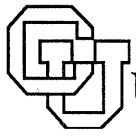
A FRAMEWORK FOR EXECUTION ORDER DISTORTION IN SHARED MEMORY MULTIPROCESSOR EVENT TRACES

Zulah K.F. Eckert & Gary J. Nutt

CU-CS-708-94



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A Framework for Execution Order Distortion in Shared Memory Multiprocessor Event Traces

Zulah K. F. Eckert* and Gary J. Nutt
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO, 80309-0430
email:{eckert,nutt}@cs.colorado.edu

Abstract

Event traces have been fundamental to performance evaluation methodology since the 1960s. Collecting event traces from parallel programs is complicated by program and execution architecture nondeterminism. The process of collecting a trace can perturb the execution of a parallel program causing distortions in the resulting trace. While delays and changes in instruction interleaving can be accounted for during simulation, execution order distortions (changes in program execution path) cannot.

Currently, we know of no characterization of the degree to which a program suffers from execution order distortion. This question is of fundamental importance for other questions about executions (e.g., trace migration, trace comparison, etc.). Our research is an ongoing effort to quantify execution order distortion in program executions. This paper presents a formal framework that we will use to realize this goal. We extend the work of Holliday and Ellis [17] characterizing the causes of execution order distortion by proposing a taxonomy of program instructions based upon their contribution to execution order distortion. We claim that the points induce structure in executions and that this structure can be exploited. The framework is a hierarchy of program execution equivalence relations that can be used to prove properties about equivalence relations, parallel program executions, and about parallel programs. We present some results for determining feasibility for an execution. Finally, we discuss three applications of this framework: multiprocessor event traces; comparison algorithms for executions; and trace migration.

1. Introduction

Event traces have been fundamental to performance evaluation methodology since the 1960s. Event traces have been used for processor workloads in trace driven simulations of memory hierarchies in shared memory computer architectures [2] [12], to study the behavior of cache systems [29], in the study of page replacement algorithms [23], etc. Event traces are used for parallel language debugging [24]. Recently, the area of

* Supported by a grant from Convex Computer Corporation

performance analysis and visualization has used event traces in performance tuning environments [25] [27].

In this paper we generalize the notion of an event trace by considering program executions. A program *execution* 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. The *execution order* (or path) for an execution, is the actual ordering of program events in 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 paper focuses on causal executions.

Here, executions are defined in a formal model similar to that of [24] which is based upon Lamport's theory of concurrent systems [20]. In this model and execution is represented by a set of events (that occurred during the execution), a temporal relation representing the temporal ordering that occurred during the execution, and a shared data dependence relation representing the shared data dependencies observed during the execution.

The issue of correctness of an execution is crucial for those areas of research that use event traces or execution information. Problems arise in the collection of execution information for parallel programs that do not occur in collecting execution information for sequential programs. These problems include perturbation of the outcome of a program via instrumentation [22] and distortions of an execution execution order distortion [31]. In both cases, nondeterminism (program level and execution architecture nondeterminism[1]) is the cause of these problems.

Research in collecting event traces for parallel programs on shared memory multiprocessors has focused on instrumentation techniques and perturbation analysis [22] [10] [14] [7], establishing causal relations between events and activities during execution of a program [10] [21] [17], data filtering and management of potentially large event traces [30] [10] [1], and trace validity [5] [18] [16] [13] including trace migration [17]. Our research focuses on the issue of trace validity.

Due to program level nondeterminism, executing a parallel program two different times with a fixed data set, can result in two distinct executions. In order to collect

1 It should be noted that first-come first-serve scheduling can be a source of nondeterminism in a multiprocessor workload however, we currently consider only program level nondeterminism.

execution information, programs must be observed during execution. An execution is considered to be *correct* if the execution is the same execution that would have resulted in the absence of observation. Changes in execution path that result from observation are called *distortions*. Distortion arises in three forms: wait time distortion; access order distortion; and execution order distortion. Both wait time and access order distortions can be accounted for by a simulation however little work has been done to account for execution order distortions. Our work focuses on the issue of trace validity and in particular on characterizing and quantifying execution order distortion.

In this paper, we extend the work of Holliday and Ellis [17] by proposing a taxonomy of intermediate code instructions based upon an instructions contribution to execution order distortion. The taxonomy relies on the notion of a *Trace Change Point* or TCP[2] in a source program. A TCP is any program instruction whose outcome depends on a nondeterministic value computed in the program. For example, a conditional statement whose outcome depends on the value of a shared variable, is a TCP. Trace change points are also exactly the points of a program where execution order distortion can occur. The outcome of TCP instances in an execution induce a structure on that execution. We believe that this structure can be exploited for some algorithms on executions (e.g., comparison algorithms).

Combining the basic model for program executions together with the notion of TCPs, provides a framework for studying execution order distortion. This framework is a hierarchy of execution equivalence relations that we use to prove properties about classes of equivalence relations, classes of parallel program executions, and about classes of parallel programs.

Our framework consists of four classes of equivalence relations: *data*, *strong structural*, *weak structural*, and *arbitrary* equivalences. Data and TCP equivalences are based loosely on the notions of *strongly determinate* and *weakly determinate* task systems (see [6]). Informally, an equivalence is a data equivalence if it equivalences executions that write the same sequence of values to shared memory. An equivalence is a strong structural equivalence if it equivalences executions that execute the same set of TCP instances. A weak structural equivalence only requires that a subset of the TCP instances in equivalent executions be same. Finally, an arbitrary equivalence is any equivalence that does not fit into one of the other three classes. We prove some results for determining feasibility for an execution within the hierarchy.

Finally, we have developed the hierarchy in hopes of being able to quantify execution order distortion. The degree to which a program suffers from execution order distortion is closely tied to other problems of interest. For example, the trace migration problem is more difficult for a program that suffers from execution order distortion than for one that does not. For the latter, trace migration is trivial. In addition, some problems are not reasonable unless executions have some exploitable structure (e.g., trace

2 This is the *Address Change Point* of [17].

comparison). Ultimately, any measure of execution order distortion for a program, must rely a quantification of the variance of structure possible for the set of program executions.

In Section 2, we discuss background for trace validity and execution order distortion as well as some related work. Section 3 provides our execution architecture and language assumptions and the basic model for program executions. In Section 4, we propose a taxonomy of program statements and instructions based upon their contribution to execution order distortion. Section 5 present the hierarchy together with some results for determining feasibility for executions. Finally, in Section 6, we discuss three applications of this framework: multiprocessor event traces; comparison algorithms for executions; and trace migration. Section 7 concludes and proposes future directions for this research.

2. Background and Related Work

Areas of research for event traces include the collection of traces and trace validity. Trace collection methods include hardware, microcode, and software based collection methods. For the purposes of this paper, we focus on trace validity. For a detailed overview of both trace collection and trace validity, see [31].

2.1. Execution Validity and Execution Order Distortion

The need to accurately observe program executions arises in several areas of research (e.g., performance analysis, monitoring and visualization). Such observation can perturb program execution causing distortions in the execution that is actually observed. An observed execution is considered to be *correct* if the observed execution is the same execution that would have resulted in the absence of observation.

Problems arise in parallel program observation that do not occur for sequential programs. In either case, observation can cause changes in the time at which a program event actually occurs. However, for parallel programs, the order in which events occur may also be affected. Therefore, a parallel program that is executed multiple times with the same data set may produce different executions. This is due to program and execution architecture nondeterminism. We focus on program level nondeterminism.

Variations in executions over subsequent runs, using the same data set, are termed *distortions*. These distortions fall into three distinct categories [31]: access order distortions; wait-time distortions; and execution order distortions. Access order distortion occurs when instructions are reordered that are not accesses to shared memory — instructions having different possible interleavings. Wait-time distortion occurs when the time that a process waits at a synchronization point is modified. Execution order distortion represent a change in the *execution path* for a program and occur when the order of access to shared variables is modified.

Access order distortion is not considered to be an issue in execution validity. The semantics of parallel programming languages allow for alternate interleavings of instructions. Wait-time distortions are as well, not considered to be a issue in execution validity. The wait time for processors can be corrected postmortem. In contrast, currently, execution order distortion cannot be accounted for postmortem nor can it be discounted on the basis of language semantics.

