A Virtual Machine Approach to Parallel Debugging

by

KUNHUA LIN

B.Eng., Wuhan Technical University of S&M, 1983
M.Eng., Institute of Computing Technology, Academia Sinica, 1989

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the Faculty of Graduate Studies

Department of Computer Science

We accept this thesis as conforming to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

July, 1993

© Kunhua Lin, 1993
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Computer Science
The University of British Columbia
6356 Agricultural Road
Vancouver, Canada
V6T 1Z2

Date: August 30, 1993
Abstract

Debugging is generally considered to be difficult. The increased complexity and nondeterminism of parallel programs makes it even more difficult. It is one of the reasons that parallel machines are not widely used for computationally intensive applications even though recent progress on VLSI technology has significantly reduced the cost of building these machines.

In this thesis, a two-phase approach to debugging parallel applications based on task-oriented virtual machines is proposed. In the first phase, a message passing event history of the application is constructed and analyzed against the specifications of the application to automatically identify and localize fatal errors in terms of the task number, the processor identity and the user program process. In the second phase, the code segments identified in the first phase are debugged in a simulated environment using effective sequential debugging techniques. This provides us with a technique to deal with the complexity of parallel programs through high-level abstractions and a divide-and-conquer strategy. As it is solely based on the execution history, this approach avoids the non-reproducibility that results from the nondeterministic characteristics of a parallel program.

We have implemented a monitor and trace analyzer on a transputer-based multicomputer for two virtual machines, PFVM (Processor Farm Virtual Machine) and DCVM (Divide and Conquer Virtual Machine). The effectiveness and limitations of our approach are demonstrated and evaluated.
# Table of Contents

Abstract

Table of Contents

List of Figures

Acknowledgment

Dedication

1 Introduction

1.1 The Problems

1.2 Motivation

1.3 Thesis Overview

2 Overview

2.1 Terminology and a Framework

2.2 General Overview of Parallel/Distributed Debugging

2.3 Display the Behaviours of Communicating Processes

2.4 Replay

2.5 Assertion Checking

2.6 Summary
3 VM-based Applications and Their Message-passing Behaviour 22
   3.1 Introduction to VM-based Programming Environments ............... 23
   3.2 Processor Farm Virtual Machine ...................................... 27
      3.2.1 PFVM Interface .................................................. 29
      3.2.2 Intermediate PFVM .............................................. 30
   3.3 Divide and Conquer Virtual Machine ................................. 31
      3.3.1 DCVM Interface .................................................. 32
      3.3.2 Intermediate DCVM .............................................. 34
   3.4 Debugging VM-based systems .......................................... 34
      3.4.1 Classification of Errors ......................................... 35
      3.4.2 Discussion ....................................................... 37

4 A Two-phase Approach to Debugging VM-based Applications 38
   4.1 Two-phase Debugging Approach ...................................... 38
   4.2 Definitions ............................................................ 41
   4.3 Fault Models ........................................................... 43
   4.4 Summary ................................................................. 45

5 Implementation ............................................................... 47
   5.1 Overall Design of the System ....................................... 47
   5.2 Instrumentation ......................................................... 49
   5.3 Event Tracing ............................................................ 49
      5.3.1 Environment ....................................................... 50
      5.3.2 System Structure of Tmon ....................................... 53
      5.3.3 Modification to Tmon ............................................ 55
# List of Figures

2.1 A Debugging Process Model ........................................ 8
2.2 Animation of Message-passing in Belvedere .......................... 14

3.3 Overall System Structure of a VM-based System .................... 24
3.4 Overall System Structure of a Multi-VM System ................... 25
3.5 Specification of an Example Program ............................... 26
3.6 Processor Farm Virtual Machine ................................. 28
3.7 IPFVM in a Chain .............................................. 30
3.8 IPFVM in a Binary Tree .......................................... 31
3.9 Divide and Conquer Virtual Machine .............................. 33

5.10 Overall Design of the System .................................... 48
5.11 System Design of deTrEK ....................................... 62
Acknowledgment

First and foremost I would like to express my sincere thanks to both of my supervisors, Dr. Alan Wagner, without whose friendly guidance this thesis may never have been complete, and Dr. Samuel Chanson, whose advice has often gone far beyond academic research.

I wish to thank Mandeep Dhami for his friendship and for proof-reading the thesis. Without his help, this thesis would be much less readable. I also wish to thank the other members of my research group, H. V. Sreekantaswamy, Sameer Mulye, Xiangdong Cai and David Feldcamp for their discussions and friendship. My special thanks go to Norman Goldstein, who introduced me to DSRG's transputer system and its magic wires. Also I would like to thank Jeff Beis for providing me with a test case for demonstration and evaluation of my tool.

I would also like to say thanks to all the people who have made my stay in UBC a rewarding experience due to their friendship, in particular, Grace Wolkosky, Ming Koon Lau, Roland Mechler, Marcelo Walter, Charles Mathieson, Helen Wong, Sijiang Zhang, Xun Li, Lianwen Zhang, Ruping Qi, and numerous others.

Finally, I am indebted to my wife, Caiping Zhang, for her love and constant support.
To my father, Limin Lin
Chapter 1

Introduction

Due to the progress of VLSI technology the cost of building massively parallel computers has been significantly reduced. However, the difficulty in programming these machines prevents them from being used widely. A programming environment based on virtual machines has been developed at UBC to ease parallel programming\cite{Sre93, Fel92}. The focus of this thesis is to provide debugging support in such an environment. A new debugging approach is explored and the design and implementation of a real-time monitor and post-mortem trace analyzer are presented.

1.1 The Problems

Debugging is a difficult job that requires considerable experience in program development because it relies on heuristic insights. There are several quite effective debuggers available for sequential code, such as “xcodecenter” and “dbx”, which are based on the traditional debugging method – stop-and-look. However, this method is no longer sufficient, or even feasible, for parallel programs because of the following two characteristics inherent in all parallel programs.
Chapter 1. Introduction

Complexity

In a parallel system there are multiple physical processors and often multiple processes on each processor. Multiple processors and processes result in multiple threads of control which execute concurrently and interact with each other. Thus there is extensive communication and synchronization between them.

Understanding the execution of distributed systems with multiple threads of control is difficult. Even if we attached a sequential debugger, such as “dbx”, to each process the simultaneous coordination and management of such a large number of sequential debuggers would be a challenging task. Finally, because there is no single locus of control, problems such as the probe effect and error latency are more serious.

Non-determinism

The different behaviour or different results of different runs of the same program with the same input is referred to as non-determinism. There are two sources of nondeterminism in message passing systems. First, there is the non-deterministic behaviour inside a process. This is caused by nondeterministic constructs in the programming language, such as ALT in OCCAM, and access to real time clocks. Second there is the non-deterministic behaviour due to external events, such as messages, timers or interrupts. Asynchronous communication between processes is also a major source of non-determinism. The order in which messages arrive at the destination may be different in different executions with the same input. This order can affect the execution of the process that receives the messages. In addition, the runtime system that manages the message passing and processes is also a source of non-determinism.
The major problem with non-determinism is "non-reproducibility", a consequence of non-determinism that may make it impossible to reproduce an erroneous state. Traditional debugging is based on repeated executions of the program; setting the breakpoints, observing and changing the values of variables to verify some hypotheses about the program. Non-reproducibility of the execution makes this ineffective as the repeated executions might not manifest the same behaviour. Furthermore, breakpoints are difficult to define because there is no single point of control.

1.2 Motivation

In general, to deal with the increased complexity of parallel programs one can develop higher level abstractions and tools. One such approach at UBC is based on the definition of virtual machine. A virtual machine (VM) is an abstract model of computation that encapsulates a parallel programming paradigm, such as processor farm, SPMD, etc. It is a restricted model of computation corresponding to a widely used strategy for parallelizing a program.

The objective of developing a VM-based programming environment is to ease programming and promote reusability while minimizing the loss of performance one might expect from using high-level abstractions. Although the VM-based programming environment significantly reduces the complexity of parallel programming, it lacks debugging tools. The chief motivation for this thesis was to investigate tools to help the programmer understand the behaviour and debug VM-based applications.

Certainly, one simplifies parallel programming by hiding the lower level management of communication and process scheduling. For the task-oriented systems we have investigated this has been accomplished in the VM-based environment where the user is only
required to supply a few sequential functions. Intuitively this can simplify debugging as it is now possible to debug the sequential functions individually as sequential programs. However, there remains some errors that will not appear until the program runs on a parallel system. Inevitably therefore one has to debug their program in a parallel environment. There is the interesting question as to whether the restricted models of computation can be used to ease debugging and if so, how?

1.3 Thesis Overview

To overcome the complexity and non-reproducibility of parallel programs we have investigated a two-phase hierarchical approach to debugging VM-based applications. It utilizes the restricted model of computation in the VM-based programming environment to divide the debugging into two phases. In the first phase, the execution of the system is monitored and a high-level message passing history is recorded. The event history is then compared with the specification of the application to identify the location of errors at a level close to the user’s conceptual model of programming. This process can be done automatically after the fault models for the virtual machines have been developed. The construction of these fault models is simplified because of the restricted model of computation on which it is based. In the second phase, a simulation environment is provided, which takes as input the event history, to ensure that the errors located in the first phase are reproducible by repeated execution of the problem. The debugging environment in the second phase can be implemented as a driver program to simulate the VM along with a traditional debugger. As a result, programmers debug their code in a familiar sequential debugging environment using proven effective knowledge of debugging, without the non-reproducibility and complexity often associated with debugging parallel programs.
Chapter 1. Introduction

Chapter 2 reviews previous work in the area as it pertains to this work. After defining some basic terms it introduces a general framework for debugging, and classifies and evaluates various approaches to the debugging of parallel programs based on this model. Approaches that relate closely to our method are discussed.

Chapter 3 introduces a programming environment based on models of two task-oriented virtual machines, a Processor Farm Virtual Machine (PFVM) and a Divide-and-Conquer Virtual Machine (DCVM). The message-passing behaviour of the applications based on these two virtual machines is discussed. Finally, the problems of debugging a VM-based system is discussed and possible errors are classified and analyzed.

Chapter 4 proposes a two-phase approach to debugging VM-based applications. Accordingly, fault models for PFVM and DCVM are developed for the automation of the first phase.

Chapter 5 describes our implementation work. First, a brief overview of the underlying hardware and software environment is given, followed by the description of the modifications that needs to be made to Tmon [WJC93], a parallel performance monitor for the transputers. Next, the algorithm for generation of the vector logical time stamps is presented, followed by the design and implementation of the trace analyzer. The chapter ends with a brief description of the user interface to the trace analyzer.

Chapter 6 demonstrates how the monitoring and trace analyzing tools can be used to debug VM-based applications. The experience of debugging an actual PFVM application is described.

Chapter 7 concludes by summarizing the contributions of this thesis with suggestions for enhancements to the debugging tool and future research.
Chapter 2

Overview

In Section 2.1, we introduce some terminology and describe a general framework for debugging. A general overview of distributed/parallel debugging is given in Section 2.2. In the remaining sections, we discuss the various approaches that have been proposed for debugging parallel or distributed systems. Most approaches to parallel debugging have, as a first step, an event tracing of the program execution. A similar approach is proposed in this thesis. In the second step, depending on what is done with the trace, these approaches fall into three categories: displaying the behaviour, re-execution and assertion checking. Each of these approaches are discussed in Section 2.3 to 2.5. Section 2.6 summarizes this chapter.

2.1 Terminology and a Framework

Debugging begins when the programmer discovers that the execution behaviour deviates from the specified behaviour of the program. This difference is usually noticed by simple problems such as an incorrect value printed or the program not terminating. This is referred to as a program failure, the result of an erroneous program state. An error is the part of an erroneous state that differs from a valid program state, such as an
incorrect value for a variable or the program following an unexpected execution path. The invalid state transition that first caused an erroneous state is referred to as a fault. The manifestation of a fault produces errors in the program state which in turn lead to program failure. A bug is a fault and the corresponding error(s).

Naturally, debugging is the process of detecting errors, locating faults and correcting them. Most errors are detected by program testing and most of the debugging time is spent locating faults. In locating faults, programmers first develop hypotheses about the errors and the faults, and verify or refute these hypotheses by examining the program. In correcting the faults, programmers again develop hypotheses about how to modify the program and once again verify or refute them. Based on this three step process of debugging, Araki, Furukawa and Cheng [AFC91] proposed, as a general framework, the debugging process model shown in Figure 2.1. In this model, programmers begin with a set of hypotheses, modify that set, select hypotheses for verification, and verify or refute the selected hypotheses until bugs are fixed, or more generally some level of confidence has been reached. An extended version of this framework is described below in more detail.

