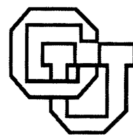
Trace Extrapolation for Parallel Programs on Shared Memory Multiprocessors

Zulah K. F. Eckert
Gary J. Nutt

CU-CS-804-96



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

Trace Extrapolation for Parallel Programs
on
Shared Memory Multiprocessors

Zulah K. F. Eckert and Gary J. Nutt
Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, CO - 80309

CU-CS-804-96 May 1996

University of Colorado at Boulder

Technical Report CU-CS-804-96
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Copyright © 1996 by
Zulah K. F. Eckert and Gary J. Nutt
Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, CO - 80309

Trace Extrapolation for Parallel Programs on Shared Memory Multiprocessors

Zulah K. F. Eckert*and Gary J. Nutt
Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, CO - 80309

May 1996

Abstract

Event traces are fundamental to performance evaluation of massive multiprocessor systems. This thesis focuses on the problem of extrapolating an event trace from a parallel program when program level nondeterminism is a possibility. Trace extrapolation refer to this task of taking a trace collected on an existing multiprocessor system and extrapolating the trace to a parametrically different multiprocessor system. The goal is to ensure that the extrapolated trace represents a correct execution of the program in the new environment. This problem is fundamental to many areas of multiprocessor systems research (e.g., performance analysis, perturbation analysis, race detection, etc.) and is the crux of the controversy surrounding the validity of trace-driven simulation of multiprocessor systems.

The trace extrapolation problem is complicated when parallel programs capable of exhibiting nondeterminism are considered, because the execution pattern exhibited by a program can change over subsequent executions using the same data set. The set of program instructions at which the execution pattern of a program might change can be identified; however, we demonstrate that the problem of determining this set of instructions is an NP-hard problem. We present two algorithms for approximating this set: the first is a global data flow algorithm similar to that used in constant propagation, and the second is a simplification of this method for iterative statements and procedures.

We present a formal framework, for representing executions and traces, and prove that the trace extrapolation problem is NP-hard. It is common practice to use approximations when confronted with difficult problems. Approximations, in general, trade a decrease in the complexity of a problem for an increase in the potential for an inaccurate solution to the problem. Asynchronous trace-driven simulation is an example of an approximation technique. The use of approximations necessarily changes our notion of the correctness of an extrapolated execution. Within our model, we characterize approximations and demonstrate the fundamental limitations in determining the accuracy of an extrapolated execution. Using this characterization, we propose metrics for trace accuracy and demonstrate that these metrics can be collected in time linear in the length of an extrapolated execution.

*This work was supported in part by Convex Computer Corporation.

Contents

1 Introduction **8**

1.1 Motivation . . . 8

1.2 Contribution . . . 13

1.3 Thesis Overview . . . 14

2 Background and Related Work **18**

2.1 Event Traces . . . 18

2.2 Trace Validity . . . 20

2.3 Trace Accuracy . . . 21

2.4 Trace Collection . . . 22

2.5 Simulation . . . 23

2.6 Trace Extrapolation . . . 28

2.7 Race Detection . . . 30

3 Preliminaries **32**

3.1 Trace Extrapolation and Decidability . . . 32

3.2 Environment and Assumptions . . . 32

3.3 Language Assumptions . . . 33

3.4 A Model of Program Executions . . . 37

4 A Categorization of Execution Pattern Distortion **44**

4.1 A Taxonomy for Execution Pattern Distortion . . . 44

4.2 Execution Pattern Determinism and Nondeterminism . . . 47

4.3 Locating Trace Change Points . . . 49

4.4 Execution Structuring Assumptions . . . 52

5 The Structure of Parallel Program Executions **54**

5.1 Motivation . . . 54

5.2 Paths . . . 55

5.3 The Graph of an Execution . . . 58

5.4 The Outcome of TCPs . . . 59

5.5 The Structure of Executions . . . 60

5.6 The Behavior of Executions . . . 65

6 A Framework for Execution Pattern Distortion **67**

6.1 The Shared Data Dependence and TCP Precedence Relations . . . 67

6.2 A Hierarchy of Execution Equivalences 70
6.2.1 The Hierarchy 70
6.2.2 Examples 71
6.2.3 The Hierarchy is Nontrivial 73
6.3 A Family of Execution Equivalences 76
6.4 Approximations 80

7 The Complexity of Trace Extrapolation **83**
7.1 Trace Extrapolation is Intractable 83
7.2 Complexity and the Real World 86
7.3 Feasible Executions 88

8 Computing TCP Sets **91**
8.1 The Shared Variable Dependence Relation 92
8.1.1 SVD variables 92
8.1.2 A Motivating Example 93
8.2 Computing the *SVD* Relation 94
8.2.1 Assignment Statements 94
8.2.2 Conditional Statements 96
8.2.3 Iterative Statements 97
8.3 Preprocessing 98
8.4 A Worklist Algorithm 99
8.4.1 Simple Constant 99
8.4.2 The SVD Variable Problem 101
8.4.3 Complexity and Correctness 103
8.5 An Asymptotically Faster Approximation 103
8.5.1 The SVD Graph 104
8.5.2 The Algorithm 106
8.5.3 Complexity and Correctness 110
8.6 Procedures and Functions 110

9 Measurement **112**
9.1 The Accuracy of Approximate Trace Extrapolation 112
9.2 The Accuracy of Measurement 114
9.3 Measuring the Accuracy of an Execution 117
9.4 Computing Distance Metrics for Executions 120

10 Comparing Executions **122**

10.1 Correspondence 122

10.2 The Theorem of Correspondence 123

10.3 Correspondence Algorithm 125

10.4 The Preprocessing Phase 125

10.4.1 The TCP Basic Block Flow Graph 125

10.4.2 The Comparison Phase 127

10.5 Correctness and Complexity 129

10.5.1 Correctness 129

10.5.2 Complexity 131

10.6 Computing Optimal Correspondences 132

11 Approximate Trace Extrapolation **135**

11.1 Overview 135

11.2 Measurement 135

11.3 Data Structures and Preprocessing 138

11.4 Measurement Algorithms 141

11.5 Extrapolation 144

11.6 Complexity, Correctness, and Measurement Results 144

12 Conclusion **147**

12.1 The Classification of Instructions 147

12.2 Trace Change Points 148

12.3 Determinism, Nondeterminism, and Traces 149

12.4 A Formal Model of Parallel Program Executions 149

12.5 Trace Extrapolation is Intractable 150

12.6 Measurement and Accuracy in Approximations 151

12.7 Corresponding Parallel Program Executions 151

12.8 Interprocess Communication 152

12.8.1 Message Passing 152

12.9 Other Memory Consistency Models 153

12.9.1 Weak Memory Consistency 153

12.10Future Research Directions 154

12.10.1Trace-Driven Simulation 154

12.10.2Time 154

12.10.3Trace Accuracy 154

12.10.4Trace Reconstruction 154

12.10.5 Program Level Nondeterminism 155

List of Figures

1 Trace Extrapolation 9
2 Approximate Trace Extrapolation 12
3 Thesis Overview 17
4 A parallel program exhibiting multiple possible execution paths and patterns 19
5 Feasible and infeasible executions of the program in Figure 2.1 19
6 Trace-driven simulation 24
7 Direct simulation 26
8 Trace Collection 28
9 Trace Extrapolation 28
10 A program using spinlocks for exclusive access to shared memory 34
11 The program of Figure 3.1 represented in intermediate code form 35
12 A program fragment and associated (hypothetical) instructions (assuming that variable x is in register r_0, variable y is in register r_3, and variable j is in register r_2) . . 38
13 An illustration of Axiom 2 40
14 A parallel program exhibiting multiple execution patterns 54
15 (a,b,c) Possible executions, correspondence, the graph executions of the program in Figure 5.1 56
16 A pattern nondeterministic parallel program 75
17 A program fragment and (hypothetical) instructions (assuming that variable x is in register r_0, variable y is in register r_3, and variable j is in register r_2) 77
18 Compression of TCP events in two executions, from the program of Figure 6.2, using a TCP higher-level view 77
19 The family of sets in the hierarchy 79
20 Execution from the program of Figure 6.1 and the associated sets S_0 and S_0^A 80
21 Family of sets and corresponding approximations in the hierarchy 82
22 A parallel program containing a simple TCP requiring complex path information to decide 87
23 A parallel program illustrating SVD set calculation 93
24 A program fragment containing assignment statements 95
25 A program fragment containing array reference assignment statements 96
26 A program fragment containing a non TCP conditional statement 97
27 A program fragment containing a TCP iterative statement 98
28 An example control flow graph 100
29 The lattice and associated rules for simple constant 100

30 The lattice for the SVD variable problem and associated rules 102
31 SVD graph of program fragment in Figure 8.5 104
32 An algorithm for computing the SVD graph for a parallel program 105
33 Reduction of the graph in Figure 8.9 107
34 An algorithm for approximating the set of TCP statements for a program 108
35 An algorithm for approximating the set of TCP statements given an SVD graph . . 109
36 Trace extrapolation with error metric D 113
37 Possible executions under consideration in an approximate trace extrapolation method 114
38 Possible distances between source, target, and actual target traces 114
39 A parallel program exhibiting pattern nondeterminism 116
40 A possible source, target, and actual execution for the program in Figure 9.4 118
41 An algorithm to compute a correspondence M given two executions and a program in TBB form. 127
42 A parallel program producing executions for which the correspondence of Figure 10.1 is not optimal 134
43 Trace extrapolation coupled with distance calculations 136
44 Possible node types for the TBB flow graph 140
45 Two possible lock acquisition scenarios during extrapolation 142
46 An algorithm for calculating distance measures 143
47 An extrapolation algorithm 145
48 The parallel program instruction and statement taxonomy 147
49 An example parallel program using the message passing operations **send** and **receive** 153

1 Introduction

1.1 Motivation

Event traces have been fundamental to performance evaluation methodology since the 1960s [49]. Event traces have been used for processor workloads in trace-driven simulations of memory hierarchies in shared memory computer architectures [1] [22], to study the behavior of cache systems [51], in the study of page replacement algorithms [41], etc. Event traces are also used for parallel language debugging [43]. The area of performance analysis and visualization has used event traces in performance tuning environments [44] [46]. Shared memory massive multiprocessors are increasingly being produced and used, yet little is known about the theoretical limitations of the correctness of trace executions used in simulations and analyses of these systems. This thesis demonstrates several of these theoretical limitations.

In this thesis, we generalize the notion of an event trace by considering program executions. A program *execution*[1] is a sequence of event occurrences in the context of a particular *execution architecture* for the program. That is, when a program is executed by a particular runtime system with a particular operating system on a particular hardware platform, the execution is a sequence of event occurrences that were observed during the execution together with a temporal description of the execution.

Executions can be recorded either as *timestamped* event streams or as *causal* event streams. A timestamped execution incorporates the causal order, but also adds the virtual time at which each event occurred. The type of execution considered depends directly on the application under study. For event traces, causal traces are useful when the trace defines a load on a model whereas timestamped traces are best used for direct analysis, since they already incorporate the resultant performance behavior. This thesis focuses on causal executions.

Event traces are collected by observing the execution of a program on a particular execution architecture. For sequential programs executing on a uniprocessor platform, the process of collecting a correct causal trace is well understood [38]. The process of observing parallel programs executing on multiprocessor platforms can perturb the execution of a program. Perturbation of the execution of a parallel program can result in a collected trace that is a different interleaving or different set of events than might have occurred in the absence of observation. A change to the set of events exhibited in a trace is called *execution pattern distortion*. Using correct traces to drive simulations ensures that inaccuracies due to trace collection overhead/artifact do not effect simulation results. More precisely, a trace is considered to be *correct* if it is a trace of the program execution that

[1] For now, we will use the terms *trace* and *execution* interchangeably. In Section 3, we formally define execution and trace.

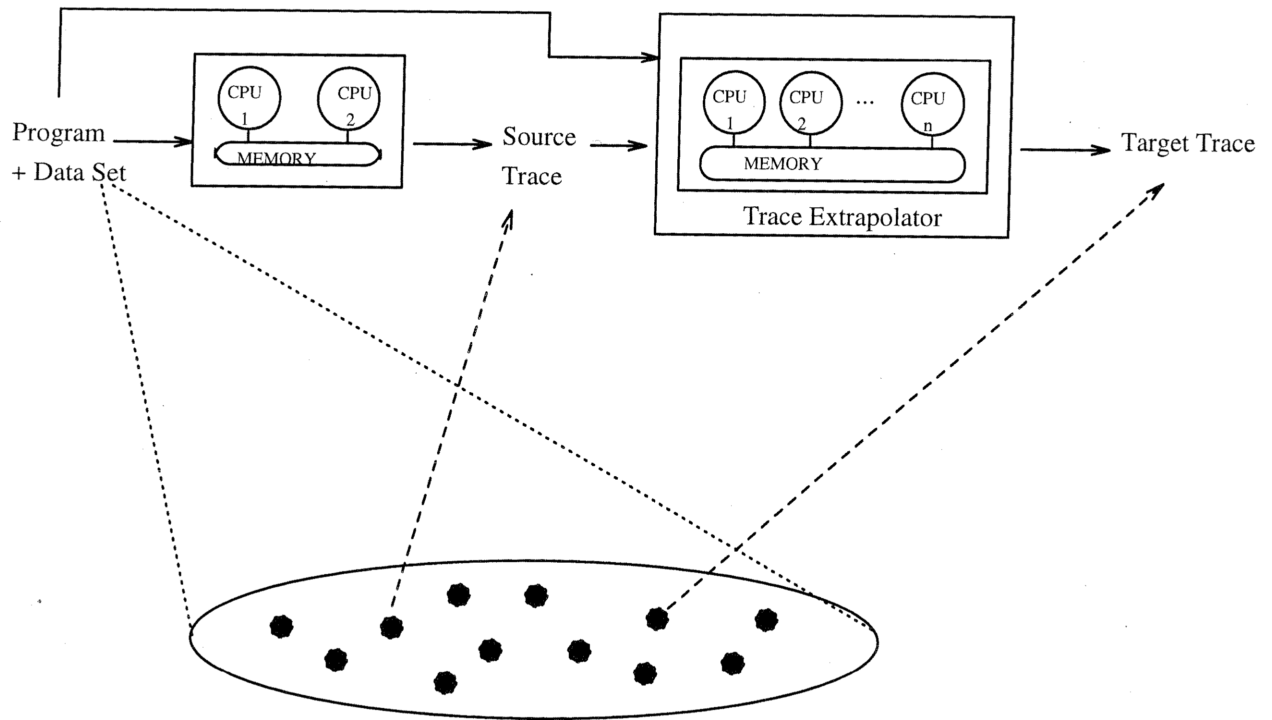


Figure 1: Trace Extrapolation

would have resulted in the absence of observation. A conflict arises between the need to collect correct traces and the necessity of observation in order to collect events.

Even if accurate traces can be collected, inaccuracies still arise because the execution architecture under study may not exist. In this case, ensuring a correct trace means that the collected trace must be moved from the execution architecture of collection to the execution architecture under study. This process is called trace extrapolation and is depicted in Figure 1. Here, a *source* trace is collected on an existing execution architecture that is parametrically related to the execution architecture under study. This trace is a member of the set of causal executions possible given a parallel program and an input data set. The process of moving the trace to a new execution architecture may cause the interleaving of events or the actual set of events in the trace to change (i.e., execution order distortion) yielding a different execution. This new trace, the *target* trace must be a member of the set of possible executions for the program and data set choice and must also be the correct execution for the target execution architecture (i.e., the execution architecture under study). Ideally, the source trace should be *reused* to produce the target trace for a different machine configuration. That is, the program was used to produce the source trace and the portions that contributed to the source trace should not be *reexecuted* during production of the target trace. More precisely, the problem of *trace extrapolation* is that of taking a trace collected on an existing execution architecture, given a program and data set, and adjusting the trace without the use of program reexecution to produce a trace that is a correct trace for a parametrically different execution architecture (i.e., resulting in the trace that would have occurred had the program and data set been run on the hypothetical execution architecture). The trace extrapolation problem is the fundamental question of whether or not trace correctness can be guaranteed for simulations. If trace extrapolation is not possible, then direct simulation must be used for simulations requiring a correct trace. This thesis addresses the complexity of the trace extrapolation problem.

Holliday and Ellis [32] present an algorithm for extrapolating parallel program traces. The algorithm relys on static analysis of a parallel program to identify the points in the source code that can induce changes in an address reference trace given a fixed data set. However, the complexity and correctness of the algorithm are not considered. We extend Holliday and Ellis' work by further identifying the structure that a parallel program induces on a set of executions. This structure is the set of points at which any two executions of a program, on a given data set, can *diverge* and where they must *converge* after a period of divergence. The divergence points are called *trace change points* (TCPs) and the convergence points are called *convergence points* (CPs). We demonstrate that the problem of statically locating TCPs (and CPs) is an NP-hard problem and present an algorithm to compute a conservative *approximation* of these sets, in $O(N \times E \times V^2)$ [2] time.

[2] N is the number of statements in the program, E is the number of edges in the program statement flowgraph, and V is the number of variable in the program.

In order to concisely identify programs that produce executions sets for which trace extrapolation is tractable, the level of nondeterminism of a parallel program must be considered. Characterizations of program nondeterminism exist (see [11] and [21]), however these characterizations identify the nondeterministic behavior of a program based on the data values that the program computes. That is, a program is *data nondeterministic* if it has the possibility of producing more than one set of data values on a given input data set. This characterization is not sufficient for our purposes since a data nondeterministic program may produce traces comprised of a single set of events. We call such programs *execution pattern deterministic*, or pattern deterministic. For pattern deterministic programs, trace extrapolation is a simple process. Whereas for pattern nondeterministic programs, those producing traces comprised of possibly different sets of events, trace extrapolation is complicated. The structure present in the set of executions from a parallel program, allows us to concisely identify the program as pattern deterministic or pattern nondeterministic.

This thesis provides a formal model in which to study the trace extrapolation problem. The model is based on our characterization of nondeterminism in parallel programs. The model uses equivalence of program executions to group sets of structurally similar executions. Equivalences are classified as either *data and pattern deterministic*, *data nondeterministic and pattern deterministic*, or *pattern nondeterministic* with the following relationship:

$$\textit{Data and Pattern Det.} \subset \textit{Data Nondet. and Pattern Det.} \subset \textit{Pattern Nondet.}$$

We identify a family of equivalences that are pattern nondeterministic. This family can be viewed as a means of differentiating nondeterministic program behavior and represents an indication of the size of a trace extrapolation problem instance. Within the framework, equivalences that lead to sets of executions for which trace extrapolation is tractable can be identified. These equivalences identify program behaviors that lead to tractable or intractable trace extrapolation. Using the framework, we demonstrate that, for the smallest choice of equivalence in the family of pattern nondeterministic equivalences, the trace extrapolation problem is in general NP-hard. This set is significantly smaller than the set used in Holliday and Ellis' solution to the problem and demonstrates that their solution is in general an unreasonable one.

Trace-driven simulation is the process of using an event trace to drive a simulation of an execution architecture. Trace-driven simulation is the seminal example of the necessity of trace extrapolation. The target environment for a trace-driven simulation may not exist and the trace used for simulation may be from a parametrically different environment. While trace-driven simulation is accurate for sequential programs running on uniprocessor environments, where extrapolation is unnecessary, the technique has been shown to be inaccurate for multiprocessor computer systems if trace extrapolation is not performed [26]. Our work with trace extrapolation demonstrates that approximations to the trace extrapolation process must be considered. The issue of using poten-

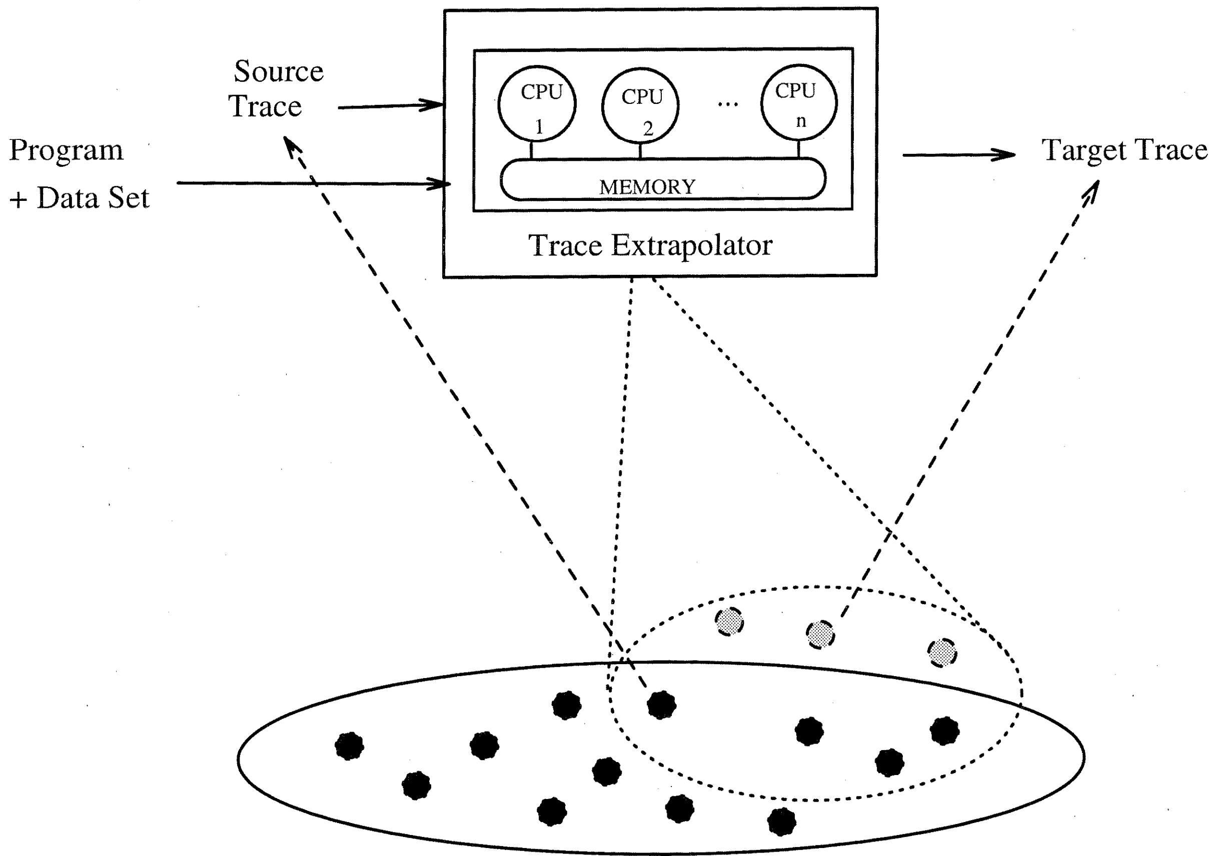


Figure 2: Approximate Trace Extrapolation

tially incorrect traces for trace-driven simulations has not been resolved [6] [26]. We extend our framework to include approximations.

The process of approximating trace extrapolation is depicted in Figure 2. Here, a source trace is generated on an existing execution architecture. This trace is then extrapolated to a new trace based on the behavior of the execution architecture under study. In order to achieve tractability, approximate techniques must change the set of executions considered for extrapolation either by reducing the size of the set, adding executions that are not feasible[3] , or both. Clearly there is a trade off between the tractability of an approximate trace extrapolation approach and the accuracy of the trace produced. For some choices of execution sets, extrapolation will not be tractable and some choices of approximations will not provide adequate simulation results (i.e., will not be sufficiently accurate).

While there are many opinions about what makes a trace accurate, currently, there are few characterizations of the accuracy of traces used in simulations (see [26] [6] [27] [32] [35] for examples). We propose several measures of trace accuracy based on our knowledge of the inherent structure of traces. We use these measures to demonstrate the fundamental limitations of measuring trace accuracy.

Calculating measures is in itself a problem. Parallel programs are used for large applications and are likely to produce very large traces. We contend that for an algorithm for comparing traces to be reasonable, it must run in linear time, preferably on-the-fly. While algorithms exist that compare sequences in linear time [20], these algorithms will be unreasonable for large sequences due to the necessity of reversing an input sequence. We demonstrate that executions are sequences with identifiable structure and that a correspondence between two executions can be computed in linear time using linear space. This correspondence identifies periods of convergence and divergence in traces, highlighting the areas where measurement must occur. Finally, we demonstrate an approximate trace extrapolation algorithm that produces a trace and accuracy measures for the trace in linear time using linear space if the trace is stored in a single contiguous file and constant space if the trace is stored with a thread of the trace per file.

1.2 Contribution

The main goal of this thesis is to provide a mathematical framework in which to characterize trace extrapolation and approximate trace extrapolation.

To achieve these goals, we extend Holliday and Ellis' [32] identification of the causes of execution pattern distortion in parallel program traces by presenting a taxonomy of parallel program

[3]Given a program and data set, an execution is *feasible* if it resides in the set of possible executions for the program and data set.

instructions, and statements. This taxonomy classifies instructions based upon their behavior at execution time – static, fixed over one run, or dynamic. The taxonomy also takes in to consideration the effects of externally introduced nondeterminism (e.g., processor scheduling). In addition, we introduce convergence points and demonstrate that a program induces identifiable structure in an execution. Using the taxonomy, programs are identified as either execution pattern deterministic or execution pattern nondeterministic based on their behavior with respect to the set of traces that they produce.

The problem of the static location of the set of TCPs in a program is open. This thesis demonstrates that this problem is NP-hard and presents a data flow analysis algorithm that computes a conservative estimate of this set.

The problem of the complexity of trace extrapolation is open. Using the characterization of program level nondeterminism and our knowledge of the structure of executions, we present a formal model of parallel program executions. This model is based on equivalence of executions. Using this model, we demonstrate that the trace extrapolation problem is NP-hard.

Goldschmidt and Hennessy have demonstrated that trace extrapolation must occur in order for trace-driven simulation of shared memory multiprocessor systems to be accurate [26]. Given the complexity of trace extrapolation, we address the use of approximate techniques. We extend our model to include approximations to executions. Few characterizations of the accuracy of a trace exist. Based on our model, we propose accuracy measures for traces designed to aid in determining the accuracy of simulation results that depend on approximation techniques. There are fundamental limitations to the ability to determine the accuracy of a trace and this thesis presents those limitations.

We present an algorithm for corresponding parallel program executions in linear time on-the-fly. We use this algorithm to demonstrate an approximate trace extrapolation algorithm that produces a target trace and accuracy measures for that trace in linear time, on-the-fly.

1.3 Thesis Overview

The organization of the thesis is depicted in Figure 3. Section 1 motivates the trace extrapolation problem and identifies the open problems related to trace extrapolation. The contribution of the thesis is identified.

Section 2 presents background for the thesis. This includes a discussion of trace validity, existing characterizations of trace accuracy, trace collection, trace simulation systems, and motivating work in trace extrapolation. Our work benefits from research in parallel program debugging and race detection. We present a brief discussion of relevant work.

Sections 3-6 comprise the details of our model, with Section 6 presenting the formulation of the model. Section 3 contains basic assumptions about the execution architectures and the language

under consideration. The basic underlying model for our formal model is presented. Sections 4 and 5 characterize the behavior of parallel programs and their executions. In Section 4, a taxonomy of instructions is presented. This taxonomy identifies the degree of trace reuse possible for an instruction or statement. The taxonomy also identifies trace change points and proposes execution pattern determinism and execution pattern nondeterminism as a characterization of program level nondeterminism with respect to the set of traces a program can produce. Section 4 concludes with a proof that the problem of determining whether or not a program is pattern deterministic is intractable. A corollary of this result is that the problem of statically locating TCPs is NP-hard.

In Section 5, it is proven that executions from the same parallel program and data set experience identifiable periods of convergence and divergence. The points of divergence are TCPs and the points of convergence are termed convergence points, or CPs. In Section 6, we use the behavior of executions, induced by a program, to distinguish the set of parallel programs that are execution pattern nondeterministic. Section 6 defines the TCP precedence relation $\rightarrow_{tp}$ and adds this relation, together with the shared data dependence relation $\rightarrow_{sd}$, to our basic model of executions. These relations are used to characterize data determinism and execution pattern determinism in a set of executions. The model characterizes execution equivalences in a hierarchy that distinguishes the behavior of a program, based on the behavior of its set of executions. Using knowledge of the structure induced on a set of executions by a parallel program, a family of execution pattern nondeterministic equivalences is defined. This family is critical for the addition of approximations to our model. In addition, it characterizes the behavior of execution pattern nondeterministic programs.

In Section 7, we demonstrate that for data and pattern deterministic programs, and data nondeterministic and pattern deterministic programs, trace extrapolation is tractable. The use of equivalences in the model and the family of executions defined in Section 6 identifies the smallest structural equivalence that pattern nondeterministic. Using this set, we demonstrate that the trace extrapolation problem is NP-hard.

In Section 8, two algorithms are provided for making conservative approximations of the set of TCPs in a parallel program. The first uses data flow analysis techniques to compute the set in $O(N \times E \times V)$ time. A simplification of this algorithm allows for the analysis of procedures and functions. We use this simplification in a second algorithm for iterative constructs resulting in an $O(N \times V)$ time bound.

Section 9 demonstrates the fundamental limitations of measuring trace accuracy during approximate trace extrapolation. Three measures of trace accuracy are proposed based on the work of Section 6.

Section 10 relys on information in Section 5 to prove that executions can be corresponded in linear time. Specifically, in Section 5, it is demonstrated that in an execution, the CP associated

with a particular TCP can be located and that periods of divergence and convergence can also be located. Using this information, in Section 10, it is shown that a CP associated with a TCP can be located on-the-fly. We prove that executions can be corresponded in linear time. An algorithm for correspondence is presented and proven correct.

Section 11 presents an algorithm that approximates trace extrapolation. This algorithm produces an approximate trace and accuracy measurements in linear time. This is accomplished using the correspondence algorithm in Section 10 to compute a correspondence between execution on-the-fly, identifying potential periods of divergence and convergence.

Finally, Section 12 concludes the thesis and identifies other areas of application for the trace extrapolation problem and the implication of the results of this thesis for those areas. Possible future research directions are presented.

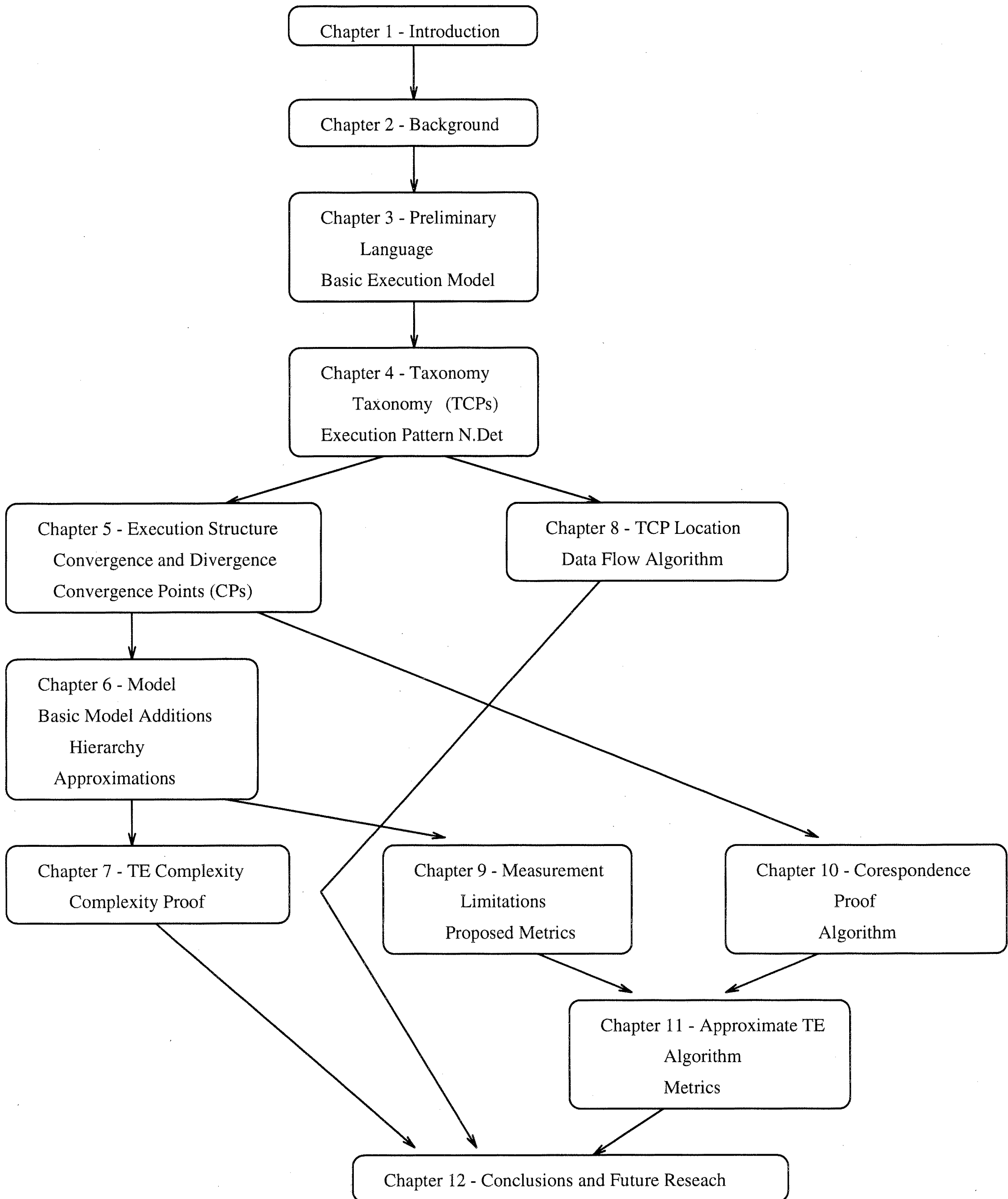


Figure 3: Thesis Overview

2 Background and Related Work

In this section, we present background and related work for this thesis. It should be noted that the trace extrapolation problem is not decidable. In Section 3, we discuss the implications of the decidability of this problem and other problems considered in this thesis.

2.1 Event Traces

An *event trace*, or *trace*, is a record of events that occur during the execution of a program on a particular execution architecture. An *execution architecture* is a particular hardware, operating system, and runtime system platform on which to execute a user program. An *event* results from the execution of a program instruction (e.g., loads and stores to particular memory locations are events resulting from the execution of load and store instructions). The set of events collected depends on the intended application for the trace (e.g., architectural simulation, performance simulation).

The set of events recorded can consist of hardware, operating system, and user-level events. *Hardware events* are generated by the hardware during the execution of a program. Some hardware events can be recorded (e.g., memory requests across a bus), however events that occur on-chip cannot be observed or recorded (e.g., a load from an on-chip cache). *Operating system events* are generated from the execution of instructions belonging to the source code of the operating system (i.e., requests for operating system services cause operating system code to be executed resulting in operating system events). *User-level events* are those events that result directly from the execution of an instruction from a user program.

Traces can be either timestamped or causal depending on their intended use. A *timestamped trace* consists of a set of events together with the real time at which each event occurred (i.e., a timestamp). Timestamped traces are collected in order to gain information about the behavior of the particular execution architecture from which they are collected. A *causal trace* is a record of the causal order of events in an execution. A causal trace is useful when the trace is to be used as a workload for simulation of an execution architecture.

We assume that an *execution* is a record of all of the user-level events that occur during the execution of a particular program given a particular data set together with their causal order. An execution is *feasible*, given a program and data set, if it is an execution that could have occurred from executing the program with the data set. The *path* of an execution is the temporal ordering placed on the set of events of the execution. The *pattern* of an execution is the set of temporal orderings placed on the set of events of the individual processes (or threads) of the execution. Figure 5 demonstrates some of the possible executions for the program of Figure 4 as well as some infeasible executions. Notice that execution E1 represents a different path than execution E2. However, they represent the same pattern. Execution E3 represents a different pattern than E1

```
                        A0: begin;
                                shared int x = 0;
                                shared lock l1;
                        A1: task_create(2)
task 0                                                  task 1
    int i;
B0: lock(l1);                                           C0: lock(l1);
B1: i = x + 1;                                          C1: x = x - 1;
B2: unlock(l1);                                         C2: unlock(l1);
B3: if i > 0 then                                       C3: end;
B4:         j = i;
B5: end;
                        A2: task_terminate(2);
                        A3: end;
```

Figure 4: A parallel program exhibiting multiple possible execution paths and patterns

E1	E2	E3	E4
A0	A0	A0	A0
A1	A1	A1	A1
B0	B0	C0	B0
B1	B1	C1	B1 ←
B2	B2	C2	B2
C0	C0	B0	C0
C1	B3	C3	C1 ←
B3	C1	B1	C2
C2	B4	B2	B3 ←
B4	C2	B3	B5 ←
C3	B5	B5	C3
B5	C3	A2	A2
A2	A2	A3	A3
A3	A3		

Figure 5: Feasible and infeasible executions of the program in Figure 2.1

and E2. Execution E4 is an infeasible execution since whenever B1 executes before C1, i will be greater than zero at line B3 (i.e., statement B4 would have executed). Two paths are *equal* if they represent the same interleaving of the same set of events. In Sections 3 and 4 we formalize these notions.

2.2 Trace Validity

Accurate observation and recording of program executions is considered critical for accurate simulation and analysis of computer systems. The act of observing a program during execution in order to collect a trace can introduce inaccuracies in the collected trace. A trace is considered valid, producing accurate simulation results, if the trace is correct. Specifically, a collected trace is *correct* for a program, data set, and an execution architecture, if it is the trace of an execution that would have resulted in the absence of observation, given the program, data set, and execution architecture.

A correct causal trace can be collected from a sequential program executing on a uniprocessor. This is because sequential programs are deterministic (both in the traditional sense and in the sense of data and pattern deterministic introduced in Section 1), producing a single output value for a given input, and therefore producing a single path of execution. For parallel programs executing on shared memory multiprocessors, the act of observing the execution of a program can perturb program execution resulting in a recorded trace that may represent a different path or pattern than would have occurred in the absence of observation. In addition, for timestamped traces, time perturbations may be introduced into the timestamps. Methods exist to limit the amount of perturbation induced by collection [38] [27] and to remove time perturbations post mortem [39]. Changes in the path and pattern of an observed execution are due to program level nondeterminism (e.g., a program produces one of a set of possible outputs for a given input) and execution architecture nondeterminism (e.g., due to the use of processor scheduling algorithms that are not deterministic).

More precisely, variations in the path or pattern of an execution over subsequent runs, using the same data set, are termed *distortions*. For parallel programs, distortions fall into three categories [53]: access order distortions; wait-time distortions; and execution order distortions. An *access order distortion* occurs when instructions are reordered that are not accesses to shared memory (i.e., instructions having different possible interleaving resulting in different possible execution paths). A *wait-time distortion* occurs when the time that a process waits at a synchronization point is modified. An *execution order distortion* represents a change in the *execution pattern* for a program and can occur when the order of access to shared variables is modified.

2.3 Trace Accuracy

Traces can be incorrect yet accurate enough for use with simulations. The importance of a particular type of distortion depends on the intended use of the trace. Traces can be directly analyzed to produce performance measurements. This analysis requires a high degree of trace accuracy since any inaccuracies are likely to be carried into the resulting measurements. For trace-driven simulation, access order and wait-time distortion can be accounted for by re-adjusting the time of events during simulation [53] [18]. In contrast, current research cannot account for execution order distortion based on language semantics or postmortem analysis [53] [26] [6] [32]. Trace-driven simulation is known to be valid for traces exhibiting access order and wait time distortions. However, for traces exhibiting execution order distortions, the validity of trace-driven simulation is unknown.

Methods exist that increase trace accuracy and ensure accurate collection of traces. Perturbation analysis is used to increase the accuracy of a collected timestamped trace by removing the effects of time perturbations, due to instrumentation, using postmortem trace analysis [39]. Time dilation can be seen as a measure of the accuracy of a trace collection method. *Time dilation* is the ratio of the execution time of the perturbed program to that of the unperturbed program. Time dilation can, in the worst case, create a situation where all instructions appear to take the same amount of time. This affects the behavior of the program and hence of the resultant trace (e.g., the path or even the pattern of a trace may change). In addition, for timestamped traces, the timing information present in the trace is inaccurate. Time dilation provides useful information for determining the accuracy of a timestamped trace. That is, less time dilation implies a more accurate trace. However, no such correlation between time dilation and accuracy has been shown for causal traces (see [35] and [26]). Fujimoto and Hare [24] have demonstrated that uniform time dilation across all processors results in an accurate trace.