2.2. Related Work

It has been shown that very long address traces [5] are needed for accurate simulations[3]. In addition, the work serves as a justification for on-the-fly analysis of trace. The problems that arise in storage and buffering of very long address traces create a need for on-the-fly analysis of traces.

In their study [13], Flanagan et al. demonstrate the need for complete trace information in trace driven simulations. Complete[4] and accurate traces[5] were collected with a hardware monitor capable of collecting long address trace. Performance studies were conducted on the traces in various stages of completeness. The results of this study show that simulation results using traces without system references and multiprogram workloads can be in error as much as a factor of 50 to 110 as compared with the same simulations using complete traces. The authors show that these errors can lead to substantial errors in performance estimates.

Koldinger, Eggers, and Levy [18] study the effects of time dilation[6] in traces on the accuracy of trace-driven simulation results. Traces are produced in which time dilation are artificially placed (using a software based trace generation system). Their results indicate that increasingly larger dilation factors did not significantly impact the results of their simulations.

Holliday and Ellis [17] identify the causes of execution order distortion for a medium grained intermediate code language. In addition, they present an algorithm for migrating traces[7]. A detailed discussion of programs that make migration difficult is presented. However, it remains unclear whether or not migration is feasible for the language that they use (containing spinlocks). The question that remains is whether there exist classes of nondeterministic programs for which migration is feasible or if migration is simply catastrophic in the sense that program re-simulation cannot be avoided for anything but trivial programs.

---

3 Specifically for studies of architectures using large second level caches.

4 The authors defined *complete* traces to contain all cpu generated references including those produced by interrupts, all system calls, exceptions and handler references, references due to any supervisory activities, and user process references.

5 It should be noted that there are some minimal effects on traces due to perturbation. See [13] for more details.

6 Time dilation is the ratio of program execution without instrumentation to that of execution with instrumentation.

7 Trace migration is the process of using a source trace collecting on a multiprocessor to produce a correct trace on another multiprocessor environment. These multiprocessors must be in the same *family* of multiprocessors. That is, changes in environment are only parametric (e.g., changes in the number of available processes, memory size, etc.).

Recently, a study by Goldschmidt and Hennessy [16] demonstrated that traces containing time dependencies lead to inaccurate simulation results. They compare trace-driven simulation results to those of direct simulation[8] and conclude that for workloads based upon traces from programs that contain nondeterminism or first-come first-serve scheduling algorithms, trace-driven simulation produces inaccurate results.

The parallel compilers and debugging community have done much research into detection and formal characterization of race conditions in parallel programs. In addition, notions similar to that of the *Address Change* and *Affecting Points* of [17] have been introduced. In his thesis [24], Netzer studies race condition detection and debugging for shared memory multiprocessors. Race condition detection differs from our work not only in intent, but also in the scope of traces considered. Since the goal of race condition detection is to detect bugs in parallel programs, feasible sets of program executions in general, contain only executions that are a prefix of an execution of interest. However, in describing sets of executions with differing shared data dependencies, Netzer formally describes the process of the outcome of one event effecting another. This notion is similar to our characterization of the causes of execution order distortion. It should be noted that other similar notions include the *hides* relation of Allen and Padua [4] and the *semantic dependencies* of Podgurski and Clark [26]. We adopt Netzer's basic model of program executions (based upon that of [20]).

Perturbation analysis [22] has been used to remove the perturbation introduced into a trace by collection method. Traces are analyzed postmortem and an approximation to the actual trace that would have occurred in the absence of observation is produced. That is, time perturbations are removed and the instruction interleaving of a trace is adjusted to reflect these time adjustments. During analysis, removing perturbations may cause the choice of trace to shift among the traces in the set of feasible traces — a different instruction interleaving may be chosen. A feasible set of traces is a set in which all members execute the same set of events. However, if removing a perturbation changes the execution path of the program, an approximation to the correct trace is made. That is, the closest possible feasible path is chosen.

3. Preliminaries

In this section, we present basic assumptions that we make about executions. These include assumptions about both the execution architectures on which programs are executed, and programming language assumptions. In addition, we define executions in a formal model of program executions similar to that of [24] which is based upon Lamport's theory of concurrent systems [20].

---

8 Direct simulation does not use traces. Instead, a *workload interpreter* produces events (one at a time) and latencies for those events are fed back into the interpreter before a new event is generated.

3.1. Environmental 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).

We assume sequential consistency[9] [19] (e.g., the Convex SPP and the KSR-1) for the shared memory multiprocessor. 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 multiprocessor that are not sequentially consistent (e.g., the Convex C-series and the Sequent Symmetry).

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

Finally, we assume that programs are correct. In particular, we assume that all programs terminate normally.

3.2. Language Assumptions

The next section considers the causes of execution order distortion from both a statement and instruction level perspective. In order to make this discussion intelligible the language under consideration is a simple 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(x)*[10] operation. This canonical form allows for a single shared variable or array reference per statement. Any program 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 a hypothetical programming language.

The **task_create(n)** statement executes n processes concurrently; one for each of the n tasks being executed. In Figure 3.1, there are two such tasks. Each tasks 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.

Figure 3.2 (below) 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

9 In a sequentially consistent shared memory multiprocessor 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).

10 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
        int aval;
        int k = 0;
    A: lock(lk);
    B: index = CONSTANTVAL;
    C: unlock(lk);
    D: aval = array[index];
    E: for (k = k+1; k < aval; k++);

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

task_terminate(n);

end;
```

Figure 3.1 A program using spinlocks for exclusive access to shared memory.

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.

```
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;
        A: acopy = test-and-set(lock);
        B: if acopy == 1 goto A;
        C: index = CONSTANTVAL;
        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;
        A: bcopy = test-and-set(lock);
        B: if bcopy == 1 goto A;
        C: index = CONSTANTVAL;
        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 3.2 The program of Figure 3.1 represented in intermediate code form.

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 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. Similarly 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 possible intermediate code statements listed below are from Holliday and Ellis [17]. We have added the 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*$(p_1,p_2,...,p_n)$ 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 machine language instructions (in the machine language of the host and target machines) which we will refer to as *instructions*.

Any program can be transformed into an intermediate code form obeying restrictions 1 and 2 using simple syntactic transformations. In Section 6, we will demonstrate that this transformation is consistent with our model.

3.3. A Model for Program Executions

We present a model for program executions based on a language that includes the intermediate code statements listed in Section 3.2, the *task_create* and *task_terminate* process control statements, and the *lock* and *unlock* synchronization statements. 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[11]. A *program event* is an execution instance of one or more consecutively executed intermediate code statements (or instructions). Given a fixed data set, each program has a set of possible executions.

In Lamport's theory of concurrent systems [20] there is no assumption of atomicity for language statements (or instructions). This provides a formalism with which to reason about program executions 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* $\neg(b \rightarrow_T a)$[12]. That is, neither a nor b completes before the other begins — their executions overlap.

A program execution is a triple consisting of a set of *events*, a *precedes* relation $\rightarrow_T$, and a *shared data dependence* relation $\rightarrow_{sd}$. The set of events represent all instances of statements (or instructions) performed during execution. A pair of events *a* and *b* are members of *direct shared data dependence* relation if the *a* affects a shared variable that *b* directly depends on and at least one of these shared data accesses modifies a variable. The shared data dependence relation, $\rightarrow_{sd}$, is the transitive closure of this relation. More formally,

---

11 Notice that if a program is transformed from one using the *testandset* statement, the latter is always true.

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

Definition 3.3.1 : A *program execution*, P, is a triple $<E,\rightarrow_T,\rightarrow_{sd}>$ where E is a finite set of events, $\rightarrow_T$ is the temporal ordering relation defined over E, and $\rightarrow_{sd}$ is the shared data dependence relation also defined over E.
■

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. It is possible to construct executions that could not actually have occurred simply by executing statements in an order that violates program semantics. 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 i^{th} 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 for the relations $\rightarrow_T$ and $\rightarrow_{sd}$ are those given in [24]. These axioms constrain these relations to be consistent with the semantics of our chosen intermediate code language. Axioms A1 and A2 make $\rightarrow_T$ consistent with the notion that an ordering based upon the start and finish times of all events is total.

A1. $\rightarrow_T$ and $\rightarrow_{sd}$ are irreflexive partial ordering relations.

A2. 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.

Axiom A2 states that if events b and c *overlap* and event a completes before b begins and event c completes before event d begins, 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 3.3, from [20], illustrates A2.

