TIMING ANALYSIS IN BINARY-TO-BINARY TRANSLATION

by

INKYU KIM

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 1997

" Timing Analysis In Binary-To-Binary Translation ," a dissertation prepared by Inkyu Kim in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:

Chair of the Examining Committee

May 23 1997
Date

Commitee in charge: Dr. Zary Segall, Chair
Dr. Arthur Farley
Dr. John Conery
Dr. Nancy Melone

Accepted by:

Vice Provost and Dean of the Graduate School

An Abstract of the Dissertation of

Inkyu Kim for the degree of Doctor of Philosophy

in the Department of Computer and Information Science

to be taken June 1997

Title: TIMING ANALYSIS IN BINARY-TO-BINARY TRANSLATION

Approved: _____
Dr. Zary Segall

Research in TIBBIT(Timing Insensitive Binary to BInary Translation) proposed a methodology to provide timing equivalence as well as semantic equivalence in binary-to-binary translation of real-time applications by inserting synchronization at regular intervals. However, the timing equivalence of programs generated by TIBBIT is not guaranteed. In this dissertation, we provide a method to guarantee the timing equivalence of the generated target binary programs.

We use an interval-based approach, first invented by Allen and Cocke, to test timing equivalence of a translated target binary program with respect to a source binary program. We introduce the concept of timing sensitivity, the maximum timing difference between the source and target programs, to judge how closely the target program will mimic the source program's timing.

We say that a target program is executable with timing equivalence if the timing sensitivity can be reduced to zero, and with timing invariance if the timing sensitivity

can be bounded by a constant. We have discovered necessary and sufficient conditions to provide timing equivalence and invariance for target programs.

If a target program is executable with timing equivalence or invariance, it must be enforced by a synchronization scheme. When the target program is executable with timing invariance, the timing sensitivity depends on how and where the target program is synchronized. We use a local synchronization scheme that removes local timing error to provide timing invariance. We develop a static method to measure timing sensitivity with the local synchronization scheme. We also develop optimization techniques to minimize timing sensitivity.

In summary, the main research contributions of this dissertation are 1) necessary and sufficient conditions for timing equivalent and invariant translation 2) an algorithm to find timing sensitivity of the target program and 3) techniques to minimize timing sensitivity of the target program.

CURRICULUM VITA

NAME OF THE AUTHOR: Inkyu Kim

PLACE OF BIRTH: Kyungbuk, Korea

DATE OF BIRTH: February 26, 1960

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Oregon State University
Yeungnam University, Taegu Korea

DEGREES AWARDED:

Doctor of Philosophy in Computer and Information Science, 1997,
University of Oregon

Master of Science in Computer Science, 1990,
Oregon State University

Bachelor of Engineering in Electronic Engineering, 1985,
Yeungnam University

AREAS OF SPECIAL INTEREST:

Binary-to-Binary Translation
Real-time Systems
Optimizing Compiler
Program Debugging
Parallel Processing

PROFESSIONAL EXPERIENCE:

Research Assistant, Department of Computer and Information Science,
University of Oregon, Eugene, 1993-1997

Research Staff, Department of Computer Science and Engineering,
Oregon Graduate Institute of Science and Technology,
Portland, 1992-1993

Research Assistant, Department of Computer Science,
Oregon State University, Corvallis, 1988-1992

DEDICATION

This dissertation is dedicated in loving memory of my father

Heetae Kim

who did not wait for my long studies.

ACKNOWLEDGEMENTS

Finally done (or, Phinally Done(Ph.D)!). Many thanks goes to many people:

First to my advisor, Zary Segall, for his patience, encouragement, support and ideas. To my committee members, Arthur Farley, John Conery and Nancy Melone for their comments and encouragements. Without them this would not be possible.

To the staff in the CIS department, Betty Lockwood, my American mom, Shelley Carlson, Wanda Weber and Jan Saunders, who gave me lots of smiles and help. They provided some of the best support.

To the graduate students who made rough times tolerable, Ferenc Rákóczi, Gunnar Sacher, Gerd Kortuem, Odile Wolf, Takunari Miyazaki, Ted Kirkpatrick, Yong Xiao, Kevin Glass, Jens Mache and Bei Li. They are wonderful to work with.