Other characterizations of trace accuracy exist. For example, in [7] it was demonstrated that very long address reference traces are needed to accurately characterize the behavior of cache memories. This example demonstrates that even a correct trace may not yield good simulation results.

Thus, for correct traces and for incorrect traces researchers are interested in knowing the accuracy of a trace. While very few quantitative characterizations of trace accuracy exist, qualitative characterizations are discussed in the literature including: whether or not a trace is representative of the behavior of the program [27] [26] (e.g., as opposed to being an outlier in the set of possible executions); whether or not a trace was likely to have occurred given a particular execution architecture; whether or not a trace is feasible [6]; the sensitivity of a parallel program; and the nondeterminism present in a program.

2.4 Trace Collection

Trace collection or generation is the process of observing the execution of a program to produce a temporal stream of events (a trace) as they occurred during execution. Collection of events can be performed in three ways: unintrusive monitoring, intrusive monitoring, and execution architecture emulation/simulation. *Unintrusive monitoring* can be performed with hardware monitoring devices. *Hardware monitors* collect only those events that are present at a location where a physical instrumentation probe can be attached. The use of hardware collected traces is limited, as many computers have on-chip events that cannot be collected (e.g., cache events). In addition, many hardware monitoring systems collect only address reference events. Hardware monitoring has the advantage of capturing all references both user and operating system; however sometimes it has the disadvantage of collecting only segments of traces due to memory and bandwidth limitations.

It is possible to construct a simulation of the execution architecture under study, thus eliminating the need for trace collection using probes. Simulation of the hardware and firmware, or *emulation*, is a time consuming process. A hardware and firmware emulation coupled with a software simulation of the runtime environment for the execution architecture under study, called *emulation/simulation* of the execution architecture, can be used for trace collection and for the collection of other information (i.e., processor execution time). One obvious drawback to this method is the time to produce such an emulation/simulation system[4]. Even if an emulation/simulation exists, performing simulations on these systems is a time-intensive process. These drawbacks coupled with the lengthly runtime of parallel programs and a large volume of trace data makes this method an unpopular one.

The shortcomings of hardware monitoring and the complexity of emulation/simulation have popularized the use of software based intrusive monitoring techniques. *Intrusive collection* techniques can be characterized by the fact that they perturb program execution in order to collect events. The difficulty with perturbing the execution of a program in order to collect events is that the collected trace may no longer be correct. For example, a timestamped trace requires accurate timing information taken during the execution of a program. A perturbation necessarily changes the timing of a program resulting in an incorrect timestamped trace. Similarly, perturbing the execution of a parallel program can lead to changes in the causal ordering of the observed execution path (i.e., access order distortions) or the observed execution pattern (i.e., execution order distortions).

Intrusive monitoring techniques fall into three categories: interrupt-based, program instrumentation, and microcode alteration. *Interrupt-based* trace collection relies on the ability of a computer

[4] At some stage in the development of a shared memory multiprocessor, such a system is likely to exist, but may not be widely available.

to interrupt the execution of a program after each instruction, allowing the details of the instruction execution to be recorded. Unfortunately, regular interrupts of the program by the operating system represent a significant increase in the overall runtime of the program (e.g., the time to collect the trace) and represent significant intrusion on program behavior. Eggers and Katz [18] have used interrupt-based techniques to record traces.

Trace collection systems using *program instrumentation* are abundant (e.g., [27] [19] [52] [7] [13] [8] [47]). Programs can be instrumented at compile time, in executable (binary) form, or in assembly language form. Instrumentation techniques vary, but in general, a call to an address analyzing routine is placed in the code wherever trace information has to be collected. During program execution, information is collected by the address analyzing routine resulting in a trace. Instrumentation systems reduce the level of program intrusion by introducing static (compile time) analysis techniques to determine exactly which events must be collected at runtime and which events can be determined either statically or postmortem. In [38], instructions were classified as *easy*, *difficult*, and *impossible*. Easy instructions could be determined given a loadmap of the program (e.g., the outcome of a load on a global variable). Difficult instruction could be determined based upon the result of an impossible instruction and a simple arithmetic calculation. Finally, impossible instructions could not be determined statically or post mortem. These instructions require instrumentation. Using this characterization, Larus was able to take an approach to instrumentation that significantly reduced the intrusion due to instrumentation.

Traces have been collected using a method that requires *microcode alteration* [2]. The ATUM system introduces an execution time dilation of approximately 20 times that of normal program execution. This method produces highly accurate traces that include all user and system address references, however the obvious drawback is a loss of portability and the necessity to modify the microcode of a machine.

2.5 Simulation

There are five methods available to researchers for evaluating the performance of computer systems: simulation; analytical modeling; measurement; emulation; and hardware prototyping. Time and economic considerations make hardware prototyping the least desirable of these methods. Emulation/simulation requires a detailed hardware and firmware simulation (an emulation) together with a software simulation of the runtime system. Emulation/simulation is only reasonable if either an emulator exists, or if one is willing to construct one. Measurements are produced by direct analysis of a collected trace and the accuracy of a trace determines the accuracy of measurements. Measurement is only reasonable if the execution architecture under study exists from which an accurate trace can be collected.

Some analytical modeling techniques use trace analysis to extract parameters from a trace for use in analytical models (i.e., mathematical models). Both simulation and analytical modeling use event traces. In *simulation* the behavior of a computer system is simulated given a particular workload (i.e., an event trace). Some *analytical modeling* techniques use analysis of events traces to extract parameters on which an analytical model can be based. Here, an event trace is collected and analyzed to infer the behavior of a program on a given execution architecture. Modern multiprocessor computer systems exhibit complex interactions that cannot be adequately characterized using analytical techniques. For this reason, simulation is the most widely used method for performance analysis of shared memory multiprocessors. See [33] for more details of the study of performance analysis of computer systems.

Simulation of massive multiprocessor systems [33] [53] [26] using traces fall into two categories: trace-driven simulation and direct simulation.

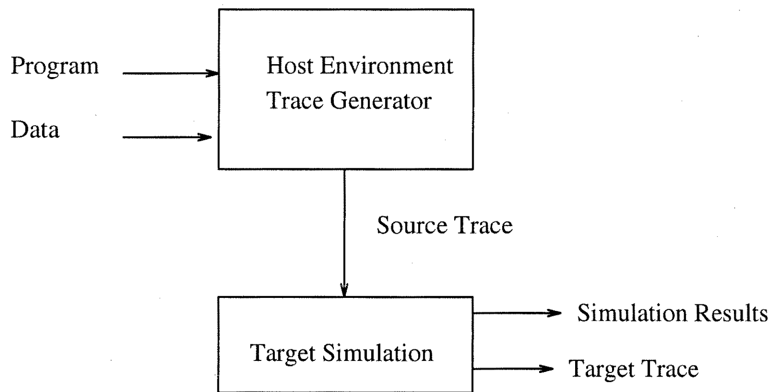


Figure 6: Trace-driven simulation

Trace-driven simulation uses an existing source trace, collected from a host execution architecture, to drive a simulation of a target execution architecture. One difficulty with trace-driven simulation is that the target execution architecture may not exist. Therefore, the host execution architecture (used for trace collection), is unlikely to be the same execution architecture as the target execution architecture (although, the instruction set of the execution architectures must be the same). Figure 6 demonstrates the trace-driven simulation process. A host execution architecture is used to collect a source trace that serves as input to a simulation of a target execution architecture. The target simulation may or may not use trace extrapolation to adjust the source trace to fit the target execution architecture. The target trace is the product of this extrapolation process and is used for input to the target simulation. The target simulation may produce results and if required, may also produce the target trace.

There are two methods of trace consumption in trace-driven simulation: synchronous and asynchronous [6]. In *synchronous trace-driven simulation*, the simulation makes no attempt to re-order events in the causal source trace (e.g., there is no attempt to extrapolate the source trace to produce an accurate target trace). In *asynchronous trace-driven simulation* the simulation re-orders events in the source trace based on the timing present in the simulation (i.e., the source trace is extrapolated producing a target trace). In addition, synchronization wait-times may be adjusted to the timing of the simulation. The result is a causal target trace that differs from the causal source trace in path. Ideally, the extrapolation process should produce a target trace that represents a correct trace for the target execution architecture. However, in the case of asynchronous trace-driven simulation, an infeasible trace can result.

There are drawbacks to trace-driven simulation. While trace compression can reduce the size of a trace, storage requirements for a large trace may be unreasonable. Trace-driven simulation can be executed concurrently with a trace generation system such that a trace is produced at the same time as it is consumed (on-the-fly). However, the ability to re-order instructions and changes to wait-times that can occur during simulation can lead to a bottleneck requiring some trace storage.

The validity of trace-driven simulation for multiprocessor computer systems has been questioned. Deterministic timing dependencies in a source trace can be accounted for in the target trace [26] [18] [53]. For example, instructions can be re-interleaved and access to critical subsections can be allowed based on timing and dynamic behavior of the target execution architecture. However, when program level nondeterminism exists, these changes to the source trace can result in a target trace that represents an impossible program causal ordering (i.e., an infeasible trace). For this reason, Bitar [6] argues that results from trace-driven simulations cannot be guaranteed to be accurate when program level nondeterminism is present and this being the case, trace-driven simulation should not be used for programs exhibiting program level nondeterminism. However, Bitar assumes that traces must be correct in order to provide accurate simulation results.

Goldschmidt and Hennessy [26] demonstrate experimentally that synchronous trace-driven simulation yields inaccurate results and should not be used for simulation of multiprocessor computer systems. This is because it is unlikely that the source and target trace exhibit similar behavior with respect to synchronization. As a result, synchronous trace-driven simulation is likely to result in increased latency times due to the stalling of processors in the target trace in order to match the synchronization exhibited in the source trace. This result implies that extrapolation of the source trace must occur in order to produce accurate simulation results.

The existence of a reasonable trace extrapolation algorithm would ensure the validity of trace-driven simulations. However, as we will demonstrate, the trace extrapolation problem is intractable. This implies that either the trace extrapolation process must be approximated or the trace-driven simulation technique must be ruled invalid for the simulation of multiprocessor computer systems

when program level nondeterminism is possible. It should be noted that nondeterministic scheduling of work to processors can also result in changes to the execution pattern of a program. While the issue of processor scheduling is interesting and raises important questions, it is out of the scope of this research.

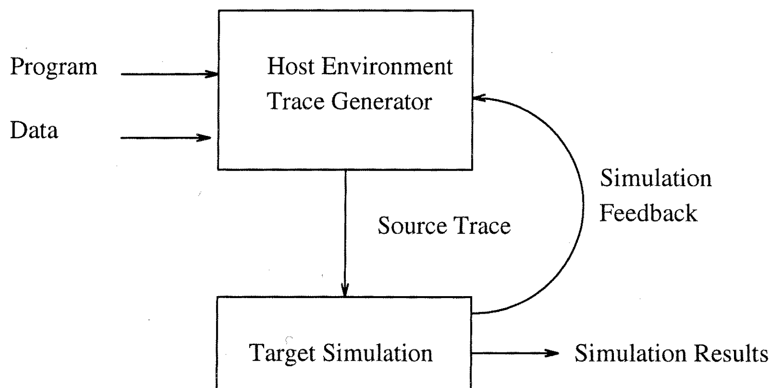


Figure 7: Direct simulation

Recent research in intrusive collection methods together with questions about the validity of trace-driven simulation have given rise to a plethora of direct simulation systems. In *direct simulation*, the environment of trace generation is tightly coupled with that of simulation, see Figure 7. The trace generation system feeds events of interest to the simulation system, and the simulation system supplies feedback to the trace generation system. This feedback, or coupling, varies and includes: timing information, used by the trace generation system to make adjustments to timing based on the behavior of the simulation; and processor or thread scheduling information, used to determine the outcome synchronization events. The exact degree of coupling depends on the direct simulation system. The TANGO [13] simulation system couples trace generation and simulation by supplying events of interest (e.g., memory reference events), generated by the trace generator, to the target execution architecture simulator. The target execution architecture simulator in turn returns timing and latency information, resulting from the simulation of the most recent events, to the trace generation system. The trace generation system maintains the timing for trace generation and uses timing information from the simulation to adjust the trace generation time as needed. In contrast, the SPAE [27] trace generation system places the control for trace generation with the simulation. In this case, the simulation requests events from the trace generator and determines from which thread of execution the events will come. Here, the simulation determines processor interleaving. This requires a runtime library that determines synchronization.

Direct simulation systems can be coupled such that the trace generator has maximal independence from the simulation system or, such that the trace generator simply services requests for

events, as in the case of SPAE. Regardless of the degree of coupling, the result is a system that generates a trace without emulating the target environment and that is flexible enough to account for timing and execution pattern behavior of the target environment.

The trace generation process can be accomplished using either execution driven simulation, software interpretation, or a hybrid technique, such as code synthesis [40]. Software interpretation is generally considered too slow and time consuming for trace generation. *Execution driven simulation* uses native execution and direct simulation to produce simulation results. In *native execution*, a program is directly executed on a host machine that has the same instruction set as the target machine being simulated. Native execution is accomplished using the program instrumentation technique for trace collection. This relieves the trace generator of the burden of keeping track of program state information. However, this method requires switching between the various contexts being executed. The trace generation system must keep track of this context switching information (e.g., registers). Execution driven simulation is the most widely used method for trace generation (see [13] [12] [27] [19] [8] [47]). Hybrid methods exist that combine software interpretation and native execution [56]. *Code synthesis* is the process of encapsulating for execution an inline block of code, containing no internal events of interest, for execution. These blocks can be natively executed while the overall state of the program is maintained by the interpreter. This has the effect eliminating unnecessary context switching and represents a speed up over the native execution approach [56].

There are drawbacks to using direct simulation. Direct simulation can be time consuming. The *Tango* system has a dilation factor of 1 to 18000 depending on the events of interest, demonstrating the potential for large simulation times. Because direct simulation systems rely on native execution, they tend to apply solely to a specific instruction set. All of the aforementioned direct simulation systems introduce time perturbations into a trace. Time dilation factors range from potentially huge (on the order of 1-18000 times for *Tango*) to relatively small (1-3 times for *MPtrace*). The *MINT* [56] and *Tango Lite* [26] systems are exceptions. The *Tango Lite* system uses program instrumentation, and removes the effects of this instrumentation to produce a correct trace. The cost of trace correctness is increased simulation overhead. Goldschmidt and Hennessy [26] note that the 45X dilation factor for *Tango Lite* makes it unreasonable for some problem sizes. The *Mint* system eliminates the need to instrument code by using the hybrid method of code synthesis.

Finally, direct simulation systems are deterministic. That is, given a program, data set, and a simulation, a direct simulation system will produce the same trace every time. An advantage is that results are reproducible. However, the trace chosen may not yield accurate simulation results (e.g., for cache simulations, the trace used for simulation may not be long enough [7]).

2.6 Trace Extrapolation

It is often the case that the execution architecture one is interested in simulating is hypothetical version of an existing execution architecture (e.g., more processors, larger memory, faster CPU), see Figure 8. In these situations, trace extrapolation can reduce costly direct simulation time.

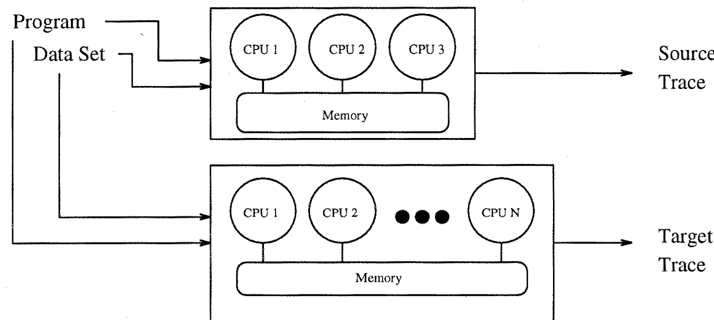


Figure 8: Trace Collection

Trace extrapolation is the process of moving from a source trace to a target trace. This process is illustrated in Figure 9. Here, a *source* trace is collected on an existing execution architecture that is parametrically related to the execution architecture under study. This trace serves as input to the trace extrapolator (see Figure 9). The trace extrapolator produces a target trace that can serve as input to a trace-driven simulation.

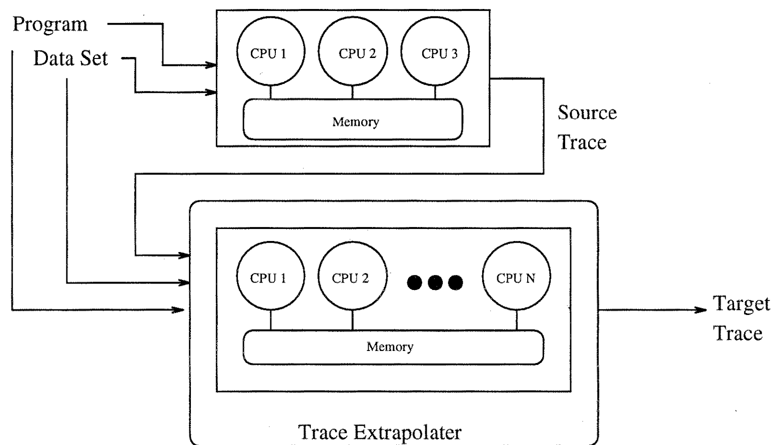


Figure 9: Trace Extrapolation

The process of moving the trace to a new execution architecture may cause the interleaving of events or the actual set of events in the trace to change (i.e., execution order distortion) yielding a different execution. This new trace, the *target* trace, must be a member of the set of possible

executions for the program and data set choice and must also be the correct execution for the target execution architecture. Ideally, the source trace should be *reused* to produce the target trace. That is, portions of the program were executed to produce the source trace and those portions should not be *reexecuted* during production of the target trace.

More formally, the problem of *trace extrapolation* is that of taking a trace collected on an existing execution architecture, given a program and data set, and adjusting the trace without the use of program reexecution to produce a trace that is a correct trace for a parametrically different execution architecture.

In their paper [32], Holliday and Ellis address the issue of the accuracy of traces used in trace-driven simulation of massive multiprocessor systems and propose an extrapolation algorithm. To accomplish this, they identify the points at which the execution pattern of a trace might change when the trace is reinterpreted in a new execution environment. These points are *Address Change Points*, ACPs, and are exactly the points in the source code of a program at which the address reference pattern exhibited by a trace can change. These points include conditional statements and array reference statements whose outcome can change depending on the value of a shared variable. Consider again the example in Figure 4 and the executions in Figure 5. This program contains two ACPs, B4 and B5. Notice that B4 and B5 are ACPs because these instructions are exactly the points at which two executions of the program can vary. The instruction B3 is an *ACP operand.* Notice that this is the case for executions E2 and E3.

Holliday and Ellis propose a method of trace-driven simulation that guarantees that a simulator can adjust a source trace to an appropriate target trace for the target environment. To avoid reexecution of the source program, they propose a method of *deciding* the outcome of an ACP operand based on the set of paths that determine the branching direction (or reference outcome in the case of an array reference) for an ACP operand statement. That is, each ACP operand in the program is effectively equipped with a set of paths (represented as a *path expression*) for which the ACP operand branches positive. During simulation, whenever an ACP operand is encountered, the current path of the target trace is checked against this set. If the path is in the set, then the positive branch of the program is taken, otherwise the negative is taken. Consider again the program in Figure 4. The two possible patterns describing the outcome of the ACP operand B3 are $\Sigma^+ B_0 \Sigma^+ C_0 \Sigma^+$ and $\Sigma^+ C_0 \Sigma^+ B_0 \Sigma^+$. The former represents the set of paths for which B3 branches positive and the latter represents the set of branches for which B3 branches negative. Whenever the branch direction implies that code not previously executed must be reexecuted, the reexecution is performed in an unspecified manner. A statement of the Holliday and Ellis trace extrapolation problem is given below:

Trace Extrapolation Problem: Given a program P, a fixed data set D, and a source trace s that is the result of executing P on D in some fixed environment M, and given

a new target environment M', can a target trace t, that is a correct trace for P on D in environment M' be constructed from s and P such that no portion of P executed during the execution resulting in s is ever reexecuted in the construction of t.

Holliday and Ellis discuss characterizations of programs that are reasonable to extrapolate. They point out that a program that requires no additional execution of P is easier to extrapolate. These programs are called *path-traceable.* A program is *graph-traceable* if the set of ACPs of the program can be decided without reexecution of the program. They pose the question of the complexity of the construction of path expressions for an arbitrary program.

ACPs are an important idea for this thesis. However, we characterize the structuring points of traces in a slightly different manner. This thesis focuses on traces consisting of arbitrary program events as opposed to just focusing on address reference traces. We use the notion of an ACP operand and rename then Trace Change Points (TCPs). For example, in Figure 4 B4 and B5 are ACPs and B3 is an ACP operand. We define TCPs to be the point after which a change can occur and hence in Figure 4 B3 is a TCP. For array reference statements, the address reference that can change is a TCP. Notice in executions E2 and E3, the pattern of the execution changes at B3.

2.7 Race Detection

In his work, Taylor [54] demonstrated that for Ada tasks, many questions about synchronization structure were intractable. It follows that the problem of determining the possible synchronization orderings for a program is intractable. Race detection attempts to determine orderings that could have occurred between events in an execution in alternate executions of a program. Specifically, race detection determines whether or not two shared memory reference events could have occurred unordered (e.g., a race condition). It follows from Taylor's work, that the problem of detecting races is in general intractable. Trace extrapolation also attempts to determine alternate ordering for a program execution – specifically, an alternate path. We adapt results from Netzer's thesis [43] and show that the problem of statically locating TCPs is an NP-hard problem.

The study of race detection for parallel programs can be divided roughly into two areas: static detection efforts, and dynamic detection efforts. Static race detection attempts to detect race conditions using static analysis (data flow analysis) of a parallel program (e.g., see [5] [9] [55]). Other static detection methods consist of state space reconstruction for a program (e.g., see [54] [30].) Dynamic methods attempt to detect race conditions either post mortem from a trace of the program execution (e.g., see [4] [42]) or on-the-fly during execution (e.g., see[15] [14]). These methods detect races visible in a single execution of a program.

Many variants of the race detection problem can be solved in polynomial time for branch free programs (see Table 2.2 of [31] for a complete listing). However, the result most applicable to those

of trace extrapolation is the work of Netzer in his thesis [43]. Netzer's thesis work demonstrates that the problem of detecting race conditions in branch free programs using unlimited general semaphores is NP-hard. Given an execution, it is shown the problem of determining what might have occurred in alternate executions of the given execution, is intractable[5]. Three sets are constructed from a given execution. First, F_{SAME}, the set of all executions that exhibit the same shared date dependence. Second, F_{DIFF}, the set of all executions that execute the same set of instructions. Finally the set F_{SYNC}, the set of all executions, feasible or infeasible, that execute the same set of instructions. Netzer shows that detecting race conditions over any of these sets is intractable. Recently, Helmbold and McDowell [31] have shown that the problem of determining race conditions in programs with branching and fork/join style synchronization is NP-hard.

[5]NP-complete, NP-hard, and Exponential problems are commonly referred to as *intractable* problems. For our purposes, any problem that is at least as difficult as an NP-complete problem will be referred to as intractable.

3 Preliminaries

3.1 Trace Extrapolation and Decidability

The trace extrapolation problem is in general not decidable. This can be demonstrated with a classic decidability proof; a demonstration that deciding the trace extrapolation problem implies that the halting problem can be decided. In addition, as is the case with many problems requiring the static analysis of source code, trace extrapolation is is undecidable in a more practical sense. This is due to the possibility of the presence of library calls in source code and the potential for complex conditional statements. For the former, static analysis of the source code of a library routine may not be possible, making trace extrapolation undecidable. For the latter, conditional statements can contain, or depend on, expressions that are complex enough as to be effectively undecidable without program execution.

The fact that trace extrapolation is in general undecidable does not make the problem less interesting. That is, even a partial algorithm for trace extrapolation would be useful. Any algorithm correctly extrapolating parallel program traces is a *partial algorithm* in the sense that the algorithm extrapolates traces but is not guaranteed to halt on a given input. Both the complexity of the trace extrapolation problem, Theorem 7.1, and the complexity of TCP location, Theorem 4.3, are partial results.

3.2 Environment and Assumptions

Our work focuses on shared memory multiprocessors. We therefore make assumptions about execution architectures being addressed, program content (to simplify our discussion), and program correctness (to avoid detailed discussion of this research area).

In the determination of results, we assume sequential consistency[6] [37] for shared memory multiprocessors (e.g., the Convex SPP and the KSR-1). This restriction rules out nondeterminism due to memory references that are not caused by program nondeterminism. It should be noted that there are shared memory multiprocessors that are not sequentially consistent (e.g., the Convex C-series and the Sequent Symmetry). In Section 12 we explain how the results of this thesis could be extended to shared memory multiprocessors using weak memory consistency, and to message passing systems.

Each processor has its own local memory and the set of processors shares a virtual address space. In addition, the processors are assumed to be synchronous, that is, actions of the processors

[6] A shared memory multiprocessor is *sequentially consistent* if each processor issues memory requests in the order specified by the executing program and requests from the set of processors are serviced in the order in which they are received (first-in-first-out).

occur at clock cycles, and there is one clock for all of the processors. We assume lightweight process threads as the schedulable unit of computation.

3.3 Language Assumptions

In order to avoid addressing issues that are out of the scope of this thesis (e.g., debugging, race detection, program correctness, and termination), we make the following assumption:

> **A1**: All programs are syntactically correct, error free, and halt on any possible input data.

The work of this thesis includes static analysis of programs. The analyses presented here cannot be performed on unstructured programs (i.e., those relying on the unstructured use of **goto** statements). This implies (among other things) that programs have a single entry and exit point for iterative constructs[7].

> **A2**: Programs are well structured.

Finally, the issues of mapping between executable and source code and of the effects of compiler optimizations are also outside of the scope of this work.

The language used to describe computations is a simple, C-like, procedural language in the three-address intermediate code form of [3] modified to include the **task_create** and **task_terminate** constructs and an atomic unary **test-and-set** operation[8]. This canonical form allows for a single shared variable or array reference per statement. Any program that is expressible in this intermediate form is a possible program. The intermediate code uses arrays, pointers, and local and shared variables. All standard arithmetic operations are possible. Figure 3.1 is an example of a parallel program (using spinlocks) in the hypothetical programming language.

The **task_create(n)** statement creates n processes and begins to execute them concurrently; one for each of the n tasks being executed. In Figure 3.1, each task executes to completion and the **task_terminate** statement causes the parallel threads of execution to merge to a single thread. In Figure 3.1, the program ends immediately after this point.

[7] Most modern codes do not use unstructured **goto** statements. For example, in the SPLASH suit of programs [50], only one program, PTHOR, uses **goto** statements. These statements are used in a structured manner and can easily be replaced, for example, by a function call.

[8] The test-and-set(x) operation sets the value of x to 1 and returns the previous value for x.

```
                 begin

                 shared int index = 0;
                 shared int lk = 0;
                 shared int array[10];
                 int n = 2;
                 task_create(n)

task 0           task 1

    int aval;                                   int bval;
    int k = 0;                                  int j = 0;
A:  lock(lk);                               A:  lock(lk);
B:  index = CONSTANTVAL_1;                  B:  index := CONSTANTVAL_2;
C:  unlock(lk);                             C:  unlock(lk);
D:  aval = array[index];                    D:  bval = array[index];
E:  for (k = k+1; k < aval; k++);           E:  for (j = j+1; j < bval; j++);

                 task_terminate(n);

                 end;
```

Figure 10: A program using spinlocks for exclusive access to shared memory

```
begin

shared int index = 0;
shared int lock = 0;
shared int array[10];
int n = 2;

task_create(n)

task 0

      int aval;
      int acopy;
      int k = 0;
      int temp;

A:    acopy = test-and-set(lock);
B:    if acopy == 1 goto A;
C:    index = CONSTANTVAL_1;
D:    lock = 0;
E1:   temp = index;
E2:   aval = array[temp];
F:    k = k + 1;
G:    if k < aval goto F;

task 1

      int bval;
      int bcopy;
      int j = 0;
      int temp;

A:    bcopy = test-and-set(lock);
B:    if bcopy == 1 goto A;
C:    index = CONSTANTVAL_2;
D:    lock = 0;
E1:   temp = index;
E2:   bval = array[temp];
F:    j = j + 1;
G:    if j < aval goto F;

task_terminate(n);
end;
```

Figure 11: The program of Figure 3.1 represented in intermediate code form

Figure 3.2 is the program of Figure 3.1 translated into three address intermediate code form. Notice that **lock** and **unlock** statements of Figure 3.1, lines A and C, are replaced by **test-and-set**, lines A, B, and D of Figure 3.2. In subsequent examples, we use the abstract events **lock** and **unlock**, rather than the **test-and-set** construct, to simplify the presentation of examples, definitions, and proofs. The for loop, line E of Figure 3.1 is replaced by an increment branch combination, lines F and G of Figure 3.2.

In addition, we have the following restrictions.

Restriction 1. At most one operand for a statement can be an array component or use a dereference operator.

Restriction 2. At most one load or store instruction, per statement, can be to a shared variable. This restriction does not apply to the **test-and-set** atomic operation.

These restrictions ensure that any statement can be the cause of at most one access to shared memory or one array access. In addition to these transformations, expressions are transformed to a single operator form using temporary variables. Conditional statements are transformed into a single-branch form and loops are transformed into increment and conditional statements. Access to critical sections are transformed into instances of the **test-and-set** operator with the actual busy waiting being done via single branching.

The intermediate code statements 1-5 listed below are from Holliday and Ellis [32]. We have added a process creation statement (6), the function call (7), and the program initial and final statements (8). (Each statement in Figure 3.2 is in one of these forms).

1. Assignment statements of the form $x := y$ or $x := y \; op \; z$.
2. Indexed assignment statements of the form $x[i] := y$ or $y := x[i]$, for arrays.
3. Address and pointer assignments of the form $x := \&y$, $x := *y$, $*x := y$, where $\&$ is the *address-of* operator and $*$ is the *dereferencing* operator.
4. Unconditional branching of the form **branch** L where L is the label of the next statement to be executed.
5. Conditional branch statements of the form **if** x **relop** y **goto** L where relop is a standard relational operator and L is the label of the next statement to execute (if the condition is true).

6. Process creation and termination statements of the form **task_create(n)** and **task_terminate(n)** where n is the number of processes being created.

7. Function invocation statements of the form **function_name**$(p_1, p_2, \ldots, p_n)$ where n is the number of parameters for the function call.

8. The program initial statement **begin** and the final statement **end.**

Each intermediate code statement has a corresponding sequence of native instructions, in the native language of the host and target machines, which we will refer to as *instructions.* Any program in the hypothetical language can be transformed into an intermediate code program obeying restrictions 1 and 2 using simple syntactic transformations.

3.4 A Model of Program Executions

We present a model for program executions based on a language that includes the intermediate code statements listed in the previous section with the **lock** and **unlock** representation of **test-and-set**. This model can be adapted to languages with other forms of synchronization by replacing the appropriate axioms.

A *program* is any sequence of statements with an initial statement, **begin**, and a final statement, **end**, and such that for every **task_create** statement, there is a corresponding **task_terminate** statement and for every **lock** statement, there is a corresponding **unlock** statement[9].

A *program event* is comprised of one or more consecutive intermediate code statements (or instructions). An *execution event*, or *event*, is the result of executing a program event. A single program event can produce multiple events and may be executed multiple times during an execution. Therefore, a program event may correspond to multiple events and multiple event instances.

Consider the example of Figure 12 depicting a C program fragment together with its corresponding assembly language instructions (in a hypothetical assembly language). Here, a program event, shown in the left hand column, has a corresponding assembly language instruction(s) (or events), shown in the right hand column. Comparing events that occur due to different program events will not be of interest in this thesis. Rather, only events that are instances of the same program event are compared for outcome. For example, the store instructions, **store** r_3**, loc(y)**, generated by program events A3 and A4 are identical. However they will not be comparable in our work since they are instances of different program events. Definition 3.1 defines comparable events.

[9]Notice that if a program is transformed from one using the **test-and-set** statement to one containing only **lock** and **unlock** statements, this is always true.

```
.
.
.
A1: x = x + 1;          add r - 1, 1
                        store r1,loc(x)
A2: if (j > 0)          branchle r2, 0, A4
A3: y = y - 10;         sub r3, 10
                        store r3,loc(y)
                        branch A5
A4: add r3, 10
                        store r3, loc(y)
A5: j++;                incr r2
.
.
.
```

Figure 12: A program fragment and associated (hypothetical) instructions (assuming that variable x is in register r_0, variable y is in register r_3, and variable j is in register r_2)

Definition 3.1 *Given two executions from the same program and data set. Two events are* comparable *only if they are event instances of the same program event.*

□

Given this restriction on the set of comparable events, we define two events to be equivalent only if they are comparable and equal.

Definition 3.2 *Given two executions from the same program and data set, one execution containing event a and the other containing event b. Events a and b are equivalent, denoted $a \equiv b$, if a and b are comparable and $a = b$.*

□

Therefore, two events are only equivalent, if they are the same event and are instances of the same program event.

Given a fixed data set, each program has a set of possible executions (i.e., programs have the possibility of exhibiting nondeterminism). In Section 6, we present a model in which sets of

executions can be formalized. The goal of the model of this section is to provide a formal means of describing a single execution from a parallel program.

We use a basic model of executions based on Lamport's theory of concurrent systems [36]. In this theory, there is no assumption of atomicity for language statements (or instructions). This provides a formalism with which to reason about a program execution at different levels of detail. For our purposes, we consider each event to conceptually have a start and finish time. The relation $a \rightarrow_T b$ implies that event a executes and finishes before event b begins. Therefore, a can causally affect b, but b cannot affect a. Whenever two events execute *concurrently*, we have $\neg(a \rightarrow_T b)$ **and** $\neg(b \rightarrow_T a)$[10]. That is, neither a nor b necessarily completes before the other begins — their executions overlap. The temporal order relation $\rightarrow_T$ is an irreflexive partial order relation.

A program execution is a pair consisting of a set of events and the temporal order relation $\rightarrow_T$. The set of events represent all instances of statements (or instructions) performed during execution. More formally,

Definition 3.3 *A* program execution, *P,is a pair $< E, \rightarrow_T >$ where E is a finite set of events and $\rightarrow_T$ is the temporal ordering relation defined over E.*

□

It is possible to construct executions that could not actually have occurred simply by executing statements in an order that violates program semantics. A program execution P is an *actual execution* if P represents an execution that could actually have occurred given the program at hand — P is a member of the set of possible executions for the program. Our assumption that E is finite is consistent with the assumption that all programs terminate.

Given our intermediate code language, it is possible that more than one *thread* of execution exists at a given time. We call these threads of execution *processes*. Given a program and a set of events E for that program, we use E_p to denote the set of events for process p and $e_{p,i}$ to denote the ith event in process p. We occasionally use $E_{p,j}$ to denote the set of events executed by process p for execution j. We use $E_{u(v)}$ to denote the set of all *unlock* events on shared variable v, and $E_{l(v)}$ to denote the set of all *lock* events on shared variable v.

The axioms below for the relation $\rightarrow_T$ are similar to those given in [36] and [43]. The axioms constrain the temporal order relation to be consistent with the semantics of our intermediate code language. Axioms 1 and 2 make $\rightarrow_T$ consistent with the notion that an ordering based upon the

[10]Because of the assumption of sequential consistency, a single relation $\rightarrow_T$ can be used to describe the temporal behavior of executions.

start and finish times of all events is total.

Axiom 1 *$\rightarrow_T$ is an irreflexive partial ordering relation.*

Axiom 2 *If $a \rightarrow_T b$, $\neg(b \rightarrow_T c)$ and $\neg(c \rightarrow_T b)$, and $c \rightarrow_T d$ then $a \rightarrow_T d$.*

For Axiom 2, notice that the definition of $\rightarrow_T$[11]. implies that if two events in an execution are unordered then they must overlap. To see this, consider a fixed execution such that there is no order between two events b and c, $\neg(b \rightarrow_T c)$ and $\neg(c \rightarrow_T b)$, then b could not have executed and completed before c and c could not have executed and completed before b and therefore, some portion of the executions must overlap. Axiom 2 states that if in an execution events b and c are unordered (i.e., they overlap), then a must complete before d begins. This is because b must either start or end sometime during event c at which point a has already completed and d has not yet started. Figure 13, from [36], illustrates Axiom 2.

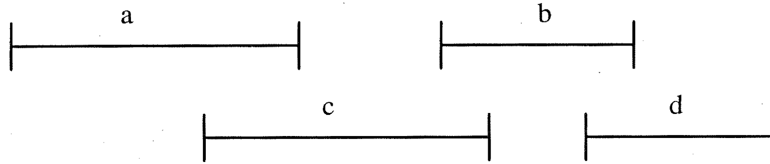


Figure 13: An illustration of Axiom 2

Axiom 3 ensures that a linear ordering of events is preserved within a process (thread).

Axiom 3 *$e_{p,i} \rightarrow_T e_{p,i+1}$ for all processes p and events $e_{p,i} \in E_p$, $p \neq c$.*

Axiom 4 ensures that each *task_create* instruction has a corresponding *task_terminate* instruction.

Axiom 4 $\text{task_create}_{p,i} \rightarrow_T e_{c,j} \rightarrow_T \text{task_terminate}_{p,i+k}$ *for all $e_{c,j} \in E_c$.*

[11]Notice that the current definition of time does not include the initiation and completion times for events. Instead, executions are ordered only if one starts and completes before the other starts. While this notion of time order is sufficient to model causal traces, it is not sufficient for timestamped traces.

Finally, language semantics guarantee that no two corresponding critical sections are entered in an overlapping period of time. That is, given that the set of locks for a program are all initially unlocked, in any execution of the program, for each lock l, a lock event on the lock must be followed by a unlock event on the same lock. Given a set of locks S and sets U_l and L_l, the set of unlock and lock events for lock l in an execution,

Axiom 5 *For every* $l \in S$ *with* U_l *and* L_l, $\forall i \in L_l (\exists j \in U_l (\forall k \in \{L_l - i\}(i \rightarrow_T j \;\; and \;\; (j \rightarrow_T k \;\; \vee k \;\; \rightarrow_T i))))$ *and* $|L_l| = |U_l|$.

Each lock has an associated set of lock and unlock events. Each lock event has an associated unlock event for which there are no intervening lock or unlock events in the execution. Axiom 5 and Axiom 1 guarantee that each lock event on a particular lock is followed by an unlock event (i.e., that lock and unlock events match).

One of the benefits of using Lamport's model is that an execution can be viewed at different levels of detail. This allows our final model to encompass the notions of abstract events as well as that of a trace.

Lamport [36] defines a higher-level view of a program execution. Here, we use a variation of this definition to define the instruction-level view and the trace view of an execution. The instruction-level view allows and execution to be viewed at the finest level of detail. A trace view allows certain details of an execution to be removed from the execution, and corresponds to the notion of a trace of an execution.

A higher-level view of a program execution is a derived program execution in which the set of events has been partitioned and a new relation $\rightarrow_T$ is defined that is consistent with the original execution. More formally,

> **Definition 3.4** *A* higher-level *view of a program execution* $P = < E, \rightarrow_T >$ *is another program execution* $P_H = < H, \rightarrow_{T_H} >$, *where* H *partitions* E *and for all* $A, B \in H$: $A \rightarrow_{T_H} B \Leftrightarrow \forall a \in A$ *and* $b \in B, a \rightarrow_T b$.
>
> □

It is possible to construct higher-level views that are not valid program executions (i.e., do not obey all of the above axioms). Definition 3.4 does not rule out the possibility that events from different processes are grouped together nor the possibility that an event is part of the implementation of more than one higher-level event (e.g., ends up in more than one partition). Also, it allows synchronization and process control events to be partitioned as to violate Axiom 5. We are interested

in only those higher-level views that produce valid executions. Any higher-level view obeys Axioms 1-3. Whenever we use a higher-level view it will be shown to obey Axioms 4 though 5.

The ability to view an execution at a low level of detail is key to the notion of a program trace. The instruction-level view of a program execution is a new program execution in which the set of events is broken down into individual instructions and new relation $\rightarrow_T$ is created that is consistent with the original execution. We assume that there is a mapping from program statements to instructions that obeys the semantics of our programming language. More formally,

Definition 3.5 *An* instruction-level *view of a program execution* $P = < E, \rightarrow_T >$ *is another program execution* $P_I = < E_I, \rightarrow_{T_I} >$ *where each computation event in* E_I *comprises a single instruction and* P *is a higher-level view of* P_I.

