicate. It can represent the slice in form of directed graph, queue of events or component graph.

In case of conjunctive predicate, optimal slicing algorithm as described in reference [20] is used. In case of disjunctive predicates, slicing is done bottom up. First the slice with respect to each of the clauses is calculated. Then these slices are *joined* together using grafting. Right now, the predicate parser and slicing module support only these two categories of predicates.

5.5 Fast Forwarding module

The task of this module is to determine all the possible global cuts in the slice that are reachable from the current cut at any given point during the execution of program. It uses the slice generated by the slicing module for the purpose. The information about the next cut is passed on to tracing module which suspends the processes at the requested points and returns the control to the fast forwarding module.

After all processes are halted at the cut, the module checks if indeed the predicate is true at this cut (remember that a slice may have extra cuts at which predicate is not true and also grafting can lead to approximate slices). In case of a false detect, the processes are resumed and the information about next cut provided to tracing module. Otherwise the module goes into debugging mode and presents the user with a prompt for examining variable values at the individual processes. After the user is through with his examination, the processes are resumed to continue in the normal fashion till the next cut of the slice is reached.

The correctness of the fast forwarding assumes that the program is retracing exactly the same steps as in the first run from which the trace to calculate the slice was taken. The instrumented mpiJava layer takes care of replaying the exactly same communication order. Also we assume that the user input is reproducible exactly every time and there is no randomness involved.

It should be noted that there can be more than one ways to fast forward a computation. Since just like the computation, a slice is also a poset, there may be alternative paths for the same execution. So it is possible that starting from some cut of the slice, there may be many next cuts. In such a case, user has to choose one of them to continue fast forwarding. In future, some criterion for choosing one preferred path may be worked out.

6 Experimental Results

A 16 node cluster of dual processor P-III nodes running linux operating system along with lam 6.5.1, mpiJava 1.2.5 and java 1.4.2 was used during the development and testing of the system. Table 1 shows performance data for two of the example programs, ScalProd and Mandelbrot.

ScalProd calculates the scaler product of a pair of vectors of 100,000 floating point numbers each. Mandelbrot computes the Mandelbrot set. In the table, normal is the time taken by the programs with unmodified mpiJava library. Interpreted mode time is the execution time when the user program is run in interpreted mode. Third column shows the execution time when

	Execution Time (mili seconds)				
Program (No. of processes)	normal	interpreted mode	Tracing without Predicate	Tracing with Predicate	Fast Forwarding
ScalProd (5)	324	616	22754	22997	27036
Mandelbrot (5)	1156	1589	15862	16132	74705

Table 1: Time taken by example programs in various modes

no predicate is specified and only communication events are traced. Fourth and fifth columns show execution times for tracing a program with respect to a specified predicate and then fast forwarding it respectively. The time in final column has been measured by using an automated script. The script simply asks the fast forwarding modlue to continue whenever it stops at a cut.

As is evident from the table, the main overhead comes from tracing the communication events. The number of events logged when tracing without any predicate are 24 and 16 for Scalprod and Mandelbrot respectively. So overhead of tracing one communication event comes out to be ~900 mili sec. Tracing a variable incurs little cost in comparison. When tracing with a predicate, extra events logged in Scalprod and MAdelbrot are 5 and 7 respectively. Thus the overhead incurred by one variable change event is around ~40 mili sec. So any complex break condition that involves many variables, can be handled without increasing the overhead much.

Also there is a huge increase in the execution time while fast forwarding the Mandelbrot program. The reason is that while fast forwarding as we choose a particular path through the slice, the computation happening in parallel can be dramatically reduced. This can increase the execution time unexpectedly.

7 Conclusion and Future Work

In this project, we have implemented a system to record/replay and fast forward a distributed java programs. The computation is first recorded depending on the predicate that is provided by the user. On this computation, slicing is used for predicate detection. The program is than replayed using the recorded computation and slice is used to halt and debug the process at the global states at which the predicate is true. Thus Fast Forwarding provides the same functionality as global data breakpoints.

In the system, the onus on the programmer is to express the condition that he wants to check in term to appropriate boolean predicates. Hence we would like to support as many classes of predicates as possible. The system however handles only conjunction and disjunction of local predicates. In principal, it can handle all the classes of predicates with respect to which we can calculate the slice efficiently. So the first step is to extend the predicate parser to handle at least all those classes for which slicing algorithms are known. Also it is expected that the next version of JPDA will have facilities for tracing the local variables of methods. So than we can handle the predicates on all type of variables in the program. One more way to extend the debugging capabilities of the system can be the introduction of checkpointing. Using checkpointing, we do not every time need to start the process from the beginning.