Hypothesis set generation. In Araki et al’s model, hypotheses are related to program source code, its specification and its behaviour. Programmers hypothesize about the errors in the program, including the locations in the program where errors may occur, the causes of the errors, what constitutes a correct or incorrect behaviour, and modifications to correct the faults. These hypotheses include the facts proven so far about the properties of the program and its errors, and the programmer’s empirical knowledge about the development and specification of the program. As most errors are detected by program testing, for simplicity, Araki et al treat the error report as the initial hypothesis set.
However, communication and synchronization errors in parallel programs are difficult to detect by program testing. Therefore, we consider error detection to also be an important part of debugging and extend the framework by including error detection in this step.

**Hypothesis set modification.** As the process of debugging proceeds, programmers modify the hypothesis set by creating new hypotheses, by further constraining or refining the existing hypothesis set, and by changing the authenticity of hypotheses based on the
Chapter 2. Overview

results of previous hypothesis verification.

**Hypothesis selection.** Randomly selecting a hypothesis to verify is not efficient. Instead, a hypothesis should be selected according to a strategy; using tactics to simplify the error condition, narrow the suspicious region, expand the certified region, change the point of view, or by weighing the significance of each hypothesis.

**Hypothesis verification.** A hypothesis is verified by examining the program and its behaviour. As a result of these examinations, a hypothesis is either verified as true or false, or remains unverified. Even when a hypothesis is unverified, one may have obtained some information about the program that will help the debugging process.

Most of today’s debugging tools provide mechanisms for observing program execution and helping programmers understand program behaviour. Source-level debugging, enforced execution control, backward tracing, and other techniques are often used to verify hypotheses. These tools support the verification of hypotheses that programmers develop, but do not support hypothesis generation or selection. A complete debugging tool must support every stage of the debugging process: hypothesis generation, modification, and hypothesis selection and verification. In evaluating debuggers for parallel programs, the following four criterion were considered.

- **Adequacy:** A parallel debugging tool should support all the steps in the process model of debugging. Specifically, the debugger should address the complexity of parallel programs by presenting the execution behaviour of the program in an understandable way. The debugger should also eliminate nondeterministic execution so that errors are reproducible.

- **Productiveness:** Faults can potentially appear anywhere in a software system and whereas some faults are relatively easy to detect, locate and correct, other
faults are not. A good debugger should aid in finding and correcting all faults, especially those that are the most difficult to find.

- **Feasibility:** A debugger is a tool that is eventually implemented in some specific underlying system. It is important to ensure that the debugger is implementable in the targeted system.

- **Efficiency:** The overhead introduced by a debugger varies from one approach to another. Even though the acceptable range of the overhead is relatively large compared to other programming activities, it should not be overwhelming.

### 2.2 General Overview of Parallel/Distributed Debugging

In recent years, interest in debugging has increased, particularly in parallel and distributed systems. McDowell and Helmhold [MH89] give an extensive survey of distributed debuggers developed up until 1988. They classify approaches to debugging into four groups: traditional debugging, event-based, graphic display and static analysis. The event-based and graphic display approaches require, as a first step, the monitoring of the execution. While the execution is viewed as a sequence of events in the event-based approach, it is viewed as a flow of control and data in the graphic approach. Since both of these approaches require events to be recorded we will use only three categories to describe these approaches.

1. **Extending traditional debugging techniques to parallel programs:** The most important feature of a typical sequential debugger is the capability to set breakpoints. Breakpoints allow the programmer to stop the execution and examine the state of the program. A straightforward debugging technique for parallel programs is to
treat the communicating processes as a set of sequential processes, and to attach to each process a sequential debugger. An obvious disadvantage of this technique is that it is too low level. As a result, some debuggers incorporate some high-level abstractions [Smi85]; however it is still difficult to coordinate and synchronize these debuggers, especially as the number of concurrent processes grows larger. In addition, since this technique dynamically controls the execution of a parallel program, it can potentially change the behaviour of the program. In particular, because of non-determinism, it may not be possible to re-exhibit the bugs. In conclusion, this technique does not address the complexity and the nondeterminism present in parallel programs.

2. Based on the event history of program execution: The approaches in this category view the execution of a parallel program as a sequence of events which taken together constitute an event history. After the generation of the event history, by monitoring the execution, the history can be used in several ways. It can be presented to the user as debugging information, such as textually, time-process diagrams, or animation. The history can also be used to re-play the execution of the program.

More sophisticated debuggers provide assertion checking facilities for behaviour comparison. It allows the user to specify a behaviour and compare it with the behaviour shown in the history. These approaches do not actively control the execution as a traditional sequential debugger does, but rather they are passive. Through high-level abstraction of the events these approaches can reduce the complexity of parallel programs, and, compared to sequential debugging event based
monitoring significantly reduces interference to the execution behaviour of the program. However, it is not possible to completely eliminate the probe effect since probes do still have to be inserted into the program and thereby affect the execution. The only way to make monitoring non-intrusive is by hardware monitoring. These approaches are discussed in more detail in Section 2.3 to 2.5.

3. **Static analysis of program:** The approaches in this category are based on symbolic flow analysis of parallel programs. Static analysis can be used to detect errors, such as race conditions and synchronization errors [MC89, MA87, McD89, TO80, Tay84]. It solves the problem of nondeterminism, however it is limited to error detection. Its computational complexity also limits its use.

Because of the complexity and nondeterminism of parallel programs, traditional debugging is not sufficient for parallel debugging. Given its complexity, even if the facilities existed, it would be difficult for the programmer to manage and control the execution. The computational complexity of static analysis makes it infeasible. In contrast, approaches based on event histories that do not actively control the execution during testing and abstraction can reduce the complexity of the debugging information. These advantages make event based debugging more practicable and effective in parallel/distributed environments.

### 2.3 Display the Behaviours of Communicating Processes

One natural way of helping the programmer to understand the program is to show its behaviour, recorded as an event history. Highly parallel programs are often best understood in terms of logical patterns of interprocess communication [HC89]. In displaying
communication interactions, time-process diagrams are often employed to provide a static after-image of communication over time. In most cases, a time line is drawn along one dimension, while individual processes are distributed across the other. Moving back and forth along the time line reveals the sequence of communications recorded during execution. Message passing is depicted by connecting the two communicating processes (see [FLM89]). Hough and Cuny [HC89] [HC90] go further in this direction by animating the time process information. By describing and animating abstract user-defined communication events, their debugger *Belvedere*, helps users to compare the intended patterns with the patterns that occurred during execution. *Belvedere* is a trace-based, post-mortem debugger, intended for distributed memory architecture machines. A prototype has been implemented on a multiprocessor simulator.

To facilitate the identification of communication patterns, *Belvedere* allows the user to define these patterns as abstract events and use *perspectives* that restrict the displayed behaviour according to its perspective. There are two perspectives, the *process perspective* and the *consensus perspective*. From the process perspective only the events seen by a single process are animated. From the consensus perspective high-level events consistent with that seen by all participating processes are animated. As shown in [HC90], *Belvedere* can be used to detect sequencing errors, missing communication errors and extraneous communication errors.

Hough and Cuny exploit the fact that most parallel programs exhibit recurring or uniform patterns of interprocess communications. *Belvedere* allows the user to specify the spatial arrangement of processes to reflect these logical patterns. Figure 2.2 shows a sample frame from an interaction animation, configured as a hypercube. Highlighted arrows represent SENDs, highlighted ports indicate messages received via GETs, and
A disadvantage of this type of animation is that it is not sufficient for more complex interactions, especially as the number of processes increases. As Pancake and Utter [PU89] pointed out the effectiveness of visualization is bounded by the degree to which the representations of running time behaviour correlate with the language constructs used to incorporate parallelism, as well as the logical framework adopted by the programmer. How to select a perspective remains a problem even when the perspective technique can
deal with the complexity. This approach also does not address the problem of displaying events for those programs in which the patterns of interprocess communication are not regular. A further disadvantage of debugging parallel programs by portraying behaviours is that often many concurrently executing processes have the same, or similar, behaviour. Without any "filtering" the programmer faces an overwhelming amount of repetitive information. There also remains the question of accuracy. How accurately does the post-mortem animation of the abstracted events depict actual program behaviour?

The debugging tool developed by Caerts, Lauwereins and Peperstraete[CLP91] is also based on the animation of a program represented hierarchically and graphically. Francioni and Gach [JMFA91] experimented with the use of sounds to portray the behaviours of program execution. As previously mentioned a limitation to these approaches is that visualization or auralization only helps to detect errors. There is no support for locating faults or verifying hypothesis, as there is no tool in the debugger to show where these anomalies are located in the source program.

2.4 Replay

Because of the nondeterminism of parallel programs, successive executions of the same program may lead to different behaviours and even different results. To solve this problem, an approach called execution replay has been proposed [Els89, Wit89, Sto89, PL89, CMN91, LSS90, SS90]. In this approach, information about critical events, such as system calls and message passing, are recorded during the initial execution of the program. This information is then used to control the re-execution (replay) of the program so that it results in an equivalent execution. The traditional approach to debugging is often used during replay.
According to Leu, Schiper and Zramdini [LSS90], there are two types of execution replay techniques, “data driven” and “control driven”. Data driven replay techniques [PL89, Els89, Sto89] are relatively easy to implement for most common underlying communication systems. During replay, an individual process can be re-executed separately. However, the amount of information that needs to be recorded is very large and the time to gather the information significant. In terms of message passing, it requires not only the type and order of messages, but also their contents. The overhead of monitoring can significantly modify the initial execution and make the execution replay meaningless. In contrast, control driven replay techniques [LSS90, FLM89, Wit89] limits the amount of information. It only records the relative order of the events pertaining to process interaction. The contents of the message are generated during replay. Thus individual processes can not be individually replayed, all the processes have to be replayed together.

In addition, some debuggers not only record message events and system calls, but also periodically save all variables at checkpoints. During replay, execution can be restarted from a checkpoint. This approach to debugging is called reverse execution [PL89, Wit89]. As a checkpoint usually contains a lot of information and checkpoints may have to be recorded often, this method increases the overhead of recording the execution history. To reduce this overhead, Choi, Miller and Netzer [CMN91, MC89] introduce a mechanism called incremental tracing. It is based on the idea of need-to-generate and attempts to reduce the amount of information logged during execution. The gaps between the information gathered in the log and the information needed to do the flowback analysis are filled in incrementally by information obtained by statically analyzing the program. This information includes a static program dependence graph and a program database. They have introduced a technique called flowback analysis which allows the programmer
to examine dynamic dependence in a program's execution history without having to re-execute the program. Their debugger, PPD (Parallel Program Debugger), works by showing the dynamic dependence among the program objects, such as variables and procedures, through the *dynamic program dependence graph*.

We believe that the dependence graph may still be too complicated for the programmer to examine efficiently in flowback analysis. Simply showing the graph to the user is not enough since the graph differs from the user's conceptual model of the problem. Therefore it is necessary to develop a high-level tool to access the information in the graph and present it to the user in a meaningful way. In addition, the algorithmic complexity of generating the dependence graph remains to be addressed and, with respect to the general framework, flowback analysis does not support the verification of the hypothesis set.

Replay techniques aim at reproducing the execution behaviour, usually in a traditional debugging environment. However, it inevitably modifies the behaviour during the monitoring of the initial execution. It is possible to permanently embed monitoring probes into the program, however, this is only feasible when the overhead of monitoring is acceptable by the user. A final disadvantage of replay is that during replay, traditional debugging is used where again there is little help to deal with the complexity of parallel programs.

