# Design of a noninterfering debugger for embedded real-time systems

Banda, Venu Prabhakar, Ph.D.

The University of Michigan, 1990

# DESIGN OF A NON-INTERFERING DEBUGGER FOR EMBEDDED REAL-TIME SYSTEMS

by

Venu Prabhakar Banda

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1990

To my parents

# ACKNOWLEDGEMENTS

My deepest gratitude is to my advisor Professor Richard A. Volz for his constant, untiring encouragement and support throughout the course of this work. I am grateful to him for his guidance and for being immensely patient with me. This work would not have been possible without his guidance and support.

I am also grateful to other members of my advisory committee, Professors Chaitanya Baru, Trevor Mudge, Toby Teorey and John Sayler for their timely critiques and guidance.

Finally, I am grateful to my parents, sister and my wife for their confidence and patience.

# TABLE OF CONTENTS

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

## 1.1 Motivation

Lubarsky's Law of Cybernetic Entomology:
There is always one more bug.

A bug is a deviation from the expected behavior. Debugging is the process of detecting and correcting the bug, to exactly match the behavioral specification. Errors or bugs can be introduced at various stages in the software lifecycle. Typically, the software lifecycle starts out with a requirements specification followed by functional decomposition (consisting of logical design and physical design), coding, testing and debugging. Usually one has to go through a number of iterations in the above cycle before arriving at a final product.

The cost of correcting a bug varies depending on the stage at which it was introduced. An error introduced at the design stage is significantly more costly to rectify than one introduced at the implementation stage, largely because there are more stages dependent on the design stage than on the implementation stage. While it is true that more emphasis must be placed on eliminating bugs as early as possible in the life cycle, it does not lessen the importance of providing better tools to eliminate or at least decrease the probability of introducing bugs in the latter phases. Afterall, a bugged program is as useless as any, regardless of the stage at which the error was introduced. Moreover, a bug free design does not in any way guarantee a bug free implementation, because the transformation from design to implementation could be flawed. It is generally true that a significant amount of time is spent in the testing and debugging phases. This is exemplified by the increasing attention being paid to structured programming, type abstraction etc., notions that aid readability,understanding and the writing of bug free code.

Conceptually, non real-time sequential programs are the simplest to debug since bugs in such programs are reproducible during each re-execution of the program. However, techniques used to debug non real-time sequential programs cannot be extended to debug parallel or real-time programs (for reasons which will be discussed in detail in a subsequent section). While there has been considerable research in the area of debugging parallel programs, very little has been accomplished in debugging real-time programs. The real-time environment gives rise to a special class of errors that do not arise in parallel non real-time programs. In this thesis, we will sytematically analyze and classify the various errors that can occur in parallel real-time programs and design a debugger that will reduce the problem of debugging parallel real-time programs to that of non real-time sequential programs for a certain class of errors.

## 1.2 Problems in Debugging Parallel programs

Debugging parallel (concurrent or distributed) programs is much more difficult than debugging sequential programs for several reasons. [GMGK84] gives the following reasons.

- A parallel program has multiple loci of control, thus making it more complex to understand.
- If processes of the program are distributed across processors then identifying the processor in which a process has failed becomes difficult. This point was also mentioned by [LeD86] as *complexity of error localization*.

- If the program is distributed onto processors that are geographically dispersed, then communication delays between processors might hinder access to information necessary for debugging.

- Parallel programs that are distributed tend to be large and generally too complex to perform exhaustive testing (complexity of testing [LeD86]).

- Global Clock:

  Absence of a global sense of time in a distributed system, makes it difficult to sequence events across processors. In a distributed environment events occur simultaneously on different processors. The order in which events occur cannot be easily determined. A program that works correctly one time may fail subsequently if the timing between processors changes [Gor85].

- Monitoring:

  Traditional techniques for detecting errors, by introducing monitoring statements within the program, are inadequate, because they may perturb the behavior of the program. As a consequence, a timing error during a particular execution may not repeat itself after the introduction of the monitoring statements. Alternatively, a program may work fine with the monitoring statements but fail to do so once the monitoring statements are removed.

  The inadequacy of monitoring a program by introducing monitoring statements is usually referred to as the *Heisenbug Uncertainity Principle* and is often stated as follows [LR88]:

  - Measuring Program state, perturbs said state.
  - The degree of perturbation is proportional to fraction of the state that is captured i.e, the greater the volume of information collected, the lower the accuracy.

- Too much Data:

  Debugging through traces and dumps is very messy and produces huge amounts of data [GMGK84].

- Irreproducibility:

  A distributed program is made up of a number of processes co-operating via messages or shared variables or both, to perform some computation. One or more of these processes can be active at any given instant of time.

  Consider the execution of a distributed program on an underlying operating system(we refer to such environments as Normal, as explained in chapter 3). Within the constraints of the semantics of the language, the decision as to which process gets the CPU is made by the underlying operating system scheduler. The point of decision and the outcome of the decision made by the operating system scheduler could, in general, differ across two different executions of the same program. The difference could be because of different system load conditions when the program was run or a number of other reasons outside the control of the program.

  Even in domains where the application program is executed on the bare machine, the program could behave differently across different executions, because of external interrupts. Since external interrupts are triggered by real-world processes, the application program has no control over their point of occurrence ( in time or w.r.t instruction number within the main program).

  In general, regardless of whether the application program is executed on an underlying operating system or not, the sequence of process activations and hence the sequence of instructions executed by the program could differ between different runs. It is possible, therefore, that an error occurring during the first execution may not manifest itself during subsequent executions.

- Taking a snapshot:

  Generally, when a program crashes due to an error, information about all the variables within each of the processes when the crash occurred could be quite helpful in determining the error. Even if the program does not crash, checkpointing the program at regular intervals could prove to be quite time saving (because if re-execution is required it need not be started from the beginning). If all the processes of the target program are being executed on the same processor, then it is possible to halt all of them instantly and take a consistent snapshot of the program ( of course one must make sure that the checkpointing program cannot be interrupted ). However, it is not possible in a loosely coupled distributed system, where more than one processor may be executing at the same time, because the desire to checkpoint on one processor cannot be broadcast simultaneously to all other processors due to network delays (although one could approximate a firing squad solution, wherein all the processors could come to agreement on a future time to checkpoint). The inability to checkpoint all processes at the same time could destroy some critical information about the set of processes.

## 1.3 Debugger Classification

Debuggers can broadly be classified according to the type of application programs they are targeted to, as follows:

- Sequential

  Sequential debuggers are targeted towards applications that involve only a single thread of control. Most debuggers in this environment provide the user with the ability to monitor the control flow and data flow of the program. Commands to monitor the data flow usually consist of the ability to read and modify the variables visible at the current point of control. The more sophisticated debuggers also allow one to peruse through the data state outside the current scope as defined by the programming language. Control commands provide the user with the ability to set breakpoints (conditional and unconditional) in the program. In sequential programs, successive executions of the program result in identical behavior (since the sequence of events remains the same because there is only one thread of control ). Existence of a single thread of control therefore leads to much of the simplicity of debugging in these environments.

- Concurrent Debuggers:

  Concurrent debuggers are targeted towards applications that have more than one process active at a time, all of them executing on a single processor. Each process corresponds to one thread of control. The processor is multiplexed between the various thread. While it is true that at any given instant of time only one thread has the CPU, more than one threads could be active. Complexity in these domains essentially stems from the existence of multiple threads of control. While the programmer is aware of the parallelism between the processes through the high level language constructs, he is not aware of when a particular thread becomes active, since that is abstracted out by the underlying operating system. This inability to think of all possible thread interleavings (something outside the domain of the application) coupled with the natural misconception that each high level construct is atomically executed leads to a semantic gap that in turn increases the complexity of debugging in these domains.

- Distributed Debuggers:

  Distributed debuggers are targeted towards programs that are physically distributed across more than one processor. These domains are not only characterized by multiple threads of control being active at any time but also have multiple threads of control executing at any given time. Conceptually however they are at least as difficult to understand and debug as the concurrent programs.

## 1.4 Real-Time

Real-time systems give rise to a new class of timing errors. In this section we will define the notion of real-time and identify the new different timing error in these environments.

There are numerous definitions of the term Real-Time in the literature. We shall adopt the one given in the Internation Journal for Time-Critical systems, which is as follows: A real-time application is one in which the correctness is defined not only by the value of the output that is computed but also by the time at which it is computed. A correctly computed value that is done so even slightly late is considered as incorrect. In certain applications the timing factor is so important that an incorrect value computed within a certain deadline is more acceptable than a correct computation that misses the deadline. Real-time systems are further classified into hard real-time systems and soft real-time systems. Hard real-time systems are ones where missing a deadline could be catastrophic leading to loss of human life or very costly equipment. On the other hand soft real-time systems are those where missing a deadline is not all that catastrophic.

The processes of real-time software are classified into two categories periodic and sporadic. Periodic processes are those that need to be executed once during each predefined period. Each of these processes is characterized by a 3-tuple $(c_i,p_i,d_i)$, where $c_i$ indicates the worst case execution time of the process, $p_i$ indicates the period of the process and $d_i$ indicates the deadline for completion of the computation within the process relative to the beginning of execution. Also $c_i <= d_i <= p_i$. For periodic processes, usually the deadline is the same as the period. A sporadic process on the other hand is not periodic and could arrive at any time (interrupts e.g, represent sporadic processes). For scheduling purposes they are sometimes modeled as periodic processes, where the period is defined as the minimum inter-arrival time for the sporadic process. Sporadic processes are also characterized by the same 3-tuple as periodic processes.

Since meeting deadlines is so crucial in real-time systems, it is necessary to schedule the processes appropriately to meet the timing constraints. An incorrect schedule could lead to a missed deadline. A deadline that is missed because of an incorrect schedule is called a scheduling error and will be considered in this thesis.

## 1.5 Our Goal

Our goal through out the thesis is to design a method of debugging which reduces the complexity of debugging a parallel program to that of debugging a sequential program. The general framework of our method can be characterized as two phase debugging. The first phase involves monitoring the execution of the program and the second phase consists of replaying the execution under the control of the user. Within this framework, the majority of this thesis will concentrate on the architectural issues involved in providing *non-intrusive* monitoring of real time programs. More specifically the following are some of the issues that this thesis will look into:

- Since monitoring program execution is a must for any debugging strategy, what should one monitor? i.e, is it necessary to monitor every event that occurs during the execution? or are certain events *more* important than others for achieving behavioral reproducibility?

- At what level of granularity should the above monitoring occur? What are the intrinsic lower bounds on debugger intrusiveness as a function of data collection granularity?

- What memory architectures are needed to handle data collection at a particular level of granularity and how do they effect debugger intrusiveness?

- What effect do various advanced architectural features like caching and pipelining have on the ability to perform non-intrusive monitoring?

- What inherent differences in kind are associated with debugging multicomputer systems at varying levels of granularity, within the above two phase methodology?

- What inherent differences in kind are associated with debugging multicomputer systems at varying levels of granularity, within the above two phase methodology?

- What inherent differences in kind are associated with debugging multicomputer systems with varying interconnection topologies, within the two phase methodology?

- How is the replay of the program handled?

# CHAPTER 2

# RELATED WORK

## 2.1 Introduction

Debugging is the process of finding and eliminating bugs from the program. A program in turn is said to have bugs if its execution behavior differs from its specification behavior. A typical debugging session starts out with the user executing the program and noting the difference in behavior from his/her notion of a correct behavior. The user then builds a theory of what might have led to the deviation and corrects his program accordingly. Subsequently, he/she re-executes the program to verfiy if the corrections made do infact result in the expected behavior. The process of debugging can be identified by the following steps:

- Execute the program and observe its behavior.
- If it does not match the programmers notion of a correct behavior then the program has a bug.
- Identify plausible reasons for the deviation in behavior identified above.
- Modify appropriate parts of the program to correct for the deviation.
- Re-execute program to verify if behavior of program is indeed as expected.

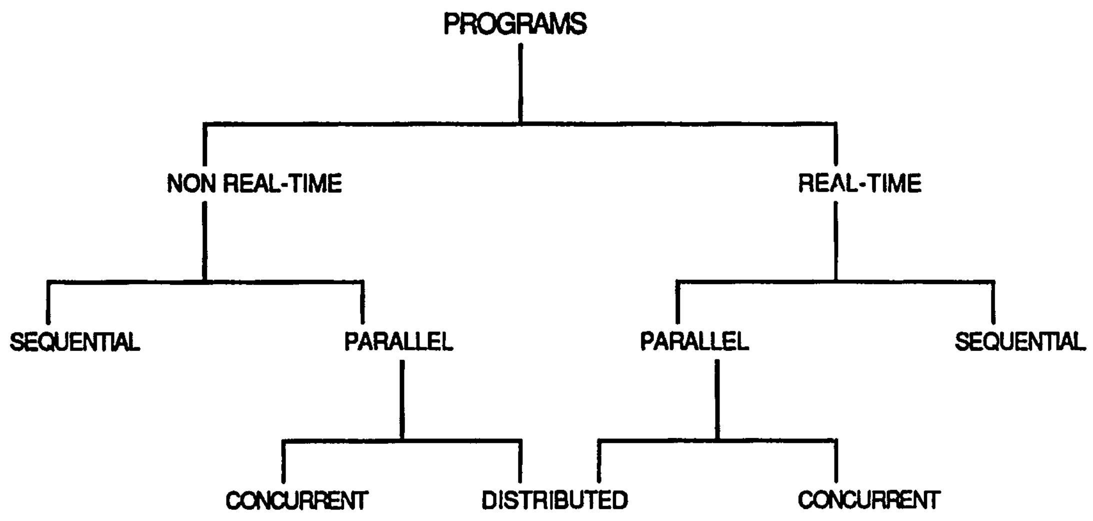


Figure 2.1. Classification of Programs

The above steps are carried out repeatedly until the execution behavior of the program matches the programmer's notion of a correct execution. The above process is sometimes referred to as *Cyclic Debugging*. One of the implicit assumptions in using cyclic debugging, is that the act of observing itself does not perturb the behavior of the program. This however is not true for all types of programs. Figure 2.1 shows the classification of programs according to various parameters real-time, sequential, parallel etc. With the exception of non real-time sequential programs, all other program classifications shown in figure 2.1 suffer from the fact that observing their behavior causes perturbation of said behavior. This, in essence, is the problem with debugging real-time programs or parallel programs.

Before delving into the survey of the overall strategies for debugging parallel/real-time programs, we shall briefly discuss some of the tools and language level constructs available to help the user identify certain errors. We shall categorize the tools as *Static* or *Dynamic* depending on whether they can be used before or during execution of a program.

- Static Tools
  - Compilers automate the detection of compile-time errors, e.g,type mismatch, undeclared variables etc.
  - Static Analyzers automate the detection of errors that are usually not captured by compilers, but can be detected without executing the program. The basic techniques used for static analysis are *control flow analysis* and *data flow analysis.* In control flow analysis, a graph is built to show the allowed flow of control between sections of code. Using this graph model of the program, questions about reachability can be answered by verifying whether or not there is a path between the necessary statements in the corresponding graph. Data flow analysis is also done by modeling the program as a graph except that each node in the graph denotes a statement, the execution of which can cause a variable to be updated. Software errors such as un-initialized variables can be detected by such data flow analysis. Taylor et. al [TO80] used static program analysis for reporting deadlock errors. The problem with this approach is that because of state explosion the method can only be applied to very simple programs.
- Dynamic Tools
  - Exception handlers can be used to provide run-time support for detecting certain classes of errors. An exception is *raised* when a program reaches a specified state. An exception handler is a block of code that is executed whenever a particular exception is raised. Once a handler is executed there are various handler responses depending on the exception model supported by the language: termination, resumption, retry, propagation and transfer of control. Antonelli's thesis [Ant88] gives a detailed description of the various exception models.

- Debuggers help the programmer observe the execution of a program by allowing him/her to inspect the control and data states of the program through conditional breakpointing, single-stepping, viewing traces etc.

In order to facilitate a discussion of the overall strategies for debugging parallel programs, we will categorize the approaches based on some criteria. For the purposes of this thesis, we shall use the term **Phase** to mean an execution of the program. As examples of its usage, *Single-phase Debuggers* would mean debuggers that help the user identify errors after executing the program *once* and *Two-phase Debuggers* would mean debuggers that require the user to execute the program twice (usually, the second execution is called replay). The other criteria on which we shall categorize debugging strategies is *Interference*, i.e, whether or not the method of debugging perturbs the behavior of the program. Nearly all the approaches to debugging require some form of monitoring the execution. If the act of monitoring/observing an execution could potentially change the behavior of the program, then the act is referred to as *interfering*.

Fig 2.2 depicts a taxonomy of the various approaches to debugging parallel programs, categorized according to the number of phases and whether or not the approach is interfering. Further classification is done depending on the approach.

## 2.2 Sequential

In the early days of computing, debuggers provided facilities for inspecting the control and data states of a program. They provided for setting breakpoints, inspecting and changing values of variables, but only at the machine language level. Later, emphasis shifted from debugging at machine level to providing similar facilities to perform debugging at source language level, specially in terms of abstractions of the source language. A significant advance was achieved in the development of EXDAMS,[Bal69], which was an on-line debugging tool for PL-I programs. A pre-processor inserted calls to a run-time monitoring package into the source code. The run-time monitor

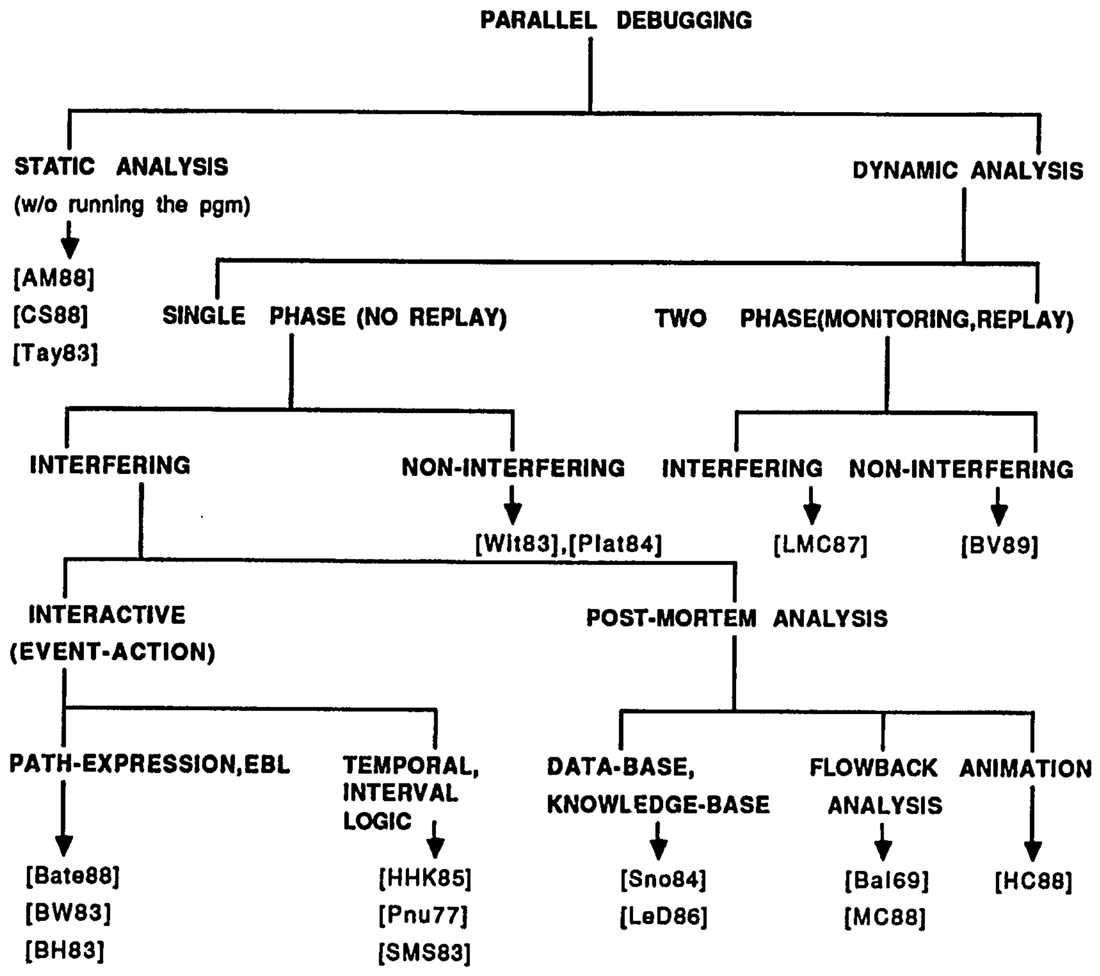


Figure 2.2. A Taxonomy of Debugging Techniques

kept a complete history of execution in a file, enabling reverse execution ! (undoing instructions).

The end of the 1970's saw the advent of language independent interactive debuggers, with the obvious advantage of being useful with a wide variety of source level languages [Joh78]. The debugger allowed the user to intervene at the occurrence of certain events and execute the action corresponding to that event. The various event-action languages differed in the primitives for specifying events and actions. One of the earliest event-action languages was implemented for debugging SNOBOL [Han78]. The user was given the capability to dynamically change the function or action associated with the event. He could also enable or disable certain event actions at run-time. The primitive events were variable assignment, interrupts, function call etc. The user was not given the ability to form more complicated events by combining these primitive events. To overcome this disadvantage [Joh78] developed an event-action language DISPEL for his language independent debugger RAIDE. DISPEL allowed the user to cluster a set of primitives events to define a higher event. These were termed as debugging procedures. The user could place all these procedures in a library for use at a later time. It also had the facility for associating more than one action with an event. All these actions were pushed onto a stack and when the event occured the most recent one was executed.

## 2.3 Parallel Debugging Methods

### 2.3.1 Static Analysis

Static analysis tools provide the information necessary to detect and isolate certain classes of errors, before the program is ever executed. However, static analysis simulates a large number of possible execution sequences and could therefore prove to be very costly in terms of computation time. To overcome such inherent difficulties with static analysis, [AM88] are trying to study the effect of combining static analysis with other debugging and development tools. The additional tools they are investigating for complementing the effectiveness of static analysis are *parallelization assistant* and a *dynamic debugger*. Parallelization Assistant is a developmental tool. We will not describe any details of this tool. The interested reader is referred to [AM88]. They have developed a static analysis tool called START (STatic Anomaly Reporting Tool) which interacts with their dynamic debugger to provide for efficient static analysis.

Static analysis performed by START is based on the creation of Concurrency History Graph (CHG), which was first proposed by Taylor [Tay83], for analyzing Ada programs. Before describing what each node of the CHG contains, we will define a few terms. The *synchronization state* of a task corresponds to the last synchronization operation performed by the task. A *concurrency state* specifies the synchronization state of each task along with the state of all variables used for the synchronization. Each node in the CHG is representative of one or more potential concurrency states of the corresponding program. The edges in the CHG correspond to one or more tasks making a transition from one synchronization state to another.

CHG's can be used to detect two classes of errors, namely, *synchronization errors* and *data-usage errors*. Synchronization errors include deadlock and wait-forever. Data-usage errors include detection of un-initialized variables and parallel data-usage errors like simultaneous update of shared variables by two different processes.

Deadlock and wait-forever both correspond to situations where a node in the CHG has no successors but the node contains active tasks. However, detecting simultaneous update of shared variables requires augmenting the CHG with more information. Recall that each edge in a CHG corresponds to a task advancing from one synchronization state to another. The edges are augmented with a list of shared variables that are read or written by the sequential statements between the two synchronization statements of the task. To detect simultaneous updates it is only necessary to take the intersection of the sets of variables written by any two edges emerging from the same concurrency state. All variables in the intersection can be accessed simultaneously.

Since CHG's tend to be huge even for simple programs, the authors in [AM88] suggest the use of static analysis along with a dynamic debugger. The dynamic trace of a program is used to guide the static analysis. They suggest the user select the two concurrency states bounding the portion of the dynamic trace to be investigated. START then constructs the partial CHG that shows all the possible alternate concurrency states for the portion of the selected trace.

Other research in Static analysis tools for debugging parallel programs includes, PTOOL [CS88]. PTOOL is used to help identify when and how results of a program depend on the scheduling of its component tasks. More specifically it is used to analyze asynchronous parallel loops. When various iterations of a loop are parallelized, and when more than one iteration accesses the same memory location and at least one iteration modifies it, PTOOL reports a schedule dependence for the loop. It does so by constructing a control flow graph, where each node is a task and the edges between tasks indicate dependency (two tasks are dependent if they share the same memory location and at least one of them modifies it).

#### 2.3.1.1 Critique of Static Analysis

We believe static analysis techniques suffer from the fundamental problem of state explosion. The technique is useful only for analyzing short programs, where the concurrency is manageable. Even if the problem of state explosion were solved, the technique is itself deficient in that it cannot detect all classes of errors. For example, a program could crash because of run-time input conditions which cannot be determined by static analysis.

### 2.3.2 Dynamic Analysis

In this section we shall discuss some of the techniques used in dynamic analysis, where the program is analyzed by actually running it (as opposed to static analysis). We shall further sub-divide techniques in dynamic analysis depending on the number of phases involved in debugging and whether the technique is interfering or not. Fig 2.2 shows a taxonomy of the various methods in debugging parallel programs.

Nearly all dynamic debugging schemes have one or more phases over which the debugging activity is conducted (where the phase refers to the number of times the program is executed). In single phase schemes the program is executed only once. Within the single phase scheme further differentiation can be made depending on when the programmer receives control (interactively or only after the execution). All interactive debuggers are based on some variation of the event-action languages. Here the programmer specifies an event occurrence (using path-expressions, temporal logic,interval logic etc) and the corresponding action that the system must take corresponding to that occurrence. The event-action specification is therefore a run-time check to ensure that the behavior of the system matches its specification.

#### 2.3.2.1 Interfering, Single Phase

The various methods under this category have been further subdivided into interactive, post-mortem and animation techniques.

#### 2.3.2.2 Interactive Debugging

In this section we shall review interactive techniques for debugging parallel programs. All these techniques fall under the category of Single-Phase interfering form of debugging. All these methods rely on the event-action paradigm of specifying the behavior of the program.

##### 2.3.2.2.1 Interval-Logic

Assertion monitoring is a means of monitoring the execution behavior of a program through a set of assertions. Some sequential programs provide primitive forms of assertion, namely, conditional breakpoints with the condition being based on the state of a program variable. A distributed program may not be amenable to such a global view of "state of a program". It is desirable therefore, to be able to specify invariant conditions that hold over an interval of program execution.

IDD (Interactive Distributed Debugger) [HHK85], is a debugger that uses interval-logic to specify assertions. Interval-logic is an extension of linear-time temporal logic. Interval-logic is a powerful language for expressing behaviors of programs over intervals of time. The notation is of the form [I]R, which asserts that if the sequence (of execution) contains the interval described by I, then over that interval, the assertion R is true. The interval I is generally constructed by use of the operators $\rightarrow$ and $\leftarrow$ to denote the interval between two endpoints described by their operands. For example the assertion $[P \rightarrow Q] \Box R$ should be read as, "from the next time that P becomes true and until the first time that Q becomes true, the assertion R should remain constantly true". The statement $[P \leftarrow Q] \Box R$, should be read as, "over the interval extending backwards from the next time that Q becomes true to the closest previous time that P becomes true, R remains constantly true".

The difficulty of specifying assertions ( a common complaint against assertion monitoring) is circumvented in IDD by two mechanisms, namely, a graphics interface and incremental assertion generation. Typically, when the program crashes due to an error, the programmer has a conjecture of what possibly went wrong. He/she then writes an assertion for a prefix of the erroneous sequence so that he/she can get control when that sequence repeats. The graphics interface allows the programmer to view the message traffic in the system and allows him/her to focus on messages of interest.

Consider the following interval logic formula used to express the fact that the variable x should be greater than Z at all times during the interval when x is assigned y and y is greater than 16, $[x{=}y \Rightarrow y > 16] \Box x{>}z$. Because of the great expressibility of interval-logic, on-line testing of interval-logic assertions becomes a major hurdle. More clearly, it is not always possible to determine exactly when the interval [I] begins. The authors in [HHK85] give an example $[(\Diamond X \ \& \ \Diamond Y) \rightarrow Z]P$, where the correct left endpoint of the interval can only be known later, depending on when and whether X and Y become true. Difficulty in predicting the future is one of the big problems with using interval-logic for assertion monitoring.

#### 2.3.2.2.2 Generalized Path Expressions

Generalized path expressions is another way of specifying event-action languages for high level debugging [BH83]. A basic path expression is a regular expression consisting of the operators (*) for repetition, (;) for sequencing and (—) for exclusive selection. Path expressions are a means of separating the specification of operations to be performed on shared data objects from the synchronization needed in order to enforce appropriate orderings of operations. [And79] extended path expressions by adding predicates to form predicate path expressions. Here the path expression can contain history variables and predicates. The system automatically defines three history variables for every operation on a data object. The three pre-defined variables are **REQ, ACT, TERM.** The REQ variable defines the number of times the caller has attempted to perform the operation. ACT defines the number of times the caller has started to perform the operation. The TERM variable defines the number of times the caller has terminated the operation. Generalized path expressions are an extension of predicate path expressions for debugging purposes. The grain of execution that can be expressed by a predicate path expression is the entire operation on the data object. However, in a debugging context one might want a finer granularity than operations over data objects, e.g, it might be desirable to operate on a statement-by-statement basis. Generalized path expressions provide such a fine degree of control.

There are two categories of generalized path expressions, namely **FINDPATH** and **CHECKPATH.** If a GPE (Generalized Path Expression) is prefixed by a FINDPATH keyword, then the specified execution sequence is matched against the observed execution sequence and the corresponding path action is executed depending on whether there is a match or not. This is basically used to search for patterns of execution. For example the following GPE,

FINDPATH Beginloop
Whileloop[ACT(postline)=N and ACT(Placelines)=1]

is true whenever the whileloop (which is a previously defined set of statements) is executed at an instant when the procedure Postlines has been called N times and the procedure Placelines has been called once.

If a GPE is preceeded by a CHECKPATH keyword, then the specified execution sequence is matched against the observed execution sequence and the corresponding path action executed depending on whether there is a match or not. This form of GPE is used to enforce a particular execution sequence. For example the GPE,

CHECKPATH loop whileloop[ACT(whileloop) < 6 — P1*

says that the group of statements labeled as whileloop should not be executed 6 or more times before a call to procedure P1.

Each of the associated FINDPATH and CHECKPATH expressions have an associated PATHACTION that specifies the action to be taken in case of a match and a mismatch. Each path expression-path action is called a path rule. A typical debugging session can have many path rules. A facility is provided for disabling certain path rules within a path action.

#### 2.3.2.2.3 Event Definition Languages

Yet another event-action specification language is Bates and Wileden's EDL (Event Definition Language) [BW83], EBBA [Bat88]. Their event-based behavioral abstraction is a high-level debugging approach which treats debugging as a process of creating models of actual behavior from the activity of the system and comparing these models to expected system behavior. The differences in the two models are used to further investigate the erroneous system behavior. The actual model of behavior is constructed by observing the system through *primitive events*. They also provide operations on these events, namely, filtering and clustering. Filtering is used to remove events that are not useful in explaining the behavior under study. Clustering is a way of constructing a higher level of abstraction from other primitive and high level events. EDL only contains constructs for event detection and not for actions (i.e, as a response to a particular sequence like PATHACTIONS).

Smith in [Smi84] describes a message based debugger for distributed systems. He treats each process as a black box and considers interprocess communication as the *only* interesting event. Each event of type message consists of four primitive events namely the message leaving the sender, entering the kernel, leaving the kernel and entering the receiver. The user is given control to modify, create or destroy these interprocess events. The set of tools provided to the user include debugging daemons, which provide event-driven control over events and a transcriber which records all events executed for a process for later examination and replay. The event detection language consists of boolean expressions constructed from predefined fields in the message. When a trigger becomes true an associated daemon process is activated. The daemon process contains debugging commands that can modify, create or suspend interprocess events or redirect events, wait for other daemons or cause waiting daemons to continue.

#### 2.3.2.3 Critique of Interactive Techniques

Apart from the disadvantage that all these techniques are interfering, each of them requires the user to be able to express the behavior of the program in some event -action language, which may not always be that easy. They require the user ([BW83], [Bat88]), to exhaustively describe all interesting events that might occur in a bottom-up fashion, which may not be possible. If the system is complex, it might be difficult to envision all erroneous situations and be able to express them in an event-action language ([BH83],[HHK85]). Also none of these techniques support the cyclic form of debugging.

### 2.3.2.4 Post-Mortem Techniques

Post-mortem techniques fall under the category of single phase debuggers. As the program is executing, information about the dynamic activity of the program is captured and stored into a database. Once the execution is completed or the program is halted, the user can query the database extract information about the program and thereby deduce why the error might have occurred. This scheme is interfering or non-interfering depending on how the trace information is extracted while the program is running. In this section we will survey various post-mortem techniques.

Carol LeDoux [LeD86], has built a debugger called YODA for debugging Ada programs. It is a retrospective approach to debugging, which provides tools for analyzing and observing a program's execution history. The information that is extracted is maintained in a database that the programmer can access to test assertions about the program's behavior. It uses temporal specifications to test assertions against a trace database. Specifically for Ada, YODA captures events associated with task synchronization, task status, variable reference and variable definition. YODA parses an Ada program, generates a symbol table and embeds diagnostic output statements into the source program. When the annotated program is compiled and executed, the diagnostic statements invoke a monitor to capture the relevant trace data. The main contribution of YODA is that it supports time-related queries on a program's behavior and also knowledge based debugging. To support such time-related queries, the historical database is maintained as a collection of time-dependent facts. Associated with each fact is a *time-stamp*, indicating the time at which the corresponding fact was true. The knowledge-based aspect is implemented by having a set of diagnostic rules for predefined error classes (Lost update, Deadlock, Wait forever etc.) which guide the user in detecting these errors. Although the database is maintained as a collection of primitive events, YODA supports queries over abstractions of events.

Garcia-Molina et.al in [GMGK84] describe a trace-based mechanism for debugging distributed systems. Processes are distributed across a set of nodes. A particular node is designated as the master node and runs the master debugger. The master node does not have any application processes running on it. Each node maintains a trace-file containing the activity of processes on that node. Once the program is executed, the programmer queries the trace-files from the master node.

They propose that each process maintain a complete trace history of its execution on a trace file. For each event, information about the type of the event, process of origin, time-stamp and other event-related data is stored in the trace file. The burden of storing information in the file is shared between the user and the system. The system automatically stores an entry for each pre-defined important event. Some of the predefined important events are, process-start, process-end, message receive, message send, resource lock, resource unlock, enter critical region etc (for a more detailed listing, see the reference above). In a distributed system, the physical clocks at the various sites are bound to go out of synchronization. Time adjustment is periodically made and the correction is itself recorded as a time adjust event. Apart from the pre-defined important events, the programmer is also allowed to insert events into the trace fiie that he/she thinks is important, by explicitly introducing write statements into the source program.

Once the trace data at each of the individual nodes have been collected, the programmer treats the collection of trace files as a distributed database and uses a relational query language to extract information from the database. To quote a sample query as given in [GMGK84], if a programmer wishes to know what messages were sent by process 352 after 10:00AM, he/she could issue the following SQL query:

```
SELECT   event-type, time-stamp, transaction-id, event-dependent-data
FROM     R
WHERE    (event-type='Message send' or
         event-type='message receive')
         and process-id=352
         and time-stamp > 10:00AM
```

### 2.3.2.5 Critique of Post-Mortem Techniques

All post-mortem techniques suffer from the fact that information gathered during the monitoring phase may not be adequate to answer all queries that a user might pose during post-mortem analysis. Each approach has a different set of *important events* (events about which information is gathered during phase I). As long as there is no formal basis for the notion of an important event, the monitoring phase will remain ad hoc. Also, since the technique does not support replay, it does not support cyclic debugging (which we think is the most natural form of debugging).

### 2.3.2.6 Animation Debugging

Animation debugging is a relatively new research direction that is being used to debug **highly parallel** programs. The large number of processes in highly parallel programs which result in an explosion in state spaces, has led researchers to believe that traditional state-based, break/examine techniques while successful in the sequential domain are not suitable for debugging parallel programs. Instead they believe that these programs are best understood in terms of the logical patterns of inter-process communication. Furthermore, for applications running on non-shared memory architectures, this communication is often very **structured**.

Belvedere [HC88] is a traced based post-mortem debugger that relies on animation for debugging highly parallel programs running on a non-shared memory architecture. In Belvedere, the programmer is allowed to specify abstract events using the notion of constrained expressions (Bates and Wileden). The abstract events are then automatically animated within the debugger. In a parallel programming environment, a number of processes could be participating in different events. Animating all these events will result in superimposition of the animations which in turn will be difficult to interpret. Belvedere uses *perspectives* that restrict the displayed behavior according to certain criteria. More specifically the programmer can use the process perspective, where the displayed behavior is restricted to the view seen by a single process. They also support a consensus perspective which animates high-level events in a sequence consistent with that seen by all participating processes.