A special thanks goes to Dr. Bryce Cogswell for many discussions and suggestions. He read the first version of this thesis thoroughly and corrected many mistakes.

Even more thanks goes to my family, my wife, Geeyeoun for her persevering love and support throughout the years, my two sons, Hyunwoo and Hyunjin for the joy they bring me, my sister, Hyunkyu for her delicate mind and my two brothers, Junkyu and Changbum for their brotherhood. They are the source of my energy.

Last but not least thanks goes to my parents, one in this world, Byoungok Choi and the other in the other world, Heetae Kim. They are the reason. It was tough when he did not wait for me to finish my long studies in this world. I know he is happy now but I wish I could see his face with a big smile.

This work was sponsored in part by ARPA, monitored by the Air Force, under Contract Number F33615-93-C-1311, and Martin Marietta under TTM748365.

The view, opinions, and/or findings contained in this dissertation are those of the Author, and shall not be construed as an official Advanced Research Projects Agency, the United State Air Force, or University of Oregon position, policy or decision, unless designated by other documentation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION AND RATIONALE

The goal of this thesis is to investigate timing issues in Binary-To-Binary Translation (BBT) of real-time applications. While BBT provides a means of smooth migration of legacy software to newer architectures, it does not handle real-time applications properly. In real-time applications, it is often the case that doing something too quickly is as unacceptable as doing it too slowly. Thus, the translation of real-time applications must preserve timing equivalence of all visible events as well as semantic equivalence of the source application programs. The TIBBIT [14, 13] (Timing Insensitive Binary to BInary Translation) project introduced the problem of providing timing equivalence in binary-to-binary translation of real-time applications. The TIBBIT system delays the execution of the target program at regular intervals assuming the target machine is faster than the source machine. However, the TIBBIT system does not guarantee timing equivalence of the generated target binary programs when some instructions take more time on the target machine. This thesis provides a method to guarantee the timing equivalence of the TIBBIT-generated target binary programs. Major questions this thesis answers include:

- How to guarantee timing equivalence of the TIBBIT-generated target binary programs.

- What are the necessary and sufficient conditions to guarantee timing equivalence of the generated target programs.
- How to convert a target binary program that is not timing equivalent into an equivalent one.
- How to minimize timing differences between source and target programs.

Binary-to-binary Translation

Binary-to-binary translation (BBT) is a method that takes a binary executable program (source binary) for a machine (source machine) and translates it into another executable program (target binary) to run on another machine (target machine), without referencing the original source program[51, 5]. The translated target binary program is a sequence of target machine instructions that reproduces the behavior of the source binary program. Figure 1 depicts the binary-to-binary translation scheme.

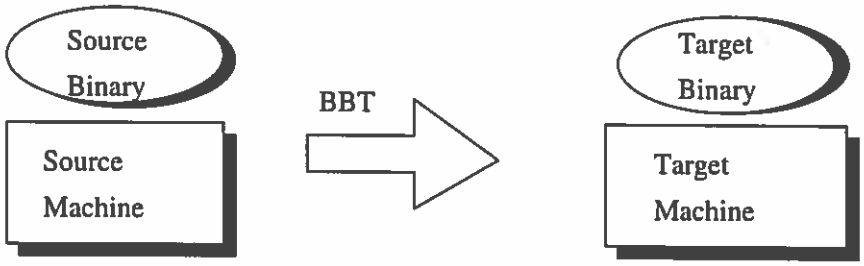


Figure 1: Binary-to-binary translation

The main advantage of this approach is that it is not necessary to have access to the original source program in order to perform the translation. A complicated application includes many software components developed using many different tools and compilers. One example of this is Microsoft Word, which is partly written in its

own macro language, with other portions written in C and assembly language. Porting this complicated software to a new architecture requires all tools and compilers available on the target machine before one starts. BBT provides a method to avoid this complication. The BBT technique has been used by many companies such as IBM [50], DEC [51], Tandem[5], and Apple.