□

Returning to the definition of a higher-level view, we know that E partitions E_I and that for all $A, B \in E$: $A \rightarrow_T B$ implies that for all $a \in A$ *and* $b \in B$, $a \rightarrow_{T_I} b$. Any instruction-level view obeys Axioms 1-3. It is a simple matter to demonstrate that Axioms 4 and 5 are also obeyed (although the current notation used in these axioms may not apply). To see this, recall that the mapping from program statements to instructions obeys the semantics of our programming language. Because of this, whenever a program execution obeys Axioms 4 and 5, its corresponding instruction-level view must also obey these axioms.

Definition 3.6 provides a formalization of the notion of a trace based on our model of program executions. A trace is a derived execution in which part of the view is eliminated. Any trace has a set of *traceable* instructions (e.g., all load and store instructions). A trace view of a program execution is an incomplete program execution in which some members are removed from the set of events and new relation $\rightarrow_T$ is created by removing all members (of the relation) that relate a removed event. More formally,

Definition 3.6 *Given a set of instructions I a* trace *view of a program execution $P = < E, \rightarrow_T >$ is another program execution $P_t = < E_t, \rightarrow_{T_t} >$ where given the instruction-access view of P, $P_I = < E_I, \rightarrow_{T_I} >$ the following are true,*

1. *E_t is a subset of E_I constructed by removing any instruction execution instance from E_I that is not an instance of an instruction of I, such that $E_I - E_t = E_{I-t}$ (possibly empty),*
2. *$\rightarrow_{T_t}$ is a subset of $\rightarrow_{T_I}$ such that $\rightarrow_{T_I} - \rightarrow_{T_t} = \rightarrow_{T_{I-t}}$ (possibly empty),*
3. *P is a higher level view of the execution $< E_t \cup E_{I-t}, \rightarrow_{T_t} \cup \rightarrow_{T_{I-t}} >$.*

□

A trace view of a program execution may not be a valid program execution. A trace view obeys Axioms 1-3 however, may fail to obey Axioms 4 and 5. A trace view is *valid* if it is a trace view of a valid program execution. A valid trace view of a program execution is called a *trace* of that program execution.

4 A Categorization of Execution Pattern Distortion

In this subsection, we propose a categorization of program statements (and instructions) based on the ability of that statement (or instruction) to change the execution pattern of the program (i.e., the degree to which a statement (or instruction) contributes to execution pattern distortion). This categorization has some relationship[12] to the categorization of Larus [38].

A program execution contains only the state information that is embodied in its execution path. That is, the details of the program state present at any point during program execution are not in general present in an execution. Instead, only control flow information is present. This leads us to a notion of program determinism and nondeterminism that differs from traditional definitions. We define the notions of *execution pattern determinism* and *execution pattern nondeterminism* for a parallel program, which we also call *pattern determinism* and *pattern nondeterminism* respectively. These definitions hold for parallel programs that have a single execution pattern or many execution patterns respectively.

In this Section, we demonstrate that for programs in our simple language (even excluding the iterative construct) the problem of determining statically whether or not a program is pattern determinate or pattern nondeterminate is NP-hard. An ancillary result is that the problem of accurately determining the set of Trace Change Points for a program is an NP-hard problem.

4.1 A Taxonomy for Execution Pattern Distortion

An intermediate code statement can be classified as static, runtime, dynamic, or externally dynamic based on whether or not its outcome can be determined statically, is fixed after one run, varies over subsequent runs (on the same data set), or varies over subsequent runs due to externally introduced nondeterminism (respectively). Since a statement is comprised of a sequence of instructions, we will also classify an instruction as static, runtime, dynamic, or externally dynamic.

An execution can be viewed as either a sequence of statement execution instances or a sequence of instruction execution instances. A particular statement can be viewed as a sequence of instructions. The *execution contribution* for a statement is an instance of its associated instruction(s). For example, the statement $x = 1$, which translates into the instruction *store* $addr(x)1$, has the same outcome regardless of the state in which it is executed. Its execution contribution is a store to $addr(x)$. Some instructions (and statements) have the possibility of a different execution contributions in different states. We return to Figure 3.2 to demonstrate our taxonomy.

[12] Our notions of static and runtime differ from the categorization of Larus who attempts to categorize instructions based on the necessity of instrumenting to collect a trace [38]. Since we begin with an execution, we have instances of instructions and need only categorize with respect to how they might differ over an entire execution.

A *static* instruction is one whose outcome is fixed regardless of the number of times a program is executed or of the current program state (variable values) at the point at which the instruction is executed. These instructions have an execution contribution that is constant, regardless of the program data set or the number of times the statement has been executed. For example, a load or store to a variable has a fixed outcome since the address of the variable is known. The outcome of static instructions can be determined (statically) before program execution. A static statement is one made up of static instructions. In Figure 3.2 , statements D, C, and G are static statements. We distinguish two types of static statements :

Static Statements These are statements comprised of instructions that generate no stores to shared variables.

Static Affecting Points These are static statements that contain a store to a shared variable.

The static affecting points are distinguished from static statements because they may affect a change in the execution pattern for a program. However, their execution contribution can still be determined statically.

A *runtime* instruction is one whose outcome is fixed after a single execution of the program for each instruction execution instance. That is, the execution contribution of a runtime instruction may vary from execution instance to execution instance but the overall contribution is a function of the input data set. A runtime statement is a statement made up of static instructions and at least one runtime instruction. Note that runtime instructions and hence runtime statements have no dependency on the value of shared variables. Array and pointer accesses that depend on values computed in the program are runtime instructions (e.g., statements that produce instructions like *load R1, array + R2*) whenever they have no dependency on the values of shared variables. All runtime statements can theoretically be determined statically except[13] for statements that result in indirect addressing instructions (i.e., statements like *load R1, 4(R2)*). Runtime statements fall into three categories :

Deterministic Conditional Statements These statements determine the *deterministic* flow of control for the program (i.e., do not contain or depend on shared variables).

[13] It should be noted that the cost of determining these instructions statically may be prohibitive.

Runtime Dependent Statements These statements depend on a runtime determined value (e.g., index, pointer and arithmetic statements dependent on computed values).

Runtime Affecting Points These statements are runtime statements that generate a store to a shared variable.

A *dynamic* instruction is one whose execution contribution depends on program nondeterminism. That is, the instruction has more than one possible outcome due to dependence on the value of a shared variable (either directly or indirectly). Unlike runtime instructions, the execution contribution of a dynamic instruction is not a function of the input data set. Rather, the outcome (or occurrence) of an execution instance for a dynamic instruction may vary on subsequent runs with the same data set. A dynamic statement is one made up of static and/or runtime instructions and at most one dynamic instruction[14]. For example, a conditional statement whose outcome depends on the value of a shared variable is a dynamic statement. Dynamic statements fall into three categories:

Nondeterministic Conditional Statements These are statements that define the *nondeterministic* flow of control for the program (e.g., conditional statements dependent on shared variables)

Dynamic Dependent Statements These statements are dependent on the outcome of a shared variable (e.g., index, pointer and arithmetic statements).

Dynamic Affecting Points These statements are dynamic statements that generate a store to a shared variable.

We call dynamic affecting points, runtime affecting points, and static affecting points *trace affecting points* (TAPs) because these are exactly the points in a program that can *affect* a change in the execution pattern of a program trace.

A fourth category of instructions is needed for task control statements if the tasks created by a task control statement can be scheduled dynamically. This is because dynamic task scheduling has the potential to introduce nondeterminism into a program. We categorize task control statements

[14]This is due to the restrictions on three address intermediate code given in Section 3.

as *externally dynamic.*

Task Control Statements These are statements that contain a **task_create** statement.

Externally dynamic statements can also be dynamic statements if they depend upon the value of a shared variable.

Nondeterministic conditional statements and dynamic dependent statements are called *trace change points* (TCPs)[15] because these are exactly the statements whose execution can result in a change in execution pattern for a program. Notice that the **lock** statement is a dynamic statement (and a TCP) because it is dependent on the value of a shared variable (i.e., a lock value). The **unlock** statement is a static statement (a static affecting point) since its outcome is fixed. Returning to Figure 3.2, statements B, E, and G are dynamic statements. Statements B and G depend on the value of a shared variable for their outcome. Statement E depends on the value of shared variable *index* for the address of the array element to be loaded and subsequently stored into *aval* or *bval*.

The relationship between trace affecting points and trace change points is that trace affecting points affect a change in the state of the program that *may* cause a change in the execution pattern of the program but only at a trace change point. [16] To see this relationship, recall the notion of a use-definition chain, or ud-chain, and the notion of a reaching definition from compiler theory (see [3] for details). A *reaching definition* for a use of a variable is any definition of the variable that reaches the use. A *ud-chain* for a variable is one of a set of chains of reaching definitions for that variable. Any definition of a shared variable occurring along the ud-chain for a variable used in a TCP is a TAP. Using subscripts to denote processor numbers, in Figure 3.2, statement B_0, a TCP, has two use-definition chains: (A_1) and (D_1, A_0). The TAPs for TCP B_0 are A_0, A_1, and D_1.

4.2 Execution Pattern Determinism and Nondeterminism

Using the instruction taxonomy, we now distinguish programs as deterministic or nondeterministic with respect to execution pattern distortion. A program is *path deterministic* if it contains a single thread of execution. Sequential programs are path deterministic. A program is *path nondeterminis-*

[15] Holliday and Ellis, in [32], define *Address Change Points.* The difference is that ACPs refer to the statements at which actual changes in the address trace occur. For conditional statements, we use TCP to refer to the actual conditional statement. In addition, Holiday and Ellis use medium grained processes which do not use process control statements (e.g., **task_create**).

[16] This is the difference between race detection and trace extrapolation. Race detection is the problem of locating trace affecting points while trace extrapolation is the problem of deciding the outcome of trace change points.

tic if it contains multiple threads of execution (i.e., a program that has the possibility of interleaved events).

Intuitively, a program exhibits execution pattern determinism if it does not suffer from execution pattern distortion (i.e., the program contains no TCPS).

Definition 4.1 *A parallel program is* execution pattern deterministic *if it contains only static, runtime and externally deterministic statements that are not also dynamic statements.*

□

Pattern determinism should not be confused with the traditional notions of program *data determinism.* Consider an execution that is data nondeterministic (i.e., yields different possible resulting values over subsequent executions of the program given the same data set) but that contains no TCPs. This program is pattern deterministic since there is a single execution pattern for the program.

There are other classifications of program nondeterminism but these classifications deal strictly with data determinism and nondeterminism. A widely used classification of program data nondeterminism has been proposed by Emrath and Padua [21]. Programs are either deterministic or nondeterministic and there are two subcategories in each of these categories. Deterministic programs are either *internally determinate* – the program can be serialized, or *externally determinate* – the program computes a function regardless of the possibility of race conditions. Nondeterministic programs are either nondeterministic *solely* due to round off errors that occur due program execution or exhibit *true* nondeterminism.

A parallel program is execution pattern nondeterministic if executing the program on the same data set multiple times can result in more than one execution pattern (i.e., the program contains TCPs).

Definition 4.2 *A parallel program is* execution pattern nondeterministic *if the program contains at least one TCP.*

□

To distinguish this notion of nondeterminism from the traditional notion, we also call a parallel program that is execution pattern nondeterministic *pattern nondeterministic* and a program that is nondeterministic in the traditional sense *data nondeterministic.*

4.3 Locating Trace Change Points

The problem of statically determining whether or not a program is pattern deterministic or pattern nondeterministic is an NP-hard problem, because event orderings cannot, in general, be determined statically. An ancillary result proves that the set of TCPs for a program cannot be exactly determined.

In this subsection, we present a variation of a proof presented in [43] that demonstrates that the problem of determining statically whether or not a program is pattern deterministic or pattern nondeterministic is NP-hard.

Recall that a problem is in NP, or said to be NP-complete, if there exists a polynomial time transformation transforming an arbitrary instance of the problem to some problem in NP. If problems in NP can be solved in polynomial time, then so can the transformed problem. If instead, a problem that is known to be in NP can be transformed, in polynomial time, to another problem, then the latter problem is referred to as NP-hard. That is, the problem is at least as hard as the problem known to be in NP. See [25] for more details on the theory of NP-completeness.

The problem of Three Conjunctive Normal Form Boolean Satisfiability (3CNFSAT) [25] is known to be in NP. In this subsection, we present a proof that reduces 3CNFSAT to the problem of locating TCPs in a source program. Therefore, since 3CNFSAT is an NP-complete problem, the problem of locating TCPs in a source program is an NP-hard problem. Specifically, we will transform an arbitrary instance of the 3CNFSAT problem, an NP-complete problem, to an instance of the problem of locating TCPs in a source program and this transformation will require at most polynomial time.

3CNFSAT is a subproblem of the problem of determining Boolean Satisfiability (SAT) . In 3CNFSAT, logical variables appear in a boolean expression that is a conjunction of clauses. Each clause is itself a boolean expression that is the disjunction of three variables (or their negation). For example, $a \wedge (b \vee c) \vee (\neg b \wedge d)$ and $(a \vee b \vee \neg d) \wedge (c \vee e \vee d)$ are both boolean expressions and therefore are both SAT problem instances. The first boolean expression is not a 3CNFSAT instance. The second is a 3CNFSAT instance because the expression is a conjunction of two clauses $(a \vee b \vee \neg d)$ and $(c \vee e \vee d)$ and each of these clauses contains a disjunction of three variables or their negation. A possible solution to a boolean expression (or to a SAT or 3CNFSAT problem instance), is an assignment of TRUE or FALSE values to each variable. The expression is satisfied if it is TRUE given the variable assignment. Each member of a clause is called a *literal* (i.e., variables or their negation). For example, in the previous expression, $\neg d$ and c are literals. A clause is TRUE if a single literal in the clause is TRUE. A set of variable assignments is a solution to a boolean expression, if each clause of the expression yields TRUE under the given variable assignment.

We proceed by constructing a program that solves a 3CNFSAT problem in two phases. In the first phase, locks are used to simulate the nondeterministic choice of a possible solution to the

3CNFSAT problem. In the second phase, the solution is *checked* for correctness. Given a lock for each variable and also for each variable's negation, each initially locked, the competition proceeds by either the variable or its negation being unlocked. This represents choosing a value of TRUE for a variable or its negation (respectively). Once all values are chosen, the second phase proceeds. In this phase, the outcome of a particular clause is represented by a lock, initially locked. Access to unlock each clause is guarded by the literals (i.e., the lock corresponding to the negation of a variable or a variable) for the clause. Any literal assigned to TRUE in the first phase, will result in the clause being unlocked. Finally, a single process attempts to lock each clause representing a solution to the problem. A second *clean-up* pass unblocks all program locks (allowing the program to terminate normally). If in some execution, every clause is locked before this second pass begins, then there is a solution to the 3CNFSAT instance, otherwise there is no solution.

This transformation represents a transformation from an arbitrary 3CNFSAT problem instance to an instance of our problem such that the resulting program is pattern nondeterministic if and only if, there is a solution to the arbitrary 3CNFSAT instance. Specifically, a solution to the 3CNFSAT problem will exist only if a TCP branches TRUE. If the TCP statement fails (i.e., branches FALSE), then there is no solution to the 3CNFSAT problem instance. For the proof, we assume that a lock that is set to 0 is locked, and a lock that is set to 1 is unlocked.

Theorem 4.3 *Given a parallel program in our intermediate language, the problem of determining statically whether or not the program is pattern nondeterministic is NP-hard.*

□

Proof: We reduce an arbitrary instance of 3CNFSAT into an instance of a program in our language such that the instance of 3CNFSAT is satisfied only if a conditional statement is a TCP. Given a set of variables, $X_1, X_2, \ldots, X_n$ and a set of clauses $C_1 \vee C_2 \vee \ldots \vee C_m$ where each clause is constructed from the disjunction of three literals (variables or their negations), we reduce this instance of 3CNFSAT to a program containing $4n + m + 1$ locks and $3n + 3m + 2$ processes.

```
begin
    shared int x = 0
    shared locks
        X[1..n] = 0,
        Xnot[1..n] = 0,
        Mutex[1..n] = 0,
        pass[1..n] = 1,
```

```
        C[1..m] = 0,
        test = 1

    task_create(3n + 3m + 2)
...
    task_terminate(3n + 3m + 2)

end
```

The program proceeds in two phases. The first $3n$ processes choose values for the X_i. For each X_i we have three processes:

```
lock(Mutex[i])     lock(Mutex[i])      unlock(Mutex[i])
unlock(X[i])       unlock(Xnot[i])     lock(pass[i])
                                       unlock(Mutex[i])
```

In the first phase, the first two processes for each X_i compete to determine the value of the variable. The third process allows the initial competition and then in a second phase, unblocks the n processes still waiting from the initial competition.

The next $3m$ processes represent the C_j clauses. For each clause C_j we have 3 processes where L_i represents the ith literal of the clause:

```
lock(L1)           lock(L2)            lock(L3)
unlock(C[j])       unlock(C[j])        unlock(C[j])
unlock(L1)         unlock(L2)          unlock(L3)
```

If a literal is unlocked by the value selection phase, then the literal accesses the lock, unlocks its clause, and unlocks the literal again (so that all other clause processes see the value of this literal as TRUE).

The final two processes represent a correct solution to the 3CNFSAT problem and the start of the unblocking phase in the event of a correct or incorrect solution.

```
lock(C[1])          lock(test)
```

```
.                      z = x
.                      unlock(test)
.                      if z then
lock(C[m])             a: skip
lock(test)             unlock(pass[1])
b: x = x + 1           .
unlock(test)           .
                       .
                       unlock(pass[n])
```

The first process attempts to acquire all of the clause locks. Once this has occurred, the lock *test* is set and the value of the shared variable x is incremented. The second process first locks *test* and stores the value of x in a local variable z. If z has a value of one (i.e., is TRUE) then the statement a executes. In either case, each of *pass*[i] are subsequently unlocked starting the unblocking phase.

To demonstrate that the reduction is valid, recall that the conditional statement is a TCP, if it has the possibility of branching differently depending on the value of a shared variable. Statement a is only executed if the 3CNFSAT instance is satisfiable. That is, if the 3CNFSAT instance is satisfiable, then the conditional statement is a TCP. Otherwise, it always fails and is not a TCP. To see this, consider that the value of x (and z) is only one in the case that statement b executes before statement a. This can only occur if each of the clauses of the 3CNFSAT instance yield true allowing each of the lock statements preceding statement b to acquire their locks. This only occurs if the statement is satisfied. Since the conditional statement is the only statement in the program that has the possibility of being a TCP, the program is pattern nondeterministic only if the 3CNFSAT instance is satisfiable.

□

As a direct result of Theorem 4.3 we have Corollary 4.4:

Corollary 4.4 *The problem of determining the set of TCPs for an arbitrary parallel program is NP-hard.*

□

4.4 Execution Structuring Assumptions

For the remainder of the thesis it will be useful to view TCPs as a means of structuring executions. For this reason, we now include the program **begin** statement and any **task_create** statements in the set of TCPs for a program. We assume that external nondeterminism is not possible. Therefore,

adding these statements does not affect whether or not a program is pattern deterministic or nondeterministic.

5 The Structure of Parallel Program Executions

5.1 Motivation

In this Section, we demonstrate that given well structured programs (assumption A2), executions from the same program on the same data set exhibit periods of divergence and convergence. TCPs are the points at which executions have the potential to diverge. We introduce the notion of *convergence points* as instructions at which divergent executions *must* converge.

The existence of convergence and divergence points imply that two executions can be *pinned* together at convergence subsections. This pinning locates the differences between two executions. In Section 10, the pinning provides a means of corresponding executions used for measurement of metrics over two executions. In this subsection, we present the theoretical background for correspondence.

```
A0: begin
    shared int x = 0;
    shared int lk = 0;
    int n = 2;
A1: task_create(n)

task 0                          task 1
    int j = 0;                      int sum = 0;
    int k = 0;                      int index = 0;
                                    int i = 0;
B0: lock(lk);                   C0: lock(lk);
B1: x = x + 1;                  C1: x = x + 1;
B2: j = x;                      C2: index = x;
B3: unlock(lk);                 C3: unlock(lk);
B4: if j == 2 then              C4: for i = 1 to index do
B5:     j = j - 1;              C5:     sum = sum + i;
B6: else
B7:     k = k + 1;

A2: task_terminate(n);
A3: end;
```

Figure 14: A parallel program exhibiting multiple execution patterns

To illustrate these ideas, consider the example in Figures 14 and 15a-c. The parallel program in Figure 14 contains two processes that first compete for access to a shared variable and then execute a conditional statement B4 and an iterative statement C4 that are dependent on this shared value.

Figure 15 a demonstrates that this dependency can lead to different possible executions depending on the interleaving of the shared variable access operations. Notice that in the first execution, the value of j is 2 after statement B3 completes and the conditional statement is true, while the iterative statement C4 iterates a single time. In the second execution, the value of j is 1 and the condition fails, while the iterative statement C4 iterates twice. Notice that this results in an increase in the length of the second execution in Figure 15 a and represents an insertion of additional elements. Figure 15 demonstrates how we correspond these executions. First, statements A0-A1 correspond in both executions (1), as with statements A2-A3 (8). Statements C0-B3 and B0-C3 correspond in both executions (2) because the same statements are being executed. We correspond the first iteration of the iterative statement C4 (3) and the second check on the index value (6). However the second iteration of the loop in execution 2 (7) has no correspondence to the first execution (and represents an insertion of an additional loop iteration). The instances of the conditional statement B4 correspond in each execution (4) however B5 and B6-B7 do not correspond but must be compared when calculating a measure (5). We will return to Figure 15 c shortly.

5.2 Paths

We define the notion of a path in a program execution. Paths will play an important role in formal discussions of executions. Intuitively, a path represents a subset of an execution that is contiguous in time (i.e., a subsequence of an execution). For example, execution 1 in Figure 15 c contains paths from C0 to C2 and from C0 to C4[17].

Definition 5.1 *Given a program execution* $P = \langle E, \rightarrow_T \rangle$ *a* path *of* P *is a pair* $p(h,t) = \langle E_{(h,t)}, \rightarrow_{T_{(h,t)}} \rangle$ *such that*

1. $E_{(h,t)}$ *is a non empty subset of* E*,*
2. $\rightarrow_{T_{(h,t)}}$ *is a subset of* $\rightarrow_T$*,*
   such that the following are true for any $a, b \in E_{(h,t)}$*,*
   - $a \rightarrow_T b$ *if and only if* $a \rightarrow_{T_{(h,t)}} b$*,*
   - *and* $a \rightarrow_{T_{(h,t)}} c$ *and* $c \rightarrow_{T_{(h,t)}} b$ *if and only if* $c \in E_{(h,t)}$*,*
     with the following conditions on h *and* t*,*
     - $h \in E_{(h,t)}$ *and there does not exist* $a \in E_{(h,t)}$ *such that* $a \rightarrow_{T_{(h,t)}} h$*,*
     - *and* $t \in E_{(h,t)}$ *and there does not exist* $a \in E_{(h,t)}$ *such that* $t \rightarrow_{T_{(h,t)}} a$*.*

□

[17]There are many other possible paths

Execution 1	Execution 2
A0	A0
A1	A1
C0	B0
B0	C0
C1	B1
C2	B2
C3	B3
B1	C1
B2	C2
B3	C3
C4	C4
C5	C5
B4	B4
B5	B6
C4	B7
A2	C4
A3	C5
	C4
	A2
	A3

Figure a. Two possible executions.

	Execution 1	Execution 2
(1)	A0	A0
	A1	A1
(2)	C0	B0
	B0	C0
	C1	B1
	C2	B2
	C3	B3
	B1	C1
	B2	C2
	B3	C3
(3)	C4	C4
	C5	C5
(4)	B4	B4
	B5	B6
(5)		B7
(6)	C4	C4
(7)		C5
		C4
(8)	A2	A2
	A3	A3

Figure b. Correpondence of executions.

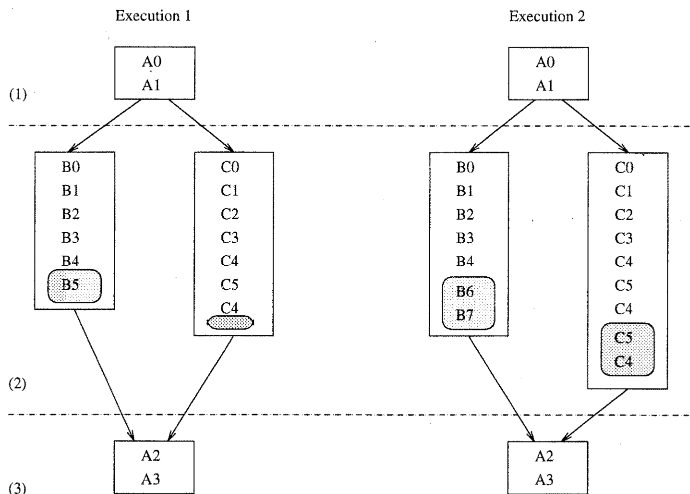


Figure c. Graph of executions.

Figure 15: (a,b,c) Possible executions, correspondence, the graph executions of the program in Figure 5.1

h and t are the *head* and *tail* respectively of the path. A *prefix* of an execution is any valid path such that h is the initial instruction for the program. Likewise, a *suffix* of a program execution is any valid path such that t is the final instruction for the program. In Figure 15 c, execution 1 has the path p(A0,B1) which is a prefix and the path p(C2,A3) which is a suffix.

We define two paths to be equivalent whenever they are the *same* path and are disjoint whenever they have no equivalent instruction instances in common (where equivalence for instruction instances is defined by assumption A3). In Figure 15 c, the paths p(B0,B4) are equivalent in both executions. The paths p(B0,A2) in execution 1 and p(B0,A2) in execution 2 are not equivalent and the paths p(B1,B5) in executions 1 and p(B6,B7) in execution 2 are disjoint.

Definition 5.2 *Two paths* $p(h_1, t_1) =< E_{(h_1,t_1)}, \rightarrow_{T_{(h_1,t_1)}} >$ *and* $p(h_2, t_2) =< E_{(h_2,t_2)}, \rightarrow_{T_{(h_2,t_2)}} >$

1. *are* equivalent, *denoted* $p(h_1, t_1) = p(h_2, t_2)$ *whenever*

   $E_{(h_1,t_1)} = E_{(h_2,t_2)}$

   $(a_1, b_1) \in \rightarrow_{T_{(h_1,t_1)}} \Leftrightarrow (a_2, b_2) \in \rightarrow_{T_{(h_2,t_2)}}$ *such that* $a_1 \equiv a_2$ *and* $b_1 \equiv b_2$*,*

2. *are* disjoint*, whenever* $E_{(h_1,t_1)} \cap E_{(h_2,t_2)} = \emptyset$*.*

□

The *successor* of an instruction instance e in an execution is the instruction instance that immediately follows e in time (in the execution). Likewise, the *predecessor* is an instruction instance e is the instruction instance that immediately preceded the instruction instance in time (in the execution). In Figure 15 c, the successor of instruction instance B5 in execution 1 is instruction instance A2 and the predecessor of instruction instance B5 is instruction instance B4.

Definition 5.3 *Given an execution* $P =< E, \rightarrow_T >$ *for any event e,*

1. *the* successor *of e is an event s such that there exists a path* $p(e, s) =< E(e, s), \rightarrow_{T_{(e,s)}} >$ *such that* $E_{(e,s)} = \{e, s\}$*.*

2. *the* predecessor *of e is an event p such that there exists a path* $p(p, e) =< E_{(p,e)}, \rightarrow_{T_{(p,e)}} >$ *such that* $E_{(e,s)} = \{p, e\}$*.*

□

5.3 The Graph of an Execution

For the purposes of comparing executions, we propose simplifying a program execution by removing parallel thread interaction. Consider the example in Figures 14. For the two executions in Figure 15 a we can remove parallel thread interaction resulting in the comparison in Figure 15 c. Here, the processes associated with tasks 1 and 2 in each execution are matched (2) and the main thread of the executions (1 and 3) are matched. In this example, statements B0-B4 correspond in each execution because each execution executes these statements with the same outcome. The first difference appears after the condition at B4 branches true or false. The highlighted box signifies that statements B5 in execution 1 and B6 and B7 in execution 2 are different and must be measured for difference. Similarly, the dark highlighted box in execution 2 with corresponding empty highlighted box in execution 1, signifies that there are no corresponding statements for this extra iteration. Notice that information about interleaving is irrelevant.

Removing parallel thread interaction from an execution accentuates the structure of the execution. Thread interaction serves only as a record of the *cause* (an interleaving of instruction instances) of a change in the execution path. Since we are interested only in the actual changes that occur from one execution to another, this record of the cause of change is not important. Removing all parallel thread interaction, from an execution, results in the *graph* of an execution. Figure 15 c depicts the graph of executions 1 and 2 of Figure 15 a.

Definition 5.4 *Given a program execution* $P = < E, \rightarrow_T >$ *the* graph *of* P, $P_g = < E, \rightarrow_{T_g} >$ *is the result of the following transformation on* P*:*

> *For* $\rightarrow_T$ *perform the following resulting in* $\rightarrow_{T_g}$
> *for each* $a, b \in E - (\mathbf{task_terminate(E)} \cup \mathbf{task_create(E)})$ *such that* $a \rightarrow_T b$, *if* $a \in P_i(E)$ *and* $b \in P_j(E)$ *and* $i \neq j$, *remove* (a, b) *from* $\rightarrow_T$.

□

Definition 5.4 simply removes all members of the temporal and shared data dependence relations that relate events from different threads of execution. For the remainder of this chapter, we use the term execution to refer to the graph of an execution.

5.4 The Outcome of TCPs

Two TCP event instances can have a different outcome. For example, for a conditional statement TCP, the outcome of the conditional statement can be a branch TRUE or FALSE. In two executions that each have an instance of the same TCP event, the outcome of the events can make the two executions diverge.

Definition 5.5 *Given two executions* $P_1 = < E_1, \rightarrow_{T_1} >$ *and* $P_2 = < E_2, \rightarrow_{T_2} >$ *and events* $b_1 \in TCP(E_1)$ *and* $b_2 \in TCP(E_2)$ *that are comparable, the outcome of the events in each execution are* outcome equivalent, *denoted* $Outcome(P_1, b_1) = Outcome(P_2, b_2)$ *if one of the following is true:*

1. *if the TCP is a conditional instruction (then* $b_1 \equiv b_2 = b$*) and there exists* $c \in E_1 \cap E_2$ *such that* $b \rightarrow_{T_1} c$ *and there does not exist* $a \in E_1$ *or* $d \in E_2$ *such that* $b \rightarrow_{T_1} a$ *and* $a \rightarrow_{T_1} c$ *or* $b \rightarrow_{T_2} d$ *and* $d \rightarrow_{T_2} c$.
2. *if the TCP is a* **task_create**(p) *instruction (then* $b_1 \equiv b_2 = b$*) and for each* $c \in E_{i,1}, i = 1$ *to* p, *such that* $\{b \rightarrow_{T_1} c \wedge \neg(\exists d | d \in E_{i,1} \wedge b \rightarrow_{T_1} d \wedge d \rightarrow_{T_1} c)\}$ *implies* $\{b \rightarrow_{T_2} c \wedge \neg(\exists d | d \in E_{i,2} \wedge b \rightarrow_{T_2} d \wedge d \rightarrow_{T_2} c)\}$
3. *if the TCP is an array reference and* $b_1 \equiv b_2$
4. *if the TCP is the* **begin** *statement of the program.*

□

Two instances of a conditional instruction have the same outcome if (for the process in which they are contained) they are immediately followed by equivalent events. Two **task_create** instructions have the same outcome if they result in the production of the same threads of the same processes (i.e., the same number of threads with the first instruction executed in each thread being the same). For array reference instructions, if the events are the same, then their outcome is the same (e.g., an array reference TCP that results in the event **load** *addr* will either have the same address reference in both executions or the outcome of the TCP is not the same). Finally, the **begin** statement for an execution always has the same outcome.

5.5 The Structure of Executions

Executions have periods of *convergence*, at which they are equivalent (i.e., they execute the same code), and of *divergence*, at which they are disjoint (i.e., they execute completely different subsections of code). Returning to the example of Figure 15 c, the highlighted areas represent periods of divergence between the two example executions. Execution 1 executes statement B5 while execution 2 executes statements B6 and B7, and execution 2 executes a second iteration of the loop while execution 1 executes no additional statements. We formally define divergence and convergence for executions in Definition 5.6.

Definition 5.6 *Given two program executions* $p(h_1, t_1) =< E_{(h_1,t_1)}, \rightarrow_{T_{(h_1,t_1)}} >$ *and* $p(h_2, t_2) =< E_{(h_2,t_2)}, \rightarrow_{T_{(h_2,t_2)}} >$ *with two fixed but arbitrary paths* $p(h_1, t_1) =< E_{(h_1,t_1)}, \rightarrow_{T_{(h_1,t_1)}} >$ *and* $p(h_2, t_2) =< E_{(h_2,t_2)}, \rightarrow_{T_{(h_2,t_2)}} >$ *(respectively),*

1. *there is a* divergence point instance d *if* $p(h_1, t_1) \neq p(h_2, t_2)$*, the predecessor of* t_1 *is* c_1*, the predecessor of* t_2 *is* c_2*, the paths* $p(h_1, c_1)$ *and* $p(h_2, c_2)$ *are equivalent, and* $c_1 \equiv c_2$ *is the divergence point* d*.*
2. *there is a* convergence point instance d *if* $p(h_1, t_1) \neq p(h_2, t_2)$ *and they are not disjoint, the predecessor of* t_1 *is* c_1*, the predecessor of* t_2 *is* c_2*, the paths* $p(h_1, c_1)$ *and* $p(h_2, c_2)$ *are disjoint, and* $t_1 \equiv t_2$ *is the convergence point* d*.*

□

Intuitively a divergence point occurs whenever two paths are equivalent up to some instruction instance and then execute different instruction instances or have a different outcome on the same instruction instance. A convergence point occurs whenever two paths are disjoint, or completely different, up to some instruction instance and then execute an instance of the same instruction with the same outcome.

The notions of divergence and convergence in executions are critical to our ability to correspond executions with reasonable time and space usage. Divergence points for executions of parallel programs were first characterized in [32] and further studied in [17] [16] and are *trace change points.* Notice that statement B4 in Figure 14 is dependent on the value of j which is dependent on the value of shared variable x making statement B4 a TCP. Similarly, statements B0, C0, and C4 are TCPs.

The problem of detecting a TCP instance in an execution is that of locating the TCP instance and determining if divergence has occurred. Reference TCPs complicate the correspondence process because it is possible for the CP of a TCP instance to differ in two possible executions (i.e., if the CP of a conditional or reference TCP is itself a reference TCP). In practice, as an execution is traversed, the number of instructions between a TCP and its CP is known. Hence, it is possible to determine if two different reference events represent the same instruction instance. However, this location process is uninteresting in the following discussion and can be ignored for clarity. Therefore, we assume that reference TCPs each have a **no-op** instruction marking the TCP. That is, each reference TCP has a **no-op** instruction instance, followed by the reference instruction instance. This assumption guarantees that a period of divergence is caused by a single TCP and that convergence will occur before the beginning of the next period of divergence.

We use the notation $TCP(E)$ to denote the set of TCP instances for an event set E. It will be necessary to refer to the sequence of TCPs executed by a program during a particular execution. We use the notion of the *TCP successor* of a TCP instance and *TCP predecessor* of a TCP instance in referring to this sequence. The TCP successor (and predecessor) of a TCP instance is the next (previous) TCP instance that occurs in the graph of a program execution. More formally,

Definition 5.7 *Given a program execution* $P = < E, \rightarrow_T >$ *for each instruction instance* $a \in TCP(E)$ *there exist* $s, d \in TCP(E)$*, the TCP successor and TCP predecessor of* a *(respectively), such that*

1. *Either* a *is the initial statement or there exists a path* $p(d,a) = < E_{(d,a)}, \rightarrow_{T_{(d,a)}} >$ *such that* $(E_{(d,a)} - \{d,a\}) \cap TCP(E) = \emptyset$
2. *Either* a *is the final statement or there exists a path* $p(a,s) = < E_{(a,s)}, \rightarrow_{T_{(a,s)}} >$ *such that* $(E_{(a,s)} - \{a,s\}) \cap TCP(E) = \emptyset$ *and*

□

We extend the work of [32] and [17] by defining the set of convergence points (CPs) for a parallel program execution. A CP is defined to be the *first* instruction or statement that any program execution *must* execute after a period of divergence. Lemma 5.8 below demonstrates the convergence points exist and shows that convergence points are continuations of either conditional statements, iterative statements, **lock** statements, or array and pointer reference statements. We use the notation $CP(E)$ to denote the set of CP instances in an event set E. We use the notation $TCP(P)$ and $CP(P)$ to denote the set of TCPs and CPs (respectively) for program P.

Lemma 5.8
Any program P with reduced TCP set $T = TCP(P) - \{\textbf{task_create}, \textbf{begin}\}$ has a set of convergence points denoted $CP(P)$ each of which is either the continuation of an iterative, conditional, **lock**, *or array reference statement.*

□

Proof: With the **task_terminate** and *end* statements removed from the set of TCPs of a program, the set of TCPs are either array reference, iterative,**lock**, or conditional statements. Hence, it suffices to show that there is a CP for each. There are four cases:

case 1 Without loss of generality, consider the TCP containing statement a[i] = c. Any execution instance of this statement will end with an instance of the instruction **ld** @a+i which represents a possible point of divergence for any two executions. Any execution in which the statement is executed must execute the continuation of the array reference statement (the statement immediately following it) immediately following the **ld** instruction. If this statement is not itself an array reference TCP, then is a CP. A special case occurs in the event that the continuation of an array reference TCP is itself an array reference TCP. Since the program cannot have an infinite string of array references that are TCPs, the CP for each TCP in this sequence is the first statement in the sequence that is not and array reference TCP.

case 2 Without loss of generality, consider a conditional TCP containing statement.

```
if (c) then
    a
else
    b
d
```

Any two executions in which this statement was executed must have the branch on c in common and therefore be subject to a possible divergence. Any two execution where the outcome of c is different diverge (since a and b are disjoint by Assumption 3) must execute d immediately following

the execution of either a or b since by Assumption 1, the program is structured. Therefore the continuation of a conditional statement, d, is a CP. Once again, there is a special case whenever the continuation for a conditional statement is an array reference TCP. In this case, the continuation for the conditional statement is the continuation of the array reference TCP (see case 1).

case 3 Because a **lock** statement does not change the program flow of control, it follows from case 1 that each **lock** statement has a CP.

case 4 Because an iterative statement is a repetitive branching construct, case 3 follows from case 2.

□

The intuition for this proof is that given the assumption that programs are structured, any statement (conditional, iterative, **lock**, or referencing) has single entry and exit point. Therefore each statement has a single continuation and this point is a CP if the statement was itself a TCP.

For structuring purposes, we add the program **end** statement and all **task_terminate** statements to the set of CPs for a program corresponding to the **begin** and **task_create** statements (respectively).

Definition 5.9 *The* **end** *statement and any* **task_terminate** *are members of the set of convergence points for a program. These convergence points correspond to the* **begin** *statement TCP and to the appropriate* **task_create** *statement (respectively).*

□

Given Lemma 5.8 and Definition 5.9, it follows that there exists a correspondence between program TCPs and CPs such that for every program TCP, there exists a corresponding program CP.

Corollary 5.10 *For any program P, there exists a correspondence between the sets $TCP(P)$ and $CP(P)$ and this correspondence is an onto mapping.*

□

Notice that we cannot claim that this mapping is one-to-one since some TCPs will share the same CPs. For example, nested conditional statements will share the same CP.

While Lemma 5.8 and Definition 5.9 identify the correspondence between program TCP and CP pairs, it will be necessary to locate these instance pairs in an execution. In particular, we want to find the *correct* CP instance for every TCP instance. Consider the example of a conditional statement containing a TCP. If the TCP is the conditional instruction itself, then we do not want to match, or *correspond*, an instance of this instruction with an instance of its CP that is a result of a later execution of the conditional statement. Simply put, we want to match the correct TCP and CP instances. Lemma 5.11 demonstrates that there exists a path between a TCP instance and its corresponding CP instance and that there are no further instances of either the CP or TCP on the path. This is a direct result of Assumption A2.

Lemma 5.11 *Given a program execution* $P =< E, \rightarrow_T >$ *and given an* $s \in TCP(E)$ *there exists* $c \in CP(E)$ *if* s *and* c *correspond, then there exists a path* $p(s,c) =< E_{(s,c)}, \rightarrow_{T_{(s,c)}} >$ *and there does not exist* $s' \in TCP(E_{(s,c)})$ *such that* $s' \equiv s$ *nor does there exist* $c' \in TCP(E_{(s,c)})$ *such that* $c' \equiv c$.