The authors in [HC88] give a number of examples to show that pattern oriented debugging helps in finding three types of errors, namely, 1) sequencing errors, communication patterns occurring in the wrong sequence 2) Missing communication errors and 3) extraneous communication errors, in which expected communication patterns occurred along with additional unintended communications.

Other related work in pattern-oriented debugging is VOYEUR [SBN88], which supports a hierarchy of views ranging from fully abstract views showing images of a computational domain to language specific views showing the state of program variables.

#### 2.3.2.7 Interfering, Two Phase

Under this category, we further subdivide the approaches into Replay-based and Reversible execution systems. Replay based approaches force the replay to start from the beginning of the program while reversible execution systems allow for rollback to a previous state that is not necessarily the start-state of a program.

#### 2.3.2.8 Replay Based Debugging

Debugging schemes that support replay can be classified as two phase systems. The first phase is used for monitoring the behavior of the program i.e, gathering information about *important* events and event sequences. The second phase is used to replay the execution of the program. They support the traditional cyclic form of debugging employed in sequential programs. The various schemes within this framework differ in the type of events gathered during phase I and the strategy employed for replaying the execution.

LeBlanc and Mellor-Crummey [LMC87], describe their debugger, termed *instant replay*. *Instant Replay* is a debugger used for repeatable execution of highly parallel programs in tightly coupled systems. During the first phase, it saves the relative order of *important events*, as opposed to the data associated with these events [Smi84], [LR85], [CW82]. In their approach all interactions between processes are modeled as operations over shared objects. Consequently, *Instant Replay* designates operations over shared objects as the only significant events and monitors the sequence of these events in the first phase of the execution. The system maintains the current version number of each object and the number of readers for each version of each object. To support version-monitoring, the program requires that the application or system have an underlying protocol for access to shared variables. Each time a process is created, a blank history tape is allocated to it, in which the version numbers of every shared variable access is recorded. Each process that reads or writes into a shared variable must do so through a appropriate entry and exit procedures that keep count of the version number of the shared object and log this information in the appropriate history tape. In the replay phase, the history tapes are scanned and the execution is steered so that each process sees the same versions of the shared objects as it did in phase I, thus achieving an identical computational behavior.

Flowback analysis was first proposed by Balzer [Bal69] as a method of providing causal relationships between events in a program's execution without repeated executions of the program during debugging. It is different from the database approach described above, in that it uses semantic analysis of the program, such as dataflow analysis, inter-procedural analysis etc, to extract the causal relationship between events. While Balzer's flowback analysis was described for sequential programs, Miller et. al [MC88], describe a method called incremental tracing that makes flowback analysis for parallel programs, practical by generating only a small number of traces during execution. Miller et. al, use semantic analysis to divide the program into segments of code called *emulation blocks* (e-blocks). Each e-block starts with code to generate a pre-log and ends with code to generate post-log. The pre-log contains the values of variables that are read-accessed in the e-block and the post-log contains values of variables that are write-accessed in the e-block. The time interval between pre-log and post-log is called the *log interval.* Pre-logs and post-logs are maintained for every e-block in the program. To reproduce the program, during the debugging phase, they use the same code corresponding to the log interval and the same input as originally fed to the program which is in the corresponding pre-log. The post-logs allow for restoration of program state to previous points of program execution. More clearly, restoring a program state to the end of the i-th e-block, is accomplished by using all the post-logs starting from post-log(1) corresponding to the first e-block to post-log(i) corresponding to the i-th e-block.

Detection of race conditions in parallel programs is accomplished through the use of *parallel program dependence graphs*(PPD graph). A PPD graph consists of a synchronization node and two types of edges, a *synchronization edge* and a *internal edge*. A synchronization edge is an edge connecting synchronization nodes. An internal edge represents zero or more local events (local to a process) between two synchronization nodes (both of which belong to the same process). Ordering events belonging to different processes reduces to ordering the internal edges to which those events belong. If two edges cannot be ordered, then they are called simultaneous edges. Once simultaneous edges are identified, they use semantic information (post and pre logs) to detect potential race conditions.

Tai and Obaid in [TO86], have considered the problem of reproducible testing of Ada programs. They model the execution of a concurrent program as a sequence of P operations on shared semaphores. They call this sequence as a P-sequence. Each element in the P-sequence corresponds to a P-operation by a specific process. Thus a P-sequence is a total order of all synchronization operations that occur in the program. They point out that P-sequences can be constructed by the programmer or can be reproduced during replay. Replay consists of having a monitor scan this P-sequence and reproduce the same sequence of P-operations during replay.

Chiu's [Chi84], describes a technique for replaying program execution in an atomic transaction system. It involves checkpointing each version of every atomic object along with a time-stamp for each atomic action during execution. The debugger traverses the action tree, corresponding to the nested atomic actions in the execution of a program and allows the programmer to view the action sequences and also the states of the atomic objects before and after each atomic update. In an atomic action system, checkpointing is performed as part of the support for recovery of actions and, therefore, there is not much additional effort (in terms of checkpointing) required by the debugger. However, the technique is not as easily extendable to systems where the computation does not fit the nested atomic action paradigm.

#### 2.3.2.9 Reversible Execution Systems

Feldman and Brown have developed a debugging system called IGOR. Some of the facilities provided by this system are reversible execution, selective searching of execution history, and substitution of data and executable parts of the program. One of the main deficiencies of the system is that it can handle only sequential programs.

The general philosophy of debugging espoused by IGOR is to execute the program until (Normal or abnormal) termination and then subsequently restore the state of the program to an interim point (between start of execution and termination) and replay the program (for a greater detailed scrutiny of the program behavior). IGOR performs incremental checkpointing and uses the virtual memory hardware to checkpoint the program. The virtual memory hardware maintains a "dirty bit" for each page in memory, that is set whenever a page is modified. IGOR then saves a copy of every page that has been changed since the last check. The information as to which pages have been modified is obtained through a UNIX system call, *pagemod*, which returns the list of pages that were modified since the last call to *pagemod*. The frequency of checkpointing is set through another UNIX system call,*ualarm*, which interrupts the program after a specified interval of time. The information stored at a checkpoint consists of all the modified pages and values of on-chip registers and program counter. The replay phase in IGOR is trivial because IGOR is targetted towards single thread programs.

In conclusion the following are some of the problems with IGOR's approach

- The method of checkpointing, though interesting, is intrusive and results in considerable overhead (the authors report overheads ranging from 70 to nearly 400 percent). Such overheads are difficult to justify in a real time setting.
- Generalizing the approach to multiple threads is non-trivial. In IGORS single thread model, the sequence of instructions executed between two checkpoints is fixed, thereby making replay simple. Replay in multiple-thread programs is considerably more difficult.

Pan and Linton in [PL88] describe their tool called *Recap* which supports reversible execution of parallel processes that communicate through shared memory. During execution, recap logs the results of system calls, shared memory reads, as well as times that asynchronous events (signals) occur. Recap allows for checkpointing on each process and also a group of processes at a time. It uses the fork feature provided by the operating system to checkpoint a process. To create a checkpoint, a process simply forks a copy of itself. The parent process continues while the child is suspended until it is needed for replay. Forking a process, automatically saves the processes register and address spaces. To reverse execution, the suspended checkpoint process is continued. Input to the checkpoint process, e.g, shared memory reads etc, are obtained from the event log for the checkpoint process. During replay, system calls are not made but affected variables are set using the log.

#### 2.3.2.10 Critique of Interfering-Two phase Techniques

While replay-based mechanisms have the advantage of supporting cyclic debugging, all the approaches are interfering. In other words, in order to support replay, they monitor the program intrusively. This offsets some of the advantages of cyclic debugging, in that, the replay phase reproduces the behavior of the (Program + Monitoring statements) and not just the original program.

Apart from the issue of interference, with the exception of [PL88], none of the approaches address the issue of replay of programs that communicate by sharing memory. LeBlanc et. al in [LMC87], consider parallel programs sharing memory, but there all access to shared memory occurs via a lock. There is always the possibility that the programmer forgets to use a lock in which case their approach is inadequate. Pan et. al [PL88] do address this issue of unrestricted access to shared memory, but their approach is intrusive.

#### 2.3.2.11 Non-Interfering, Single Phase

One of the first full scale non-interfering debuggers, the Bell system 1-A [Wit83], was developed by Bell laboratories, for their electronic switching system application.

The application was so time critical that Bell decided to build the processor with special hardware to ease debugging during program development. Non-interference is achieved through circuit duplication. Actually, the system 1-A is unique in that it provides debugging capability at varying degrees of interference (interfering, moderately interfering and non-interfering). The hardware monitoring is achieved by a programmable device called UTC (Utility Test Console). The UTC consists of a circuitry to detect the occurrence of an address or data or any arbitrary bit pattern with the data. Each of these low level events can trigger an action which could be to start or stop a trace or interrupt the computer. UTC also provides the ability to combine these events using the logical connectives *AND, OR* and *NOT*. Moderately interfering monitoring is achieved through a software monitor resident in the operating system. The ESS(Electronic Switching System) resident monitor is invoked by an interrupt from the UTC. Not all events are visible at the UTC level. The ESSR(ESS Resident) monitor allows the construction of more complicated event sequences than the UTC. All events visible at the UTC are visible at the ESSR and all events visible to the ESSR are also visible to the mini-computer, which handles monitoring at the interfering level. To summarize, all three levels of the monitor implement some form of the event-action language. The difference lies in their ability to cluster primitive events to view more abstract events.

The system 1-A was a pioneering effort and proved to be very useful for their ESS application. The main drawback however, is that one has to know what to look for before execution i.e, one has to prepare the debugging session before executing the program.

The other significant effort at providing non-interference debugging was by Bernhard Plattner [Pla84],[Pla81]. Plattner also achieves non-interference by monitoring program execution through explicit monitoring hardware that basically listens to the bus between the CPU and main memory. The idea of bus listener was pioneered by

Fryer [Fry73] who used it as an aid in program development for real-time applications on dedicated computers. In Plattner's system, the monitoring hardware collects information off the bus and stores into a phantom memory. A second processor then analyzes the phantom memory searching for event sequences which are specified by the programmer via a predicate action language. The programmer is given control as soon as any of the predicates evaluate to false. This scheme also suffers from the same problem as system 1-A in that one has to know of all the correct event sequences and be able to express them in the predicate language before the program is executed. Also, once the user is given control, further events, in terms of point of occurrence and relative order, may be different from what might have happened if the user was not given control.

Among the more recent efforts at providing non-interference debugging is the work by Rubin et. al in [RRZ88]. Their philosophy is similar to [Pla84] in that they provide for non-intrusive monitoring and interactive debugging. Non-intrusion is achieved through the use of a second processor, that shadows the main one. Events are filtered and clustered through special filtering stages and only the *interesting* events are displayed for the user to view. The animation and graphical view presentation software runs on the monitoring machine, thereby providing non-interference monitoring. One of the drawbacks of the system, as acknowledged by the authors, is that the shadow processor is an identical copy of the main one and is not used to its fullest potential. In other words the monitoring hardware is as costly as the computational hardware.

## 2.4 Summary Critique of various Approaches

In this section, we will critique the various approaches taken to debug parallel programs. The criteria for criticism will range from inherent power of the approach (*can it detect all types of errors?*) to efficiency and ease of use.

First some general comments. We have found that with the exception of the Bell system 1-A, [Wit83], none of the other approaches provides for a truly non-interfering debugger. More clearly, all the approaches in the literature describe techniques in which the debugger could potentially perturb the behavior of the program. Theoretically at least, there could always exist a time critical bug that none of these approaches can track. What is therefore required is a fundamental study of the issues in non-interference monitoring, as pointed out by [LeD86]. This thesis is precisely aimed at such a study. Also few of the approaches address the issue of debugging programs that execute for prolonged periods of time.

Most of the techniques fall under one of the following categories: Single-phase, Two-phase and Other. We will critique each of them separately.

### 2.4.1 Single Phase techniques

These are techniques that require the program to be executed only once. They can be further categorized into *event-action* and *Post-Mortem* techniques.

The problem with event-action techniques like [HHK85], [BH83], [BW83], [Bat88] (Bate's approach is not strictly an event-action paradigm because one can only specify events and not corresponding actions), are the following

- They require the user to exhaustively describe all interesting events that might occur during execution [Bat88], which begs the question *What if the user cannot do so?*.

- In many situations the system could be so complex that it might be difficult to envision all erroneous situations [BH83]. The timing properties could be very complex and the process of writing the specifications in interval-logic [HHK85] could itself be a non-trivial and erroneous process.

- None of these approaches support cyclic debugging.

The problems with Post-Mortem techniques like [LeD86], [GMGK84] etc are as follows

- Approaches under this category depend on monitoring for *important events* during the first phase. The notion of *important events* is informal and based on intuition.
- Because of this informal approach to monitoring, there is always the possibility of *During post-mortem analysis of the execution, what if the programmer wants a piece of information that was not captured during the monitoring phase?.*
- This technique is not suitable for debugging programs that run for prolonged periods of time. More clearly, they make an implicit assumption that the database is large enough to store all the information about the execution of the program.
- Does not support cyclic debugging.

### 2.4.2 Two-Phase Techniques

These are techniques that support cyclic debugging either in the form of being able to replay the whole program or parts of it.

The problems with *Instant Replay* [LMC87] are the following:

- The whole approach is dependent on the fact that processes accessing shared memory do so by using locks. *What if an erroneous program forgets to use locks?.* Instant Replay cannot maintain version numbers in that case.
- In Instant Replay, each process logs the version numbers of shared variables that it accesses onto a history tape. The technique does not address the issue of *What if there is not enough memory on tape to hold the version number information, which could happen for a program that runs for a long time?*

- Since every shared variable access has to go through a lock that maintains version numbers and writes on a history tape, the efficiency can be seriously affected.

Pan and Linton's [PL88], Recap supports reversible execution of parallel programs. The problems with it are as follows:

- Process of checkpointing, using forks, is an interesting idea but could be quite time consuming depending on the implementation of the fork system call.
- It interferes with the execution of the target program.

### 2.4.3 Other Techniques

Significant among the other techniques is the work by Witschorik et. al [Wit83] in developing the debugging monitor for the Bell system 1-A. While their approach provides for non-intrusive monitoring of the program, they do not provide an ability to replay the program. As a direct consequence their approach suffers from the same problem as the other single phase techniques, namely, it requires a very careful planning of the monitoring phase (in terms of what to monitor). Approach to the monitoring phase is informal, in the sense that there is no proof that the collected information is in any way sufficient for the subsequent post-mortem.

## 2.5 Summary

We believe that cyclic debugging is essential to debugging parallel programs. This arose out of the following observations:

- Most parallel programs are difficult to understand. The interaction between the processes of the program could be complex. As a result, specifying the behavior in a bottom-up fashion, as proposed by [Bat88], may not always be possible. This inability to specify the complete behavior of a program, prevents the use of single phase debugging methods.

- Specifying complete assertions about the program in interval-logic or as generalized path expressions is as difficult as writing the program.
- Gathering information about the program for later perusal is an adhoc technique without a formal basis.

The literature in debugging is lacking in a complete strategy for debugging parallel programs that supports cyclic debugging. More clearly the following features are desirable:

- A non-intrusive monitoring scheme.
- A more complete and formal approach to monitoring i.e, identifying and proving the sufficiency of *important events* with respect to the subsequent phase.
- Both of the above must be accomplished within the practical constraint of limited memory to hold the information collected during the monitoring phase.
- Again, the literature is surprisingly scarce when it comes to debugging in real-time (i.e, systems where it is important not only to compute the right value but to do so within strict time constraints).

This thesis is a thorough study of all the above issues.

# CHAPTER 3

# DEBUGGING PHILOSOPHY

## 3.1 Goals

The intent of this thesis is to study the various aspects of debugging parallel real-time programs. Specifically, we are interested in providing the ability of cyclic debugging i.e, the ability to execute the program over and over again until the user locates the error. Such a facility, though available in most non real-time sequential programs, is not yet possible for parallel real-time programs, largely because it is difficult to reproduce a parallel execution. Cyclic debugging involves the activities of *monitoring* and subsequent *replay*, performed as two distinct phases. Replaying an identical execution of the program requires, at the very least, that inputs to the program be reproduced identically. While this ensures that the local behavior of the process is reproduced, it does not guarantee the reproducibility of that part of the behavior that is determined by the interactions with other co-operating processes executing in parallel. In short, replaying a parallel program is non-trivial and requires a powerful and comprehensive monitoring phase. Aside from the issue of what to monitor, there is the issue of interference, i.e, the activity of monitoring a program, perturbs the very program whose behavior it is purportedly monitoring. This raises the issue of how useful the information collected by such a monitoring scheme is in indicating the behavior of the program without the monitoring activity. The faithfulness of a replay to the original execution of the program is directly determined by the degree of interference introduced by the monitoring activity. The lesser the interference, the more reliable is the monitoring data which consequently leads to a more faithful replay. Since the replay should be as faithful as possible to the original execution, a significant part of this thesis will be devoted to studying the issues of providing non-interference monitoring of the program.

In short, we shall attempt to provide answers to the following questions:

- What characterizes the *behavior* of a parallel program, a real-time program?
- What aspect of program execution should be monitored?
- What additional hardware, if any, is necessary to monitor the above aspects of an execution ?
- Is non-intrusive monitoring really possible?
- How do architectural features like pipelining, caching etc. affect the process of monitoring?
- Does the replay guarantee an identical execution sequence?, Does it guarantee a behaviorally identical execution sequence?

## 3.2 Operating Environment

The debugger we are designing is basically targeted towards embedded real-time systems. These systems are dedicated to a single user and a single application. Embedded systems are generally used in time critical applications where the over-riding concern is to meet the deadlines dictated by the application. These applications are characterized by a number of tasks, each of which handles an aspect of the application. Some of these tasks, for example, could be involved in monitoring or controlling a sensor that interfaces to the external world. As an example, a sensor could be monitoring the temperature within a boiler and a task within the application could be dedicated to querying the sensor periodically to get the temperature within the boiler. Since the task is monitoring a real-world parameter, the times at which a task is scheduled (i.e, its period) is dictated by the rate at which the real-world parameter changes. In a typical application, there are a number of real-world parameters and associated tasks monitoring one or more of these parameters. One of the concerns of the embedded system is to schedule the tasks so as to meet all the deadlines associated with each of the tasks. Failure to meet any of the deadlines could range from acceptable to catastrophic, in the sense that it might involve loss of human life or of very costly equipment. This is one distinguishing feature between embedded systems and other less time-critical systems, where failure to meet deadline may not be very severe.

Because of the time criticality, the resource management overhead in embedded systems is kept at a bare minimum. Many of the facilities provided by the operating system in a time shared system, for example, are eliminated in an embedded system in favor of providing a faster response time to the tasks and more importantly greater predictability. In fact certain embedded systems completely eliminate the operating system layer and have the application tasks run on the bare machine with the application tasks managing the scheduling among themselves. Some of the typical characteristics of these systems are as follows:

- All processes(or tasks) of the application program are always resident within the main memory. This is done to eliminate the unpredictability of the overhead involved in paging code or data from the secondary memory.

- All tasks are statically allocated, again for reasons of predictability.

## 3.3 Variations In Program Behavior

In this section, we shall study the various reasons that might lead to non-reproducibility of program behavior.

A typical parallel program consists of many co-operating parallel processes (units of execution). Each process performs a part of the computation and shares its partial results with other processes which in turn use these results to compute further partial results. This activity of computing and sharing either terminates when all required results are generated or goes on forever, depending on the application. The rate at which a process generates results, depends on how frequently it is scheduled to get the CPU. The more CPU time that a process receives, the faster is the rate at which it computes partial results (assuming of course, that the process is not wasting processor time in a wasteful loop). Usually, the scheduler decides which process to schedule next, depending on the set of processes active at that time based on criteria defined by the scheduling policy (e.g, priority, deadlines etc.). The outcome of a scheduling decision, could be different across successive executions for any of the following reasons:

- The unpredictable occurrence of aperiodic events could alter the criticality of other periodic processes and alter the outcome of the scheduling decision.

- The starting conditions of the program need not necessarily be the same over repeated executions. For example, the absolute value of clock when the program is begun will definitely be different for the various executions. If the program logic depends on the absolute value of time, then the behavior between successive executions may not be the same.

- The same set of processes may not be active, at a given scheduling point. This is because of the occurrence of asynchronous events like external interrupts.

The fundamental reason for non-reproducibility of behavior, inspite of identical input conditions, is the unpredictability in the occurrence of aperiodic events and the general inability to reproduce asynchronous interrupt timing. This results in a variation in process schedules across successive executions.

A variation in process schedules, across executions, will result in increasing or decreasing the rate at which a process generates results. If the communication between co-operating processes is not synchronized, then a process may generate results so fast that it overwrites its own partial results before other processes can consume them. Alternatively, the rate of a process could decrease so much that it may not be able to produce results fast enough for other processes to consume. As a result the other other processes could consume stale results.

## 3.4 Error Classification

In this section, we will discuss the various types of errors that can occur in parallel real-time programs.

- Algorithmic Errors: Error arising out of incorrect coding of the algorithm, also known commonly as logic errors. These errors satisfy the property that they are repeatable i.e, they surface during each re-execution of the program under the same input conditions.

- Timing Errors: We define this as a situation where the behavior of the program is determined by the relative rates of computation by its constituent processes i.e, the program produces different output for the same input, during successive executions. These errors are usually characteristic of parallel programs. The following are some of situations during which they arise:

  - Improper Protection Of shared Variables:

    Consider a program consisting of two processes which share a variable, say

X. Since X is shared, any writes to it must be protected within a mutual exclusion region to satisfy the serializability criteria for correctness of parallel processes. Suppose the programmer, while coding these processes, forgets to protect write accesses to the shared variable X. As a direct result of this, both processes could simultaneously write into it giving rise to unpredictable and potentially non-repeatable behavior. It is called a timing error because the behavior depends on who writes last into X, which in turn depends on the relative progress of the individual processes, which may vary due to external interrupts.

  - Asynchronous Events: Interrupts are used by the external world to communicate with real-time processes that control them. The interrupt service routine that services the corresponding interrupt could share registers and memory locations with the thread that it interrupts. Normally, the ISR saves the registers of the interrupted thread of control before proceeding with its own computation. However, if this is not done, the state of the interrupted thread could be modified. The effect of the modification could be different depending on when the interrupt occurs. Since the occurrence of the interrupt is asynchronous, it is possible to see differing behaviors during successive executions.

- Scheduling Errors: These errors arise in real-time programs, where, it is not only important for an object to have the right value but also to receive that value at or within a certain time. In fact in some situations timing is more important than the value i.e, computing the right value after the deadline is useless. Missing a deadline is therefore perceived as an error. These errors are also not reproducible but the reasons for their irreproducibility is not just limited to relative process speeds. The following are some of the situations during which a scheduling error may occur:

  - Incorrect Scheduling: Usually in time-critical programs where meeting the deadlines is of utmost importance, the program manages the scheduling of control between its processes. A wrong task schedule, for example, could result in a deadline being missed. Even though the program explicitly manages the scheduling, the exact schedule need not be the same during each re-execution because of asynchronous events like interrupts. As a result, the deadline could be missed sometimes and not at others. We shall label this type of error as *class I* scheduling error.
  - Unpredictable Timing: Sometimes, however, a deadline may be missed even though the schedule was correct. Each periodic real-time process can be characterized as a tuple ($c_i$, $p_i$), where $c_i$ is the estimated execution time and $p_i$ is the period. A deadline could be missed because of an incorrect estimation of $c_i$. An incorrect estimation of $c_i$ would lead to an incorrect instruction-level schedule which in turn could lead to missing a deadline. In conclusion, a deadline may be missed, in spite of a correct task schedule, because an instruction or a set of instructions took more time than anticipated. We shall label this as *class II* scheduling error.

- Overrun Errors: We define these as situations where the behavior of an object depends on where it is allocated. For example, if the compiler does not perform bounds checking, then an erroneous program could run off an array boundary and overwrite the object allocated next to it. The object allocated next to the array element could be different during different executions, thereby resulting in different behaviors. These errors are also potentially irreproducible.

- Hardware Timing Error: All the errors discussed above dealt with unpredictability in software scheduling. A deadline could also be missed because of an unpredictability in the behavior of the hardware components. For example, an instruction that took t units of time during one execution could take more than t units of time. The increased time could be due to an increased instruction fetch time (because of a bus retry attempt ) or increased execution fetch time (because of increased operand fetch time). These errors are also non reproducible. We will however, not address this error in our thesis, i.e, we will assume that the behavior of the underlying hardware does not change across successive executions.

As far as this thesis is concerned, we shall deal with only timing, scheduling and overrun errors.

## 3.5 Our Debugging Philosophy

In keeping with our overall goal of providing cyclic debugging of parallel programs, we chose the two-phase method of debugging. The first phase is dedicated to monitoring the execution of the program and the second phase is used to replay a *behaviorally equivalent* execution, using the information gathered during the monitoring phase. Architectural enhancements are suggested for performing the monitoring phase in a totally non-intrusive manner. We also describe a novel memory scheme with the ability to perform non-intrusive checkpointing. Some of the issues we will be studying in this framework are:

- What is the replay strategy?
- What should one monitor, to support the replay strategy?
- At what level of abstraction should monitoring take place?
- Are there any events that are more *important* than others, as far as reproducibility of behavior is concerned?
- If such a class of *important* events exists, can these events be monitored non-intrusively? If not, what additional hardware support is necessary?

- How do various advanced architectural features effect the monitoring activity, in terms of maintaining the condition of non-invasiveness?

- How do multiple-processors effect all the above issues?

- What strategies are needed to avoid storing event traces from the start of the program?

- What sort of strategy is needed to support debugging long-term errors i.e, errors that surface long after the program is started? How does such a strategy mesh with non-intrusiveness criteria?

## 3.6 Interference: A Definition

Since the main idea of this thesis is to provide for cyclic debugging of parallel programs with a non-interfering monitoring phase, we shall define what we mean by the term *non-interference.*

A machine instruction, i, executed during a particular run of the target program P, is said to be *interfering* with the execution of P, if and only if any of the following are satisfied:-

- i is a temporary instruction (introduced solely for debugging or testing purposes) within the target program and will be removed before the final version of the program is released.

- i is a part of the debugger and is executed on the same processor as the genuine instructions (those contributing to the computation) of the target program.

- i is a part of the debugger and is not executed on the same processor as the genuine instructions of the target program, but i consumes a non-zero number of bus cycles from the target processor.

The bus-cycle condition emphasises the fact that having an auxiliary processor on which the debugger runs does not in itself guarantee non-intrusiveness. More specifically, if the auxiliary processor shares the same memory as the target processor, then instructions executed on it will steal some cycles from the target processor to access the memory and will therefore contribute to the interference.

It is important to note that according to the above definition, an instruction i left permanently within the target program is not considered as interfering. The activity of monitoring is called non-interfering if during the course of execution, no interfering instructions are executed.

## 3.7 Monitoring issues: General

In this section we shall discuss some of the tradeoffs involved in performing the activity of monitoring in hardware versus doing it in software or some combination of hardware and software. Before proceeding further, it is worth emphasizing that choosing to perform non-interfering monitoring does not preclude software monitoring.

- Purely Software Means:

  Purely software means of monitoring involves inserting monitoring statements within the target program. If these statements are left *permanently* within the target program, then this method of monitoring will qualify as a non-intrusive method (as defined above).

  Software monitoring has many advantages. First and foremost is the flexibility and ease of performing the task as opposed to a hardware scheme. Secondly software probing has a greater power of expression i.e, one could specify more complicated sequences of events on which to trigger the activity of monitoring. Thirdly, software monitoring allows a greater control over the total amount of information that is gathered. However, the biggest disadvantage of software monitoring is that it reduces the efficiency of the program because it does not contribute towards the computation. Efficiency is further reduced because the output of the various monitoring statements must be enclosed within a costly mutual exclusion construct so that output from multiple threads does not get intermingled. Also for software monitoring one has to know apriori where to place the monitoring instructions.

- Hardware-cum-Software means:

  We are advocating the use of both hardware and software to perform the task of monitoring a parallel program. Later in the thesis we shall identify events that are *more important* than others and prove that monitoring their sequence is *sufficient* to reproduce the behavior of the program. Hardware monitoring, therefore, will concentrate on observing these events and recording their sequence. We are advocating software assistance in determining *long term* errors i.e, errors that surface long after the beginning of the program. Of course, the software assistance should effect the efficiency only minimally. The assistance is basically in the form of a macro instruction that triggers the start of a non-intrusive checkpoint. Details of the instruction and the activity of monitoring will be dealt with in greater detail in subsequent sections.

## 3.8 Monitoring issues: Specific

Phase I is concerned with monitoring the program execution. An obvious question is: At what level of abstraction should monitoring take place?

A high level instruction (statement) in a source language like ADA is interpreted by a number of levels before finally being executed by the machine. More clearly, a source-level instruction is interpreted by a set of machine instructions and each machine instruction is in turn interpreted by one or more micro-instructions. Each of these levels can be considered as an abstraction of the machine presented to the user. An abstraction level is said to have the property of *total precedence* iff the following is true:

- If $i_1$ and $i_2$ are any two instructions in the same abstraction level, then during an execution, either all the instructions interpreting $i_1$ (at lower levels of abstraction) occur before all the instructions interpreting $i_2$ or vice-versa i.e, the execution of instructions interpreting $i_1$ are not intermingled with those of $i_2$.

Clearly, the activity of monitoring should take place at an abstraction level that satisfies the *total precedence property*. In other words we need an abstraction level where instructions are executed atomically i.e, without being interrupted. Generally, in most architectures, the machine instructions are executed atomically. Certain interrupts e.g, a page fault while fetching an instruction that spans a page boundary, have to be accepted. In such an event the context is switched to the operating system which fetches the necessary page and restarts the instruction. Note that in general, the operating system, after fetching the page, would not switch the context to another application process (because that process might throw away the page just fetched, in which case all the time spent in fetching the page is wasted). Therefore, even though certain interrupts could be accepted in the middle of a machine instruction, execution of the machine instruction is atomic with respect to other application processes of the target program. The total precedence property is still valid. We therefore advocate that monitoring be conducted at the machine instruction level.

Having decided on monitoring at the machine level, we will explore the type of machine instructions which should be monitored for the purpose of reproducibility

## 3.9 Notion Of Relevant Instructions

As discussed in a previous section, we concluded that the machine instruction level is the most appropriate level at which to carry out the activity of monitoring a program.

In this section we shall answer the question of *Which Instructions to monitor?, all? or, is a subset sufficient?*

On a single processor, the execution of a program is characterized by a sequence of machine instructions. Each machine instruction executes atomically and advances the program to a new state. This sequence of machine instructions contains instructions from various processes of the program, interleaved in an unpredictable manner. The processes of the program co-operate and communicate with each other by accessing memory locations shared by the corresponding processes. Normally, shared locations are enclosed within guarded regions and access to these regions is restricted to provide a consistent view to all processes.

Consider a situation where the programmer inadvertently omits guarding a shared region i.e, a region accessed by many processes. Since the region is unprotected, the processes of the program can be unpredictably interleaved within this region i.e, a read performed on a location within the region by a particular process can be immediately followed by a write performed on the same location by a machine instruction belonging to another process. The behavior produced by such sequencing could potentially lead to a timing error. In particular, a change in the computational rate of a process during a subsequent execution, could alter the sequence of reads and writes over shared memory locations. Since a read and a write over the same object are generally not commutable in that the final state of an object is different if a read is followed by a write than the other way around, the behavior could potentially be different and dependent on the computational rates of the constituent processes. The behavior of the object, therefore depends on the exact sequence of read and write operations over it.

Consider, on the other hand, operations over objects local to a process. If the process is re-executed under the same input conditions, the sequence of machine instructions executed will remain the same as before. As a result, the objects will undergo the same sequence of state transitions. These instructions are therefore free from interference from instructions in other processes since the two operate over disjoint memory locations.

From the above discussion it is clear that, if the input conditions are the same, then

- The sequence of machine instructions operating on locations local to a process will remain the same over successive executions (provided of course that all inputs to the process remain the same).
- The sequence of machine instructions operating on locations shared by different processes may not remain the same over successive executions.

Clearly therefore, machine instructions over shared memory are in a way *more important*, in that they determine the behavior of program. We term these instructions as being *relevant*, from a behavioral standpoint. These are however, only some of the relevant instructions. We shall, in a later section, define the notion of relevancy more formally and define a criteria by which a particular machine instruction can be categorized as relevant. We shall also prove that the sequence of relevant instructions, uniquely determines the behavior of a program. It is therefore sufficient to monitor their execution sequence.

## 3.10 Behavior - A definition

Before proceeding further, it is imperative that we describe our notion of *behavior*. While we defer a rigorous and formal definition to a later section, we will describe the broad categories of behavior and the need for such a classification in this section. We classify behavior broadly into two categories *non real-time* and *real-time*.

- Non-Real-time: The behavior of an object (addressable by the program) is characterized by the sequence of values attained by that object during a particular execution. The behavior of a program is characterized by the behavior of its constituent objects . The behavioral reproducibility of a non real-time program implies that objects obtain the same sequence of values as they did under the original execution.

- Real-time: The real-time behavior of an object is characterized by a sequence of value-time tuples of the form $< v, t >$, where v is the value of the object read or written, from/to, the object at time t. Some of these objects must meet deadline constraints dictated by the application. Behavioral reproducibility of a real-time program therefore, implies that the objects should have the same value-time behavior under the replay as they did under the original execution.

Clearly, replaying a real-time execution is more difficult than replaying a non real-time execution because of the extra temporal constraints that the replay should satisfy. Consequently, the monitoring phase for replaying a real-time execution has to be more detailed. We shall see in subsequent sections that the class of instructions that need to be monitored depends on the type of behavior that the replay should satisfy. In other words the notion of relevancy is tied to the type of behavior that needs to be reproduced.

## 3.11 Modelling

### 3.11.1 Uniprocessor

We model the execution of a program on a single processor as a sequence of atomic machine instructions (actions). Re-execution of the same program with the same set of inputs could lead to a different linear sequence sequence of actions, for the reasons described earlier, the computational rates of the constituent processes could vary across executions. Each linear sequence corresponds to a particular behavior(real-time and non real-time) of the program. It is also possible that multiple linear sequences of actions could correspond to the same non real-time behavior. In subsequent sections we will split the real-time class into two categories, namely, *weak* and *strong* real-time. Multiple linear sequences could correspond to the same weak real-time behavior, while strong real-time is uniquely characterized by the total sequence of actions. Clearly then, it implies that, for certain classes of behavioral reproducibility, the replay need NOT reproduce the exact sequence of actions as the original execution. As we saw in the previous section, this characteristic is the property of an instruction being relevant. We shall describe a more formal notion of relevancy and prove that for each of the behaviors there exists a class of *relevant* instructions, whose relative sequence within the overall execution, identifies the execution as a whole with a unique behavior class. In other words, for achieving certain classes of behavioral reproducibility, the replay is only constrained to ensure the same sequence of relevant instructions (NOT all instructions).

### 3.11.2 Multi-Processor

In a shared memory multiprocessor, more than one instruction could simultaneously be executed. As a direct consequence, the execution of a program cannot be characterized as a linear sequence of actions. We will characterize the execution of a program, on a shared multi-processor containing n processors, as a linear sequence of n-tuples. The sequencing of tuples is with respect to a logical clock with a discrete time model. The logical clock is incremented each time an action writes to shared memory.

Execution in the interval (t,t+1) is represented by a sequence of n-tuples (n is the number of processors in the system). The i-th component within the n-tuple is either $\epsilon$ denoting a null action, or denotes an action executed at the i-th processor. The tuple representing the activity at time t is a special tuple called the *update* tuple. An update tuple has exactly one non-null component corresponding to the processor that actually writes to memory. All other components within an update tuple are $\epsilon$, representing the fact that only one processor can write to shared memory at any one time. The definition of the tuples within the interval (t,t+1) is as follows:

- If m denotes the maximum number of instructions executed at any one of the n-processors in the logical time interval (t,t+1), then the interval (t,t+1) is represented by m n-tuples.

- The i-th component of the j-th n-tuple, representing the activity between logical times t and t+1, denotes the j-th action completed by the i-th processor after logical time t and before t+1. If j is greater than the total number of actions executed by the i-processor in the logical time interval (t,t+1), then the corresponding component within the j-th n-tuple is $\epsilon$. To further clarify the sequencing, let there be 3 processors in the shared memory multi-processor system. Suppose the first processor executed 2 instructions, the second executed 3 instructions and the 3rd processor executed 5 instructions within some interval (t,t+1). The interval (t,t+1) will then be represented as a sequence of five 3-tuples. The first action executed by the each of the 3 processors will belong to the first tuple. The second action executed by the 3 processors will belong to the second tuple. Since the first processor executed only 2 instructions in the interval (t,t+1), the first component in the 3rd, 4th and 5th tuples will be represented by $\epsilon$. Similarly for the 2nd processor, the second component in the 4th and 5th tuples will be $\epsilon$.