The translation of a binary code involves two program translations. One is the application binary code translation and the other is operating system code translation as shown in Figure 2.

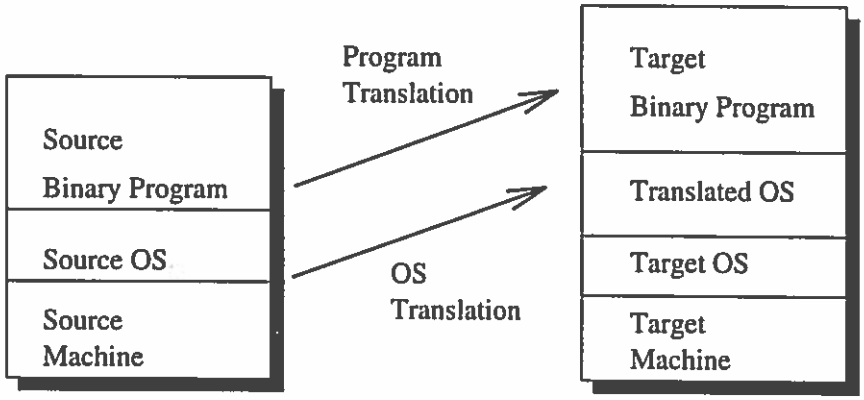


Figure 2: Components of binary-to-binary translation

Translation of operating system code involves many complicated issues as discussed in [50]. We concentrate on real-time embedded-systems where the operating system and application binary program are combined together.

The main disadvantage of the BBT approach is performance degradation. In general, for the given set of instructions on the source machine, finding an optimal set of instructions on the target machine that reproduces the behavior of the source instruction set is a difficult problem. Also, some conditional statements must be added to make sure the architectural differences between the two machines are handled

correctly. Detailed discussion on these issues can be found in [50].

Binary-to-binary Translation of Real-Time Applications

Previous Binary-to-binary Translation (BBT) approaches provide only semantic equivalence between source and translated target programs where the optimization goal is reducing the total execution time of the program on the target machine.

For real-time applications, however, the translator must preserve the timing equivalence as well as semantic equivalence of the source binary program. In this case, the goal of optimization is to reduce the timing difference between source and target programs. Figure 3 depicts the problem of preserving the timing equivalence of all visible events (i/o events).

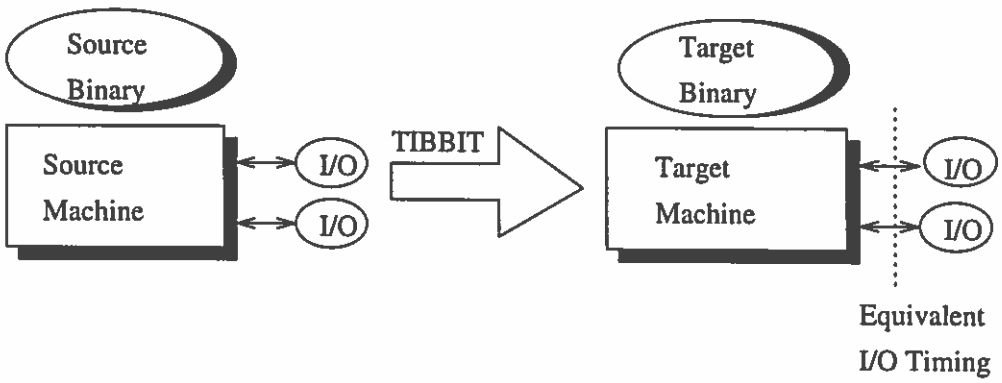


Figure 3: The problem of providing the timing equivalence between source and target programs

The TIBBIT project [14, 13] addressed this timing issue, i.e., the problem of preserving the timing equivalence between source and translated target programs. In TIBBIT, the time required for each basic block on the source machine is computed. While running on the target machine, the TIBBIT system compares source and target timing at regular intervals and adjusts the execution of the target program. The

translated target program executes faster than the source on the target machine, but is periodically delayed.