□

Proof: The lemma is trivially true for the program **begin** and **end** TCP and CP pair (since there is only a single instance of each). We demonstrate that such a path exists for the remaining TCP and CP pairs and that the path is the shortest path between a TCP instance, call this instance s, and an instance of its corresponding CP, call this instance c. Let $p(s,c) =< E_{(s,c)}, \rightarrow_{T_{(s,c)}} >$ be this path. Suppose that the lemma is not true. Then there must be either an instance of the TCP that s is an instance of, call this instance s', or an instance of the CP that c is an instance of, call this instance c', or both on the path from s to c. Clearly if c' exists and $c' \in E_{(s,c)}$ then $p(s,c)$ is not the shortest path from s to an instance of its CP ($p(s,c')$ is the shortest path). Consider that $s' \in E_{(s,c)}$. If this is the case then the program in question violates Assumption A2 since there is no way to reach s' before c except through the use of a **goto** statement. Hence if $p(s,c)$ is the shortest path from s to an instance of its corresponding CP, s' cannot be on the path. Finally, that c is the corresponding CP instance for TCP instance s follows from Assumption A2.

□

Notice that the Lemma implies not only that we can determine the correct TCP and CP instance correspondence in an execution, but also that once a TCP instance is located, it is a simple matter to locate its matching CP instance. In fact, it is simply a matter of scanning the execution for the next instance of the CP for this TCP. Corollary 5.12 follows directly from Lemma 5.11 and Corollary 5.10.

Corollary 5.12 *For any execution of a program there is a correspondence between the TCP instances and CP instances and this correspondence is an onto mapping.*

□

5.6 The Behavior of Executions

Thus far, we defined TCP and CPs in parallel programs. Instances of these points represent divergent and convergent points in actual executions. We have demonstrated that a logical correspondence between TCP and CP exists and that corresponding instance pairs can be easily located. Next we turn our attention to the behavior of convergence and divergence in executions. First, in Lemma 5.13, we demonstrate that whenever two executions have a TCP instance in common and the outcome of those instances are the same, then the paths (in each execution) from the TCP instance to its TCP successor are equivalent.

Lemma 5.13 *Given two program executions* $p(h_1, t_1) =< E_{(h_1,t_1)}, \rightarrow_{T_{(h_1,t_1)}} >$ *and* $P_2 =< E_2, \rightarrow_{T_2} >$ *for any TCP instance* $a \in TCP(E_1) \cap TCP(E_2)$ *with TCP successor* $s_1 \in TCP(E_1)$ *and* $s_2 \in TCP(E_2)$ *respectively. If* $outcome(P_1, a) = outcome(P_2, a)$ *then* $p(a, s_1) = p(a, s_2)$

□

Proof: There are no TCPs other than a and s in either path since if there is an embedded TCP, s is not the successor TCP for a. Given this, there can be no change in execution path from a to s in either execution. Hence $outcome(P_1, a) = outcome(P_2, a)$ implies that $p(a, s_1) = p(a, s_2)$.
□

To see how Lemma 5.13 works, consider the example of Figure 15 c. The paths $P(B0, B4)$ in each execution are an example of equivalent paths where the head of the path, B0, is a TCP which has the same outcome in each execution, and the tail of the path is the TCP successor of $B0$, $B4$.

A similar result for divergence shows that if two executions have a TCP instance in common and the outcome of the instances are different in each execution, then the path from the TCP instance to the corresponding CP instance (in each execution) will be disjoint with the exception of the CP and TCP instances themselves.

Lemma 5.14 *Given two program executions* $p(h_1, t_1) =< E_{(h_1,t_1)}, \rightarrow_{T_{(h_1,t_1)}} >$ *and* $P_2 =< E_2, \rightarrow_{T_2} >$ *for any TCP instance* $a \in TCP(E_1) \cap TCP(E_2)$ *and corresponding CP* $c_1 \in E_1$ *and* $c_2 \in E_2$ *respectively, if* $Outcome(P_1, a) \neq Outcome(P_2, a)$ *then for* $p(a, c_1)$ *and* $p(a, c_2)$, $E_{(a,c_1)} \cap E_{(a,c_2)} = \{a, c\}$ *where* $c_1 \equiv c_2$ *is* c *and* $s_1 \equiv s_2$.

□

Proof: Assume that the lemma is false. First it follows from Lemma 5.11 that $c_1 = c_2$ (they are instances of the same instruction). It must be the case that there is another CP on paths $p(a, c_1)$ and $p(a, c_2)$ that both paths have in common. Then by Lemma 5.10, c is not the corresponding CP for TCP a.

□

To see how Lemma 5.14 works, the path $p(B4, A2)$ in each execution (in the example of Figure 15 c) is an example of a path that begins with a TCP, for which the outcome is different in each execution, and ends with a convergence point (A2). Notice that if we remove the head and tail of this path in each execution we have the paths $p(B5, B5)$ in execution 1 and $p(B6, B7)$ in execution 2 which are disjoint.

Finally it follows that whenever two executions have a CP instance in common, then the paths from that CP to the next TCP in each execution are equivalent.

Corollary 5.15 *Given two executions* $p(h_1, t_1) =< E_{(h_1,t_1)}, \rightarrow_{T_{(h_1,t_1)}} >$ *and* $p(h_2, t_2) =< E_{(h_2,t_2)}, \rightarrow_{T_{(h_2,t_2)}} >$ *if there exists a path in* P_1, $P_1(h, t) =< E_{(h,t)_1}, \rightarrow_{T_{(h,t)_1}} >$ *and a path in* P_2, $p_2(h, t) =< E_{(h,t)_2}, \rightarrow_{T_{(h,t)_2}} >$ *and there does not exist an* $i \in E_{(h,t)_1} \cap E_{(h,t)_2}$ *such that if* $h \rightarrow_{T_{(h,t)_1}} i$ *and* $i \rightarrow_{T_{(h,t)_1}} t$ *or* $h \rightarrow_{T_{(h,t)_2}} i$ *and* $ih \rightarrow_{T_{(h,t)_2}} t$ *then* $p_1(h, t) = p_2(h, t)$.

□

Lemmas 5.11, 5.13, and 5.14 are critical for demonstrating the correctness of an algorithm for corresponding executions. Lemma 5.11 guarantees that an execution can be traversed and that an algorithm will not get lost during the traversal. Lemmas 5.13 and 5.14 guarantee that the behavior of executions is such that they will exhibit periods of convergence (during which they are equivalent) and divergence (during which they are disjoint) and nothing more. In Section 10 we prove that two executions can be traversed making the correct correspondence of TCP instances and CP instances between the executions (i.e., that executions can *pinned* together) and provide an algorithm for correspondence.

6 A Framework for Execution Pattern Distortion

In this chapter we present a formal model of execution pattern distortion in program executions in which to study trace extrapolation. The goal of this model is to characterize sets of executions for which trace extrapolation is tractable. This goal is realized with the use of equivalence relations on program executions. The model is extended to include a formalization of approximation of trace extrapolation.

The model is an extension of the basic model given in Section 3; it is augmented with the shared data dependence relation and the TCP equivalence relation for executions. The shared data dependence relation for an execution is the temporal structure present in the execution with respect to accesses to shared data. The TCP equivalence relation is introduced in this chapter as a means of characterizing executions that represent the same pattern.

Equivalences are categorized in a hierarchy designed to characterize the behavior of parallel programs with respect to the executions that they produce. The hierarchy embodies the notions of pattern determinism and nondeterminism and data determinism and nondeterminism. The hierarchy is designed to distinguish classes of programs for which trace extrapolation is trivial and those for which trace extrapolation is intractable. The notion of equivalence relations between program executions makes it possible for the hierarchy to embody the full spectrum of possible choices of program execution equivalences. At one end of the spectrum, we have the notion that all executions are unique and at the other the notion that all executions are equivalent.

A family of execution equivalences is added to the hierarchy designed to allow any parallel program to be characterized, within the model, by the degree of execution pattern distortion exhibited by the program (on a given input). This provides a consistent framework in which to ultimately describe approximation to the trace extrapolation problem. In addition, the family provides a set small enough such that extrapolation over the set is reasonable, yet large enough to demonstrate that trace extrapolation is intractable (i.e., containing TCPs). Finally, the family will prove useful in characterizing the size of an extrapolation problem instance.

6.1 The Shared Data Dependence and TCP Precedence Relations

There exists a *direct shared data dependence* between two events a and b of an execution, denoted $a \rightarrow_{dd} b$, if a affects a shared variable that b directly depends on and at least one of these shared data accesses modifies a variable. A *shared data dependence* exists between a and b if there exists a chain of direct dependencies in an executions such that $a \rightarrow_{dd} x_1, x_1 \rightarrow_{dd} x_2, ..., x_n \rightarrow_{dd} b$. The *shared data dependence relation*, $\rightarrow_{sd}$, is the transitive closure of the set of shared data dependencies present in an execution.

Two executions that have the same shared data dependence also have the same order of accesses to shared memory. Executions that exhibit the same access order to shared memory have the property that any data values computed at any point in either execution are equivalent. Therefore, executions that have the same shared data dependence are data deterministic. It is also true that a set of data deterministic executions are guaranteed to execute the same set of events and are therefore pattern deterministic. We formally define the *shared data equivalence* for parallel programs:

Definition 6.1 *Two executions $P_1 =< E_1, \rightarrow_{T_1} >$ and $P_2 =< E_2, \rightarrow_{T_2} >$ with shared data dependence relations $\rightarrow_{sd_1}$ and $\rightarrow_{sd_2}$ respectively, are* shared data equivalent, *denoted $P_1 \equiv_{SD} P_2$, if for each $(a_1, b_1) \in \rightarrow_{sd_1}$ implies that there exists $(a_2, b_2) \in \rightarrow_{sd_2}$ such that $a_1 \equiv b_1$ and $a_2 \equiv b_2$.*

□

Programs that are pattern deterministic suffer no execution order distortion (i.e., they have a single execution pattern for a given input). Unfortunately, knowing that a program is pattern deterministic does not guarantee that a program is also data deterministic. The set of TCPs present in an execution coupled with their outcomes (see Definition 5.5) are used to formalize execution pattern determinism.

In Definition 6.2 we define the *TCP precedence* relation for a program execution, denoted $\rightarrow_{tp}$. This relation can be derived from the temporal ordering relation $\rightarrow_T$ by removing all non TCP events from the relation and removing all interaction between parallel threads of execution. Intuitively, the TCP precedence relation is the relation: $a\mathbf{R}b$ if a and b are TCPs, a executes before b, and a and b are not in parallel threads of execution. We use $TCP(E)$ to denote the set of all TCP events that occur in an event set E.

Definition 6.2 *Given a fixed but arbitrary program execution $P =< E, \rightarrow_T >$, the* TCP precedence *relation for P, denoted $\rightarrow_{tp}$, is the transitive closure of the following relation: For any $a, b \in TCP(E) : a \rightarrow b$ if and only if*

1. *$a \rightarrow_T b$ and either*
   - *there exists a process p such that $a \in E_p$ and $b \in E_p$ or*
   - *there exist processes p_1 and p_2 such that $a \in E_{P_1}$ and $b \in E_{P_2}$, and p_1 and p_2 do not execute concurrently in P.*

□

The $\rightarrow_{tp}$ precedence relation can be view as a means of removing the details of an executions instruction interleaving and focusing on the set of TCP instances present in the execution and the order in which these instances are executed. We return to execution pattern determinism shortly.

We now add three additional axioms to the basic model of Section 3 defining the behavior of the $\rightarrow_{sd}$ and $\rightarrow_{tp}$ relations. Axiom A6 ensures that the two relations are irreflexive partial order relations.

A6. $\rightarrow_{sd}$ and $\rightarrow_{tp}$ and are irreflexive partial order relations.

Axioms A7 and A8 represent the law of causality: if event a affects event b, then a must precede b in time.

A7. If $a \rightarrow_{sd} b$ then $\neg(b \rightarrow_T a)$.

A8. If $a \rightarrow_{tp} b$ then $\neg(b \rightarrow_T a)$.

Definition 5.5 defines the outcome of a TCP. Two programs that execute the same set of TCPs may not have executed the same set of events, because the outcome of the TCPs may not have been the same. For this reason, the TCP precedence relation is not sufficient to ensure pattern determinism for parallel programs. Two executions have the same TCP structure if they are TCP equivalent, (see Definition 6.3). Intuitively, two executions are TCP equivalent if they execute the same set of TCP instances and have the same outcome for each instance.

Definition 6.3 *Two executions* $P_1 = < E_1, \rightarrow_{T_1} >$ *and* $P_2 = < E_2, \rightarrow_{T_2} >$ *with TCP precedence relations* $\rightarrow_{tp_1}$ *and* $\rightarrow_{tp_2}$ *respectively, are* TCP equivalent, *denoted* $P_1 \equiv_{TCP} P_2$,

1. *if* $(a_1, b_1) \in \rightarrow_{tp_1} \Leftrightarrow (a_2, b_2) \in \rightarrow_{tp_2}$ *and* $a_1 \equiv a_2$ *and* $b_1 \equiv b_2$,
2. *for each* $b \in TCP(E_1)$ *there exists* $b \in TCP(E_1)$, *such that* $Outcome(P_1, b) = Outcome(P_2, b)$.

□

A set of executions, from a program and data set, are pattern nondeterministic if they are not pattern deterministic. We have the following relationship between pattern determinism and data determinism:

$$pattern\ determinism \Rightarrow data\ determinism \tag{1}$$

The converse is not true. A set of executions that is pattern nondeterministic is data nondeterministic. That is:

$$pattern\ nondeterminism \Rightarrow data\ nondeterminism \tag{2}$$

That is, a program must be data nondeterministic for the pattern of executions to change, however data nondeterminism is not a sufficient condition for pattern nondeterminism.

6.2 A Hierarchy of Execution Equivalences

6.2.1 The Hierarchy

Continuing with the idea of creating equivalences between executions, a hierarchy of equivalences can be defined using the shared data dependence relation and TCP equivalence. We define three levels of execution equivalence based on data and path determinism and data and path nondeterminism. They are: *data and pattern deterministic*; *data nondeterministic and pattern deterministic*; and *pattern nondeterministic*. We have the following relationship:

$$DataandPatternDet. \subset DataDet.andPatternNondet. \subset PatternNondet. \tag{3}$$

A *data and pattern deterministic execution equivalence* is one in which executions are equivalent if they have the same shared data dependence. This implies that each partition induced by such an equivalence represents a single execution pattern. More formally,

Definition 6.4 *An equivalence relation $\equiv$ on a set of program executions (for a given program and fixed but arbitrary data) is a* data and pattern deterministic *execution equivalence if for any two program executions $P_1 =< E_1, \rightarrow_{T_1} >$ and $P_2 =< E_2, \rightarrow_{T_2} >$, if $P_1 \equiv P_2 \Leftrightarrow P_1 \equiv_{SD} P_2$.*

□

Any equivalence relation satisfying Definition 6.4 partitions a set of executions into data deterministic partitions. It follows, from equation 1 the previous subsection and from Definition 6.4 that any partition induced by such an equivalence relation is a pattern deterministic partition (i.e., $P_1 \equiv_{TCP} P_2$). Given a source trace and a data and pattern deterministic equivalence relation on the set of possible program executions, there is a single partition of which the source trace is a member. If the set of executions for a parallel program using fixed but arbitrary input data, partitions into a single set under a data and pattern deterministic equivalence then the program is a *data and pattern deterministic program* (for the input data set).

A *data nondeterministic and pattern deterministic execution equivalence* is one in which two executions are equivalent if they are TCP equivalent. The requirement that two executions be TCP equivalent does not preclude the executions resulting in different output values. It does however guarantee that the programs execute the same set of events resulting in pattern equivalent executions. More formally,

Definition 6.5 *An equivalence relation $\equiv$ on a set of program executions (for a given program and fixed but arbitrary data) is a* data nondeterministic and pattern deterministic *execution equivalence if for any two program executions $P_1 =< E_1, \rightarrow_{T_1} >$ and $P_2 =< E_2, \rightarrow_{T_2} >$, if $P_1 \equiv P_2 \Leftrightarrow P_1 \equiv_{TCP} P_2$.*

□

If set of executions for a parallel program using fixed but arbitrary input data partitions into a single set under a data nondeterministic and pattern deterministic equivalence but does not for a data deterministic and pattern deterministic equivalence, then the program is a *data nondeterministic and pattern deterministic program* (for the input data set).

A *pattern nondeterministic execution equivalence* is one in which executions are equivalent if they have at least one TCP in common. More formally,

Definition 6.6 *An equivalence relation $\equiv$ on a set of program executions (for a given program and fixed but arbitrary data set) is a* pattern nondeterministic *execution equivalence if for any two program executions $P_1 =< E_1, \rightarrow_{T_1} >$ and $P_2 =< E_2, \rightarrow_{T_2} >$, $P_1 \equiv P_2 \Leftrightarrow TCP(E_1) \cap TCP(E_2) \neq \emptyset$.*

□

If a program, for a fixed data set, is not a data and pattern deterministic program or a data nondeterministic and pattern deterministic program, then if the set of executions for the parallel partitions into a single set under a pattern nondeterministic equivalence then the program is a *pattern nondeterministic program* (for the input data set).

6.2.2 Examples

We now have a framework that embodies the spectrum of possible equivalences that can be placed on program executions. Next, we present examples of equivalences of interest and identify where

they are in the hierarchy. In the examples we define equivalences that reside a various levels of the hierarchy.

Given two executions $P_1 =< E_1, \rightarrow_{T_1}>$ and$P_2 =< E_2, \rightarrow_{T_2}>$ with TCP precedence relations $\rightarrow_{tp_1}$ and $\rightarrow_{tp_2}$ respectively.

Example 1: two data and pattern deterministic equivalences:

(1.1) $P_1 \equiv P_2$ if $P_1 = P_2$.

This equivalence implies that E_1 and E_2 are equal and that $\rightarrow_{T_1}$ and $\rightarrow_{T_2}$ are *equal.* This equivalence partitions the set of executions of a parallel program into partitions that contain a single execution per partition.

(1.2) $P_1 \equiv P_2$ if $\rightarrow_{sd_1} = \rightarrow_{sd_2}$.

This equivalence is the largest data and pattern deterministic equivalence. The set of executions of a program are partition such that equivalences exhibiting the same shared data dependence reside in the same partition.

The difference between (1.1) and (1.2) is that (1.2) allows for the interleaving of instructions that are not shared memory references within a partition, while (1.1) does not allow any alternate interleaving of an execution within a partition.

Example: two data nondeterministic and pattern deterministic equivalences:

(2.1) $P_1 \equiv P_2$ if $P_1 \equiv_{TCP} P_2$

This equivalence is the largest data nondeterministic and pattern deterministic equivalence.

In [19], processor wait times at locks are abstracted into single *abstract* locking events. We can represent this characterization in the hierarchy. using a higher-level view of executions.

Definition 6.7 *The* abstract higher-level view *of a program execution $P =< E, \rightarrow_T>$ is a higher-level view $P_H =< E_H, \rightarrow_H>$ such that:*

1. *$\forall e \in E_i$ either $e \in E_H$ or e is an event instance of a* **test-and-set** *operation and there exists an event $e' \in E_{H_i}$ such that $e \in e'$, where i denotes the set of events for processor i,*
2. *$\forall (a, b) \in \rightarrow$ either $(a, b) \in \rightarrow_H$ or:*
   - *if a and b are* **test-and-set** *operations then if $a \in E_i$ and $b \in E_j$ and $i \neq j$, then $a \in e_{h_i}$, for some $e_{h_i} \in E_H$, and $b \in e_{h_j}$, for some $e_{h_j} \in E_H$, and $(e_{h_i}, e_{h_j}) \in \rightarrow_H$,*
   - *if a is a* **test-and-set** *then for $e_h \in E_H$ such that $a \in e_h$, $(e_h, b) \in \rightarrow_H$,*
   - *if b is a* **test-and-set** *then for $e_h \in E_H$ such that $b \in e_h$, $(a, e_h) \in \rightarrow_H$,*

□

This higher-level view compresses the **test-and-set** operations e present in a single thread in to a single **test-and-set** TCP event, e_h. That is, a series of TCP events and their corresponding outcomes are compressed into a single event. The resulting execution is a valid execution. If this execution were to be extrapolated, the wait times at processors would have to be resimulated in the target execution architecture to produce a correct target trace.

(2.2) $P_1 \equiv P_2 \Leftrightarrow$ the abstract higher-level view, P_{H_1} and P_{H_2}, yields $P_{H_1} \equiv_{TCP} P_{H_2}$.

There exists a higher-level view for any set of pattern nondeterministic executions such that application of the higher-level view to the executions results in a set of executions that are pattern deterministic executions. This is true because TCP events together with their outcomes can be compressed into other program events and effectively forgotten. This process is not reversible and resimulation must occur in order to produce a correct target trace. Another example of a construct that lends itself to compression is the iterative statement which can be compressed into a single statement (if the events executed during iteration are constant). This process can be reversed providing that the loop index can be reconstructed.

Example: two pattern nondeterministic equivalences:

We use **CT**(E) to denote the set of **task_create** and **task_terminate** statement instances in an execution.

(3.1) $P_1 \equiv P_2$ if:

$\mathbf{CT}(E_1) = \mathbf{CT}(E_2)$,

for all $a, b \in \mathbf{CT}(E_1), a \rightarrow_{T_1} b$ implies a $\rightarrow_{T_2} b$.

The equivalence given in (3.1) is a pattern nondeterministic equivalence. Executions reside in the same partition only if they have the same **task_create** (and therefore also **task_terminate**) structure.

(3.2) $P_1 \equiv P_2$ if $\rightarrow_{tp_1} = \rightarrow_{tp_2}$.

The equivalence given in (3.2) is also a pattern nondeterministic equivalence because it requires no TCP instance outcomes to be the same. Executions reside in the same partition as long as they execute the same set of TCPs without regard to outcome.

(3.3) $P_1 \equiv P_2$ if P_1 and P_2 are executions from the same program (on a fixed data set).

Equivalence (3.3) also a pattern nondeterministic equivalence, equivalences all executions to reside in the same partition. This typifies the view that any execution of a parallel program is as representative as any other.

6.2.3 The Hierarchy is Nontrivial

In Theorem 6.8 we demonstrate that the hierarchy is nontrivial. In order to prove that the hierarchy is nontrivial, we first show that the hierarchy is populated (by a specific *witness* equivalence) at each level. The examples given above are sufficient for this purpose. Next, we must show that there exists

a sufficiently complex program such that given a set of executions from the program (on a fixed data set) the witness equivalences yield distinct partitioning of the set. Some intuition about this sufficiently complex program is needed. We will use the program in Figure 16 below. The important statements (i.e., those containing trace change points) of the program are labeled. We will use these labels to uniquely describe an execution. In order to distinguish data and pattern deterministic equivalence from data nondeterministic and pattern deterministic equivalence, a program must have trace change points whose outcome, for the given data set, is independent of the interleaving of shared memory operations. In the program of Figure 16 statement F provides this behavior. Regardless of the interleaving of statements E and J, this statement will have the same outcome (for certain choices of input data). To distinguish data deterministic and pattern nondeterministic equivalence from pattern nondeterministic equivalence, a program must retain some similar TCP structure over a subset of execution in which TCPs with different outcome occur. In the program of Figure 16, this is accomplished by having executions with the same **task_create/task_terminate** structure but that have different TCP outcome over task 0.

Theorem 6.8 *The hierarchy is nontrivial.*

□

Proof: We must show that the hierarchy is populated by at least one witness equivalence at each level and in addition, we must show that there exists a sufficiently complex program such that each witness equivalence distinctly partitions a fixed set of executions of the program. Equivalences (1.1), (2.1), and (3.1) above are sufficient to demonstrate that the hierarchy is populated. The program of Figure 16 is sufficient to demonstrate that each of these equivalences yields a distinct partitioning of a fixed set of program executions. We choose the set of program executions that occur given the input data 3. If suffices to show that a subset of the executions reside in different partitions given each of the equivalences.

Consider the four executions below,

(S1) A B I C E J F H

(S2) A B I C J E F H

(S3) A B C D I E F J G H

First, notice that the data and pattern deterministic equivalence (1.1) places each of these executions in a different partition due to the interleaving of statements E and J (for S1 and S2) and occurrence of statement G (for S3).

For the data nondeterministic and pattern deterministic equivalence (2.1) statements S1 and S2 end up in the same partition. This is because TCP C branches whenever B completes before I (resulting in $t = -1$). If E completes before J then $t = -1$, and if J completes before E, $t = 1$.

```
                    begin

                    shared int i = 0, k = 0, j = 0;
                    shared int lki = 0, lkk = 0, lkj = 0;
                    int n = 2, m = 3;

                    read(k);
                    A: task_create(n)

    task 0                                      task 1

    int t = 0;

B:  lock(lki);                              K:  lock(lki);
    i = i - 1;                                  i = i + 2;
    t = i;                                      unlock(lki);
    unlock(lki);                            L:  lock(lkj);
C:  if t < 0 goto E                             j = j + 2;
D:  lock(lkk);                                  unlock(lkj);
    k = k + i;
    unlock(lkk);
E:  lock(lkj);
    j = j - 1;
    t = j;
    unlock(lkj);
F:  if k+t > 1 goto H
G:  lock(lkk);
    k = k + j;
    unlock(lkk);
H:  if k > i+j goto J
I:  task_create(m)
    task 3              task 4              task 5

    lock(lki);          lock(lkj);          lock(lkk);
    i = 0;              j = 0;              k = 0;
    unlock(lki)         unlock(lkj)         unlock(lkk)

    task_terminate(m);
J:  lock(lkk);
    k = k + i + j;
    unlock(lkk);
                    task_terminate(n);
                    end;
```

Figure 16: A pattern nondeterministic parallel program

In either case for $k = 3$, F branches. These executions execute the same set of TCPs with the same outcome (even though they are interleaved differently). Execution S3 delays the execution of statement I causing the value of k to be reduced to 2 allowing the branch at F to fail. This places S3 in a different partition from S1 and S2.

For the pattern nondeterministic equivalence (3.1), executions S1, S2, and S3 end up in the same partition. These executions have the same **task_create** and **task_terminate** structure.

Since equivalences (1.1), (2.1), and (3.1) exist and partition a set of executions from the program of Figure 16 distinctly, the hierarchy is nontrivial.

□

6.3 A Family of Execution Equivalences

Distinguishing pattern nondeterministic equivalences serves several purposes. First, proving the complexity of trace extrapolation requires a set small enough for the problem to be reasonable yet large enough to contain TCPs. Second, pattern nondeterministic programs can be distinguished with respect to their structure. Finally, the problem of determining the *size* of a trace extrapolation problem is complex. Clearly, the set of TCPs of a program has an impact on the size of a trace extrapolation problem. Therefore, a corollary of Theorem 4.3 is that the size of a trace extrapolation problem cannot be determined statically.

Corollary 6.9 *The problem of statically determining the size of a trace extrapolation problem instance is at least an NP-hard problem.*

□

We propose that distinguishing pattern nondeterministic programs leads to the identification of an indicator of the size of a parallel program with respect to trace extrapolation.

The largest possible data and pattern deterministic equivalence partitions a set of possible program executions such that each partition contains executions that exhibit the same shared data dependence. The partition to which a source trace is a member under the aforementioned equivalence will be called S_{DATA}. Similarly, the largest possible data nondeterministic and pattern deterministic equivalence partitions a set of possible program executions such that each partition contains executions that are TCP equivalent. The executions in such a partition execute the same set of events with the possibility of exhibiting a different shared data dependence. The partition to which a source trace is a member under the aforementioned equivalence will be called S_{PATT}.

.
.
.

A1: x = x + 1;	add $r-1$, 1
	store r_1,loc(x)
A2: if (j > 0)	branchle r_2, 0, A4
A3: y = y - 10;	sub r_3, 10
	store r_3,loc(y)
	branch A5
A4: add r_3, 10	
	store r_3, loc(y)
A5: j++;	incr r_2

.
.
.

Figure 17: A program fragment and (hypothetical) instructions (assuming that variable x is in register r_0, variable y is in register r_3, and variable j is in register r_2)

Execution 1	Events	Compressed	Events
A1	add r1, 1	A1	add r1, 1
	store r1, loc(x)		store r1, loc(x)
A2	branchle r2, 0, A4	A2	e(A2)
A3	sub r3, 10		
	store r3, loc(y)		
	branch A5		
A4	incr r2	A4	incr r2

Execution 2	Events	Compressed	Events
A1	add r1, 1	A1	add r1, 1
	store r1, loc(x)		store r1, loc(x)
A2	branchle r2, 0, A4	A2	e(A2)
A4	add r3, 10		
	store r3, loc(y)		
A5	incr r2	A5	incr r2

Figure 18: Compression of TCP events in two executions, from the program of Figure 6.2, using a TCP higher-level view

Using the structure of executions, a family of execution equivalences is defined, denoted $\rightarrow_{S_k}$. Given the *TCP* higher-level view, of Definition 6.10, a family of pattern nondeterministic equivalences is created. This view effectively *hides* the outcome of TCP events. That is, the TCP higher-level view simply compresses a TCP event and all of the events, in the same process, up to, but not including, the CP event, corresponding to this TCP instance. The intended effect is the creation of a TCP event, the outcome of which is the CP event corresponding to the TCP. This allows TCPs with differing outcome to be viewed as TCP equivalent. Consider the example of Figure 17 and Figure 18. Figure 17 depicts a program fragment and its corresponding hypothetical events. Assume that the program has a single TCP, A2. In Figure 18, two executions with differing outcome on A2 are shown together with their set of events. The result of compressing the TCP A2 in each execution is the compressed execution and resulting events. We use $e(A2)$ to denote the compressed TCP event. This choice is arbitrary and the **branchle** instruction could have been chosen instead. The example demonstrates that a TCP can be compressed such that its outcome in two different executions is hidden.

Definition 6.10 *Given a program execution* $P =< E, \rightarrow_T >$, *the* TCP higher-level view *of a program execution is another program execution* $P_H =< E_H, \rightarrow_{TH} >$ *where*

1. $E_H \subset E$ *such that for each* $e \in TCP(E)$ *and corresponding* $c \in CP(E)$ *with path* $p(e,c) =< E_{(e,c)}, \rightarrow_{(e,c)}>$ *in* E *and predecessor of* c, c_{pred}, *with path* $p(e, c_{pred}) =< E_{(e,c_{pred})}, \rightarrow_{(e,c_{pred})}>$ *in* E, *there exists events* $e_H \in TCP(E_H)$ *and* $c_H \in CP(E_H)$ *such that* $E_{(e,c_{pred})} = e_H$ *and* $c \in c_H$. *Any event that is not a TCP in E or is not on the path from a TCP event up to, but not including, its corresponding CP event represents an event in* E_H.
2. *for all instructions* $A, B \in E_H$, $A \rightarrow_{TH} B$ *if and only if* $\forall a \in A$ *and* $\forall b \in B (a \rightarrow_T b)$.

□

The TCP higher-level view can be used to define the $\rightarrow_{S_K}$ family of execution equivalences. Executions are grouped, by these equivalences, into sets of executions that never differ in the execution of more than k TCPs during a period of divergence.

Definition 6.11 *Given two program executions* $P_1 =< E_1, \rightarrow_{T_1}>$ *and* $P_2 =< E_2, \rightarrow_{T_2}>$ *if there exist TCP higher-level views* $P_{H1} =< E_{H1}, \rightarrow_{T_{H1}} >$ *and* $P_{H2} =< E_{H2}, \rightarrow_{T_{H2}} >$ *respectively, such that if* $P_{H1} \equiv_{TCP} P_{H2}$ *and choosing the smallest* k *such that* $\forall e_{H1} \in TCP(E_{H1})$ *and* $e_{H2} \in TCP(E_{H2})$, $TCP(E_{H1}) <= k$ *and* $TCP(E_{H2}) <= k$ *then* $P_1 \rightarrow_{S_k} P_2$.

□

Consider the equivalence $\rightarrow_{S_0}$ which groups all TCP events with their outcome up to, but not including, the corresponding CP events. Hence, no two TCP events are compressed into a single event. Any partition induced by this equivalence contains executions that never differ in the execution of a TCP. That is, executions that execute the same set of TCPs with possibly differing outcome, reside in the same partition. We consider these executions to have a TCP distance of zero from each other. Similarly, the equivalence $\rightarrow_{S_1}$ results in partitions in which executions never differ in more than a single TCP before returning to the same execution pattern. That is, each compressed TCP event will contain no more than a single additional TCP event. Given a program executed with a fixed but arbitrary data set and a source trace resulting from that execution, the partition induced by an equivalence $\rightarrow_{S_k}$ is denoted by the set S_k. The $\rightarrow_{S_k}$ family of equivalences are pattern nondeterministic equivalences. Figure 19 below demonstrates the placement of the sets S_{DATA}, S_{PATT}, and S_k in the hierarchy.

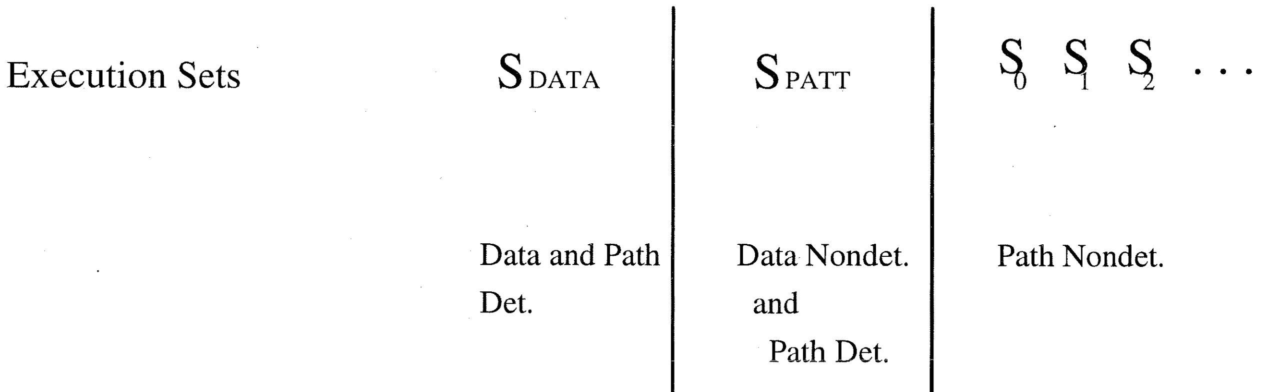


Figure 19: The family of sets in the hierarchy

To see how the $\rightarrow_{Sk}$ family can be used as an indicator of the size of a trace extrapolation problem consider the following. Any parallel program, not containing iteration statements, coupled with a data set result in a set of executions that partitions into a single set which is either S_{DATA}, S_{PATT}, or S_k for some k. For nondeterministic programs, with no iteration statements, exhibiting pattern nondeterminism, there is a value k that represents the largest *TCP distance* between any two program executions. This measure indicates the level of execution order distortion for a program. Iteration statements can be included if they are not TCPs (i.e., do not iterate based on the value of a shared variable). Programs that iterate on the value of a shared variable complicate the measurement process because no bound can be placed on the number of times the loop will iterate. Therefore, the number of TCP instances possible on a given path in an execution is unbounded.

The TCP distance measure indicates the level of execution order distortion that can occur in an execution. Clearly, other indicators are necessary in order to accurately determine the size of a trace extrapolation problem instance. For example, one such indicator might be the number of TCPs that a program must execute. The $\rightarrow_{Sk}$ also identifies this indicator which is the number of TCPs present in the set of events possible given the S_k partition for a program with TCP distance k. Hence, a description of the set of TCPs that *must* execute in any execution is provided.

6.4 Approximations

Execution 1	Execution 2
A	A
I	B
B	I
C	C
D	D
E	E
J	J
F	F
G	G
H	H

S_0 = { AIBCDEJFGH, AIBCDJEFH, ABICJEFH, ABICEJFH }

S_0^A = { ABICDEJFGH, ABICDEJFH, ABICEJFGH, ABICDJEFGH, ABICDJEFH, ABICJEFGH, ABICDJEFGH, AIBCJEFGH, AIBCDEJFH, AIBCEJFGH }

Figure 20: Execution from the program of Figure 6.1 and the associated sets S_0 and S_0^A

One problem that exists for trace extrapolation is that of determining whether or not a target execution is feasible. In the next chapter, we demonstrate that this problem is only tractable for extrapolation over S_{DATA}. Therefore, approximations of this set are unnecessary. For all other sets, S_{PATT} and the S_k sets, approximations are necessary in order to make this problem tractable.

Extrapolation over the set S_{PATT} can result in executions that could not occur due to the fixed outcome of TCPs and the ability to interleave shared memory references. For this reason, the approximation set S^A_{PATT} represents the set of executions to which a source execution can reasonably be extrapolated. That is, S^A_{PATT} is derived from S_{PATT} by allowing any possible synchronization of shared memory events. Consider the example of Figure 16; Figure 20, execution 1 is a feasible execution of the program, while execution 2 is an infeasible execution (recall that execution 2 is infeasible because interleaving statement B followed by statement I, and statements E and J, preclude the execution of statement G).

It is possible that performing extrapolation on the sets S_{PATT} and S^A_{PATT} for a source execution may not provide an accurate enough approximation. It may not be reasonable to perform extrapolation on a program that is S_k for some k using the potentially smaller set S^A_{PATT}. This is because forcing the target execution to be taken from a relatively small set of executions may result in a target execution that is unlikely or possibly unrepresentative of the correct trace. However, performing approximation on a smaller set has advantages. For example for S^A_{PATT}, no TCPs need be decided during extrapolation making extrapolation a simple process. It is reasonable to assume that determining the accuracy of a particular choice of approximation necessarily implies that the more accurate approximation set S_k be known.

For each set S_k, there is a corresponding approximation set, denoted by S^A_k (respectively). Recall that S^A_{PATT} includes all possible synchronization orders. This is because a feasible execution cannot be distinguished from an infeasible execution in an arbitrary S_{PATT} partition. Similarly, the approximations S^A_k must include both all possible synchronization orders as well as all possible length k TCP (or less) branch patterns. That is, we now have the problem of being able to distinguish whether or not a particular branch was possible. To nullify the problem of determining whether or not a branch direction was feasible, all branch directions are considered feasible. Returning to the example of Figure 20, the set S_0 for execution 1 is given. Each of these executions executes each program TCP (i.e., executes statements C and F). The set S^A_0 contains the additional (infeasible) executions, the set of all possible feasible and infeasible executions for the program of Figure 6.1 on input 3; because the program in Figure 6.1 has a zero TCP distance between any two executions. Trace extrapolation is trivial for any of these approximations sets, however a target trace taken from one of these sets may be incorrect.

Figure 21 below lists the various sets and their relationship in the hierarchy together with their corresponding approximation sets.

Approximations	N/A	S^{A}_{PATT}	S^{A}_{0} S^{A}_{1} S^{A}_{2} ...
Execution Sets	S_{DATA}	S_{PATT}	S_{0} S_{1} S_{2} ...
	Data and Path Det.	Data Nondet. and Path Det.	Path Nondet.

Figure 21: Family of sets and corresponding approximations in the hierarchy

We will return to these sets in Section 9. Measurement will require the ability to place a bound on the behavior of alternate executions. Approximation sets serve the following purpose in this work; because in general it will be impossible to determine whether or not a particular execution is feasible, extrapolation will be approximated and restricted to a given set of executions that are all assumed to feasible. This amounts to removing the problem of determining feasibility and implies that any measurements will represent upper bounds on the behavior of alternate executions.

7 The Complexity of Trace Extrapolation

In this section, we demonstrate that the trace extrapolation problem is in general intractable. The problem is proven to be NP-hard by transforming an arbitrary instance of Boolean Satisfiability (SAT) [25] to a parallel program with a single TCP.

The notion of feasibility of an execution for a program (and data set) is a fundamental one. The problem of determining feasibility is a central one for trace extrapolation and other trace related problems (e.g., perturbation analysis, debugging). All of these problems use a feasible execution to create an extrapolated execution which may or may not be feasible. The problem of determining whether or not an execution is feasible is a difficult one that in general requires program state information. We demonstrate that, in the absence of state information, this problem is undecidable for all but data deterministic programs.