## 3.12 Theoretical Basis for Uniprocessors

In this section we shall define more formally, among other things, the notions of behavior, reproducibility of behavior and notion of relevancy of an instruction. We will be assuming a uni-processor environment.

**Def 3.12.1 (Object)** Every non-literal operand of a macro-instruction is an object. The state of an object is characterized by its contents.

**Def 3.12.2 (State)** The state of a computation is characterized by the union of states of the individual objects of the computation.

**Def 3.12.3 (Atomic Action)** An atomic action is an activity on one or more objects that groups together operations on the system state into a unit that is indivisible with respect to other atomic actions. An *Atomic Action* results from the execution of a macro-instruction. An atomic action is said to be a mutable action with respect to an object if the state of the object changes as a result of the action. It is called immutable otherwise.

Mathematically, we write $m_j^k$, to denote the atomic action resulting from the k-th execution of the machine-instruction $m_j$. Every atomic action is therefore uniquely identified. For brevity, we shall use $a_i$ to refer to an action, with the implicit understanding that it is of the form $m_j^k$, for some $m_j$, k. A machine instruction represents a static entity, whereas an action represents an execution of the machine instruction. It is used to differentiate between the various executions of the same machine instruction. Execution of a program (on a uni-processor) can then be represented as a linear sequence of actions.

**Def 3.12.4 (Program-trace)** A Program-trace T=$a_1a_2 \ldots a_m$, of a program P executing on a single processor, is a sequence of actions, where

- $\forall$ i, $1 \leq i \leq m$: $a_i \in$ A, where A represents the set all actions executed by the program.
- $\forall$ x $\in$ A, $\exists$ j, $1 \leq j \leq m$ such that x=$a_j$.
- $\forall$i, $1 \leq i \leq m$ and $\forall$j, $1 \leq j \leq m$ such that i < j, $a_i$ is executed before $a_j$.

**Def 3.12.5 (Belongs)** An action $a_j$ belongs to a program trace T=$a_1a_2..a_n$ iff $1 \leq j \leq n$.

**Def 3.12.6 (Domain(action))** The domain of an action $a_i$, denoted as DOM($a_i$), is the set of all objects referenced by the macro-instruction corresponding to $a_i$.

**Def 3.12.7 (Disjoint)** Two actions $a_i$ and $a_j$ are said to be disjoint iff $DOM(a_i) \cap DOM(a_j) = \emptyset$.

**Def 3.12.8 (Input values,Input/Output actions)** Each action within a trace can be looked upon as consuming some data values and generating data values to be consumed by subsequent actions. Certain actions, however, could consume data values that are not generated by any other action of the program. These values are referred to as **input values** i.e, they are values that are read from the external environment in which the program is being executed.

As an example, the values read from an input port are input values. Actions of the program that consume input values are termed as **input actions.**

In a similar fashion, values written out to an output port are termed output values (they are not consumed by any other action of the program). Actions of the program that generate output values are called **output actions.**

**Def 3.12.9 (Input Conditions)** A program is said to be re-executed under identical *input conditions* if the input values consumed at corresponding input actions remain the same as the original execution.

**Def 3.12.10 (Power-trace)** The Power-trace of a program P, is defined to be the set of all possible execution sequences of P, while keeping the input conditions the same. Each trace within the power-trace of a program P represents a different, but possible, execution sequence of action in P. The program-traces generated during successive executions of P could differ because of the following reasons.

- The timing of asynchronous events is unpredictable. As a direct consequence, the process schedule sequence need not be identical over repeated executions. A change in the scheduling sequence basically alters the relative rates of computation of the constituent processes, thereby altering the linear sequence of actions of P.

**Def 3.12.11 (Precedes)** An action $a_i$ is said to precede action $a_j$ , denoted as $a_i$ => $a_j$, iff

- $\forall$T $\in$ Power-Trace(P) $a_i$ occurs-before $a_j$ in trace T.

**Def 3.12.12 (Behavior)** Let T be a trace of the program P. The behavior of P under trace T is characterized by the sequence of state transitions of each object. More specifically, if T' is another trace of program P, then T and T' are said to be behaviorally equivalent if the sequence of state transitions of each object is identical under both traces. Similarly, the behavior of an object under a trace T is characterized by the sequence of state transitions of that object. This behavior will also be referred to as *non real-time behavior:*

If x is an object. Then its behavior under trace T can be represented as Behavior(x,T) = $< s_0, s_1, \ldots, s_n >$, where $< s_0, s_1, \ldots, s_n >$represents the ordered sequence of state transitions that object x undergoes under trace T.

If P is a program, then its behavior under trace T will be denoted as Behavior(P,T) = { Behavior($x_i$,T): $x_i$ is an object of P }

**Def 3.12.13 (Commutability)** Two actions $a_i, a_j$ in trace T, are commutable iff all the following three conditions are satisfied:

- They are disjoint, i.e, $DOM(a_i) \cap DOM(a_j) = \emptyset$ OR
  - $DOM(a_i) \cap DOM(a_j) \neq \emptyset$ AND
  - $a_i$ and $a_j$ contain immutable actions wrt all objects in $DOM(a_i) \cap DOM(a_j)$ AND
  - $\forall$ x $\in$ T: $a_i \prec x \prec a_j$, x and $a_j$ are commutable, where $x \prec y$ represents the relation 'x occurs before y in T'.
- The actions belong to different processes.
  – If they belong to the same process, interchanging them would alter the semantics of the program.
- Neither $a_i$ nor $a_j$ precedes the other.

In general therefore, the fact that two tuples are not commutable, tells us that they interact by operating on the same object in such a way that their sequence of execution determines the behavior of the object. In other words, it tells us that if all non-commutable tuples occured in the same sequence during a re-execution, then the behavior of the program will be the same as before. This notion is specified more clearly in the next definition.

Refer to example 1.1 below for a clearer understanding.

**Def 3.12.14 (Relevant)** An action $a_i$ occuring within a program trace T=$a_1 a_2 .. a_n$ is said to be relevant if either of the following is true:

- $\exists$ action $a_j \in$ T: $a_j$ *occurs before* $a_i$ within T AND
  $\exists T' \in Power-trace(P) : a_i$ *occurs before* $a_j$ in T' AND
  $a_i$ and $a_j$ are NOT commutable.

- $\exists$ action $a_j \in$ T: $a_i$ *occurs before* $a_j$ within T AND

  $\exists T' \in Power - trace(P) : a_j$ *occurs before* $a_i$ in T' AND

  $a_i$ and $a_j$ are NOT commutable.

See example 1.1 below.

In other words, an action $a_i$ is relevant if there exists the possibility of $a_j$ occuring such that the behavior of the program varies depending on whether $a_i$ occured before $a_j$ or after $a_j$. Such information is essential e.g, if $a_i$ occured before $a_j$, during phase I then it must do so in phase II also. Relevant actions as defined above, will also be referred to as non real-time relevant actions.

**Example 3.12.1** Let $P_1$ and $P_2$ be two processes of a program P, with the following instructions in each. The instructions are in a single operand format i.e, the accumulator is the default operand. Thus *Load x* means load the value of x into the accumulator.:

$P_1$: [ ... ;Load 1 ; $m_{11}$: store x ; $m_{12}$: Send.$P_2$(y) ; ... ; ... ]

$P_2$: [Load 10 ; $m_{21}$: store x ; ... ; $m_{22}$: load x ; $m_{23}$: add 1 ; $m_{24}$: store x ; $m_{25}$: Receive.$P_1$(z) ; $m_{26}$:... ; ...]

- Objects

  x is an object, since it appears as an operand in the machine instructions above.

- Atomic Actions

  $m_{11}$ , $m_{12}$ , $m_{21}$ ... $m_{26}$ are all macro-instructions. The atomic actions arising out of say $m_{11}$, will be of the form $a^i_{jk}$ denoting the atomic action arising out of the i-th execution of the macro-instruction $m_{jk}$. For notational simplicity, we shall henceforth ignore the super-script on atomic-actions and use it only to distinguish multiple actions arising out of the same macro-instruction. A superscript of 1 will be assumed for all actions. $a_{11}$ , $a_{21}$ , $a_{22}$ , $a_{23}$ and $a_{24}$ are all atomic actions

Mutable actions: $a_{11}$ , $a_{21}$ , $a_{24}$ are mutable wrt the object x

Immutable actions: $a_{22}$ is immutable wrt object x.

- Program Traces

  Some examples of uni-processor action traces are

  $t_1$: ...$a_{11}a_{21}a_{22}a_{12}a_{23}a_{24}a_{25}a_{26}$...

  $t_2$: ...$a_{21}a_{11}a_{22}a_{23}a_{24}a_{12}a_{25}a_{26}$...

  $t_3$: ...$a_{21}a_{22}a_{11}a_{23}a_{24}a_{12}a_{25}a_{26}$...

  $t_1$ ,$t_2$ and $t_3$ represent different but possible execution traces. Also, the behavior of the two processes is different under $t_1$ and $t_2$ since the state transition sequence for the object x is different under $t_1$ than under $t_2$.

- Precedes

  Action $a_{12}$ precedes action $a_{25}$ because, in all possible execution sequences, $a_{12}$ occurs before $a_{25}$. Control could reach $a_{25}$ before $a_{12}$, but it cannot proceed unless and until $a_{12}$ is executed. Since all actions within each process are sequential, all actions occuring before $a_{12}$ in $P_1$ precede those occuring after $a_{25}$ in $P_2$. In particular, $a_{11}$ precedes $a_{25}$ and $a_{26}$.

- Domain

  Domain($a_{11}$) = x

  Domain($a_{12}$) = y

  Domain($a_{25}$) = z

- Commutability

  Commutability is defined with respect to a trace. So let T be a trace defined as follows

  T: $a_{21}a_{22}a_{11}a_{23}a_{12}a_{24}a_{25}a_{26}$

  $a_{11}$ is NOT commutable with $a_{22}$ because their domains overlap and $a_{11}$ is a mutating action.

$a_{11}$ is NOT commutable with $a_{26}$ because of two reasons. The first is because $a_{24}$ lies between them in T and $a_{24}$ and $a_{11}$ are not commutable. The second is because $a_{11}$ precedes $a_{25}$ and hence $a_{26}$.

$a_{11}$ and $a_{12}$ are NOT commutable because they belong to the same process, namely $P_1$.

- Relevant

  Consider trace T as defined above. Action $a_{11}$ is relevant within trace T because there exists another action $a_{22}$ such that interchanging the order of $a_{11}$ and $a_{22}$ gives rise to a new possible execution sequence in which the behavior of x is different. If the new trace is say T', then

  Behavior(x,T) = < 10 , 1 , 11 >

  Behavior(x,T') = < 10 , 1 , 2 >

  Since the behavior of x is different, the behavior of the program under T is different from that under T'.

**Theorem 3.12.1** If T and T' are two valid traces of a program P such that:

- They are obtained by executing P under identical input values.
- The same set of actions occur in T and T', though not necessarily in the same order.
- The sequence of relevant events in T is identical to that in T'.

THEN they are non real-time behaviorally equivalent from the level of atomic actions.

Proof: By contradiction:

Suppose, on the contrary, that T and T' do not exhibit the same behavior. This implies that

$\exists$ an object x, that does not undergo the same sequence of state transitions under T and T'.

Let $< S_0^{x,T}, S_1^{x,T}, \ldots, S_n^{x,T} >$ be the sequence of states that object x goes through under T. Similarly, let $< S_0^{x,T'}, \ldots, S_n^{x,T'} >$ be the sequence of states that x goes through under T'.

Further, let the i-th state be the first state that is different between T and T'. i.e, $\forall k, 0 \leq k < i, S_k^{x,T} = S_k^{x,T'}$.

Let $a_i^{x,T}$ and $a_i^{x,T'}$ be the differing actions on x in traces T and T' respectively. Mathematically, $S_{i-1}^{x,T} \overset{a_i^{x,T}}{\vdash} S_i^{x,T}$ and

$S_{i-1}^{x,T'} \overset{a_i^{x,T'}}{\vdash} S_i^{x,T'}$

Since the same set of actions occur within T and T' (by assumption) . there must be an action in T that is identical to $a_i^{x,T'}$. Let that be denoted as $a_j^{x,T}$. Now we have , $a_j^{x,T} = a_i^{x,T'}$

Now consider $a_j^{x,T}$ and $a_i^{x,T}$:

- x $\in$ DOM($a_j^{x,T}$) $\cap$ DOM($a_i^{x,T}$). Therefore DOM($a_j^{x,T}$) $\cap$ DOM($a_i^{x,T}$) $\neq \emptyset$ .
- Since they occur in different order in T and T', they must belong to separate processes
- Actions $a_j^{x,T}$ and $a_i^{x,T}$ cannot both be immutable actions wrt x, because $S_i^{x,T} \neq S_i^{x,T'}$

- Neither $a_i^{x,T}$ nor $a_j^{x,T}$ precedes the other because we know that their order is reversed in T' and T' $\in$ Power-trace(P).

This implies that $a_j^{x,T}$ and $a_i^{x,T}$ are both relevant events that occur in different order relative to each other in T and T', which contradicts the assumption that the sequence of relevant events are identical in T and T'.

Thus we conclude that if T and T' represent traces of program P executed under identical input conditions and if the sequence of relevant actions within T and T' are maintained in exactly the same order, then the behavior of T is the same as that of T'.

□ Q.E.D

**Theorem 3.12.2** Let R be a relation defined on the Power-trace(P) of some program P, with the following meaning

- $\forall\, T_1, T_2 \in$ Power-trace(P), $T_1$ is related to $T_2$ (denoted as $T_1$ R $T_2$), if the relative sequence of relevant instructions in $T_1$ is identical to that in $T_2$.

then, R is an equivalence relation on Power-trace(P).

Proof:

R is **Reflexive:** $T_1$ is obviously related to itself.

R is **Symmetric:** If $T_1$ is related to $T_2$, then the relative sequence of relevant instructions is $T_1$ is identical to that in $T_2$ (By definition). This immediately implies that the reverse is also true, i.e, the relative sequence of relevant instructions in $T_2$ is identical to that in $T_1$ (because the equality relation is symmetric). Thus $T_2$ R $T_1$ and hence R is symmetric.

R is **Transitive:** if $T_1$ R $T_2$ and $T_2$ R $T_3$ then obviously $T_1$ R $T_3$, again because the equality relation is transitive.

Since the relation R is Reflexive, Symmetric and Transitive, we conclude that R is an equivalence relation [Kor74].

Since any equivalence relation on a set, divides the set into a number of disjoint equivalence classes, the relation R divides the Power-trace(P) into a number of equivalence classes. All traces within an equivalence class exhibit the same non real-time behavior.

□ Q.E.D

## 3.13 Theoretical Basis for Multi-processors

**Def 3.13.1 (Null-action)** A Null-action, denoted as $\epsilon$, is used to model the fact that the processor is idle within some time interval. The domain of a Null-action is a null set.

**Def 3.13.2 (Local-time)** Time maintained at each processor site is called Local-time and is incremented upon the execution of an instruction by that processor. A null-action takes up some non-zero time of the processor.

**Def 3.13.3 (Global-time)** Global-time is the time as seen by the bus interconnecting the processors to the shared memory. It is incremented on each access (read/write) to shared memory M. Note here, that several register instructions (i.e, instructions that don't access shared memory) can be executed "at" a given global time.

Just as Local-time sequences the actions at a processor, Global-time defines a partial order between actions executed on different processors. If $a_i$ is an action executed by the i-th processor at Global-time t, and $a_j$ is the action executed by the j-th processor($i \neq j$) also at Global-time t then $a_i$ and $a_j$ are assumed to have been executed in parallel even though their execution may not have actually been concurrent.

**Def 3.13.4 (Action-tuple)** An action-tuple of a program P, distributed across n processors, sharing memory M, is an n-tuple of the form $(a_1, \ldots, a_n)$, where:-

- $\forall$j, 1$\leq$j$\leq$n, action $a_j$ is either $\epsilon$ or represents an action executed by the j-th processor.
- $\forall$j: 1$\leq$j$\leq$n, $a_j$ does not access shared memory M for read/write. All actions within an action-tuple have the same Global-time associated with them.

Note that from the definition of the action-tuple above, it is clear that the domains of all the actions within an action-tuple are disjoint.

**Def 3.13.5 (Memory-tuple)** A Memory-tuple of a program P, distributed across n processors, sharing memory M, is an n-tuple of the form $(a_1, \ldots, a_n)$, where:-

- $\exists$ exactly one j, 1$\leq$j$\leq$n: $a_j$ accesses shared memory M for read/write AND
- $\forall$i, 1$\leq$i$\leq$n: i$\neq$j, $a_i$ is $\epsilon$. Here too, all actions within the Memory-tuple have the same Global-time associated with them.

A Memory-tuple demarcates groups (could be a single action-tuple) of action-tuples associated with successive Global-times.

**Def 3.13.6 (Reduced-Action-tuple)** A Reduced-action-tuple of a program P, distributed across n processors, sharing memory M, is an n-tuple of the form $(a_1, \ldots, a_n)$, where:-

- $\exists$j, 1$\leq$j$\leq$n, action $a_j$ is NOT $\epsilon$.
- $\forall$i, 1$\leq$i$\leq$n such that i$\neq$j, action $a_i$ is $\epsilon$ (i.e, it is a null action).

A reduced-action-tuple therefore is a special kind of action-tuple which contains exactly one non-null action. The idea behind defining such a notion is that we will show later that every action-tuple is equivalent to a sequence of reduced-action-tuples. It will be used to show that a multi-processor trace is equivalent to some uni-processor trace. The difference between reduced-action-tuple and an Memory-tuple is that in a reduced-action-tuple, the single non-null action in the tuple need not necessarily access shared memory M for read/write.

**Def 3.13.7 (Occurs-Before(action))** Action $a_i$ is said to Occur-Before action $a_j$ iff

- $a_i$ and $a_j$ are executed by the same processor P and the Local-time of $a_i$ is less than that of $a_j$. OR
- $a_i$ and $a_j$ are executed by different processors and the Global-time of $a_i$ is less than the Global-time of $a_j$.

Note here that if $a_i$ and $a_j$ are executed by different processors and have the same Global-time associated with them then they are not comparable.

**Def 3.13.8 (Occurs-Before(tuple))** A tuple (Memory-tuple or Action-tuple) $t_i$ is said to occur-before tuple (Memory-tuple or Action-tuple) $t_j$ iff:

- The Global-time of $t_i$ (Which is the same as the Global-time of any of the actions in $t_i$. [1]), is less than the Global-time of $t_j$ OR
- If the Global-times of $t_i$ and $t_j$ are the same and if every non-null action in $t_i$ Occurs-Before the corresponding non-null action in $t_j$ (here the comparison is with respect to the local-times of corresponding actions in the two tuples), then $t_i$ Occurs-Before $t_j$.

**Def 3.13.9 (Multi-trace)** A Multi-trace $P_T=t_1t_2\ldots t_m$, of a program P distributed across n processors, is a sequence of action tuples and Memory-tuples, where:

- $\forall$i, $1\leq i\leq m$, $t_i$ is an action-tuple or a Memory-tuple.
- $\forall$i,j: $i < j$, Global-time($t_i$) $\leq$ Global-time($t_j$).
- Every action $a_i$ executed within the program P, belongs to some tuple $t_j$, $1\leq j\leq m$.

Note that the ordering between (action-tuples or Memory-tuples) is based on a non-descending order of Global-time. Ordering between tuples with the same Global-time is based on an ascending order of local-time between corresponding actions in the two tuples.

[1] Note that the Global-time of all actions in a Memory-tuple or Action-tuple have to be the same, by definition.

**Theorem 3.13.1** Every action-tuple is behaviorally equivalent to some sequence of reduced-action-tuples.

Proof:

Let $(a_1,a_2,\ldots,a_n)$ be an action-tuple with r non-null actions in it. Without loss of generality, suppose that the first r actions i.e, $a_1,a2,\ldots,a_r$ be the r non-null actions. Also let $S_0$ be the initial state of the multi-processor system before the execution of the action-tuple $(a_1,\ldots,a_n)$. Again let the part of the state that is modified by the execution of $a_i$ in isolation, be $s_i$ ($\forall i$, $1\leq i\leq n$). We will use the notation $S_0+s_i$ to denote the fact that the state $S_0$ has been modified by accomodating the changes dictated by $s_i$.

If $S_0$ is the initial state, then the following is a list of action-tuples and the corresponding final states.

$(a_1,\epsilon,\ldots,\epsilon) \rightarrow S_0+s_1$

$(a_1,a_2,\ldots,\epsilon) \rightarrow S_0+s_1+s_2$

This is because the domains of $a_1$ and $a_2$ are disjoint and therefore they modify different parts of $S_0$. The final state therefore is the summation of changes made by the individual actions. In other words, executing $a_1$ and $a_2$ in parallel has the same effect as executing them sequentially in either order.

$(a_1,a_2,\ldots,a_n) \leftarrow S_0+s_1+\ldots+s_n$

Again, since all $a_i$ are mutually disjoint, executing them in parallel is equivalent to executing them in any sequence. In particular executing the action-tuple $(a_1,\ldots,a_n)$ is equivalent to the following sequence of reduced-action-tuples:

$(a_1,\epsilon,\ldots,\epsilon)$

$(\epsilon,a_2,\ldots,\epsilon)$

$\vdots$ $(\epsilon,\ldots,a_n)$

Note that the above sequence of reduced-action-tuples is not unique. The reduced-action-tuples above can be commuted to arrive at a different but equivalent (to the original action-tuple) sequence of reduced-action-tuples.

**Def 3.13.10 (Reduced-Multi-trace)** A reduced-Multi-trace of a program P distributed across n processors is obtained by replacing each action-tuple in the multi-trace of P by the sequence of reduced-action-tuples, as described in theorem 3.13.1.

Note that a reduced-Multi-trace logically represents a uni-processor trace and therefore all the definitions and results proved for uni-processor traces in the previous section can now be applied to the reduced-Multi-Trace and hence a multi-processor trace.

## 3.14 Real Time Behavior

The behavior of a non real-time program has been characterized by the behavior of its constituent objects in the data state which in turn is defined as the sequence of values attained by that object during the course of execution. This definition, however, is inadequate for describing real-time programs. More clearly it does not describe the temporal nature of the program, in the sense that the state of an object is solely determined by its value (content) without any indication of the time at which it received that value.

As mentioned earlier, correctness of a real-time program is determined by not only by the values of the constituent objects but also the times at which those values were obtained.

The *real-time* behavior of an object is the sequence of tuples of the form $< v, t >$, where $v$ is the value attained by the object at time t. The behavior of a real-time program is characterized by the *real-time* behavior of its contituent objects.

Note that with the above definition, two traces of a program executed under identical input conditions represent the same behavior iff they are themselves identical i.e, the relative sequence of every instrution has to be identical between the two traces.

Considerable overhead (in terms of storage) is necessary to monitor a program to reproduce its real-time behavior. We shall therefore define two classes of real-time behavior, namely, *weak* and *strong* real-time behavior. Before defining them formally, here are some definitions.

**Def 3.14.1 (Thread-Of-Control)** A thread-of-control, C, is a 3-tuple (CPU-state, PGM-space, DATA-space), where PGM-space is a set of ranges, $\{ [l_1, l_2], \ldots , [l_i, l_{i+1}] \}$ of memory locations containing non-modifiable machine instructions that can only be executed. DATA-space is also a set of ranges $\{ [d_1, d_2], \ldots , [d_j, d_{j+1}] \}$ of memory locations containing the necessary data objects for the above instructions to operate on. CPU-state is again a 3-tuple (PC, PSW, Reg-set) where PC is the program or instruction counter and always lies within one of the ranges defined by the pgm-space, PSW contains processor dependent information and Reg-set is the set of registers used by the program.

**Def 3.14.2 (Belongs)** An action $a_i$ belongs to a thread-of-control C if the machine instruction $m_i$ corresponding to $a_i$ lies within one of the ranges defined by the pgm-space of C.

**Def 3.14.3 (Thread Transfer Point(TTP),Entry Action)** An action $a_i$ within a trace T of a program P, is referred to as a **thread transfer point**(TTP) iff $a_i$ and $a_{i+1}$ belong to different threads-of-control. $a_{i+1}$ is called the **entry action**, signifying that it is the action executed next in the new thread-of-control (i.e, it is the action where control enters or re-enters the new thread of control). Note that the thread of control refered to here is that of the application program and not the operating system. Usually, switching threads of control requires the intervention of the operating system. Therefore, the first action executed after a TTP will be some action within the operating system. However, we are not interested in the operating system. Instead we are interested in the first action of the application program executed after a TTP ( and we will refer to that as an entry action). Of course, one could include the operating system if desired.

Note that, as opposed to the notion of relevancy, a TTP is not a characteristic of the machine instruction itself but rather it is the instruction at which the corresponding thread-of-control is preempted. Also an entry action is not necessarily the first action within a thread-of-control but could occur anywhere within the thread-of-control.

**Def 3.14.4 (Annotated Trace)** The annotated trace $T^A$ of a trace T=$a_1 \ldots a_n$ of a program P, is obtained by applying the following transformation on the sequence described by trace T

- transform $a_1$ to the tuple $< a_1, 0 >$
- If $a_i$ is a relevant action, then , if $a_{i-1}$ is transformed to the tuple $< a_{i-1}, t >$, then $a_i$ is transformed to the tuple $< a_i, t+1 >$.
- If $a_i$ is a TTP, then , if $a_{i-1}$ is transformed to the tuple $< a_{i-1}, t >$, then $a_i$ is transformed to the tuple $< a_i, t+1 >$.
- If $a_i$ is neither a relevant instruction nor a TTP, then, if $a_{i-1}$ is transformed to the tuple $< a_{i-1}, t >$, then $a_i$ is transformed to the tuple $< a_i, t >$.

The annotated trace $T^A$ is the sequence $< a_1, 0 > .. < a_i, t_i > .. < a_n, t_n >$ with the annotations derived according to the transformation described above.

Basically, the idea behind an annotated trace is as follows. The tuple $< a_i, t >$ indicates that action $a_i$ was executed at logical time t, where the logical time starts out as 0 and is incremented by 1 on every context switch and on the execution of a non real-time relevant instruction.

**Def 3.14.5 (Weak Real-Time Behavior)** Weak real-time behavior of a program P with trace T, is determined by the behavior of *relevant* objects (i.e, objects that belong to the domain of relevant instructions) under the annotated trace $T^A$. The behavior of a *relevant* object is characterized by a sequence of value-time tuples of the form $< v, t >$, where $v$ is the value attained by the object at time $t$, where $t$ is the value of the logical clock determined according to the transformation used to derive $T^A$. If $< a_i, t >$ is a tuple in $T^A$, then all mutable objects in DOM($a_i$) are assumed to attain their new values at logical time t. The weak real-time behavior of a program is characterized by behavior of the constituent relevant objects.

The reason for calling this a weak real-time behavior is that it does not characterize the timing behavior of **all** objects. In particular, it characterizes the timing behavior of only relevant objects. Hence the term weak.

**Lemma 3.14.1** If $T_1$ and $T_2$ are two traces of a program P, which exhibit the same weak real-time behavior, then

- The number of entry actions between two relevant actions in $T_1$ is equal to the number of entry actions between the corresponding relevant actions in $T_2$.

Proof: Let $a_{i1}$ and $a_{i2}$ be two relevant actions in $T_1$ Also, let the corresponding relevant actions in $T_2$ be $a_{j1}$ and $a_{j2}$ ( $a_{j1}$ and $a_{j2}$ must exist because $T_1$ and $T_2$ exhibit the same non real-time behavior. )

Let x denote the number of entry actions between $a_{i1}$ and $a_{i2}$ in $T_1$ and y denote the number of entry actions between $a_{j1}$ and $a_{j2}$ in $T_2$.

Also, let r denote the number of relevant actions between $a_{i1}$ and $a_{i2}$. Note that the sequence of relevant actions between $a_{i1}$ and $a_{i2}$ is the same as that between $a_{j1}$ and $a_{j2}$ (because $T_1$ and $T_2$ belong to the same weak real-time behavior class and hence the same non real-time behavior class). We conclude therefore, that r also denotes the number of relevant actions between $a_{j1}$ and $a_{j2}$.

Consider the annotated traces $T_1^A$ and $T_2^A$. Let t be the logical time at which $a_{i1}$ occurs in $T_1^A$. Since $T_1$ and $T_2$ exhibit the same weak real-time behavior, $a_{j1}$ must occur at the same logical time t, in $T_2^A$ (otherwise the real-time behavior of the relevant object in DOM($a_{j1}$) will not be the same under the two traces.)

With the above assumptions and observations, let us now compute the logical time at which $a_{i2}$ and $a_{j2}$ occur in the annotated traces $T_1^A$ and $T_2^A$. Since there are r relevant actions between $a_{i1}$ and $a_{i2}$ and there are x TTPs , the logical time at which $a_{i2}$ occurs in $T_1^A$ is t+r+x (since non-relevant actions do not affect the logical time, by definition). The logical time at which $a_{j2}$ occurs is similarly t+r+y. Since x $\neq$ y, by assumption, the logical times of $a_{i2}$ and $a_{j2}$ are different. But again this is not possible because then the real-time behavior of relevant objects in the domain of $a_{j2}$ (or $a_{i2}$) will be different under $T_1$ and $T_2$, violating the fact that $T_1$ and $T_2$ are weak real-time behaviorally equivalent.

We conclude therefore, that the number of entry actions between any two relevant actions in $T_1$ is equal to those between the corresponding relevant actions in $T_2$.

□ Q.E.D

**Def 3.14.6 (Weak Real-time Behavior Classes)** Two trace $T_1$ and $T_2$ of a program P, belong to the same weak behavior class iff the weak real-time behavior of P, under the annotated traces $T_1^A$ and $T_2^A$, are identical.

Recall that the power trace of a program P is divided into a number of equivalence classes, where all traces belonging to the same equivalence class represent the same non real-time behavior. In a similar fashion all traces belonging to the same non real-time-behavior-equivalence class are further subdivided into a number of equivalence classes, each representing a particular weak behavior.

**Theorem 3.14.1** If T1 and T2 are two traces of the program P, with the following conditions:

- T1 and T2 are obtained by executing P under identical input conditions.
- The set of all actions in T1 is the same as the set of all actions in T2.
- The sequence of all relevant actions T1 is the same as that in T2 i.e, T1 and T2 belong to the same non real-time behavior equivalence class.
- The number of entry actions between two relevant actions in T1 is the same as the number of entry actions between the corresponding relevant actions in T2.

- If $a_h$ is a entry action occuring between two relevant actions $a_1$ and $a_2$ in T1, then $a_h$ occurs between the corresponding relevant actions in T2. In other words the position of entry actions relative to relevant actions is maintained between T1 and T2. Note that the relative position of entry actions among themselves need not be the same for T1 and T2.

then the following is true,

- T1 and T2 belong to the same weak real-time behavior class.

Proof: By contradiction

Suppose, on the contrary, that T1 and T2 do not belong to the same weak real-time behavior equivalence class. Clearly then, by the definition of weak real-time behavior, the following is true

- $\exists$ a relevant object X, such that the real-time behavior of X under trace T1 is different from its real-time behavior under T2.

Let X be a relevant object whose real-time behavior under T1 is different from that under T2. Let $< v_i^{T1}, t_i^{T1} >$, for successive values of i, denote the real-time behavior of X under trace T1. A similar sequence can be defined under T2. Let the k-th tuple in this sequence, represent the first instance in which the value-time tuples differ under the two traces.

Mathematically, therefore $< v_i^{T1}, t_i^{T1} > = < v_i^{T2}, t_i^{T2} >$, $\forall\ 1 \leq$ i $<$ k and $< v_k^{T1}, t_k^{T1} > \neq < v_k^{T2}, t_k^{T2} >$, where tuple equality is defined to be true iff corresponding components are equal.

Since the T1 and T2 belong to the same non real-time behavior equivalence class, the non real-time behavior of X must be the same under T1 and T2. We conclude that $v_k^{T1} = v_k^{T2}$.

The only reason for the behavior of X to differ under the two traces is because $t_k^{T1} \neq t_k^{T2}$. Let $a_i^{T1}$ denote the relevant action on X that transformed its behavior from $< v_{k-1}^{T1}, t_{k-1}^{T1} >$ to $< v_k^{T1}, t_k^{T1} >$. Similarly let $a_j^{T2}$ denote the relevant action that performed the corresponding transformation on X under T2.

Let $< a_{r1}, a_{r2}, .., a_{rn} >$, denote the sub-sequence of all relevant actions executed under trace T1, up to $a_i$, i.e, $a_i = a_{rn}$ (where r1 < r2 < .. < rn)

Similarly, let $< a_{s1}, a_{s2}, .., a_{sn}$, denote the subsequence of all relevant actions executed under trace T2, up to $a_j$, i.e, $a_{sn} = a_j$ (where s1 < s2 < .. < sn).

Since the sequence of relevant actions is the same under T1 and T2 (by assumption), clearly, $a_{r1}$=$a_{s1}$ and $a_{r2} = a_{s2}$ and so on up to $a_{rn} = a_{sn}$.

Now, the number of entry actions between any two relevant actions in T1 is the same as the number of entry actions between the corresponding relevant actions in T2 (by assumption). Therefore the logical time at which $a_i$ is executed in T1 must be the same as the logical time at which $a_j$ is executed in T2 (since logical time is incremented only on the execution of a relevant instruction and a entry action). We conclude therefore, that $t_k^{T1} = t_k^{T2}$.

Thus there is no relevant object X, whose real-time behavior differs under trace T1 and T2. Hence T1 and T2 exhibit the same weak real-time behavior.

□ Q.E.D

**Theorem 3.14.2** Let R be a relation defined on the Power-trace(P) of some program P, with the following meaning. $\forall\, T_1, T_2 \in$ Power-trace(P), $T_1$ is related to $T_2$ (denoted as $T_1$ R $T_2$) if

- The relative sequence of relevant instructions in $T_1$ is identical to that in $T_2$ AND
- The number of entry actions between any two non real-time relevant actions in T1 is the same as the number of entry actions between the corresponding relevant actions in T2.

then, R is an equivalence relation on Power-trace(P).

Proof:

R is **Reflexive:** $T_1$ is obviously related to itself, i.e, $T_1$ R $T_1$.

R is **Symmetric:** If $T_1$ is related to $T_2$, then the relative sequence of relevant instructions is $T_1$ is identical to that in $T_2$ (By definition). Also the sequence $x_1$,$x_2$..$x_n$, where $x_i$ denotes the number of entry actions between the ith and (i+1)th non real-time relevant action, remains the same for T1 and T2 (by assumption). Thus $T_2$ R $T_1$ and hence R is symmetric.

R is **Transitive:** if $T_1$ R $T_2$ and $T_2$ R $T_3$ then obviously $T_1$ R $T_3$, again because the sequence of relevant actions and the number of entry actions between relevant actions is invariant between T1, T2 and T3. Thus R is transitive.

Since the relation R is Reflexive, Symmetric and Transitive, we conclude that R is an equivalence relation.

Since any equivalence relation on a set, divides the set into a number of disjoint equivalence classes, the relation R divides the Power-trace(P) into a number of equivalence classes. All traces within an equivalence class exhibit the same weak real-time behavior.

□ Q.E.D

**Def 3.14.7** *Weak Real-time relevancy* An action $a_i$ within a trace $T_1$ of a program P, is said to be weak real-time relevant iff

- $a_i$ is a non real-time relevant action. OR
- $a_i$ is a entry action.

The theorem proved above demonstrates that if the sequence of weak real-time relevant actions in trace T1 is maintained in T2, then T1 and T2 will exhibit the same weak real-time behavior. Thus the above two actions are **sufficient** for reproducing not only the non real-time behavior of all objects but also the timing behavior of relevant objects, except for operating system interference. Note that since the replay is being conducted in a simulated environment during phase II, the issue of operating system interference does not arise.

**Def 3.14.8 (Strong-Annotated Trace)** The strong-annotated trace $T^{SA}$ of a trace T=$a_1 \ldots a_n$ of a program P, is obtained by applying the following transformation on the sequence described by trace T

- transform $a_1$ to the tuple $< a_1, 0 >$
- if $a_{i-1}$ is transformed to the tuple $< a_{i-1}, t >$, then $a_i$ is transformed to the tuple $< a_i, t+1 >$.

The strong-annotated trace $T^{SA}$ is the sequence $< a_1, 0 > .. < a_i, t_i > .. < a_n, t_n >$ with the annotations derived according to the transformation described above.