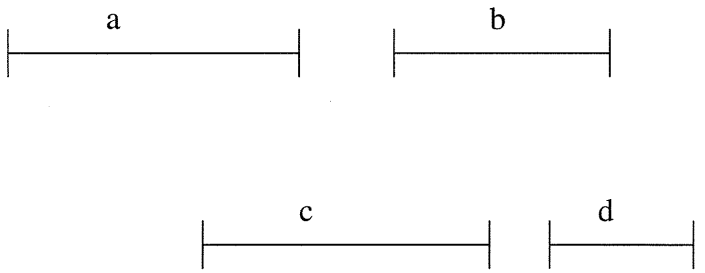


Figure 3.3 An Illustration of A2.

Axiom A3 is the law of causality: if event a affects event b, then a must precede b in time.

A3. If $a \rightarrow_{sd} b$ then $\neg (b \rightarrow_T a)$.

Axiom A4 ensures that a linear ordering of events is preserved within a process (thread).

A4. $e_{p,i} \rightarrow_T e_{p,i+1}$ for all processes p and events $e_{p,i} \in \mathbf{E_p}$.

Axiom A5 ensures that each *task_create* instruction has a corresponding *task_terminate* instruction.

A5. $task\_create_{p,i} \rightarrow_T e_{c,j} \rightarrow_T task\_terminate_{p,i+k}$ for all $e_{c,j} \in \mathbf{E_c}$.

Finally, Axiom A6 ensures that the number of lock statements that have already occurred at any point in the program is less than or equal to the number of unlock statements that have either already occurred or have begun together with the number of locks. It is assumed that each lock is initially unlocked and that the number of locks is given by n,

A6. For every $L \subset E_{l(v)}$,

$$| \{ u \mid u \in E_{u(v)} \text{ and } \exists l \in L (u \rightarrow_T l \vee \neg (u \rightarrow_T l \mid l \rightarrow_T u) \} | + n \geq | L |.$$

Finally, it is important to point out that executions carry only that state information which they embody. That is, if state information can be determined from the execution path of a program, then it is present, otherwise, it is not.

4. A Taxonomy for Execution Order Distortion

In this section, we categorize program statements based on the effect of that statement on the execution path realized in an execution. That is, we determine the degree to which a statement (or instruction) contributes to execution pattern distortion.

Executions contain only state information that is embodied in an execution path. This leads us to a notion of program determinism and nondeterminism that differs from traditional definitions. We define the notions of *execution order determinate* and *execution order nondeterminate* for a parallel program. These definitions hold for parallel programs that have a single execution path or many execution paths respectively.

4.1. Instructions and Execution Pattern Distortion

Intermediate code statements can be classified as static, runtime[13], or dynamic based on whether or not their outcome can be determined statically, are fixed after one run, or vary over subsequent runs (on the same data set) respectively. Since statements are comprised of sequences of instructions, we will also classify instructions as static, runtime, or dynamic.

In general, an execution consists of statement or instruction executions instances (depending upon the level of detail present in the execution). Here, we discuss executions of both kinds. We refer to the outcome of an individual instruction (or statement) instance as having an *execution contribution*. This contribution is simply that a particular instance occurs. We return to Figure 3.2 to demonstrate our taxonomy.

A *static* instruction is one whose outcome is fixed regardless of program state. 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. Thus, the outcome of these 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 :

> **Constant Static Statements** These are static statements comprised of instructions that generate no stores to shared variables.
> **Static Affecting Points** These are static statements that contain a store of a shared variable.

The static affecting points are distinguished from constant static statements because they may affect a change in the execution pattern for a program. However, they can still be determined statically and in both cases, the trace contribution is fixed regardless of program state.

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 state-

13 These are the *easy*, *hard*, and *impossible* instructions of [21]. The *hard* and *impossible* instructions are our *runtime* statements.

ments can be determined statically[14] except 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).

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 or Trace Affecting Points These statements are runtime statements that are comprised of instructions one operand of which generates a store to a shared variable.

We call 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 dynamic instruction is one whose trace contribution depends on program nondeterminism. That is, the instruction depends on the value of a shared variable (either directly or indirectly). Unlike runtime instructions, the trace outcome 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[15] For example, a conditional statement whose outcome depends on the value of a shared variable is a dynamic statement. Dynamic statements fall into two 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).

Notice that there is no notion of a dynamic affecting point due to the restrictions in Section 3.2. In addition, the outcome of task control statements can vary over subsequent runs of a program on a fixed data set. This is due to changes in processor scheduling from one run to the next. The following statements are dynamic statements:

Task Control Statements These are statements that contain either a **task_create** or a **task_terminate** statement (regardless of dependency on shared variable

14 It should be noted that the cost of determining these instructions statically may be prohibitive.

values).

Nondeterministic conditional statements, dynamic dependent statements, and task_create statements are called *Trace Change Points*[16] (TCPs) because these are exactly the statements whose execution can result in a change in execution path for a program. The task_terminate statements are not TCPs since their outcome does not depend on any variable values. Notice that both the *lock* and *unlock* statements are dynamic statements or TCPs because they are comprised of statements that are TCPs. Returning to Figure 3.2, statements B, E, and G (as well as the process control statements) 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/bval*. Notice too that function invocations are not TCPs. Rather, for the purposes of determining the set of TCPs for a program, functions can be treated as in-line code. The set of TCPs for a function must be fixed and is therefore taken to be the union of the sets of TCPs from each possible invocation of the function. For the model, we make no assumptions about side effects. Finally, for structuring purposes, we consider the *begin* statement to be a TCP.

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 path of the program but only at a trace change point. 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 Trace Affecting Point. In Figure 3.2, statement B_0, a TCP, has two use-definition chains: (A_1) and (D_1,A_0) (where subscripts denote processor numbers). The TAPs for TCP B_0 are A_0, A_1, and D_1.

4.2. Computing the Set of Program TCPs

The definition and use of the model does not require that the set of TCPs for a program actually be calculated. However there are applications that will use this set and require its calculation. To our knowledge this is the first attempt to address this problem[17].

---

15 This is due to the restrictions on three address intermediate code given in Section 3.2.

16 Holliday and Ellis, in [17], define *Address Change Points*. The difference is that ACPs refer to the statements at which actual changes in the address trace occur, while in the case of 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., **Btask_create** and **task_terminate**).

17 The problem of computing the set of TCPs for a program was not addressed in [17].

The problem of determining the set of program TCPs requires that the set of variables dependent on shared variables be known at any point in the program. Since the set of TCPs for a program must be fixed, the largest set of variables dependent on shared variables must be computed for each point in the program. The program fragment in Figure 4.1 demonstrates the need for a conservative algorithm to compute the set of variables dependent on shared variables at any point in the program. Notice that statement D will never be executed (unless there is another thread accessing shared variable s). In general, it is either impossible or unreasonable to determine whether or not a statement will actually be executed. For this reason, we claim that any algorithm computing the set of program TCPs should be a conservative algorithm (i.e., an algorithm that may compute a slightly larger set including statements that are not TCPs but containing all statements that are TCPs).

The problem of determining the set of program TCPs can be computed using data flow analysis [15]. In [9] we present a data flow formulation of the problem and an algorithm for computing the set. The algorithm produces a conservative estimate of the set of TCPs and runs in $O(n^2)$ time (where n is the size of the program source).

4.3. Execution Order Determinate and Nondeterminate Programs

Given the taxonomy, it is useful to distinguish programs as determinate or nondeterminate with respect to execution order distortion[18]. Intuitively, an execution order

```
shared int s = 0;
   .
   .
   .
A: s = 1;
B: v = s;
C: if v != 0 goto E
D:    k = 20;
E:    .
   .
   .
```

Figure 4.1 A Contrived Example

18 Other classifications of parallel programs exist. A widely used classification has been proposed by Emrath and Padua [11]. 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 er-

determinate parallel program is one that does not suffer from execution order distortion. That is, a program that contains no trace change points (except the initial and final program statements).

> **Definition 4.4.1** A parallel program is *execution order determinate* if it contains only static statements, runtime statements, and process control statements whose outcome are not dependent on the value of shared variables.
> ■

This notion of determinate should not be confused with the traditional notions of determinate programs. Notice that the definition encompasses programs that can have a different state upon termination for the same input but that contain no trace change points in which the execution order for an execution can vary. Such a program is not a deterministic program. However, if our view of program execution is based upon execution order, such a program does have a *fixed* execution path.