### 2.5 Assertion Checking

Typically, once an event history has been obtained, a programmer wants to test whether or not the history conforms to certain properties. If it does not conform, the programmer
needs to find out what the differences are and where they occur. Based on this observation, Hseush, Kaiser and Ponamgi\cite{HK90, HK88, PHK91} describe a “data-oriented” debugging approach for concurrent programs. This debugger provides \textit{data-path expressions} (DPE) as a formal notation for the user to describe expected or unexpected execution behaviours of a program. Generally, a DPE is in the form of \{<events>\} \{<debugging actions>\} <operator>. The events (<events>) can be a data event, such as variable $X$ becomes equal to 0, or control events, such as a function or procedure is called, or message events, such as sending a message or receiving a message. The debugging actions are the actions to perform when the event occurs, for example printing the value of a variable. These actions also can be used to change the execution path at run-time or to assert additional control, such as counting the number of times a specified variable is accessed. The operator specifies the temporal relation between adjacent events; sequencing($;$), repetition($*$), selection($|$) and total concurrency ($&&$) are some of the relations. Operators are only allowed when there are DPEs following them. During execution, the debugger automatically compares the actual behaviors with those described in the DPE. As an implementation mechanism, they propose \textit{predecessor automata} for event recognition. A predecessor automata is a finite state automata in which each transition is labeled with both a process event and a set of immediate predecessor events, on which the event is causally dependent. Using DPE, the behaviour of a program can be described at a level of abstraction specified by the programmer. Users are free to debug their programs at the level of individual source code statements or at the level of interprocess activity or any combination of the two.

Bates\cite{Bat89} views a system’s behavior as a stream of primitive event occurrences and provides powerful operators for composing primitive events into high-level models that
can be recognized by the debugger. In this debugger, behaviours are specified in an *Event Definition Language* which can be automatically compared with the actual behaviours. For more complicated patterns, it provides a *graphical communication description language* (GCDL) to describe the expected logical patterns of interprocess interactions. The language allows flexibility in describing patterns, in a top-down fashion, at all levels of abstraction. This approach is similar to that of the debugger developed by Francioni and Gach [FG90] except that Francioni *et al* 's debugger provides a user graphical interface to display the expected communication patterns.

Rosenblum[Ros91] uses yet another language called a *Task Sequencing Language* (TSL) to describe concurrent systems written in Ada. By providing meaningful comparison of behaviours, these debuggers help users to verify their hypotheses about the errors. However, the expressibility of the specification language depends on the implementation. There also remains the problem of specifying the correct behaviours to be compared with the actual ones. Both of these assertion checking approaches assume reproducibility and do not address the problem of nondeterminism. Another disadvantage of these approaches is that the user has to learn another language in order to debug the program. Finally, it is non-trivial to describe the behaviour of large complex problems and this itself might introduce bugs!

Another debugger which provides assertion checking is that of Goldszmidt *et al* [GKY89] [GYK90]. It is an integrated system for debugging distributed programs written in a concurrent high-level language based on message-passing for interprocess communication (i.e., OCCAM). It provides a variety of user-interface, monitoring and analysis tools integrated around a uniform interprocess communication model. After collecting the events of program execution, the available analysis tools are *Queries, Assertion Checker*
and Replay. Queries provides a facility for the programmer to ask about the state of program execution at some point defined by a logical time stamp or some other condition. Assertion Checker allows temporal assertions to be checked against the event history. The Replay tool aids in visualizing the evolution of a computation by displaying a simulated replay of the events. The tools also include a Scheduler that accepts control commands from the user and accordingly schedules the running of application processes, and a Driver that enables the user to manually simulate an external environment for running a single process or a set of processes without the rest of the environment.

Their debugging tool integrates the traditional approach of stop-and-look debugging with an event based high-level approach in a simulated environment. A disadvantage is that in a real distributed environment tools such as the Driver and the Scheduler would be difficult to implement and the overhead unacceptable. Finally, to manually simulate a process environment is not feasible for debugging parallel application programs that are large and execute for a long time.

2.6 Summary

Traditional debugging approaches require active control of the execution of a program. It greatly changes the behaviour of the parallel program and may cause the bugs to disappear when a sequential debugger is simply attached to each process. Furthermore, it is difficult to simultaneously coordinate and manage a large number of sequential debuggers, even when nondeterminism is not a problem. The static analysis approach avoids the execution of the program, thereby eliminating the problem of non-reproducibility caused by nondeterminism. However, it is usually limited to error detection and the algorithmic complexity of the analysis is intractable. Event-based debugging approaches record the
behaviour by monitoring the execution events. They do not control the execution activity. An event history can then be presented to the user for replay or assertion checking. Through high-level abstraction of events, these approaches do reduce the complexity of understanding the behaviour of parallel programs.

Debugging parallel/distributed programs is a difficult problem, there is no simple solution. As shown in the debugger developed by Goldszmidt [GYK90], a parallel/distributed debugger needs to integrate a variety of techniques. The traditional debugging approach remains attractive if the complexity of a parallel program can be reduced and the reproducibility of execution assured. To reproduce the execution, event-based monitoring and control driven replay is effective and feasible. The assertion checking provides a facility for high-level abstraction and automatic hypothesis verification and is a promising technique to deal with this complexity. It is desirable to integrate these techniques so that the assertion checking localizes the errors in a high level abstraction and after that, the error prone parts can be debugged in a traditional debugging environment where reproducibility is assured by using the event history to control the re-execution.
Chapter 3

VM-based Applications and Their Message-passing Behaviour

A virtual machine is a restricted model of computation that corresponds to an ideal architecture for a specific parallel programming paradigm. It consists of a number of abstract processors and a communication network. An abstract processor performs specific operations on high-level abstract data units and interacts by message-passing in a network. The communication network provides the facility for these processors to communicate with each other. A Task-oriented Virtual Machine is a virtual machine with the following characteristics:

- A single input and output stream,
- abstract data units called tasks which can be processed independently, and
- the flow of tasks is the only communication in the network.

Since we consider only task-oriented virtual machines throughout this thesis, unless stated otherwise, it will simply be referred to as a virtual machine (VM).

The basic objective in developing a VM-based programming environment is to promote ease of use, and reuse, as far as it is consistent with expressiveness and efficiency.
Virtual machines are implemented as pluggable software components (i.e., templates) for use as basic building blocks in the construction of parallel applications. It allows users of the system to develop parallel applications without requiring detailed information about the underlying architecture and run-time system. Each template corresponds to a parallel programming paradigm (e.g., processor farm, divide and conquer, vector model).

In Section 3.1, we describe the concept of VM-based programming and its environment. In Section 3.2 and 3.3, two specific VMs, processor farm and divide and conquer are described and their behaviour analyzed. In Section 3.4 we classify possible errors and discuss the problem of debugging these systems.

3.1 Introduction to VM-based Programming Environments

In a distributed memory multiprocessor system a parallel program must communicate via message-passing. A VM-based programming environment provides an easier way to program such systems since it hides the scheduling, load-balancing and other issues of distributed (parallel) computing.

As shown in Figure 3.3, a VM-based programming environment is a hierarchical system consisting of three layers: an application layer, a virtual machine implementation layer and an implementation machine layer.

The middle layer of the system is the virtual machine implementation. It consists of two parts, the virtual machine interface and the Intermediate Virtual Machine (IVM). The interface is the language interface of the VM, while the IVM is the implementation of the VM run-time system. The interface is what the programmer sees while programming the system. The IVM is the combination of the VM's conceptual model and the topologies which are suitable for this virtual machine. Depending on the VM and the
implementation machine there may be more than one IVM corresponding to a single VM. Multiple IVMs are provided so that the programmer can tune performance by changing the configuration without altering the program.

The top layer is the application: the user code built on the virtual machine. The bottom layer is the implementation machine, which can be further divided into two sublayers: toolkit and target architecture. The toolkit sublayer is a set of primitives
Figure 3.4: Overall System Structure of a Multi-VM System

- Machine Dependent Part
- Machine Independent Part

VM Interface: VM 1, VM 2, ..., VM m

IVM 1, IVM 2, ..., IVM n

Target Architecture

Multiprocessor Toolkit

Implementation Machine

Application

Machine Dependent Part

Machine Independent Part
provided for the implementation of the virtual machine. It could be an operating system or a set of tools, which supports the communication among processes in the system and provides other services such as input and output, loading the executable code into the system etc.. The target architecture is a distributed memory multiprocessor system. Each processor has its own private memory, there is no shared memory, thus the only form of communication is via message passing.

**Specification of VM-based Application**

```plaintext
begin
    Virtual_Machine
        type = PFVM;
    Interface
        input_stream = stdin;
        output_stream = stdout;
    Configuration
        topology = BTREE;
        total_tasks = 100;
        total_processors = 16;
    Programs
        data_generator = file1.c;
        comp_function = file2.c;
        result_receive = file3.c;
end
```

Figure 3.5: Specification of an Example Program

An example of how a program can be specified is shown in Figure 3.5. There are five parts to the specification. The `Virtual_Machine` specifies the type of VM, which is the name of the virtual machine. The `Interface` describes the input stream and the output stream of the program. The `Configuration` part gives the parameters that
define the size of the problem and the virtual machine. These parameters affect the performance of the system. Depending on the type of VM different topologies are possible, for example, a chain or a binary tree are possible topologies for the processor farm virtual machine (PFVM). The total_tasks is the total number of tasks in the system. Together with the size of the tasks (i.e., granularity), it gives the total amount of work. The total_processors defines the number of processors in the system and in the Programs part, the names of the files containing the user defined functions are specified.

Note that a single virtual machine may be too restricted for many problems. Thus a practical programming environment based on virtual machines, as shown in the Figure 3.4, will generally have more than one VM. One extra layer, a VM Metalanguage, is added to the overall system structure between the application and the virtual machine. It provides a metalanguage for the programmer to “glue” the virtual machines into a single system which models the problem. Accordingly, facilities for interactions between virtual machines must also be present. In terms of debugging a multi-VM application, one possible approach is to first debug the individual VMs and then combine them together. The communication between VMs may introduce bugs in the final system, however, this problem is beyond the scope of this thesis.

A VM-based programming environment makes it easier to program a parallel system while maximizing performance. As previously mentioned, it also promotes software reuse. For details of these issues, interested readers are referred to Sreekantawamy [Sre93].

### 3.2 Processor Farm Virtual Machine

Many scientific and engineering applications require repeated executions of the same program with different initial data (tasks) [Sre93]. In addition, the processing of tasks
is independent from each other and there is no interaction in the execution of these
tasks. However, the execution paths may vary from one task to another even though the
processors execute the same program.

These applications fit into a manager/worker computational model, where a manager
process generates tasks for a set of worker processes, without specifying which process
should perform the computation for a particular task. The manager is responsible for
generating the tasks, distributing tasks to workers, and then collecting the results from
the workers.

An ideal architecture for this kind of application is a Processor Farm Virtual Ma-
chine(PFVM) which consists of a manager processor and a “farm” of worker processors.
The manager processor reads the initial data from the input stream, generates the tasks,
collects the results and outputs results to the output stream. Each worker processor is a task processing unit. They receive tasks from the manager, execute the tasks, and return the results. Each worker processor executes the same program (with a data dependent execution path) but works on different data.

3.2.1 PFVM Interface

Conceptually, a user can assume that the PFVM has as many worker processors as there are tasks. This conceptual model of programming a PFVM is shown in Figure 3.6. To use the PFVM, the user needs to supply the definitions of three sequential functions for task generation (\texttt{data-generator()}) result receiving (\texttt{result-receiver()}) and task computation (\texttt{compute-fn()}). The first function, \texttt{data-generator()}, executes on the manager and is used to decompose the problem into a sufficient number of tasks. After generating a task it calls \texttt{do-task()} to send the task to the worker processors. The second function \texttt{compute-fn()} is the user defined function invoked by the worker on each of the tasks. On each worker, \texttt{compute-fn()} receives tasks from the manager and, after computing the function, returns the result. There is also another user function, \texttt{result-receiver()}, that runs on the manager node and receives results from the workers. In the implementation of TrEK [Sre93] there is another function, \texttt{init-master()}, which performs some initialization and can be considered to be a part of the \texttt{data-generator()} function.

User programs are linked with the VM code to produce an executable which is then loaded onto the processors.
3.2.2 Intermediate PFVM

The target architecture used by PFVM can be any arbitrary tree-connected network of processors. A chain and a binary tree are two common architectures. Their corresponding Intermediate Processor Farm Virtual Machines (IPFVM) are shown in Figure 3.7 and 3.8, where the nodes represent processors, the edges the communication links, and the arrows the direction of task flow.