Basically, the idea behind is as follows. The tuple $< a_i, t >$ indicates that action $a_i$ was executed at logical time t, where the logical time starts out as 0 and is incremented by 1 on the execution of each instruction.

**Def 3.14.9 (Strong Real-time behavior)** The notion of Strong Real-time is defined ONLY for uni-processors. The strong real-time behavior of an object is a sequence of value-time tuples of the form $< v_i, t_i >$, where $t_i$ denotes the time at which the value $v_i$ (of the object) is read/written. $t_i$ is the logical time (local time) that is computed according to the transformation rule defined for deriving the strong-annotated trace from T. The strong real-time behavior of a program P with trace T is characterized by the strong real-time behavior of every object of the program under the strong-annotated trace $T^{SA}$.

**Theorem 3.14.3** If $T_1 = a_1..a_n$ and $T_2 = b_1 b_2..b_n$ are two traces of a program P executed under identical input conditions and if

1 $a_1 = b_1$

2 $a_i$ is a TTP in $T_1$, iff $b_i$ is a TTP in $T_2$.

3 If $a_i$ is a TTP, then $a_{i+1}$ and $b_{i+1}$ belong to the same thread of control in P.

then

- $a_i = b_i$, $\forall$ i, $1 \leq i \leq n$

Proof: By induction

**Basis:** $a_1$=$b_1$, By assumption.

**Hypothesis:** If $a_i$=$b_i$ $\forall$ i such that $1 \leq i \leq k$, then $a_{k+1} = b_{k+1}$

Proof of Hypothesis: There are three cases for $a_k$ and $b_k$.

**Case I:** $a_k$ is neither a entry action nor a TTP.

Since $a_k$=$b_k$ and $a_k$ is not a TTP, $a_{k+1}$ and $b_{k+1}$ must belong to the same thread-of-control. Again since $a_i$=$b_i$, $\forall$ i $\leq$ k and the input conditions are the same for both the traces, clearly, $a_{k+1} = b_{k+1}$.

**Case II:** $a_k$ is a TTP.

Since $a_k$ is a TTP, by assumption (2) above we conclude that $b_k$ is also a TTP. Now, $a_{k+1}$ and $b_{k+1}$ belong to the same thread-of-control (By asumption 3), say $c_j$. Note that $\forall$ i, $1 \leq i \leq k$, $a_i$ and $b_i$ belong to the same thread of control (this directly follows from all the three assumptions above). This immediately implies that the number of instructions executed within $c_j$, before $a_{k+1}$ under $T_1$ is exactly equal to the number of instructions executed within $c_j$, before $b_{k+1}$ under $T_2$. Since the progress within $c_j$ up to $a_{k+1}$ in $T_1$ is identical to that up to $b_{k+1}$ in $T_2$ and the input conditions are the same (by assumption) for both traces, we conclude that $a_{k+1}=b_{k+1}$.

**Case III:** $a_k$ is a entry action.

Since $a_k$ is an entry action, $b_k$ is also an entry action ($a_k=b_k$, by hypothesis). $a_k$ and $b_k$ therefore, belong to the same thread-of-control, say $c_n$. Since $a_k$ is not a TTP, $a_{k+1}$ and $b_{k+1}$ belong to the same thread of control, $c_n$. Note that, $\forall$i, $1 \leq i \leq k$, $a_i$ and $b_i$ belong to the same thread of control (this directly follows from all the three assumptions listed above). This implies that the number of instructions executed in $c_n$, before $a_{k+1}$ in $T_1$ is exactly equal to the number of instructions executed before $b_{k+1}$ in $T_2$. Since the progress within $c_n$ up to $a_{k+1}$ in $T_1$ is identical to that up to $b_{k+1}$ in $T_2$ and the input conditions are the same for both traces (by assumption), we conclude that $a_{k+1}=b_{k+1}$.

Thus we conclude that the hypothesis is true.

Since $a_1 = b_1$ and the hypothesis is true, we conclude that $a_i = b_i$, $\forall$ i. i.e, the two traces are identical.

□ Q.E.D

## 3.15 Interrupts

In this section we shall discuss the issue of interrupts with respect to relevancy, i.e, How are interrupts treated? Are they relevant? How does one reproduce an interrupt during replay?

An interrupt is an asynchronous event used by various entities in the system to attract the attention of the processor. Usually, a processor does not acknowledge an interrupt until the currently executing atomic instruction is completed. After completing the current atomic instruction, the processor acknowledges the interrupt (usually by sending a signal to the appropriate device indicating acceptance of interrupt) and performs the following steps

- Save the program counter and the program status word of the currently executing thread-of-control. The program counter indicates the address to which the processor must return to after executing the interrupt service routine (ISR). The program status word contains the status of condition flags just before the processor acknowledges acceptance of interrupt.
- Load the address of the interrupt service routine into the program counter.
- Execute the instructions in the ISR.
- After completing the ISR, restore the old values of PC and PSW from the stack.
- Resume execution of interrupted thread-of-control.

Note that in the above steps, while switching the context from the target program to the ISR, the state of the registers was not saved. Instructions within the ISR save and restore the registers that the ISR needs to use. However, while coding the ISR, the programmer could forget to save and restore register values upon entry and exit from the ISR. Consequently, actions within the ISR could overwrite registers used by the target program. Clearly, in such a case, the behavior of the program depends on where in the program the interrupt occurs (because different locations within the target program will have different register values). This could be a very nasty bug to detect, because the behavior may not be repeatable until and unless the interrupt occurs at exactly the same instruction with respect to the target program.

The problem discussed above occurs because the ISR could potentially share registers with the interrupted thread-of-control. The following are some of the solutions

**Solution 1** According to the theorem 3.12.1, all instructions that operate over registers that are used by the ISR are relevant (since registers are being shared). Note that this increases the number relevant instructions significantly.

**Solution 2** An alternate solution can be devised by first observing that the execution of the ISR is atomic with respect to the interrupted thread-of-control i.e, execution of instructions within the ISR are not interleaved with those of the interrupted thread-of-control. In other words, if we can identify the instruction within the interrupted thread at which the interrupt was accepted, then, during replay, we can recreate the same behavior by transferring control to the ISR immediately after the execution of the instruction identified above. The problem with this solution is that identifying the instruction in the interrupted thread, at which an interrupt is accepted, is non trivial, as we shall see in chapter 5.

**Solution 3** Yet another solution could be to stipulate that all registers be saved and restored upon entry and exit from the ISR (this could be achieved by forcing the hardware to save all the registers upon the occurrence of an interrupt). In effect then, the ISR would communicate with the interrupted thread via shared memory only. The disadvantage being the increase in response time because of the need to save and restore all registers. With such an assumption, it is not necessary to reproduce the interrupt at the same instruction within the interrupted thread, during replay. In other words no special treatment needs to be given to interrupts. An ISR is treated just like any other independent thread of control. Finally, adopting this solution would require hardware support to enforce the saving of all registers upon each context switch.

Clearly, solution 1 is not acceptable since that would force every register access to be relevant and as such would be required to be logged. If solution 2 above is adopted, then, there is a need to identify the instruction at which the interrupt is accepted. In the next section, we shall examine in greater detail, how an interrupt can be associated with the address of instruction after which it was accepted. Solution 3 on the other hand, is restrictive in the sense that it forces all registers to be saved and restored upon entry and exit from the ISR. However, the potential advantages of such a restriction are in simplified hardware necessary to monitor interrupts, as will be seen in later chapters.

### 3.15.1 Identification of Interrupt location

Let an interrupt be accepted by the processor after executing action $a_i$. Action $a_i$ is of the form $m_j^k$ for some integer value of k. Basically this signifies that $a_i$ is an action resulting out of the k-th execution of the machine instruction $m_j$. Identifying an interrupt with $a_i$, therefore means that during a particular execution, an interrupt was accepted after the k-th execution of the machine instruction $m_j$. During replay, therefore, we must be able to trigger the same interrupt after the k-th execution of $m_j$. This requires that we keep track of k (execution count) for each instruction and the memory address corresponding to the instruction. Note that an instruction will be executed more than once only if a branch to a previous address (logical address) is made. Hence the value of k will automatically be reproduced if we mark all *backward branches* as a relevant. The interrupt itself is identified with $m_j$. If the interrupt occurs after the k-th execution of $m_j$, then in the relevant event sequence, the interrupt will be logged as occurring after k backward branches. Obtaining the address of $m_j$ at run-time will be discussed in chapter 5.

## 3.16 Sufficient conditions for Relevancy

### 3.16.1 Non Real-time behavior

As proven in theorem 3.12.1, it is **sufficient** to monitor for the relative execution sequence of the following instructions, in order to reproduce an identical non real-time behavior during the second phase:

- **Shared actions:** All instructions operating (read/write) over shared memory locations are relevant. This is evident from the definition of relevancy, an action is relevant if there is a possibility of another action occurring, either before or after it, and the relative order modifies the non real-time behavior of the program. Clearly all actions (read or write) over shared memory are therefore relevant because their relative order determines the behavior of the shared object.

- **input/output actions:** As proven above, one of the conditions for reproducing identical behavior is that the program be executed under identical input conditions. This immediately implies that all input actions, i.e, actions resulting in an input from the environment must necessarily be reproducible. Here, not only the relative order of the input actions but also the values received from the environment by the action is important. The values received from the environment may determine the course of events within the process and hence across processes. These values, therefore, must be trapped and reproduced during subsequent executions for obtaining the same behavior.

- **Interrupts:** There are two possibilities here depending on how the interrupt problem described in a previous section is solved.

  - If the ISR shares registers with the interrupted thread-of-control, then the following are relevant instructions:

    1. Instruction at which the interrupt was accepted.
    2. All instructions that branch backward.

  - If the ISR does not share registers with the interrupted thread, then the first instruction within the ISR is relevant.

### 3.16.2 Weak Real-time Behavior

As proven in theorem 3.14.1, monitoring for the relative execution sequence of the following instructions is sufficient to reproduce an identical weak real-time behavior during the second phase:

- All instructions that are relevant for non real-time behavior.
- The first instruction (of the target program) after a context switch.

### 3.16.3 Strong Real-time behavior

As proven in theorem 3.14.3, monitoring for the relative execution sequence of the following instructions, is sufficient to reproduce an identical strong real-time behavior during the second phase:

- Interrupts: Instructions at which interrupts are accepted(both timer and external interrupts).
- First instruction (of the target program) after each context switch.
- input/output instructions:

Note, that the above instructions are simpler (than those required to for Non real-time and Weak real-time behaviors) to monitor. An obvious question then is, "If the instructions required for monitoring a Strong real-time behavior are simpler and Strong real-time behavior class subsumes the other behavior classes, then why bother about the other behavior classes?". We will answer this question in the next section.

## 3.17 Behavior Class-Relevancy Relationship

In the previous sections we categorized behavior into 3 different categories, namely, non real-time, weak real-time and strong real-time behavior. In this section we shall discuss the results obtained (in terms of relevancy) for each of these behavior classes and explain the advantages of categorizing them as above.

- **Non Real-time:**

  Non real-time behavior of a program characterizes the non real-time behavior of its objects. Correctness of the behavior of an object is judged solely by the sequence of values attained by the object without any regard to the times at which the values were obtained. Then we introduced the notion of relevancy, from the point of view of reproducing the behavior of the program. Note that the replay phase produces a trace belonging to the Power-trace of the program P (see definition in 3.12). There are a number of traces within Power-trace(P) that are non real-time behaviorally equivalent. We defined a relation between traces, where a trace $T_1$ is related to trace $T_2$ if the sequence of relevant instructions in $T_1$ is identical to the sequence in $T_2$. This is an equivalence relation and therefore divides the traces in Power-trace(P) into a number of equivalence classes. All traces within an equivalence class exhibit the same non real-time behavior. All of them share the common property that the sequence of relevant instructions in each of the traces is identical to that of any other trace in the equivalence class. This is an important result, in that, the replay phase does not have to ensure an identical trace in order to achieve identical behavior. More clearly, the replay can follow any of the traces belonging to the equivalence class that is uniquely identified by the original execution. The implications of this result as far as monitoring the program are concerned, are that during phase I, it is not necessary to monitor the relative sequence of every machine instruction. In fact, it is **sufficient** to monitor the relative sequence of relevant instructions within the original trace. If the replay can guarantee an identical sequence of relevant instructions, then one can achieve non real-time behavioral reproducibility.

Note that the replay is performed in simulated time, i.e, in the original execution, if the relevant instruction $r_1$ was executed at time $t_1$, then in the replay, the programmer can determine that fact. In other words, during phase I, the absolute value of time at which each relevant instruction was executed, is trapped (to a certain degree of accuracy). So, if a certain deadline is missed, the programmer will know that the deadline was missed in phase I. However, he/she may not be able to determine the reason as to why the deadline was missed. This is because, the relative time between two relevant instructions is not maintained between the original execution and the replay (only the relative sequence of relevant instructions). Therefore, there is a need to define a different behavior class to capture the relative timing between relevant instructions.

- **Weak real-time:**

  We defined the notion of weak real-time to address the issue of relative timing discussed above. Weak real-time behavior of a program is determined by the weak real-time behavior of relevant objects. Two traces are weak real-time behaviorally identically if the sequence of non real-time relevant instructions is the same in both of them *and* the process-level scheduling sequence between any two relevant instructions is also the same in both of them. The idea here is to close the possible gap of timing between relevant instructions in the original execution and the corresponding ones in the replay. The reason for calling this weak real-time is that this definition still does not close the gap completely. More clearly, while the process-level schedule between two relevant instructions is maintained (between the original execution and the replay), the instruction-level scheduling sequence between two relevant instructions is not maintained.

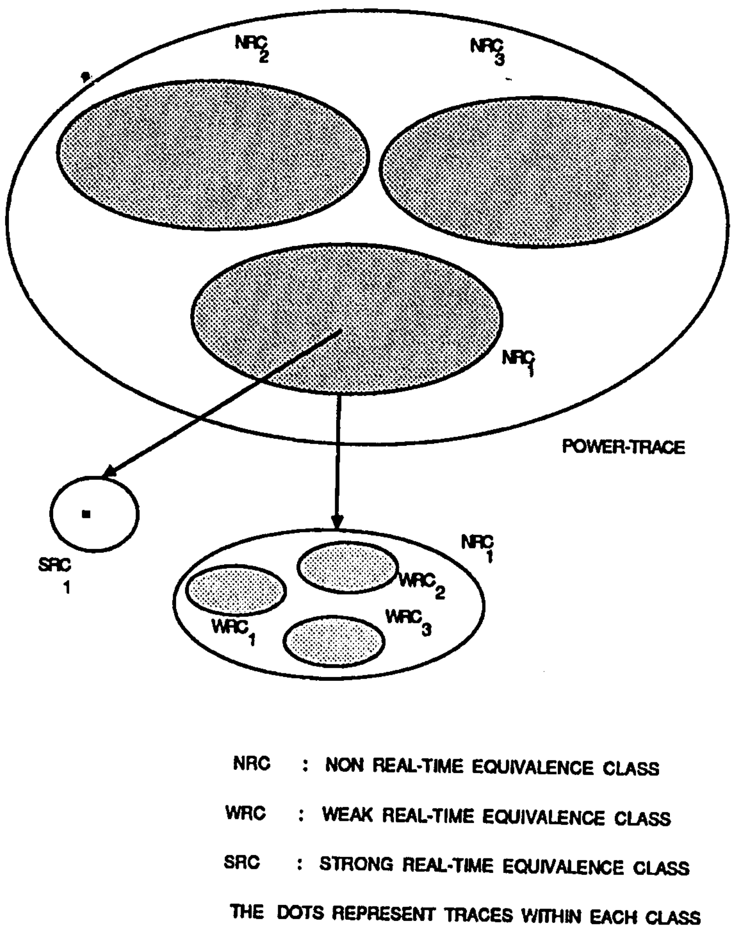

NRC2
NRC3
NRC1
POWER-TRACE
SRC1
NRC1
WRC2
WRC1
WRC3
NRC : NON REAL-TIME EQUIVALENCE CLASS
WRC : WEAK REAL-TIME EQUIVALENCE CLASS
SRC : STRONG REAL-TIME EQUIVALENCE CLASS
THE DOTS REPRESENT TRACES WITHIN EACH CLASS

Figure 3.1. Structure of Behavior classes

Each non real-time equivalence class is further subdivided into a number of weak real-time equivalence classes. Refer fig 3.1. Any two traces within the same weak real-time equivalence class must have an identical sequence of weak real-time relevant instructions.

From the point of view of replay, it is more advantageous for the programmer to work with a weak real-time trace since he has a better chance of identifying the cause for a missed deadline (compared to a non real-time equivalent trace). However, it is still possible that the programmer may not be able to identify the cause for a missed deadline. This is because the missed deadline could be due to a class II scheduling error. Recall from section 3.4, that a class-II scheduling error arises out of an incorrect estimation of $c_i$ in a ($c_i$, $p_i$, $d_i$) triple used to characterize a periodic process. Weak real-time only guarantees that the number of context switches between any two non real-time relevant instructions remains the same between the original execution and the replay. It does not guarantee that the number of instructions executed between any two relevant instructions is the same between the original execution and the replay. Therefore, to identify the cause of a class-II scheduling error, one needs to reproduce a strong real-time behavior during replay.

- **Strong real-time:**

  The notion of strong real-time behavior was defined to further close the timing gap between two relevant instructions in the original execution when compared to that in the replay. Here, two traces are said to be strong real-time behaviorally equivalent if and only if they are totally identical with respect to the sequence of every single machine instruction. In other words, not only is the sequence of relevant instructions the same, the instruction-level scheduling sequence between any two relevant instructions is also maintained (between the original execution and the replay). As an aside, it is interesting to note that the notions of weak and strong real-time coincide in a non-preemptive scheduling policy, where each task runs until completion before giving up the processor. In general, however, the notions of weak and strong real-time behavior are different. Each trace in the Power-trace(P) forms a strong real-time equivalence class on its own.

  From the point of view of replay, this is an ideal trace to work with. However, the tradeoff is in the amount of monitoring information needed to replay an identical trace (see section 3.16) and the complexity of the hardware support needed to collect the necessary information non-intrusively.

Finally see figure 3.1 for an overall diagram of the various behavior classes and their relationship. Note here that the notion of strong real-time behavior is strictly for uni-processors. In a multi-processor environment, there is no unified model for describing the timing behavior of an object because the execution is distributed across different processors which have different clocks. The problem boils down to the classic distributed clock synchronization problem. Also there is no centralized concept of state since the objects of the program are distributed between main memory and on the various processors (in registers). On the other hand, weak real-time behavior (as we have defined it) extends very easily to a multi-processor case. Weak real-time behavior, as opposed to strong real-time behavior, characterizes the timing behavior of only relevant objects. In a shared memory multi-processor, the relevant objects are all located in main memory (assuming no caching, or only a write through cache) and therefore the timing behavior of these objects is centralized. The concept of weak real-time behavior is therefore more general than strong real-time in that it is extendable to multi-processors more easily.

*If a strong real-time behavior subsumes all other forms of behavior, then why have the notions of non real-time and weak real-time behavior classes (Note that this question is valid only within the framework of a uni-processor)?*. The answer to this question lies in the hardware support necessary to collect the information corresponding to a strong real-time behavior. Identifying the address of the instruction at which an interrupt is accepted (TTP), requires the monitoring hardware to follow (not log) every instruction (relevant or not). While this may be possible in non pipelined architectures, it poses a major problem in pipelined architectures where the time between the execution of two instructions is smaller than the time it takes the target processor to inform the monitoring hardware about the execution of an instruction (these aspects will be clarified in chapter 5). In summary, although the amount of information needed to reproduce a strong real-time behavior is less than or at least comparable to that of a weak real-time behavior, the corresponding hardware support needed to collect such information is more complicated in the case of a strong real-time behavior.

## 3.18 Monitoring Relevant Instructions

Relevant instructions described in previous sections, can be grouped into two different classes depending on their monitorability.

- Static: Those instructions which can be determined as relevant at compile time. Examples of these are instructions over shared memory, input and output instructions and backward branches (see section 3.15). The compiler can identify and tag these instructions as relevant so that their sequence can be monitored while they are being fetched from memory over the bus (this aspect is treated in a separate chapter).
- Dynamic: Not all instructions can be determined as relevant at compile time itself. Certain TTPs, for example those arising out of external interrupts, cannot be determined at compile time because they are asynchronous events (i.e, their occurrence is not synchronized with any particular instruction execution). Their occurrence is determined at run-time. The details of run-time detection of relevant instructions will be discussed in chapter 5.

### 3.18.1 Compiler Support

We propose compiler support for identifying static relevant instructions. Having the compiler detect relevancy, shifts the burden from the monitoring hardware that would otherwise have to do it. Apart from the severe timing constraint that would have to be satisfied if the detection were done by the monitoring hardware, the monitoring hardware may not always have enough information about whether a particular instruction is relevant. We shall therefore assume that the compiler detects all static relevant instructions and tags them appropriately by setting a bit in the opcode which is specially reserved for indicating relevancy.

The problem of finding and tagging shared relevant instructions reduces to the problem of detecting whether a particular object is shared. Detection of shareable objects, in general is a language issue. In particular, it depends on whether the language requires explicit declaration of shared objects. Objects can be classified into one of three categories. They are:

- Private: These are objects that are local to a particular thread of control and are accessible only by instructions belonging to that thread of control.
- Static Shared: These are objects that are accessible to multiple threads of control by declaration i.e, these objects are determinable at compile time itself.
- Dynamic Shared: Certain objects within the program may not be shared by declaration but could achieve the property at run-time. For example if a statically shared object is passed as a parameter to another object (by reference), then the new object (i.e, a syntactically new object) inherits the shared property. Detection of these objects might require that the compiler perform a static flow analysis of the program. A possible implementation could be to keep track of whether a variable is shared or not in the symbol table entry for that variable. The property (of sharing) is propagated with each assignment to a shared variable.

In summary, it is possible for the compiler to classify objects into one of the above three categories. Note that an object classified as being dynamically shared may not actually be shared at run-time (e.g, if the function passing the statically shared object as a reference parameter is not executed). The compiler would still have to tag these instructions as relevant and the monitoring hardware will have to incur the overhead of tracing their sequence.

## 3.19 Phase II: The Replay Phase

We have, until now, discussed the issues of monitoring a program in terms of *Where to perform monitoring?*, *What instructions to monitor?*, *What information to collect corresponding to each relevant instruction?*. In this section we shall concentrate on issues involved in replaying the program. Note that the idea behind replaying the program is to provide the user with a behaviorally identical execution of the program, to detect the error. We have defined three different classes of behavior namely, non real-time, Weak real-time and strong real-time. Each of these behavior classes has a different set of relevant instructions. The programmer has to choose the behavior class he desires, before beginning phase I. The replay is then performed in simulated time.

### 3.19.1 The Replay Scheme

If P is the program that has been executed during phase I with input X, then for replaying the program during phase II, we modify the program P to a new program P' and execute P' with input (X,E) where E is the event table that was constructed during phase I. The event table specifies the sequence of relevant instructions along with temporal (time at which the relevant instruction was executed) and environmental information (external input values, if any, consumed by the relevant instruction) for reproducing the appropriate behavior. The modifications to the program P are as follows. For each entry in the event table, do the following:

**Step1** Extract the virtual address of the instruction corresponding to this entry.

**Step2** Locate the instruction corresponding to this address in the target program P.

**Step3** If the current entry is NOT an interrupt entry, then insert an unconditional branch instruction with M (a Monitor) as target, BEFORE the instruction (located in step 2 above).

Step4 If the current entry is an interrupt entry, then insert the unconditional branch instruction to a Monitor, AFTER the instruction (located in step 2 above). If A is the address corresponding to an interrupt entry in the event table, then it means that during phase I, the interrupt was accepted AFTER the execution of the instruction located at address A. Therefore, the branch instruction to the monitor in program P', is located AFTER the instruction at address A.

In the replay phase, program P' with the above modifications, is executed under the control of a Monitor M. The purpose of the monitor is to do the following

- Schedule the processes within P', so as to achieve a behaviorally identical trace (depending on which behavior class was chosen during phase I).
- Ensure that all input instructions of P' receive the same input values as the corresponding instructions in P.
- Provide the user with most of the usual sequential debugging features like single stepping, breakpointing, checking (NOT modifying) program variables. Note that modifying can be allowed if the programmer realizes that he/she is no longer guaranteed the same behavior.

The scheduling policy of the monitor M is as follows:

- Read the next entry in the event table.
- Determine the thread-of-control in which this entry lies (from the virtual address). Let it lie in process $p_i$.
- Check if this entry is an input instruction.
- If it is an input instruction, move the actual value field in this entry into the appropriate port.
- Save the state of the current process and perform a context switch to process $p_i$.

### 3.19.2 Correctness

Strictly speaking, theorem 3.12.1 is not applicable because the traces generated during phase 1 and phase 2 are not from the same program. During the replay we execute a modified program. The modification consists of a few extra branch instructions inserted into the target program at appropriate locations as described above. However, the Monitor does not modify any of the program objects, so we contend that the two traces are indeed comparable and that the theorem is applicable to them

**Theorem 3.19.1** Let T be the trace of an execution of program P and T' be the trace of P', where P' is obtained by modifying P as described in the section above. If X is a an object of program P, then

- The non real-time behavior of X under P' is identical to the non real-time behavior of X under P.
- The weak real-time behavior of X under P' is identical to the weak real-time behavior of X under P.
- The strong real-time behavior of X under P' is identical to the strong real-time behavior of X under P.

Proof:

Note that if $a_i$ is an action that is executed under trace T of program P, then it is also an action that is executed under trace T' of program P'.

However, there exists actions under T' of P' that are not executed under T of program P. These are the branch instructions that are inserted into P and the instructions executed within the monitor M (see section above). Let us call these instructions as *auxiliary actions*. Notice that according to the replay algorithm, none of the auxiliary actions modify any of the objects in P. All of them only read the states of objects in P. As a direct consequence, the non real-time behavior of the objects in P' remains identical to that under P.

Note again, that P' is executed in simulated time. Time is used as an input parameter during the replay. Since none of the actions in P' consume any real time, the simulated real-time behavior (strong or weak) of X under T' is the same as that under T.

Since the behavior of every object in P under trace T is identical to that in P' under trace T', we conclude the replay-scheme correctly replays the behavior of P.

□ Q.E.D

## 3.20 Assumptions

Now that we have established the foundations for our two-phase strategy of debugging, here are a number of general assumptions that we will be making in subsequent chapters. More specific assumptions regarding the architecture of machines that we will be monitoring will be described in chapter 5.

- The target program consists of a set of processes, executing on a single processor and communicating via shared memory.
- The instructions within each of the processes are non-modifiable machine instructions that can only be executed.
- The debugging activity is restricted to bugs within the target program i.e, we will assume that the underlying run-time system is free from bugs.
- Execution of all ISRs is atomic with respect to the target program. However, execution of the ISR may not be atomic with respect to other interrupts which may occur while the current ISR is executing.
- The ISR communicates with the processes of the target program only through shared memory i.e, no information is passed via registers. See section 3.15 for a more detailed discussion of why this assumption is being made.

## 3.21 Information To Be collected

In this section we shall describe the type of information that needs to be collected corresponding to each relevant action for subsequent replay. The *Event table*, in which the monitoring information is logged during phase I, is a hardware buffer that is controlled by the monitoring hardware. The format of each entry in the event table depends on the type of relevant action corresponding to that entry. Before discussing the format of each type of entry, we will describe the generic information that needs to be captured for any type of relevant action. The generic information consists of two components:

- Identity of the action: It is necessary to identify an action so that the program can be modified for the replay phase (see section 3.19.1 for details). Identification of an action in itself requires two components, namely, the address of the machine instruction and the execution count of the machine instruction which the above action represents. There are certain actions which are relevant because of a special property of the corresponding machine instruction (e.g, shared, input, output actions). All executions of such instructions are therefore relevant. Identification of such actions requires only one component, i.e, the address of the corresponding machine instruction (the other component, i.e, the execution count, is implicitly provided in their relative position in the event table).

  Certain actions, however, are relevant not because of any special property held by their corresponding machine instructions, e.g, TTPs. Identification of a TTP requires both components, i.e, the address of the instruction and the execution count representing the TTP. One of ways of keeping such a count, without perturbing the target program, is by forcing all *backward branches* to be relevant. By doing so, the position of a TTP, with respect to backward branches, within the event table will implicitly provide the execution count corresponding to a

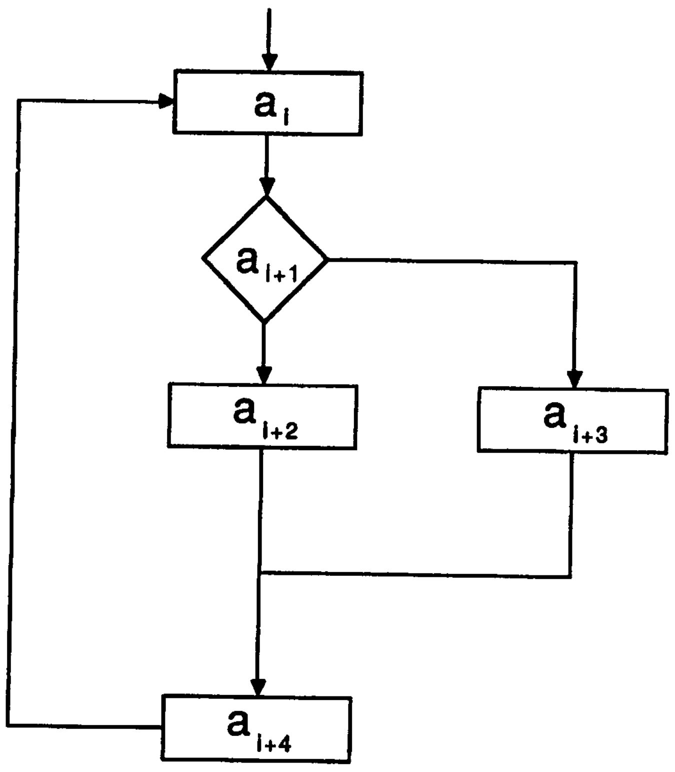


Figure 3.2. Program Flow Graph

TTP.

- Environment information: Since the second phase is being executed in a simulated environment, it is necessary to collect environment dependent information for certain types of actions, e.g, the input values read from the environment corresponding to an input action.

**Theorem 3.21.1** Consider the the flow of control of a program, as shown in figure 3.2. In order to identify an external interrupt, say E, with an action, it is sufficient (for purposes of recreating the environment for the interrupt during replay), to capture the following information during the first phase

- The address of the instruction corresponding to the action at which the external Interrupt was fielded.

- A count of the number of backward branches executed before the external interrupt E.

Proof: In figure 3.2, $a_{i+1}$ is a test instruction that checks some condition. $a_{i+2}$ and $a_{i+3}$ are executed on different branches depending on the outcome of the test. $a_{i+4}$ denotes the action that branches back to the instruction address corresponding to the action $a_i$.

Consider the set of all sequences of actions accepted by the control flow graph shown in figure 3.2. It can be represented by the following regular expression, namely, $(a_i a_{i+1}(a_{i+2}|a_{i+3})a_{i+4})^*$. If we consider the sequence of actions from $a_i$ to $a_{i+4}$ as a **frame**, the execution trace of the flow-graph will contain a sequence of frames. Each of the frames need not be identical, since in some frames the conditional test $a_{i+1}$ may turn out to be true and in others it may turn out to be false. However, $a_i$ and $a_{i+4}$ will always be the first and last actions in every frame. It is clear therefore that the number of frames in any execution is equal to the number of times $a_{i+4}$ is executed. It may also be noted that corresponding frames between the original execution and the replay will be identical because the replay is being executed under identical input conditions.

Now consider an interrupt, E, that is fielded at $a_{i+2}$. During replay, interrupt E has to be reproduced at exactly the same instruction and also within the same frame number (as the one in which it was fielded in phase I). As discussed above, the frame number is given by the number of times $a_{i+4}$ is executed before the fielding of E. If $a_{i+4}$ is designated as *relevant*, then the relative position of E with respect to $a_{i+4}$ will implicitly give the frame number in which E was fielded during phase I.

If $a_{i+4}$ is forced to be relevant, then it is sufficient to trap only the address of the instruction at which the external interrupt was registered. □

We list below each type of relevant action and and the specific environment information that needs to be trapped for the corresponding relevant action.

- Shared instructions: Virtual address of instruction. Note that by assumption the MMU (Memory management Unit) is external to the CPU chip. This enables us to trap the virtual address of an instruction.

- Input Instruction: Virtual address of the instruction and the value of the input read when the instruction is executed. Also the absolute value of time when this instruction was executed.

- Output Instruction: Virtual address of instruction and the absolute value of time when this instruction is executed. An output instruction is the means by which a real-time program communicates its results to the external world. The time stamp is necessary so that the programmer can check if a deadline has been missed.

- TTP: Virtual address of instruction and the absolute value of time when this instruction is executed. This gives the time at which each new task has been scheduled.

- Backward branches: Virtual address of instruction.

- Interrupts: Virtual address of instruction at which the interrupt occurred and the absolute value of time.

The overall structure and mechanism of logging the above information corresponding to each relevant instruction, will be discussed in greater detail in chapter 5.

# CHAPTER 4

# CHECKPOINTING

## 4.1 Motivation

The two phase scheme outlined in the previous chapter, provides the user with an ability to monitor and subsequently replay an appropriate behavior (non real-time, weak or strong real-time) of the program. The replay phase is faithful only if the sequence of **relevant instructions** is maintained from the beginning of the program. Since the relevant instructions are at such a low level of granularity, the number of such instructions being executed can be relatively large. Since available memory is finite, it is not possible to store the relevant instruction sequence for arbitrarily long lengths of time. If available memory can support the storing of relevant instructions for t units of time, then the two-phase scheme is useful for replaying programs that execute for no more than t units of time. Clearly, this is not acceptable because the effects of a timing error may be realized long (probably more than t units) after the occurrence of the error itself. What is required therefore is a method for replaying the program, without having to start from the beginning of program execution. More clearly, there is a need for checkpointing the state of the program at intermediate points.

The two-phase strategy when augmented with checkpointing scheme will provide the user with an ability to rollback the state of the program to a previous point and execute forward from there (ensuring a behaviorally identical execution). This overall strategy eliminates the need to maintain the relevant instruction sequence from the beginning of the program. Instead we maintain the relevant instruction sequence for the period of time between two checkpoints (called a checkpoint frame). Whenever an error is detected, the program is rolled back to a previous checkpoint and the relevant instruction sequence corresponding to this checkpoint frame is used to replay the program forward. If the error still cannot be detected then the program is rolled back by two checkpoint frames and executed forward from there.

The challenge in devising a checkpoint scheme will be to do it non-intrusively, because providing non-intrusive debugging is an overriding concern in this thesis. Some of the issues we will be looking at are

- Is it possible to capture the state of the whole program in a totally non-interfering manner?
- If so, what is the nature of the hardware/software support necessary to accomplish the above task?.
- How does one get a handle on this huge information base?
- Is checkpointing done process-wise or processor-wise?

## 4.2 Literature Survey

The problem of checkpointing is quite well-known and occurs quite frequently in such areas as fault-tolerance, debugging etc. Checkpointing is necessary in fault-tolerant systems in order to recover from failures. There are a number of factors like time to recover from a failure, cost of the system for providing the checkpoint facility, overhead incurred by the checkpointing algorithm etc. Different checkpoint schemes attempt to optimize one of these factors. In this thesis we will primarily be interested in checkpointing schemes that have negligible overhead and therefore satisfy the non-intrusive criteria discussed in the last section. Most of the literature in checkpointing techniques has been done in the area of fault-tolerance and recovery. As a consequence we will review some of those schemes.

Fault-tolerant applications can broadly be classified as belonging to one of two domains:

- Process Control: Here, the application is organized as a set of concurrent or distributed processes that communicate either through shared memory or via message passing. This type of model is typically used for real-time processing. Since real-time constraints typically require immediate recovery, the processes are structured so as to achieve considerable redundancy so that failures can be masked as soon as they occur. Providing computational redundancy requires increased computational resources to replicate and execute a number of identical tasks in parallel. Increased system cost is offset by a reduced failure recovery time.

- Transaction Processing: Here, the application is divided into units of work called *transactions*. The system guarantees the atomicity, serializability and permanence properties for each transaction. Atomicity guarantees that each transaction has an all-or-nothing execution semantics, i.e, a failure never allows the intermediate states of a transaction to be visible to other transactions. Usually, transaction processing systems have less severe deadlines and so the system cost for failure recovery can be reduced in favor of increased recovery time.

By definition, transaction boundaries always define consistent system states from which a computation can recover from. One of the drawbacks of the transaction model is that programs must explicitly use the transaction paradigm by announcing the beginning and end of the transaction.