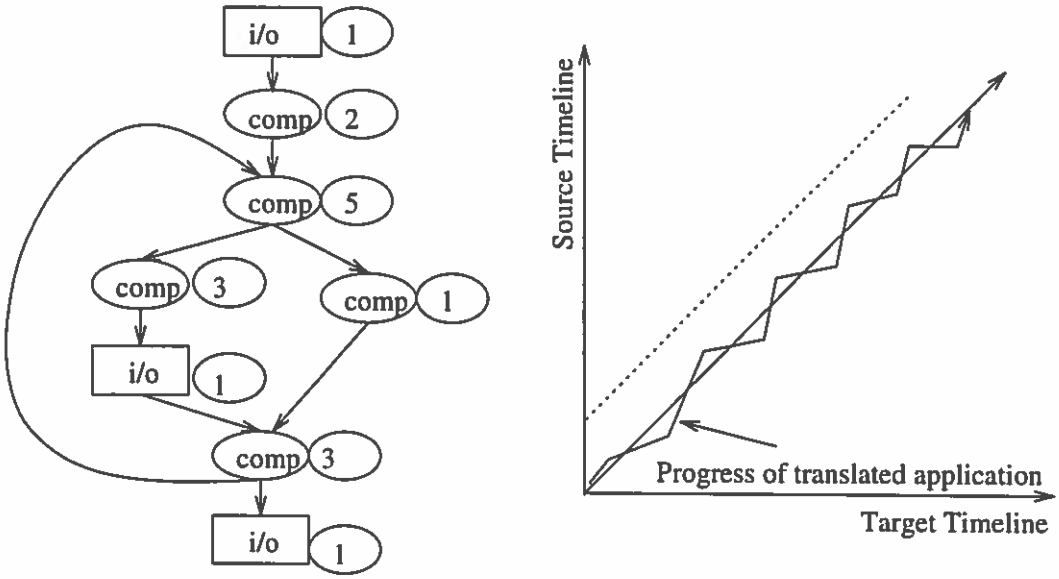


Figure 4: TIBBIT algorithm to provide timing equivalence

The example shown in Figure 4 depicts the algorithm used in TIBBIT. A node v in the control flow graph has an extra node that contains the execution time required on the source machine M_S, denoted by $(ExecTime(v, M_S))$. A limitation of TIBBIT is that it generates timing equivalent target binary programs only if all basic blocks take less time on the target machine.

Rationale

To provide timing equivalence in binary-to-binary translation, it is important to know the processing speed difference between the two machines, source and target. At minimum, the target machine must be faster than the source machine. However, the concept of processing speed difference of two machines is ambiguous. Modern

processors, such as RISC-based architectures, achieve their speedup over old processors through optimization of the most frequently used instructions while paying some penalty for less frequently used ones. On average, these newer processors are faster than older ones, but some instructions are actually slower than those of corresponding instruction sequences in the old processors.

Also, some translation overhead exists. As mentioned, it is a difficult problem to find the optimal corresponding set of target machine instructions for the given set of source machine instructions. In addition, there exists runtime overhead added to target binary programs. Some information that is available to higher-level translators is not available to a binary-to-binary translator. Thus, even when all corresponding instructions in the target machine take less time than the source machine instructions, some basic blocks in the translated target binary program may take longer on the target machine, simply because of translation overhead.

For these two reasons, the current TIBBIT system is not sufficient. When there exist some basic blocks that take a longer time on the target machine, the execution time difference between source and target program cannot be maintained by delaying the program execution (synchronization) on the target machine.

Consider the example given in Figure 5. We assume that the time required to execute a given basic block is computable for both source and target machines. We also assume that the required execution time for all i/o instructions remain the same on the target machine. Each node v in the control flow graph has an extra field containing the execution time required on both source and target machines for the node v, represented as $(ExecTime(v, M_S)/ExecTime(v, M_S))$. The target machine is faster than the source machine on average. However, one basic block

takes a longer time on the target machine. When executed on the target machine, the application may execute faster or slower depending on how often this slow basic block executed at the particular execution. When the application executes faster on the target machine, inserting synchronizations on the target program provide timing equivalence. However, when the application executes slower on the target machine, inserting synchronization does not provide timing equivalence.