Conceptually, the execution of the application on such a virtual machine can be viewed as a flow of task through the system. For example, in the IPFVM using a chain topology, in order for a task to be executed successfully it must first be passed from the manager processor to worker 0. After the task reaches worker 0, it can either be executed there or passed on to the next worker processor in the chain, etc. After a task has been executed, its result is passed back along the chain until it reaches the manager processor. From a message-passing viewpoint, the behaviour of such a system is completely defined by the sending and receiving of tasks between neighboring processors. The result of a task must bear the same identity as the task itself, thus it is not necessary to distinguish between the task and its result.
3.3 Divide and Conquer Virtual Machine

The Divide and Conquer Virtual Machine comes from the well-known problem solving technique, Divide and Conquer. This technique is widely used in a variety of areas including graph theory, matrix computation, etc.. In divide and conquer, a problem is recursively divided into several subproblems until they are small enough whereupon each
subproblem is solved directly. Subproblems are independent from each other. After all the subproblems have been solved, the final solution is obtained by recursively combining the results in the reverse order.

The Divide and Conquer Virtual Machine (DCVM) is the ideal architecture for such applications. As in the PFVM, it also consists of a manager and several workers. The manager reads the initial data from the input stream and generates the tasks in the same manner as the PFVM manager does. However, a worker not only forwards tasks but also splits the tasks along the way. Each task is split recursively to a certain level before being processed by a worker. Once the results are obtained, they are combined and forwarded to the result receiver. Figure 3.9 shows a DCVM, in which each task is recursively split twice before being processed.

3.3.1 DCVM Interface

To use the DCVM, the user must define five sequential functions, a task generator (data-generator()), a result receiver (result-receiver()) and a task executor (compute-fn()). Two additional functions are also required, a task splitter (split-fn()) and a task joiner (join-fn()).

In programming the DCVM, the user defines the number of subtasks into which a task is split and the maximum number of splits. Note that the actual number of splits is decided dynamically by the system and will not exceed the maximum number. In general the number of splits depends on the topology of the virtual machine. Conceptually, the DCVM may be of any degree or depth.
Figure 3.9: Divide and Conquer Virtual Machine
3.3.2 Intermediate DCVM

As in PFVM, the target architectures used for DCVM may be any tree-connected network, or spanning tree of a connected network of processors. The Intermediate Divide and Conquer Virtual Machine (IDCVM) with the topology of a binary tree is shown in Figure 3.8. In contrast to the PFVM, every non-leaf worker processor has, in addition to the computation process, a splitting process and a joining process. The flow of tasks in the system is different from the PFVM, a task is no longer passed from one worker to the next but rather subtasks proceed from level to level in the tree. When results are received, a node returns them back to its parent.

3.4 Debugging VM-based systems

Since applications built using a virtual machine consist of several sequential functions, once again it is natural to debug these functions in a sequential debugging environment. However, there may remain errors that can not be found until it runs in the target distributed environment. This is especially true for large systems in which it may be too time-consuming to thoroughly test each function. There are also other errors that are environment-dependent. Memory errors, for example, will depend on the amount of available memory. In addition, there remains the problem of non-determinism. For example, the actual ordering of inputs of the user defined function result-receiver() in the PFVM is not known until the application runs in a distributed environment. This order may change from run to run. In conclusion, it is necessary to debug the application as a distributed program.
3.4.1 Classification of Errors

In a VM-based programming environment, errors may appear in the following locations: the application (the user code), the VM implementation, the operating system under the VM, or the hardware system as in the case of an incorrect configuration of the communication links. In this section, the types of errors possible in the system are investigated and classified.

In a distributed system based solely on message-passing, errors can be divided into the following three classes: errors in expressing parallelism, errors in the message passing and errors inside a process. These errors are discussed below. Note that common symptoms, such as deadlock, are not listed as an error because deadlock may be due to a variety of errors such as a message omitted (an error in message passing) or the untimely death of a process (an error in expressing parallelism).

Errors in message-passing. This class of errors is detected when the message-passing behaviour between processes, particularly between processes in different processors, is incorrect. Errors that are due to incorrect message passing include:

- omitted messages. When an expected message does not arrive, the program often deadlocks.

- unanticipated messages. When two processes communicate, it is possible that one process sends more messages than the receiving process expects. From the receiver’s point of view, this is an error of unanticipated messages. This error may or may not cause a system failure depending on the message-passing mechanism and the receiver’s execution path. There are a number of possibilities. (a) If the message is a blocking send, the sender blocks forever since there is not a corresponding
receive call. (b) If the message is a nonblocking send, and this continues to occur, messages eventually overflow the buffers in the destination processor, again leading to deadlock. (c) If the receiver is expecting more messages after the unanticipated message, it may treat this message as the expected one and incorrectly process the data. (d) If the message is a nonblocking send and the destination buffers do not overflow, the system may continue to run and finish successfully. In this case, the error is some un-received messages left in the system.

- messages received in an unexpected order. A series of messages arriving in an unexpected order may cause the failure of the receiver of these messages. The symptoms are similar to the case of unanticipated messages.

**Errors in expressing parallelism.** This class of errors appears at the level of processes, detectable by observing an individual process as a black box. From the programmer’s point of view, these errors are incorrect process behaviour, such as too many or too few processes created. The examples of errors in this class are:

- a process created when it should not have been,
- a process waiting for the completion of an unscheduled process,
- a process waiting for the completion of another process that is already guaranteed to have completed, and
- untimely process death.

**Errors inside a process.** Different from the above two classes of errors, this class of errors is common in both sequential programs and distributed programs. These errors
can only be observed by examining the internal state of a process, such as the value of a variable. In a distributed program, these errors can manifest themselves as errors of the previous classes. A common error in this class, which is also difficult to debug, is the misuse of pointers. Other errors in this class include:

- inconsistency in common data's declarations,

- variables not initialized properly, and

- misuse of variables, including pointers.

### 3.4.2 Discussion

In general, the faults that directly result in the first two classes of errors are algorithmic faults. They are often the faults in algorithms, such as an incorrect synchronization between processes. The faults that result in the third class of errors are usage faults. Assuming that the underlying VM system is correct, the only faults in a VM-based application are usage faults. However, as mentioned, these usage faults may exhibit themselves as errors in the first two classes. Both PFVM and DCVM have a predictable message passing pattern. Tasks always start at the manager, are forwarded to one or more workers where they are computed and their results returned along the same route. This pattern can be used to debug programs that use the PFVM or DCVM.
Chapter 4

A Two-phase Approach to Debugging VM-based Applications

Based on the analysis in the last chapter, we propose a two-phase approach to debugging a VM-based system. In the first phase, error identification and localization are automated at the conceptual level using fault models. The results from this phase are used to control the partial replay of the system in the second phase. In Section 4.1, we describe our two-phase approach. In Section 4.2, we define several abstractions needed to define the fault models. In Section 4.3, we define the fault models for programs that use the PFVM and DCVM. In the construction of these fault models it is assumed that the VM system is correct. Finally in Section 4.4, a summary is presented.

4.1 Two-phase Debugging Approach

As discussed in the previous chapter, PFVM and DCVM differ only in that DCVM splits the task before computation and combines results whereas in PFVM the tasks can only be forwarded without being split. Thus when observed at the level of interprocessor communication, PFVM may be viewed as a degenerate case of DCVM where a task is
split into only one task, the task itself. Therefore, in the following discussion we will not
distinguish between these two virtual machines. In the case of PFVM we assume that
there are pseudo splitting and joining processes.

In programming a virtual machine, the programmer can assume as many processors
as required. A task can be split any number of times before being computed, and in each
split, the number of sub-tasks can be arbitrary. This defines a conceptual model of the
distributed program which corresponds to the virtual machine layer of the environment.
In this model, the logical computation of each task can be represented by a task graph in
which each node represents one of the following processes: the generator, the splitter, the
joiner, the executor (computational process) or the post-processor (process that collects
and computes the results). The arcs between these nodes represent the communication
between them. A formal definition is given in Section 4.3.

These processes still have to be mapped onto the target architecture to be executed.
The execution of these processes can be viewed as a flow of tasks into the architecture.
This flow defines an execution model of the distributed program which corresponds to the
IVM layer of the environment. In this model each node is a processor. Each processor
attempts to grab tasks from its parent. The processor keeps the task if it is currently
idle (not executing a task), otherwise it splits the task into a number of subtasks and
puts them into the output queue to be distributed to its children processors. Thus, at
any given moment there is only one task being processed on a node.

By analyzing the task flow behaviour of the system at this level, we would like to map
the execution model back to the conceptual model of the programmer. By narrowing the
errors down to the process executing a particular task, the programmer can debug the
application in a sequential environment. This is consistent with the objectives of VM-based programming – to hide the details of distributed computing while providing the performance of a distributed (parallel) system. Based on this methodology we propose the following two-phase approach to debugging.

The first phase – Automatic monitoring and trace analysis

In this phase, the message passing history is traced and analyzed to locate possible errors as an initial guide for the programmer to debug in the second phase. As mentioned in the last chapter, we target those errors that can be identified by the message-passing pattern. As we collect only message passing events, it is not possible to actually pinpoint the location of the error at this abstraction level. Instead, the errors are identified in terms of its associated process, task and processor. This information is then used in the second phase to locate the errors in the source code.

The second phase – Sequential debugging

In this phase, a simulation execution environment is provided so that the programmer can replay those parts of the program identified in the first phase. A sequential debugging environment is used to pinpoint the location of errors, verify and correct the faults.

As a result of the analysis in the first phase the problem has been reduced to one of debugging a sequential segment of code. As the second phase reduces to debugging sequential programs we have concentrated on the first phase. In the following sections, we show that the first phase of debugging can be automated by building the appropriate fault models for the VMs.
4.2 Definitions

In this section, we define the abstractions needed to build the fault models. For simplicity, a processor in the execution model is called a node and the master processor is always assumed to be the first node in the topology.

- An event, \( e \), is the tuple \((sid, type, did, tid, ptime, ltime)\), where
  - \( sid \) is the node on which the event is initiated.
  - \( type \) is either send or recv, which represents an abstract task send or task receive. Note that it does not directly correspond to a message-passing call in the program.
  - \( did \) is the destination node when the \( type \) is send, or the source node when the \( type \) is recv.
  - \( tid \) is the task associated with this event.
  - \( ptime \) is the physical time stamp as recorded during the tracing. It linearly orders the events with the same \( sid \).
  - \( ltime \) is the vector logical time stamp, a temporal partial ordering that records the “happen-before” order relation.

- A message-passing history, \( H \), is a set of events.

- The event set of a task \( tid \), \( E \), is the set of all events in \( H \) which have the same \( tid \) and the following two pseudo events:
  - A source event, \( e_s \), which is defined as the tuple \((FirstNode, recv, FirstNode, tid, PtimeStart, LtimeStart)\), where \( FirstNode \) is the identity of the first
node in the VM topology, i.e., the node where the task generation part of the application runs. For example, the FirstNode of a binary tree DCVM is the root of the tree of processors. PtimeStart is an assigned physical time stamp which is smaller than the physical time stamp of all other events on FirstNode. Similarly, LtimeStart is a logical time stamp assigned so that this event happens before all the other events in this set. The source event is assumed to have been generated by the data generating process.

- A sink event, ek, which is defined as the tuple (FirstNode, send, FirstNode, tid, PtimeEnd, LtimeEnd), where FirstNode is as defined in the source event. PtimeEnd is an assigned physical time stamp which is larger than the physical time stamp of all other events on FirstNode. Similarly, LtimeEnd is a logical time stamp assigned so that this event happens after all the other events in this set. The sink event is assumed to be generated by the result receiving process after the result of the task is received.

- An event graph of a task, G(V, E), is a directed S-T (single source and single sink) graph where V is the event set of the task and E is a set of arcs (a, b). There is an arc (a, b) ∈ E wherever a, b ∈ V (a ≠ b) and b receives a message from a, or a and b are initiated from the same process and a happened before b. Thus, an event graph represents the causality relations between all the events related to a single task.

- A coalesced event graph of a task, G(V, E), is a collapsed event graph obtained by combining all the vertices initiated by the same process into a single vertex. A vertex is considered completely coalesced if it coalesces the correct number of events. Otherwise, it is called an incompletely coalesced vertex. The correct number of events for the data generator process is two. First is the source event and second
is the send event that passes the task to the next processor. Similarly, the number of events for the post-processor process is also two, first is the recv event which receives the result of this task and second is the sink event. For the split process, the correct number of events is one more than the degree of the task graph. Of them, one event receives the task, and other events send the sub-tasks to the next processors. Similarly, the correct number of events for the join process is also one more than the degree of the task graph.