In a system in which computations interact by exchanging messages or sharing data, rolling back a failed computation to an arbitrary checkpoint may result in an inconsistent global system state. Rollback should never result in a system state in which there are computations that appear to have received (or read data values) values that have not yet been sent (or written). There are three distinct strategies by which global system state consistency is guaranteed. They are:

- Application-Specific Recovery : In application-specific recovery, recovery is explicitly programmed as part of the application [SY85], [SGMW84]. In such cases, the recovery code usually involves intimate knowledge of the application domain and the underlying hardware.

- Optimistic Recovery: This strategy optimistically assumes that failures will not occur. If they don't then everything is fine. However, the system collects enough information along the way to roll back those computations necessary to establish a consistent system state. Strom and Yemini in [SY85], propose a scheme that allows checkpointing and message logging to stable storage to occur asynchronously with respect to computations but maintains the causality information necessary to recover to a globally consistent state. Johnson and Zwaenepoel propose a scheme where messages are logged in the nonvolatile memory of the sender rather than the receiver resulting in an even greater asynchrony of stable storage writes with respect to computation.

  An unfortunate drawback of optimistic strategies is that recovery time is difficult to bound since in addition to the failed component an arbitrary number of other computations may have to be rolled back. This problem is more commonly known as the *domino effect*, [Ran75].

- Pessimistic Recovery: As an alternative to the above strategy, pessimistic strategy structures the checkpointing mechanism in such a way that recovery involves only those computations affected by the failure. Pessimistic schemes synchronize checkpointing with global interactions of computations. For example, if computations were to be checkpointed after each send operation, rolling back only the failed ones would guarantee global state consistency. Clearly then, the cost is shifted from recovery to checkpointing. To avoid substantial delays associated with checkpointing to stable storage, pessimistic schemes typically contain checkpoints in the context of a backup computation on another processor.

While there are numerous software checkpointing schemes that capture a consistent state of the program in a concurrent or distributed environment, most of them incur a considerable overhead in their implementation and are therefore not compatible with our overall goal of providing non-intrusive debugging. We will, therefore, review those techniques that are hardware oriented and are relatively less intrusive.

Among the techniques we will be reviewing are, the recovery block concept by Horning et. al which is a software-cum-hardware scheme proposed for writing fault-tolerant software, Demonic memory for reconstructing process histories, Time warp which is a special purpose chip used for implementing state saving and rollback functions and finally COPE, an integrated program development environment used for providing general recovery facilities in interactive systems. With the exception of Time Warp, all other techniques perform process-wise checkpointing.

### 4.2.1 Recovery Blocks

The concept of recovery blocks was first proposed by Horning et. al in 1974 and reprinted in [HLMSR85] in the context of writing fault-tolerant software. Recovery blocks are regions within the program from which it is possible to restore the state of the program to the one that existed when the block was entered, upon the occurrence of an error within the block. Recovery blocks provide for both error detection and recovery. Each recovery block consists of a conventional block which is provided with a means of error detection (an acceptance test), a primary program region and zero or more stand-by spares (the additional alternates). Upon entering a recovery block, the primary program is executed. After exiting the primary region, the acceptance test, which is a logical expression, is evaluated. If it evaluates to true then the corresponding recovery block is exited. If it evaluates to false (indicating an error), the state of the program is restored to that when the recovery block was entered and the next alternate region is executed. A further alternate, if one exists, is entered if the preceding alternate fails to complete or fails the acceptance test. However before an alternate is so entered, the state of the process is restored to that current just before entry to the primary region. If the last alternate fails to pass the acceptance test, then the entire recovery block is regarded as failed, so that the block in which it is embedded fails to complete and recovery is then attempted at that level. For further details of the recovery block concept and its implementation, we refer the reader to a host of papers [HLMSR85], [Ran75], [LGH80]. We shall concentrate on the implementation of their state restoration scheme.

Implementation of the state restoration scheme has been variously called "Recovery cache" or "Recursive cache" mechanism. Their overall strategy for backing up is as follows. Whenever a process has to be backed up, it is to the state it had reached just before entry to the primary alternate. Therefore, the only values that have to be restored are those of non-local variables that have been modified within the recovery block. Again, only the first change (since entering the recovery block) to the non-local variable need be saved. The recovery mechanism detects first time changes to non-local variables at run-time and saves the variable before modification into a recovery cache. Run-time detection is achieved by connecting the cache box (a hardware snoop that monitors the activity between the processor and memory) to the bus between the processor and main memory. It monitors the bus activity and converts all store "write-cycles" to "Read-Modify-write" cycles and caches prior value. Locality of a variable is determined by associating a recovery level index with each word in memory and maintaining the current recovery level in a separate machine register. If a word's recovery level matches the current recovery level then it is a local variable (local with respect to the current recovery block).

One of the drawbacks of the recovery block concept and hence the recovery cache mechanism, is that it is not easily extendable to multiple concurrent processes. Though [Ran75] has suggested the use of *conversations* (a structure that co-ordinates the recovery block structures of concurrent processes) as a mechanism for error recovery among interacting processes, they are very restrictive. However, our checkpoint scheme is more primitive and is not associated with a process (like recovery blocks). The checkpoint macro -instruction provides for global checkpointing of the program rather than process-wise checkpointing.

### 4.2.2 Demonic Memory

Demonic memory is a form of reconstructive memory for process histories proposed by Wilson and Moher. As a process executes, its states are periodically checkpointed, generating a history in discrete time. Following the initial generation, any prior state of the process can be reconstructed by starting from a checkpointed state and re-executing the process up through the desired state. If the original process execution depends on non-deterministic events like user input, these events are recorded in an exception-list and replayed at appropriate points during re-execution.

Demonic memory divides the process history into groups of consecutive steps called *chapters*. A state change record is created the first time a location is modified within a chapter. Subsequent changes to the location within the chapter are overwritten. Thus a state change record contains the last value within the chapter, since all intervening values are overwritten. The authors suggest that if sufficient locality of state changes is exhibited by the program, then the efficiency of demonic memory can be improved by operating on a memory page at a time. Each time a page is modified for the first time within a chapter, a new version of the page is created.

One of the main problems with demonic memory is that checkpointing is done on a process-wide basis rather than a processor-wide basis. It is similar to Feldman et. al [FB88] scheme of checkpointing single thread programs. Again demonic memory performs a lot of dynamic memory allocation (creation of new pages) which brings in a lot of unpredictability in timing and is therefore not suitable for real-time systems.

### 4.2.3 Time Warp

Time warp is a synchronization protocol distinguished by its reliance on lookahead-rollback. In a system using Time Warp each process executes without regard to whether there are any synchronization conflicts with other processes. Whenever a conflict is discovered, the offending process or processes are rolled back to the time just before the conflict and then re-executed along a revised path. The detection and the rollback are both transparent to the user. A detailed discussion of virtual time systems and the Time Warp mechanism itself can be obtained from [Jeff85].

Fujimoto et. al [FTG88] have built special purpose hardware for implementing the time warp mechanism. The chip was built as a special component in a special purpose discrete event simulation engine (the Utah Simulation Engine). The rollback chip implements state saving and rollback functions for a single processor in the simulation engine. Each simulation process is assigned a single data segment known as a version controlled memory. The chip supports 4 operations other than the traditional read and write, namely, Reset, Mark, Rollback(k) and Advance(k). The Reset operation just initializes the chip before use. The operation Mark, marks the current state of memory. Each marked state represents a state to which computation can be rolled back to at a later time. Rollback(k) restores the version controlled memory to the k-th previously marked state. Advance(k) removes the k oldest marked states. Each time a Mark operation is executed, the whole state of the version controlled memory is pushed onto a stack.

### 4.2.4 COPE

COPE (CO-operative Program development Environment) [AC81], [ACS84] is an integrated program development environment consisting of an interactive execution supervisor and a file system. It was intended to provide general recovery facilities in interactive systems, such as editors and program development environments. In these systems, the user generally is allowed to construct and modify objects (programs). COPE allowed the user to *Undo* the past actions on an object and restore it to its previous state.

COPE has a single file system that is used for every facility provided by the system. The user explicitly generates files for his program, input data and results. The system implicitly generates files for interactive screen input and output. Internally, all tables, stacks etc, used by various phases of the compiler, editor are treated as files. In other words, all system activity within COPE is accomplished exclusively through changes to some file. Files are implemented as a sequence of fixed-size blocks. Whenever an attempt to change a block is made, a complete new copy of a block is saved without overwriting the previous version. Since everything in the system is viewed as a file, the effect of executing a command is a sequence of changes to blocks. COPE represents these changes as a pair of block identifiers, naming the old and the new blocks. So, each command that is executed has an associated list of block changes. Maintenance of this list is transparent to the user.

Undoing a command is trivial. To Undo a command, the system restores the set of blocks changed by the command to their old values. The list of block changes corresponding to a command is maintained in a log. Since this log grows continuously as commands are executed, the system reclaims the space occupied by old versions of blocks changed by the oldest command, when it runs out of free space. The authors note that the block changes made while editing the program will be considerably fewer than when executing the program, which has the effect of reducing the recovery capacity during execution.

## 4.3 Issues

The state of a program at each processor site can be looked upon as a two-tuple, namely, the state of the program resident on the processor chip and the state resident in the main-memory. The state of a program, in general, consists of a data state and a control state. The data state consists of the values of all the data objects defined by the program, while the control state consists of the state of each of the threads of the program (the control state of a thread consists of the contents of the program counter, PC, and the program status word, PSW). The data state of a program is distributed between the processor chip and main memory. This is done largely to decrease the time to access a data object. Frequently used data objects are therefore stored in registers (or on chip data caches) to increase speed of execution.

The control state and part of the data state are therefore resident on the chip. Mathematically, let $s_i$(t) denote the state of the program, executing on processor site i, at time t. $s_i$(t) is a tuple $< onc_i(t), ofc_i(t) >$ where

- $onc_i$(t): This consists of the control state and part of the data state, as described above. This component of the state resides on the chip, in registers (or on-chip caches if any).

- $ofc_i$(t): This is the state of the component that is resident off the chip i.e either in the local or global memory or both, depending on the architecture of the distributed system.

Any checkpointing strategy has to checkpoint both $ofc_i$(t) and $onc_i$(t). In general $onc_i$(t) is not available on the pins of a cpu chip (that would blow up the number of pins on a chip to an impractical quantity). Checkpointing $onc_i$(t), therefore, requires the participation of the processor (by flushing this state to main memory). $ofc_i$(t) on the other hand, is accessible to external monitoring hardware and can be checkpointed without halting the processor (although it will require a change in the memory architecture as we shall see later). It appears, therefore, that non-intrusive checkpointing is impossible since $onc_i$(t) cannot be captured without the help of the processor. We solve this problem by including a macro-instruction (CHKPT) which is permanently embedded in the target program. The CHKPT instruction forces the $ofc_i$(t) to be flushed to memory to be captured by the monitoring hardware. Since the CHKPT instruction is permanently embedded into the target code, it does not contribute to interference (see definition of interference). Again, since $onc_i$(t) is a small fraction of the state of the program, the loss of efficiency due to the CHKPT instruction is negligible. We shall now discuss the issues involved in non-intrusive checkpointing of the $ofc_i$(t) component.

### 4.3.1 Constraints of Non Intrusive checkpointing

Before delving into our solution of how to perform non-intrusive checkpointing, the constraints that any solution should follow are:

- Existence of auxiliary processor: Any non-intrusive scheme for checkpointing has to have an auxiliary processor that performs the checkpointing, since the target processor cannot be involved in it.

- Dual-ported memory: The auxiliary processor references the memory for checkpointing while the target processor references it for computation. Since the memory is being shared between the two processors, there is a potential for conflict and hence intrusion. To minimize intrusion, the memory must be dual-ported. Of course, the possibility of contention still exists but that can be resolved by giving the target processor higher priority over the auxiliary checkpoint processor.

- Exclusive communication paths: Again, since the main memory is being shared between the auxiliary and target processors, the communication paths from the processors to the memory must be distinct (otherwise there is a possibility of intrusion through cycle stealing).

It is interesting to note that a checkpointing scheme that satisfies all the above criteria could still be intrusive. The possibility of intrusion arises because the two processors share the individual locations within main memory. More clearly if both the processors want conflicting accesses (one for read and the other for write or both for write), to that same location then there is a possibility of increasing the time to access the location and hence perturbing the behavior of the target program. What is, therefore, subtly implied by this problem is that the domain of main-memory that the auxiliary processor looks at be different from that of the target processor i.e, there must be a duplication of locations in memory. Note again that the two domains belonging to the auxiliary and target processors must be related in some way because the auxiliary processor has to checkpoint the state seen by the target processor. Hardware support is therefore necessary to transmit values from one domain to the other to be checkpointed.

## 4.4 Domain Interference

Let $X_T$ be the value of location X, as seen by the target processor and $X_A$ be the value of the same location (in a semantic sense) as seen by the auxiliary processor. Note that

$X_A$ and $X_T$ are two physically different entities representing the same resource X. $X_A$ and $X_T$ must be kept physically disjoint in order to eliminate the interference caused due to accesses to X by the auxiliary processor (for checkpointing) precisely when the target processor needs access to X (for computation). Apart from the interference aspect, there is a need to define when $X_T$ and $X_A$ synchronize with each other. *Are $X_T$ and $X_A$ always synchronized?, Is there an associated cost for keeping them in synchrony at all time?, Can the synchronization be performed less frequently?, What are the tradeoffs involved in alternative synchronization schemes?.*

## 4.4.1 Synchronization Schemes

- Global Synchronization: In this scheme the elements of the domains accessible by the target processor and the auxiliary processor are not synchronized between two checkpoints. However, when a checkpoint is initiated (on the execution of a special CHKPT machine instruction by the target processor), the complete target domain is copied into the auxiliary domain. The copying is done by hardware in parallel. Suppose that a checkpoint instruction is executed at time t' and subsequently another checkpoint is executed at time t'' (t'<t''). Mathematically, the contents of an element, $X_A$, in the auxiliary domain, at any time t, is as follows:

$$\begin{aligned} X_A(t) &= X_T(t'), \forall t' <= t < t'' \\ &= X_T(t''), \forall t \text{ such that } t >= t'' \end{aligned}$$

If we define the region between the execution of two checkpoint instructions as a **checkpoint frame**, then the state of the program seen by the auxiliary processor is the state that existed at the end of the previous checkpoint frame ( and just before the start of the current checkpoint frame). More clearly, the contents of an element $X_A$ at any time t are the same as that of $X_T$ at the end of the previous checkpoint frame. In other words $X_A$ and $X_T$ are out of synchronization for the duration of a checkpoint frame and are synchronized at a checkpoint boundary.

The hardware support necessary to implement this synchronization mechanism will be in the form of copying the target domain into the auxiliary domain, when a CHKPT instruction is executed by the target processor. We will discuss the merits of this scheme in comparison to the other schemes in the next section.

- Distributed Synchronization: The main disadvantage in the previous synchronization scheme was that the complete target domain was required to be copied into the auxiliary domain. Usually, the time interval between two checkpoints is so small that a significant majority of the elements in the target domain remain unchanged. In view of this, copying all the elements of the target domain into the auxiliary domain is wasteful and inefficient in both storage space and time required to do so (although the activity of the auxiliary processor can proceed in parallel with the target processor, the time taken by the auxiliary processor to copy the contents of the auxiliary domain into an archival storage will be quite high. This time will define a minimum time for the initiation of the next checkpoint, because the next checkpoint initiation cannot start before the current one has been processed).

  Distributed synchronization scheme is a method that involves synchronizing the target and auxiliary domains on an element-by-element basis rather than on the whole domain at a time. Distributed synchronization is a means to alleviate most of the problems with discrete synchronization. Assume again, that a checkpoint has been initiated at times t" and t', as shown in figure 4.1. Further, assume that we are positioned at the instruction pointed to by the arrow (see figure 4.1) and that the current time is t. we now proceed to answer the following question:

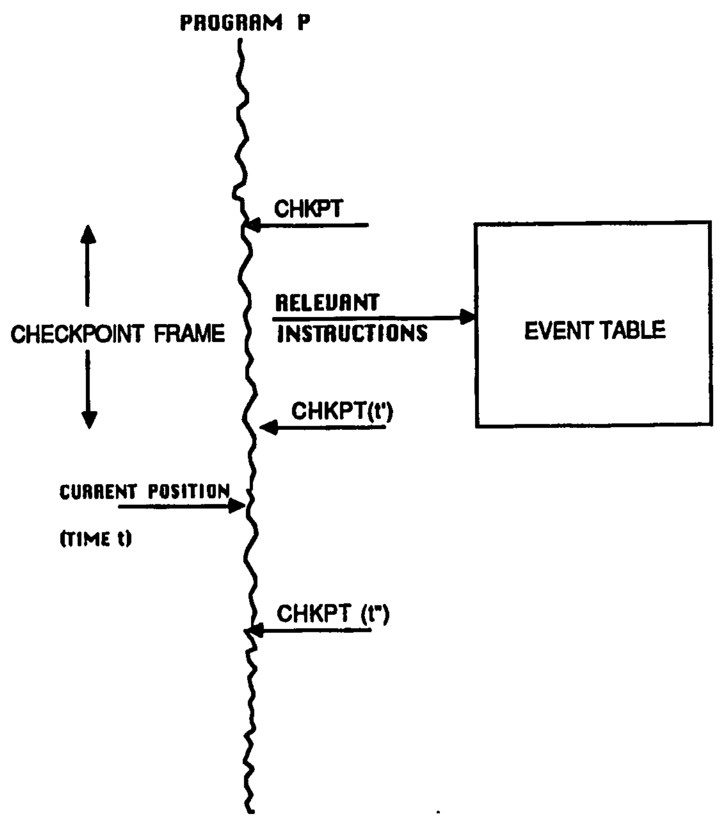


Figure 4.1. Distributed Synchronization

*What information needs to be captured at time t' or during the interval (t',t) so that a rollback from the state at t to that at t' can be accomplished with the minimum overhead (in terms of space and time)?* .

The answer to the above question lies in the following observation: a variable, say X, could be updated numerous times in the interval (t',t). However, to restore the contents of X, at time t, to the value it had at t', it will suffice to do the following:

- In the interval (t',t) watch for the first time that X is updated.
- When it gets updated the first time, capture the previous value of X. We will denote this value as x.
- Ignore all further changes to X, after the first update.

To restore the value of X to that at time t', copy x into $X_T$. Note here, that the above technique is sufficient for rolling back a program by a *checkpoint frame* (it is insufficient for any smaller granularity).

Let t' denote the closest checkpoint initiation time before time t. Further, let t" denote the first time that the target processor writes into $X_T$ after t'. Mathematically, the contents of $X_A$, determined by the above scheme are as follows:

$$\begin{aligned} X_A(t) &= X_A(t'),\ t' < t < t'' \\ &= X_T(t'),\ t > t'' \end{aligned}$$

As seen above, $X_A$ is updated when $X_T$ is written into the first time after t' and the value written into it is the previous value of $X_T$ i.e, its value just before t". The implementation of this synchronization scheme will be the subject of a subsequent section.

### 4.4.2 Criteria

In this section, we will evaluate the advantages and disadvantages of the two synchronization schemes described in the previous section.

- Time: In the global synchronization scheme, checkpointing is performed on the memory as a whole, where as in the distributed synchronization scheme it is done on an element-by-element basis. Ability to checkpoint on an element-by-element basis allows the auxiliary processor to perform certain optimizations (e.g, not checkpointing a location that has not changed since the last checkpoint.) and is therefore superior.

- Storage: A scheme requiring lesser storage space is clearly desirable. On this count element-by-element synchronization scheme is better than the global synchronization scheme, because a small fraction of the whole memory would be modified within a checkpoint frame.

- Setup Time: We define this time as the time that the target processor has to wait after a checkpoint initiation, before starting the execution of a new checkpoint frame.

  In a global synchronization scheme this time is 0 (assuming of course that the interval between two checkpoints is larger than the time for the auxiliary processor to finish with the previous checkpoint).

  In distributed synchronization scheme, the target processor has to wait for the auxiliary processor to finish checkpointing elements changed within the current checkpoint frame before it can proceed with the next checkpoint frame. Note that this wait period is necessary regardless of the length of the interval between two checkpoints.

## 4.5 Distributed Synchronization: Our Approach

We shall be concentrating on the issue of distributed synchronization and how it can be achieved, in more concrete and practical terms. This section describes a memory architecture containing two domains, one accessible to the target processor and the other to the auxiliary processor. It also describes how the two domains are synchronized and how the auxiliary processor views a consistent state of the system without perturbing the target processor.

As an example, consider the 2-level memory of size 3, shown in fig 4.2 . Each element in this memory is a tuple, consisting of 2 locations, one belonging to the top layer and the other to the bottom layer. The top layer of memory, which contains 3

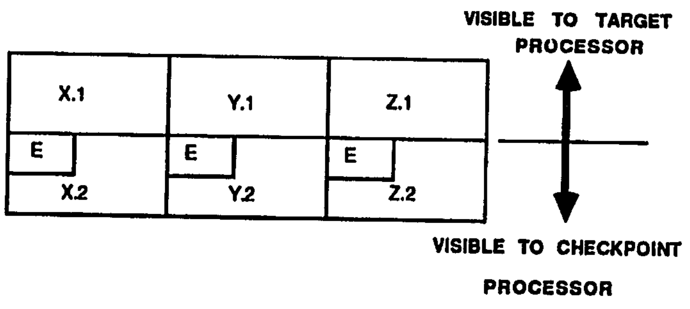


Figure 4.2. TWO-LEVEL MEMORY SCHEMATIC

locations, is visible only to the target processor and the bottom layer of memory, which also contains 3 locations is visible only to the checkpoint processor. For simplicity of reference let us associate the variables X,Y and Z with the 3 locations on both the layers. To differentiate between the layers, we shall annotate the variable names with the processor to which they are visible. Let the number 1 denote the target processor and the number 2 denote the checkpoint processor. With this terminology, the top 3 locations are labeled as "X.1", "Y.1" and "Z.1" and the locations in the bottom layer are labeled as "X.2", "Y.2", "Z.2" as shown in figure 4.2. The reason for the having 2 layers is to avoid any possible interference to the target by the checkpoint processor

The mapping between the two domains is as follows:

At any time t, $f_t(\mathrm{v}.2) = f_{t'}(\mathrm{v}.1)$ , where

t' = Max[t'': $f_{t''}(\mathrm{v}.1) \neq f_{t''+\delta t}(\mathrm{v}.1)$, for an arbitrarily small but positive $\delta t$. ]

where $f_t(\mathrm{x})$, represents the value of variable x at time t.

In other words, the mapping forces a stack-like relationship between the corresponding locations in the two domains. Any attempt to write to a location in the domain of the target processor results in the trickling down of the current value(prior to writing) to the corresponding location in the auxiliary domain. The value in the auxiliary domain is overwritten. Note that at any time t, the auxiliary domain represents a consistent state of the system i.e, the state of the system at some time t' < t.

Each element (i.e, the tuple consisting of a location from the top layer and the corresponding one from the bottom layer) is considered as a stack. Whenever an enabled location in the top layer is written into, the old value in the top layer is pushed to the corresponding location in the bottom layer and the new value is written into the location in the top layer. The old value in the bottom layer is lost. The stack action is controlled by an enable bit, E-bit, that is associated with every element in memory. If the E-bit is 0, the element operates like a stack as described above. If the E-bit is 1, then any attempt to write to the top layer, modifies only the top location and does not effect the bottom location. This simple memory architecture is very powerful and very useful in performing non-intrusive checkpointing.

When a checkpoint is initiated (we will discuss aspects of checkpoint initiation in a later section) by the program, the auxiliary hardware automatically does the following:

- Sets all E-bits to 0. When E is 0, the memory acts like a stack, as explained above.
- On the first pushdown, after a checkpoint (i.e, after the first write into a location after a checkpoint) the E-bit of the corresponding location is set to 1, having the effect of locking out the bottom location after the pushdown.
- The bottom location of the corresponding stack is checkpointed by the auxiliary processor onto a disk, while the target processor continues executing. Checkpointing is therefore done non-intrusively.

Consider a program P, executing on a single processor, which initiates checkpoints at times t, t' and t". Changes occurring in the time period t'-t" are stored during the same period. Of course changes to a variable in the time period t' and t" after the first change are ignored. If an error occurs and is detected in the period t'-t", then

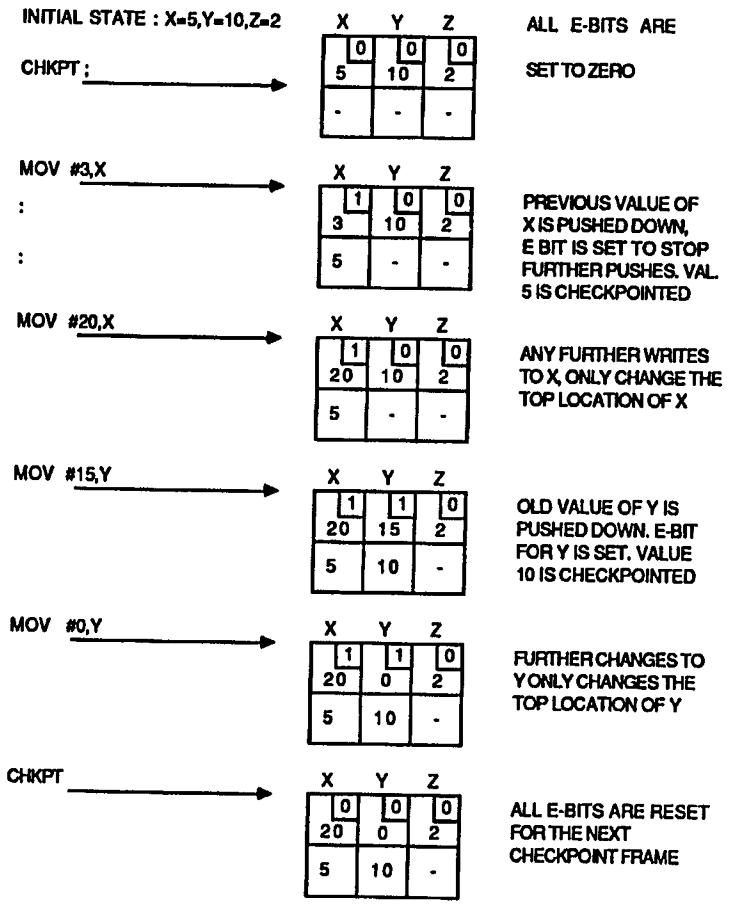

INITIAL STATE : X=5,Y=10,Z=2
CHKPT ;
MOV #3,X
MOV #20,X
MOV #15,Y
MOV #0,Y
CHKPT
ALL E-BITS ARE SET TO ZERO
PREVIOUS VALUE OF X IS PUSHED DOWN, E BIT IS SET TO STOP FURTHER PUSHES. VAL. 5 IS CHECKPOINTED
ANY FURTHER WRITES TO X, ONLY CHANGE THE TOP LOCATION OF X
OLD VALUE OF Y IS PUSHED DOWN. E-BIT FOR Y IS SET. VALUE 10 IS CHECKPOINTED
FURTHER CHANGES TO Y ONLY CHANGES THE TOP LOCATION OF Y
ALL E-BITS ARE RESET FOR THE NEXT CHECKPOINT FRAME

Figure 4.3. Checkpoint Example

the system can be easily restored to the state at t' by reverse pushing all stacks whose E-bits are set to 1. Further backup, to the state at time t, is achieved by undoing the changes that occurred during the period from t and t'.

It is interesting to note that the E-bit also indicates whether or not the corresponding variable has been changed since the last checkpoint. So, the checkpoint processor need not checkpoint every location. Instead it stores only the locations that have been changed since the last checkpoint. For a clearer understanding of the whole scheme, refer to figure 4.3, which shows the changing state of main memory after each macro instruction. Note that the variable Z has not changed in value and is therefore not checkpointed. Also only the first change to X and Y are checkpointed. So, in the above scheme, if one wants to restore the system to the previous checkpoint, all one has to do is, restore the checkpointed values, i.e, set X to 10 and Y to 5.

### 4.5.1 Advantages of the above scheme

The advantages of a distributed synchronization scheme are as follows:

- The whole operation is non-intrusive: The auxiliary processor operates in parallel with the target processor.
- Minimal scheme: The scheme outlined above checkpoints only those locations that have changed, for the first time, since the last checkpoint.
- Since backward deltas are being maintained, it is very easy to restore the program to a previous state (as opposed to a scheme that maintains forward deltas).
- The whole operation is atomic, i.e, all processes of the program are halted at the same time. This avoids the domino-effect present in schemes that perform process-wise checkpointing.

## 4.6 Checkpoint Initiation

In the previous sections we have described how our two-level memory approach checkpoints the state of the memory. In this section we will describe the interface between the target and the checkpoint processors.

As described up to now, the checkpoint processor and the target processor operate independently and asynchronously. The target processor executes target program instructions and the checkpoint processor checkpoints the state of memory. The checkpoint information is not useful if one does not know the location (instruction) within the target program at which the target program had the said state. There is a need therefore, for the two asynchronous processors to rendezvous with each other. The rendezvous is needed for another reason, namely, flushing the on-chip state so that the checkpoint processor can capture the on-chip state.

We are proposing that the rendezvous be performed as part of a new macro instruction called CHKPT. When the CHKPT instruction is executed by the target processor, it flushes the on-chip state to memory and cues the checkpoint processor about the start of a fresh checkpoint frame. The cueing is performed via a line on the bus that is monitored by the checkpointing hardware. The target processor must in turn wait for the checkpoint processor to reply that it is ready for the next checkpoint (otherwise an inconsistency might arise, as explained below)

Consider the following program:

{CHKPT(1);...;x:=5;CHKPT(2);x:=15;....}

The numbers attached to the CHKPT instructions are just for purposes of differentiating between them. Suppose that the value of x is 10 before CHPT(1). Also suppose that the first assignment to x occurs just before CHKPT(2), as seen in the above program. X is then updated again immediately after CHKPT(2). Note that when the target processor executes the CHKPT(2) instruction, the checkpoint processor may not have finished recording changes to variables in the region between CHKPT(1) and CHKPT(2). If the instruction CHKPT(2) resets all the E-bits, then the fact that the value of X has changed to 5, will be lost and an inconsistent state will be recorded.

The above possible inconsistency can be avoided by having the checkpoint processor inform the target processor about its readiness for the next checkpoint. We propose that this be accomplished through a RESUME signal. Finally, the semantics of the CHKPT instruction are:

- Save the processor registers into main memory.
- Flush the on-chip caches, if required. The instruction cache need not be flushed because no entry within it would have been modified (assuming no self-modifying

code). Even, in the data cache we need flush only those data items that have been modified.

- Assert a line on the bus to cue the checkpoint processor about the start of a new checkpoint frame.
- Wait for the resume signal (on another line on the bus) from the checkpoint processor.
- On receiving the RESUME signal, proceed with next macro instruction within the program.

The target processor, therefore, has to wait for the RESUME signal from the checkpoint processor before proceeding with next instruction. Note, that the waiting time will be very small because of two factors:

- The checkpoint processor checkpoints only the *first* change to a variable in the checkpoint frame. Successive changes to a variable, within a checkpoint frame, after the first change are NOT recorded.
- The checkpoint processor, records first-time changes, in parallel with the execution of the target processor.

### 4.6.1 Use of CHKPT instruction

As described above, the CHKPT instruction is very powerful and must be used with great caution. Since it does not contribute towards the target program computation, frequent use of this instruction can result in performance penalty. On the other hand a very low frequency of checkpointing (i.e, a huge checkpoint frame) is also not advisable because to replay the behavior of this checkpoint frame, we will need another huge event table. Note that a longer checkpoint frame will result in the execution of more relevant instructions which will in turn require a larger event table to store their sequence. So the optimal size of a checkpoint frame will be attained when the storage required to log the sequence of relevant instructions within the frame matches the available size of event memory. If E is the size of the event memory, R is the maximum rate of execution of relevant instructions and W is the maximum storage required to log one relevant instruction, then the minimum time taken to fill up event memory is E/(R*W). Hence the optimal checkpoint frequency is once every E/(R*W) units of time. Checkpointing at a frequency higher than the optimal frequency is wasteful in the sense that it will degrade the performance of the target program (since each CHKPT instructions incurs an overhead). On the other hand checkpointing at a frequency lower than the optimal frequency is insufficient since the event table cannot hold all the relevant instructions executed within the checkpoint frame.

Note here, that during a typical execution, multiple checkpoints along with their associated event tables are stored onto the disks, to allow for roll-backs beyond just a single checkpoint frame.

## 4.7 Auxiliary Processor

In this section we shall discuss how the auxiliary processor stores the changes indicated by the E-bit in the two level memory, onto stable storage. An obvious approach is to have the auxiliary processor examine the E-bit of each location and store the values of those locations whose E-bit is set. This, however, is very time consuming since many of the E-bits would be zero and the auxiliary processor would be wasting its time examining each of the E-bits. Also, having just one E-bit, causes interference when it is accessed simultaneously by the target processor and the checkpoint processor.

An alternative solution which results in a more efficient (time-wise) checkpoint can be obtained by modifying the operation of the two-level memory as follows:

- An E-bit is maintained for each location in the auxiliary domain also, thereby

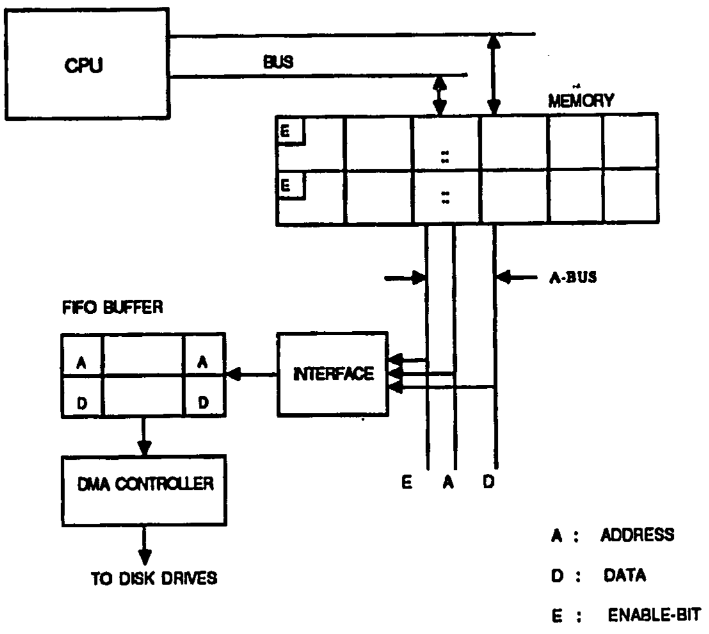


Figure 4.4. Mechanism of Checkpointing

decoupling the checkpoint and target processors from referencing the same E-bit.

- On each write to memory, if the E-bit is set, then the data part of the memory is not pushed (as before). However, the value of the E-bit is pushed down one level always. Thus the E-bits on the lower level always reflect the previous value of the E-bit in the higher level. Pushing the E-bits always, allows the checkpoint processor to determine if it needs to checkpoint a particular location.

- On each write to memory, immediately after the value is written into the higher level, the contents of the lower level of the corresponding location are put out on the A-bus, as shown in figure 4.4.

- The interface between the auxiliary processor and the A-bus checks if the value for E-bit on the corresponding line on the A-bus is 0. If it is, then the address and data bus contents of the A-bus are loaded onto a FIFO. A 0 on the E-bit means that this is the first change to the location after a checkpoint instruction and therefore the value of the location must be checkpointed.

- The auxiliary processor keeps reading entries of the FIFO and storing them onto a disk.

## 4.8 Effect of Caching

In this section we will discuss issues in checkpointing in the presence of caches. More clearly, how the various cache update policies affect our checkpointing strategy. There are three main cache update policies namely, write-through, copy-back and write-once.

- **Write-through:** Here, every time an item in cache is written into, it is immediately written out to main memory (via the bus ). The main memory is therefore, always consistent and current. Note that from a checkpointing standpoint, this is ideal, because at a checkpoint there is no need to flush the data caches, because the main memory is already upto-date. This, therefore, reduces the overhead involved in the CHKPT instruction. However, in multi-processor situations, when there is considerable contention for the bus, a write-through policy reduces the performance because it uses the bus frequently and inefficiently.