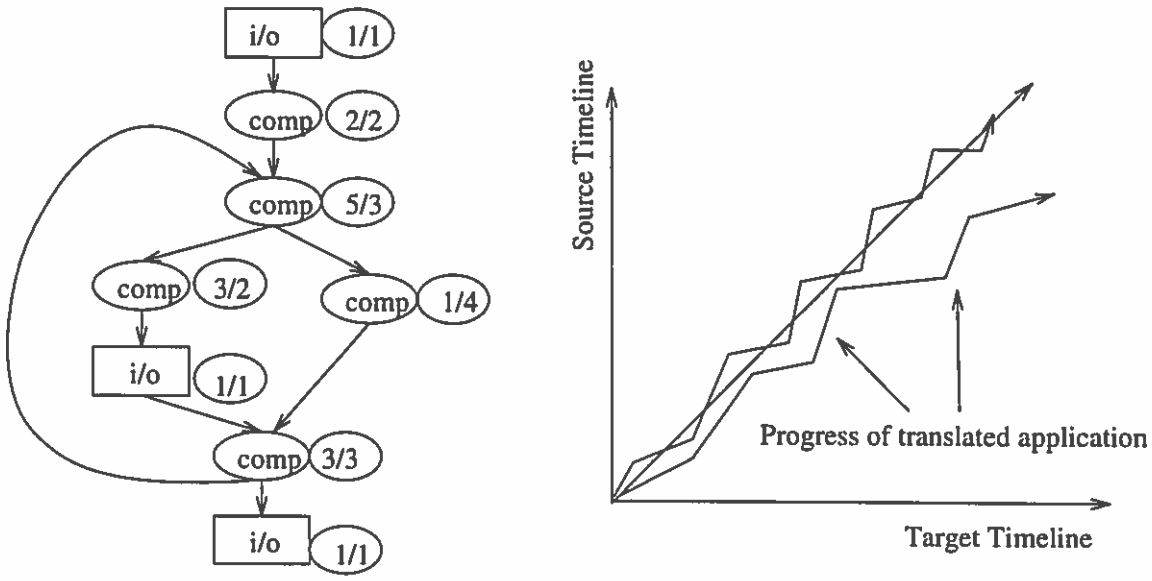


Figure 5: Timing equivalent translation with slower instructions

Considering these factors, the goal of this research is to develop a method that analyzes a program to determine if it is always possible to execute a program timing equivalence.

Here, we are to test if the maximum timing difference between source and target (the maximum drift) can be bounded by a constant. The maximum drift between source and target applications is shown in Figure 6. In the graph, the timing difference between source and target is bounded by a constant denoted by the distance between

two dotted lines.

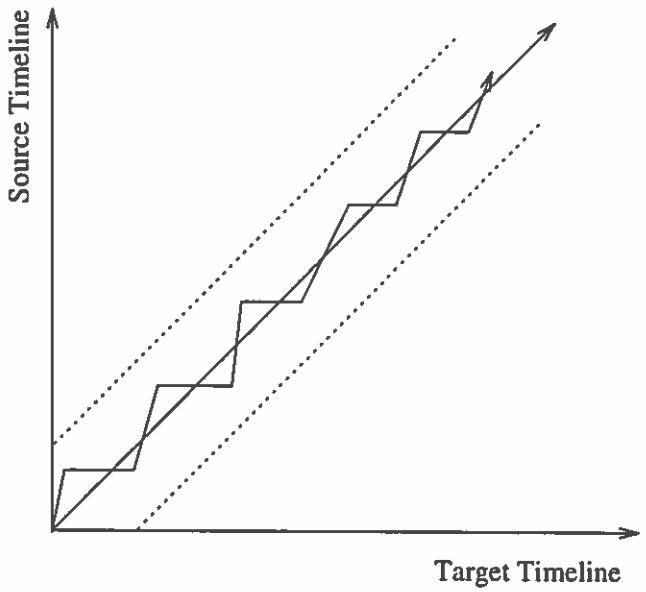


Figure 6: Timing difference between source and target applications limited by a constant

This dissertation answers the following questions.