- Of all events in the event set of a task, those associated with the vertices which are reachable from the source event and do not have any outgoing arcs are called frontier events. The frontier node of a frontier event is the node sid when the event is a recv, or the node did when the event is a send. The frontier node is where the data flow of this task stops. The set of frontier events of an event graph is called the frontier set of the event graph.

### 4.3 Fault Models

As shown in the first section, the architecture and the task graph are the most essential parts in the specifications of a VM-based application. In the following, we give their formal definitions.

An architecture, $A(V_a, E_a)$, is a directed graph where $V_a$ is a set of processors and $E_a$ is the set of arcs such that $(n_1, n_2) \in E_a$ whenever $n_1, n_2 \in V_a(n_1 \neq n_2)$ and there is a communication link from $n_1$ to $n_2$.

A task graph, $T(V_t, E_t)$, is a directed graph where $V_t$ is the set of processes of computing this task and $E_t$ is the set of arcs such that $(p_1, p_2) \in E_t$ whenever $p_1, p_2 \in V_t(p_1 \neq p_2)$ and there is message passing from $p_1$ to $p_2$. An arc $(p_1, p_2)$ is labeled “Distributing” if
the message passed is a task, or "Collecting" if the message passed is a result.

A *task computation graph*, $C(V_c, E_c)$, is the directed graph obtained by combining an architecture $A(V_a, E_a)$ and a task graph $T(V_t, E_t)$. $V_c$ is the set of vertices such that $V_c \subseteq V_a \times V_t$ and $E_c$ is the set of arcs such that $((n_1, p_1), (n_2, p_2)) \in E_c$ if $(n_1, p_1), (n_2, p_2) \in V_c$, $(n_1, p_1) \neq (n_2, p_2)$ and $(n_1, n_2) \in E_a$ and $(p_1, p_2) \in E_t$.

Given an architecture $A$ and a task graph $T$, the family of task computation graphs, $C$, is the set of all correct task computation graphs.

Given an event graph and a task computation graph, the event graph is said to *map* onto the task computation graph if there is a one-to-one correspondence between its coalesced event graph and the task computation graph. If there is also a one-to-one correspondence from the task computation graph to the coalesced event graph, the event graph is said to *match* the task computation graph.

The specifications of a VM-based application define a family of task computation graphs. Given a message history, as defined in the last section, every event graph will map onto a computation graph. Based on whether or not the event graph matches some computation graph, we have the following fault models.

- **Success.** Given the family of task computation graphs $C$ defined by the specification of a VM-based application and an event history $H$ which defines a set of coalesced event graphs $E$, the event history is considered a success if, for every event graph $E_t \in E$ corresponding to some event $t$, there exists a $C \in C$ such that $C$ is a match of $E_t$. That is, the message passing behaviour of the system is as expected. Note that this can only detect errors that are exhibited in the message passing. It is up to the programmer to detect the other types of errors. Once the task number is identified, by comparing the result of the tasks with the expected, the user can
proceed to the second phase, where its event graph is used to control the replay.

- **Failure.** Given the family of task computation graphs, $C$, defined by the specification of a VM-based application and an event history, $H$, which defines a set of coalesced event graphs $E$, it is considered a failure if, for every event graph $E_t \in E$ corresponding to some event $t$, there does not exist $C \in C$ such that $C$ is a match of $E_t$. In other words, the message passing behaviour of the system does not match a correct execution because the computation of some task failed to complete.

For each such coalesced event graph, there are $m$ errors in computing this task, where $m$ is the number of frontier events of $E$. For every frontier event in these event graphs, there is a fatal error located at $(tid, pid, nid)$ where $tid$ is the task number. If the associated vertex of this frontier event is not a completely coalesced vertex, then $nid$ is the node associated with the vertex of the coalesced event graph and $pid$ is the process associated with the vertex. Otherwise, $nid$ is the node associated with the destination vertex of the outgoing arc of this vertex in the task computation graph, and $pid$ is the label of the outgoing arcs.

The event graph of a task represents the causality relation between events associated with this task. Each frontier event in the graph represents a location where the computation of a certain sub-task fails to complete, that is, an error. The location of this error is the location of the next expected event.

### 4.4 Summary

By analyzing the message-passing behaviour of a VM-based program, we have developed fault models so that the first phase of debugging can be automated. As we have assumed
that the underlying VM implementation, the operating system and the hardware are all correct, the fault model is able to detect errors in the application that manifest themselves as errors in the message passing behaviour. Errors that do not cause any deviation from the message-passing behaviour but return a erroneous result can not be detected by this technique. Once the task in which an error occurred has been identified the programmer can use the second phase to sequentially debug the code.
Chapter 5

Implementation

In this chapter, we describe the design and implementation of the first phase of the debugging method proposed in the last chapter. Section 5.1 gives an overview of the system. Section 5.2 describes the event tracing and discusses some of the problems in collecting traces for the purpose of debugging. Section 5.3 describes the implementation of the fault model and the automation of the error identification and localization. Finally, in the last section we discuss the implementation, including event tracing and time stamping of events.

5.1 Overall Design of the System

Figure 5.10 depicts the overall structure of the system. The implementation of virtual machines in this case, TrEK (Tree Execution Kernel) must first be instrumented by inserting special probes. It is then compiled and linked with the application and the resulting executable is downloaded onto the transputers. As it executes events are generated. A monitor Tmon-de collects these events into an event trace file. The trace analyzer, deTrEK, takes the event trace as input and produces a fault list. The second phase debugger has not been implemented. It would take as input the user code, TrEK,
the fault list and the event trace and sequentially simulate the VMs.

The major parts of the implementation were Tmon-de, a debugging version of Tmon, the trace analyzer, deTrEK, and the instrumentation of TrEK.
5.2 Instrumentation

As described in Chapter 3, TrEK consists of several processes on each processor. To monitor the task flow between processors, special probes were inserted into the processes that manage the communication channels. In TrEK, the passing of a task or the result of a task is done by passing two messages, the size of the data and the data itself. A probe was inserted after these two communication calls. The probe generated an event, as defined in Chapter 4, which was then assigned a physical time stamp by the monitor.

As the result of this instrumentation, the event tracing is at a level that more closely corresponds to the virtual machine. The resulting events are a set of abstract events, each of which is a send or receive of a task (or its result), instead of a message. Tracing abstract events reduces the overhead of event tracing and the interference to the programs being monitored. It also simplifies the filtering of the trace and further processing needed for the analysis.

5.3 Event Tracing

Event tracing is an important part of any post-mortem debugging method. The accuracy of the results depends on the completeness and accuracy of the tracing mechanism. The objective of our design of the monitor was to obtain high accuracy and completeness while minimizing the probe effect and the overhead in generating the events, collecting them and downloading them to the host.
5.3.1 Environment

In this subsection, we give a brief description of the underlying instrumentation environment, including the hardware architecture, the operating system and TrEK.

Multicomputer Architecture

The target hardware architecture for our debugging system is a 75-node transputer-based system in the Department of Computer Science at UBC. The system consists of a Sun 4 workstation as the host and 75 processors. On each processor, or node, there is an IMS T800 transputer with 4 Kbytes of on-chip RAM, 4 bidirectional serial links and 1 to 16 Mbytes of local memory. The 75 nodes are interconnected through 10 programmable crossbar switches and the system is connected to the host by a Sbus interface. There are currently four connections between the host and the transputers and two connections between the host and the crossbar switches, which are used to control the configuration of the crossbars, and thus the topology. Nodes that are not directly connected to the host can only communicate with the host through intermediate nodes. The interconnection topology of the nodes is statically reconfigurable by a process on the host that sends switch setting commands to the crossbar switches.

Trollius Operating System

The multiprocessor toolkit in our implementation is the Trollius\(^1\) Operating System [Bur88]. It is a parallel operating system designed for distributed memory multicomputers, developed jointly at Ohio State Supercomputing Centre and Cornell Theory Centre.

\(^{1}\)Trollius is a trademark of the Ohio State University and the Cornell Research Foundation
Trollius provides a uniform programming environment that extends from the parallel processing units to the host. It enables the programmer to utilize the program development power of the UNIX\(^2\) operating system on the front end and the computational power of the parallel machine on the back end. Trollius provides a message passing facility between the processes on different nodes, as well as inside the same node. There are two levels of message-passing in Trollius. One is for intra-processor communication at the kernel level. It provides a blocking message passing service to processes on the same node. The other is for inter-processor communication at the network level. It loosely resembles the OSI network model and provides datalink, network and transport layer services. The synchronization between the message sender and receiver is via arbitrary numbers called events, instead of the process identifier. Other features of Trollius include multitasking, access to host’s remote file system and process loading. These services are provided as a C library for process creation, process destruction, signal handling and access to the remote file system. Many of these services are also available as command programs which are executed from the UNIX shell. For a detailed description of Trollius readers are referred to [Bur88] [Tro92b] [Tro92a].

The Tree Execution Kernel

The virtual machine implementation in our VM-based programming environment is a Tree Execution Kernel (TrEK) [Sre93]. It is a runtime kernel that controls and coordinates the activities of a virtual tree of processors. The kernel on each node is responsible for accepting tasks from its parent processor. Upon receiving a task, it may, depending on the load, execute the task locally or pass it on to one of its children processors. The

\(^2\)UNIX is a trademark of AT&T
kernel is also responsible for returning the results to its parent, and eventually to the root processor that generated the tasks. Independent of the underlying topology of the target architecture, the kernel provides a virtual tree machine with the topology specified by the programmer.

The task distribution strategy in TrEK is a scheme called flood-filling. In this scheme, a child processor greedily tries to grab tasks from its parent. A child processor keeps the task if it is currently idle (not executing a task), otherwise it splits the task into a certain number of subtasks and adds the subtasks into the output queue to be distributed to the children processors. The number of subtasks split from a task is controlled by the user program. Results are returned to its parent processor as soon as they become available. That is, there is no guarantee that the results will be returned in the order of the distribution of tasks. The whole system is not a FIFO queue.

In terms of implementation, TrEK is organized as a set of communicating processes. On a typical node, there are 8 high priority processes that manage the 4 input and 4 output communication links and are responsible for task/result receiving and forwarding. In addition, there are 4 other processes: a data manager, a result manager, a split process and a join process. Briefly, the data manager is responsible for making task scheduling decisions, the result manager gets the result from the user-defined computation process and passes it to the parent processor. The split process and the join process invoke the user split and join function respectively as required.

TrEK can support both the processor farm and divide-and-conquer virtual machine. The processor farm can be viewed as a degenerate case of divide and conquer, where each task is only “split” into one subtask, the task itself. Thus task splitting and result joining is unnecessary. A small change to configuration file allows TrEK to be configured
for either PFVM or DCVM. For a detailed description of the TrEK, readers are referred to [Sre93].

5.3.2 System Structure of Tmon

Tmon[WJC93] is a parallel monitoring system implemented on the transputer-based multicomputer for the purpose of performance analysis and tuning. It uses a global interrupt mechanism for clock synchronization, to get relatively accurate timing and global state information. To minimize the interference to application communication and performance degradation because of the monitoring, an adaptive reporting mechanism was developed to unload the trace data to the host when the communication and computation load of the local processor is detected to be under a predefined limit.

There are three major components in the system: data generation and collection, global control and data analysis, and display. One transputer in the network is distinguished as the master node for global control. All the nodes being monitored are treated as slave nodes. The application runs on these slave nodes where the operations of data generation and collection are performed.

The master node is capable of simultaneously interrupting all the slave nodes in the system to perform clock synchronization and for generation of sampling events regarding node usage. There are two processes on the master node: an interface process and a controller process. The interface process accepts monitoring commands from the user to start or stop a monitoring session. The controller periodically generates the global interrupt signal to synchronize the activities of the slaves.

On each slave node, there are several processes that perform the data generation and collection of events. Event probes are inserted into the application to generate trace
data. As they are generated, a *meter* process collects these event traces and puts them into the trace buffers. A *buffer manager* periodically checks the status of the buffers and processors, and flushes the buffers when they are full. There are two trace data buffers, organized as a double buffering system, allowing the *meter* process to fill one buffer while another is being flushed by the buffer manager. The trace data is sent to the host using the global message passing services provided by the underlying operating system, Trollius. A *backend* process accepts the global interrupts for synchronization, performs clock synchronization and generates sampling events of node usage.