- **Copy-back:** This is an improvement over write-through in terms of performance. In this policy, writes to locations in cache are not immediately written to main memory. Main-memory is updated only when the corresponding item in cache is replaced. In such a system, contents of the main-memory are not always current. The performance advantage comes from the fact that, all changes to an item are collected and the bus is used once every so often to update main memory. However, with respect to our 2-level memory scheme, the policy may result in inconsistent checkpoints. More clearly, if an item in data cache is not flushed between two CHKPT instructions, then the corresponding E-bit in main memory will remain 0 and the checkpoint processor will not checkpoint this item. An obvious solution is to flush the data caches just before a checkpoint. The target processor would then have to wait for the checkpoint processor to checkpoint all the necessary locations and reset all the E-bits, before proceeding to the next instruction. The problem with this scheme is that the checkpoint processor is mostly idle between two CHKPT instructions and is suddenly loaded with a number of checkpoints just before a CHKPT instruction. It would be more efficient if it could be kept busy between the CHKPT instructions, so that its load is more distributed. This would also enhance the performance of the target processor, in that it has to wait for a shorter duration at a CHKPT instruction for the checkpoint processor to ready itself for the next checkpoint. While it is not possible to keep the checkpoint processor busy between two CHKPT instructions in a Copy-back scheme (because of the nature of how Copy-back works), it is possible to do so with a different caching scheme, namely, the Write-once scheme.

- **Write-Once**: This policy is a good compromise between the write-through and copy-back policies. Here, the first time an item is written into, it is written out to main memory. However, succeeding writes are not written out to main memory until the item is flushed from the cache. In essence the policy is write-through for the first write and copy-back for subsequent writes. In our two level memory scheme, the first write-through trickles down to main memory, thereby setting the E-bit, so that the checkpoint processor can checkpoint it. After the first write, since the E-bit is set, further writes do not effect the checkpoint processor anyway. The write-Once policy therefore, distributes the checkpointing load, in that it keeps the checkpoint processor busy between two CHKPT instructions. Note that even in this scheme, one has to flush the caches to update main memory for the next checkpoint frame. The important point is that few of the flushes will actually be first-time-writes and therefore the checkpoint processor has fewer locations to actually checkpoint (Note that only those locations that have changed for the first time since the last checkpoint are actually checkpointed by the checkpoint processor). As a direct consequence, the reset time for the checkpoint processor is reduced. Also the write-Once policy meshes very well with cache coherency protocols which use the first write to invalidate copies of the item in other caches, in multi-processor systems.

## 4.9 Summary

In summary, we described a ncn-intrusive scheme of asynchronously checkpointing the state of the system while the target processor is executing. The checkpointing scheme takes snapshots of all the processes as a whole, thus avoiding the well known domino problem [Ran75]. The specific hardware enhancements necessary to implement the scheme are as follows:

- A 2-level memory as described in the chapter.
- A new CHKPT machine instruction to initiate checkpoints.
- A *checkpoint line* on the bus, used by the target processor to indicate to the checkpoint processor to start a new checkpoint frame.
- A *Resume line* on the bus, used by the checkpoint processor to indicate to the target processor that it is ready to start a new checkpoint frame.

It is important to note that the "CHKPT" instruction is executed by the target processor. However, this does not contribute to any interference because it is always there within the program and is a part of the target program (see definition of interference in chapter 3). Although the "CHKPT" instruction does not contribute to any interference, it does contribute to a loss in efficiency (since it is an overhead and does not contribute to computation). The loss in efficiency is essentially due to the waiting time for the RESUME signal from the auxiliary checkpoint processor, which as discussed in the chapter is negligible.

# CHAPTER 5

# ARCHITECTURAL ENHANCEMENTS

## 5.1 Architectural Enhancements

In the previous chapters we classified the behavior of a program into various categories. Within each behavioral category, we partitioned all possible traces into equivalence classes and proved that the behavior of each trace within an equivalence class was indistinguishable from that of any other trace in that equivalence class. Additionally, the assignment of a trace into an equivalence class was proved to be uniquely determined by the sequence of relevant instructions in the trace for the corresponding behavioral category.

In this chapter, we will describe the necessary architectural modifications required to non intrusively monitor and log the sequence of relevant instructions corresponding to a behavioral category.

In section 3.8, we argued that activity of monitoring must be performed at the machine instruction level. Given this, the only place where relevant instructions, at the machine instruction level, can be monitored non intrusively, is by watching the CPU-Main memory bus. Unfortunately, the bus does not reflect all the activity within the target processor. The architectural modifications, suggested in this chapter, will therefore consist of additional bus lines that will aid the monitoring hardware watching the bus to deduce the sequence of relevant instructions being executed by the target processor.

The nature of hardware support necessary, will depend on the architecture of the target processor. For ease of later discussion, we will classify the architectures into various categories. The categories are as follows:

- Type-I: The type-I architecture is the simplest form of architecture we will be addressing. It fetches an instruction and executes it completely before fetching the next instruction, i.e, there is no parallelism between fetching and executing an instruction.
- Type-II: A type-II architecture parallelizes instruction fetching and execution, but has no parallellism within the execution phase. In other words, it has an on-chip instruction buffer (for performing look-ahead fetching of instructions) but has no internal pipeline for parallelizing the processing of instructions that are fetched.
- Type-III: A type-III architecture is similar to a type-II architecture, except that it has an internal instruction pipeline that operates on multiple instructions at a time.
- Type-IV: This has all the features of a type-III architecture and an off-chip instruction and data cache.

Throughout this chapter, we will also assume support from the compiler for tagging all static relevant instructions (by setting a bit in the opcode field of the corresponding instruction, as explained in chapter 3). For each of the various architectural categories we will address the following issues:

- How does the monitoring hardware detect the execution of relevant instructions and how does it subsequently log them into event memory?
- What is the organization of the monitoring hardware and how does it keep up with the target processor?

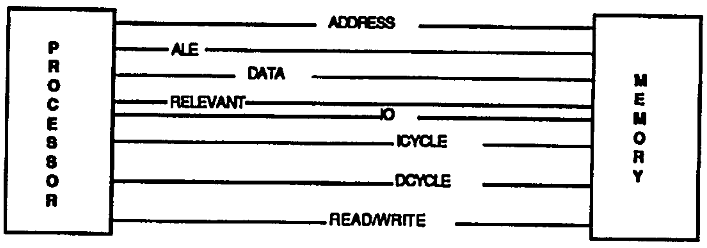


Figure 5.1. Target Processor Organization

## 5.2 Type-I Architecture

In this section we will outline the architectural enhancements necessary for monitoring the sequence of relevant instructions, for each of the three behavior classes, for a type-I architecture.

### 5.2.1 Target CPU Organization

Figure 5.1 depicts the organization of the target processor with a type-I architecture. This is the simplest target processor organization that we will be studying. It does not have any performance enhancing features such as instruction buffers, pipelining or caching. A typical instruction execution goes through the following phases:

- Instruction Fetch Phase: The instruction is fetched from main memory over the bus.
- Decode Phase: The instruction is decoded to determine if it requires any operands to be fetched from memory. If so, the operands are fetched from memory.
- Execute Phase: The instruction is executed and the results stored appropriately.

All the above steps are carried out in strict sequence i.e, there is no overlapping between the phases. An instruction that is fetched from memory is always executed. External interrupts are not acknowledged until the execution of the current instruction is completed.

### 5.2.2 Bus Signals

We shall now explain each of the bus signals shown in figure 5.1.

- ADDRESS: These are a set of lines, which collectively specify an address. The target processor uses these lines to access a particular location in memory.

- ALE: Address Latch Enable
  This is a line driven by the processor, to indicate (to memory) the validity of the contents of the address lines.

- DATA: These are a set of lines that can be driven either by the processor or by the memory. When driven by the processor, the memory stores contents of these lines into the location identified by the address lines. When driven by memory, they indicate the contents in memory of the location addressed by the address lines.

- READ/WRITE: This line is used to indicate the direction of data transfer. If memory is being accessed for read it is asserted. If memory is being accessed for write, it is not asserted. If memory is not being accessed for either read or write, it is usually tri-stated.

- ICYCLE: This line driven by the CPU to indicate the state it is in. Usually the processor goes through three basic phases. The instruction fetch phase, data fetch phase and execute phase. The ICYCLE line is asserted during the instruction fetch phase.

- DCYCLE: This line is similar to the ICYCLE line described above. It is asserted by the CPU to indicate that it is in the data fetch phase.

- IO: In an I/O mapped system, there is a need to differentiate the memory address space from the IO address space. This line is driven by the processor to indicate that the contents of the address line belong to the IO address space as opposed to the memory address space.

- RELEVANT: This is a new line we propose to be added to the bus to allow the monitoring hardware to log the sequence of relevant instructions. Recall from chapter 3, that the compiler tags certain instructions as relevant by setting a bit (reserved for this purpose) in the opcode field of the instruction. When an instruction is being fetched this bit will be transmitted onto a specific line on the data bus. This line is called the relevant line. It can be driven by either the memory unit or the processor unit. In other words it is wire-ored between the processor and memory (we will explain later why this is so).

### 5.2.3 Assumption

During the discussion of the architectural issues, we shall assume the underlying machine has the following features:

- All instructions are fixed-length instructions. The length of the instruction is no more than the width of the data bus.
- The machine is a load-store architecture.
- None of the machine instructions are modifiable, i.e, the instructions are NOT self-modifying.
- Every machine instruction has at most one memory operand.
- Operands for an instruction are fetched only after all the operands for the previous instructions have been fetched.
- Input/output are accomplished through explicit I/O instructions, i.e, the system is I/O mapped (as opposed to memory mapped).

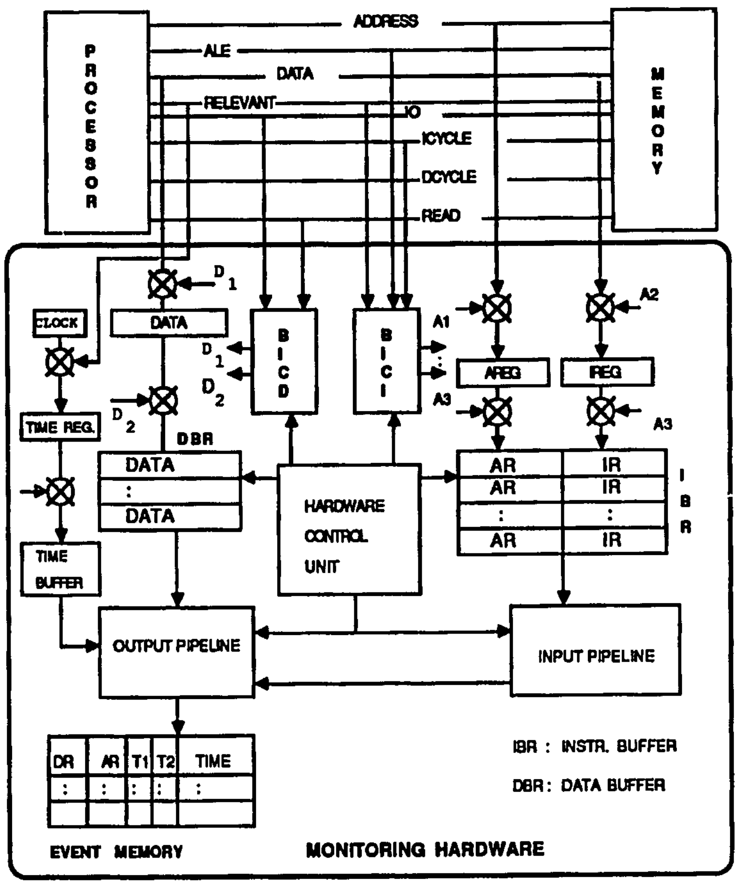


Figure 5.2. Snoopy-Target interface for I/O mapped systems

- The system does not have a DMA capability.
- The ISR (Interrupt Service Routine) does not share registers with the interrupted thread, i.e, all registers are saved before switching the context to the ISR.
- No on-chip timer i.e, it is assumed that the timer is an off-chip I/O device.

### 5.2.4 Structure of Monitoring Hardware

Figure 5.2 shows the organization of the monitoring hardware for the simple target architecture shown in Figure 5.1. All the components within the darkened rectangle with rounded edges belong to the monitoring hardware and together accomplish the task of logging the sequence of relevant instructions. All the monitoring hardware will be collectively referred to as Snoopy. We shall explain the purpose of each of these components:

- BICI: Bus Interface Controller for Instructions

  This component is a hardwired controller that controls the transfer of information between the bus and the instruction buffer (IBR). In particular, it strobes in the contents of the ADDRESS and DATA lines, as shown in Figure 5.2, into the registers AREG and IREG respectively. This transfer of data is controlled through the control signals A1 and A2. Subsequently, it transfers the information to the instruction buffer (IBR). Input to BICI is through three lines on the bus, namely, the *RELEVANT line* and the *ICYCLE line* and the *ALE line*. The ICYCLE line when active indicates that the memory is currently being accessed to fetch an instruction (as opposed to data). During an ICYCLE, an asserted RELEVANT line indicates that the instruction currently being fetched is relevant. An ALE line on the other hand indicates that the address on the ADDRESS lines is valid.

  Whenever the BICI senses the ALE line to be high, it strobes in the contents of the ADDRESS lines into AREG by asserting control signal A1. Subsequently, when the ICYCLE line is asserted, BICI strobes in the contents of the data bus into IREG, by asserting control signal A2. During the next cycle, it transfers the contents of AREG and IREG into the IBR by asserting control signal A3.

  The Bus Interface Controller for Instructions is therefore responsible for sensing relevant instructions being fetched by the target processor and logging them into the instruction buffer along with their addresses. Note here that only relevant instructions are logged into the instruction buffer.

- BICD: Bus Interface Controller for Data

This component is similar to BICI described above except that it interfaces the DATA lines to the Data buffer (DBR), as shown in figure 5.2. Transfer from the Data lines on the bus to the DBR occurs over two phases. The first phase involves transferring the contents of the DATA bus to Snoopy's Data register through the control signal D1 and the second phase involves transfer from the data register DATA to DBR through the control signal D2.

Recall from chapter 3, that it is necessary to trap only input data for purposes of subsequent replay (input data is data that is not generated by another instruction in the program i.e, it is data that is read from the external environment). BICD, therefore, needs to trap only I/O reads (all other non-I/O reads will be reproduced anyway during replay). The BICD asserts D1 and D2 depending on the two lines IO and READ. The target processor asserts the IO line whenever it executes an Input/Output instruction (the system is IO mapped by assumption). The BICD therefore asserts D1 whenever an I/O read is in progress. The data on the data bus during this period corresponds to the input to an input instruction. Recall from chapter 3 that the input to an input instruction is relevant and must therefore be trapped for reproducing the same value during the subsequent replay phase.

- IBR: Instruction Buffer

  This component stores all the relevant instructions fetched by the target processor from memory. Note that for the simplified target processor architecture being discussed (see section 5.1), every instruction fetched from memory has to be executed (i.e, there is no instruction look-ahead facility, we shall discuss this in a later section). Therefore, the sequence of relevant instructions within the IBR have to be logged into memory. They cannot be directly logged into memory from IBR, because certain instructions require associated input data to be logged along with them. The instructions from IBR, are therefore passed through various other components before being logged into memory. These are the Input/Output pipeline units which are described below.

- INPUT PIPELINE:

  Notice that as shown in Figure 5.2, instructions and their corresponding data have been separated into two different buffers, namely, the IBR and the DBR respectively. The separation is done because that's how the target processor operates, i.e, it separates the instruction fetch phase from the data fetch phase. As described in chapter 3, there are certain relevant instruction which require logging the instruction along with the associated data (namely, *INPUT actions*). Before logging the sequence of relevant instructions into memory, we need to associate the INPUT actions in IBR with their corresponding DATA in DBR. The INPUT PIPELINE is a component that appends tags to entries in IBR that differentiates instructions within the IBR, that require an associated data item (from the DBR) before being logged into memory, from those that don't. This component is internally pipelined and hence the name. Details of the components within this component will be discussed in a subsequent section.

- OUTPUT PIPELINE:

  This component decodes the tags set by the INPUT PIPELINE and associates appropriate entries within the DBR with those in IBR and logs then into event memory. This component will be discussed in detail in a subsequent section.

- Time buffer:

  This component maintains the times at which the relevant instructions were fetched from memory. Each time a relevant instruction is fetched from memory, a clock read is initiated as shown in figure 5.2. The output of the clock read is logged into the time buffer via a time register. Each time a relevant instruction is logged into event memory, the corresponding fetching time of the instruction is also recorded along with the instruction.

- Event Table:

  This is the event memory that contains the sequence of relevant instructions and their associated data.

- Hardware Control Unit:

  This is the hardwired controller that controls all transfer of information between all the above components. The details of this unit will be discussed in a subsequent section.

### 5.2.5 Control Signals

In this section, we shall describe the various control signals that control the movement of data between the various components of Snoopy, as shown in Figure 5.2.

### 5.2.6 Input to Snoopy

Here we shall describe all the control signals that are needed to log relevant instructions and their corresponding addresses into Snoopy's instruction buffer, IBR.

A1 : This control signal is generated by BICI. It strobes in the contents of the address lines on the bus to Snoopy's AREG. As described earlier, the validity of the contents of the address lines is indicated by another bus signal labeled as ALE in Figure 5.2. BICI, therefore, activates the control signal A1 whenever the ALE line goes high (note that ALE line is input to BICI, as shown in figure 5.2).

A2 : This control signal is also generated by BICI and is used to strobe in the contents of the data lines on the bus into Snoopy's IREG. BICI generates this signal in response to the ICYCLE line on the bus (which, as explained before, indicates that the current memory fetch is fetching an instruction as opposed to data).

A3 : This is generated by BICI and is used to strobe the contents of AREG and IREG into IBR, as shown in Figure 5.2. A3 is activated only if the relevant line on the bus is activated (recall that this line is activated only if the current instruction fetch is fetching a relevant instruction). The signals, A1, A2 and A3 collectively log relevant instructions along with their addresses into IBR.

### 5.2.7 Instruction Buffer

The IBR as shown in Figure 5.2, is a hardware buffer, i.e, all movement of data through the buffer is controlled by hardware. Each time an entry is inserted into IBR, all the other entries must be advanced by one in order to make room for the new entry. Advancing of all entries within IBR is controlled by the *Hardware Control Unit* (HCU) as shown by the arrow from HCU to IBR, in Figure 5.2. The control signal responsible for this will be labeled as $M_1$ (the M stands for management, i.e, buffer Management). Note that according to the setup shown in Figure 5.2, control signal $M_1$ must be activated at least as frequently as that dictated by the minimum inter-arrival time between relevant instructions. A buffer between IBR and the AREG and IREG may be required to smooth out timing mismatches when $M_1$ cannot be asserted fast enough to keep up with the minimum inter-arrival time between relevant instructions.

In Figure 5.3, the component IBR, is shown with a control signal $s_1$. $s_1$ is a status signal used to inform HCU about whether IBR contains any entries at all. If the IBR does not contain any valid entries, i.e, it is empty, then the controller does not generate any buffer management signals.

### 5.2.8 Logging Input Data

The control signals $D_1$ and $D_2$ are used to strobe the contents of the data lines on the bus into DBR. Note that it is only necessary to store input values that are read from the external environment (i.e, I/O reads) because they are potentially ir-reproducible across successive executions of the program. Control signal $D_1$ causes the contents of the data lines to be strobed into the DATA register. $D_1$ is generated by BICD when the I/O and READ lines on the bus are activated. $D_2$ is activated to transfer the contents of the DATA register into DBR.

The Data Buffer Register, DBR, like IBR, is a hardware queue. Movement of data through it is controlled by hardware signals. Control signal $M_2$, as shown in Figure 5.3 is used by HCU to manage DBR.

### 5.2.9 Input/Output Pipeline

We shall now discuss the control signals within the Input Pipeline as shown in Figure 5.3.

$C_1$ : Control signal $C_1$ is generated by HCU. It is used to read the first entry in IBR for processing by rest of the input pipeline. Note that once $C_1$ is turned off, IBR and Reg. 1 are essentially disconnected.

$C_2$ : After the entry in IBR is strobed into REG. 1, the components COMP1 and COMP2 analyze the IR-part of the REG. 1, as shown in figure 5.3. Component COMP1 checks if the instruction is a *CHKPT* instruction. Knowledge of this instruction occurrence is required to delimit checkpoint frames (explained in chapter 4. Basically COMP1 consists of a comparator that compares the opcode of IR to that of a *CHKPT* instruction. The output of COMP1 is either 1 or 0 depending on whether the instruction was a *CHKPT* instruction or not.

Component COMP2 on the other hand, is also a comparator that checks whether the instruction IR is an input instruction or not (why this is needed is explained later). The output of COMP2 is a 1 if the instruction is an input instruction and a 0 otherwise.

Control signal, $C_2$, activated by HCU is used to strobe the outputs of the components COMP1 and COMP2 into REG. 2, as shown in Figure 5.3.

$C_3$ : Control signal $C_3$ is used to strobe the output of REG. 2 into REG. 3, as shown in Figure 5.3. Note here that the registers are being used to decouple the various components within Snoopy, in order to enable parallel operation. More clearly, REG. 1 e.g, decouples IBR from components COMP1 and COMP2. As a direct result, buffer management of IBR can occur while COMP1 and COMP2 are processing an entry.

Note from Figure 5.3 that tag T2 is "anded" with $C_3$ to generate a control signal for reading DBR. Recall that T2 is set to 1 if the instruction is an input instruction. Also, every input instruction has a corresponding data entry within DBR that corresponds to it. Hence, if T2 is 1, we read off the entry in DBR that corresponds to it and log the whole REG. 3 into event memory. Note that the T2 shown to be "anded" with $C_3$ in figure 5.3 comes from REG. 2.

$C_4$ : Control signal $C_4$ is used to actually write into event memory. The MAR component shown in Figure 5.3, is the Memory Address Register, used to address the event memory. Each time an entry is logged into event memory, MAR is incremented appropriately to point to the next available memory location in event memory.

$C_5$ : Control signal $C_5$ is used to strobe in the new incremented MAR value back into MAR, as shown in Figure 5.3.

### 5.2.10 Data Flow

In this section we shall discuss how Snoopy logs relevant instructions. The whole operation of logging relevant instructions is done through hardware. The data path

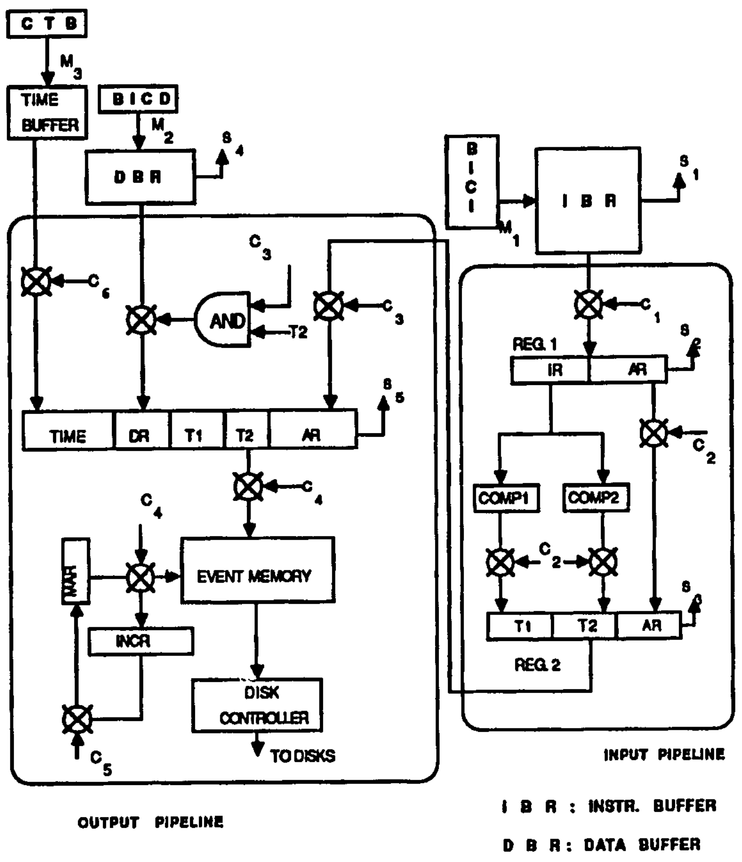


Figure 5.3. Organization Of INPUT/OUTPUT PIPELINES

within Snoopy is controlled through a hardwired control unit. The control signals generated by the control unit guide the flow of relevant instructions through various stages of processing before finally logging it into the event memory. The decision to perform the monitoring activity totally in hardware arose out of a number of unsuccessful attempts to do the same in software. The over-riding concern while designing a monitoring scheme is to be able to perform the activity fast enough to keep up with the target processor. Figure 5.3 gives a more detailed outline of the structure of the input and output pipelines.

We shall now explain the monitoring scheme as shown in figure 5.3. The instruction and data buffers are directly loaded from the bus. Each entry within the instruction buffer consists of a tuple (AR,IR) i.e, the address of the instruction and the actual instruction. This tuple is then directed to a comparator via a register. Registers placed in between functional units hold intermediate results and allow parallelism between the units. The data path between the instruction buffer and register 1, as shown in figure 5.3, is controlled by control signal $C_1$. Notice that control signals $C_1$, $C_3$ and $C_5$ can be activated at the same time since they control independent data paths (as opposed to say $C_1$ and $C_2$). As a result multiple data transfers can take place concurrently, thus speeding up the monitoring activity (we shall come back to this later).

Note that only relevant instructions (determined by the appropriate bit in the opcode field) are logged into the instruction buffer. All other instructions are discarded. After leaving the IBR, the pair (AR,IR), is routed to the comparator unit. There are two comparators COMP1 and COMP2, as shown in figure 5.3. One of them, COMP1, checks if the instruction is a *CHKPT* instruction and the other checks if the instruction is an input instruction. *CHKPT* instructions need to be identified in order to delimit the sequence of relevant instructions occurring within a checkpoint frame (explained later). Input instructions must be identified in order to associate entries within the data buffer with instructions in the instruction buffer (as will be evident shortly).

The target processor (as shown in Figure 5.1), operates in three distinct phases. They are the instruction fetch phase, the operand fetch phase and the execute phase. For a given instruction the three phases occur in strict sequence, because only after the instruction is fetched can the operand phase begin. Similarly, the execution phase can begin only after the operands are fetched. As shown in figure 5.3, data read during the operand phase is captured in DBR (note that every read during this phase is not captured, only external reads are) and instructions fetched during the instruction fetch phase are captured in IBR (here also every instruction fetched is not captured). Since the instruction and its associated input data have been separated, there is a need to join them again before logging the instruction. Note that the joining need be done only for input actions (because the input to these actions is potentially ir-reproducible, since the input is an external operand). From the point of view of replaying a program, all other relevant instructions can be logged without their associated operands (because those operands are produced by other instructions in the program and are therefore reproducible).

Every input instruction in the IBR has an associated data item in the DBR (this is because every instruction that is fetched has to be executed and therefore has to have all its operands fetched). Also since each instruction has atmost one memory operand (see assumptions), the k-th data item (chronologically) within DBR is associated with the k-th input action in the IBR. The algorithm for associating instructions in IBR with data items in DBR is simple. Each time the instruction read (by the input pipeline) is an input instruction, it sets a tag. The output pipeline decodes this tag and associates the first entry in DBR (treated as a queue) with the instruction if the tag is set. If the tag is not set then it logs the instruction and leaves the DBR as is. We shall henceforth refer to this tag as T2 (T2 is set if the instruction in IBR is an input instruction).

As explained above, tag T2 helps identify, whether there is a need to store an input value along with the instruction address into the event memory. The output of the input pipeline is of the form (T1,T2,AR), where AR is the address of the instruction. T1 is set to 1 if the instruction is a *CHKPT* instruction (explained later). The (T1,T2,AR) pair is next routed to a register before finally being logged into the event table. Notice that the data buffer is read only if T2 is set as shown in Figure 5.3 (T2 is "anded" with control signal $C_3$). Control signals $C_4$ and $C_5$ are used to store the instruction sequence into event memory. Since event memory is limited, its contents are transferred to disks via disk controllers.

### 5.2.11 Controller Design

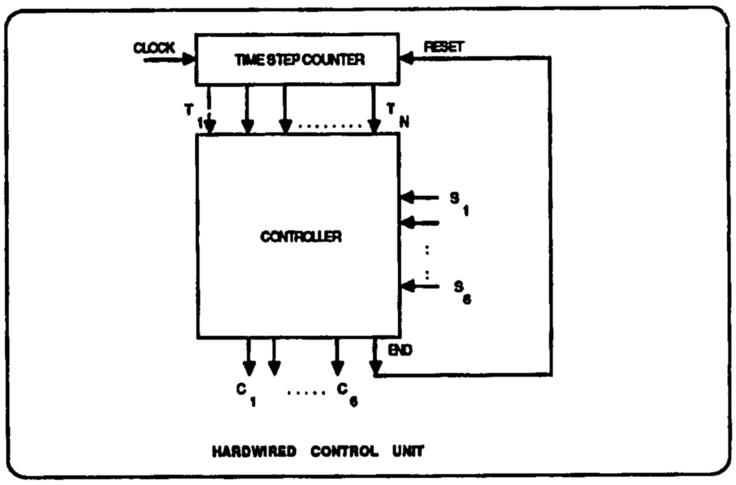
The control unit, shown in Figure 5.4, generates the control signals mentioned in the previous sections. The controller also takes care of the timing constraints between the control signals. For example, control signals $C_1$ and $C_2$ cannot be activated simultaneously. At the same time, however, it is desirable to activate as many control signals as possible in order to maximize parallelism.

The controller for the target architecture described in section 5.2.1 operates in time cycles. Each time cycle consists of many discrete time steps. It generates a subset of Snoopy's control signals during each time step. The duration of each time step depends on the control signals during the corresponding time step. As shown in Figure 5.4, the time steps are defined by a time step counter. The duration of each time step is an integral multiple of the basic clock feed to the time step counter. The time step counter, shown in Figure 5.4, is an encoder with N output lines (corresponding to an N-step counter). During each time step, exactly one of the N output lines is activated. Depending on the time step ($T_1$, ..., $T_n$) and the status signals $s_1,...,s_n$, the controller decides which control signals to activate. The control signal END is a special signal that terminates the current time cycle and begins a new time cycle. As shown in figure 5.4, the END control signal is used to reset the time step counter so that a new time cycle can be initiated.

Since the control signals generated during each time step is fixed, the hardware for generating the control signals is very simple. The control signals and the time step during which they are activated for Snoopy as shown in Figure 5.3 are as follows:

- Time Step 1 ($T_1$): $C_1$, $C_3$ and $C_5$.
- Time Step 2 ($T_2$): $C_2$, $C_4$, $M_1$ and $M_2$.
- Time Step 3 ($T_3$): END.

Hardware for generating the signals is very simple and consists of simple AND gates. Figure 5.4 shows the hardware necessary for generating $C_1$ during time step 1. $C_1$ is derived by "and'ing the signals $T_1$ and the status signal $s_1$.



1. TIME STEP 1 : ACTIVATE CONTROL SIGNALS $C_1$ , $C_3$ , $C_6$

2. TIME STEP 2 : ACTIVATE CONTROL SIGNALS $C_2$ , $C_4$ , $C_5$

Figure 5.4. Control Unit

### 5.2.12 Timing Analysis

Note that apart from generating control signals that control the data path, the controller must also generate signals that perform buffer management e.g, advancing all the entries in the instruction and data buffers. The signals labeled as $s_1$, $s_2$ etc in Figure 5.3 indicate the status of the corresponding units to the control unit . The status during a particular clock cycle indicates if the corresponding unit has valid data during that cycle. For example, if the instruction buffer is empty during a particular clock cycle, then the control unit does not generate the control signal $C_1$ during that cycle.

### 5.2.13 Time Steps

- Time step $T_1$: Activate control Signals $C_1$, $C_3$, $C_5$. Let $t_l$ denote the time to latch a value into a register. Then the duration of the control signals are as follows:

Duration of $C_1$ := $t_l$ (Latching output of instruction buffer)
Duration of $C_3$ := $t_l$ (Latching output of Data buffer)
duration of $C_5$ := time to increment MAR by 1.

Duration of time step $T_1$ is dictated by the maximum durations of $C_1$, $C_3$ and $C_5$.

- Time step $T_2$: Activate control signals $C_2$, $C_4$ and signals to manage the FIFO buffers (Instruction and data buffers), $M_1$ and $M_2$.

  Duration of $C_2$:= time taken by comparator
  Duration of $C_4$:= time to write into event memory. Duration of FIFO control signals $M_1$, $M_2$ := Time to move all the entries in the buffer up by 1

The control signal labeled END, in Figure 5.4 is activated at the end of $T_2$, so that the counter can be reset to start the next cycle.

### 5.2.14 Weak Real-Time Relevant Actions

The monitoring scheme discussed up to now can log the sequence of non real-time relevant instructions. Specifically, the scheme can detect and log the following relevant instructions:

- Shared actions: The machine instruction corresponding to a shared action has the bit in the opcode field, reserved for indicating relevancy (as discussed in chapter 3), set to 1. Consequently when this instruction is fetched over the bus, BICI strobes it into the IBR, as described earlier.

- Input/Output actions: These are actions corresponding to input/output instructions. These instructions also have the appropriate bit in the opcode field (reserved to indicate relevancy), set to 1. These are also trapped by BICI and strobed into the IBR when they are fetched from memory. The input value read corresponding to the input instruction is strobed into the DBR by the BICD, as described earlier.

There are, however, actions that cannot, in general, be determined to be relevant at compile time. The scheme described up to now cannot, therefore, log the sequence of these instructions. In chapter 3, we termed these as *Dynamic relevant actions*, as their relevancy can only be determined at run-time.

Let us take a closer look at the conditions in which an entry action can be executed. Recall from chapter 3 that an entry action corresponds to the execution of the first instruction (belonging to the application program) after a context switch. A context switch occurs in response to an interrupt. One of the following things can happen as a result of the interrupt:

- If the interrupt is an external interrupt (i.e, an interrupt for which the programmer has written the ISR), then the first instruction (of the application program) will be the first instruction within the corresponding ISR. This particular instruction is detectable by the compiler and could therefore be statically tagged to be relevant.

- If the interrupt is an internal interrupt (i.e, an interrupt for which the programmer has not written the ISR, e.g, the timer-interrupt), then the Header action is not evident. Typically, the ISR would choose one of the processes in the ready queue and transfer control to that process. Execution within that process would continue from the instruction immediately after the one that was last executed. The last instruction executed within the ISR would therefore be equivalent to (if not identical to) a JUMP to the appropriate address within that process. The first instruction execution executed within that process is the desired entry action that needs to be trapped. In the next section we will examine a hardware scheme to detect such instructions.

#### 5.2.14.1 Hardware Scheme for Dynamic Relevant Instructions

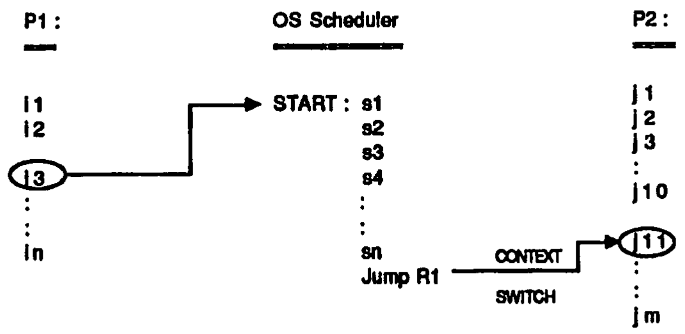
The instruction fetched after the last instruction executed within the ISR is the instruction that we are interested in detecting, because that is precisely the entry action that needs to be trapped into the IBR. Note that while fetching the instruction immediately after the last instruction within the ISR, if the relevant line is asserted, then the mechanisms for logging static relevant instructions will take over and log this instruction. Note that to snoopy it appears as if this were a static relevant instruction.

To further clarify the issue of dynamic relevant instructions, consider the situation depicted in Figure 5.5. P1 and P2 are two processes. A timer interrupt that occurs while P1 is executing, forces the control to be transferred to the *OS Scheduler* which then decides to transfer the control to process P2, through a JUMP instruction as shown in Figure 5.5

Since the processor knows when the instruction after the JUMP is being fetched, we suggest that it assert the relevant line. Thus, the relevant line on the bus is wire-ored between the processor and memory. Thus, the mechanism involves using a special instruction as the last instruction in the ISR that informs the processor that the subsequent instruction that is fetched from memory is relevant. The instruction fetch unit within the processor can then assert the relevant line corresponding to the subsequent instruction fetch.

### 5.2.15 Strong Real-Time Relevant actions

In the previous sections, we outlined the architectural support for monitoring the sequence of relevant actions which are sufficient to reproduce a non real-time and a



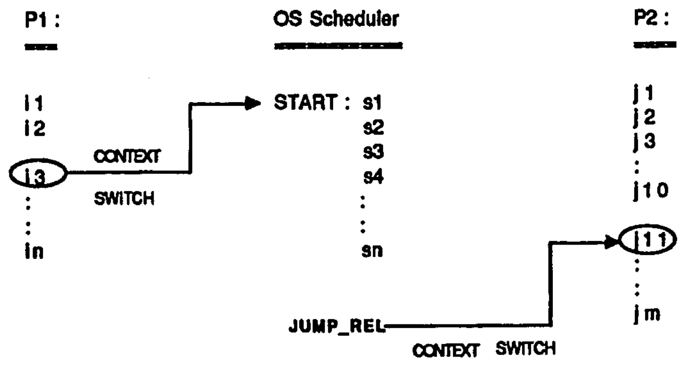


Figure 5.5. Detection Of Dynamic Relevant Instructions

weak real-time behavior class. In this section we shall examine the support necessary to reproduce a strong real-time behavior.