References

[1] R. Netzer and B. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. In Supercomputing '92, 502-511. Minneapolis, MN, November 1992.

[2] C. Clemencon, J. Fritscher, M. Meehan and R. Ruhl. An Implementation of Race Detection and Deterministic Replay with MPI. Technical Report CSCS TR-94-01, Swiss Scientific Computing Center, January 1995.

[3] J. Chassin DE Kerogommeaux, M. Ronsse and K. De Bosschere. MPL: Efficient Record/Replay of Nondeterministic Features of Message Passing Libraries. In J. Dongarra, E. Luque and T. Margalef, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface(EUROPVMMPI99), vol 1697, LNCS, 234-239. Springer, September 1999.

[4] M. Neyman, M. Bukowski and P. Kuzora. Efficient Replay of PVM Programs. In J. Dongarra, E. Luque and T. Margalef, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface(EUROPVMMPI99), vol 1697, LNCS, 234-239. Springer, September 1999.

[5] D. Kranzlmuller and J. Volkert. NOPE: A Nondeterministic Program Evaluator. In Proceedings of ACPC99, 490-499, February 1999.

[6] M. Ronsse, J. Chassin DE Kerogommeaux and K. De Bosschere. Execution Replay for an MPI based Runtime System. In E. D'Hollander, G. Joubert, E. Peters and H. Sips, editors, Parallel Computing: Fundamentals and Applications, Proceedings of the International Conference ParCo99, 656-663. Imperial college Press, 2000.

[7] M. Weiser. Programmers use slices when debugging. *Communications of the ACM, 25(7):446-452,1982*

[8] B. A. Davey and H. A. Priestley. *Introduction to lattice and order.* Cambridge University Press, Cambridge, UK, 1990.

[9] L. Lamport. Time,Clock and the Ordering of Events in a Distributed System. *Communications of the ACM,* 21(7):558-565, July 1978

[10] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG),* pages 215-226. Elsevier Science Publishers B. V. , 1989.

[11] K.M.Chandy and L.Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems,* 3(1):63-75, February 1985.

[12] Lucia Maria de A. Drummond and Valmir C. Barbosa. Distributed breakpoint detection in message-passing programs. Journal of Parallel and Distributed Computing 39 (1996), 153-167.

[13] B. Miller. and J. Choi. Breakpoints and Halting in distributed programs. Proceedings of the Eighth International Conference on Distributed Computing Systems, 316-324, 1988.

[14] Y. Manabe and M. Imase. Global Conditions in Debugging Distributed Programs. Journal od Parallel and Distributed Computation, 62-69, 15(1992).

[15] M. Spezialetti. An Approach to reducing delays in recognizing distributed event occurrences. Proceedings of the ACM Workshop on Parallel and Distributed Debugging, 155-166, 1991.

[16] R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. Proceedings of the ACM Workshop on Parallel and Distributed Debugging, 167-174, 1991.

[17] V. K. Garg and B. Waldecker. Detection of Unstable Predicates in Distributed Programs. Proceedings of 12th Conference on Foundations of Software Technology and Theoretical Computer Science, 253-264, 1992.

[18] V. K. Garg and B. Waldecker. Detection of Weak Unstable Predicates in Distributed Programs. IEEE Transactions on Parallel Distributed Systems, 299-307, 5(1994).

[19] V.K. Garg and N. Mittal. On Slicing a Distributed Computation. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems(ICDCS),pages 322-329,Phoenix, Arizona, April 2001.*

[20] N.Mittal and V.K.Garg. Computation Slicing: Techniques and Theory. In *proceedings of the Symposium on Distributed Computing(DISC),pages 78-92,Lisbon,Portugal,October 2001*

[21] Neeraj Mittal and Vijay K. Garg, Software Fault Tolerance of Distributed Programs using Computation Slicing, Proc. IEEE International Conference on Distributed Computing Systems (ICDCS), Providence, Rhode Island, pages 105-113, May 2003.

[22] mpiJava. http://www.hpjava.org/mpiJava.html

[23] JPDA. http://java.sun.com/products/jpda/index.jsp