A *collector* process on the host runs as a daemon. It collects trace data from all of the slave nodes and sends the data to the *data display*, which can display the performance results graphically in real time. It also dumps the trace data to trace files for further *data analysis*.

There are basically three types of events being monitored in Tmon. The first type of events are the node usage events which report the status of the communication links and the CPU. The second type are the standard events of message passing and process creation and destruction, including message send, message receive, calling of a message receive, the initiation of a process and the destruction of a process. The third type of events are the user-defined events. A simple command line user interface is provided to enable/disable monitoring. Two functions are also provided in the library so that the monitoring can be enabled/disabled from the user program.
5.3.3 Modification to Tmon

Performance Analysis vs. Debugging

There are some similarities and differences between monitoring for performance analysis and monitoring for debugging. Generally, the objective of monitoring is to understand the execution behaviour of the program. Both types of monitoring need to dynamically extract information about the execution of the program. And both attempt to minimize the interference to the program being monitored and the overhead of monitoring, which include the following:

- CPU time for event generation in the program,
- CPU time of running the monitoring software, i.e., is to collect the event trace,
- communication cost for transferring the trace data,
- memory for storing the trace data.

However, whereas performance analysis is concerned with the efficiency of the program, debugging is concerned with its correctness. As a result they require different information. The issues they both address have different priorities. In debugging, the probe effect is more critical than the overhead introduced. As long as the overhead is within an acceptable range, first priority is to minimize the probe effect. That is, the deviation of execution behaviour because of the monitoring, in terms of correctness, should be minimized. Also, performance monitoring assumes algorithmic correctness of the system while the monitoring for debugging is concerned with the anomalous termination of the program being monitored.
Modification

As seen in the last section, Tmon is designed for performance analysis. To modify it for debugging monitoring, we have made a number of changes.

- The global interrupt for clock synchronization has been disabled due to unavailability of the hardware hooks. Thus there is no master node which enforces global control. Instead, the control is now distributed on the slave nodes. The monitoring on a slave is turned on and off by a local interface process, which accepts the control commands from the host. As the control commands are sent via the message passing mechanism provided by Trollius, the slave nodes are not stopped simultaneously. If the monitoring is turned off on a slave node, the execution of the application will be suspended when it tries to generate the next monitoring event. The execution of the application is resumed as soon as the monitoring is turned back on. In this way, turning on/off the monitoring indirectly controls the execution of the program.

Hardware clock synchronization is not used. The clocks are reset when the monitoring is turned on, after that, the clocks in each processor progress according to their own speed. Thus clock drift is inevitable. Fortunately, we do not depend on the physical time stamps in our debugging method. As long as the events that occur on the same node keep a linear order, clock drift is not important.

- The unloading of trace data is controlled by a sequence of commands that is specified statically. Each command is in the format

\[
\langle \text{Time} \rangle \ \langle \text{LowerBound} \rangle \ \langle \text{UpperBound} \rangle
\]
The command specifies that the trace data must be unloaded immediately to the host whenever there are more than \(<\text{UpperBound}\) number of events in the buffer. If the number of events in the buffer is less than \(<\text{UpperBound}\) but greater than \(<\text{LowerBound}\), and the CPU and communication links are not busy, the trace is unloaded. These two bounds are in effect until the clock reaches time \(<\text{Time}\), at which time it steps to the next command. Of course, these bounds will be overwritten with the buffer size if it is smaller.

These modifications to the monitor have kept the adaptive reporting of Tmon, while providing the facility to control the unloading of trace data in order to manage the loss. If this facility had not been implemented, up to 256 events could be lost when a node with two full buffers crashes. This loss of information would have made the trace analysis far less accurate.

- A number of functions have been added to the trace collector on the host. To begin with, all the events are recorded into a file by the trace collector instead of filtering the events into two different files. Afterwards, a program filters the data into the different files needed for performance analysis, visualization and debugging. As does the original Tmon, node usage events are recorded into one file and the standard events of Tmon into another. The format of those events is unchanged and all the related tools for analyzing that data can be applied without any change. The user-defined events (probes) are written into a separate file. Also, the configuration file in Tmon is no longer generated by the slave nodes. Instead, it is generated by hostmon, which is invoked with the nodes explicitly specified on the command line. In addition, a number of statistics are shown when the monitoring session ends. This includes the total number of events received and the time stamps of the last
event. This information tells the programmer how far the program executed before failure. The time stamps can also be used to modify the shipping control sequences for the next monitoring session.

- A simple mechanism for termination is provided. If the program executes successfully, the monitoring session ends when the process “hostmon” receives an “end” signal from each node being monitored. In the case of a node crash, the monitoring session can be aborted by sending a “quit” command from the monitoring controller on the command line. Then, depending on the extent of the crash, all remaining monitored data is flushed to the host.

5.4 Trace Analysis

5.4.1 Algorithm for Logical Time Stamping

In event tracing, events on the same processor are linearly ordered. However, events occurring on different processors are not. There does, however, exist a partial order between them. We implemented the vector logical clocks of Fidge [Fid91] for keeping a partial order relation between events occurring on different processors.

Based on the dynamic algorithm of Fidge [Fid91], the following algorithm generates vector logical time stamps for the events after all the events have been recorded. It is a centralized, post-mortem algorithm even though it can also work dynamically. The algorithm assumes asynchronous communication where the send is buffered and non-blocking and the receive is non-buffered and blocking (it also works with synchronous communication). The algorithm is based on the replay of the message-passing events. On each processor, there is a queue which stores future events in order and a list which
Chapter 5. Implementation

contains the send events to be matched. The first event in the queue is the event that is pending. The list acts a buffer for the incoming messages.

Input. The number of processors \( N \) and a file that contains all the abstract events linearly ordered for the same \( sid \). Each event has a type which is either a send or a recv, and an initiating processor identity.

Output. A list of logically time stamped abstract events.

Method. Suppose that the processor ID's are numbered from 0 to \( N-1 \). Let \( S \) be a stack of processor IDs, \( Cltime \) be an array of current logical time stamps, one for each processor, \( Q \) be an array of queues for pending events to be time stamped and \( L \) be an array of lists for send events which have been time stamped but remain to be matched with their corresponding recv events. The algorithm is shown below.

```c
/* initialization */
for (i=0; i<N; i++) {
    init_queue(Q[i]);
    init_list(L[i]);
}
init_stack(S);

while (read_event(E)) {
    insert_queue(Q[E.sid], E);
    push_stack(S, E.sid);
    while (pop_stack(S, P)) {
        if (find_queue(Q[P], E1)) {
            if (E1 is a send) {
```
Cltime[P][P] += D;
E1.1time = Cltime[P];
insert_list(L[El.did], E1);
push_stack(S, El.did);
push_stack(S, P);
output_event(E1);
delete_queue(Q[P]);
}
else if (match_search_list(L[P], E1, E2)) {
    Cltime[P][P] += D;
    Cltime[P] = maximum(E2.1time, Cltime[P]);
    E1.1time = Cltime[P];
push_stack(S, P);
output_event(E2);
delete_queue(Q[P]);
delete_list(L[P], E2);
5.4.2 System Design

The trace analyzer deTrEK takes the event trace file produced by the monitor as its input and outputs an error list whenever one or more errors have occurred in the message passing behaviour. As shown in Figure 5.11, the design of deTrEK consists of eight modules, approximately in a hierarchy of three layers. The top layer, Main, is the main control module. The next layer consists of four modules that implement the major algorithms. The pre-processing module performs the conversions of task identities and the logical time stamping. The Event Graph Construction constructs the event graphs for each task. The Event Graph Validation module validates the graphs before passing them to the Error Identification and Localization module, which identifies and localizes the errors exhibited in the event graphs. Three modules on the bottom, Queue, List and Input and Output provide the tool set for the other modules. Of them, Queue and List are abstract data types, an extended queue and a list. The Input and Output module performs the I/O functions. In our implementation, we did not generate the task computation graphs according to the specification implied by the fault models. Generating that large number of task computation graphs is inefficient. Instead, the event graph is directly checked against the properties of the task computation graphs.

The program begins with the pre-processing of the trace events. It includes sorting the events so that the events initiated from the same node appear in the order in which they occurred. This is required to ensure the correctness of the logical time stamping. It also converts the task IDs of the trace events into a format that is suitable for trace analysis. The events are then logically time stamped.

After that, the message passing history is augmented by adding the two pseudo events for each task as described in Chapter 4. The event graphs are then constructed. Each
Figure 5.11: System Design of deTREK

Main

- Pre-Processing
- Event Graph Construction
- Event Graph Validation
- Error Identification and Localization

Queue
Input and Output
List
event graph is represented by a structure which contains two queues; one for vertices and another for arcs. Each vertex is represented by a structure which contains a pointer to the associated event. Each arc is represented by a structure which contains pointers to the two associated events, the source and destination vertex. The history is then scanned to build the vertex queues of event graphs. Task generation errors are detected at this time. Finally, the arc queue for each event graph is generated.

The validation of an event graph checks the topology of the graph according to the specification of the application. This operation assures correctness of the tracing mechanism and the application specifications. Without validation the correctness of error identification and localization can not be guaranteed.

In the final step, the frontier events are extracted from each of the event graphs. These events form a list, and for each frontier event on the list, an error is identified and localized based on the position of the event in its event graph and its time relationship with the other frontier events.

5.4.3 User Interface

A simple command line interface to the trace analyzer, called deTrEK, is provided for the programmer. A number of parameters of the application, including the type of VM, the number of processors, the number of tasks and the topology are specified in the command line. A man page for deTrEK is shown below.

- **NAME**
  
  - deTrEK — A tool to analyze a trace file for the purpose of debugging.

- **SYNTAX**
• **DESCRIPTION**

- dcTrEK is a tool for analyzing the message-passing history of applications written for TrEK. It takes as input a history of message passing events and specifications of the application and outputs a list of possible errors and their locations by analyzing the message-passing history for conformance to the specifications.

• **OPTIONS**

- `-h` Prints out a help message showing the options and their usage
- `-v` Turns on the verbose mode, the default mode is off
- `-c` Turns off the conversion of the task identity
- `-p <FilePref>` Specifies the prefix of the two file names. If this option is not specified, the default prefix “data” is used.

The first file is the specifications of the application “<FilePref>.spf”. It is a common text file that includes the following 5 integer tuples representing the specifications of the application:

- `<NumProc>` – number of processors in the target architecture.
- `<Topology>` – type of topology, 1 represents a chain; 2 a binary tree; 3 a ternary tree. The current implementation of this tool only supports these 3 topologies.
- `<NumTasks>` – The number of tasks generated by the `data-generator()`.
- `<TaskDegree>` – Degree of the task graph.
<TaskDepth> – Depth of the task graph.

For example, a DCVM-based application that runs on a binary tree of 16 nodes with 100 tasks where each split generates 4 tasks and the maximum number of splits is 5 will look like:

16 2 100 4 5

The second file is the abstract event history, “<FilePref>.tvm”. Each line of this file is an abstract event of the following format:

<Ptime> <Sid> <Type> <Did> <MainTid> <SubTid>

where <Ptime> is its physical time stamp; <Type> is its type, that is either send(send) or receive(recv); <Sid> is the initiating processor; <Did> is the destination processor for a send event or the source processor for a receive event; <MainTid> is an integer, the major task associated with this event; <SubTid> is a bit sequence in hexadecimal which further identifies the sub-task numbering in the DCVM.

-o Outputs the error list into a file “<FilePref>.bug”, instead of the default standard output. “<FilePref>.bug” is also a text file. Each line of the file reports an identified error in the following format:

<Error> <TaskNo> <ProcNo> <Process>

where <Error> is the type of fault, <TaskNo> is the task associated with that error, <ProcNo> is the processor where the error occurred, <Process> defines the process in which the error occurred, and is one of the generating, splitting, joining, computing and post-processing processes where the error occurred.
Chapter 6

Experiments and Evaluation

6.1 Experiments with Simulation

To test and demonstrate the monitor and trace analyzer, we developed a general simulated application for a task-oriented virtual machine. The application takes as input a task description file and simulates the computation of generating, splitting, joining, computing and post-processing of the tasks. This is useful for testing since the load and error conditions can be easily controlled and varied. Actual debugging of real programs is reported later in this chapter.