Recall from section 3.16 that monitoring the following relevant instructions is sufficient for reproduction of a strong real-time behavior.

- TTP (Thread Transfer Points)
- Entry actions
- Input actions
- Output actions

The hardware support necessary for monitoring Input, Output and Entry actions has already been outlined in previous sections. We will now turn our attention to monitoring and detection of TTPs.

Recall again that an action is an instance of an execution of a machine instruction. Identification of an action therefore requires two components, namely, the machine instruction address and the execution count. In the case of Shared, Input and Output actions, every execution of the corresponding machine instruction was relevant. Therefore, identification of Shared, Input and Output actions required only one component, namely, the address of the corresponding machine instruction. In the case of a TTP, however, not every execution of the machine instruction corresponding to the TTP is necessarily relevant. Identification of a TTP, therefore, requires capturing both the components necessary for identifying an action (the machine instruction and the execution count). In Chapter 3, we proved that the execution count can be implicitly obtained if we force all backward branches to be relevant. Backward branches can be detected by the compiler and appropriately tagged like all the other static relevant instructions.

For a type-I architecture as shown in Figure 5.1, trapping the address of the machine instruction corresponding to a TTP is also very simple. Since the fetching of an instruction implies the completion of the execution of the previous instruction, Snoopy can keep track of the currently executing instruction. Whenever an interrupt is acknowledged, the address of the currently executing instruction gives the address of the TTP. No special hardware support (in terms of additional bus lines) is necessary to monitor TTPs. Figure 5.6, shows the Snoopy-target interface for logging TTPs.

As shown in Figure 5.6, CIAR is a register that holds the currently executing instruction address. Whenever the interrupt acknowledge line is asserted, the contents of this register are transferred into the TTP buffer. At a later time, the controller dequeues this entry from the TTP buffer and logs it into the event memory. The design

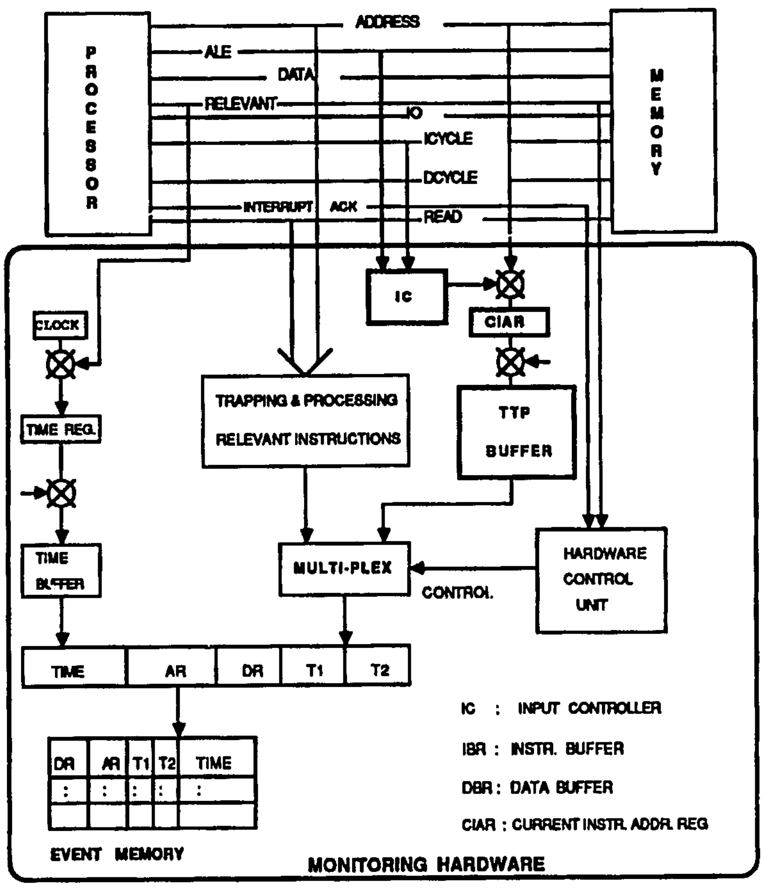


Figure 5.6. Interface for Strong Real-time Monitoring

of the controller is slightly different from that described in the previous sections (for non real-time and weak real-time instructions). The controller is different from that in previous cases because now the controller has to decide whether to log an entry from the TTP buffer or the instruction buffer (IBR). Note that with this new setup it is possible that an instruction is logged twice (this will happen if a TTP coincides with a relevant instruction). Such situations can easily be detected by the pre-processor before the replay phase is begun. Since the design of the controller for logging strong real-time relevant actions is similar to that required for logging non real-time relevant actions in a type-II architecture, we will defer the discussion of its design to a later section.

### 5.2.16 Hardware Support for I/O mapped systems

In summary, the following is the hardware support necessary for monitoring the sequence of relevant instructions for a type-I architecture as described in section 5.1 under the assumptions described in section 5.2.3.

- Non Real-time Relevant Instructions: A RELEVANT line on the bus as described above.
- Weak Real-time relevant Instructions: Hardware support is required for detecting the execution of header actions. Hardware support is in the form of a bus-line that is wired-or between the memory and the processor.
- Strong Real-time Relevant instructions: Same hardware support as for weak real-time relevant instructions.

### 5.2.17 Memory Mapped I/O

In the previous section, we assumed that the system was I/O mapped and described the mechanisms for monitoring for relevant instructions in such systems. In this section, we will do the same for memory mapped systems, i.e, systems where the I/O address space and the memory address space are the same. In memory mapped systems there are no I/O specific instructions. In other words one cannot distinguish between instructions that perform I/O from those that don't.

From a monitoring standpoint, what this means is that, Snoopy cannot determine whether or not an instruction in the IBR is an I/O instruction. Note that it cannot do so even after looking at the source or destination addresses. This is because the actual address of a source or destination operand might be computed within the processor (base + offset, computation). Since Snoopy cannot make the distinction, trapping the actual value corresponding to an input action is difficult (because it cannot differentiate an input from a normal memory read). There are two possible ways to solve this.

- Trap all memory reads and writes performed by relevant instructions. This assumes that the compiler can detect memory mapped I/O instructions to be relevant. Note that in some sense this an overkill, because we don't need to trap values written to memory because they are perfectly reproducible. However, trapping both reads and writes (although expensive in storage space), reduces the complexity of Snoopy. If we trap only values read from memory (by relevant instructions), Snoopy has to sort out relevant instructions that read from memory from those that write to it and appropriately tag them so that it can associate instructions with values read from memory. This additional amount of work results in added Snoopy complexity and overall time to log an instruction. On the other hand, trapping all memory reads and writes (by relevant instructions) eliminates such a complexity (Since every Class-I relevant instruction reads or writes to memory. Note that dynamic relevant instructions do not have this property. They will be treated later.).
- Trap only reads performed by relevant instructions. The disadvantage of this scheme, as mentioned above, is the added Snoopy complexity of separating relevant instructions that read from memory from those that write to memory.

### 5.2.18 Data Flow

In this section we shall describe Snoopy's monitoring strategy in memory mapped I/O systems. As noted in the previous section, we shall adopt solution 1 for its resulting hardware simplicity i.e, we will trap all relevant memory reads and writes. As pointed out in the previous section, there is still the problem with dynamic relevant instructions. A dynamic relevant instruction may not access memory, so Snoopy should be able to appropriately tag this type instruction in order to correctly establish a correspondence between entries in the data buffer and Snoopy's instruction buffer (see figure 5.7). Every non real-time relevant instruction has a corresponding entry in the data buffer. However, a dynamic relevant instruction may not always have. In particular, if a dynamic relevant instruction is not already a relevant instruction (as determined by the compiler), then it may not have a corresponding entry in the data buffer. Snoopy should therefore possess the capability to differentiate between static and dynamic relevant instructions. Note that previously we suggested that the relevant instruction line on the bus be a wired-or between the CPU and main memory. If this strategy is employed then Snoopy cannot differentiate between static and dynamic relevant instructions. What is, therefore, necessary are two different lines for indicating relevancy, one is activated by the CPU and the other activated by the main memory. Main memory activates one of these lines by simply placing the bit reserved for indicating relevancy onto this line. and the CPU knows when a dynamic relevant instruction is being fetched by a special instruction executed prior to fetching of the instruction in question (the special instruction was introduced in section 5.2.14. Snoopy can then monitor both these lines to differentiate dynamic relevant instruction from static relevant instructions.

There are, however, a few differences from the structure of Snoopy for memory mapped systems and that for I/O mapped systems described previously. The following are the differences:

- Separation of the relevant instruction line driven by the CPU from that driven by memory. The two lines are labeled as CPU rel. and MEM. Rel. as shown in Figure 5.7 .

- An extra input from "CPU REL." into BICI as shown in Figure 5.7. This is so that the Bus Interface Controller for Instructions can strobe relevant instructions that are indicated to be so by the CPU.

- The structure of the input pipeline is exactly the same as that in Figure 5.3

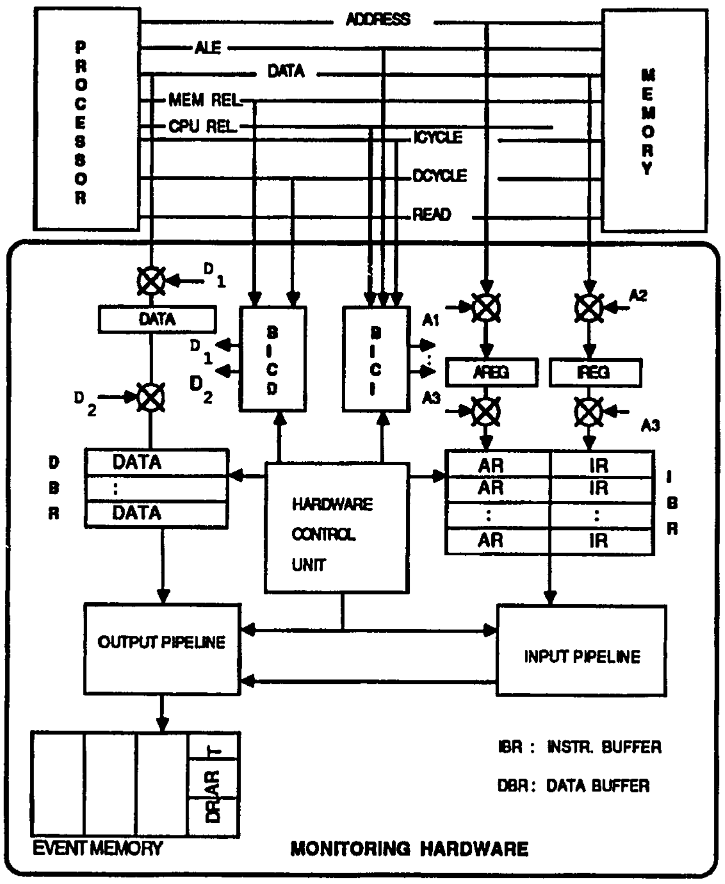


Figure 5.7. Snoopy-Target interface for memory mapped I/O systems

for I/O mapped systems. However, the interpretation of tag T1 is different. In I/O mapped systems, tag T1 indicated whether the instruction was an input instruction. In memory mapped systems, tag T1 is used to indicate whether the instruction is a static relevant instruction or a dynamic relevant instruction (the reason behind such an interpretation was discussed in the previous section). Note that since the IBR traps only relevant instructions (static and dynamic), an instruction whose relevancy bit (within the opcode field) is not set is obviously a dynamically relevant instruction. Setting tag T1 therefore, involves a simple bit comparison.

- On the data buffer side, each time main memory is accessed for read/write by a relevant instruction, the CPU activates the CPU REL. line indicating to Snoopy that current memory access is relevant. Snoopy then strobes the value read/written into its data buffer, DBR. CPU REL line therefore is used during the instruction fetch and data fetch cycles.

### 5.2.19 Summary

In this section we described the necessary hardware support for monitoring the sequence of relevant instructions for a type-I architecture with an I/O mapped and a memory mapped architecture. Note that the one bit reservation for indication of relevancy(in the opcode) and the corresponding support from the compiler to set this bit for static relevant instructions is assumed and will not be mentioned for each architectural case. The following is a summary of the results:

- Non real-time relevant instructions
  - I/O mapped systems: No additional hardware support (in terms of additional lines on the CPU-Main memory bus) is necessary.
  - memory mapped systems: One additional bus line, MEM REL.
- Weak real-time relevant instructions: one additional bus line (driven by CPU), CPU REL and an additional instruction (JUMP_REL).
- Strong Real-time relevant instructions: Same as for weak real time.

## 5.3 Type-II Architecture

In this section we will outline the architectural enhancements necessary for monitoring the sequence of relevant instructions, for each of the three behavior classes, for a type-II architecture.

### 5.3.1 Target CPU Organization

In the previous section, a very simple organization for the target processor. In this section, we shall add a new dimension of complexity and study how this affects the activity of monitoring. In particular, we will assume the target processor has an on-chip instruction buffer. In this organization, the target processor overlaps the fetch and execute phases of the program. Most programs exhibit spatial locality of reference, i.e, if instruction i is currently being executed then it is very likely that the next instruction will be the one that immediately follows i (in memory). Having an on-chip instruction buffer allows the target processor to perform look-ahead fetches, i.e, fetch the next instruction (since it most likely to be executed) while the current one is being executed. As a result of this overlap, the apparent time to execute an instruction is reduced.

Figure 5.8 depicts the organization of the target processor. A typical instruction execution goes through the following phases:

- Instruction Fetch Phase: The instruction is fetched from the on-chip instruction buffer.
- Decode Phase: The instruction is decoded to determine if it requires any operands to be fetched from memory. If so, the operands are fetched from memory.
- Execute Phase: The instruction is executed and the results stored appropriately.
- Memory fetch: Instructions are fetched from memory whenever the cpu-main memory bus is free. Instruction fetched from memory are then stored into the on-chip buffer

As pointed out earlier, the above phases are not carried out in strict sequence. Since instructions are fetched before they are actually needed, there is a possibility that occasionally an instruction that is fetched might not be executed. This could happen

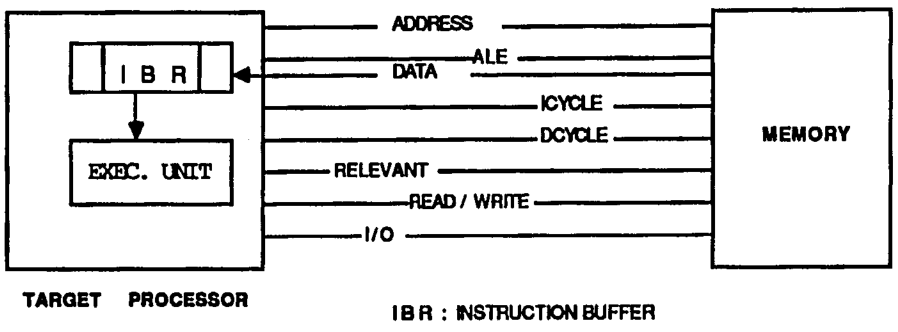


Figure 5.8. Target Processor Organization

because of branch instructions that could alter an otherwise sequential flow of control. As a direct consequence, the monitoring hardware described in the previous section is inadequate for the above target processor organization. More clearly, there must be a means to determine whether a particular instruction that was fetched from memory was finally executed or not. In this section we will discuss the nature of support necessary to monitor the sequence of relevant instructions for processor organizations described above.

### 5.3.2 Assumption

During the discussion of the architectural issues, we shall be assuming that the underlying machine has the following features.

- All instructions are fixed-length instructions. The length of the instruction is no more than the width of the data bus.
- The machine is a load-store architecture.
- Every machine instruction has at most one memory operand.

- Operands for an instruction are fetched only after all the operands for the previous instructions have been fetched.
- In machines that have an on-chip instruction pipeline, there is no shuffling of instructions within the pipeline, i.e, we will assume that instructions are NOT executed out of order.
- Input/output are accomplished through explicit I/O instructions, i.e, the system is I/O mapped (as opposed to memory mapped).
- The system does not have a DMA capability.
- The ISR does not share registers with the interrupted thread, i.e, all registers are saved before switching the context to the ISR.
- No on-chip timer i.e, it is assumed that the timer is an off-chip I/O device.

### 5.3.3 Structure of Monitoring Hardware

Fig 5.9 shows the organization of Snoopy for the target processor architecture shown in Figure 5.8. The organization of Snoopy is, for the most part, similar to that described in the previous section. There are however some differences which we shall describe below.

- ARIB: Auxiliary Relevant Instruction Buffer
  The need for this buffer which lies between the INPUT and OUTPUT pipelines stems from the fact that, in type-II architecture, an instruction that is fetched may not be finally executed. This in turn could happen because of an successful intervening branch that branches to an address that lies after the instruction in question or because of asynchronous interrupts that might occur before the instruction in question is executed.

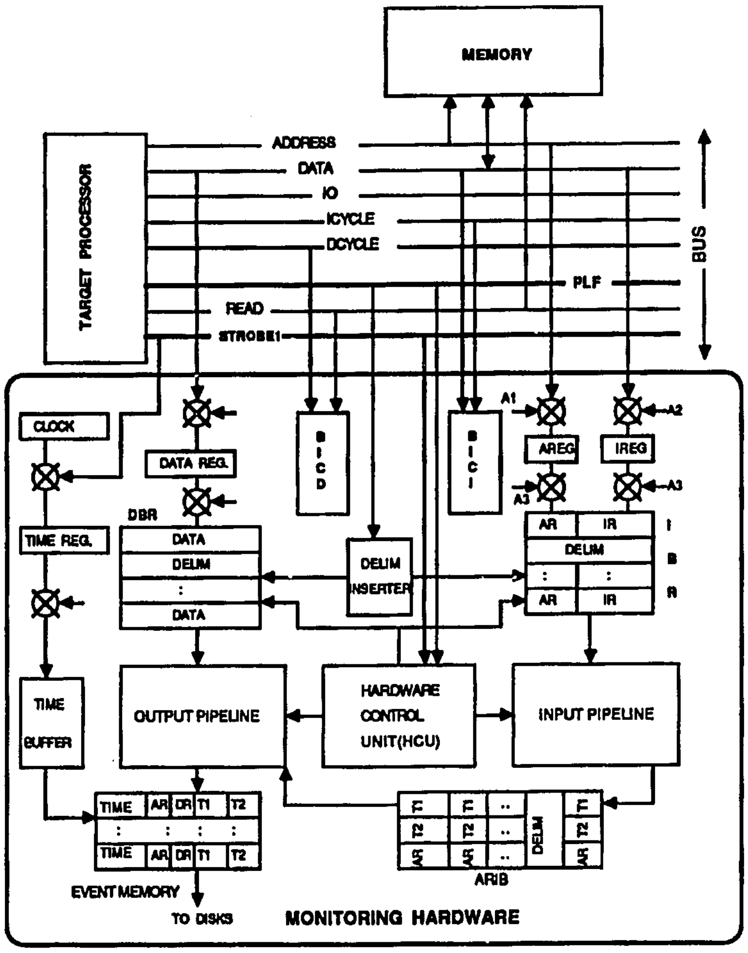


Figure 5.9. Snoopy-Target interface

All relevant instructions that are fetched by the target processor from memory are strobed by the BICI into the IBR, as described in the previous section. The only difference between the entries in IBR in this case and in the previous case is that now the IBR can contain two types of entries (Relevant instructions and Delimiter entries which we shall explain later). Each entry in the IBR is appropriately tagged by the input pipeline and is moved to the ARIB, where the instructions wait either to be discarded or logged into event memory by the output pipeline. The decision to discard or to log is made by the controller based upon certain lines on the bus, which will be explained in detail later.

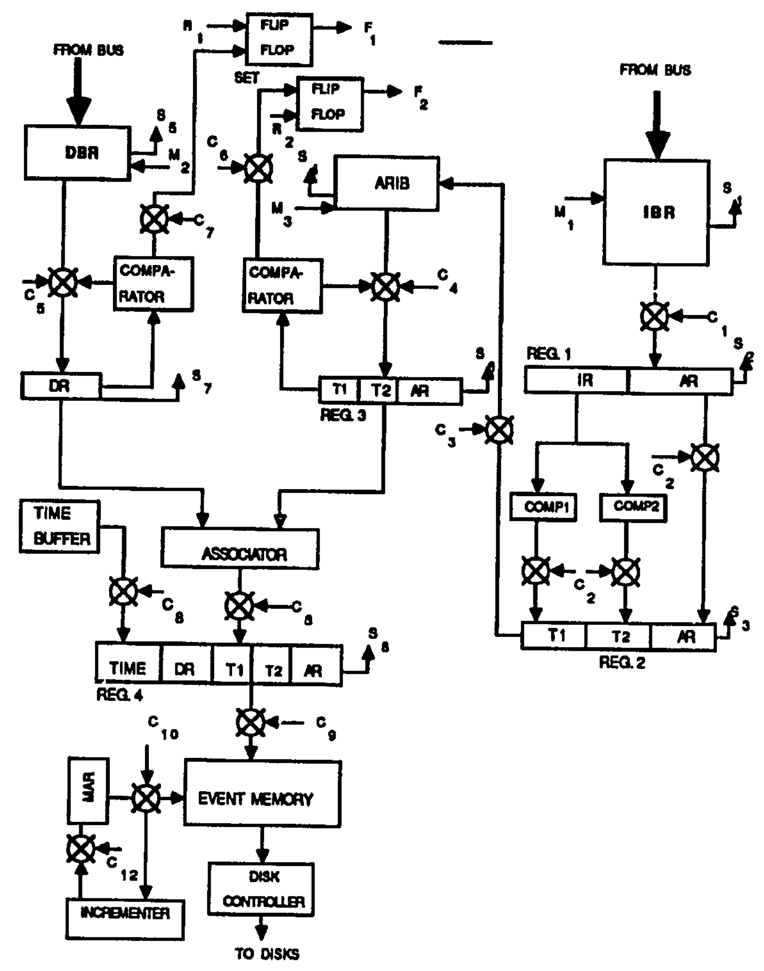


Figure 5.10. Organization of Monitoring Hardware

- Delimiter Inserter:

  A delimiter is a special pattern that is different from all instructions, so that it can be distinguished from the other entries in the IBR. It is used to delimit entries within IBR that might later on be discarded (This aspect will be explained in detail later).

The darkened bus lines shown in Figure 5.10 are the modifications being suggested by us, to monitor the sequence of relevant instructions for a type-II architecture. The following is a description of each of these lines.

- PLF: PipeLine Flush

This line is driven by the processor and is used to indicate to external hardware that the on-chip instruction pipeline has been flushed. As shown in Figure 5.10, this signal is sent to the hardware control unit, which sequences the events that are to be logged. This signal is used by the hardware control unit to decide whether to discard an entry in the ARIB or to log it.

- Strobe_1:

  This line is also driven by the processor to indicate to the monitoring hardware that it has just executed a relevant instruction. Note that this line is different from the "Relevant Line" introduced earlier. The relevant line is used to indicate the *fetching* of a relevant instruction from memory, whereas, strobe_1 is used to indicate the *execution* of a relevant instruction. The need for this distinction arises because, in a type-II architecture, fetching an instruction does not necessarily imply that it is going to be executed. This signal is routed to the hardware control unit which decides which event to log into the event memory next.

Note that all these units belonging to the monitoring hardware, operate in parallel with the target processor and do not in any way interfere with the execution of the target processor.

### 5.3.4 Data Flow

The process of logging relevant instructions, is almost the same as that described for the previous architecture. BICI samples the appropriate relevant line and strobes in all relevant instructions (both dynamic and static) into Snoopy's own instruction buffer. BICI does this by appropriately controlling the signals A1, A2 and A3. Apart from relevant instructions, the IBR can contain *delimiter* entries as shown in Figure 5.9. A delimiter entry is a special pattern of bits that differentiate it from all other relevant instructions. Recall from Figure 5.8 that an instruction fetched from memory may not be executed. This could happen because of successful branch instructions or asynchronous interrupts that are serviced before the instruction in question is executed. If a successful branch or interrupt does occur, then Snoopy has to purge some of the entries within its own instruction buffer. Delimiter entries are used to help Snoopy identify how far down the instruction buffer (or Auxiliary relevant instruction buffer) it has to purge.

Each time the target processor flushes its on-chip instruction buffer, it asserts the PLF line on the bus. The *delimiter inserter* continuously monitors this line and inserts a delimiter entry into IBR when the PLF line is asserted. This information about the pipeline flush is also recorded by the HCU (Hardware Control Unit), which actually controls the purging of Snoopy's buffers at a later time (this aspect will be explained later). The delimiter entries and the normal relevant instruction entries leave the IBR for the input pipeline, where they are appropriately tagged. The tag setting mechanism is exactly the same as that described for the previous architecture, i.e, tag T1 is set to 1 if the relevant instruction is an input instruction and T2 is set if the instruction is a CHKPT instruction. For a delimiter entry, both are set to 0. The organization of the input pipeline is exactly the same as that described for the previous architecture. After leaving the input pipeline, the tagged entries are queued up in another buffer, called ARIB (Auxiliary Relevant Instruction Buffer), where they await either to be discarded or logged into event memory. The process of logging an entry into the event memory is the same as that described for the earlier architecture.

The decision to discard or to log an entry from ARIB, is made by the controller. The details of how the controller makes these decision will be a topic of discussion in a separate section. As shown in Figure 5.10, the two comparator units near ARIB and Data buffer responsible for discarding entries from the corresponding buffers. The comparators compare the entry from ARIB (or DBR) with a delimiter entry pattern. Their output is routed to a flip-flop through a control signals $C_6$ and $C_7$. The success or failure of the comparison is indicated to the HCU through the outputs $F_1$ and $F_2$. This feedback to the HCU is necessary because it has to keep generating signals to purge entries in ARIB and DBR until the delimiter entries are found in each buffer. When a delimiter entry is found, the corresponding flip-flop is set.

### 5.3.5 Hardware Control Unit

The design of the controller for Snoopy for a type-II architecture is different from that for a type-I architecture. In a type-I architecture, every relevant instruction that was fetched had to be executed. Consequently the controller was designed to simply dequeue entries from the relevant instruction buffer and log them into event memory. However, the design of a controller for a type-II architecture is not as simple because of differences in the architecture.

The complexity in the design of a controller for Snoopy for monitoring a type-II architecture stems from the following reasons:

- As explained earlier, a type-II architecture performs "look-ahead" instruction fetching and stores these instructions in an on-chip instruction buffer. All fetches from main memory are visible to Snoopy and it maintains its own replica of the on-chip instruction buffer (except that it logs only relevant instructions). The target processor, on the other hand, may flush its on-chip instruction buffer for reasons discussed earlier. The act of flushing is communicated to Snoopy via the PLF line on the bus. The controller for its part must log the signal on the PLF line and accordingly clear Snoopy's instruction buffer to make it consistent with the target processor's instruction buffer. Note that the logging of relevant instructions and the clearing of the instruction buffer must be appropriately sequenced by the controller. This aspect was not needed in the controller for a type-I architecture.

- Another source of complexity in a type-II controller is in the implementation of clearing buffer entries corresponding to a PLF signal. In general the controller does not know ahead of time the number of entries to be cleared from the instruction buffer. All entries up to the delimiter entry are cleared. In order to implement such a capability, the controller must be designed as a feedback controller.

- A third source of difficulty (which has more to do with the behavior class than with the architectural type) is in handling multiple buffers when monitoring for a strong real-time behavior. In a type-I architecture (non real-time and weak real-time), all relevant instructions were accumulated in one place, namely, the relevant instruction buffer. In the case of a strong real-time behavior, the buffer for static relevant instructions and header actions is different from the TTP buffer (merging the two buffers will create an even greater complexity in buffer management). With multiple buffers, the controller has to sequence the activity of logging between the various buffers. Note that this latter difficulty also exists while designing a controller for strong real-time behavior for a type-I architecture.

Having explained the differences between the controllers for a type-I and a type-II architecture, we will now describe the design for a type-II architecture. The operation of the controller can be likened to that within a general purpose processor. Instructions within the processor are decoded and appropriate control signals generated to effect its execution. In a similar vein, the target processor events like execution of a relevant instruction or a pipeline flush, are recorded into an event buffer as shown in Figure 5.11. The event buffer is serviced by the controller in FIFO order. The controller decodes each entry within the event buffer (to check if it is a relevant instruction event

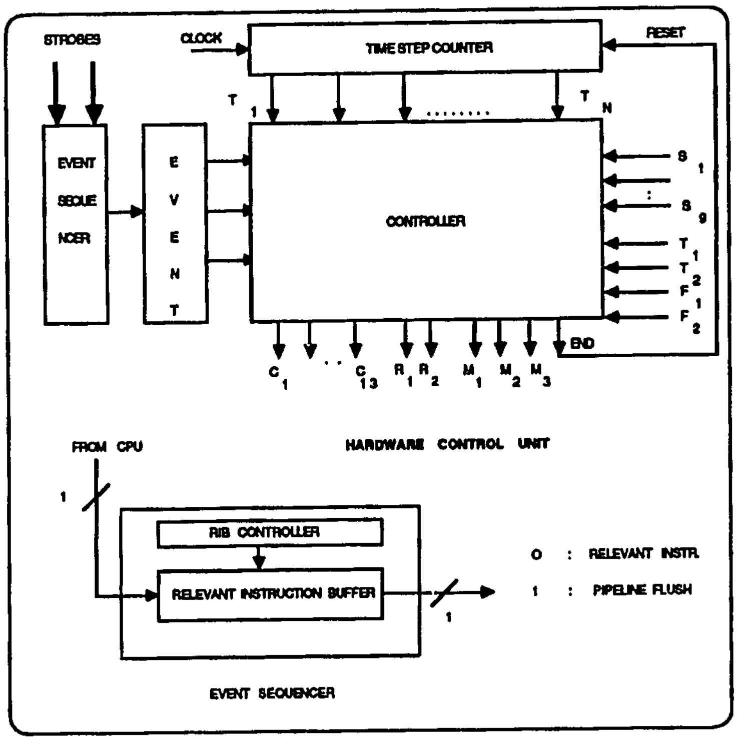


Figure 5.11. Type-II Hardware Control Unit

or a pipeline flush) and generates the appropriate signals(as described below) to the rest of Snoopy in order to log an instruction of that event type into event memory. As an example, if the entry within the event buffer is a “relevant instruction type”, then the controller generates control signals to read entries from ARIB (and DBR if necessary) and log the resulting entry into event memory. On the other hand, if the entry in the event buffer is a pipeline flush, then the controller generates signals to drain all entries within ARIB and DBR until a delimiter entry is reached in each buffer. Control signals $F_1$ and $F_2$ are fed back to the controller so that it knows if a delimiter entry has been reached. While doing all this the controller also generates signals to advance entries within the input pipeline.

### 5.3.6 Control Signals

In this section, we will describe the sequence of control signals that need to be generated in order to log a relevant instruction and handle pipeline flushes.

#### 5.3.6.1 Control Signals for logging a Relevant Instruction

**Time Step 1** : $C_1$, $C_3$, $C_8$, $C_{12}$

All the above signals are generated(in parallel) during this time step (see Figure 5.10). We list below specifically what each signal accomplishes, in a register-transfer level language.

$C_1$: IBR $\rightarrow$ REG. 1 ; dequeues a relevant instruction from the relevant instruction buffer.

$C_3$: REG. 2 $\rightarrow$ ARIB ; Loads a previously processed and tagged relevant instruction into the auxiliary relevant instruction buffer.

$C_8$: ASSOCIATOR $\rightarrow$ REG. 4 & Time Buffer $\rightarrow$ REG. 4 ; Time stamps a relevant instruction along with input data (if any) and moves it into REG. 4

$C_{12}$: MAR $\leftarrow$ (MAR + 1) ; Prepares the Memory address register for storing the next entry into event memory

**Time Step 2** : If T1=1, then $C_5$ (T1 in REG. 3)

$M_1$, $M_3$, $C_2$, $C_9$, $C_{10}$

During this time step, T1 in REG. 3 is checked to see if the current relevant instruction that is going to be logged is an input instruction. If it is, then the entry in DBR reflects the input corresponding to this instruction and so $C_5$ is asserted to dequeue an entry in DBR and load it into DR.

$C_5$: DBR $\rightarrow$ DR ; asserted only if the relevant instruction currently being processed is an input instruction.

$C_2$: REG. 1 $\rightarrow$ REG. 2 ; Processes a relevant instruction from IBR by setting its tag bits appropriately.

$C_9$: REG. 4 $\rightarrow$ Event memory ; Logs the previously processed relevant instruction in REG. 4 into event memory.

$C_{10}$: This signal provides the memory address (in event memory) at which the above entry should be logged.

$M_1$: IBR Buffer management ; All entries in the buffer are shifted down by an entry in order to make room for the next entry from the bus.

$M_3$: Buffer Management of ARIB ; All entries in ARIB are shifted forward by an entry in order to make room for the next processed and tagged relevant instruction from REG. 2

**Time Step 3** : $C_4$, If T1=1 then $M_2$

END

$C_4$: ARIB $\rightarrow$ REG. 3 ; Load the next relevant instruction to be processed

$M_2$: If the previous value of T1=1 then generate this signal to perform buffer management of DBR in order to make room for the next entry from the bus.

END: This signal is asserted to indicate the end of time steps for logging a relevant instruction.

Note that all the above time steps are initiated by the controller in response to a "Relevant Instruction entry type" in its event buffer.

### 5.3.6.2 Control Signals for Pipeline Flush

The following time steps are initiated by the controller in response to a "PLF event" in its event buffer. Essentially the controller generates signals to read off entries from ARIB and purge them until a delimiter entry is reached. The feedback about whether a delimiter has been reached is provided through the control signals $F_1$ and $F_2$.

**Time Step 1:** $C_1$, $C_3$, If not($F_1$) then $C_7$, If not($F_2$) then $C_6$

$C_1$: Process next instruction in IBR while purging of ARIB is going on.

$C_3$: Load the previous tagged relevant instruction into ARIB

$F_1$: Initially this is not set. If it is not set, it means the delimiter entry has not yet been reached. $C_7$ is then asserted so that the output of the comparator (which compares the entry to a delimiter entry) can be used to set the flip-flop $F_1$ for the next round.

$F_2$: Similarly, $F_2$ is initially set to 0 and as long as it is not asserted, $C_6$ is used to channel the output of the comparator to the flip-flop, as shown in Figure 5.10.

**Time step 2:** If not($F_1$) then $C_5$,

If not($F_2$) then $C_4$

$C_2$, $M_1$

If $F_1$ and $F_2$ are not set, then generate $C_4$ and $C_5$ to read off the next entries in DBR and ARIB for processing. In the mean time process the instruction read from IBR in time step 1 and perform buffer management of IBR.

**Time Step 3:** $M_3$, If ($F_1$ and $F_2$) then R1, R2, $C_4$, $C_5$, END Else Repeat all the above time steps.

Perform buffer management of ARIB by asserting $M_3$.

If both $F_1$ and $F_2$ are set (then delimiter entries have been reached in both buffers, ARIB and DBR) then clear the flip-flops by generating the reset signals $R_1$ and $R_2$. $C_4$ and $C_5$ are asserted to read off the next entries after the delimiters in the respective buffers. If any one of $F_1$, $F_2$ is not set, then there is at least one buffer where the delimiter entry has not been reached, so repeat the above time steps.

## 5.4 Type-III Architecture

A type-III architecture is identical to that of a type-II architecture except that the processing of instructions within the processor (decoding, fetching operands etc) is pipelined. More clearly, while an instruction is being executed, operands for the next instruction are fetched from memory while a third instruction is decoded. A type-II architecture exploited the potential parallelism between fetching an instruction and processing another instruction. A type-III architecture goes one step further in parallelising the various stages involved in processing an instruction, namely, decoding, fetching of operands from memory and execution.

Externally, however, as far as Snoopy is concerned, there is no difference (except in speed of instruction execution) between a type-II and a type-III architecture. From a monitoring standpoint, the time between the execution of two instructions is less in type-III architectures compared to type-II architectures. This is because in a type-III architecture, the execution phase of an instruction is itself pipelined (while an instruction is being executed, the operands of the next instruction are being fetched in parallel. This leads to a shorter time gap between the completion of one instruction and the next). As a result the line on the bus used to indicate the execution of a relevant instruction (Strobe_1, in the case of a type-II architecture) will be activated more frequently in a type-III architecture.