A parallel program is execution order nondeterminate if it can result in more than one execution path. That is, the program contains trace change points (in addition to the initial and final program statements).

> **Definition 4.4.2** A parallel program is *execution order nondeterminate* if the program contains dynamic statements at least one of which is dependent on the value of a shared variable.
> ■

In the remainder of this paper, we use the terms execution order determinate and determinate interchangeably. Similarly, we use the terms execution order nondeterminate and nondeterminate interchangeably.

5. A Framework for Execution Order Distortion

In this section, we present a formal model in which to reason about execution order distortion. We use the model of program executions given in Section 3.3. In this model program executions are represented as a set of event instances together with the temporal order and shared data dependence observed during execution.

---

rors that occur due program execution or exhibit *true* nondeterminism. Our classification differs from this classification in intent. Programs that are internally determinate would be execution order determinate. Programs that are externally determinate would be execution order nondeterminate (since we are concerned with execution path, not program values). Likewise, true nondeterminism would also fall into this category. Programs that are nondeterministic due to round off error could fall into either category (depending on whether or not roundoff errors effect the program executions path).

We define a hierarchy of possible equivalence relations with which to partition a set of program executions. We use the notion of Trace Change Points to partition sets of executions based upon the TCP structure of an execution — that is, executions can be viewed as having structure based upon the TCP instances that they execute. The hierarchy has four levels each based upon a different level of structural equivalence of executions.

An equivalence relation on program executions allows the set of program executions to be grouped into like sets — executions with similar structure according to a relation, reside in the same partition. Imposing a hierarchical structure on the universe of such equivalences allows us to alter the size of partitions, thus providing a consistent framework in which to study the similarities in the structure of executions (for a given program). The hierarchy embodies 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.

The notion of *feasibility* of an execution for a program (and data set) is a fundamental one. An execution is feasible if it is one that could have actually occurred in an execution of the program. We prove some general results about the problem of determining feasibility for the hierarchy.

5.1. The TCP Precedence Relation

Programs that are execution order determinate suffer no execution order distortion (i.e., they have a single execution path for a given input). Such programs can be determinate or nondeterminate. Unfortunately, knowing that a program is determinate provides little information about the set of executions paths for that program. For example, a program that is determinate but not execution order determinate can have many possible execution paths. On the other hand, knowing the TCP structure of a program can provide useful information about the set of execution paths for a program. For example, a program that always executes the same set of TCP instances (possibly with differing outcomes) has a very closely related set of execution paths. Trace Change Points are singled out as the cause of changes in execution order for a program. We propose that executions have structure, their *TCP structure*, and that this structure can be used to answer useful questions about parallel programs (e.g., how difficult is trace migration for this program, or to what degree does this program suffer from execution order distortion).

In Definition 5.1.1, we define the *TCP precedence* relation for a program execution. This relation can be derived from the $\rightarrow_T$ relation 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 **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 5.1.1 : Given a fixed but arbitrary program execution $P=<E,\rightarrow_T,\rightarrow_{sd}>$. The *TCP precedence* relation for P, denoted $\rightarrow_{tp}$, is the following relation:

For any a,b$\in$ TCP(E): a $\rightarrow_{tp}$ b if and only if

1. a $\rightarrow_T$ b and

2. $\exists p$ such that a$\in E_p$ and b$\in E_p$ or a$\in$ task_create(E) or $\exists$ c$\in$ task_terminate(E) such that a $\rightarrow_T$ c and c $\rightarrow_T$ b.

■

Restriction 2 requires the TCP precedence relation to *forget* the interleaving of TCPs that actually occurred in an execution among different processes. For example, for one possible execution, events e_1 (a TCP event of process 1) and e_2 (a TCP event of process 2) we might have $e_1 \rightarrow_T e_2$ and in another execution we might have $e_2 \rightarrow_T e_1$. In either case, this information is omitted from the TCP precedence relation. Rather than focus on the interleaving of individual TCP instances, we are concerned with which TCPs did occur, the outcome of these TCPs, and the overall structure of the execution. Notice that the interactions among processes are reflected only in the set of TCPs executed rather than in the set of TCPs and in synchronization information.

Two programs that execute the same set of TCPs may not have executed the same set of instruction instances. This is because the outcome of the TCPs may not have been the same. Definition 5.1.2 presents the notion of the *Outcome* of a TCP. Recall that TCPs come in four forms: conditional instructions, array references, task creation instructions, and the program begin instruction.

Definition 5.1.2 : Given two executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$ and an instruction instance b such that b$\in$TCP(E_1) and b$\in$TCP(E_2), the outcome of the instance in each execution are *outcome equivalent*, denoted Outcome(P_1,b) = Outcome(P_2,b) if one of the following is true:

1. b is a conditional instruction and there exists c$\in E_1$ and c$\in E_2$ such that b $\rightarrow_{T_1}$ c and there does not exist a d$\in E_1 \cup E_2$ such that b $\rightarrow_{T_1}$ d and d $\rightarrow_{T_1}$ c or b $\rightarrow_{T_2}$ d and d $\rightarrow_{T_2}$ c.

2. b is a task_create(p) instruction and for each c$\in E_{i,1}$, i = 1 to p, such that

$$\{ b \rightarrow_{T_1} c \wedge \neg (\exists d \mid d \in E_{i,1} \wedge b \rightarrow_{T_1} d \wedge d \rightarrow_{T_1} c)\}$$

implies

$$\{ b \rightarrow_{T_2} c \wedge \neg (\exists d \mid d \in E_{i,2} \wedge b \rightarrow_{T_2} d \wedge d \rightarrow_{T_2} c)\}$$

3. b is an array reference or the begin statement of the program.

■

Two instances of a conditional instruction have the same outcome if for their thread of execution, they are immediately followed by the same instruction instance. Two task_create instructions have the same outcome simply if they result in the same number of threads and that the first instruction executed in each thread is the same. Finally, array reference instructions and the begin statement for an execution always have the same outcome. That is, the occurrence of b in both executions implies that the outcome of these instructions was the same (e.g., an array reference that results in an instructions **load** *addr* will either have the same address reference in both executions or b won't be a member of both E_1 and E_2).

Finally, two executions have the same TCP structure if they are TCP equivalent, see Definition 5.1.3. Intuitively, two executions are TCP equivalent if they execute the same set of TCP instances and have the same outcome for each instance.

Definition 5.1.3 : Two executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$with TCP precedence relations $\rightarrow_{tp_1}$ and $\rightarrow_{tp_2}$ respectively, are *TCP equivalent*, denoted $P_1 \equiv_{TCP} P_2$,

1) if $\rightarrow_{tp_1} = \rightarrow_{tp_2}$,

2) for each b $\in$ TCP(E_1) such that b $\in$ TCP(E_1), Outcome(P_1,b) = Outcome(P_2,b).

■

5.2. A Hierarchy of Execution Equivalence

Given a notion of TCP structure for a program, we define a hierarchy of execution equivalences. While the hierarchy is motivated by data and TCP determinate task systems of [6], our notions of data and TCP differ. We define four levels of execution equivalence based upon these notions. They are: *data*; *strong structural*; *weak structural*; and *arbitrary* execution equivalences. We have the following relationship:

$$\text{data} \subset \text{strong structural} \subset \text{weak structural} \subset \text{arbitrary}$$

Recall that a *strongly determinate* task system is one in which the sequence of values written to shared memory are the same for any execution. A task system is *weakly terminate* if for any execution, the final outcome of shared memory is the same for each task. Notice that our notion of an execution does not contain a literal record of the values of shared variables. This information is not saved in an execution and the only state information present being that which is embodied in the execution path of the program.

A *data* equivalence is one in which executions are equivalent if they have the same shared data dependence. This implies that the same sequence of values were written to shared memory for every execution in a partition. More formally,

Definition 5.2.1 : An equivalence relation $\equiv$ on a set of program executions (for a given program and fixed but arbitrary data) is a *data* program execution equivalence if for any two program executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$, $\rightarrow_{sd_1} = \rightarrow_{sd_2}$.
■

If the set of executions for a parallel program using fixed but arbitrary input data partitions into a single set under a data equivalence then the program is a *data* equivalent program.