7.1 Trace Extrapolation is Intractable

One difficulty with trace extrapolation is considering a set of executions that is small enough such that the problem is reasonable[18]. The trace extrapolation problem, as stated by Holliday and Ellis, is an unreasonable problem even for programs resulting in the smallest set of arbitrary pattern nondeterministic programs S_0. This is because the set of possible paths in S_0 is bounded by 2^N where N is the number of TCP instances in the set of events for an execution. That is, the set S_0 is already exponential in size. Holliday and Ellis' algorithm would have to construct path expressions that take into consideration all possible execution paths (which in this case is a very large set).

The trace extrapolation problem can be phrased as a reasonable problem by deciding TCPs only on demand. The set S_0 can be substantially pruned by requiring that when a TCP is decided, the set of executions under consideration are TCP equivalent prefixes of each other up to the last TCP. The problem is now reduced to one that considers only the paths between each TCP in the target execution. Theorem 7.1 demonstrates that even if the set of paths under consideration is pruned, the problem is still intractable. The proof reduces an arbitrary instance of the NP-complete problem of Boolean Satisfiability (SAT) [25] to an instance of the trace extrapolation problem (e.g., a program containing a single TCP).

Theorem 7.1 *Given a parallel program in our intermediate code language and a source trace s, the problem of extrapolating s to a target trace in the set S_0 induced by s, is NP-hard.*

[18] A problem is unreasonable to compute if the search space for the problem is exponential in the size of the problem input. In this case, the input to the problem is the source program and the search space, the set of possible executions, is exponential in size based on the number of TCPs in the source program.

□

Proof: Given an arbitrary instance of Boolean Satisfiability, we reduce the instance to a program, that results in a single partition of executions S_0 for any input, such that the instance of Boolean Satisfiability is satisfied only if a TCP branches TRUE on its condition. Given a set of variables, $X_1, X_2, ..., X_n$ and a set of clauses $C_1 \wedge C_2 \wedge ... \wedge C_m$ where each clause is constructed from the disjunction of a finite set of literals (variables or their negations), we reduce this instance of Boolean Satisfiability to a program containing $4n$ locks and $3n + 1$ processes.

```
begin   shared int x = 0
        shared locks
                X[1..n] = 0,
                Xnot[1..n] = 0,
                Mutex[1..n] = 0,
                pass[1..n] = 1,
        task_create(3n + 2)
        ...
        task_terminate(3n + 2)
end
```

The program proceeds in two phases. The first $3n$ processes choose values for the X_i. For each X_i we have three processes:

```
lock(Mutex[i])       lock(Mutex[i])        unlock(Mutex[i])
unlock(X[i])         unlock(Xnot[i])       lock(pass[i])
                                           unlock(Mutex[i])
```

In the first phase, the first two processes for each X_i compete to determine the value of the variable. The third process allows the initial competition and then in a second phase, unblocks the n processes still waiting from the initial competition.

The final process represent a correct solution to the Boolean Satisfiability problem and the start of the unblocking phase in the event of a correct or incorrect solution.

```
if (Boolean_Satisfiability_Instance)
        a: printf("Satisfied!")
unlock(pass[1])
⋮
unlock(pass[n])
```

The process first tests if the Boolean Satisfiability instance has been satisfied. If statement a executes, then the instance was satisfied, if not, then it was not satisfied. In either case, the remaining blocked processes from phase one are unblocked.

Clearly, the set of executions from the above program on any input partitions into a single set S_0. To demonstrate that the reduction is valid, recall that our problem is to determine whether or not we can decide the set of paths (feasible or infeasible) that yield true for a program TCP (i.e., the set of paths that include statement a). For the constructed program, it is either the case that all paths are feasible, or the Boolean Satisfiability instance is not satisfiable. That is, statement a is only executed if the Boolean Satisfiability instance is satisfiable. Therefore, we can decide the set of paths (feasible or infeasible) that contain the statement a only if we can determine whether or not the Boolean Satisfiability instance is satisfiable. □

One might be convinced that including more state information in a collected trace will reduce the complexity of the trace extrapolation problem. However, Theorem 7.1 demonstrates the this variant of the problem is also NP-hard. This is because the state information is only applicable to a single path and is therefore of little use after the first change in execution pattern occurs.

7.2 Complexity and the Real World

One difficulty with complexity proofs is that they may have little to do with real world situations. For example, the SPLASH suit of programs [50] contains relatively simple expressions in conditional statements. Given the complexity of the conditional expression used in the proof of Theorem 7.1, the proof is somewhat unsatisfying. Trace extrapolation is difficult for several reasons; synchronization structure, the potential for arbitrarily complex expressions in conditional statements, and the ability to construct simple conditionals that require complex path information.

Holliday and Ellis [32] explore the complexity of trace extrapolation via a series of small but increasingly complex program fragments and pose the question of exploring the complexity of the information necessary to decide a TCP. Clearly, parallel programs can use complex calculations in determining the value of a variable that if used in a conditional TCP statement, will require complex path information to decide. For example, many numerical methods iterate until some tolerance is reached (e.g., successive overrelaxation, some global relaxation problems).

It is a simple matter to construct scenarios where deciding simple conditionals requires detailed information about a set of paths. Consider the program of Figure 7.2. This program consists of two tasks that repeatedly compete to modify a shared variable *word*. One task adds to the value of word and the other shifts *word* up one bit. Statement F contains a simple conditional expression that depends on the exact sequence of lock acquisitions by task 1 and task 2 in order to decide its outcome.

Another difficulty with complexity results is that many problems shown to be intractable (e.g., problems that are in general NP-complete) are in practice reasonable to solve. This is because, in practice, the size of the problem instances remain relatively small. For trace extrapolation, given the complexity of the problem, the potential for conditional expressions that require complex path

```
begin

shared int word = 0;
shared int lk = 0;
shared int wordlen = 16;
int n = 2;
read m;
task_create(n)

task 0

    int x, k = 0;
A:  for (k = 0; k < wordlen; k++)
B:          lock(lk);
C:          word = word + 1;
D:          unlock(lk);
E:  endfor;
F:  if (word == m) then
G:          x = word;

task 1

    int j = 0;
A:  for (j = 0; j < wordlen; j++)
B:          lock(lk);
C:          word = word * 2;
D:          unlock(lk);
E:  endfor;

task_terminate(n);
end;
```

Figure 22: A parallel program containing a simple TCP requiring complex path information to decide

information to decide, and given the potential size of trace extrapolation problem instances, it is unlikely that there is a trace extrapolation algorithm, for parallel programs exhibiting program level nondeterminism, that performs well in practice.

7.3 Feasible Executions

In Section 11, we present an approximate trace extrapolation algorithm that produces a target execution in time linear in the length of the input source execution. The difficulty with approximate trace extrapolation is that a target execution may not be feasible. The ability to determine whether or not a target execution was feasible, would bolster the credibility of approximate trace extrapolation by identifying infeasible executions that might produce invalid simulation results.

Using program reexecution to determine whether or not an execution is feasible defeats our purposes. Thus, the only information present for determining feasibility is the information present in the source and target executions, and the equivalence used in trace extrapolation. Given the model of the previous section, this implies that a demonstration of equivalence, for some execution equivalence, must be sufficient to demonstrate that feasibility of an execution. In this process, we call the source execution the *witness* execution because it is known to be feasible and a member of the partition induced by the given equivalence. We refer to this problem as the problem of *determining feasibility given a witness.* For execution equivalences at most levels of the hierarchy, having a witness execution does little to help infer the correctness of another execution.

Determining feasibility given a witness execution requires that two conditions be checked; that the target execution is an execution (i.e., Axioms 1-5 hold) and that the source and target executions are equivalent. Any equivalence is required to be a *checkable* equivalence (i.e., the equivalence can be established without program execution or reexecution) and whenever we refer to an equivalence, this is assumed.

Theorem 7.2 below demonstrates that for a data and pattern deterministic equivalence, the feasibility of a target execution can be determined given a witness execution. If two executions have the same shared data dependence, then the correctness of one can be inferred from the other. Unfortunately, this result does not hold for any other class of equivalence. Theorem 7.3 demonstrates that any weaker equivalence does not provide sufficient information to infer the correctness of executions.

Theorem 7.2 *Given a fixed but arbitrary data and pattern deterministic equivalence and a feasible source execution, the feasibility of of an arbitrary target execution can be inferred from the source execution.*

□

Proof: Consider a witness execution $P_1 =< E_1, \rightarrow_{T_1} >$ and a fixed but arbitrary data and pattern deterministic equivalence $\equiv$ and assume that execution $P_2 =< E_2, \rightarrow_{T_2} >$ which may or may not be feasible is also given. We must shown that $P_1 \equiv P_2$ and P_2 not violating Axioms 1-5 implies that P_2 is feasible. Note that because $\equiv$ it can be determined whether or not $P_1 \equiv P_2$ and it is a simple matter to verify whether or not P_2 violates Axioms 1-5. If $P_1 \equiv P_2$ then the following are true:

1. $E_1 = E_2$
2. $\rightarrow_{sd_1} = \rightarrow_{sd_2}$

Assume that P_2 is not feasible. Given that P_2 does not violate Axioms 1-5, P_2 is infeasible due to executing an event that could not have happened given the program source. If this is the case, then $E_1 = E_2$ implies that P_1 is not feasible. Hence, by contradiction, P_2 must be feasible.
□

Theorem 7.3 *Given a fixed but arbitrary data nondeterministic and pattern deterministic (or pattern nondeterministic) equivalence, and a feasible source execution, the feasibility of of an arbitrary target execution cannot be inferred from the source execution.*

□

Proof: Given the relationship between data nondeterministic and pattern deterministic and pattern nondeterministic equivalences, it suffices to show that the theorem is true for an arbitrary data nondeterministic and pattern deterministic equivalence. The theorem can be shown to be true by providing a single example of a data nondeterministic and pattern deterministic equivalence, feasible source execution, and target execution such that the feasibility of the target execution cannot be inferred from the source execution. We demonstrate that theorem is true using the example of Figure 16 from the previous section together with the data nondeterministic and pattern deterministic equivalence (2.1) (from the previous section), and the following feasible source execution:

(e1) A I B C D E J F G H

Notice that in the program of Figure 16, the execution of statement I before the execution of statement C ensures that statement C fails to branch and therefore, statement D must execute. The following execution is equivalent to e1 under the equivalence given by (2.1).

(e2) A B I C D E J F G H

First, notice that $e1 \equiv e2$ (i.e., they are TCP equivalent) and that e2 does not violate axioms A1-A5. It remains to be shown that e2 is not a feasible execution. Notice that if statement I follows statement B, then the value of t at statement C is -1 and the branch is taken. Therefore, statement D could not have executed and e2 is an infeasible execution.

□

Theorem 7.3 implies that in general, for approximate trace extrapolation, it cannot be determined whether or not a target execution, derived during extrapolation, is feasible. In Section 9, we demonstrate additional limitations on what one might hope to determine about a target execution. Specifically, we demonstrate that there are theoretical limitations on ones ability to bound the error present in the target execution.

8 Computing TCP Sets

Theorem 4.3 demonstrates that the problem of locating the exact set of TCPs for a program is NP-hard (see Corollary 4.4). The problem of locating TCPs is a global flow analysis problem similar to that of constant propagation. There has been extensive research in the area of static detection of race conditions in parallel programs. While the problems of locating races and locating TCPs sets are different, an accurate estimate of the actual set of race conditions present in a parallel program can be used, together with the TCP location methods presented in this section, to provide an accurate estimate of the set of TCPs present in a program.

In this section, we present two algorithms that approximate the set of TCPs for a parallel program. Both algorithms provide a conservative approximation to the actual set of TCPs in a parallel program. That is, it is assumed that any conditional or iterative statement that is dependent on the value of a shared variable, either directly or indirectly, is a TCP. This reduces the problem of locating TCPs to that of determining the set of variables that are dependent on the value of a shared variable either directly or indirectly. We call this set the *shared variable dependent variable* set (SVD variable set). Computing the set of SVD variables is similar to the problem of constant propagation, a global flow analysis problem [57]. As with constants, SVD variables *propagate* through a program. There are two fundamental differences between the problems. First, the goal of constant propagation is to determine the actual values of constants (whenever possible). In contrast, the SVD variable set problem requires no computation of program values. Instead, the knowledge of whether or not a variable is an SVD variable is collected. Second, a variable is a constant prior to the execution of a statement if it was a constant on all paths leading to the statement and the constant value was the same. A variable is an SVD variable prior to the execution of a statement if it was an SVD variable on some path leading to the statement.

Iterative statements complicate the process of locating dependencies in a program. For programs containing no iteration, each statement can only depend on preceding statements. That is, dependencies can be computed with a single top to bottom traversal of the program. For programs containing iterative statements, each statement can depend on succeeding statements due to a cycle in the control flow graph. This implies that collection of the set of SVD variables for an iterative statement requires iteration over the set of statements in the body of the iterative statement until the set of SVD variables converges. This method has the advantage of accurately locating the changing dependencies that may be present in an iterative statement. We present an algorithm using this method based on Wegman and Zadeck's solution to constant propagation, see *simple constant* [57].

The cost of accurate computations for iterative constructs is a worst case run time of $O(N \times E \times V^2)$. We present a simplification requiring only $O(N \times E)$ for SVD variable location that trades time

for computational accuracy. Unlike constant propagation, the order in which statements occur is not as important for the SVD variable problem because actual program values are not needed. We introduce the *shared variable dependence graph* (SVD graph) of a loop and present a simplification for iterative statements using this representation. The SVD graph of a loop is a representation of the variable dependencies within the loop. Whenever a loop is encountered, the graph is sorted based upon the current SVD variables. This sorting results in a set of SVD variables for the loop. The loop can then be analyzed for TCP statements. This approximation can be less accurate than the previous algorithm but has a worst case runtime of $O(N \times V)$.

8.1 The Shared Variable Dependence Relation

Given two variables a and b, we say that a is *directly dependent* on b, denoted $a \rightarrow b$, if b appears on the right hand side of an assignment to a or if b is used to decide a conditional statement (or as an iterative statement index) in the body of which an assignment to a appears. If b is a shared variable, then a is *directly shared variable dependent* on b, denoted $a \rightarrow_{sv} b$. A variable a is *shared variable dependent* if it is either directly shared variable dependent on some shared variable or there is a chain of dependencies such that $a \rightarrow_{sv} v_1 \rightarrow_{sv} \ldots \rightarrow_{sv} v_n$ where v_n is a shared variable. The *shared variable dependence relation* (SVD relation) is the set of all shared variable dependencies for a program.

8.1.1 SVD variables

Determining the set of TCP statements for a program can be accomplished by determining the set of program variables dependent on the value of a shared variable. A local variable's dependence on a shared variable can change over the execution of program statements. For example, a local variable may be assigned the value of a shared variable in one statement and assigned to a constant value in a subsequent statement. Immediately following the first statement, the local variable is SVD dependent, however immediately following the second, it is no longer SVD dependent. The SVD relation can be used to estimate the set of program variables dependent (either directly or indirectly) on a shared variable (the set of *SVD variables* for the program). The set of variables present in the SVD relation at any program statement is the set of SVD variables for that statement. For each variable, membership in the SVD variable set is either *preserved*, *killed*, or *generated* during execution of a statement. The SVD variable set of maintained after the execution of each statement by removing all killed variables from the set and by adding any generated variables to the set.

There are two ways in which a variable can become an SVD variable. First, a dependency on a shared variable may arise as a result of a chain of dependencies due to assignment statements, a

definition dependency. Second, a dependency may result due to a chain of dependencies in which at least one dependency is due to a conditional or iterative statement, an *indirect dependency.*

8.1.2 A Motivating Example

```
                         A0:begin
                         shared int x = 0, y = 0;
                         int n = 2, z = 0;
                         A1:read(z);
                         A2:task_create(n)
task 0                                              task 1
int i,j,k,l;                                        int p,q,r,s;
B0: read(j,k);                                      C0:read(p,q);
B1: for i = 1 to j do                               C1:x = x + 1;
B2:     if k > j then                               C2:for r = 1 to x do
B3:         l = y;                                  C3:    p = p + x;
B4:     k = i + x;                                  C4:    q = q + p;
B5: end;                                            C5:    s = 1;
                         C6:end;
                         A3:task_terminate(n);
                         A4:if z > n then
                         A5:    z = n;
                         A6:print(n);
                         A7:end;
```

Figure 23: A parallel program illustrating SVD set calculation

In this section, we use an example to motivate the calculation of the SVD variable set and location of possible TCP statements. Figure 23, is used to demonstrate the calculation of the set of SVD variables and the location of TCP statements in a parallel program. We use S to denote the set of SVD variables and N to denote the set of variables that are not SVD variables. The contents of these sets will vary at different points in the program. Before any calculation begins, statements can be classified depending on whether or not they have to possibility of being TCP statements. For example, an assignment statement containing no array references cannot be a TCP statement and **task_create** statements are all TCP statements. We call statements that have the possibility of being TCP statements *candidate TCP statements.* Figure 23 contains five *candidate TCP statements*: statements A2, B1, B2, C2, and A4. Three of these statements are are TCP statements: statements A2, B2, and C2. Statement A2 is a TCP statement because it is a **task_create** statement. The set of SVD variables is used only to determine whether or not a candidate statement is a TCP statement.

Initially $S = \{x, y\}$ and $N = \{z, n\}$. For task 0, the sets are initially $S = \{x, y\}$ and $N = \{i, j, k, l\}$ and for task 1 they are initially $S = \{x, y\}$ and $N = \{p, q, r, s\}$. The candidate TCP statements for task 0 are statements B1 and B2. Statement B0 generates no generated or killed definitions, resulting in no change to the S and N sets. Statement B1 is not a TCP statement because $j \in N$ immediately prior to this statement. Statement B2 is a TCP statement since on some iteration of the loop, it is possible that $k \in S$. For task 1 the candidate TCP statement is statement C2. Prior to statement C2, $S = \{x, y\}$. The iterative statement C2 depends on the value of the shared variable x and the sets have the following values at the end of the loop $S = \{x, y, p, q, r, s\}$ and $N = \{\}$. The dependencies that include variables p, q, and s in S are indirect (note that there are dependencies due to a definition in C3 and C4). Statement C2 is a TCP statement because $x \in S$. For the main thread, statements A2 and A4 are candidate TCP statements. At the termination of tasks 0 and 1, the SVD variable sets are merged for global variable (in this case there are none) and $S = \{x, y\}$ and $N = \{n, z\}$. Statement A3 has no effect on these sets. Statement A4 is not a TCP statement because neither $z \in S$ nor $n \in S$. Therefore, statements A2, B2, and C2 are the only TCP statements.

8.2 Computing the *SVD* Relation

Shared variables are by definition always SVD variables (i.e., members of the SVD variable set). Local variables become SVD dependent variables either as a result of a direct dependency, due to an assignment statement, or as a result of indirect dependencies that arise as a result of control flow[19].

8.2.1 Assignment Statements

Any assignment statement has the possibility of generating new definitions or killing old definitions. An SVD variable is *generated* by an assignment statement, if the variable of the left hand side of the statement is not already an SVD variable, and if some variable of the right hand side is an SVD variable. If the left hand side of an assignment statement is currently an SVD variable and is not a shared variable, and the right hand side of the assignment statement contains no SVD variables, then the variable is removed, *killed*, from the current set of SVD variables. Consider the example of Figure 24. Statement A1 generates a single SVD variable, i. If the set of variables dependent on shared variables is $S = \{x\}$ and $N = \{i, j, k, m\}$ before the execution of statement A1, then after its execution we have $S = \{x, i\}$ and $N = \{j, k, m\}$. Statement A2 does not generate or kill

[19] We put off a discussion of the problem of determining SVD variables for procedure or function bodies until the end of this section.

any SVD variables. Statement A3 generates SVD variable j yielding $S = \{x, i, j\}$ and $N = \{k, m\}$. Statement A4 kills the SVD variable i resulting in $S = \{x, j\}$ and $N = \{i, k, m\}$.

```
int i,j,k,m;
shared int x;
:
A1: i = 2 * x + k;
A2: k = k + 1;
A3: j = k * i;
A4: i = k + 1;
A5: if j > 0 then
A6:        k = k + 2;
A7: else
A8:        m = m + k;
A9: ...
```

Figure 24: A program fragment containing assignment statements

The presence of array references complicates this simple calculation. Array elements are treated as individual variables except when an actual location (i.e., array element) cannot be determined. Because the actual element of the array that is SVD is unknown, the entire array must be considered to be SVD variables[20]. An array reference can become SVD dependent in two ways. First, if a reference appears on the left hand side of an assignment statement and an SVD variable appears on the right hand side of the same assignment statement then the reference is an SVD variable. Second, if the variable appears anywhere in an assignment statement and the index of the array reference is itself an SVD variable. In either case, the array element in question is added to the set of SVD variables. If an array reference appears on the left hand side of an assignment statement, the index for the reference is not an SVD variable, and there is no SVD variable appearing on the right hand side of the assignment statement, then if the array location was an SVD variable, the reference represents a kill. Consider the example of Figure 25. $S = \{x\}$ initially and $N = \{i, j, k, m, a[1], a[2], a[3], a[4], a[5]\}$ immediately preceding the execution of statement B1. Statement B1 is a TCP statement since the array reference $a[x]$ depends on the value of $x \in S$. This statement does not generate any shared variable dependencies. Statement B2 generates i as an SVD variable. Statement B3 is again a TCP statement since the reference $a[i]$ depends on the value of $i \in S$. This statement generates j as an SVD variable, $j \in S$. Statement B4 is not a TCP statement since

[20]Methods for array sectioning exist but are out of the scope of this work. See [45] for more details

the array reference $a[k]$ is not dependent on a shared variable. However, $a[k]$ is generated (because its value depends on SVD variables j and x) If we can determine the location of the array that is shared variable dependent, then it is added to S. If we cannot determine the exact location, then in order to guarantee that all TCP statements are located, the entire array is placed in S. Statement B5 may or may not be a TCP statement depending on which of the above decisions we make at statement B4 (i.e., statement B5 depends upon the contents of S). In the algorithm that follows, we omit the details of determining dependencies for array references, see [45] for details of array analysis in data flow problems.

```
int i,j,k,m;
int array a[1..5];
shared int x;
:
B1: a[x] = i + j;
B2: i = i + x;
B3: j = a[i];
B4: a[k] = x + j;
B5: if a[m] then
:
```

Figure 25: A program fragment containing array reference assignment statements

8.2.2 Conditional Statements

A conditional statement can result in indirect dependencies. In the event of an indirect dependency, any definition that occurs within the body of the conditional statement, regardless of the outcome, is generated by the execution of the conditional statement. Consider statement A5 in Figure 24. Before execution of this statement $S = \{x, j\}$ and $N = \{i, k, m\}$ hence A5 is a TCP statement. After execution of this statement (immediately preceding the execution of statement A9), we have $S = \{x, j, k, m\}$ and $N = \{i\}$. The variables k and m are generated by the conditional statement.

If a conditional statement is not a TCP statement, then the SVD variable set is calculated along each branch of the conditional and TCP statements are located. Each branch is evaluated with the SVD variable set that is present on entry to the conditional statement. Because each branch may possibly execute and execution will occur independent of one another, the resulting SVD variable sets are merged at the end of the conditional statement. That is, the SVD set at the end of a conditional statement is the union of the SVD sets from each branch of the conditional

```
int i,j,k,m;
shared int x;
⋮
C1: if j > 0 then
C2:     i = i + x;
C3:     j = j + 1;
C4: else
C5:     j = a[i] + 1;
C6:     k = x + 1;
C7:     m = i + k;
C8:
⋮
```

Figure 26: A program fragment containing a non TCP conditional statement

statement. Consider the example of Figure 26. Initially, $S = \{x\}$ and $N = \{i, j, k, m\}$. Statement C1 is not a TCP because $j \in N$. The SVD variable set S after the execution of the conditional statement (i.e., at the beginning of C8) is $S = S_1 \cup S_2$ where S_1 is the SVD variable set generated by statements C2 and C3 and S_2 is the SVD variable sets generated by statements C5, C6, and C7. Given the initial set $S = \{x\}$, statements C2 and C3 generate $S_1 = \{x, i\}$ and given the initial set $S = \{x\}$, statements C5, C6, and C7 generate $S_2 = \{x, k, m\}$. The set $S = S_1 \cup S_2 = \{x, i, k, m\}$ and $N = \{j\}$.

8.2.3 Iterative Statements

An iterative statement can induce indirect dependencies. In this case, as with conditional statements, any definition occurring within the body of the iterative statement represents a generation by this statement (of SVD variables). Iterative statements that do not fall into this category require a more complicated calculation to determine dependencies. Consider the following example:

On entry to statement D1 we have $S = \{x, y\}$ and $N = \{q, r, s, t, n\}$. For the first iteration of the loop, this is true until statement D7 is executed and generates q. On the second iteration of the loop, after the execution of statement D4 (and immediately preceding the execution of statement D7) we have $S = \{x, y, q, s, t\}$ and $N = \{r, n\}$. At this point, we have identified statement D4 as a TCP statement but have failed to identify statement D2 because t was generated by statement D6 in the second iteration of the loop. Statement D4 will be identified as a TCP statement in

```
shared int x, y;
int q, r, s, t, n;
⋮
D1: for r = 1 to n do
D2:     if t > n then
D3:         t = t - n;
D4:     if q > s then
D5:         s = s + t;
D6:         t = t + r;
D7:     q = (x * y) + (t - r);
D8: end
```

Figure 27: A program fragment containing a TCP iterative statement

the third iteration of the loop. In order to identify all TCP statements, the loop will have to be scanned until the set of SVD variables remains constant over successive iterations.

8.3 Preprocessing

We use a control flow graph [23] of the program decorated with a single pointer to the convergence point for a TCP (if there is one in a block), and a flag denoting whether or not a block contains a TCP. To ensure that each block contains only a single TCP candidate, we include in the set of block delimiters, for the control flow graph, assignment statements containing array references. This suffices to ensure a single TCP candidate per block since all TCP candidates are conditional statements or other parallel program structuring statements (e.g., task_create). We call this control flow graph the *TCP control flow graph* for a parallel program.

We use an array, one element for each program variable, to maintain the set of SVD variables.

In the following sections, we use N, E, and V to describe the size of an input problem. Let N be the number of assignment, conditional, and iterative statements in the program, E be the number of edges in the modified program flow graph, and V be the number of variables in the program.

The program flow graph and SVD variable array requires $O(N \times V)$ space, and $O(N)$ time to construct.

8.4 A Worklist Algorithm

The problem of computing the set of SVD variables for a program is similar to that of constant propagation [57]. Both problems can be solved using data flow analysis techniques. Wegman and Zadeck's version of Kildall's simple constant algorithm [34], determines the lifetime and extent of constant definitions in a program when simple constant propagation is considered. The algorithm can be modified to determine the SVD variable sets for a program. Wegman and Zadeck's algorithm is a worklist algorithm [29]. We will use this algorithm to illustrate the worklist algorithm technique.

8.4.1 Simple Constant

Both simple constant and the SVD variable set computation for programs represents the computation of a property of variables that vary over the program (e.g., a variable is constant or not constant, a variable is an SVD variable or not an SVD variable, etc.). The desired property of the variable is either killed, preserved, or generated by a statement. For example, in Figure 28, if the variable y is constant and variable z is not constant prior to executing statement B, then after execution, the variable y is no longer constant (i.e., killed by the statement) and the variable z remains not constant (i.e., preserved by the statement). Clearly, the property of individual variables reaching a statement can effect the resulting property of variables at termination of a statement. In the previous example, if z had been constant, then y would be constant at the termination of statement B. Iteration and branching complicates the calculation of a property by introducing the potential for conflicts between multiple possible paths reaching a statement. Notice that in Figure 28, statement A has two predecessors due to the control flow of the program. If y is a constant and z is not a constant, then for the first iteration of the loop, x is a constant (using simple constant propagation) at the termination of statement A. However for subsequent iterations, y is not a constant (due to statement B), and x is not a constant at the termination of statement A.

In simple constant, and for the SVD variable computation, a lattice of possible states for a variable is used to resolve conflicts in property. Conflicts may arise when a statement has multiple predecessors for which the property of a variable (or state) is different on termination. Recall that a lattice is a partially ordered set containing a greatest element $\top$ and a least element $\bot$. The elements of a lattice correspond to states. For simple constant, the elements of the lattice correspond to the three possible states of interest for a variable, those of *no information* ($\top$), *constant*, and *not constant* ($\bot$). In reality, each possible constant value (of which there are many) must represent a different state, because if for a particular variable, two paths leading to a statement are constant but have different values, then the variable is not constant prior to the execution of the statement. Figure 29 has been simplified to include only the integer constants. The state lattice is equipped with an operation $\sqcup$ that serves to resolve conflicts between states in the lattice. The rules for $\sqcup$

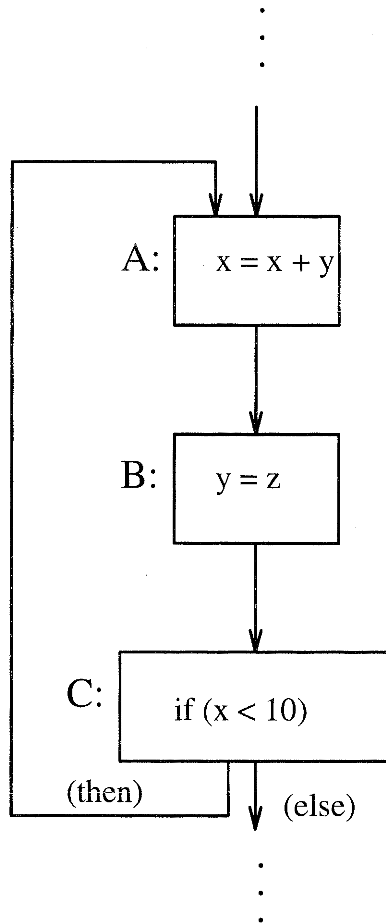
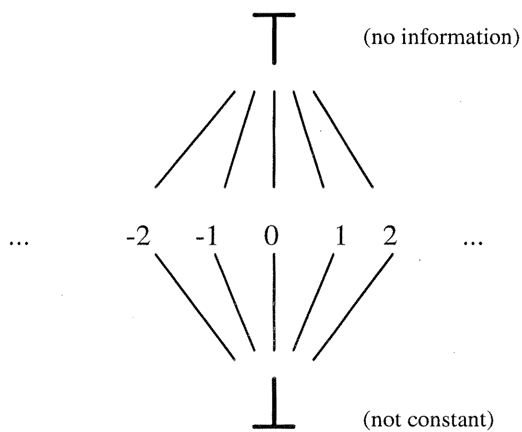


Figure 28: An example control flow graph



(1) any $\sqcup \top$ = any

(2) Const $\sqcup$ Const = Const

(3) any $\sqcup \bot = \bot$

Figure 29: The lattice and associated rules for simple constant

are given in Figure 29. Rule 1 indicates that if on two paths leading to a statement, the state of a variable is unknown and *any* state, then the conflict is resolved by assigning the state of the variable on entry to the statement to *any* (e.g., any information about the state of a variable overrides no information). Returning to Figure 28, for the first iteration of the loop, y is an unknown state for the first path leading to statement A and is a constant for the second, therefore, it is a constant on entry to statement A. Rule 2 indicates that if two paths have the same constant state for a variable, then the variable remains a constant. Finally, rule 3 indicates that if on any path leading to a statement, a variable has been determined to be not constant ($\perp$), the the variable is not constant on entry to the statement.

A statement level control flow graph of the program suffices to identify statements and the possible paths leading to them. Each node in the control flow graph has two arrays representing the state of each variable in the program on entry to the node and on exit. A worklist algorithm iterates over the nodes of the control flow graph in order to determine the desired property for variables at each node in the graph. The algorithm begins by adding the root node for the control flow graph to the worklist and proceeds by removing nodes from the worklist and processing the nodes until the worklist is empty. Initially, the state of all variables at the entry and exit of each node are $\top$ (no information). When node is removed from the worklist the killed, generated, and preserved definitions are computed for the node based on the state of incoming variables to the node. If at any time, the state of a variable is changed by the killed or generated definitions of a node, all nodes that are immediate successors of the current node, in the control flow graph, are added to the worklist. These nodes need to (re)processed to reflect changing variable states.

8.4.2 The SVD Variable Problem

Notice that simple constant requires an intersection over the predecessors of a statement[21]. This is illustrated in Figure 29 rules 2 and 3, where all of the predecessor states of a variable must be the same constant value in order for a variable to be a constant on entry to a statement and if in any predecessor state a variable is $\perp$ (i.e., not constant), then the variable state is $\perp$ (rule 3) on entry to a statement. In contrast, the SVD variable set computation requires the union over the predecessors of a statement[22]. Here, a variable is an SVD variable if it was an SVD variable for some predecessor. The algorithm described below is identical to Wegman and Zadeck's except that the lattice under consideration has been redefined for the SVD variable computation.

For SVD variable set computations, each statement can affect the set of SVD variables, however only those changes to the set of SVD variables immediately preceding and immediately following a TCP candidate are of interest. We use the modified program flow graph of the previous section

[21] Simple constant is a top down intersection data flow problem.

[22] The SVD variable problem is a top down union data flow problem

to represent a program as opposed to a statement level control flow graph. Once a statement has been determined to be a TCP statement, it is added to the set of TCP statements for the program and can never be removed. Two arrays of size V are associated with each block – one for the block entry state of the SVD variable set and one for the block exit state.

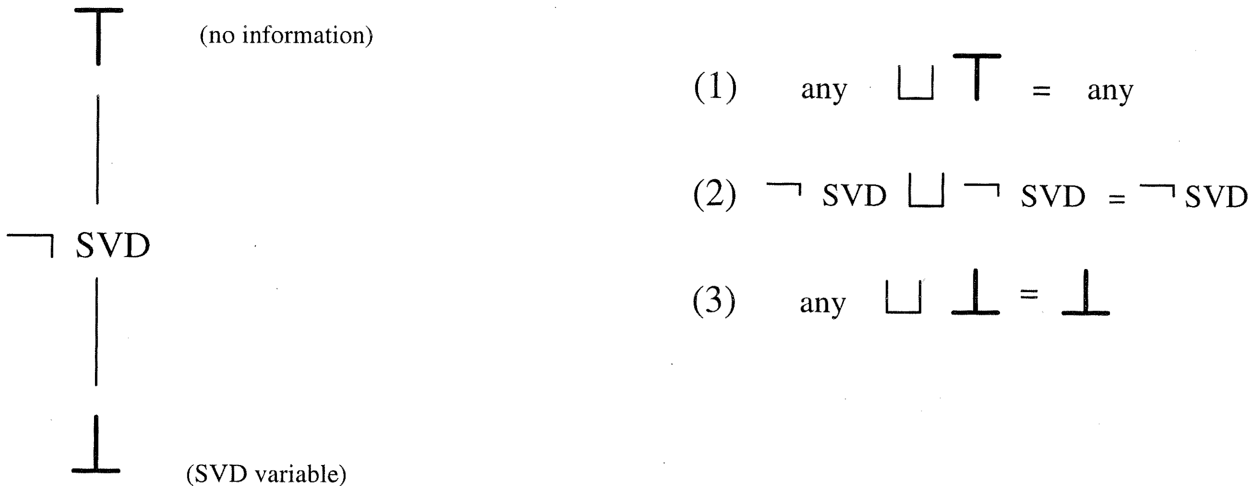


Figure 30: The lattice for the SVD variable problem and associated rules

The algorithm uses a lattice containing three states and propagates values in a program flow graph. The lattice for SVD variables is shown in Figure 30. Notice that we are only interested in whether or not a variable is an SVD variable (i.e., the value of the variable is not of interest). This simplifies the lattice. Figure 30 includes the $\sqcup$ rules for the lattice. The highest element, $\top$ represents an undetermined state for a variable. That is, if a variable has a value of $\top$, then it is not known whether or not the variable is in the SVD variable set. The lowest element, $\perp$, represent the set of SVD variables. The middle element $\neg SVD$ represents variables that are currently known not to be in the SVD variable set. Our intuition about the state of variables is the following: If we have no information about the state of a variable on some path leading to a node, then information on other paths overrides this lack of information (Figure 30, rule 1). If a variable is not SVD on all paths leading to a node, then it is not SVD for the node (Figure 30, rule 2). If a variable is found to be SVD on some path leading to a node, then it is SVD for that node (Figure 30, rule 3).

The algorithm begins by adding the start node of the graph to the worklist of nodes to be processed. Initially, the value of every variable is $\top$ with the exception of the shared variables which have a value of $\perp$. Nodes are removed from the worklist and processed until the list is empty. The entry state values for each node are the $\sqcup$ of the exit state values from all predecessor nodes. The variable *flag* is used to denote when a TCP conditional (or iterative) statement has been encountered. The algorithm processes statements as outlined in Section 8.2. The assignment statements in the node are evaluated by taking the $\sqcup$ of the state values for all variables on the

right hand side to be the state value for the variable on the left hand side. As each statement in a node is processed, candidate statements are checked against the current state to determine whether or not they are TCP statements. Once a statement has been determined to be a TCP statement, it cannot be removed from the set of TCP statements. When a TCP conditional or iterative statement is encountered, a flag is set signifying that each node encountered between the node and its continuation must be processed such that any variable appearing on the left hand side of an assignment statement is an SVD variable. If during processing of a node, the exit state does not match the entry state (for some variable(s)), then each successor to the node must be added to the worklist for processing. If the value of the entry state does not change for a node, then it need not be processed. Similarly, if the exit state for a node does not change, then no successor nodes are added to the worklist.

8.4.3 Complexity and Correctness

In simple constant, the state of any variable can only change twice (i.e., the lattice value for the variable can only be lowered twice per block). The same is true for an SVD calculation. Given that the number of predecessors for a node is I, each node can only appear on the work list a maximum of $2 \times V \times I$ which is $O(E \times V)$ for each node. At each node, $C \times V$ meet operations are performed, where C is a constant describing the number of statements in a block[23]. The worst case time for the algorithm is $O(N \times E \times V^2)$ and the space requirement is $O(N \times V)$. The lower bound for runtime for the algorithm is $O(N \times E \times V)$ in the case that each node appear on the work list a single time.

In [57], Wegman and Zadeck present formulations of the constant propagation problem with asymptotically better worst case run times. It is unknown if these methods can be applied to the SVD variable problem.

8.5 An Asymptotically Faster Approximation

The problem of constant propagation require more than a simple determination of whether or not a variable is a constant at a particular point in the program. In addition, the value of the constant is of interest. The SVD variable set calculation does not require detailed information about values being computed. Rather, the only information of interest is whether or not a variable is a member of the SVD variable set. This difference makes it possible to simplify the SVD calculation for a program by reordering program statements. This reordering can produce a program that computes incorrect values, with the same statements contained in each block. For example, it is possible to

[23]The number of statements in a block must either be a constant when compared to N, or the program flow graph can consist of a single statement per block.

reorder statements within an iterative statement but it is not possible to move statements into or out of an iterative construct. Returning to the example of Figure 27, notice that if it were possible to reorder the statements of the iterative statement body, then a single pass would suffice to collect SVD variables. This implies that a variable has a single state within the entire loop – that is, it is either an SVD variable or it is not. We know that this is not guaranteed to be the case and this assumption may result in a potentially larger set of TCP statements than the previous method. However, we are guaranteed that the set of TCP statements located contains the set of actual TCP statements. If a variable is generated by any statement within the body of the iterative statement, then it is generated by the iterative statement. If a variable is killed by a statement within the body of the iterative statement and is not generated by any other statement within the body of the iterative statement, then it is killed by the iterative statement.