## 5.5 Type-IV Architecture

A type-IV architecture is similar to a type-III architecture, except that it has off-chip instruction and data caches. The processor fetches its data and instructions from the caches instead of from main memory. A cache access is a lot faster than a main memory access because of the following:

- The memory technology used in the construction of the cache is better than that used in main memory. The cache is a lot smaller (in storage capacity) than the main memory, thus making the use of improved memory technology afordable for caches.

- Reduced Load: The bus between the cache and the CPU is dedicated for communication between the two. On the other hand, the CPU-Main memory bus is shared by other units, e.g, caches, i/o units etc. This reduced load, on the CPU-Cache bus, translates to smaller signal delays and improved response time.

From a monitoring standpoint, it is necessary for Snoopy to be placed between the CPU and the cache, rather than between the CPU and main memory. This is because instruction and data are now fetched from the caches. However, hooking up Snoopy on the CPU-cache bus would increase the load on that bus and result in poorer performance for the target processor.

In a type-iv architecture, the time difference between the completion of one instruction and the next is smaller than that in a type-III architecture. From a monitoring standpoint therefore, Strobe_2 is activated more often in a type-IV architecture than it is in a type-III architecture. As more and more instructions get executed per unit time, a point may be reached where Strobe_2 cannot keep up with the pace (i.e, the time to activate the Strobe is greater than the time interval between the completion of instructions). In such an event, an additional on-chip buffer would be needed to store the strobes and a controller would also be needed to transmit the contents of this buffer (via the Strobe_2 line) to Snoopy's controller. As the rate of execution increases, therefore, additional hardware support is necessary to guarantee a strong real-time behavioral replay. There is therefore a penalty for requiring strong real-time behavior reproducibility as opposed to weak real-time reproducibility.

## 5.6 Summary

In this chapter we designed the hardware support necessary to monitor each of

ARCHITECTURAL TYPE

BEHAVIORAL CLASSES

| | TYPE-I | | TYPE-II | | TYPE-III/IV | |
|---|---|---|---|---|---|---|
| | I/O | MEM. | I/O | MEM. | I/O | MEM. |
| NON REAL-TIME | S | S, MR | S, ST1, PLF | S, MR, ST1, PLF | S, ST1, PLF | S, MR, ST1, PL |
| WEAK REAL-TIME | S, CR JR | S, MR, CR, JR | S, ST1, PLF CR, JR | S, MR, ST1 PLF, CR, JR | S, ST1 PLF, CR, JR | S, MR, ST1 PLF, CR, JR |
| STRONG REAL-TIME | S, CR, JR | S, MR, CR, JR | S, ST1, PLF CR, JR, ST2 | S, MR, ST1 PLF, CR, JR ST2 | S, ST1, PLF CR, JR, CT2 | S, MR PLF, CR ST2, ST JR / B* |

| | |
|---|---|
| S : | A BIT RESERVED IN THE OPCODE, SET BY THE COMPILER, FOR INDICATING STATIC RELEVANCY |
| MR : | A LINE ON THE BUS TO INDICATE THE RELEVANCY OF A MEMORY ACCESS. DRIVEN BY THE CPU |
| CR : | A LINE ON THE BUS TO INDICATE THE RELEVANCY OF AN ENTRY ACTION. DRIVEN BY THE CPU. |
| JR : | A NEW INSTRUCTION, JUMP_REL FOR INDICATING THAT THE SUBSEQUENT INSTRUCTION IS RELEVANT. |
| PLF : | A LINE ON THE BUS TO INDICATE TO SNOOPY THAT THE ON-CHIP INSTRUCTION BUFFER/PIPELINE HAS BEEN FLUSHED |
| ST1 : | A LINE ON THE BUS TO INDICATE TO SNOOPY ABOUT THE EXECUTION OF EACH RELEVANT INSTRUCTION. DRIVEN BY THE CPU. |
| B* : | AN ON-CHIP BUFFER USED TO STORE ALL THE STROBES FROM THE TARGET PROCESSOR, FOR TRANSMISSION OVER ST2 TO SNOOPY'S CONTROLLER. |
| ST2 : | A LINE ON THE BUS TO INDICATE THE EXECUTION OF EVERY INSTRUCTION. |

Figure 5.12. Summary of Architectural Enhancements

the relevant instructions described in chapter 3. The nature of the hardware support depended on the type of relevant instruction being monitored and the architecture of the target system. Consequently we defined Type-I, Type-II, Type-III and Type-IV architectures and then proceeded to define the necessary hardware support for each of them. Figure 5.12 describes, in tabular format, the type of hardware support necessary for the various architecture types and behavior classes. For all the architectures, compiler support in the form of detecting static relevant instructions and setting a bit in their opcode is essential to the whole scheme.

In conclusion, architectural features such as caches complicate the task of reproducing a strong real-time behavior. We will not discuss architectures where the caches are resident on the chip. Caches, in general, are not used in real-time applications, because of the potential unpredictability in execution time incurred by their use. In real-time applications, predictability is much more important than speed of execution. The potential payoff, of reproducing a strong real-time behavior, during debugging may not justify the cost of additional hardware needed to implement such a facility.

# CHAPTER 6

# EXAMPLES

## 6.1 Finding Maximum

In this section we will discuss simple non serializability errors in a parallel maximum finding algorithm. Consider the following algorithm for finding the maximum of an array of positive integers.

Let the size of the array be N. The task of finding the maximum among these N numbers is divided into N distinct processes. The i-th process holds the i-th number in the array as its own private copy. Let MAX denote the variable that finally holds the maximum of all these numbers. MAX is a variable that is shared between all these processes and is initialized to 0. Each process compares its private copy with that of MAX. If its private copy is greater than MAX then it sets MAX to equal its own copy. For simplicity, we will choose N to be 2 i.e, an array of size 2. We shall write the code in Ada.

```
with TEXT_IO ; use TEXT_IO ;
Procedure FIND_MAX is
     package IIO is new INTEGER_io(INTEGER) ; use IIO ;
N : constant INTEGER := 2 ;
type NUM_RANGE is INTEGER range 1..N ;
subtype ID_TYPE is NUM_RANGE ;
```

```
 type ARRAY_OF_NUM is array(1..N) of INTEGER ;
 MAX : INTEGER := 0 ;
 NUM_ARRAY : ARRAY_OF_NUM(2) ;
 task type SLAVE_TYPE is
      GET_ID(ID : in ID_TYPE) ;
 end SLAVE_TYPE ;
 task body SLAVE_TYPE is
     MY_ID : ID_TYPE ;
 begin
     – Get own id from main —
     accept GET_ID(ID : in ID_TYPE) do
          MY_ID := ID ;
     end accept ;
     OWN := NUM_ARRAY(MY_ID) ;
     if OWN > MAX then
     MAX := OWN ;
     endif ;
end SLAVE_TYPE ;
type SLAVE_ACC is access SLAVE_TYPE ;
type SLAVE_ACC_ARR is ARRAY(NUM_RANGE) of SLAVE_ACC ;
SUB_TASKS : SLAVE_ACC_ARR ;
begin
    Declare
    begin
      for I in NUM_RANGE loop
             SUB_TASKS(I) := new SLAVE_ACC ;
             SUB_TASKS(I).GET_ID(I) ;
```

```
        end loop ;
      end ; -- End of declare block
  new_line ; PUT("The maximum is : "); PUT(MAX) ; new_line ;
  end ; -- end of Main program
```

Note that in the above example, MAX is a shared memory location i.e, it is accessible to more than one threads of control. As a consequence access to it must be protected. In the program written above, MAX could be assigned in parallel by both tasks and the final outcome would depend on which task wrote last into it. The program therefore has a timing bug i.e, it may or may not work. To see this more clearly, we have to look at the machine instruction sequence corresponding to the two sub_tasks. Most of machine code of the sub-tasks is irrelevant to the bug, so we will expand only the relevant parts that operate over the shared memory location, MAX.

```
assembly code for sub_task(1)        assembly code for sub_task(2)
                  :                                    :
    10.   Load  reg1,OWN                10.   Load  reg1,OWN
    11.   CMPR  reg1,MAX                11.   CMPR  reg1,MAX
    12.   BLEQ  Fin                     12.   BLEQ  Fin
    13.   MOVE  reg1,MAX                13.   MOVE  reg1,MAX
          :                                   :
    Fin   END                           Fin   END
```

Note that sub_task(1) and sub_task(2) are parallel tasks. In general, their executions could be interleaved. We will see shortly that a change in the interleaving could lead to a different behavior and therefore the above program has a timing bug. To see this, let the two numbers in the array (num_array) be 10 and 20. The program starts out with MAX having a value of 0. Sub_task(1)'s copy of OWN contains 10 and sub_task(2)'s copy of OWN contains 20. To describe an interleaving, we shall use the notation (i,j) to denote the j-th instruction in the i-th task. Consider the following interleaving in the execution of the above tasks. ...(1,10)(2,10)(1,11)(2,11)(2,13)(1,13)... With this interleaving, the final value of MAX will be 10, which is incorrect. However the interleaving ...(1,10)(1,11)(1,12)(1,13)(2,10)(2,11)(2,12)(2,13)..., works fine and gives the correct result MAX=20. Therefore, not all interleavings are correct.

## 6.1.1 Our Approach

To show how our approach can detect the above timing error, we will assume an incorrect interleaving, namely ...(1,10)(2,10)(1,11)(2,11)(2,13)(1,13)... The event table corresponding to such a trace is as shown in the table below.

| Tags T1,T2 | Process | Instr. | Input Value |
|---|---|---|---|
| 00 | sub_task(1) | 10 | - |
| 00 | sub_task(2) | 10 | - |
| 00 | sub_task(1) | 11 | - |
| 00 | sub_task(2) | 11 | - |
| 00 | sub_task(2) | 13 | - |
| 00 | sub_task(1) | 13 | - |

**Event table for Find Max program**

The variable "max" is shared across processes P1 and P2, therefore, all operations over them are relevant. Information in the event table above is collected during phase I of execution of the program. Before phase II of the execution is begun, processes P1 and P2 of the program are modified by inserting calls to a monitor M. The modified program is as follows :

```
        assembly code for sub_task(1)          assembly code for sub_task(2)
                        :                                       :
        10.     Load    reg1,OWN               10.     Load    reg1;OWN
        10.1    Call    MONITOR                10.1    Call    MONITOR
        11.     CMPR    reg1,MAX               11.     CMPR    reg1,MAX
        12.     BLEQ    Fin                    12.     BLEQ    Fin
        12.1    Call    MONITOR                12.1    Call    MONITOR
        13.     MOVE    reg1,MAX               13.     MOVE    reg1,MAX
                :                                      :
        Fin     END                            Fin     END
```

Notice that the modification involves introducing a "Call Monitor" instruction before each relevant instruction as determined from the event table. During replay the monitor M, ensures that the execution during the second phase is restricted so that the sequence of relevant instructions (during phase II) is exactly that specified by the event table. Note that during the second phase, the programmer can also set breakpoints to examine the flow of control more closely. The algorithm for the monitor is the same as that given in section 3.19.1.

## 6.2 Timing Error : Linked List Example

Linked lists and queues are data structures that are frequently used in numerous applications. Careless use of these structures could give rise to very frustrating errors, especially in an application that has considerable parallelism, e.g, airline reservation systems, inventory control systems etc.

Consider the linked list shown in figure 6.2. It is accessed through a pointer, Header, that points to the first record and which in turn points to yet another record.

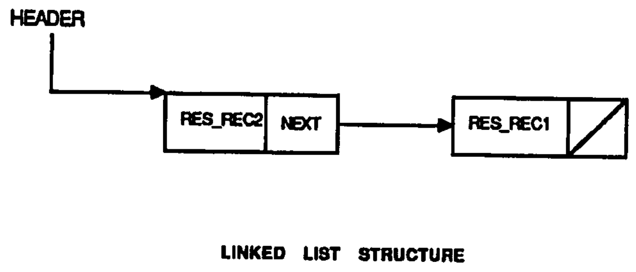


Figure 6.1. Linked List Structure

This could represent, for example, in an airline reservation system, the list of people traveling on a particular flight between some two cities. Suppose again that this is a data structure that is shared between a number of parallel processes representing various travel agents. To make reservations, the travel agents insert and delete reservations from this linked list (in parallel). Consider the following high level code in Pascal used to make reservations for passengers.

```
Code for reservations          Code for reservations
Travel_agent(1) :              Travel_agent(2) :
    ⋮                              ⋮
    P := New(Cust_rec)             Q := New(Cust_rec)
    P.next := Header               Q.next := Header
    Header := P                    Header := Q
    ⋮                              ⋮
```

Note that both the above two tasks use the same linked list to operate upon. Both the tasks operate on the list without protecting it in a mutual-exclusion construct. This could result in some reservations being lost. To see this more clearly, consider the following machine-code translation of the above high level program. Since header is the only shared variable, we will only be concerned with instructions operating over

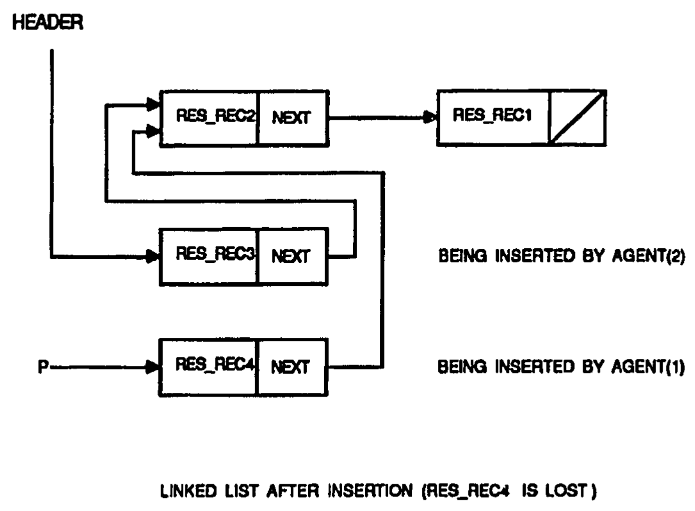

HEADER
RES_REC2
NEXT
RES_REC1
RES_REC3
NEXT
BEING INSERTED BY AGENT(2)
P
RES_REC4
NEXT
BEING INSERTED BY AGENT(1)
LINKED LIST AFTER INSERTION (RES_REC4 IS LOST)

Figure 6.2. Linked List After Insertion

it.

```
m/c code for reservations          m/c code for reservations
Travel_agent(1) :                  Travel_agent(2) :
   :                                  :
   20.    Mov    R1, Header           20.    Mov    R1, Header
   21.    Mov    P.Next, R1           21.    Mov    Q.Next, R1
   22.    Mov    R1, P                22.    Mov    R1, Q
   23.    Mov    Header, R1           23.    Mov    Header, R1
   :                                  :
```

Since the list is not protected, instructions within each task can be arbitrarily interleaved. Consider the following interleaving : ...(1,20) (1,21) (2,20) (2,21) (2,22) (2,23) (1,22) (1,23).... The resulting linked list will look as shown in figure 6.2, which results in one reservation being lost. On the other hand, if all statements of agent(1) precede those of agent(2) or vice-versa, there will be no error.

The above behavior can easily be reproduced by our approach, using the event table and modified program shown below.

| Tags T1,T2 | Process | Instr. | Input Value |
|---|---|---|---|
| 00 | agent_task(1) | 20 | - |
| 00 | agent_task(2) | 20 | - |
| 00 | agent_task(2) | 23 | - |
| 00 | agent_task(1) | 23 | - |

**Event table for Linked List program**

The modified program is as follows :

```
m/c code for reservations            m/c code for reservations
Travel_agent(1) :                    Travel_agent(2) :
    :                                    :
    19.1   Call   Monitor                19.1   Call   Monitor
    20.    Mov    R1, Header             20.    Mov    R1, Header
    21.    Mov    P.Next, R1             21.    Mov    Q.Next, R1
    22.    Mov    R1, P                  22.    Mov    R1, Q
    22.1   Call   Monitor                22.1   Call   Monitor
    23.    Mov    Header, R1             23.    Mov    Header, R1
    :                                    :
```

The Monitor ensures that the above modified program executes in a manner such that the sequence of relevant instruction execution during the second phase matches that dictated by the event table. Thus, the error will be repeated each time the program execution is replayed.

## 6.3 Scheduling Errors

In many real-time situations, reproducing the sequence of class I relevant instructions is not sufficient to help the user identify the cause of the error. The reason for missing a deadline, could for example, depend on the incorrect scheduling of tasks. There is a need in these situations to reproduce the class II relevant instruction sequence in order to detect the error. In this section, we shall discuss one such example.

Consider an embedded system used in manufacturing and packaging soaps, as shown in Figure 6.3. The company sells soaps by the carton to major supermarket stores. The soaps come off the soap manufacturing unit and are transported via a conveyor belt (Conveyor belt 1) to an empty carton lying on another conveyor belt (Conveyor belt 2). A sensor located directly above the right end of conveyor belt 1, as in Figure 6.3, senses the passage of each soap and informs the computer via an interrupt. When the number of soaps in the carton reaches 3, conveyor belt 2 is moved forward so that a new empty box is positioned below the end of conveyor belt 1. The computer system controls the movement of the conveyor belts through the appropriate motor (motor1) as shown in Figure 6.3.

We will model the whole system through the use of 3 tasks. The task profile is as shown in Figure 6.4. Suppose that the distance between two soaps on conveyor belt 1 (temporally) is 10 units of time. Every 10 units of time, the sensor sends an interrupt to the computer indicating the passage of a soap. Task A, as shown in Figure 6.4, represents the interrupt service routine corresponding to this sensor interrupt. Suppose, again, that it consumes an estimated 4 units of time (basically the ISR maintains a count of the number of soaps in the carton). Task A can therefore be modeled as a periodic task with period as 10 units of time and an estimated worst case execution time of 4.

Each time the count of number of soaps in the carton reaches its capacity (which is 3 in this case), another task is activated (task C). Task C, is responsible for activating

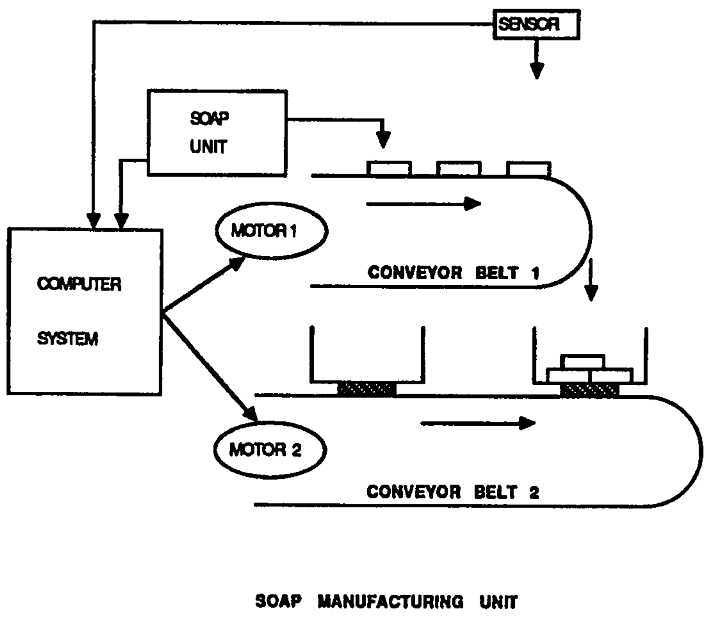


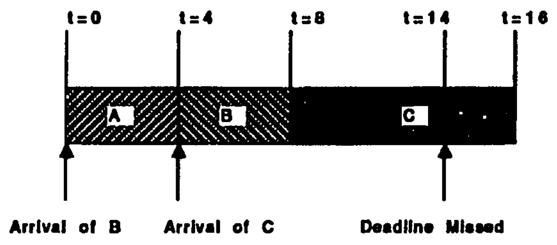
Figure 6.3. Soap Assembly Unit

the appropriate motor and moving conveyor belt 2 forward so that the next empty carton is correctly positioned beneath the end of conveyor belt 1. Task C can be modeled as a periodic task with a period of 30 units of time (time for 3 soaps to fall into the carton). Suppose that the estimated execution time is 8 units of time. Note that task C must be executed within 10 units of time from its arrival (otherwise, a soap on the conveyor belt #1 will either fall into the wrong carton or won't fall into a carton). Task C, therefore, has an associated deadline of 10 units of time from the time that it arrives (Note that its deadline is less than its period).

A third sporadic task, task B, is used for initializing certain elements within the soap manufacturing subsystem. We will not concern ourselves with the details of task B, other than that its arrival is sporadic with a minimum inter-arrival time of 20 units of time and an estimated execution time of 4 units of time. Apart from scheduling these three tasks, the system also performs certain background tasks, which are scheduled during spare time (time left after execution of tasks A,B,C). We will also assume that COUNT is the only memory location that is shared between task A (which increments

| | | PERIOD | EXEC. TIME | DEADLINE | PRIORITY |
|---|---|---|---|---|---|
| ▨ | A | 10 | 4 | 10 | 1 |
| ▧ | B | 20 | 4 | 20 | 2 |
| ■ | C | 30 | 8 | 10 | 3 |

TASK EXECUTION PROFILE OF SOAP MANUFACTURING UNIT



TASK SCHEDULE WITH RATE MONOTONIC PRORITY ASSIGNMENT

Figure 6.4. Task Structure and Schedule

count) and task C (which initializes COUNT to 0). We will also assume that Task B does not have any non real-time relevant instructions.

The scheduling policy used to schedule the 3 tasks is the Rate Monotonic(RM) algorithm [Liu73]. According to this algorithm, the priority of a task is equal to the inverse of its period. Consequently, task A is the highest priority followed by B and then C. Figure 6.4, shows the scheduling of tasks according to rate monotonic algorithm.

Consider a situation where tasks A and B arrive at the same time and C arrives 4 units of time after A, as shown in Figure 6.4. Since A has the highest priority, it is executed first, followed by B and then C. Since C was scheduled after B (in keeping with the RM priority assignment), task C misses its deadline, as shown in Figure 6.4. The deadline was missed not because of any incorrect sequencing of a non real-time relevant instruction, but because of an incorrect scheduling sequence. Note here that if C is scheduled before task B, then the deadline would not have been missed. Note also that this error is difficult to reproduce because the condition of B arriving before C may not happen always.

This could be a very tricky error to detect. Note here, that it is impossible to determine the reason for missing the deadline by just using a non real-time trace. This is because a non real-time trace may not schedule tasks A, B and C in the same order during replay (because as mentioned above, task B does not have any class I relevant instructions). However, a weak real-time trace will reproduce the scheduling sequence (because the first instruction in task B is a header action and hence a class II relevant instruction).With a weak real-time trace, the programmer can see the specific reason why the deadline was missed, namely, because task B was scheduled before task C.

The machine code for processes A and C is as follows.

```
Code for A                          Code for C
     1.    MOVE ..,..                    1.    MOVE ..,..
                :                                   :
     12.   Load   reg1,Count             10.   Clear  reg1
     13.   ADD    reg1,1                 11.   Store  Count,reg1
     14.   Store  Count,reg1                   :
                :                        20.   OUT    reg2,ROTOR2
     Fin   END                           Fin   END
```

The event table corresponding to the task schedule shown in Figure 6.4 is as follows :

| Tags T1,T2 | Process | Instr. | Input Value | Time |
|---|---|---|---|---|
| 00 | A | 1 | - | - |
| 00 | A | 10 | - | - |
| 00 | A | 12 | - | - |
| 00 | B | 1 | - | - |
| 00 | C | 1 | - | - |
| 00 | C | 11 | - | - |
| 01 | C | 20 | - | - |

**Event Table for Soap Manufacturing Unit**

Note, in the above event table that the first instruction in task B is recorded as a dynamic relevant instruction during phase I. During replay the monitor will force the execution of B before scheduling C thus causing C to miss the deadline exactly as in phase I.

## 6.4 Hard Scheduling error

Consider again the example of the soap manufacturing unit as shown in fig 6.3. Assign priorities to periodic tasks according to the rate monotonic algorithm and to sporadic tasks according to deadline monotonic scheduling policy [Sprun89]. Such a scheduling policy will assign the highest priority to task C followed by tasks A and B. So, if every 30 units of time, if task C is scheduled first then its hard deadline will be met. However, though it might be the correct schedule, the deadline could still be missed because of incorrect estimation times for the execution of the three tasks. Determining such an error would require a strong real-time trace during replay, where the sequence of every instruction is reproduced. By subtracting the time of header action in A from that in C, the programmer can conclude the execution time of C and reschedule all the tasks.

# CHAPTER 7

# CONCLUSIONS

## 7.1 Conclusions

In this section, we shall review our achievements and suggest opportunities for further research.

### 7.1.1 Contributions

A fundamental goal of this research was to design a cyclic debugging facility for debugging real-time software in embedded systems. Cyclic debugging, as explained in chapter 2, is conducted over at least two phases i.e, the program is executed once and upon encountering an error, it is re-run one or more times to identify the error. While implementing cyclic debugging for non real-time sequential programs is trivial (in the sense that no monitoring information need be collected during the first phase), it is not so for parallel real-time programs. Implementation of cyclic debugging for parallel programs involves monitoring the program during the first phase and subsequently replaying the program. Within this overall framework our accomplishments are as follows:

- A characterization of various behavior classes for describing the behavior of real-time programs.

- A characterization of various errors that can occur in a real-time program, with processes of the program communicating through shared memory. While [LMC87], [PL88], [LeD86] characterize errors in shared memory parallel programs, none of them address parallel real-time programs.

- Establishing a relationship between the various errors and behavior classes i.e, assuming a particular error type, what execution behavior during replay will guarantee the reproduction of that error?

- Identifying a set of events that should be monitored so as to guarantee that the execution behavior during replay will belong to a particular behavior class.

- Proving the sufficiency of the above events. This is very important since it mathematically guarantees that the monitoring of these events is sufficient to reproduce a behavior belonging to a particular behavior class. We feel this scheme is better than the adhoc monitoring performed by single phase knowledge base/database schemes of debugging [LeD86], [GMGK84], [Sno84], where the sufficiency of information collected is not proved.

- Proposing Architectural Modifications in order to extract the above information in a non-intrusive manner. While non-intrusive monitoring techniques have been developed before, [Fry73], [Wit83], [Pla84], none of them are geared towards replaying the program i.e, monitoring within these systems is not done with an intent to reproduce the behavior of the program. Monitoring in [Wit83], although collected non-intrusively, is in the same vein as [LeD86] where the type of information monitored is adhoc and specified by the user. Our scheme, on the other hand, proposes architectural modifications which will non-intrusively collect sufficient information that will guarantee a behaviorally identical replay.

- Developing a non-intrusive checkpointing strategy to support the cyclic debugging paradigm for programs executing over extended periods of time.

In summary, while numerous solutions have been proposed to debug parallel programs, none of them, to our knowledge, deal with **non-intrusive** debugging of **real-time** programs with the ability to **replay** a behaviorally identical execution.

### 7.1.2 Future Directions

This work concentrates on providing a non-intrusive cyclic debugging scheme for single user uniprocessor systems. An obvious extension is to provide the same for multiprocessor systems with shared memory. Some of the issues are:

- How is execution on a multi-processor system modeled? Is it still a sequence of instructions? We have addressed this question and proved in chapter 3 that a multi-processor execution can be modeled as a linear sequence of instructions.
- Are there any more relevant instructions because of multiple processors?
- What additional bus lines are needed to assist in monitoring the relevant instruction sequence?
- How does the checkpointing scheme described in Chapter 4, extend to a multiprocessor situation? How are all the processors halted at once?
- What about multiple processor systems with arbitrary interconnection topology? Specifically, What are the relevant instructions in such systems?, How is non-intrusive checkpointing achieved in such systems?

### 7.1.3 Experiments

The nature of our solution is such that testing these ideas would require actually building the processor and memory chips with our proposed modifications. Simulating the whole system requires modifying the processor chip to provide for the extra bus lines

BASIC TRACE STATISTICS

| TRACE | REFS. (1000s) | CODE | DATA | READS (DATA) | WRITES (DATA) | PRIVATE (DATA) | SHARED (DATA) | READ SHARED DATA | WRITE SHARED DATA |
|---|---|---|---|---|---|---|---|---|---|
| | | PROPORTION OF TOTAL REFERENCES | | | | | | | |
| PLOVER | 3,604 | .650 | .350 | .287 | .063 | .213 | .137 | .127 | .010 |
| PSPICE | 1,522 | .629 | .371 | .256 | .115 | .283 | .088 | .069 | .019 |
| PUPPY | 2,945 | .544 | .456 | .356 | .100 | .314 | .142 | .128 | .014 |
| TOPOPT | 3,293 | .662 | .338 | .316 | .022 | .196 | .142 | .139 | .003 |

Figure 7.1. Trace Statistics

and extra machine instructions in the form of (JUMP_REL, CHKPT) and providing a compiler that will generate code for the modified processor (generating instructions with appropriate relevant bit coded into them). Even if the herculean task of developing the simulator was accomplished, it would have to be executed in a simulated real-time environment. We were not convinced about the utility of the results that would be obtained through such a simulation and therefore decided against building such a simulator. However, we list below some of the experiments that could prove useful once the whole system is physically built.

- Relevant Events:
  - What percentage of the total number of data references are references to shared memory? Susan Eggers et. al conducted a study that characterized the extent of sharing in parallel programs. They studied various complex parallel programs like PUPPY, PLOVER and PSPICE and obtained trace

statistics to determine the percentage of shared memory references. Figure 7.1 is a reproduction of the results of their trace statistics from their paper [EK88]. It may be noted from the table that total percentage of **shared** reads and **shared** writes is less than 14% for each of the applications (total percentage of shared reads and writes for PSPICE is less than 10%).

  - What is the average time between the occurrence of these events? - This will determine the size of the relevant instruction buffer.

- Checkpointing:
  - What is the frequency of use of the *CHKPT* instruction in normal programs?
  - At a checkpoint instruction, what fraction of the data blocks in the data cache are dirty? - This will immediately determine the wait time for a CHKPT instruction.
  - What are the total number of "1st time" writes to variables, between two checkpoints? - This will determine the total amount of information that needs to be checkpointed.
  - What is the average number of relevant events occurring between two checkpoints? – This will determine the length of the event table.

# BIBLIOGRAPHY

# BIBLIOGRAPHY

[AC81] J. E. Archer and R. Conway. Cope : A co-operative programming environment. Technical Report TR81-459, Dept. Of Computer Science, Cornell University, Ithaca, June 1981.

[ACS84] J. E. Archer, R. Conway, and F. B. Schneider. User recovery and reversal in interactive systems. In *ACM Transactions on Programming Languages and Systems*, pages 1–19, January 1984.

[AM88] William F. Appelbe and Charles E. McDowell. Integrating tools for debugging and developing multitasking programs. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Disributed Debugging*, pages 78–88, 1988.

[And79] S. F. Andler. *Predicate Path Expressions : A high-Level Synchronization Mechanism*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1979.

[Ant88] C. J. Antonelli. *Exception Handling Mechanisms in Ada*. PhD thesis, Department of Computer Science, University of Michigan, 1988.

[Bal69] R. M. Balzer. Exdams : Extendable debugging and monitoring system. In *Proceedings of the 1969 Spring Joint Computer Conference*, pages 567–580, 1969.

[Bat88] P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Disributed Debugging*, pages 11–22, 1988.

[BH83] B. Bruegge and P. Hibbard. Generalized path expressions : A high-level debugging mechanism. In *Proceedings of the ACM SIGSOFT Software Engg. Symposium on High-Level Debugging*, pages 34–44, 1983.

[BW83] P. C. Bates and J. C. Wileden. High-level debugging of distributed systems : The behavioral abstraction approach. *Journal of Systems and Software*, pages 255–264, 1983.

[Chi84] Y. S. Chiu. *Debugging Distributed Computations in a Nested Atomic Action System.* PhD thesis, Laboratory for Computer Science, MIT, 1984.

[CS88] David Callahan and Jaspal Subhlok. Static analysis of low-level synchronization. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Disributed Debugging*, pages 100–111, 1988.

[CW82] R. Curtis and L. Wittie. Bugnet : A debugging system for parallel programming environments. In *Proceedings of the 3rd International Conference on Distributed Computing Systems, IEEE*, pages 394–399, 1982.

[EK88] Susan J. Eggers and Randy H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th IEEE Conference in Computer Architecture*, pages 373–382, 1988.

[FB88] Stu Feldman and C. Brown. Igor : A system for program debugging via reversible execution. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112–123, 1988.

[Fry73] Richard E. Fryer. The memory bus-monitor- a new device for developing real-time systems. In *Proceedings of the AFIPS National Computer Conference*, pages 75–79, 1973.

[FTG88] Richard M. Fujimoto, Jya-Jang Tsai, and Ganesh Gopalakrishnan. Design and performance of special purpose hardware for time warp. In *Proceedings of the 15th IEEE Conference in Computer Architecture*, pages 401–408, 1988.

[GMGK84] H. Garcia-Molina, F. Germano, and W. Kohler. Debugging a distributed conputing system. *IEEE Transactions on Software Engg.*, pages 210–219, March 1984.

[Gor85] Aaron J. Gordon. *Ordering Errors in Distributed Systems*. PhD thesis, Department of Computer Science, University of Wisconsin, 1985.

[Han78] David Hanson. Event association in snobol4 program debugging. *Software-Practice and Experience*, pages 115–129, August 1978.

[HC88] A. A. Hough and Janice E. Cuny. Initial experiences with a pattern-oriented parallel debugger. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Disributed Debugging*, pages 195–205, 1988.

[HHK85] P. K. Harter, D. Heimbigner, and R. King. Idd : An interactive distributed debugger. *IEEE Transactions on Software Engg.*, pages 498–506, March 1985.

[HLMSR85] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In Santosh K. Shrivastava, editor, *Reliable Computer Systems : Collected Papers of The Newcastle Reliability Project*, pages 53–68. Springer-Verlag, 1985.

[Joh78] Mark Scott Johnson. *The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment*. PhD thesis, Department of Computer Science, University of British Columbia, 1978.

[Kor74] Robert R. Korfhage. *Discrete Computational Structures.* Academic Press, 1974.

[LeD86] C. H. LeDoux. *A Knowledge Based System for Debugging Concurrent Software.* PhD thesis, Department of Computer Science, UCLA, 1986.

[LGH80] P. A. Lee, N. Ghani, and K. Heron. A recovery cache for the pdp-11. In *IEEE Transactions on Computers*, pages 546–549, JUNE 1980.

[LMC87] T. J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, pages 471–482, April 1987.

[LR85] R. J. LeBlanc and A. D. Robbins. Event driven monitoring of distributed programs. In *Proceedings of the 5th International Conference on Distributed Computing Systems, IEEE*, pages 515–522, 1985.

[LR88] Paul C. Lewis and Daniel A. Reed. Debugging shared memory parallel programs. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 322–324, 1988.

[MC88] B. P. Miller and J. D. Choi. A mechanism for efficient debugging of parallel programs. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Disributed Debugging*, pages 141–150, 1988.

[PL88] D. Pan and Mark A. Linton. Supporting reverse execution for parallel programs. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129, 1988.

[Pla81] B. Plattner. Monitoring program execution : A survey. *IEEE Computer Mgazine*, pages 76–93, November 1981.

[Pla84] B. Plattner. Real-time execution monitoring. *IEEE Transactions on Software Engg.*, pages 756–764, November 1984.

[Ran75] B. Randell. System structure for software fault-tolerance. In *IEEE Transactions on Software Engineering*, pages 220–232, JUNE 1975.

[RRZ88] R. Rubin, L. Rudolph, and D. Zernik. Debugging parallel programs in parallel. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Disributed Debugging*, pages 216–225, 1988.

[SBN88] D. Socha, M. Bailey, and D. Notkin. Voyeur : Graphical views of parallel programs. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 206–215, 1988.

[SGMW84] R. K. Scott, J. W. Gault, D. G. McCallister, and J. Wiggs. Experimental validation of six fault-tolerant software reliability models. In *Proceedings of the 14th Annual Symposium on Fault-Tolerant Computer Systems*, June 1984.

[Smi84] E. T. Smith. Debugging tools for mesage based communicating processes. In *Proceedings of the 4th International Conference on Distributed Computing Systems, IEEE*, pages 303–310, 1984.

[Sno84] R. Snodgrass. Monitoring in a software development environment : A relational approach. In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, pages 124–131, 1984.

[SY85] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, pages 204–226, August 1985.

[Tay83] R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *CACM*, pages 362–376, May 1983.

[TO80] R. N. Taylor and L. J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engg.*, pages 265–277, May 1980.

[TO86] K. C. Tai and E. E. Obaid. Reproducible testing of ada tasking programs. In *Proceedings of IEEE Second International Conference on Ada Applications and Environments*, pages 69–79, 1986.

[Wit83] C. A. Witschorik. The real-time debugging monitor for the bell system 1a processor. *Software Practice and Experience*, pages 727–743, April 1983.