A *strong structural* execution equivalence is one in which two executions are equivalent if they are TCP equivalent (Definition 5.1.3). We intend to convey the notion that for any two executions that are equivalent under some strong structural equivalence, the executions *appear* to have had the same state of shared memory at each TCP (since the outcome of each TCP is the same). Due to the lack of state information in an execution, we cannot determine, from two such executions, whether or not the state of the program, with respect to its shared variables, was different at a particular TCP instance. Therefore, from our perspective, the state may as well have been the same, even though the access to shared memory may differ. More formally,

Definition 5.2.2 : An equivalence relation ≡ on a set of program executions (for a given program and fixed but arbitrary data) is a *strong structural* program execution equivalence if for any two program executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$, $P_1 \equiv_{TCP} P_2$.

■

If the set of executions for a parallel program using fixed but arbitrary input data partitions into a single set under a strong structural equivalence then the program is a *strong structural* equivalent program.

We can extend the notion of a strong structural equivalence by relaxing the requirement that equivalent executions must execute exactly the same set of TCPs. We do this by requiring only that they execute a nonempty proper subset of the same TCPs. Intuitively, executions in which *some* of the program TCPs are executed yielding the same outcome are more equivalent than executions that execute no equivalent TCPs (other than the begin statement). These equivalences are the weak structural execution equivalences. A *weak structural equivalence* would be one in which, for two equivalent executions, one could not determine that the state of the program was different at a *subset* of TCP instances. More formally,

Definition 5.2.3 : An equivalence relation ≡ on a set of program executions (for a given program and fixed but arbitrary data) is a *weak structural* program execution equivalence if for any two program executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$, if for some S a nonempty subset of $(TCP(E_1) \cap TCP(E_2))$ - {begin}, for all b∈ S,

$$Outcome(P_1,b) = Outcome(P_2,b).$$

■

If the set of executions for a parallel program using fixed but arbitrary input data partitions into a single set under a weak structural equivalence then the program is a *weak structural* equivalent program.

Weak structural equivalences are required to equivalence at least one TCP instance (not including the begin statement). The arbitrary execution equivalences are derived from the weak structural equivalences by lifting this restriction altogether. More formally,

Definition 5.2.4 : An equivalence relation ≡ on a set of program executions (for a given program and fixed but arbitrary data) is a *arbitrary* program execution equivalence if for any two program executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$, if for some $S \subset (TCP(E_1) \cap TCP(E_2))$, for all $b \in S$, $Outcome(P_1,b) = Outcome(P_2,b)$.
■

Notice that the only difference between a weak structural and an arbitrary equivalence is that a weak structural equivalence requires executions to have at least one common TCP with the same outcome. While an arbitrary equivalence requires no common TCPs with the same outcome (and in fact, requires no common TCP instances). Notice that in the above definition, S may contain only the begin statement. At first glance, it might look like there is a single interesting arbitrary equivalence. This is not the case! Consider an equivalence that requires that executions share the same set of TCP instances but does not require that the outcome of these instances be the same (see (5) below). This is an arbitrary executions and this partitioning yields sets of executions that are similar in their execution paths. If the set of executions for a parallel program using fixed but arbitrary input data partitions into a single set under a arbitrary equivalence then the program is a *arbitrary* equivalent program.

Next, we present examples of equivalences of interest and identify where they are in the hierarchy. Given two executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$ with TCP precedence relations $\rightarrow_{tp_1}$ and $\rightarrow_{tp_2}$ respectively. The following are data equivalences:

(1) $P_1 \equiv P_2$ if and only if $P_1=P_2$.

(2) $P_1 \equiv P_2$ if and only if $\rightarrow_{sd_1} = \rightarrow_{sd_2}$.

The equivalence given in (1) partitions a set of executions such that there is a single execution per partition. In this view of executions simple interleaving of instructions distinguishes executions and even execution order determinate programs have a single execution, per partition. This very strict equivalence typifies the view that executions are distinct. The equivalence given in (2) places executions in the same partition if they exhibit the same access pattern to shared memory. We know that if two executions exhibit the same sequence of accesses to shared memory, then the executions computed the same final state. This equivalence typifies the view that executions are indistinguishable if they are guaranteed to result in the same final state.

(3) $P_1 \equiv P_2$ if and only if $P_1 \equiv_{TCP} P_2$

The equivalence given in (3) is a strong structural equivalence. Here, executions reside in the same partition if they execute the same set of TCPs with the same outcome. Notice that two executions can access shared memory differently and still reside in the

same partition as long as the outcome of TCPs remains unchanged. That is, executions are partitioned into execution order determinate sets of executions.

For the next example, we use **CT**(E) to denote the set of task_create and task_terminate statement instances in an execution.

(4) $P_1 \equiv P_2$ if and only if:

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

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

The equivalence given in (4) is a weak structural equivalence. Executions reside in the same partition only if they have the same task_create (and therefore also task_terminate) structure.

.5i (5) $P_1 \equiv P_2$ if and only if $\rightarrow_{tp_1} = \rightarrow_{tp_2}$.

The equivalence given in (5) is an arbitrary 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. This equivalence raises the question of whether or not we should be concerned with TCP instance outcome (as in the strong structural equivalence (3)) or whether it is sufficient to consider the potentially larger partitions induced by (5).

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

Finally, (6) is an arbitrary equivalence in which all executions reside in the same partition. This typifies the view that any execution of a parallel program is as representative as any other.

In Theorem 5.2.6 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 partitionings of the set. Some intuition about this sufficiently complex program is needed. We will use the program in Figure 5.1 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 a data equivalence from a strong structural 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 5.1 statement F provides this behavior. Regardless of the interleaving of statements E and L, this statement will have the same outcome (for certain choices of input data). To distinguish structural from strong structural equivalences, 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 5.1, this is accomplished by