8.5.1 The SVD Graph

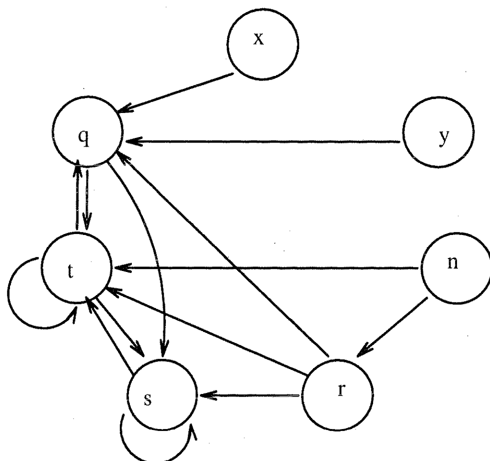


Figure 31: SVD graph of program fragment in Figure 8.5

Rather than reorder the statements in an iterative construct, the generated and killed dependencies for the body of the construct can be represented using a *shared variable dependency graph* (SVD graph). Figure 31 is the SVD graph for the iterative statement in Figure 27. Notice that q is dependent on x and y (statement D7), and that t and s are dependent on q (statements D5 and D6). In the SVD graph, a directed edge from node a to node b exists if variable b depends on variable a at some time in the body of the iterative statement. In Figure 31, variable q depends on variables x, y, t, and r. The SVD graph represents the dependencies for the entire loop as opposed

to individual statements of the loop. Dependencies can be placed in the graph during a single traversal of a section of code. Once an SVD graph has been constructed for a section of code, the graph is topologically sorted from each node known to represent an SVD variable. The topological sorting produces the set of SVD variables generated by the code. Similarly, a topological sort can be used to determine the set of SVD variables killed by the section of code. For nested iterative statements, a single shared variable dependency graph is constructed[24].

given: A section of a parallel program p and a input dependence set V.
compute: The shared variable dependence graph for p.

algorithm:

```
1  Compute_svd_graph(p, V):
2  graph G = EMPTY;
3  for each statement s of p do
4        case s of
5              assignment : given l = lhs(s),
6                           add_node(l, G);
7                           for each v ∈ rhs(s) do
                                  add_directed_edge((v,l), G);
8                           for each v ∈ V do
                                  add_directed_edge((v,l), G);
9              conditional : given V_s ∈ condition(s), p_sTRUE and p_sFALSE,
10                          G = union_graphs(G, Compute_svd_graph(p_s_TRUE, V ∪ V_s));
11                          G = union_graphs(G, Compute_svd_graph(p_sFALSE, V ∪ V_s));
12             iterative :   given v = index(s) and p_s,
13                          G = union_graphs(G, Compute_svd_graph(p_s, V ∪ V_s));
14       end case
15 end for
16 return(G);
17 end.
```

Figure 32: An algorithm for computing the SVD graph for a parallel program

Figure 32 computes the SVD graph for a section of code given a set of variables on which the entire section is dependent. This set includes the index variables from any enclosing iterative statements. Function *Compute_SVD_graph* is initially applied to the body of an iterative statement. Initially, V contains only the loop indexing variable. For each assignment statement, an edge is added to the graph from each variable appearing on the right hand side of the assignment to the variable appearing on the left hand side (line 7). In addition, the variable on the left hand side is dependent on each variable in V and an edge from each variable in V to this variable is added to the

[24]For some programs, this method can approximate the set of program TCPs poorly. An example of the type of program for which this might happen is a program that consists almost entirely of a single iterative statement.

graph (line 8). For a conditional statement, *Compute_SVD_graph* is called recursively on the body of each branch of the conditional statement with the variables appearing in the condition added to the dependent variable set. The results of each of these are added to the existing graph (see lines 9-11). For an iterative statement, *Compute_SVD_graph* is called recursively on the body of the iterative statement with the index variable added to the dependent variable set. The resulting graph is added to the existing graph (lines 12-13).

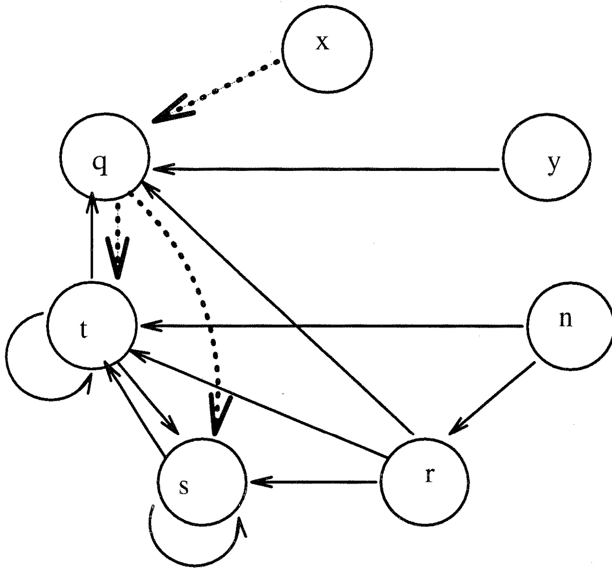
We use this representation to approximate the TCP statements within an iterative statement. Each iterative statement in a program is preprocessed to yield its associated shared variable dependence graph. During the process of approximating the TCP statements of a program, the shared variable dependence graph coupled with the current set of shared variable dependent variables is used to produce the set of shared variable dependent variables for the iterative statement. This is accomplished by computing a topological sort of the graph starting at each vertex that represents a known shared dependent variable. Consider again the shared variable dependence graph in Figure 31. If the set of SVD variables at the beginning of the loop (line D1) is $S = \{x, y\}$ then a topological sort of the graph starting at the variable x yields the tree highlighted in Figure 33. Each of the variables in this tree is added to the shared dependent variable set and deleted from the graph[25]. Figure 33 also shows the graph resulting from this deletion. A topological sort of the graph beginning with y, yields no new shared dependent variables. Finally, n and t are not determined to be shared dependent by any topological sort on known shared dependent variables. Once the set of shared dependent variables for a loop has been determined, the set can be used to approximate the set of TCP statements in the body of a loop.

8.5.2 The Algorithm

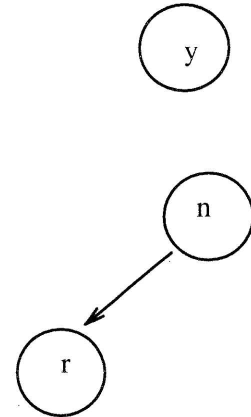
Notice that the SVD graph simplification alleviates the need for a per node copy of the entry and exit states for variables. This is because each node will now be traversed a single time allowing a single array to record the current SVD variable set.

Figure 34 approximates the set of TCP statements of a program. Starting at the top of the program flow graph p (the program begin statement) function *Approximate_TCPs* traverses the entire graph calculating the current state of the program variable (i.e., whether or not they are shared variable dependent). Each statement in the current block (line 4) is used to update the *state* of the program variables. If the statement is an assignment statement (line 6) then if $flag$ is set (signifying that this statement is enclosed by a conditional or iterative TCP statement) the left hand side of the assignment is automatically added to the current state as a shared variable dependent variable. Otherwise function *Do_Definition* determines whether or not the left hand

[25]Each variable need only be added to the shared dependent variable set a single time.



Topological sort starting at node x.



Reduction of the graph.

Figure 33: Reduction of the graph in Figure 8.9

side of the assignment represents a generated shared variable dependency (line 24) or a kill of one (line 25). If the statement is an array assignment statement then it may or may not be a TCP statement. Hence the statement needs to be checked as to whether or not it is a TCP statement (line 7) as well as determining whether or not the left hand side of the assignment is a shared variable dependent variable (line 8). If the statement is a conditional statement, it is first checked to determine whether or not it is a TCP statement (line 9). If this is the case, then *flag* is set signifying that all variables appearing on the left hand side of assignments within the body of the conditional statement are automatically placed in the shared variable dependent set. Whether or not the statement is a TCP statement, *Approximate_TCPs* is called recursively on each branch of the conditional (lines 10-11), given the state before the conditional, and the two resulting states are merged (line 12) a TCP statement. If the statement is an iterative statement, whether or not the statement is a TCP statement must be checked (line 13) and if it is, then *flag* is set. Function *do_SVD_graph*, see Figure 35, is used to determine the change in state due to the loop being executed and to determine the TCP statements present in the loop given the input state *state* (line 14). The current state and the state calculated during the loop are merged at the end of the loop (line 15). Finally, if the statement is a *task_create* statement then each thread must be evaluated using *Approximate_TCPs*.

given: a parallel program flow graph p, a state *state*, and a conditional flag *flag*.
compute: approximate the set TCP of TCPs of p.
TCP is a set containing known TCP statements, is global and initially empty. *state* is an array of flags, one element for each program variable.
algorithm:

```
1  Approximate_TCPs(p, state, flag): returning state;
2  current_block = p;
3  while current_block != NULL do
4        for each s in current_block do
5              case s of
6                    assignment :  if flag then state = add_to_state(lhs(s))
                                   else state = do_definition(s,state,flag);
7                    array_assn :  if isTCP(s,state) then TCP = TCP ∪ s;
8                                  if flag then state = add_to_state(lhs(s))
                                   else state = do_definition(s,state,flag);
9                    conditional : if isTCP(s,state) then
                                         TCP = TCP ∪ s;
                                         flag = TRUE;
10                                 lstate = Approximate_TCPs(copy(left_branch), state, flag);
11                                 rstate = Approximate_TCPs(copy(right_branch), state, flag);
12                                 state = state ∪ rstate ∪ lstate;
13                   iterative :   if isTCP(s,state) then
                                         TCP = TCP ∪ s;
                                         flag = TRUE;
14                                 loop_state = do_svd_graph(copy(loop_body), state, flag);
15                                 state = state ∪ loop_state;
16                   task_create : for each thread in s do
17                                       Approximate_TCPs(copy(thread), shared(state)), flag);
18                   otherwise :   do_nothing;
              end
        end
19      current_block = current_block → continuation;
   end
20 return state;
   end.

21 do_definition(s,state, flag): returning state;
22 if flag then return(add_to_state(lhs(s)));
23 for each v ∈ rhs(s);
24       if v ∈ state then return(add_to_state(lhs(s));
25 return(remove_from_state(lhs(s)));
26 end.
```

Figure 34: An algorithm for approximating the set of TCP statements for a program

given: a parallel program flow graph p, a state *state*, and a flag *flag*.
compute: approximate the set TCP of TCPs of p.
TCP is a set containing known TCP statements and is global. *state* is an array of flags, one element for each program variable
algorithm:

```
1  do_svdgraph(p, state, flag): returning (state);
2  new_state = create_state();
3  if flag then
4        ttree = topological_sort_with_delete(G, index_variable(p);
5        new_state = add_to_state(nodes(ttree));
   else
6        for each dependent variable v in state ∪ new_state do
7              if v ∈ G then ttree = topological_sort_with_delete(G,v);
8              new_state = add_to_state(nodes(ttree));
9        for each non-dependent variable v in state ∪ new_state do
10             if v ∈ G then ttree = topological_sort_with_delete(G,v);
11             state = remove_from_state(nodes(ttree));
12 TCP = do_code(p, state ∪ new_state);
13 return state ∪ new_state;
   end.

14 do_code(p, state): returning TCP;
   TCP = EMPTY;
15 current_block = p;
16 while current_block != NULL do
17       for each s in current_block do
18             case s of
19                   array_assn :  if isTCP(s,state) then TCP = TCP ∪ s;
20                   conditional : if isTCP(s,state) then TCP = TCP ∪ s;
21                   iterative :   if isTCP(s,state) then TCP = TCP ∪ s;
22                   otherwise :   do_nothing;
               end
         end
23 current_block = current_block → continuation;
   end
24 return TCP;
   end.
```

Figure 35: An algorithm for approximating the set of TCP statements given an SVD graph

The algorithm of Figure 35 uses the shared variable dependence graph for an iterative statement to determine the correct set of shared variable dependent variables for the loop. This set is subsequently used to approximate the TCP statements within the loop body. At line 3, if $flag$ has been set, then a simple topological sort (line 4) of the graph beginning at the index variable for the loop, then all variables that appear on the left hand side of an assignment in the body of the loop will add to the current set of SVD variables (line 5). If $flag$ is true, then this loop is either enclosed in an outer conditional statement which is a TCP statement or is itself a TCP statement. If this is not the case then $flag$ is false and the set of SVD variables generated by the loop body is calculated in lines 6-8. For each variable in the current SVD variable set, the SVD graph is topologically sorted and the resulting tree of nodes are removed from the SVD graph (lines 6-7). This continues until no nodes in the SVD variable set remain in the SVD graph. The set of SVD variables killed by the loop body is determined in lines 9-11. For each variable that is not an SVD variable the remaining graph is topologically sorted and the resulting tree of nodes removed from the graph. These nodes are killed by the loop body and are removed from the SVD variable set at line 11. Line 12 uses function *do_code* to locate the TCP statements with the loop body given the set of SVD variables that has just been calculated.

8.5.3 Complexity and Correctness

The algorithm of Figure 32 requires a single pass over each program loop with the possibility of V operations at each statement. The worst case runtime for the algorithm is $O(N \times V)$ and the space required is $O(N + V)$. The algorithm of Figure 35 requires a topological sort of the SVD graph costing $O(V)$ since a variable is only located a single time before it is removed from the SVD graph. The total time for the algorithm of Figure 35 is $(V + n)$, where n is the size of a loop body. The algorithm of Figure 34 traverses each node in the flow graph a single time with the possibility of V operations at each node. The worst case runtime for this algorithm is $O(N \times V)$ since $N \times V > N + V$, the runtime for the algorithm of Figure 34 . The algorithm requires $O(N + V)$ space.

8.6 Procedures and Functions

Procedures and functions complicate the TCP location problem. If a procedure or function can side effect the state of a program, then it has the possibility of affecting the SVD variable set immediately after execution of the procedure or function. For example, a function invocation on the right hand side of an assignment can change the SVD state of variables creating a situation where the left hand side of the assignment is either generated or killed by the assignment based on side effects to the SVD variable state as a result of executing the function. If procedures and

functions are not side effect free then they must be treated as inline sections of code at each potential invocation. For the worklist version of SVD variable approximation, the increase in complexity due to inline analysis of procedures and functions is $O(N \times E \times V \times P)$ where P is the total number of procedure and function invocations in the source program.

If procedures and functions are side effect free, the SVD graph representation of a function of procedure can be used to approximate the set of SVD variables introduced by the function or procedure body. This approximation requires a single pass over the body of each function and procedure. During the collection of SVD variables for the main program body, the set of SVD variables at *each* possible invocation of a procedure or function can be collected. That is, the set of SVD variables at the entry to a procedure or function is the union of the sets of SVD variables at each possible invocation of the function or procedure. A single application of the topological sorting algorithm at the end processing the main program, will suffice to approximate the set of TCPs in a procedure or function body. This simplification does not increase the complexity of either SVD variable approximation algorithm.

9 Measurement

The complexity of trace extrapolation, Theorem 7.1, implies that approximation techniques providing accurate simulation results must be found, or trace-driven simulation is not a valid technique for use with parallel programs exhibiting pattern nondeterminism. Theorem 7.3 demonstrates that for approximate trace extrapolation, it is not possible to ensure that a target execution is feasible. It remains to be shown whether or not the inaccuracy of a target execution can be bound.

Section 6, demonstrated that there are many possible approximations to trace extrapolation. In this section, we discuss approximate trace extrapolation and discuss the trade off between the complexity and the accuracy of the methods.

Measuring trace accuracy is for the most part an unexplored area of research. We demonstrate that there are fundamental limitations to the accuracy of measures that seek to place a bound on the accuracy of an execution. It remains to be demonstrated whether or not metrics for traces exist that provide a reasonable indication of the accuracy of simulation results for trace-driven simulation.

Ideally, researchers want several indicators of simulation accuracy. Two such indicators are the ability to correlate the level of inaccuracy of a trace with the accuracy of simulation results and the ability to correlate the sensitivity of a parallel program to the accuracy of an extracted a trace. The former would allow inaccurate traces to be used in simulations with a degree of certainty about the results of simulations. For the latter, an indication of the potential for a program to produce accurate (or inaccurate) traces may provide useful information for simulations results but may also indicate the degree of accuracy necessary for collecting useful traces from a given program. Given the results for static analysis of parallel programs in Section 4, the sensitivity of a parallel program to trace collection, if such a measure exists, will not be a static measure of a program. Any measure of the level of accuracy of a trace or of sensitivity will require the ability to actually measure a given metric over executions.

Clearly, there are many open questions in this area that will hopefully ultimately lead to a determination of whether or not parallel programs, the accuracy of traces from those programs, and simulations results based on potentially inaccurate traces can be correlated. In this section, we address a few of these questions. Specifically, the issues addressed are whether or not metrics can be calculated, whether or not useful metrics exist, and the degree to which accuracy can be measured.

9.1 The Accuracy of Approximate Trace Extrapolation

Figure 36 illustrates the process of approximate trace extrapolation for some S_k together with a distance measure D of the accuracy of an approximation. The actual target execution A is the

execution that would have resulted if program P had been executed in execution architecture M'. The source execution S is the result of executing the program on some parametrically different execution architecture. The target execution T is the result of extrapolating the source execution given a description of the target execution architecture $S(M')$, and the program source P.

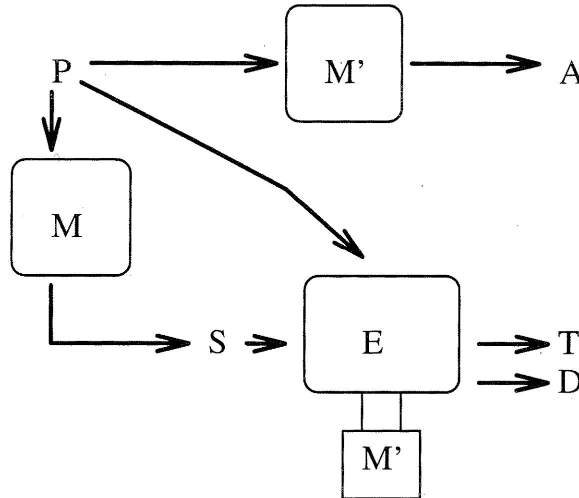


Figure 36: Trace extrapolation with error metric D

In Section 6, we characterized approximation of the trace extrapolation problem within our framework. When performing an approximation, there are two sets of executions under consideration: the set within which the actual approximation is performed; and the actual set of executions possible for a program given a fixed input. That is, in general the set of executions containing the target execution may be a different set than the set containing the actual target execution. Figure 37 demonstrates this process. As an example of this process, consider that one of the fastest methods of approximation uses the set ${S_{PATT}}^A$ for a given source execution and produces a target execution from this set. The actual set of executions for the program, the set containing the actual execution, may be larger, for example S_k for some k.

The choice of approximation (S_k) a priori determines the accuracy of the approximate extrapolation problem. A larger approximation set is more likely to contain the actual target execution. However, larger sets increase the complexity of trace extrapolation and require an increased number of infeasible executions to reduce the complexity. Thus increasing the chance of producing an infeasible target execution. Hence, there is a trade off between approximation accuracy and complexity of extrapolation.

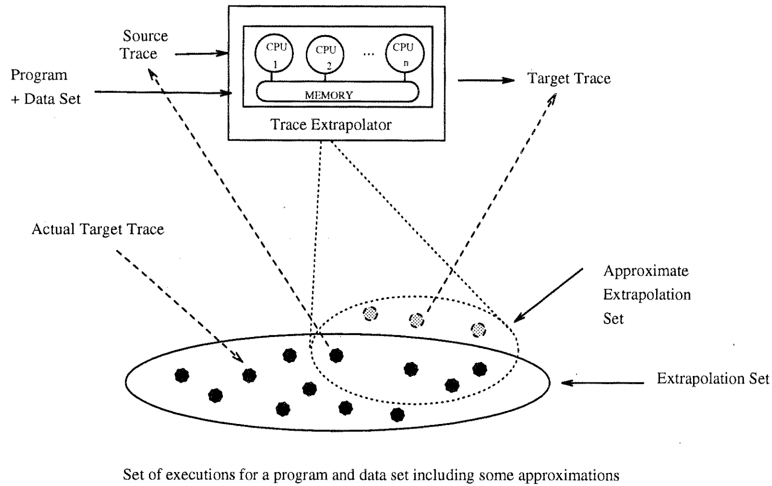


Figure 37: Possible executions under consideration in an approximate trace extrapolation method

9.2 The Accuracy of Measurement

The problem of determining the accuracy of an approximation is one of measuring how far, by some metric, the target trace is from the actual target trace. Recall that A represents the *correct* actual target execution. Consider Figure 38 below, which illustrates three possible distances between executions. These are the distance between the source and actual target execution, $d_{(s \to a)}$, the distance between the source and target execution d_f, and the distance between the target execution and the actual target execution $d_{(t \to a)}$. If execution A were available, then all of these distances are calculable with exact accuracy.

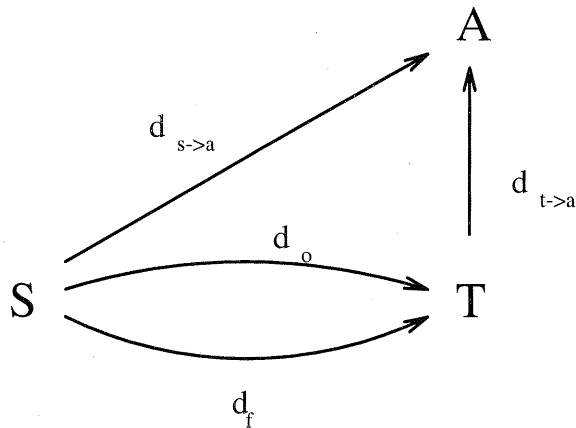


Figure 38: Possible distances between source, target, and actual target traces

For differences in the executions due to program state changes during extrapolation, the distance $d_{(s \rightarrow a)}$ is bounded by the sum of the distances d_f and $d_{(t \rightarrow a)}$. That is, the process of extrapolation and the use of program source code, provide some information about the actual target trace. This implies that using the program source and information about the process of execution that produced a target trace, a bound can be placed on the execution behavior, with respect to state changes, on a hypothetical machine. The distance d_f can be exactly calculated because both the source and target execution are present for measuring. However the distance $d_{(t \rightarrow a)}$ cannot be exactly calculated because the actual target execution is not present. How accurate an estimate of this distance will be (i.e., how tight the upper bound on $d_{(s \rightarrow a)}$ will be), remains to be demonstrated.

If a program is pattern deterministic, the $d_{(s \rightarrow a)}$ distance can be measured (e.g., because the pattern never changes, the distance is zero); the difficulty in measuring this distance is due to TCPs. Given a source and target execution, we call a TCP instance in the target execution *suspect* if it is possible that the outcome of the TCP could be different in the actual target execution. Similarly we call a variable *suspect* if the value of the variable at some point in the target execution has the potential of being different than it was at the same point in the source execution. Variables become suspect due to changes in the interleaving of access to shared memory (writes) or due to the possibility of executing alternate patterns that could change the value of the variable. Suspect TCP instances can be identified by checking if they are dependent on any suspect variables. The interleaving of instructions in the target trace determine the suspect state at any point in the execution.

The choice of approximation set defines the allowable search space for the target execution and dictates the accuracy of a measure. One advantage is that the search space is small enough that the search problem is tractable. If state changes are the only concern when determining the distance between the target and actual target executions, then an estimate of what might have occurred in the actual target execution can be made and a bound placed on the potential distance between the source and actual target executions with accuracy that is commiserate for the chosen approximation set (i.e., each approximation set introduces a degree of inaccuracy).

However, it is not reasonable to assume that identifying state changes that may occur between the target and actual target executions will suffice to provide a tight distance bound. This is because pattern nondeterministic programs suffer from sensitivity to time and TCPs become suspect not only due to changes in event interleaving but also due to shifts in time that occur during extrapolation. The difficulty with identifying suspect TCPs given time changes is that once a TCP is suspect, an execution has the potential to execute for some period of time along a different section of code. Unless this section of code is identical to the section executed in the source trace, changes to the execution time of individual processes will occur and the overall interleaving of program events will change.

```
A0: begin
      shared int x = 0, y = 0;
      lock lk1, lk2;
      int n = 2, z = 0;
A1: read(z);
A2: task_create(n)
```

```
task 0                          task 1
    int i,j,k,l;                    int p,q,r,s;

B0: lock(lk1);                  C0: lock(lk1);
B1: x = x - 5;                  C1: x = x + 1;
B2: l = x;                      C2: r = x;
B3: unlock(lk1);                C3: unlock(lk1);
B4: j = j + l;                  C4: if r > 0 then do
B5: k = k + l;                  C5:       p = p + x;
B6: j = k*l + j;                C6: else
B7: k = k + j;                  C7:       if z > 0 then
B8: lock(lk2);                  C8:            q = 2 * p;
B9: i = y;                      C9:            s = x + y - z;
B10:unlock(lk2);                C10:lock(lk2);
B11:if i > 0 then               C11:y = y + 1;
B12:      k = i;                C12:unlock(lk2);
B13:else
B14:      j = y;
```

```
A3: task_terminate(n);
A4: end;
```

Figure 39: A parallel program exhibiting pattern nondeterminism

To illustrate this problem, consider the example of Figure 39. Given a source trace from the program and an extrapolated target trace, extrapolated using the ${S_{PATT}}^A$ approximation, we demonstrate that state changes between the target and actual target executions can be bound. However, we cannot account for the effects due to time shifts during execution. The three executions in Figure 40 represent a possible extrapolation scenario. The first execution is a possible source execution. Here, task 1 acquires the lock (statement C0) first and the conditional at statement C4 subsequently branches positive (statement C5). The lock at statement C10 is acquired before the lock at statement B8, and the conditional at statement B11 branches positive. Extrapolating the source execution results in the target execution given in the figure. Notice that this execution is not feasible. The change in the order in which tasks 0 and 1 acquire the first lock guarantee that r is less than 0 at statement C4. It is a simple matter to notice that the execution may be infeasible and to measure what may have happened in the actual target execution. That is, the variable r is identified as suspect making the TCP C4 suspect. The alternate execution path for C4 containing statements C7 through C9 is the possible difference between the actual execution and the target execution. Using the actual target execution given in the figure, we consider the effects of time on difference between the actual target and target execution. Notice that we could not have predicted that the execution of statements C7-C9 would have caused an increase in the execution time in task 1 resulting in the change in the order of access to lock $lk2$. Task 0 acquires the lock first and the result is that the outcome of the TCP at B11 is changes. The variable i should be identified as suspect, and the TCP B11 identified as a suspect TCP, for an accurate measure. However, this identification cannot be made without accounting for time shifts between the target and actual target executions[26].

9.3 Measuring the Accuracy of an Execution

Given that an exact computation of the distance between the target execution and the actual target execution is impossible, the accuracy of a target execution must be characterized by some other means. The work of the previous sections suggests that the accuracy of an extrapolated execution for a target execution architecture takes on several dimensions. The execution should be a feasible execution. In general, any trace used for simulation should be representative of the overall program behavior. Given that the extrapolated execution may not be the execution that would have occurred (i.e., given that we have an approximation), it should be *representative* of the behavior of the program. Finally, the actual target execution is the most likely execution for

[26] Accurate measurements may be possible regarding the effects of shared variable values changing given the potential for time shifts. However, it should be noted that the number of paths for a divergent portion of code are exponential in the number of TCP instances possible during a period of divergence. Computing any measure requiring a calculation on all possible paths during a divergent section of code will be an intractable problem.

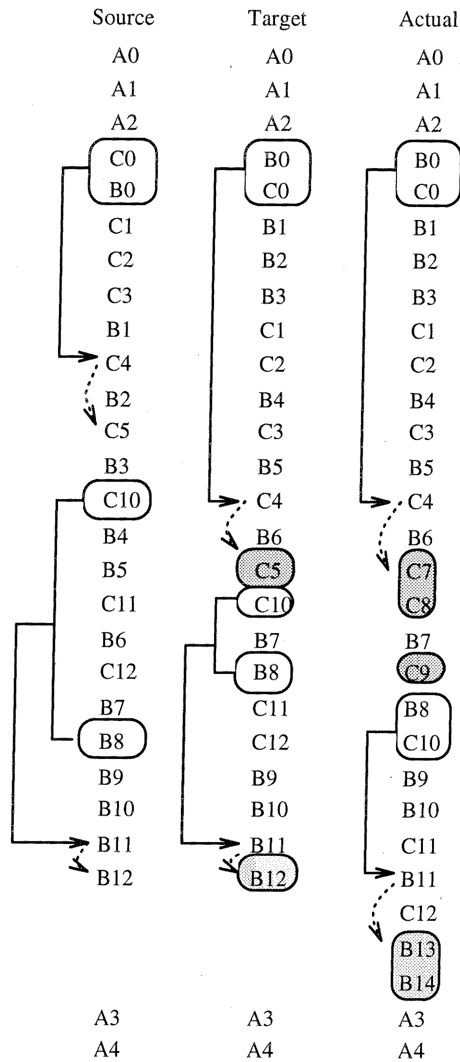


Figure 40: A possible source, target, and actual execution for the program in Figure 9.4

the given input, in the sense that it represents an execution path that the program would have taken given the target execution architecture. One possible indication of the accuracy of the target execution is that of how likely it was to have occurred given the target execution architecture.

To bound the feasibility of the target execution, we propose a measure of the distance from that target execution to the only execution known to be feasible, the source execution. This measure, the difference between the source and target executions taken as sequences, can be computed (see d_f in Figure 40). The measures places an upper bound on the correctness of the target execution and relies on changes in the interleaving of shared variable accesses actually changing the outcome of TCPs. Exactly how accurate this measure is depends on the degree of data nondeterministic and pattern deterministic behavior that the program exhibits. For data nondeterministic and pattern deterministic programs, this measure will be inaccurate. We call this the *feasibility distance* for a target execution.

The notions of representative and likely might be easily confused. Representative program behavior is desirable in order to acquire accurate simulation results. A representative trace is one that accurately characterizes the workload presented by the program. Whereas a likely trace may be indicative of program behavior but may be an outlier in the set of possible executions for a program in that it represents a behavior that is not representative of the entire set of executions. It is reasonable therefore to characterize a behavior as *representative* if it is indicative of the set of executions for a program. An approximation reduces the size of the set of executions and will make it possible to compute some metrics of representativeness. The accuracy of these computations depends on the metrics and the size of an approximation set.

Currently, we know of a single characterization of representative for a parallel program: in their work, Borg, Kassler, and Wall experimentally demonstrated that very long address traces are required for accurate cache simulations [7]. One could infer from this work that for a given parallel program, very short traces, however likely they may be, are not representative.

An example of how we might characterize representable in our model is the following. Let an execution be representable if it is above average in length for the choice of approximation set, S_i, of executions. Using the method of locating suspect TCPs from the previous section, the average, maximum and minimum length of an execution in S_i can be approximated from the source execution and source code of the program. Using the method of locating suspect TCPs from the previous section, we can also bound the distance of the target execution from the actual execution (up to the effects of time). This measure is also an example of a *representative distances.*

Finally, determining whether or not an execution is likely to occur given a program, data set, and execution architecture, is a difficult problem. We propose a measure that is an upper bound on how likely a execution is to occur. Consider that if a program has a single execution path, then that path is likely to occur (with probability one) for any execution of the program (i.e., the program

has no chance of diverging from the path). Recall that divergence from an execution path occurs at a TCP. If during extrapolation, the target trace continually tries to move from the pattern specified by the source trace, then the resulting target trace was not likely to have occurred given the target execution architecture. A count of the number of TCPs whose outcome is in question for an target execution is a metric that provides an upper bound on the likelihood that the execution pattern would have actually occurred as the actual target execution pattern. We call this metric the *likely distance* for an execution, denoted in Figure 40 by d_o.

We claim that the accuracy of a representative distance measure depends on the TCP distance between any two executions in the approximation of the actual set of executions for a program and data set. This is because of the inherent complexity of measuring what might have occurred in an alternate execution. That is, given that the set of possible paths through a divergent period is exponential in the number of TCP instances encountered during that period, an accurate measure will require an intractable computation. Note that the likely and feasibility distances, do not consider alternate sets of executions. Instead, they attempt to measure the behavior of the target execution in the target environment via the extrapolation process.

9.4 Computing Distance Metrics for Executions

The issue of actually computing distance metrics between parallel program traces is considered in this section. Sequence analysis techniques will be necessary for measuring the differences between source and target executions. During the 1960's sequence comparison was an active area of research for such practical problems as string correction and editing. During the 1970 and early 80s, sequence comparison found applications in molecular biology, human speech recognition, encoding information and error checking, and a large variety of other scientific applications. Today, there is a large body of work in sequence comparison. Hall and Dowling [28] provide a good overview of sequence comparison in computer science, while Sankoff and Kruskal [48] provide a more detailed presentation of sequence comparison.

Sequence comparison seeks to compute *distance measures* and determine optimal *correspondences* between sequences. An optimal correspondence for two sequences is one that minimizes a particular distance. A sequence is an ordered list of elements taken from an alphabet. If two sequences differ, then it is a result of substitutions, insertions and/or deletions, compression and/or expansions, and/or transposition (or swaps). A substitution is a replacement of one element for another. Insertions or deletions are simply the insertion or deletion of an element to or from a sequence. A compression is a replacement of two or more elements with a single element while an expansion is a replacement of a single element by two or more elements. A transposition is an interchange of two adjacent elements.

Distance calculations, such as Hamming distance (the number of positions in which two sequences differ) and Euclidean distance ($\left[\Sigma_{i=1}^{n}(a_i - b_i)^2\right]^{1/2}$) are intended to be used on sequences of the same length. For both of these calculations, it is assumed that no further correspondence of the sequences need be computed. Knowing the correspondence between two sequences a priori simplifies the comparison process in that a correspondence identifies comparable elements in two sequences. Given this information, a distance calculation, such as the two above, can be computed in time linear in the size of the input sequences. In addition, the space requirements for such a calculation are constant.

Unfortunately, it is possible that either a correspondence is not known a priori (e.g., the sequences differ in length) or a more accurate distance measure is needed. In either case, a correspondence must be computed. Levenshtein distances (and other similar distances), are an example of potentially more accurate distance measures. These are the smallest number of substitutions, inserts and deletes require to transform one sequence into another and the smallest number of insertions and deletions required to transform one sequence into another. These distance calculations require selection of a correspondence that minimizes the distance functions and for this reason require quadratic time in the size of the input sequences and storage of these sequences.

Currently, we know of one example of a comparison algorithm that could be directly applied to two program executions. This is the UNIX[27] *diff* file comparison filter. This filter, among other things, provides a Levenshtein distance between two files. Therefore, if it were possible to store executions on file, then diff would provide a measure in $O(n^2)$ time.

In the next section, we present a method for corresponding executions on-the-fly that results in the ability to calculate each of the measures presented in this section on-the-fly in linear time and constant space. In addition, this method provides a means of performing other distances, such as the Levenshtein distances, in optimal time (although the distance computed may not be optimal).

[27]UNIX is a trademark of Bell Laboratories.

10 Comparing Executions

The need to compare two executions from a parallel program arises from the need to measure the distance between parallel program executions. While algorithms exist that compute a correspondence between two sequences in linear time, see [20], these algorithms will not work in linear time for large sequences. Executions cannot be corresponded, in a reasonable amount of time and space, based solely on their information content. We propose that a reasonable algorithm can only be achieved by using information gathered from program source to a priori correspond executions[28]. That is, the program can be a valuable source of information which, as we show, can direct the correspondence of two executions. We present an algorithm for corresponding executions based upon program source information. It should be noted that this correspondence is not guaranteed to be optimal however the comparison is computed on-the-fly. The algorithm computes a correspondence between two executions and can be modified to produce a distance measure.

10.1 Correspondence

Before any results about corresponding executions can be proved, we need a formal notion of correspondence. Our goal is to use correspondence to efficiently and correctly compare executions. Therefore, any reasonable correspondence must adhere to the following three rules. First, Corollary 5.12 implies that any reasonable correspondence will be an onto mapping between TCP and CP instances. Second, whenever two event instances are corresponded, they should be instances of the same instruction. Finally, a correspondence must obey the temporal and shared data dependence relations for the execution.

Definition 10.1 *Given two executions* $P_1 =< E_1, \rightarrow_{T_1} >$ *and* $P_2 =< E_2, \rightarrow_{T_2} >$ *(from the same program) with common TCP and CP sets* $S_{TCP} = TCP(E_1) \cap TCP(E_2)$ *and* $S_{CP} = CP(E_1) \cap CP(E_2)$*. A* correspondence *of those executions* $M(P_1, P_2)$*, is a mapping from* $S_{TCP} \cup S_{CP}$ *to* $S_{TCP} \cup S_{CP}$ *such that:*

1. *M is onto.*
2. *whenever M maps one instruction instance to another, these instructions are an instance of the same instruction.*
3. *if M maps* $a \in E_1$ *to* $b \in E_2$ *and* $c \in E_1$ *to* $d \in E_2$ *then* $a \rightarrow_{T_1} c$ *implies that* $b \rightarrow_{T_2} d$*.*

□

[28] It is in general against the grain of sequence comparison to a priori identify a correspondence between sequences. However in this case, as we have pointed out, there is no other reasonable algorithm.

10.2 The Theorem of Correspondence

Given Definition 10.1, the Theorem of Correspondence (below) demonstrates that a correspondence between executions adhering to Definition 10.1 does exist. Lemma 10.2 is the base case for the Theorem of Correspondence. The base case consists of proving that two arbitrary executions can be corresponded at the first level of nesting of **task_create/task_terminate** statements. Simply put, we can correspond iterative statements, array and pointer references, and conditional statements, but we can only pin the outside of **task_create/task_terminate** statements (i.e., we cannot pin the contents of these statements).

Lemma 10.2 *Given any two executions of the same program there exists a correspondence M of the executions up to but not including the body of* **task_create/task_terminate** *statement instances.*

□

Proof: We prove Lemma 10.2 by induction on the number of TCP instances that two executions have in common.

Base Case: Every execution has the begin statement in common. If the program contains no other TCPs, then by Lemmas 5.11 and 5.13 it is guaranteed that the end statement will be located and that the correspondence includes only the begin and end statements from each execution. Clearly this correspondence fulfills conditions 1-3 of Definition 10.1.

Induction Hypothesis: Given fixed but arbitrary executions $P_1 =< E_1, \rightarrow_{T_1} >$ and $P_2 =< E_2, \rightarrow_{T_2} >$, assume that there exists a correspondence $\mathbf{M_k}$ corresponding prefixes $p_1(begin, s_k) =< E_{p1}, \rightarrow_{T_{p1}} >$ and $p_2(begin, s_k) =< E_{p2}, \rightarrow_{T_p2} >$ and given $\mathbf{S_k} = (TCP(E_{p1}) \cap TCP(E_{p2})) \cup (CP(E_{p1}) \cap CP(E_{p2}))$, $s_k \in \mathbf{S_k} and |\mathbf{S_k}| = k$.

Induction Step: There are three possible cases,

1. $s_k \in (CP(E_{p1}) \cap CP(E_{p2}))$. There are two cases. First, if s_k is the instance of the program **end** statement, the $\mathbf{M_k}$ corresponds the two executions P_1 and P_2. Otherwise, given the TCP successor of s_k, call these TCPs $s_{1,k+1}$ and $s_{2,k+1}$ respectively, Lemma 5.13 demonstrates that there exist equivalent paths $p_1(s_k, s_{1,k+1})$ and $p_2(s_k, s_{2,k+1})$ and that $s_{1,k+1} = s_{2,k+1}$. This implies that there exists correspondence $\mathbf{M_k + 1} = \mathbf{M_k} \cup \{s_{1,k+1}, s_{2,k+1}\}$ corresponding prefixes $p_1(\mathbf{begin}, s_{1,k+1})$ and $p_2(\mathbf{begin}, s_{2,k+1})$.

2. $s_k \in (TCP(E_{p1}) \cap TCP(E_{p2}))$ and is an instance of a **task_create** statement. Recall that M is a correspondence that corresponds E_1 and E_2 but does not correspond the body of **task_create/task_terminate** statements. It is reasonable then to assume that the paths from s_k to its corresponding CP, call these events $c_{1,k+1}$ and $c_{2,k+1}$ respectively, in each execution are

disjoint (excluding s_k and c_{k+1} from the paths). Lemma 5.11 ensures that $c_{1,k+1}$ and $c_{2,k+1}$ can be located and that $c_{1,k+1} = c_{2,k+1}$. Therefore we have correspondence $\mathbf{M_{k+1}} = \mathbf{M_k} \cup \{c_{1,k+1}, c_{2,k+1}\}$ corresponding prefixes $p_1(\mathbf{begin}, c_{1,k+1})$ and $p_2(\mathbf{begin}, s_{2,k+1})$.

3. $s_k \in (TCP(E_{p1}) \cap TCP(E_{p2}))$ (but is not an instance of a **task_create** statement). There are two possible cases depending upon the outcome of s_k in each execution. First, consider the case where Outcome(P_1, s_k) = Outcome(P_2, s_k). Then given the TCP successor of s_k, call these TCPs $s_{1,k+1}$ and $s_{2,k+1}$ respectively, Lemma 5.13 demonstrates that there exist equivalent paths $p_1(s_k, s_{1,k+1})$ and $p_2(s_k, s_{2,k+1})$ and that $s_{1,k+1} = s_{2,k+1}$. This implies that there exists correspondence $\mathbf{M_{k+1}} = \mathbf{M_k} \cup \{s_{1,k+1}, s_{2,k+1}\}$ corresponding prefixes $p_1(\mathbf{begin}, s_{1,k+1})$ and $p_2(\mathbf{begin}, s_{2,k+1})$. Next, consider the case where Outcome(P_1, s_k) $\neq$ Outcome(P_2, s_k). Then by Lemma 5.14 there exists $c_{k+1} \in (CP(E_1) \cap CP(E_2))$ and that given the predecessor of c_{k+1} in E_1, call this event c_{1p}, the predecessor of c_{k+1} in E_2, call this event c_{2p}, the successor of s_k in E_1, call this event s_{1p}, and the successor of s_k in E_2, call this event s_{2p}, then there exist paths $p_1(s_{1p}, c_{1p})$ and $p_2(s_{2p}, c_{2p})$ that are disjoint. Since there can be no correspondence between disjoint paths we have correspondence $\mathbf{M_{k+1}} = \mathbf{M_k} \cup \{s_{1,k+1}, s_{2,k+1}\}$ corresponding prefixes $p_1(\mathbf{begin}, s_{1,k+1})$ and $p_2(\mathbf{begin}, s_{2,k+1})$.