A task description file is the definition of the computation of a task. It includes the task number and the definitions of five program blocks in the following format:

\[
\text{<taskno> <gen-seg> <comp-seg> <post-seg> <split-seg> <join-seg>}
\]

where \text{<taskno>} is the number of the tasks, \text{<gen-seg>} is the definition of the simulated computation generating the task, \text{<comp-seg>}, \text{<post-seg>}, \text{<split-seg>} and \text{<join-seg>} are the definitions of the simulated computation doing computing, post-processing, splitting and joining of the tasks, respectively.

Each segment is a tuple of 3 integers shown as follows:
It simulates a segment of the program which occupies \texttt{space} bytes of memory and takes \texttt{time} microseconds to execute. In the simulation, a memory block of that size is allocated and an empty loop is executed an appropriate number of times.

For convenience, blank lines and lines starting with \texttt{#} are treated as comments. The numbering of tasks starts at 0. The total number of tasks in each execution depends on the command line parameters used when the executable code is loaded onto the nodes. By default, wherever their definitions are missing, all the tasks are defined to be the same as the most recently defined task. If the first task defined in the file is not task 0, all the tasks with smaller numbers are assumed to be built-in default tasks.

With this simulated application we can experiment with bugs in different processes and with different task granularities. In the following, we show a typical session of a test run.

1. First, the specification (\texttt{dc.spf}) of the application is constructed. Shown below is the specification of an application. It is intended to run on a binary tree with 8 nodes. There are 100 tasks in this application. Each task can be split into 2 sub-tasks, to a maximum depth of 5.

\begin{verbatim}
8 2 100 2 5
\end{verbatim}

2. Then a task description file (\texttt{dc.dat}) is constructed. It defines the granularity of each tasks and the bugs, if any, with their location. The following describes three tasks 0, 49 and 50. In the computation of task 49 there is a fatal bug (Bug-id 4) which will cause the node to crash. All the other tasks are, by default, the same as the previous one.
3. The transputers are booted, loaded, and monitor is started. The “state” command provided by Trollius is used to check whether the system processes on nodes are still alive. This command reports the status of the Trollius processes and prints out the processes and their execution state. If this command hangs in trying to report the process status on some node, it means that the node is not responding and the system has failed. In our experiment, after a while, this command reported that nodes n3 to n7 have failed. The session is ended by forcing the monitor to quit.

```
xanadu<1>spread -sc bnail.btree_32.02
Ohio Trollius 2.2 - The Ohio State University
soldering...
xanadu<2>loadgo n0-7 nodemon
xanadu<3>hostmon n0-7 -e data &
xanadu<4>loadgo n4-7 slave
xanadu<5>loadgo n1-3 slave -- last
xanadu<6>loadgo n0 master -- 8 100 dc.dat
xanadu<7>monctrl -b n0-7
...
xanadu<8>state n0-7

<table>
<thead>
<tr>
<th>NODE INDEX</th>
<th>PID</th>
<th>KPRI</th>
<th>KSTATE</th>
<th>PROGRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>n0</td>
<td>[18]</td>
<td>297985</td>
<td>BR (-91009, 0)</td>
<td>nodemon</td>
</tr>
<tr>
<td>n0</td>
<td>[19]</td>
<td>366821</td>
<td>A (-366821, 0)</td>
<td>master</td>
</tr>
<tr>
<td>n1</td>
<td>[18]</td>
<td>305141</td>
<td>BR (-91009, 0)</td>
<td>nodemon</td>
</tr>
<tr>
<td>n1</td>
<td>[19]</td>
<td>361913</td>
<td>A (-408885, 0)</td>
<td>slave</td>
</tr>
<tr>
<td>n2</td>
<td>[18]</td>
<td>305141</td>
<td>BR (-91009, 0)</td>
<td>nodemon</td>
</tr>
<tr>
<td>n2</td>
<td>[19]</td>
<td>361913</td>
<td>A (-408885, 0)</td>
<td>slave</td>
</tr>
<tr>
<td>n3</td>
<td>[18]</td>
<td>297985</td>
<td>BR (-91009, 0)</td>
<td>nodemon</td>
</tr>
<tr>
<td>n3</td>
<td>[19]</td>
<td>361937</td>
<td>A (-408909, 0)</td>
<td>slave</td>
</tr>
</tbody>
</table>

Z Suspended
xanadu<9>kill %
xanadu<10>monctrl -q
```

4. After forcing “hostmon” to quit, two files were left in the current directory, “data.config” and “data.all”. Using our filtering tool “tvmfilter”, we obtained the file “data.tvm”, in which the events initiated on the same node are ordered and ready for input to
the trace analyzer, deTrEK, which generated the following.

```
xanadu<11>tvmfilter -p data
xanadu<12>detrek -cp data
---------------------  Failure ---------------------
<table>
<thead>
<tr>
<th>TASK</th>
<th>SUBTASK</th>
<th>NODE</th>
<th>PROCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>0010</td>
<td>6</td>
<td>comp_fn()</td>
</tr>
<tr>
<td>49</td>
<td>0110</td>
<td>7</td>
<td>comp_fn()</td>
</tr>
<tr>
<td>49</td>
<td>0100</td>
<td>5</td>
<td>comp_fn()</td>
</tr>
<tr>
<td>49</td>
<td>0000</td>
<td>4</td>
<td>comp_fn()</td>
</tr>
<tr>
<td>50</td>
<td>0000</td>
<td>1</td>
<td>split_fn()/comp_fn()</td>
</tr>
<tr>
<td>51</td>
<td>0000</td>
<td>2</td>
<td>split_fn()/comp_fn()</td>
</tr>
<tr>
<td>51</td>
<td>0010</td>
<td>3</td>
<td>split_fn()/comp_fn()</td>
</tr>
<tr>
<td>52</td>
<td>0000</td>
<td>1</td>
<td>Distributing</td>
</tr>
<tr>
<td>53</td>
<td>0000</td>
<td>2</td>
<td>Distributing</td>
</tr>
<tr>
<td>53</td>
<td>0010</td>
<td>3</td>
<td>Distributing</td>
</tr>
</tbody>
</table>
...```

The trace analysis shows that task 49 was split into 4 sub-tasks which were executing on nodes 4, 5, 6 and 7 where they failed to complete. Task 50 failed in either the splitting process or the computing process on node 1 and task 51 was split into two sub-tasks which were either being split or computed when the failure occurred. Task 51 and the remaining tasks, which had been either split or not, were being distributed. This information allows us to focus on debugging the computational process of task 49. This process would be executed in a sequential environment when the second phase of the debugger is implemented. The debugging tool will use the information in the event graph to generate and split the task the correct number of times before feeding the sub-tasks to the process.

The output of the trace analyzer for another experiment of 1000 tasks running a binary tree with 16 nodes, in which a fatal error is introduced in task 500, reveals the following information.
Chapter 6. Experiments and Evaluation

Obviously, it did not give us any useful information for localizing the bug introduced. This was due to an improper unloading command sequence used during monitoring. When the system failed, "hostmon" did not receive the updated events from the nodes. Based on the statistics provided by "hostmon", we defined the unloading command sequence so that the trace data on the nodes was flushed more often near the point of failure. The application was rerun and the following output from deTrEK was obtained.

```
<table>
<thead>
<tr>
<th>TASK</th>
<th>SUBTASK</th>
<th>NODE</th>
<th>PROCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>475</td>
<td>0000</td>
<td>1</td>
<td>split_fn()</td>
</tr>
<tr>
<td>476</td>
<td>0010</td>
<td>1</td>
<td>join_fn()</td>
</tr>
<tr>
<td>476</td>
<td>0000</td>
<td>1</td>
<td>join_fn()</td>
</tr>
<tr>
<td>477</td>
<td>0000</td>
<td>1</td>
<td>Distributing</td>
</tr>
<tr>
<td>478</td>
<td>1110</td>
<td>7</td>
<td>join_fn()</td>
</tr>
<tr>
<td>478</td>
<td>0110</td>
<td>7</td>
<td>join_fn()</td>
</tr>
<tr>
<td>478</td>
<td>0010</td>
<td>3</td>
<td>Collecting</td>
</tr>
<tr>
<td>478</td>
<td>0000</td>
<td>1</td>
<td>Collecting</td>
</tr>
<tr>
<td>480</td>
<td>0110</td>
<td>7</td>
<td>Distributing</td>
</tr>
<tr>
<td>480</td>
<td>0010</td>
<td>12</td>
<td>Distributing</td>
</tr>
<tr>
<td>480</td>
<td>1010</td>
<td>13</td>
<td>Distributing</td>
</tr>
<tr>
<td>480</td>
<td>0100</td>
<td>10</td>
<td>Distributing</td>
</tr>
<tr>
<td>480</td>
<td>1100</td>
<td>11</td>
<td>Distributing</td>
</tr>
<tr>
<td>480</td>
<td>0000</td>
<td>8</td>
<td>Distributing</td>
</tr>
<tr>
<td>480</td>
<td>1000</td>
<td>9</td>
<td>Distributing</td>
</tr>
<tr>
<td>481</td>
<td>0000</td>
<td>1</td>
<td>Distributing</td>
</tr>
<tr>
<td>482</td>
<td>0010</td>
<td>3</td>
<td>Distributing</td>
</tr>
<tr>
<td>482</td>
<td>0100</td>
<td>5</td>
<td>Distributing</td>
</tr>
<tr>
<td>482</td>
<td>0000</td>
<td>4</td>
<td>Distributing</td>
</tr>
<tr>
<td>483</td>
<td>0000</td>
<td>1</td>
<td>Distributing</td>
</tr>
<tr>
<td>484</td>
<td>0000</td>
<td>1</td>
<td>Distributing</td>
</tr>
<tr>
<td>485</td>
<td>0000</td>
<td>1</td>
<td>Distributing</td>
</tr>
<tr>
<td>486</td>
<td>0000</td>
<td>1</td>
<td>Distributing</td>
</tr>
<tr>
<td>487</td>
<td>0000</td>
<td>0</td>
<td>data_generator()</td>
</tr>
<tr>
<td>488</td>
<td>0000</td>
<td>0</td>
<td>data_generator()</td>
</tr>
</tbody>
</table>
```

...
The fault was found to be located in the computing process of the subtasks of task 500, on nodes 8 to 15, exactly where we had introduced the bug.

Several other experiments introducing fatal errors revealed similar results. In most cases, deTrEK was able to point out the right process where the bug was introduced. In a few cases, where the runtime system was communication intensive, the information provided by deTrEK was less accurate because of the amount of trace data that was not received. However, as most applications in the parallel computing environment are computationally intensive, this should not be a serious problem. Also the unloading control sequence can be adjusted so that the overhead introduced by the monitor is kept within an acceptable range, and the loss of trace data is minimized.

Experiments also show that there is no significant difference when several fatal errors are simultaneously introduced. The first fatal error that occurred will be detected by deTrEK while the other errors will not appear. This conforms to the usual scenario of debugging where the programmer examines the errors one by one in their order of occurrence.

Note that our simulated application did not allow introduction of bugs into the splitting process of a subtask. If a bug is ever introduced in the splitting process, all the splitting processes will crash while splitting this task or its sub-tasks. In fact only the first splitting process will be executed as the system will crash after that. This is also the case for the computing and the joining processes. However, it is expected that deTrEK
will still be able to pinpoint the right process for a particular sub-task if it crashes, as the other successfully completed sub-tasks will be noted. For example, in the case of an error in the computation of a sub-task, all other sub-tasks will complete and the joining process leading to that sub-task will report failure. This can be tested by introducing a subtask number in the task description file, and accordingly, inside the process. The subtask number is compared with the actual subtask number before simulating the error.

### 6.2 Experience with Debugging an Actual PFVM Application

In this section, we describe our experience in debugging an actual PFVM application using our monitoring and trace analysis tools. The application was a course project done by Jeff Beis for CPSC536.

The application is an image processing program. The input to the program is an image with a number of segments. Each segment represents some part of an object. The program groups the segments belonging to one object together. It begins by loading the entire image on all the slaves. The master then sends out the segment number to be grouped, so in effect each segment number defines a task. When a slave receives a task, it attempts to group that segment with all the other segments in the image and returns the result.

When the program ran for 5 tasks using a chain of 6 nodes, as we did for the simulated problem, we obtained the following:

```bash
 xanadu<2>spread -vsc bnail.chain_16.3
 xanadu<5>loadgo n0 master -- 6 5
 Execution time = 7424 microseconds
***** Final Report*****
 Configuration : Linear Chain
 Total Nodes = 6
 Granularity = 1
 Total Buffers = 1
```
As we can see from the report printed by TrEK, two tasks executed on node 1 and 2. It appeared as if the program had finished successfully as TrEK successfully terminated and the Trollius "state" command showed that all the nodes were alive.

However, results were not being printed in a file. The error could have been that the data-generator() generated only two tasks. It could also have been that some compute-fn() of the slaves failed to return the results. It could also have been that the result-receiver() failed to receive the results. That is to say, the fault could have occurred in "any" of these three processes.

We recompiled the program with the monitoring option. No changes were made to the user program, the only change was in the compiling options in the Makefile. We rebooted the transputers, started Tmon-de, and re-executed the program.
As shown above, the final report is the same as the previous one except that the execution time had increased. At this point, we used the monitoring control command “monctrl” to end the monitoring session. The trace was collected in the file “group.all”, and the configuration was recorded in the file “group.config”. Using the “tvmfilter” to filter the trace, we got “group.tvm” in a format ready for input to deTrEK. We then created a file specifying the application, “group.spf” and used deTrEK to analyze the trace file. We obtained the following log.