having executions with the same task_create/task_terminate structure but that have different TCP outcome over task 0. Finally, in order to distinguish weak structural and arbitrary equivalences the program conditionally creates tasks 3-5.

```
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 5.1

Theorem 5.2.6 : 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), (3), (4), and (6) above are sufficient to demonstrate that the hierarchy is populated. The program of Figure 5.1 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 set {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 K C E L F H J
(S2) A B K C L E F H J
(S3) A B C D K E F L G H J
(S4) A B C D K E F G L H I J

First, notice that the data equivalence (1) places each of these executions in a different partition due to the interleaving of statements E and L (for S1 and S2) and the interleaving of statements G and L (for S3 and S4).

For the strong structural equivalence (3) statements S1 and S2 end up in the same partition. This is because TCP C branches whenever B completes before K (resulting in t = -1). If E completes before L then t = -1, and if L completes before E, t = 1. In either case for k = 3, F branches. At this point since both B and K and E and L have completed (leaving i = 1 and j = 1), H branches. Therefore, these executions execute the same set of TCPs with the same outcome (even though they are interleaved differently). Executions S3 and S4 delay the execution of statement K causes 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. This same delay in executing K coupled with a delay in executing L, causing the value of k to drop to 1, causes the branch at H to fail. This places S4 in a different partition that either S1 and S2 or S3.

For the strong structural equivalence (4), executions S1, S2, and S3 end up in the same partition. This is because statement H branches in each of these cases. Execution S4 ends up in a separate partition since the late execution of K, reducing k to 2, coupled with the late execution of L, allowing G to reduce k again to 1, results in the branch at H failing. The execution of tasks 3-5 cause S4 to have a different task_create/task_terminate structure than executions S1, S2, or S3.

Finally, all of the above executions reside in the same partition for the arbitrary equivalence (6). This is simply because (6) does not distinguish any executions.

Since equivalences (1), (3), (4), and (6) exist and partition a set of executions from the program of Figure 5.1 distinctly, the hierarchy is nontrivial.
□
trace migration.nr ?k 0

5.3. Feasible Executions

The notion of *feasibility* of an execution for a program (and data set) is a fundamental one. An execution is feasible if it is one that could have actually occurred during some execution of the program. The problem of determining feasibility is a central one for trace migration and other trace related problems (e.g., perturbation analysis, debugging, etc.). 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.

For our framework, we are interested in determining whether or not an execution is feasible and is a member of a particular partition — or feasible for the partition. We are interested in whether or not an execution embodies enough state information such that it can be used to infer the feasibility of another execution. That is, if we have an execution, e_1, from a particular partition and this execution is known to be feasible, then for any other equivalent execution, e_2, can we infer the feasibility (or infeasibility) of e_2 given that we know that e_1 is feasible? As above, we call any feasible execution that is a member of a given partition, such as e_1 above, a *witness* execution for that partition. The problem of determining given a witness execution, whether or not an execution is feasible for a particular partition the *feasibility with witness* problem.

In this section, we are interested in the decidability of the feasibility with witness problem for the various levels of the hierarchy. We show that for most levels of the hierarchy even the feasibility with witness problem will require additional state information.

Definition 5.3.1 describes the necessary conditions for an execution to be equivalent to a witness execution. Any algorithm for determining feasibility with witness must check these condition including the condition that two executions are equivalent. Clearly there are equivalences that cannot be checked without additional state information. That is, while any equivalence in the hierarchy requires some state information, we rule out the use of an equivalence that requires information beyond that which is embodied in an execution. This sort of equivalence defeats our purpose. We will require that any equivalence be *checkable* equivalence (i.e., equivalence can be established without additional state information) and whenever we refer to an equivalence, this is assumed.

Definition 5.3.1 : Given a witness program execution $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and a execution equivalence $\equiv$, an execution $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$ is a member of the same partition induced by $\equiv$ as P_1 if the following are true:

1. P_2 is an actual execution and A1-A6 hold.
2. $P_1 \equiv P_2$.

■

In Theorem 5.3.2 below, we demonstrate that for a data equivalence, determining feasibility with a witness is decidable. That is to say, if we know that two executions have the same shared data dependence, we can infer the correctness of one from the other. Unfortunately, this result does not carry through for any other class of equivalence. In Theorem 5.3.3 we show that any other equivalence does not provide sufficient information to infer the correctness of executions.

Theorem 5.3.2 : There is a decision procedure for determining membership with witness for any data equivalent program execution equivalence.
■

Proof : Consider a witness execution $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and a fixed but arbitrary data execution equivalence $\equiv$. Assume we are given an execution $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$ which may or may not be feasible. A decision procedure determining the feasibility of P_2 works as follows. First, since $\equiv$ is checkable it can be determined whether or not $P_1 \equiv P_2$. If this is not the case, then P_1 and P_2 do not reside in the same partition.. Otherwise we have the following information.

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

(notice that we do not have $\rightarrow_{T_1} = \rightarrow_{T_2}$ unless $P_1 = P_2$). (1) implies that P_2 exhibits a possible shared data dependence (specifically the one exhibited but P_1). Finally, it is a simple matter to verify that P_2 does not violate A1-A6. If it does not, then P_2 is a feasible execution. Otherwise P_2 is not a feasible execution (for any partition).
□

Theorem 5.3.3 : Determining membership with witness for any equivalence that is strong structural, weak structural, or arbitrary is not decidable.
■

Proof : Since we have the following relationship for strong structural, weak structural, and arbitrary equivalences

strong structural $\subset$ weak structural $\subset$ arbitrary

it suffices to show that there is no decision procedure for determining membership with a witness for a strong structural equivalence. Consider the example of Figure 5.1 from the previous section together with the strong structural equivalence (3) from the previous section. Consider a witness execution with the following

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

Execution e1 is a feasible execution. Notice that the execution of statement K before the execution of statement C ensures that statement C fails to branch ensuring the execution of statement D. The following execution is equivalent to e1 under the equivalence given by (3).

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

Notice that if statement K follows statement B, then the value of t at statement C is -1 and the branch is taken. Statement D cannot be executed. Hence e2 is not a feasible execution. Notice however that it took state information not contained in the execution to decide whether or not e2 was feasible. That is we had to compute the value of t for the execution. Therefore, because the feasibility of e2 could not have been decided without program state information above and beyond that which is embodied in the execution, the problem of determining membership with witness for a strong structural (weak structural or arbitrary) equivalence is not decidable.
□

The question that remains to be answered about determining feasibility for executions is how much state information is required to determine feasibility? This question is in general undecidable (e.g., library calls for which there is no source code may exist, etc.). These difficulties aside, currently, it is not known whether or not static analysis can be used to determine feasibility or if the cost of such analysis is prohibitive. Another interesting question is whether or not there exist classes of programs that require a reasonable amount of state information to determine feasibility for executions.

6. Applying the Model

The goal for this model is to establish a characterization of programs according to their susceptiblity to execution order distortion. This susceptiblity is related to the level of nondeterminism that exists in a program. We have defined the model so that we can consider issues related to requirements for tools that analyze or make use of traces in a formal setting. In addition to characterizations of execution order distortion, we are interested in the following questions (among others):

(1) What is a useful measure of nondeterminacy in parallel programs?
(2) Are there classes of programs for which trace migration is reasonable?
(3) Are there reasonable (linear) trace comparison algorithms?
(4) Given two traces from the same program, is there a useful notion of *distance* that can be used for comparison?
(5) In general, is there exploitable structure to traces that can be used to produce linear time algorithms?

Since a trace can be quite large, any reasonable algorithm used by a trace analysis tool must use the trace *on-the-fly* (i.e., must be linear in time given the size of the trace).

We demonstrate the ability to formalize both the notion of a causal trace as well as abstract events. Next we illustrate our approach by demonstrating how the model sheds additional light on the trace migration problem as well the problem of distance measures for traces.

6.1. Multiprocessor Event Traces

In this section, we consider how event traces can be represented in our framework. Recent research includes the use of event traces containing *abstract* events [14] [10] — an abstract event is an abstraction of a set of possible events. Our model is consistent with the use of abstract events in causal traces.

One of the benefits of using Lamport's model is that an execution can be viewed at different levels of detail. This allows our model to encompass the notions of abstract events as well as that of a trace. This flexibility also allows the model to include machine level nondeterminism (although we do not consider this possibility here), since instructions are not required to be atomic.