Given Assumption 1, there exists a correspondence M between any two program executions (of the same program on the same data set) that corresponds the executions up to the body of **task_create/task_terminate** statement instances. □

The Theorem of Correspondence is proved using induction on the nesting level of **task_create/task_terminate** statements given Lemma 10.2 as the base case.

Theorem 10.3 Theorem of Correspondence: *Given any two executions of the same program there exists a correspondence M of the executions.*

□

Proof: We prove the theorem by induction on the **task_create/task_terminate** nesting level of the executions.

Base Case: The basis for the induction is Lemma 10.2.

Induction Hypothesis: Given fixed but arbitrary executions $P_1 =< E_1, \rightarrow_{T_1} >$ and $P_2 =< E_2, \rightarrow_{T_2} >$ with common TCP and CP instances $\mathbf{S} = (TCP(E_1) \cap TCP(E_2)) \cup (CP(E_1 \cap CP(E_2))$. For the induction hypothesis assume that there exists $\mathbf{M_k}$ corresponding P_1 and P_2 up to but not including the body of **task_create/task_terminate** instances at nesting level k.

Induction Step: Without loss of generality, choose a **task_create/task_terminate** instance pair $s_{1,k} \in \mathbf{S} \cap E_1$ and $c_{1,k} \in \mathbf{S} \cap E_1$ such that $s_{1,k}$ is a **task_create** instance at nesting level k and $c_{1,k}$ the corresponding **task_terminate** instance at nesting level k. Then by the induction hypothesis

there exist $s_{2,k} \in \mathbf{S} \cap E_2$ and $c_{2,k} \in \mathbf{S} \cap E_2$ such that $(s_{1,k}, s_{2,k}) \in \mathbf{M_k}$ and $(c_{1,k}, c_{2,k}) \in \mathbf{M_k}$. There exist paths $p_1(s_{1,k}, c_{1,k})$ and $p_2(s_{2,k}, c_{2,k})$ which contain n threads between which there exists a one-to-one mapping. Lemma 10.2 implies that for each thread there is a correspondence up to nesting level one, or $\cup_{i=1}^{n} \mathbf{m_i}$. This implies that the correspondence defined by $\mathbf{M_k} \cup (\cup_{i=1}^{n} \mathbf{m_i})$ corresponds E_1 and E_2 up to **task_create/task_terminate** instances at nesting level $k+1$. Therefore, there exists a correspondence M between any two executions of the same program (on the same data sets).

□

10.3 Correspondence Algorithm

In this section we present an algorithm for corresponding two program executions based upon the Theorem of Correspondence. The algorithm takes as input a parallel program with its TCP and CP sets already computed and two executions from that program. The algorithm uses a preprocessing phase and a correspondence phase. In the preprocessing phase, the program is transformed into an intermediate form that will, during the comparison phase, serve to direct the traversal of the executions. In this section, we present the correspondence algorithm in the form of a function that returns a correspondence. Naturally, the correspondence itself can require a large amount of space for storage. In reality, the correspondence algorithm should be coupled with an algorithm computing some measure of difference for the executions and the correspondence should not be stored. We discuss this coupling in the next section.

10.4 The Preprocessing Phase

10.4.1 The TCP Basic Block Flow Graph

Recall that a basic block is a section of code that has a single entry and exit point. Basic blocks are groupings of logically indivisible statements. Basic blocks can be used to form a basic block flow graph of a program. A *TCP basic block* (TBB) represents a section of code that produces a logically indivisible *execution* outcome. For example, a basic block would have embedded array references and a TBB would have a single array reference (that is a TCP) per block. The TBB flow graph is a flow graph in which the nodes of the graph are the TBBs of the program. The conditions for delimiting a TBB are[29]:

1. Each block contains at most a single TCP.

[29] There are several differences between our TBB flow graph and the ABB flow graph of introduced by Holliday and Ellis [32]. Most notable is the addition of CP information and CPs as delimiters for ABBs as well as the requirement that a single TCP occupy each block.

2. Each block begins either with a CP or the target of a branch statement. The exception is the first block containing the begin statement.

3. Each block ends with an TCP, a branching instruction whose target is a CP, or an instruction immediately preceding a CP. The exception is the last block containing the end statement.

In addition, we add a **no-op** instruction to each block containing an array reference or pointer reference TCP immediately preceding the reference instruction.

The TBB flow graph will serve to direct the rapid traversal of the executions being compared. In order to facilitate this process, each TBB is decorated with additional information. Associated with each block is an instruction count for the block (not including any **no-op** instructions), a flag denoting whether or not the block contains a TCP, a flag denoting whether or not the block contains a CP, and a pointer to the block containing the CP corresponding to the TCP in this block (if there is a TCP in this block).

This information is designed to facilitate rapid traversal of groups of instruction instances that are instances of a particular TBB. That is, consider a TBB b with instruction count n, and consider an instance of block b in an execution. If it has been determined that the information in the block instance is uninteresting, then it can be rapidly traversed by moving forward n instruction instances in the execution. alternatively, if the block needs to be traversed to the end, then n-1 instruction instances are traversed. Given Lemma 5.13 whenever two equivalent TCP instances with the same outcome are located in the executions being compared, the executions are equivalent up to the next TCP. That is, the instruction instances up to the next TCP are uninteresting and can be rapidly traversed using the TBB flow graph by traversing blocks until a block with a TCP is located, using the TCP flag, and moving to the end of that block. Corollary 5.14 implies that a similar traversal will work for traversing from a CP instance to a TCP instance. Finally, disjoint sections of executions can be rapidly traversed independently by traversing an execution until the block containing the CP, for the TCP instance at the head of the disjoint section, is located. The CP pointer stored in each block is used to locate the correct block. In fact, the only association of TCP and CP pairs is through this pointer.

We forego an algorithm to compute the TBB flow graph of a program and point out that it is a simple matter to construct this data structure given a program and the set of TCP and corresponding CPs for the program. The time to construct this data structure is $O(k)$ where k is the length of the input program.

given: Executions E_1 and E_2 and the TBB flow graph for a program p_{TBB}
compute: M – the correspondence of E_1 and E_2
algorithm:

```
1 Compute_M(E1, E2, pTBB):
2       record State = {
              boolean advance = TRUE;
              instruction * s1, s2;
              tbb * b1, b2;
        };
3       correspondence M;
4 while not end(E1) or end(E2) do
5       if State.advance then
6             State = get_TCP(E1, E2, pTBB, State);
7             M = M ∪ {s1, s2};
8       if (s1 ≡ s2 ≡ task_create) then
9             for each process pair p do
10                  M = M∪ Compute_M(E1(p), E2(p), pTBB(p));
11            State = get_CP(E1, E2,State);
12            M = M ∪ {s1, s2};
13      if outcome(E1, s1) ≠ outcome(E2, s2) then
14            State = get_CP(E1, E2, State);
15            M = M ∪ {s1, s2};
16end while
17return M;
18end.
```

Figure 41: An algorithm to compute a correspondence M given two executions and a program in TBB form.

10.4.2 The Comparison Phase

Algorithm 41 below, computes a correspondence M for two program executions. This correspondence is the one discussed in the previous section. The algorithm relies on the assumption that an execution is in graph form rather than interleaved sequence form.

The correspondence algorithm is a recursive algorithm. Three data structures are used in making the correspondence. First the TBB flow graph of the program, p_{TBB}, is used to direct the traversal of each execution. Second (line 2) a data structure sufficient to represent the correspondence function M is used. Finally, the state of the executions and the current position in the TBB flow graph if each execution is retained in the record *State*. This record maintains a pointer to the current location in each execution, a pointer to the corresponding locations in the TBB flow graph

for each execution, and a flag with which to determine whether or not the executions need to be advanced to locate the next TCP instruction.

This algorithm uses the TBB flow graph of the program to *direct* the traversal of the executions. The instruction count for each block enables the determination of exactly how many instructions to pass over in each execution in order to arrive at the next TCP instance. This is also the case when locating CP instances. In this manner, the TBB flow graph serves as a *road map* for traversing the traces. Two functions accomplish this traversal: *get_TCP* and *get_CP*. These functions return the current state of the correspondence in the record *State*. Recall that the TBB has a single TCP per block. The comparison begins at the first block in the program, containing the **begin** statement, and uses the information in this block to traverse to the next TCP instances in both executions. Here, the executions are identical at least until the second TCP instance is reached. In the event that **end** statement instances are reached, *get_TCP* returns the state at the end of each execution. The case of locating a CP in which the executions are guaranteed to be disjoint illustrates the need for two separate pointers into the TBB flow graph. This function determines whether or not the CP instances located are also TCP instances. If this is the case, then *State.advance* is false and *get_TCP* does not advance the state at line 5. Finally, the function *end* is used to detect the end of each trace, and in the case of recursion, the end of each process execution.

The algorithm works as follows. At the beginning of each iteration of the main loop (line 4) both executions are in an equivalent section of instruction instances. Initially, this section is that of the beginning of the program to the second TCP (the **begin** statement being the first). A TCP instance is located (line 5) and added to M (line 7). There are three cases for the TCP instances. First, if the TCP are **task_create** instances (line 8) then each pair of process executions are compared. The result of each of these comparisons are added to M. This is accomplished by recursively calling *Compute_M* with pointers to each pair of process executions and a pointer to the beginning block (in the TBB flow graph) for these executions. Notice that since the executions begin with the same block of the TBB flow graph, a single pointer suffices. Finally, the CP corresponding to the **task_create** statement is located in each execution and added to M (lines 11-12).

The next case is one in which the TCPs have a different outcome (line 13). Here, the executions are traversed and the CP for this TCP instance is located in each execution. At this point (line 14) *State* contains the updated state of the traversal (i.e., the updated pointers into each execution and the pointers to the CP in the TBB flow graph). The CP instances are added to M (line 15).

In the final case, the outcome of the TCP instances are the same. In this case, the loop drops through and continues. Here, the executions are again equivalent at least until the next TCP instance.

10.5 Correctness and Complexity

10.5.1 Correctness

To prove the algorithm of Figure 41 correct, we first show that the loop at line 4 correctly maintains the following invariant:

(I1) At each iteration of the loop, a prefix of the executions is correctly corresponded.

We then demonstrate that the loop terminates. Lemma 10.4 below shows that given the assumption that lines 9-10 of the algorithm do nothing more than move the executions past the p process threads created by a **task_create** statement then the following invariant is maintained:

(I2) At each iteration of the loop, a prefix of the executions is correctly corresponded excluding the contents of **task_create** and **task_terminate** statement instances.

Lemma 10.4 *Given any two executions of the same program, the algorithm of Figure 41 maintains invariant I2.*

□

Proof: We prove Lemma 10.4 by induction on the number iterations of the loop at line 4.

Base Case: Clearly it is not possible for the loop to fail to iterate since a program must at least have a **begin** and **end** statement. Therefore, the first iteration of the loop locates the **begin** statement instances in each execution. Line 7 adds these statements to the correspondence M_1. The conditions at lines 8 and 13 fail and the next iteration begins. Since M_1 does not violate conditions 1-3 of Definition 10.1, the invariant I2 is maintained.

Induction Hypothesis: Given fixed but arbitrary executions $P_1 =< E_1, \rightarrow_{T_1} >$ and $P_2 =< E_2, \rightarrow_{T_2} >$ assume that there exists a correspondence $\mathbf{M_i}$ that does not violate invariant I2 and assume that the executions are corresponded up to the kth common TCP or CP in each execution. That is, there exist corresponding prefixes $p_1(\mathbf{begin}, s_k) =< E_{p1}, \rightarrow_{T_p1} >$ and $p_2(\mathbf{begin}, s_k) =< E_{p2}, \rightarrow_{T_{p2}} >$ and $\mathbf{S_k} = (TCP(E_{p1}) \cap TCP(E_{p2})) \cup (CP(E_{p1}) \cap CP(E_{p2}))$, $s_k \in \mathbf{S_k}$ and $|\mathbf{S_k}| = k$.

Induction Step: For the induction step, if s_k is not an instance of the program **end** statement, then it is either a CP or a TCP. If s_k is a CP, then Corollary 5.12 guarantees that the executions are the same up to (and including) the next TCP instance in each execution. Therefore, these equivalent instances will be located at line 5. The same holds true is s_k is a TCP by Lemma 5.13. Line 7 creates the correspondence $\mathbf{M_i'} = M_i \cup (s_{k+1}, s_{k+1})$ which does not violate conditions 1-3 of Definition 10.1 since by the induction hypothesis $\mathbf{M_i}$ does not and since there exist paths $p_1(s_k, s_{k+1}) = p_2(s_k, s_{k+1})$. Given this, there are three cases for the TCP located, s_{k+1}:

1. $Outcome(p_1, s_{k+1}) = Outcome(p_2, s_{k+1})$. In this case, the conditions at lines 8 and 13 both fail and the loop falls through. Since $\mathbf{M_{i+1}} = \mathbf{M'_i}$ is a correct correspondence, condition I2 is maintained.

2. $Outcome(p_1, s_{k+1}) \neq Outcome(p_2, s_{k+1})$. Then the condition at line 8 fails and the condition at line 13 is true. Then by Corollary 5.12 there exists $c_{k+1} \in (CP(E_1) \cap CP(E_2)$ and Lemma 5.11 ensures that c_{k+1} can be located. Given the predecessor of c_{k+1} in E_1, call this event c_{1p}, the predecessor of c_{k+1} in E_2, call this event c_{1p}, the successor of s_k in E_1, call this event s_{1p}, and the successor of s_k in E_2, call this event s_{2p}, then there exist paths $p_1(s_{1p}, c_{1p})$ and $p_2(s_{2p}, c_{2p})$ that are disjoint. Condition 2 of Definition 10.1 implies that there can be no correspondence between disjoint paths. Hence, line 15 yields the correct correspondence $\mathbf{M_{i+1}} = \mathbf{M'_i} \cup (c_{k+1}, c_{k+1})$ and invariant I2 is maintained.

3. s_{k+1} is an instance of a **task_create** statement. In this case, by assumption, lines 9 and 10 move over the process executions in each execution. The **task_terminate** CP, c_{k+1}, is located by Lemma 5.11 and line 11 creates the correspondence $\mathbf{M_{i+1}} = \mathbf{M'_i} \cup (c_{k+1}, c_{k+1})$. Since corresponding (c_{k+1}, c_{k+1}) does not violate Definition 10.1, the invariant I2 is maintained.

□

Next, given Lemma 10.4 we prove Theorem 10.5.

Theorem 10.5 *Given any two executions of the same program the algorithm of Figure 41 computes a correct correspondence of the executions.*

□

Proof: It suffices to show that the algorithm does not violate invariant I1. The proof proceeds by induction on the **task_create/task_terminate** nesting level of an execution.

Base Case: The proof of Lemma 10.4 is sufficient to demonstrate that invariant I1 is maintained by function *Compute_M* for executions with no common **task_create/task_terminate** statement instances.

Induction Hypothesis: Given fixed but arbitrary executions $P_1 = < E_1, \rightarrow_{T_1} >$ and $P_2 = < E_2, \rightarrow_{T_2} >$ with common TCP and CP instances $\mathbf{S} = (TCP(E_1) \cap TCP(E_2)) \cup (CP(E_1 \cap CP(E_2))$, assume that there exists $\mathbf{M_i}$ not violating invariant I2 corresponding P_1 and P_2 up to but not including the body of **task_create/task_terminate** instances at nesting level i.

Induction Step: For the induction step, without loss of generality, choose a **task_create/task_terminate** instance pair $s_{1,i} \in \mathbf{S} \cap E_1$ and $c_{1,i} \in \mathbf{S} \cap E_1$ such that $s_{1,i}$ is a **task_create** instance at nesting level i and $c_{1,i}$ the corresponding **task_terminate** instance at nesting level i.Then by the induction hypothesis there exist $s_{2,i} \in \mathbf{S} \cap E_2$ and $c_{2,i} \in \mathbf{S} \cap E_2$ such

that $(s_{1,i}, s_{2,i}) \in \mathbf{M_i}$ and $(c_{1,i}, c_{2,i}) \in \mathbf{M_i}$. There exist paths $p_1(s_{1,i}, c_{1,i})$ and $p_2(s_{2,i}, c_{2,i})$ which contain n process executions between which there exists a one-to-one mapping, line 9. Lemma 10.4 implies that for each process execution there is a correspondence up to nesting level one, or $\cup_{j=1}^{n} \mathbf{m_j}$ computed in line 7. This implies that the correspondence $\mathbf{M_{i+1}} = \mathbf{M_i} \cup (\cup_{j=1}^{n} \mathbf{m_j})$ upholds invariant I1.

□

Finally, we are guaranteed by Assumption A1 that programs terminate and hence there is an instance of an **end** statement in each execution. If the statement is a CP of some statement other than the **begin** statement, then it is located at line 13 and the loop terminates on the next iteration. Otherwise, it is located at line 5 and lines 8 and 12 fail and the loop terminates on the next iteration.

10.5.2 Complexity

Function *Compute_M* has linear time complexity because each instruction instance in each execution is visited only once and immediately discarded. The function need not be recursive and is written this way solely for ease of understanding. In addition, the comparison algorithm is intended to be a vehicle for making accurate distance measure calculations. For this reason, the correspondence itself need not be stored. Given this, a correspondence can be made using $O(k)$ space where k is the size of the program input. For parallel programs, the size of the program is likely to be negligible in relation to the size of the executions it produces.

We demonstrate that function *Compute_M* has linear time complexity. Function *get_TCP* traverses each execution either from a TCP instance to its TCP successor instance or from a CP instance to the next TCP instance in the execution. Lemma 5.13 and Corollary 5.12 (respectively) ensure that a simple linear search of the executions is sufficient for function *get_TCP* used at line 5. Similarly, Lemma 5.13 and Lemma 5.14 ensure that using a simple linear search will suffice for function *get_CP* used at lines 11 and 14. Finally, the recursive call at line 10 will traverse each pair of process executions a single time. These executions have not been visited before the recursive call and will not be visited after. Since lines 5, 10, 11, and 14 all represent linear traversals of the execution and since no other lines consume execution instances, the function has linear time complexity in the length of the input traces. Coupled with the preprocessing phase, if the executions are of length n and m and the program is of length k, then the complexity of function *Compute_M* is $O(k + m + n)$.

Given that the correspondence is not stored and that correspondence algorithm need not be recursive, we demonstrate that the space complexity for function *Compute_M* is $O(k)$. First, storing the program in intermediate form accounts for the majority of storage consumption. Lines 5, 11, and 14 get a TCP or CP instance from each execution, which can be immediately discarded at lines

7, 12, and 15. Given that the information collected at line 5 can be discarded at line 8, the only information used after a recursive call is the CP pointer for the CP associated with the **task_create** TCP that caused the recursive call. Notice that this CP is the *current* instruction instance in each execution *and* is the current instruction of the intermediate form. Since it can be easily located, there is no reason to store a pointer to the CP in the intermediate form. Hence the function stores only the program intermediate form and the a single copy of the record *State*.

Two assumptions were made in order to achieve the above time and space bounds. The first, assumption A2, guarantees that a statement has a single entry and exit point. There may be no lemmas that correspond to Lemmas 5.11, 5.13, and 5.14 in the absence of this assumption. It seems almost a certainty that executions will require more than a linear search if the requirement is lifted.

The second assumption is that process executions are not stored in interleaved form. If the correspondence algorithm is required to undo the interleaving of process executions, then at worst case, for a **task_create** statement one half of all of the process executions in both executions must be stored before process executions to correspond with the process executions already stored are encountered. This implies not only that the space requirement for the correspondence is at worst case $O(n + m)$ but that a second traversal of each process execution may be needed. This does not however change the worst case linear time for the correspondence.

10.6 Computing Optimal Correspondences

As previously mentioned, our choice of correspondence was not the only possible choice. In this section, we justify the choice we have made. Ideally, one would like an algorithm that computes an optimal correspondence for any pair of executions w_1 and w_2 and measure m. We conjecture that there is no such algorithm that satisfies the time and space requirements imposed by executions.

We begin with the observation that any correspondence computation that requires searching the input executions, will fail to meet time and space requirements. It therefore seems reasonable that the constraints of O(1) space usage and linear time imply that any algorithm computing a correspondence between two parallel program execution must compute a fixed correspondence.

Consider another choice of correspondence. In particular, the correspondence that corresponds loop iterations in reverse order. This leaves any unmatched iterations as the first iterations of the loop. Even this simple correspondence requires some searching – that of locating the final iterations of each loop. This correspondence will require at least O(n) space.

Next, we conjecture that there is no algorithm satisfying the linear time and constant space requirements that computes an optimal correspondence for any pair of executions and any measure. Let $m(w_1, w_2)$ represent the optimal measure of distance for m on w_1 and w_2. Let $m(\mathbf{C}(w_1, w_2))$ represent the optimal measure of distance for m given correspondence **C**. Then we have the following conjecture:

Conjecture 10.6 *Given any two executions w_1 and w_2, there is no linear time and constant space algorithm computing* **C** *such that*

$$\exists \mathbf{C} \forall m . m(w_1, w_2) \geq m(\mathbf{C}(w_1, w_2))$$

is true for all choices of w_1, w_2, and m.

□

To support the conjecture, consider the simple example program given in Figure 42 below. This program loops from 0 to i-1 and executes different sections of code based upon whether or not i-j is even or odd. Notice that there are 4 different possible executions in which the loop iterates either once, twice, three times or four times. Consider the comparison of two executions where i is even for one and odd for the other. Our method of correspondence will make a bad choice of correspondence. Here, the method outlined above would be more suitable. On the other hand, our method will give a better correspondence for executions in which i is either both odd or both even (e.g., $i = 1$ and $i = 3$ or $i = 2$ and $i = 4$). Unfortunately, any correspondence algorithm making an accurate correspondence in both cases will have to store the loop iterations and search for an optimal correspondence.

We justify our choice of an algorithm computing a fixed correspondence by Conjecture 10.6. In addition, we justify the particular choice of correspondence with two facts. First, that this correspondence can be made more efficiently (time and space) than all other possible correspondences. Second, it is a reasonable correspondence for most loops (e.g., it seems reasonable to assume that most loops will perform the same calculation given the same index value).

```
A0: begin
    shared int x = 0;
    shared int lk = 0;
    int n = 3;
A1: task_create(n)
```

```
task 0                           task 1              task 2
    int i = 0;
B0: lock(lk);                    C0: lock(lk);       D0: lock(lk);
B1: x = x + 1;                   C1: x = x + 1;      D1: x = x + 2;
B2: i = x;                       C2: unlock(lk);     D2: unlock(lk);
B3: unlock(lk);
B4: for j = 0 to i do
B5:     if (even(i-j)) then
B6:         do_something;
B6:     else
B6:         do_something_different;
B7: k = k + 1;
```

```
A2: task_terminate(n);
A3: end;
```

Figure 42: A parallel program producing executions for which the correspondence of Figure 10.1 is not optimal

11 Approximate Trace Extrapolation

This section demonstrates that distances presented in Section 9 can be calculated. We present an algorithm that extrapolates a source execution s to some execution that resides in the ${S_{PATT}}^A$ partition for s. We assume that the set S_k accurately approximates the set of executions for the given program and data set for some choice of k, and given this assumption calculate a measure from each of the three categories presented in Section 9. The algorithm takes time linear in the length of the input execution trace. This form of approximate trace extrapolation is asynchronous trace driven simulation.

11.1 Overview

Figure 43 demonstrates the process of extrapolation. The process consists of three independent modules operating on a common data structure, the TBB flow graph. The process of measurement may require look ahead on the source execution. The *source state* module maintains the state of the incoming source execution from the currently consumed portion through the look ahead portion. The *extrapolator* directs the extrapolation to the target execution and computes the appropriate measures. Using the function **advance_source()** the extrapolator can either choose a new block to consume or look ahead on the source execution. The functions **calculate_length()**, **calculate_f_diff()**, and **calculate_suspect()** calculate the representative, feasibility, and likely measures (respectively). The function **output_measures()** outputs the results of the measurements. The function **fix_killed()** adjusts the set of current suspect shared variable definitions. The *target simulation* adjusts the execution for output as the target execution (e.g., chooses instruction interleaving, adjusts duration of memory references, determines lock competition outcomes). The function **choose_context()** determines the next block to be *executed* by the extrapolator. The function **output_target()** actually outputs the target execution.

In order to focus on measurement as opposed to simulation and source state maintenance, we assume that the functions **advance_source**, **choose_context**, and **output_target** exist and discuss only necessary details of their implementation. We also assume that the functions **fix_killed** and **output_measures** exist although we discuss how these functions might be implemented. We focus on the extrapolation process and the measurement functions.

11.2 Measurement

The ability to measure a distance between two executions hinges on the ability to correspond execution (i.e., locate periods of divergence and convergence) and to locate suspect variables. Recall, from Section 9, that a shared variable v is *suspect* for the period of extrapolation beginning with a change in the interleaving of accesses, in the source and target execution, to v and ending with

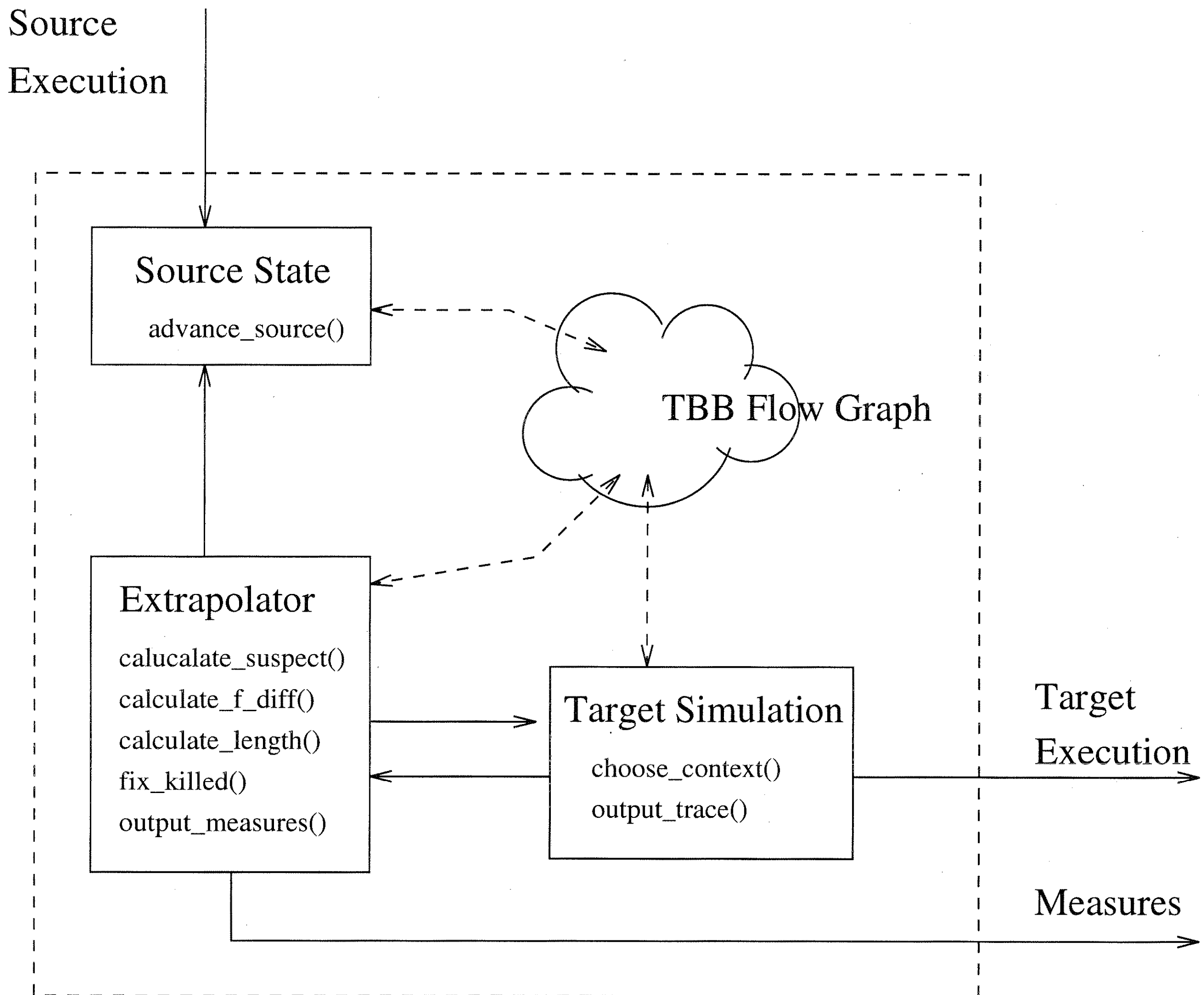


Figure 43: Trace extrapolation coupled with distance calculations

a kill of the current definition of v, that is not as a result of a suspect TCP. A shared variable becomes suspect whenever the interleaving of shared memory accesses, pertaining to the variable, in the source execution differs from the target execution. Suspect shared variables can be removed from suspect status whenever their current definition is effectively killed. This requires a kill of the current definition that is not itself as a result of a suspect TCP. Recall that in Section 9, a TCP is suspect whenever the outcome of the TCP is dependent on a suspect shared variable. Therefore, a kill is suspect if it is on a path between a suspect TCP and its corresponding CP. In Section 8, we provide a means of determining the set of shared variable definitions killed within a trace basic block. Here, we assume that this information is associated with each block of the TBB flow graph of the program and is denoted by *killed*. Locating suspect TCPs requires, for each TCP, the set of shared variables on which the TCP is dependent (either directly or indirectly). This information is known during the calculation of the set of TCPs for a program, see Section 8. Here, we assume that this set is associated with each TCP, and is denoted by *dependent*.

The feasibility measure calculated in this section is the distance, as sequences, between the source and target executions. Extrapolation over ${S_{PATT}}^A$ implies that the two executions share the same set of events and differ only in the interleaving of events. Execution are considered equivalent up to the interleaving of shared memory events. The interleaving of shared memory events determines whether or not a target execution is correct. The feasibility measure is the distance between the target execution And the only execution known to be correct, the source execution. This measure is the difference between shared memory interleavings in the source and target execution. Calculating this measure may require the storage of the order of interleaving of a shared memory reference occurring in the source execution, until the same shared memory reference is encountered in the target execution. In Section 11.4, this process is discussed.

To measure the likelihood of a target execution, the number of suspect TCPs encountered during extrapolation must be determined. For each TCP, a record of the number of instances of the TCP encountered together with the number of those instances that were suspect in the target execution are recorded. In addition, the ratio of the number of suspect TCPs to the total number of TCP instances executed is provided.

Finally, the representative measure for the target execution is, the ratio of the length of the target execution with respect to the number of instructions executed, to an estimate of the potential length of the target execution. Notice that conditional statements in three address code form of Section 3 do not contain **else** statements. That is, the body of a conditional is either executed or it is not and then the corresponding convergence point is executed. The length, in instructions, of the body of a conditional statement can be calculated, given the number of instructions produced by each statement. An estimate of the potential length of the target execution can be made by

adding the length of a path missed due to branch failure for each suspect TCP to the length of the target execution.

11.3 Data Structures and Preprocessing

The extrapolation process requires the ability to traverse the source execution together with the asynchronous creation of a target execution. The data structures for this process can be roughly divided into three categories: those that maintain the state of the source execution, those that maintain the state of the target execution, and those that maintain common state information.

The common data structure for the source and target executions is the TBB flow graph form of Section 10, Section 10.4.1. Given the set of TCP and CPs for the program, the time required to convert the source program into its TBB flow graph form is $O(N)$, where N is the size of the program. In addition, the *killed* and *dependent* information can be collected during the graph construction process with no increase in time (and an increase in space of $O(N \times V)$).

The TBB flow graph of the program can be decorated with additional information to facilitate the computation of the various measures. In particular, a block of the flow graph has a type based on the type of TCP (or runtime statement) that the block contains. Each node in the graph contains the following information:

1. a type denoting the type of node (*type*),
2. two flags denoting whether or not the node contains a CP or runtime instruction (*is_cp* and *is_rt* respectively),
3. a count of the number of instructions in the node (*i_count*),
4. a count of the number of outstanding instances of the node (*out*),
5. an instruction count for the potentially divergent portion of code (*miss_count*),
6. a contribution pointer (*contrib*),
7. and a completion flag (*done*).

The contribution pointer points to the trace contribution for the node. For example, if the source execution consisted of address loads and stores, the trace contribution for a node would be the sequence of loads and stores for the particular node. The instruction count is the count of instructions in the trace contribution for the node. The completion flag denotes whether or not the block has been completed. Initially blocks contain no information about their trace contribution. During extrapolation, when an instance of the block is initially encountered in the source execution, the block in completed by adding the trace contribution information. A block need only

be completed once, except if the block contains runtime instructions. If this is the case, then the runtime instruction contribution must be collected from the source trace for every TBB instance. This information can be stored in a linked list associated with each runtime instruction. Recall that each block contains at most a single TCP. This is the only information in a block that has the potential of changing (other than runtime instructions, see above). Therefore, the trace contribution for a block need only be stored once and the TCP outcome for the block instances can be stored in a linked list associated with the block.

There are five types of nodes: **lock**, **unlock**, **tcp**, **task_create**, and the **none** node. In addition to the above data, each block contains a pointer to its successor in the flow graph. The **tcp**, **none**, and **task_create** nodes are the only nodes with the possibility of more than one successor. The five distinct node types are given in Figure 44. The only additional information in the **none** node is the **runtime_list**. If a node contains a runtime statement, then the list of outcomes for the runtime statement is contained in **runtime_list**, otherwise this list is empty. The **task_create** node contains pointers to each successor task as well as a pointer to the node containing the CP for the **task_create** node. The **unlock** node contains a field to denote the lock being operated on, **lock**. Both the **lock**, **none**, and **tcp** nodes contain additional information used in calculating measures. During extrapolation, the source and target executions behave asynchronously. Therefore, it will become necessary to read enough of the source execution to produce the target execution. This may require that the source execution be read up to a point that the target execution has not yet reached. specifically, information about lock acquisition from the source execution will be necessary and this information must be stored in the nodes of the flow graph. In this sense, the flow graph must represent the dynamic portion of the source execution (not just the static trace contribution information). To accomplish this, the **lock** node keeps a list of the acquisitions for this lock in the order that they occur in the source execution. A count of the acquisition instances currently being maintained in the node is kept, *b_count*. Information about which lock is being operated on and the variable it protects is also maintained, **lock** and **var** respectively. The **tcp** node contains a pointer to the CP for this TCP. To aid in calculating the number of suspect TCPs encountered during extrapolation, **tcp** nodes contain a bit vector that represents the shared variables that the current TCP depends on, the *dependent* vector. If the TCP depends on a shared variable, then the location in the bit vector for this shared variable is set. Similarly, each node contains a bit vector that represents the shared variables that are killed in that node, the *sv_kill* vector. If a node kills a shared variable definition then the location in the bit vector for this shared variable is set.

At any point during extrapolation a portion of the source execution has been consumed by the extrapolator. It is assumed that the trace producer maintains information enough to allow the extrapolator to consume the source execution. It is further assumed that the function **advance_source(block)** consumes the execution up to and including the next instance of **block**

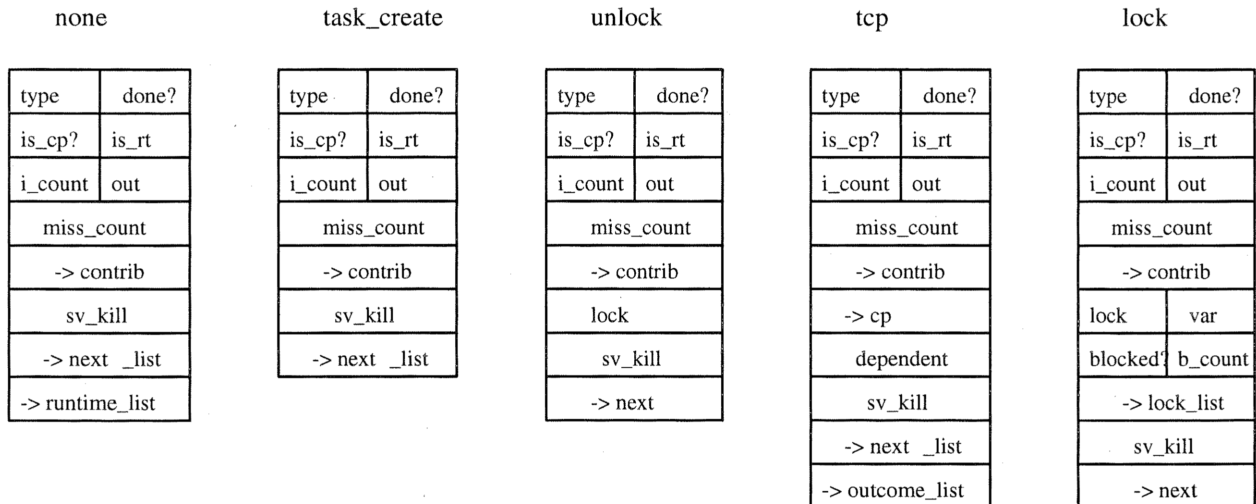


Figure 44: Possible node types for the TBB flow graph

(i.e., the function records the source execution on the TBB flow graph). This function must maintain a series of pointers (one for each thread) to the last blocks read in the source execution, **pc_source**. The function is also responsible for maintaining the current lock acquisition status in the array **source_locks** – an array of pointers to blocks in the TBB flow graph that had a particular lock acquired most recently.

The state of the target execution must also be maintained. This requires an array of pointers to the last blocks encountered in extrapolating the target execution, **pc_target**. A record of the lock acquisition in the target execution is not necessary as any out of order acquisition can be determined, and recorded, as soon as it occurs.

Each of the three measures discussed in the previous section require different information from the source and target executions. First, the feasibility measure, maintained in the variable **fdist**, requires a record of the state of all program locks acquired by the source execution. We will return to this calculation in the next section. Locating suspect TCPs requires keeping track of shared variables whose state may have changed in moving from the source to the target executions. Suspect shared variables and are maintained using a bit vector in the variable **suspect_sv**. Whenever a TCP is encountered, this bit vector is logical anded with the **dependent** vector for the TCP. If the result is zero, then the TCP is not suspect, otherwise it is. Whenever a block is finished, the suspect shared variable bit vector is exclusive ored with the **sv_kill** bit vector for the block and the resulting bit vector replaces the suspect variable vector (this process removes killed shared variables from the suspect shared variable bit vector). The likely measure requires a two dimensional array,

likely(i,j)[30] to record the number of instances of a TCP encountered and the number of suspect TCP instances. This requires that each TCP have a unique index in to the array. The chosen representative measure requires no additional information other than the length, in instructions, of each direction of a TCP. These measures are maintained in the variables **tlength** and **maxlength**.

11.4 Measurement Algorithms

The feasibility distance between the source and target executions is a measure of the number of out of order accesses to shared memory that occur in the target execution (or a measure of the number pairs of acquisitions that are out of order).

To calculate this value, information about the lock acquisition sequence in the source execution must be known when the target execution acquires a lock. For example, consider the sequence of lock acquisitions by four processors in Figure 45 between a source execution and two possible target executions. The difference between the source execution and the first target execution is the out of order acquisition by processor p_3. This results in a distance of one. This out of order acquisition forces a look ahead on the source execution up to the point where an acquisition by p_3 is encountered. The two arrows in Figure 45 represent the source execution state before and after the acquisition of p_3. The difference between the second target execution and the source execution are the first two out of order acquisitions by processors p_3 and p_2. This results in a distance of two. The two arrows on the source execution represent the state of the source execution before and after the acquisition of p_3 by the target execution. The acquisition by processor p_4 is always in order. At the point of choosing p_3 to acquire the lock in the target execution, the block for the lock in p_3 must be finished as its trace contribution is about to be output. This look ahead on the source execution makes it difficult to measure the out of order accesses. Therefore, the order of each acquisition of a reference instance in the source execution must be stored until the target execution processes the reference instance.