```
xanadu<8>deTrEK -cp group

<table>
<thead>
<tr>
<th>TASK</th>
<th>SUBTASK</th>
<th>NODE</th>
<th>PROCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>Collecting</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>Collecting</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>Collecting</td>
</tr>
</tbody>
</table>
```

The analysis showed that the task 0 and 1 had been successfully generated, processed and their results collected. In addition, the analysis indicated that task 2, 3 and 4 had been successfully generated and computed at the message passing level. Although we were not sure that the results of these tasks were correct, we now knew that the data-generator() and compute-fn() were correct. Thus we focused our attention on the result-receiver() process. On examination, we found that it was basically a loop like:

```
result_receiver()
{
    for (i=0; i<Nsegs; i++) {
        msg = (user_result *)get_result();
        /* results printing and counting */
    }
}
```

Thus we hypothesized that the loop was never executed, which led the system process for receiving results to exit before all the messages had arrived. We added a print statement in the above function as follows:
result_receiver()
{  
    printf("Nsegs = \n", Nsegs);
    for (i=0; i<Nsegs; i++) {
        msg = (user_result *)get_result();
        /* results printing and counting */
    }
}

The program was then recompiled and rerun. The printout showed that the variable Nsegs was 0. It verified our hypothesis that the loop was never executed since the value should have been 5, the number of tasks. The fault was that this variable was not being initialized properly. A further check of the program showed that Nsegs was initialized in data-generator() as:

data_generator()
{  
    Nsegs = TotalTasks;
}

where TotalTasks was a global variable shared by all the processes. According to the specification of the PFVM, all the global variables should be initialized in init-master() because the data-generator() and the result-receiver() are independent processes, each with its own copy of the shared data which is initialized by the init-master(). Thus initializing Nsegs in data-generator() does not have any effect in result-receiver(). In other words, Nsegs in result-receiver() was not being initialized. We moved the above statement from data-generator() to init-master(), recompiled and ran the program once again.

Total Execution time = 12572736 microseconds
***** Final Report*****
Configuration : Linear Chain
Total Nodes = 6
Granularity = 1
Total Buffers = 1
Task Distribution:
Node 0  0 Tasks
Node 1  3 Tasks
Node 2  2 Tasks
Node 3  0 Tasks
Node 4  0 Tasks
Node 5  0 Tasks

It showed that all the tasks have been generated, computed and their results received.
A further check of the contents of the results showed that they were correct. The error had been located and corrected.

An uninitialized variable was one of the errors described in Chapter 3. In terms of the specification of parallelism, the error manifested itself in the untimely death of the result receiving process. In terms of message passing, the error appeared as an unanticipated message from the receiving process. This example supports our approach in examining the high level behaviour (message passing) to help to localize low level faults.

### 6.3 Evaluation

As the above experiments and actual experience show, deTrEK is able to locate the fault, its task number and process. Even in the case where the system does not crash, as in the above example, the trace analysis was helpful in narrowing down the suspicious part of the program. On a Sun4 workstation, deTrEK took approximately three minutes to analyze the trace produced by the example shown in the first section (a DCVM application with 1000 tasks running on a binary tree with 16 nodes). It took only a few seconds to analyze the other traces shown above. Experience shows that the first phase of our two-phase debugging approach is practical and effective. It can be done automatically by the monitoring tool and the trace analyzer. Thus it strongly supports
the hypothesis that effective parallel debuggers can be built using the concept of VM-based programming. More generally, restricted models of parallel computation not only simplifies programming, but also debugging.

Referring back to the general framework presented in Chapter 2, our debugging approach supports not only hypothesis generation, but also hypothesis modification and selection as our trace analyzer detects errors and localizes them at the same time. The error list gives the programmer a clear idea of which task and process to debug in the second phase, where hypothesis verification is to be done. By developing the fault models at the message passing level, the programmer is not required to manage the large number of processes in the system. Even though the second phase has not yet been implemented and evaluated, our initial experience with deTrEK has shown that the complexity of debugging is significantly reduced.

Our two-phase approach to debugging integrates the techniques of assertion checking and replay. The assertion checking is done automatically. It does not require the programmer to learn yet another language to describe the behaviour. Its results can also be used to limit the size of data to be replayed.

Our trace analysis is limited, however, to detecting and localizing fatal errors that result in deviation from the expected message passing behaviour.
Chapter 7

Conclusions and Future Work

7.1 Summary

This thesis has studied the debugging of parallel programs in a VM-based programming environment. Based on the analysis of the message passing pattern in the system, a hierarchical approach of two phase debugging is proposed. Fault models for two virtual machines, PFVM and DCVM, were developed to automate the first phase. The design and implementation of an event tracer and a trace analyzer are presented. The tools are then demonstrated and evaluated using a simulated application for DCVM and an actual program for PFVM.

We started with an analysis of the message passing behaviours of task oriented virtual machines, PFVM and DCVM. The possible errors in the systems were classified. These systems have a hierarchical structure. At the higher level, a conceptual model of the VM-based program is presented to the user. At the intermediate level, a set of configuration parameters are given to the user to choose an intermediate virtual machine on which the system may execute. The execution model of the application corresponds to the intermediate virtual machine level. Based on the execution model, fault models were developed for both PFVM and DCVM. The development of fault models led to the
design and implementation of a trace analyzer, deTrEK. The project also required the instrumentation of TrEK, and the revision and enhancement of Tmon. A simulated application was set up to validate the accuracy of the monitoring results and the trace analysis. We also demonstrated how our debugger could be used to debug an actual application using the processor farm virtual machine.

In Chapter 2, we proposed several criteria to evaluate a debugging approach, namely adequacy, productiveness, feasibility and efficiency. In terms of adequacy, we have taken the strategy of using high level abstraction and divide-and-conquer to overcome the complexity of parallel programs. By taking advantage of the structured message passing behaviour our hierarchical approach simplifies the debugging of a parallel program to that of a few specific sequential code segments. In addition, the trace analysis helps the programmer in both hypothesis generation and hypothesis selection. By collecting the message passing events during execution we were able to reproduce the errors in a simulated environment. For productiveness, our trace analyzer is limited to detecting and localizing fatal errors which cause deviation from the expected message passing behaviour. Through the design and implementation of a monitor and a trace analyzer, we have shown that our approach is feasible. Efficiency is achieved by compile time instrumentation of high level events and by post-mortem analysis.

Although the monitoring tool was designed for a transputer-based system and implemented under the Trollius Operating System, the instrumentation technique and the trace unloading strategy are applicable to a wide range of multicomputers. The hierarchical approach proposed in Chapter 4 can easily be adapted to any message passing system. Of course, the related fault models have to be developed accordingly. Our trace analyzer is an independent tool which can be used to analyze the event history of PFVM
and DCVM, irregardless of how they were implemented.

7.2 Future Work

7.2.1 Enhancement and Integration

Implementation of Second Phase

The implementation of the second phase of our debugging approach is obviously part of the work to be done. To do this, besides the message passing events between nodes, we require the following data:

- message passing events between processes on the same node,
- nondeterministic events internal to each process. For example, the value of the random number generated or the real time clock, etc..

The processes must be scheduled for re-execution in an order consistent with the message passing events as recorded. The replay of each process has to behave deterministically by reading the values recorded during the tracing stage as shown in [MAP91].

Integration with the Environment

As a tool, the interface of the debugger can be significantly improved. Integration of our tools with PARSEC[Fei92] would greatly improve its ease of use. Our experience is that a general visualization tool is usually not very helpful. However it might be helpful in both debugging and performance tuning by showing the event graph and its associated coalesced event graph.
Chapter 7. Conclusions and Future Work

The integration of debugging with performance modeling might reduce the number of test runs needed for monitoring. For example, the shipping control sequence could be defined based on the output of performance prediction. Furthermore, integrating the design of the virtual machines with fault models will improve the modeling, increase the efficiency of monitoring, and simplify the debugging monitor and the second phase implementation. It could also lead to a fault-tolerant virtual machine, in the sense that the machine could continue the computation of other tasks when some tasks fail and cause a node crash. This, of course, requires more flexibility in the management of resources. However, the benefit is that only a part of the computation needs to be re-executed and debugged without recomputing the tasks.

General Virtual Machine Analyzer

To further explore the applicability of our approach to parallel debugging, more virtual machines should be investigated. A language could be developed to describe the fault models of the virtual machines. Accordingly, a general assertion checker of that language could be designed and developed. After that, the development of a trace analyzer for a new virtual machine will be equivalent to the development of its fault models and their descriptions in that language.

7.2.2 Logical Time Stamping

Memory space and processor time are the two basic resources needed by a program. Of the two, time presents a more fundamental facet of distributed systems, namely causality relation between events [Ray92]. The logical nature of time is of primary importance when designing or analyzing distributed systems. In 1978, Lamport [Lam78] first proposed a
mechanism of logical time stamping for events in a distributed system.

Since then, a number of mechanisms for logical time stamping have been proposed. Raynal [Ray92] classified them as linear time, vector time and matrix time. Linear time is the one proposed by Lamport where the time domain is the set of positive integers. The time keeps the causality relation between two events. That is, if an event $a$ happens before event $b$, then the logical time stamp of $a$ is smaller than that of $b$. However, the vice versa is not necessary true.

Vector logical time was developed independently by Fidge [Fid89, Fid91], Mattern [Mat88] and Schmuck [Sch88]. In this mechanism, time is represented by an $n$-dimensional vector of integers, where $n$ is the number of sites involved in the communication in the systems. It establishes a causality relation that corresponds to the values of the time stamps.

Matrix time goes a step further to use an $n \times n$ matrix to represent the logical time. In addition to the properties of the vector clocks, each site understands that every other site knows its progress up to a certain local time.

From linear time to vector, to matrix time, the size of a time stamp increases from 1, to $n$, to $n^2$. Whether the time stamping is done dynamically, when a distributed system is executing, or statically, after the trace has been recorded, as we did, the number of events is usually large and the space to store the time stamps is significant. In the general case, it was shown by Bost [CB91] that the size can not be less than $n$. Based on the size of the poset (partially ordered set) coordinates, which can be used as a logical time stamp [Ore62, WTT83]. Summers [Sum92] states that the lower bound of logical time stamps is the dimension of the poset, where a point of the poset is an event and the binary relation is the “happen-before” relation between events.
Thus the question arises as to whether the restricted model of computation in task-oriented virtual machines can reduce the size of the logical time stamps required. If so, what are the new limits?

Formally, the problem can be stated as follows. Given a history of the message passing, we can define a poset, where a point of the poset is a send or receive event in the history and the binary relation between events is the “happen-before” relation. Given the specifications of an application, different runs of the systems may produce different posets because of non-determinism. Therefore, the specifications define a family of posets, constrained by the restrictions of the VM’s communication pattern. The minimal size of the logical time stamps required for these histories is the maximum dimension of the posets in this family.

It is easy to see that the above defined poset depends on $n$, the number of processors, and $m$, the number of tasks. We already know that it will not exceed the width of the poset, in this case, $n$. Therefore, the bound is possibly less than $n$ and if $m \gg n$ it should not depend on $m$. To determine the value of a tighter bound remains a challenging problem.
Bibliography