We now define three alternate views of a program execution: the *higher-level* view (defined by Lamport [20]); the *instruction-level* view; and the *trace* view. A higher-level view allows an execution to be viewed at a higher level, hiding the details of an execution. An instruction-level view allows an execution to be viewed at the finest level of detail. The combination of high level and instruction level views allows one to view an execution at a spectrum of levels of detail. Finally, a trace view simply filters away unwanted details of an execution leaving a possibly incomplete record of the execution.

A higher-level view of a program execution is a new program execution in which the set of events has been partitioned and new relations $\rightarrow_T$ and $\rightarrow_{sd}$ have been created that are consistent with those of the original execution. More formally,

Definition 6.1.1 : A *higher-level* view of a program execution $P=<E,\rightarrow_T,\rightarrow_{sd}>$ is another program execution $P_h=<E_h,\rightarrow_{T_h},\rightarrow_{sd_h}>$, where P_h partitions E and the following hold for all A, B $\in P_h$,

1. $A \rightarrow_{T_h} B \Leftrightarrow \forall\ a\in A, b\in B \quad a \rightarrow_T b$

2. $A \rightarrow_{sd_h} B \Leftrightarrow \exists\ a\in A, b\in B \quad a \rightarrow_{sd} b$

■

It is possible to construct higher-level views that are not valid program executions (i.e., do not obey all of the axioms in Figure 4.3). Definition 6.1.1 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 axioms A5 and A6. We are interested in only those higher-level views that also are valid executions. Any higher-level view obeys axioms A1-A3. Whenever we use a higher-level view it is either shown to obey axioms A4 though A6 or is a valid execution by assumption.

An 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 relations $\rightarrow_T$ and $\rightarrow_{sd}$ are created that are consistent with those of 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 6.1.2 : An *instruction-level* view of a program execution $P=<E,\rightarrow_T,\rightarrow_{sd}>$ is another program execution $P_I=<E_I,\rightarrow_{T_I},\rightarrow_{sd_I}>$ where each computation event in E_I comprises a single instruction and P is a higher-level view of P_I.

■

We assume that any instruction-level view obeys axioms A1 through A4. Axioms A5 and A6 are also obeyed (although the 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 A5 and A6, its corresponding instruction-level view must also obey these axioms.

Finally, Definition 6.1.3 provides a formalization of the notion of a trace based upon our model of program executions. A trace is a view of an execution in which part of the view is forgotten. 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 relations $\rightarrow_T$ and $\rightarrow_{sd}$ are created by removing all members (of the relations) that relate a removed member (of the event set). More formally,

Definition 6.1.3 : Given a set of traceable instructions **I**, a *trace* view of a program execution $P=<E,\rightarrow_T,\rightarrow_{sd}>$ is another program execution $P_t=<E_t,\rightarrow_{T_t},\rightarrow_{sd_t}>$ where given the instruction-access view of P, $P_I=<E_I,\rightarrow_{T_I},\rightarrow_{sd_I}>$ 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. and $\rightarrow_{sd_t}$ is a subset of $\rightarrow_{sd_I}$ such that $\rightarrow_{sd_I} - \rightarrow_{sd_t} = \rightarrow_{sd_{(I-t)}}$ (possibly empty),

4. P is a higher level view of the execution $<E_t \cup E_{(I-t)}, \rightarrow_{T_t} \cup \rightarrow_{T_{(I-t)}}, \rightarrow_{sd_t} \cup \rightarrow_{sd_{(I-t)}}>$.

5.For all events A and B such that A, B $\in E$ and A, B $\subseteq E_t$,
A $\rightarrow_T$ B $\Leftrightarrow \forall$ a$\in$A, b$\in$B (a $\rightarrow_{T_t}$ b $\Leftrightarrow$ a $\rightarrow_{T_I}$ b)
A $\rightarrow_{sd}$ B $\Leftrightarrow \exists$ a$\in$A, b$\in$B (a $\rightarrow_{sd_t}$ b $\Leftrightarrow$ a $\rightarrow_{sd_t}$ b)

■

Condition 1 ensures that the event set for a trace contains only event instances of traceable events. Conditions 2, 3 and 4 ensure that while the trace itself if not an execution, it was derived from some execution for which P is a higher level view. Finally, condition 5 ensures that regardless of how instructions are grouped in the trace and higher level view the $\rightarrow_T$ and $\rightarrow_{sd}$ relations are consistent across the two executions and the trace.

Once again, a trace view of a program execution may not be a valid program execution. A trace view obeys axioms A1 through A3. Since the trace view may not contain all of the program instructions, A4 will not hold however a linear order will remain among the remaining events for each process. Similarly, A5 and A6 may not hold since process control statements and lock and unlock statements may be missing in the trace view. 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.

6.2. Comparing Executions

Research in the area of sequence comparison [28] has been useful in many areas of scientific research (e.q., genetics, biology, etc.). It seems reasonable to view execution comparison as an application of sequence comparison. Traditional sequence comparison methods fail to be applicable for several reasons. Such methods compare sequences of the same length and view sequences at an element level granularity. Well-known methods such as Euclidean distance

$$[\sum_{i=1}^{n}(a_i-b_i)^2]^{1/2}$$

and Hamming distance

$$\sum_{i=1}^{n}|a_i-b_i|$$

simply measure the number of positions in which two sequences differ. While these distance have a linear runtime and a constant space usage, they are unfortunately not useful measures for executions. Consider a program that uses the value of a shared variable to determine the number of times to iterate through a loop and following this loop uses no shared variables. The set of executions for this program would all have identical suffixes following execution of the loop but would vary in the number of times the loop body was executed. A simple comparison of the elements of an execution (instructions) will fail to recognize that the executions are identical upon exit of the loop and compare instructions within the loop to those in the suffix of an execution. Clearly what a comparison algorithm should do at this point is recognize that these executions differ *during* the loop. This requires a comparison algorithm that views the program as more than just a sequence of unrelated instructions.

The Levenshtein distance, a measure of the amount of work necessary to make two sequences identical, provides a better comparison of executions simply because sequences are viewed as a whole entity as opposed to unrelated elements. The **UNIX*** **diff** filter is an example of a Levenshtein distance measuring the difference between two files. Unfortunately, this sort of measure is unacceptable for execution comparisons because of space and time constraints. In general, a Levenshtein distance calculation will have linear space requirements and $O(n^2)$ runtime requirements. Since executions can be quite large, any reasonable comparison algorithm must run in linear time using constant space.

6.2.1. The $\rightarrow_k$ Distance Measure for Parallel Programs

Here, we propose a measure of execution difference based upon the TCP structure of execution and how that structure might differ between two executions. This measure views executions as having structure as opposed to as individual instructions strung together in a sequence. When measuring the difference between executions, we can view the executions as having converging and diverging points in common. The former are

* Unix is a trademark of Bell Laboratories.

TCPs that both executions contain that have the same outcome and the latter are TCPs that both executions contain that have different outcomes. This measure is defined in the form of a very weak execution sequence equivalence $\rightarrow_k$ and simply measures the *TCP distance* between the occurrence of identical TCP events with the same outcome in the executions being measured. For example, if two executions diverge at some point and then converge at the next TCP, their TCP distance is one. If however one of the executions continues on and executes another TCP before reaching a converging point, the TCP distance is two. The measure is that of the maximum number of TCPs executed by either execution during a period of divergence. More formally,

Definition 6.2.1 : Given two program executions $P_1=<E_1,\rightarrow_{T_1},\rightarrow_{sd_1}>$ and $P_2=<E_2,\rightarrow_{T_2},\rightarrow_{sd_2}>$, .ps 12 $P_1 \equiv_{TCP} P_2$ for some k, if

1. for some $S = TCP(E_1) \cap TCP(E_2)$, for all $b \in S$,
$$Outcome(P_1,b) = Outcome(P_2,b)$$

2. and for all $a,b \in S$ such that $a \rightarrow_{T_1} b$ and $a \rightarrow_{T_2} b$,
$$|\{c \mid c \in TCP(E_1) \wedge a \rightarrow_{T_1} c \wedge c \rightarrow_{T_1} b\}| \leq k$$
and
$$|\{c \mid c \in TCP(E_2) \wedge a \rightarrow_{T_2} c \wedge c \rightarrow_{T_2} b\}| \leq k$$

■

Condition 1 ensures that there are some TCPs with identical outcomes. This will be true for any execution taken from the same program and data set because of the inclusion of begin and end statements in the TCP sets. Condition 2 defines the distance measure by ensuring that the number of TCPs that occur between TCPs with the same outcome is less than or equal to some integer k. Naturally, we are interested in the smallest such k for any particular measure.

In [8], we hope to show that this distance measure can be calculated in linear time with constant space usage. This measure is clearly superior to a simple Hamming distance calculation (or Euclidean distance) since the measure will take into account program structure. However the cost is that source code must be available for the execution and this source code must be preprocessed to supply information about program TCPs.

6.3. Trace Migration

In trace migration, a *source* trace, for a program and fixed data set, collected on some execution architecture is used to create a correct *target* (extrapolated) trace for another execution architecture that is in the same family as the source execution architecture [17]. Two execution architectures are considered to be in the same *family* if they are a simple parametric variation of each other (i.e., variations in processor speeds, memory size, number of processors, etc.).

Trace migration relies on a source trace in constructing a target trace and the degree to which the source trace can be re-used distinguishes trace migration from other trace generation techniques. The re-use of a source trace is measured by whether or not source code that was executed while creating the source trace has to be *re-executed* in the creation of the target trace. Clearly, if the entire program is re-executed, from start to finish, in order to create the target trace, then trace migration degenerates to execution driven trace collection such as the method used in SPAE [14]. Thus the technique depends on reusing parts of the trace that do not change during extrapolation, combined with new parts of the trace that result by re-executing certain parts of the program as a result of different activity in the target trace.