A simple list will suffice to maintain a record of the lock acquisitions by the source execution. This list is maintained using the **lock_list** linked list in the TBB flow graph. Each lock node points to the lock node that previously acquired the current lock in the source execution. Due to iterative statements, it is possible that a particular block in the flow graph has more than one instance outstanding between the source and target execution. That is, during the process looking ahead on the source execution, it may be the case that more than a single instance of a particular block has been encountered. Therefore, **lock_list** is a list and the first item on the list corresponds to the least recently encountered lock acquisition on the source execution.

[30] The dimensions of the array are $2 \times |TCP|$.

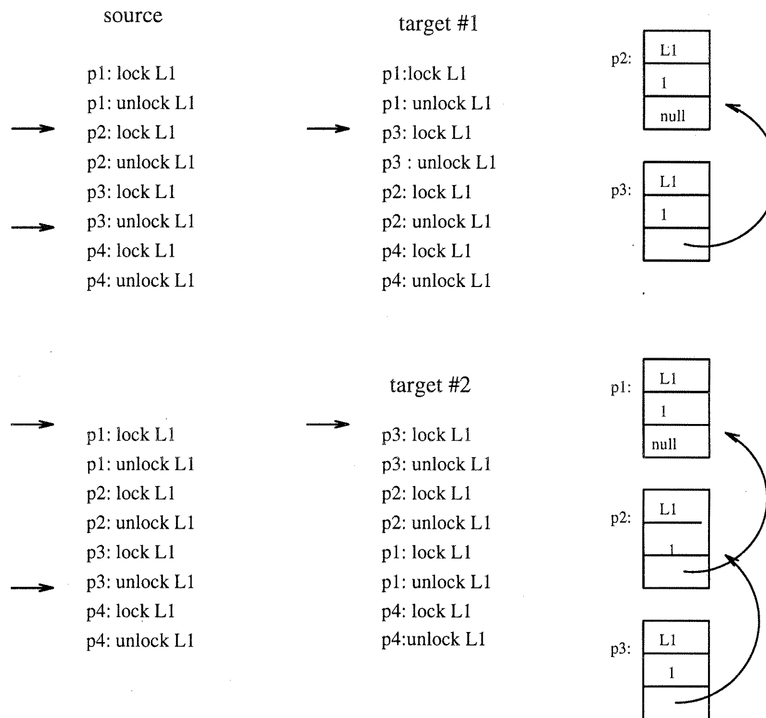


Figure 45: Two possible lock acquisition scenarios during extrapolation

This list together with the instance counter for a block, **out**, will suffice to record any out of order access to locks on the part of the target execution. Function **Calculate_f_dist** in Figure 11.4 calculates this localized distance by marking a block as encountered (line 1), if the predecessor of this block exists and has not been encountered, then the acquisition is out of order (lines 4-6). This fragments the acquisition list but does not hinder the calculation for localized pairs of acquisitions.

To see this, consider the second target execution in Figure 45. When the target execution chooses p_3 for the first lock acquisition, the source execution must look ahead to p_3. In this process, a pointer from p_2 to p_1 is stored and the outstanding count in each block is incremented. When p_1 is encountered in the target execution, if it has any predecessors in the acquisition chain that have not yet been reached in the target execution (e.g., **b_count** is greater than zero), then this acquisition is out of order. For the first target execution, p_1 is not out of order and its count is set to zero. When p_2 is encountered, it is found to be in order. However in the second target execution, the acquisition of p_2 is found to be out of order since p_2 has predecessor p_1 in the source execution and p_1 has yet to be encountered in the target execution (e.g., **b_count** is one).

The second distance measure requires the ability to determine whether or not a TCP is suspect. This is accomplished by maintaining a bit vector of suspect variables. When an out of order access to shared memory occurs, the shared variable in question becomes suspect. When a TCP is encountered, its associated shared variable dependency bit vector logical anded with the suspect shared

```
// Calculate feasibility measure - sequence distance.

// Calculate_f_dist(b):
1  b.count--;
2  if b.lock_list != NULL {
3        if b.lock_list-> count != 0 {
4              suspect(b.var);
5              f_dist++;
6              temp = b.lock_list;
               b.lock_list = b.lock_list-> next;
               dispose(temp);
         }
   }
7  end

// Calculate likely measure - estimate the number of suspect TCPs.
   Calculate_likely_measure(b):
8  likely(INSTANCE, b)++;
9  if b.dependent && suspect_vars
         likely(SUSPECT, b)++;
10 end

// Calculate representative measure - potential length of target execution.
   Calculate_lengths(b):
11 tlength += b.i_count;
12 if b.dependent && suspect_vars
         if b.outcome ≡ b.cp
               maxlength += b.miss_count;
13 maxlength += b.i_count;
14 end
```

Figure 46: An algorithm for calculating distance measures

variable bit vector determine whether or not the TCP is suspect. The algorithm for determining this measure is given in Figure 11.4 (lines 8-9).

Finally, to determine the length of the current execution and the potential length of the target execution, the CP for the current TCP must be stored. If a TCP instance is found to be suspect, then if the divergent path is larger than the path taken by the source execution, then the length of the divergent path is added to the maximum possible length for the target execution.

11.5 Extrapolation

The extrapolator is given in Figure 47. Using the simulation, the extrapolator chooses a context to *execute*, initially this will be the node containing the program **begin** statement. The program iterates until there are no more nodes to be considered. That is, **choose_context** returns **NULL** (lines 1, 2, and 11). If there are no outstanding instances of the current node, b, on the TBB flow graph, the source execution is advanced up to and including node b (line 3). The trace contribution for the node is output (line 4). Notice that this marks this node as ready for output by the simulation and the simulation is free to interleave this node with others as necessary. If the node is a **lock** node, then the feasibility distance is calculated (line 6) and if the node is a **tcp** node, then the likely distance is calculated (line 7). Otherwise, if the node is a **none**, **task_create**, or an **unlock**, no additional distance calculation is made. The lengths for the nodes are calculated (line 9). The set of suspect shared variables is adjusted based on the shared variable definitions killed in this node (line 10). Finally, the extrapolator requests the next context from the simulation (i.e., the next node to be executed) (line 11). If the simulation returns **NULL**, then the extrapolation is complete and measure can be output (line 12).

11.6 Complexity, Correctness, and Measurement Results

The extrapolation process takes time linear in the length of the source execution since each instruction instance is visited at most once by each module. The exception is the process of looking back to see which process had the last lock acquisition. This requires a look back at a single instruction instance. The time to extrapolate a source execution is $O(K)$, where K is the length of the source execution.

The correctness of the extrapolation process hinges on the correctness of the simulation and source state maintenance modules. Since we have not considered the implementation of these modules in this section, we can only argue the correctness of our measures. The measures rely on the comparison algorithm of the previous section to locate periods of divergence and convergence in the potential target and source executions. First, consider the representative measure. It suffices to show that the number of instructions in the trace contribution for each traversed node instance are

// Extrapolate a source execution given a simulation with functions **choose_context()** and **output_target()**, and given the function **advance_source()** for maintaining the state of the source execution.

```
   extrapolate():

1  TBB_node *b = choose_context();
2  while b do
3      if (b.out ≡ 0) advance_source(b);
4          output_trace(b);
5          case b.type of
6              lock:       calculate_f_dist(b);
7              tcp:        calculate_suspect(b);
8              otherwise:  skip;
       }
9      calculate_length(b);
10     fix_killed(b);
11     b = choose_context();
   }
12 output_measures();
13 end.
```

Figure 47: An extrapolation algorithm

added to **tlength** and that during periods of potential divergence, the longest possible path from TCP to CP is added to **maxlength.** During a period of convergence, the instruction count for the current node is sufficient to calculate the length of the trace contribution for that node. Line 9 of the extrapolation algorithm guarantees that the length contribution for each node is considered. Line 11 of the **calculate_length** guarantees that each node's instruction contribution is added to **length.** Line 13 adds the current node length to the total maximum length. The correctness of the maximum potential length computation hinges on the correctness of the length **miss_count.** There are two cases for which the maximum potential length calculation will be inaccurate. First, if the TCP currently under consideration is an iteration statement, then at most, one additional iteration of the loop will be added to the maximum potential length. This is because the number of times an iteration statement iterates cannot be predicted. Second, if an iterative statement occurs during a period of divergence, then it will be impossible to accurately determine the number of times that the loop might iterate. If the program under consideration contains no iteration during the divergent period, then the measure is accurate. In either case, line 12 ensures that if the TCP in question is suspect, that is, if the divergent branch may have been taken, and if the actual branch taken represents a failure of the conditional, then the length of the divergent branch is added to the total maximum potential length.

The likely measure depends on our ability to detect suspect variables. If the list of lock acquisitions from the source state is accurate, then **calculate_f_dist** correctly checks if the predecessor (in the source execution) of the current lock acquisition (in the target execution) has already been done. If this is the case, then the protected shared variable is suspect.

Finally, the **output_measures** function outputs the ratio of the length of the target execution to the maximum potential length of the target execution. In addition, the feasibility calculation **f_dist** is output. Finally, the *likely* array allows for the production of statistics about the number of instances of a TCP and the number of times a particular TCP was potentially different together with the ratio of suspect TCP instances to the total number of TCP instances executed in the target execution.

12 Conclusion

While the work of this thesis has focused on the problem of trace extrapolation as it relates to trace-driven simulation of multiprocessor computer systems, the problem has implications for several other related fields. These areas include trace collection and trace reconstruction, perturbation analysis, race detection, and the storage of traces. This section, presents a brief conclusion of the work of this thesis and discusses the implications of the results for trace-driven simulation and these related fields.

12.1 The Classification of Instructions

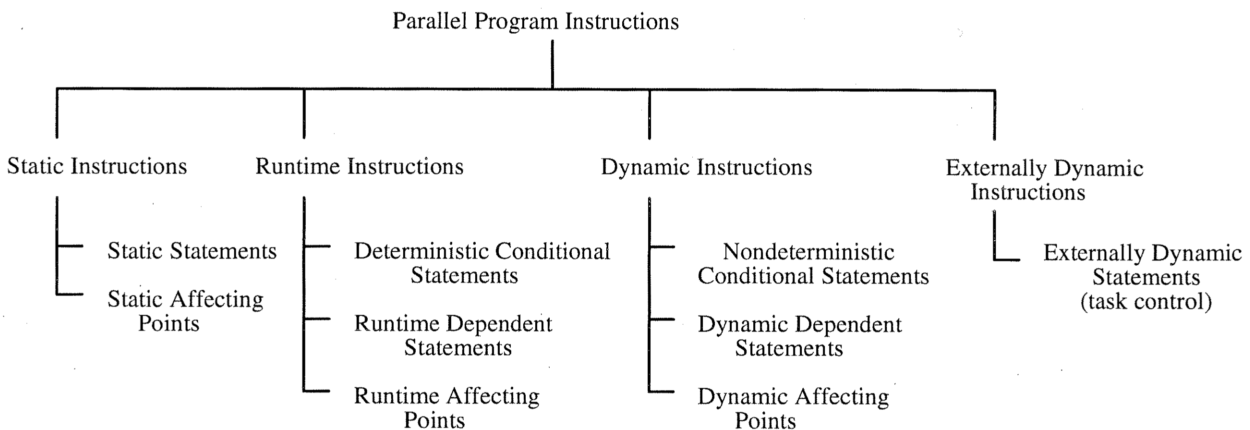


Figure 48: The parallel program instruction and statement taxonomy

The taxonomy of parallel program instructions presented in Section 4 extends the work of Holliday and Ellis [32]. Holliday and Ellis identify both address change points (our trace change points) and address affecting points in the source code of a parallel program using a language of medium grained process threads that exclude procedures and functions. Our languages is based on lightweight process threads including locks, iteration, and procedures and functions. We make the observation that instructions can be classified as to their execution contribution. Some instructions have a fixed execution contribution. These are the *static* instructions. Some instructions have a sequence of possible execution contributions that is fixed over subsequent runs of the program with the same data set. These are the *runtime* instructions. Some instructions have a sequence of possible execution contributions that varies over subsequent runs of the same program. These are the *dynamic* instructions. Finally, while task control statements have a fixed execution contribution (i.e., the set of events generated by a task control statement), this contribution, taken as a sequence, can vary (i.e., tasks can be scheduled nondeterministically). Nondeterministic scheduling policies

for processor to task assignment can (externally) introduce nondeterminism into a parallel program. Using these observations, instructions are classified as to their ability to affect the control flow of a parallel program and whether or not this affect is due to program nondeterminism. Figure 48 summarizes the instruction taxonomy.

Because of the potentially large size of parallel program traces, trace storage is an issue. A goal of trace storage is to, by some means, compress a trace and accurately recover the trace for later use. The taxonomy identifies exactly those portions of a trace that need not be stored for accurate regeneration, specifically, the static instructions. A parallel program trace can be stored on a basic block flow graph of the parallel program. Each basic block contains the execution contribution of its static instructions together with a list of the outcome of each runtime or dynamic instruction. This eliminates the need to store redundant copies of static instruction contributions. Further reduction in the size of a trace may be possible (e.g., the outcome of conditional statements can be stored using a bit vector). The taxonomy and Theorem 7.1 demonstrate exactly what information must be retained in a trace for recovery and at what point reexecution of a program is required to reconstruct a trace.

12.2 Trace Change Points

Two open problems exist pertaining to trace change points. The problems of statically locating the set of trace change points for a parallel program and of deciding the outcome of the set of trace change points for a program given some execution path prefix. The latter is of course the trace extrapolation problem. This thesis answers both of these problems.

This thesis demonstrates that the problem of locating TCPs statically in the source code of a parallel program is an NP-hard problem, see Corollary 4.4. There are several implications of this result. Any algorithm computing the set of TCPs for a program must produce an approximation. The problem of determining whether or not a program is execution pattern deterministic is an NP-hard problem, see Theorem 4.3. The size of a trace extrapolation problem is directly related to the number of TCPs in a parallel program, therefore the problem of determining the size of a trace extrapolation problem instance is an NP-hard problem, see Corollary 6.9.

We introduce the shared variable dependence relation in Section 8. This relation can be used to locate the set of TCPs in a parallel program. The problem of computing the shared variable dependence relation can be viewed as a data flow analysis problem. An algorithm for computing the shared variable dependence relation for a parallel program is provided that computes a conservative approximation to the set of TCPs in a parallel program in our assumed language. Excluding the possibility of functions and procedures, the worst case run time for this algorithm is $O(N \times E \times V^2)$ where N is the number of statements of the program, E is the number of edges in the TCP control flow graph of the program, and V is the number of variables in the program. This complexity can

be reduced to $O(N \times V)$ using an approximation to the data flow version of the algorithm. For this approximation, we introduce the shared variable dependency graph of a program statement. The SVD graph of an iterative statement eliminates the need to iterate over the blocks in the TCP control flow graph associated with the statement to produce the set of program TCPs. The cost of this approximation is a less accurate calculation of the set of TCPs of a program. The advantage is that it allows the set of TCPs contained in function and procedure bodies to be reasonably computed. The worst case run time for the data flow analysis algorithm when functions and procedures are present is $O(N \times E \times V^2 \times P)$ where P is the total number of function and procedure invocations in the source program.

12.3 Determinism, Nondeterminism, and Traces

Traditional characterizations of program nondeterminism are inadequate for characterizing the effects of program level nondeterminism on the set of executions produced by a program. This is because traditional characterizations are concern the with the effects of nondeterminism on the values computed by a program as opposed to the trace produced by the program. Section 4 introduces the characterizations of execution pattern deterministic and execution pattern nondeterministic for parallel programs. This characterization is used in our model of parallel program executions.

12.4 A Formal Model of Parallel Program Executions

Section 6 present a formal model of parallel program executions. The model provides a means for proving the trace extrapolation problem intractable. The model is flexible in that it is possible to describe different levels of permissible equivalence between executions. Approximations can be characterized within the model. The model provides an indicator of the size of a trace extrapolation problem instance.

The model is based on Lamport's temporal model of program executions [36]. This model is augmented with Netzer's [43] shared data dependence relation and axioms, and our own TCP equivalence relation. These relations provide a characterization of data and pattern determinism based on execution equivalences.

Under what conditions two executions should be considered equivalent depends on the intended application of those executions. The model uses equivalence relations on parallel program executions to provides a means of adjusting the size of sets of executions under consideration, hence the model is capable of adjusting the notion of equivalent for traces. One advantage of this characterization is that a set of executions small enough for the trace extrapolation problem to be tractable yet large enough to include TCPs can be defined.

A parallel program is classified as data and pattern deterministic, data nondeterministic and pattern deterministic, or pattern nondeterministic based on the set of executions that the program, given a fixed data set, produces. This classification allows for the identification of programs for which trace extrapolation is a trivial process. These are the data and pattern deterministic and the data nondeterministic and pattern deterministic programs. Data and pattern deterministic programs include many common programming languages (e.g., those using the *DOALL*, *DOACROSS*, *post/wait* with no clear, programs containing externally dynamic statements but no dynamic statements, and abstractable synchronization such as wait-time at barriers and locks). Included are also programs that are data nondeterministic and pattern deterministic although there are no programming constructs that fall into this category (e.g., programs that compute nondeterministically but that do not make decisions based on nondeterministic values).

The $\rightarrow_{S_k}$ family of execution equivalences provides a means of distinguishing the level of nondeterminism present in a parallel program. Section 5 indicates that there is a structure to the set of TCPs and CPs in a parallel program and that, among other things, a subset of those TCPs will be executed in any execution of the parallel program, given a fixed data set. That is, that two executions from any parallel program and fixed data set are guaranteed to have periods of divergence and convergence. The $\rightarrow_{S_k}$ family provides an indication of the amount of nondeterminism that can occur during a divergent period. This provides an indication of the size of the trace extrapolation problem for a given parallel program and data set. In addition, the family provides a means of characterizing approximations to pattern nondeterministic sets of executions. That is, divergent behavior in execution can be bounded (and hence approximated) by the number of TCPs that can occur.

In [6], Bitar implies that there are only two possible approximations for trace extrapolation. In their paper [26], Goldschmidt and Hennessy argue that there are many possible choices of approximations for trace extrapolation. Here, we have presented a family of possible approximations. This family is by no means a limit on the possible approximations that exist.

12.5 Trace Extrapolation is Intractable

One of the difficulties with extrapolating traces taken from parallel programs that are not data nondeterministic is that the problem of determining whether or not an extrapolated trace represents a valid program execution is undecidable. For data and pattern deterministic programs, the validity of a target trace can be inferred from the source trace, see Theorem 7.2. This is not the case for any other class of program, see Theorem 7.3.

Trace extrapolation as stated by Holliday and Ellis [32] is an unreasonable problem requiring information about sets of exponential size. In Section 7, we demonstrate that even if the set

of executions of a parallel program is pruned to a reasonable size, the problem is NP-hard, see Theorem 7.1.

This result has the following implications. First, trace-driven simulation is not valid for parallel program exhibiting nondeterminism unless valid trace extrapolation approximation techniques exist. The same is true for trace collection and perturbation analysis. If approximation techniques do not provide the necessary level of accuracy, then direct simulation techniques must be used for experiments involving parallel programs exhibiting nondeterminism. If approximate trace extrapolation can be shown to be valid, then the reuse of traces (or the use of *canned* traces) from parallel programs may in many cases be more desirable than direct simulation techniques because approximate trace extrapolation can be done rapidly (i.e., in linear time as in the algorithm of Section 11).

Race detection is a hard problem for programs intended to be pattern deterministic and programs intended to be pattern nondeterministic further complicate the problem. For race detection of parallel programs intended to be pattern nondeterministic using postmortem trace analysis, the result suggests that each collected trace can be used only for race detection on a specific execution pattern. If correcting an erroneous race changes the path of a program execution, then the execution cannot be considered valid for further race detection and a new trace must be generated. Corollary 4.4 complicates matters by implying that the problem of determining the set of possible patterns (or the size of the debugging problem) is itself intractable.

12.6 Measurement and Accuracy in Approximations

Ideally, if trace extrapolation, trace collection, and perturbation analysis are not guaranteed to produce a correct trace for traces taken from parallel programs exhibiting nondeterminism, they should at least produce an accurate trace. Measures of trace accuracy could provide a means of correlating the accuracy of a trace with the accuracy of the simulation results from the use of the trace. In Section 9 we demonstrate that there are fundamental limitations to our ability to determine the accuracy of a trace that results from trace extrapolation. While the effects of state change on the target trace can be determined, the effects of time shifts in an execution due to a state change cannot be determined. We present three possible measures of trace accuracy and discuss the limitations of each.

12.7 Corresponding Parallel Program Executions

An additional difficulty with accuracy measurements is the actual process of collecting those measurements. Traces can be quite large requiring linear time and space algorithms for computing measures. Fortunately executions are sequences with structure. The work of Section 5 is used in

Section 10 to prove that two executions can be corresponded a priori in linear time and space. In Section 11, this correspondence algorithm is used to create an approximate trace extrapolation algorithm that computes the three measures proposed in Section 9 in linear time and space.

The ability to correspond parallel program executions is useful for other problems pertaining to parallel program executions. For example, the collection of traces often leads to incomplete traces due to large amount of trace data and buffer limitations. The problem of repairing an incomplete trace is that of *trace reconstruction.* The ability to correspond parallel program executions indicates that the problem of locating a hole in a trace of a parallel program is solvable for programs exhibiting a certain characteristic. Recall that hardware monitors collect only off-chip memory reference events. The aforementioned characteristic is that for programs producing a set of executions whose size is equivalent to the size of the set of trace-level views, of the same executions, containing only memory references (i.e., the set of TCP and CP instances in a trace can be determined from the trace). Further, given such a program, the problem of accurately determining the extent of a hole on the program flow graph is tractable. That is, the starting instruction and ending instruction for a hole can be located. However, Theorem 7.1 indicates that the problem of actually reconstructing the missing portion of the trace is intractable.

12.8 Interprocess Communication

In this thesis, we have assumed the use of massive shared memory multiprocessors. Here, interprocess communication is achieved using shared memory. Other architectures exist using different methods of interprocess communication. Most notable are distributed architectures providing interprocess communication via message passing which we will briefly discuss.

12.8.1 Message Passing

Nondeterminism in shared memory systems arises due to competitive access to shared memory locations. For message passing systems, nondeterminism arises when multiple processes send a message to the same destination. Here, time dictates the values received at the destination. In Figure 49, two processes, P1 and P3, compete for communication with process P2. Both processes send an integer value to P2 and P2 executes a conditional statement whose outcome depends on the received value. This example illustrates that **send** operations are dynamic affecting points and that conditional and array reference statements that depend, either directly or indirectly, on a value set via a **receive** command are TCPs. Given this correspondence, the model and work presented in this thesis can be directly applied to message passing architectures and languages. For example, languages using unrestricted message passing are pattern nondeterministic.

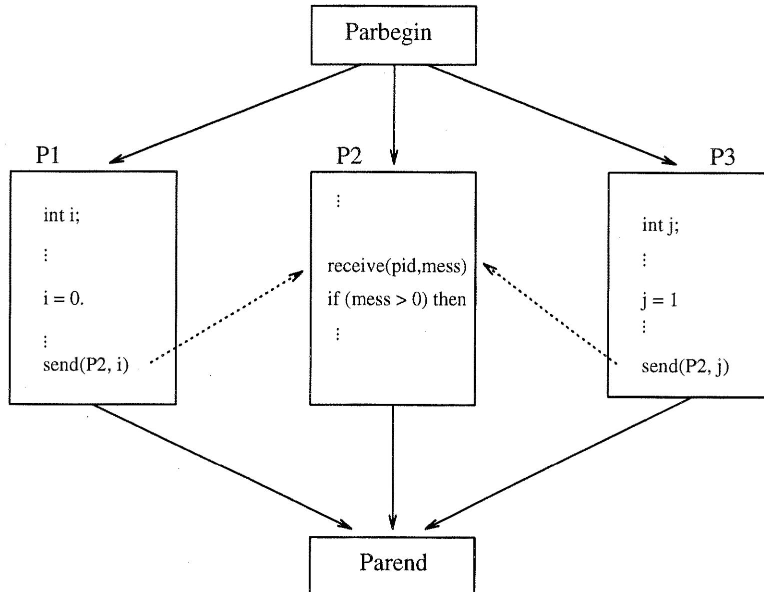


Figure 49: An example parallel program using the message passing operations **send** and **receive**

12.9 Other Memory Consistency Models

The work of this thesis assumes strong memory consistency. Other models of memory consistency for multiprocessor systems [10] exist (e.g., weak consistency, copy-in-copy-out models, etc.). Our work holds, trivially, for consistency model that eliminate nondeterministic interaction between processors (e.g., copy-in-copy-out models). The effects of weak memory consistency on the results of this thesis, are considered in this section.

12.9.1 Weak Memory Consistency

Weak consistency models remove the assumption that requests issued from processors will be serviced in the order in which they are issued.

The basic model used in this thesis for executions does not hold for a weak model of consistency. The model specifically assumes sequential consistency in that it is assumed that a single temporal relation is sufficient to describe the temporal behavior of an execution. The model can be adapted to consider weak models of memory consistency by replacing the current temporal order relation $\rightarrow_T$ with two temporal order relations, one for the time an event is issued, and a second for the completion time for the event.

A major difficulty with adapting the model to weak memory consistency is the shared data dependence relation. Notice that two executions with the same shared data dependence relation are no longer guaranteed to execute the same set of events. That is, under an assumption of weak memory consistency, the shared data dependence relation cannot be used to characterize data

determinism. However, the TCP equivalence relation still characterizes pattern determinism. This implies that our results for pattern deterministic and pattern nondeterministic programs hold.

12.10 Future Research Directions

12.10.1 Trace-Driven Simulation

The validity of using incorrect traces in trace-driven simulations of shared memory multiprocessor computers is still an open question. This thesis has shed light on the nature of inaccuracies due to approximate trace extrapolation. With this identification, experiments can be designed that test the validity of simulation results for traces from commonly used programs (e.g, the SPLASH benchmark suite of programs [50]) applied to various simulations).

12.10.2 Time

Time is a critical factor in trace accuracy. Adding time to our model will allow discussions on the sensitivity of a program and the effects of time dilation. Time stamped traces will be representable in our model. In addition, the use of time warping techniques will provide a measure of the difference between timestamped traces.

12.10.3 Trace Accuracy

One of the overall goals of this work is to provide a basis for research in to characterizations of program behavior with respect to the traces a program produces. Two levels of characterization would be useful. One characterization is a correlation between the structure of a parallel program and the accuracy of a collected trace from that program. If such a correlation exists, it would aid a researcher in making the decision to use a specific program in an experiment, to use a particular trace collection algorithm, etc. A second characterization is a correlation between the accuracy of a collected or extrapolated trace and the results of simulations using the trace. This characterization would aid a researcher in determining whether or not direct simulation is required for a particular experiment. Identification of the inaccuracies that can be induced in a trace due to trace extrapolation is a first step along this path.

12.10.4 Trace Reconstruction

As mentioned in Section 12.1, our work with trace correspondence sheds light on the trace reconstruction problem. To our knowledge, no accurate trace reconstruction algorithm exists.

12.10.5 Program Level Nondeterminism

There are many intractable problems for which the search space or size of the problem is easily determined. Knowing the size of an intractable problem can be useful. For example, a particular intractable graph algorithm may in practice work well for small problem instances. Unfortunately, there is no characterization of the size of a trace extrapolation or problem.

Program level nondeterminism is necessary for execution pattern nondeterminism but not sufficient. Shared variables affected by nondeterminism must be used in conditional of reference statements. The degree to which a single nondeterministic point in a program determines the amount of divergence that can occur in executions depends entirely on the size of the state and the number of TCPs affected by the nondeterminism.

The $\rightarrow_{S_k}$ family of execution equivalences and the TCP distance for parallel programs is an attempt to create such a characterization and should be experimentally tested for accuracy. The characterization distinguishes programs that iterate based on nondeterministic values from those iterating on deterministic program values. Clearly, these two types of programs represent a different level of difficulty for trace extrapolation and should be distinguished. It remains to be demonstrated that the TCP distance provides an adequate approximation of problem size.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprocessing workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[2] A. Agarwal, R. L. Sites, and M. Horowitz. Atum: A new technique for capturing address traces using microcode. In *Proc. 13th Int'l Symp. Computer Architecture*, pages 119–127, 1986.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[4] T. R. Allen and D. A. Padua. Debugging fortran on a shared memory machine. In *1987 Intl. Conf. on Parallel Processing*, pages 721–727, St. Charles, IL, Aug 1987.

[5] V. Balasundraram and K. Kennedy. Compile time detection of race conditions in a parallel program. In *Proceedings Third International Conference on Supercomputing*, pages 175–185, Crete, Greece, June 1989.

[6] P. Bitar. A critique of trace-driven simulation for shared-memory multiprocessors. In *Cache and Interconnection Architectures*, pages 27–52. 1990.

[7] A. Borg, R. E. Kassler, and D. W. Wall. Generation and analysis of very long address traces. In *Proc. 17th Int'l Symp. Computer Architecture*, pages 270–279, Los Alamitos, CA, May 1990. IEEE CS Press.

[8] E. A. Brewer, C. N. Dellarocas, A Colbrook, and W. E. Weihl. Proteus: A high performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, Dept. of Computer Science, September 1991.

[9] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, Madison WI, May 1988. ACM Press.

[10] K. Charachorloo, D. Lenoski, J. Laudon, A. Gupta, and K. Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proc. 16th Annual Symp. on Computer Architecture*, Seattle WA, May 1990. ACM Press.

[11] E. G. Coffman and P. J. Denning. *Operating Systems Theory.* Prentice-Hall, 1973.

[12] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The rice parallel programming testbed. In *Proc. 1988 ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pages 4–11, Santa Fe, NM, May 1988.

[13] H. Davis, S. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using tango. In *Proc Int. Conf. on Parallel Processing.* ACM Press, May 1991.

[14] A. Dinning. *Detecting Nondeterminism in Shared Memory Parallel Programs.* PhD thesis, New York University, 1990.

[15] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 70–79, Santa Cruz, CA, January 1991. ACM Press.

[16] Z. K. F. Eckert and G. J. Nutt. Parallel program trace extrapolation. In *Proceedings of the 1994 International Conference on Parallel Programming*, pages II–103 to II–107, St. Charles, Illinois, August 15-19, 1994.

[17] Z. K. F. Eckert and G. J. Nutt. A framework for execution order distortion in shared memory multiprocessor event traces. Technical Report TR CU-CS-708-94, Department of Computer Science, University of Colorado, March 1994.

[18] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluations. In *Proc. 15th Annual Int'l Symp. on Computer Architecture*, pages 373–383, Honolulu, HI, June 1988.

[19] S. J. Eggers, D. Keppel, E. Koldinger, and H. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, May 1990.

[20] A. Ehrenfeucht and D. Haussler. A new distance metric on strings computable in linear time. *Discrete Applied Mathematics*, 20:191–203, 1988.

[21] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 89–99. ACM Press, January 1991.

[22] P. G. Farber. Analysis of a shared bus multiprocessor memory system using trace driven simulation. Master's thesis, University of Colorado, 1991.

[23] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[24] R. M. Fujimoto and W. C. HareNovember 1992. On the accuracy of multiprocessor tracing techniques. Technical Report GIT-CC-92-53, Georgia Institute of Technology, Deptartment of Computer Science, November 1992.

[25] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., 1979.

[26] S. Goldschmidt and J. Hennessy. The accuracy of trace-driven simulation of multiprocessors. In *Proc. ACM SIGMetrics Int'l Conf. Measurement and Modeling of Computer Systems*, pages 146–157. ACM Press, 1993.

[27] D. Grunwald, G. J. Nutt, A. Sloane, D. Wagner, and B. Zorn. A testbed for studying parallel programs and parallel execution architectures. In *Proceedings of MASCOTS'93*, pages 95–106, January 1993.

[28] P. A. V. Hall and R. Dowling. Approximate string matching. *ACM Computing Surveys*, 12(4):381–402, 1980.

[29] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, Inc., New York, NY, 1977.

[30] D. P. Helmbold and C. E. McDowell. Computing reachable states of parallel programs. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 1991.

[31] D. P. Helmbold and C. E. McDowell. Race detection - ten years later. Technical Report No-Number-Yet!, Board of Studies in Computer and Information Sciences, University of California, Santa Cruz, October 1994.

[32] M. A. Holliday and C. S. Ellis. Accuracy of memory reference traces of parallel computations in trace-driven simulation. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):97–109, January 1992.

[33] J. Jain. *The Art of Computer Systems Performance Evaluation.* John Wiley and Son, Inc., New York, 1991.

[34] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the First ACM Symposium on Principles of Programming Languages*, pages 194–206, October 1993.

[35] E. J. Koldinger, S. J. Eggers, and H. M. Levy. On the validity of trace-driven simulation for multiprocessors. In *Proceedings of the 18th Symposium on Computer Architecture*, pages 244–253, Toronto, Canada, 1991. ACM Press.

[36] L. Lamport. A new approach to proving correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.

[37] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, c-28(9):690–691, SEPT 1979.

[38] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software – Practice and Experience*, 20(12):1241–1258, December 1990.

[39] A. D. Malony. Event-based performance perturbation: A case study. In *Ppopp.* ACM Press, 1991.

[40] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating Systems Services.* PhD thesis, Columbia University, 1992.

[41] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[42] B. P. Miller and J. Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings Sigplan Conference on Programming Design and Implementation*, pages 135–144, Atlanta, GA, June 1988.

[43] R. H. B. Netzer. Race condition detection for debugging shared-memory parallel programs. In *Doctoral Thesis.* University of Wisconsin-Madison, 1991.

[44] G. J. Nutt. A parallel program tuning environment. In *Proceedings of the 1993 International Conference on Parallel Programming*, pages II–77 to II–81, St. Charles, Illinois, August 16-20, 1993.

[45] D. A. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications ACM*, 29(12):1184–1201, December 1986.

[46] D. A. Reed and et. al. Scalable performance environments for parallel systems. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 562–569, 1991.

[47] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The wisconsin wind tunnel: Virtual prototyping of parallel computers.

[48] D. Sankoff and J. B. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley Publishing Company Inc., Reading, MA, 1983.

[49] S. Sherman and J. C. Browne. Trace driven modeling: Review and overview. In *Proceedings of Symposium on Simulation of Computer Systems*, pages 201–207. ACM Press, 1973.

[50] J. P. Singh, W. D. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. *ACM SIGARCH Computer News*, 22(1):5–44, 1992.

[51] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, SEPT 1982.

[52] C. B. Stunkel and W. K. Fuchs. Trapeds: Producing traces for multicomputers via execution driven simulation. In *Proc. ACM SIGMetrics Int'l Conf. Measurement and Modeling of Computer Systems*, pages 70–78. ACM Press, 1989.

[53] C. B. Stunkel, W. K. Fuchs, and B. Janssens. Address tracing for parallel machines. *IEEE Computer*, 24(1):31–38, January 1991.

[54] R. N. Taylor. *Static Analysis of the Synchronization Structure of Concurrent Programs.* PhD thesis, University of Boulder at Colorado, 1980.

[55] R. N. Taylor and L. J. Osterweil. Anomaly detection in concurrent6t software by static data flow analysis. *IEEE Transactions on Software Engineering*, 6(3):265–277, May 1980.

[56] J. E. Veenstra and R. J. Fowler. Mint tutorial and user manual. Technical Report TR 452, Computer Science Department, The University of Rochester, Rochester, New York, June 1993.

[57] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(3):319–349, July 1991.

Index

S_k, 109
S_k^A, 112
S_{DATA}, 106
S_{PATT}, 106
S_{PATT}^A, 110
$\rightarrow_T$, 50, 51
$\rightarrow_{S_K}$ family, 108
$\rightarrow_{S_k}$, 107
$\rightarrow_{tp}$, 93
$\rightarrow_{sd}$, 91
3CNFSAT, 65

abstract higher-level view, 99
access order distortion, 20
actual execution, 51
analytical modeling, 26
asynchronous trace-driven simulation, 28

Boolean Satisfiability, 65, 115

candidate TCP statements, 131
causal traces, 2, 17
code synthesis, 32
constant propagation, 126, 139
control flow graph, 138
convergence, 79
convergence point, 5, 71, 80
correct, 2, 19

data and pattern deterministic execution equivalence, 95
data and pattern deterministic program, 96
data determinism, 63
data flow analysis, 139
data nondeterminism, 63
data nondeterministic, 5
data nondeterministic and pattern deterministic execution
equivalence, 96
data nondeterministic and pattern deterministic program, 97
definition dependency, 129
deterministic conditional statement, 60
dependency
direct, 128, 132
indirect,
direct shared data dependence, 91
direct simulation, 30
directly shared variable dependent, 128
distortion, 20
execution driven simulation, 31
divergence, 79
divergence points, 5, 71, 80
dynamic affecting point, 61
dynamic dependent statement, 61
dynamic instruction, 60

emulation, 23
emulation/simulation, 26
event, 16, 47, 50
abstraction, 98
hardware, 16
operating system, 16
predecessor, 76
program, 47
successor, 76
trace, 16
user-level, 16
execution, 1, 17
path, 20
pattern, 21
graph of, 77
execution architecture, 1, 16
execution architecture nondeterminism, 20
execution contribution, 58
execution driven simulation, 31
execution order distortion, 20
execution pattern determinism, 57
execution pattern deterministic, 6
execution pattern distortion, 2
execution pattern nondeterminism, 57
execution pattern nondeterministic, 6
externally dynamic, 61

feasibility distance, 169
feasible, 7, 19

generated, 129, 132
graph (of an execution), 77

hardware event, 16
hardware monitoring, 23
hardware prototyping, 26
higher-level view, 54

indirect dependency, 129, 132
instruction, 47
instruction-level view, 55
instrumentation, 24
interrupt-based collection, 24
intractable, 16, 29, 37, 38
intrusive monitoring, 23

killed, 129, 132

lattice, 143
likely distance, 171

memory consistency,
  copy-in-copy-out, 226
  sequential, 41, 226
  weak, 226
microcode alteration, 24

native execution, 31
nondeterministic conditional statement, 61
NP, 65
NP-complete, 65
NP-hard, 65

operating system event, 16
outcome, 78, 92
outcome equivalence, 78

partial algorithm, 40
path, 19, 75
  deterministic, 63
  disjoint, 76
  equality, 19
  equivalence, 76
  head, 75
  nondeterministic, 63
  prefix, 75
  suffix, 75
  tail, 75
pattern, 19
pattern determinism, 57
pattern nondeterminism, 57
pattern nondeterministic execution equivalence, 97
pattern nondeterministic program, 97
perturbation analysis, 21
preserved, 129
process, 51
program, 47
program event, 47
program execution, 50
program level nondeterminism, 20

reexecution, 4, 34
representative, 22
representative distance, 170
representative trace, 169
runtime affecting point, 60
runtime dependent statement, 60
runtime instruction, 59

SAT, 65, 115
sequential consistency, 41
shared data dependence, 91
shared data dependence relation, 91
shared data equivalence, 92
shared variable dependence relation, 128
shared variable dependency graph, 128, 147
shared variable dependent, 128
shared variable dependent variable set , 126
simple constant, 127, 139, 145
simulation, 26
source trace, 4, 34
static affecting point, 59
static instruction, 58
static statement, 59
suspect
  execution, 164
  variable, 164
SVD graph, 128, 147
SVD relation, 128
SVD variable set, 126, 143
SVD variables, 129
synchronous trace-driven simulation, 28

target trace, 4, 34
task control statement, 61
TBB, 181
TBB flow graph, 181
TCP basic block, 181
TCP candidate statement, 138
TCP higher level view, 108
TCP precedence relation, 93
TCP predecessor, 81
TCP successor, 81
test-and-set, 42
thread, 51
Three Conjunctive Normal Form Boolean Satisfiability, 65
time dilation, 21
timestamped, 2
timestamped traces, 17
topological sort, 150
trace, 16, 23, 56
trace affecting point, 61
trace change point, 5, 37, 62, 80
trace collection, 23
trace extrapolation, 4, 35
trace reconstruction, 223
trace reuse, 4, 34
trace view, 56
trace-driven simulation, 27
traceable instructions, 56

ud-chain, 62
unintrusive monitoring, 23
use-definition chain, 62
user-level event, 16

view
    abstract higher-level view, 99
    higher-level view, 54
    instruction-level view, 55
    TCP higher level view, 108
    trace view, 56

wait-time distortion, 20
witness, 101
witness execution, 122