Holliday and Ellis show how to solve the trace migration problem and present examples of programs that are difficult to migrate. The major obstacle for trace migration is the effects of program nondeterminism. Specifically, the interleaving of accesses to shared memory in the target trace can lead to a state in which the outcome of program TCPs might differ from that of the source trace. The issue then is whether or not trace change points can be decided without program re-execution. Using examples, Holliday and Ellis pose the question of whether or not there exist classes of programs for which trace migration is a reasonable trace generation method. They characterize just such a class of programs using their notion of *graph-traceability*. Cast into our terminology, a program is graph-traceable if all of the program TCPs can be decided without program re-execution.

The question of whether or not a program is graph-traceable does not lend itself to complexity measures, however it is reasonable to characterize what programs are graph-traceable. In our framework, programs that are strong structural (and data) equivalent are trivial to migrate. These are programs that produces traces that can be migrated using no program state information (other that that which is embodied in the source trace). The reason is that these programs have a fixed TCP outcome for each fixed data set. Hence, all program TCPs are already decided and any interleaving of shared memory references yields the same TCP outcome. Many programming applications fall into this category (e.g., those written using the DOALL construct), and for these programs trace migration can be performed on-the-fly (or in O(1) space) on a source trace in linear time. Unfortunately, Theorem 5.3.3 demonstrates that weak structural programs require program state information for trace migration.

6.4. Taking Advantage of Structure in Traces

Strong and TCP programs are characterized by having a uniform TCP outcome for any fixed data set. Very TCP programs have some structure to their TCP outcomes however not as strict as a TCP equivalent program. Arbitrary equivalent programs have no TCP structure. It is reasonable to conjecture that programs having more TCP structure are likely to be easier to migrate that those with less TCP structure. For this reason, we hope to use the notion of a weak structural equivalent program to further characterize programs that are easy to migrate. Likewise, it is reasonable to conjecture that arbitrary

programs do not lend themselves to trace migration since the opportunity for source trace reuse is likely to be minimal.

We also hope to use the notion of weak structural equivalent programs and our framework to further characterize the difficulty to migrate programs. We believe that this question is a fundamental one that is closely linked to the question of the degree to which a program suffers from execution order distortion.

Finally, we are interested in providing some measure of the distance between individual traces and using this measure to provide a measure of the maximal distance between any two traces in a set of possible traces. We believe that the latter may be the best way to characterize parallel programs, both for trace migration and execution order distortion. Our framework provides a concise treatment of executions in which to explore such measures.

7. Conclusion and Future Directions

We have extended the work of Holiday and Ellis by presenting a taxonomy characterizing the contribution of a program statement (or instruction) to execution order distortion in an execution of that program. Statements (and instructions) are classified as either static (i.e., outcome can be statically determined), runtime (i.e., outcome is fixed after a single execution on a given data set), or dynamic (i.e., outcome varies over subsequent runs using the same data set). Dynamic statements are called *trace change points*. Using this classification, we identify programs as determinate or nondeterminate. Our notion of a determinate program includes some programs that more traditional definitions would consider nondeterminate. This is because we view programs in terms of their resulting executions rather than by the output that they produce. This allows use to consider only the program state information which is embodied in an execution path. Nondeterministic programs with no TCPs cannot be distinguished from determinate programs given this view.

A formalization of program executions based upon Lamport's theory of concurrent systems [20] has been used as a base model for a hierarchical view of program executions. This base model has been shown to encompass both traces and the notion of abstract events. In addition, we believe that it can be formalized to include machine level nondeterminism.

This model of executions serves as the base model for a hierarchical view of program executions based upon the taxonomy of program statements. In this hierarchy, execution equivalences are stratified into four categories: data, strong structural, weak structural and arbitrary. Each category represent a different level of structural information about an execution.

Set membership for a particular execution is viewed as partition membership in this hierarchical framework. Feasible execution theorems have been presented for each level of the hierarchy. While data equivalences lead to partitions in which membership is easily determined, for the other classes in the hierarchy the problem of feasibility for a partition is undecidable. The question of how much state information is needed to determine feasibility for a partition is open.

Finally, we present some applications of the hierarchy. We have shown that multiprocessor event traces have a representation in our framework and that the notion of an *abstract event* is consistent with our framework. Currently there are no measures of execution difference that the authors know about. We have proposed a measure and given motivation for what might constitute a reasonable measure of execution difference. Finally, the problem of determining what classes of parallel programs can reasonable be used in trace migration is open. In our hierarchy, programs that are data or strong structural equivalent programs have sets of executions that can be easily migrated. We have shown that this is not the case for weak structural and arbitrary equivalent programs. It remains to be seem whether or not weak structural programs have some structural characteristics that can be exploited for trace migration.

8. Acknowledgements

The first author has been supported by a grant from the Convex Computer Corporation, who are also providing general support for all of our work on the parallel program tuning environment. We would also like to acknowledge Harini Srinivasan for discussions on the problem of computing TCPs.

References

1. 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*, 1986, pp. 119-127.

2. A. Agarwal, J. Hennessy and M. Horowitz, ''Cache Performance of Operating System and Multiprocessing Workloads'', *ACM Transactions on Computer Systems 6*, 4 (Nov. 1988), pp. 393-431.

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*, St. Charles, IL, Aug 1987, pp. 721-727.

5. 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*, IEEE CS Press, Los Alamitos, CA, May 1990, pp. 270-279.

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

7. H. Davis, S. Goldschmidt and J. Hennessy, ''Multiprocessor Simulation and Tracing using Tango'', in *Proc Int. Conf. on Parallel Processing*, ACM Press, May 1991.

8. Z. K. F. Eckert and G. J. Nutt, ''Optimal Sequence Analysis for Parallel Program Traces'', in preparation, Dept. of Computer Science, University of Colorado, April 1994.

9. Z. K. F. Eckert and G. J. Nutt, ''A Data Flow Formulation of the Trace Change Point Problem'', in preparation, Dept. of Computer Science, University of Colorado, April 1994.

10. 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.

11. P. A. Emrath and D. A. Padua, ''Automatic Detection of Nondeterminacy in Parallel Programs'', in *Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging*, ACM Press, Jan. 1991, pp. 89-99.

12. P. G. Farber, ''Analysis of a Shared Bus Multiprocessor Memory System Using Trace Driven Simulation'', Masters Thesis, University of Colorado, 1991.

13. J. K. Flanagan, B. E. Nelson, J. K. Archibald and K. Grimsrud, ''Incomplete Trace Data and Trace Driven Simulation'', in *Workshop on Modeling and Simulation of Computer and Telecommunications Systems*, San Diego, Jan 1993, pp. 203-209.

14. 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*, Jan. 1993, pp. 95-106.

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

16. J. Hennessy, ''The Accuracy of Trace-Driven Simulation of Multiprocessors'', in *Proc. ACM SIGMetrics Int'l Conf. Measurement and Modeling of Computer Systems*, ACM Press, 1993, pp. 146-157.

17. 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 (Jan. 1992), pp. 97-109.

18. 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*, ACM Press, Toronto, Canada, 1991, pp. 244-253.

19. L. Lamport, ''How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs'', *IEEE Transactions on Computers c-28*, 9 (SEPT 1979), pp. 690-691.

20. L. Lamport, ''On Interprocess Communication Part I: Basic formalism'', *Distributed Computing 1* (1986), pp. 77-85.

21. J. R. Larus, ''Abstract Execution: A Technique for Efficiently Tracing Programs'', *Software -- Practice and Experience 20*, 12 (Dec. 1990), pp. 1241-1258.

22. A. D. Malony, ''Event-Based Performance Perturbation: A Case Study'', in *Ppopp*, ACM, 1991.

23. R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, ''Evaluation Techniques for Storage Hierarchies'', *IBM Systems Journal 9*, 2 (1970), pp. 78-117.

24. R. H. B. Netzer, ''Race Condition Detection for Debugging Shared-Memory Parallel Programs'', in *Doctoral Thesis*, University of Wisconsin-Madison, 1991.

25. G. J. Nutt, ''A Parallel Program Tuning Environment'', in *Proceedings of the 1993 International Conference on Parallel Programming*, St. Charles, Illinois, August 16-20, 1993.

26. A. Podgurski and L. A. Clarke, ''A Formal Model of Program Dependencies and Its Implications for Software Testing, Debugging, and Maintenance'', *IEEE Transactions on Software Engineering 16*, 9 (Sept 1990), pp. 965-979.

27. D. A. Reed and EtAl, ''Scalable Performance Environments for Parallel Systems'', in *Proceedings of the Sixth Distributed Memory Computing Conference*, 1991, pp. 562-569.

28. D. Sankoff and J. B. Kruskal, *Time Warps, String Edits, and Macromoles: The Theory and Practice of Sequence Comparison*, Addison-Wesley Publishing Comapny Inc., Reading, MA, 1983.

29. A. J. Smith, ''Cache Memories'', *ACM Computing Surveys 14*, 3 (SEPT 1982), pp. 473-530.

30. 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*, ACM Press, 1989, pp. 70-78.

31. C. B. Stunkel, W. K. Fuchs and B. Janssens, ''Address Tracing for Parallel Machines'', *IEEE Computer 24*, 1 (Jan. 1991), pp. 31-38.