- When is the maximum drift bounded by a constant?
- How to find the maximum drift (the constant) if it is bounded?
- How to ensure that bounds exist?
- How to reduce the maximum drift?

To answer these questions, we assume the following.

- The control flow graph of the translated target program is given with execution time required for each basic block on both source and target machines.
- Environments, such as networks and i/o devices, remain the same, i.e., only the processor and possibly part of the memory system are replaced.

Related Work

Binary-to-Binary Translation

Many commercial companies use binary-to-binary translation schemes to provide a speedy upgrade path to newer architectures. Most of these binary-to-binary translation schemes have been developed with a particular source and target platform in mind. The optimization goal of these binary-to-binary translators is to minimize total execution time on the target machine. Issues of providing timing equivalence for real-time applications are completely ignored.

DEC

Kronenberg [37] and Sites [51] at Digital Equipment Corporation (DEC) use binary-to-binary translation techniques to port VAX VMS, MIPS Ultrix, and 80x86-based programs to the Alpha architecture. They successfully ported a large number of applications to the new architecture in minimal time. Because the Alpha architecture was designed with binary translation of MIPS code in mind, the performance of binary translated programs are comparable to that of native compiler generated programs.

IBM

Silberman and Ebcioglu at International Business Machines (IBM) developed a binary-to-binary translator that supports migration of system code as well as self-modifying code [50]. Applications are represented in both "migrant" and "native" forms, and a hardware-assisted "migrant engine" is used to execute sequences of code when the native engine fails due to either untranslatable or self-modifying code.

Tandem Non-Stop Systems

Tandem Non-Stop Systems used a BBT system [5] to upgrade all of its vendor and user software from a proprietary stack-based CISC architecture to new R3000-based RISC machines. Both the operating system and applications were ported, and relied upon a combination of static translation and run-time interpretation.

HP

Bergh [7] at Hewlett Packard relies on a combination of object code translation and emulation to execute HP3000 software on the HP Precision architecture family, using a "compatibility mode" environment in the target operating system.

Hunter Systems

Hunter Systems developed a system called XDOS [25] for binary-translating DOS applications to UNIX environments. The system was intended to be used by developers as an aid to porting, and relied upon a combination of human- and machine-translation techniques.

Decompilation

Decompilation is a process that reads a binary executable program and translates it into an equivalent program in a high-level language. Decompilation techniques are useful in understanding the binary code during the maintenance process [8] and in verifying compiled binary programs for safety-critical systems [52, 46]. All of these systems ignore timing issues that occur in many real-time systems. One exception is the original version of the TIBBIT translator which takes an executable binary

program and generates *C* code that is both semantically and temporally equivalent to the binary code.

Breuer and Bowen

Breuer and Bowen [8] describe a method to decompile programs generated by a simple Occam-like compiler. They construct an abstract syntax tree from the executable binary code and generate an equivalent program in the logic programming language, *Prolog*.

Spector and Pavey

NASA used a decompilation technique [52] to verify the program called *System Management (SM)*, which is used in their space shuttle. The decompiler decompiles the memory images generated by the *SM* preprocessor and compare the results with the original inputs.

Pavey and Winsborrow [46] also used a decompilation technique to compare the source code and PROM contents of a safety-critical system used in the UK nuclear industry.

Gough and Cifuentes

Gough and Cifuentes [11, 10] describe issues in decompiling 80x86 binary programs. Many optimization techniques that have been used in optimizing compilers are used in their decompiler. Their goal is similar to that of binary-to-binary translation but it also provides portability.

Cogswell and Segall

A version of the TIBBIT translator [13] takes an executable binary program and generates C code to provide portability. The generated C code can be compiled for any target machine. This version of the TIBBIT translator computes execution times on the source machine for each basic block. For each basic block, the translator inserts C code that accumulates the execution time of basic blocks in a global counter that accumulates these execution times. The TIBBIT-generated target C program is compiled by a C compiler on a specific machine. The TIBBIT system interrupts the execution of the target binary program at regular intervals and delays the execution if it runs too fast.

Timing Analysis

For a real-time program, we must be able to predict the computation time of the program on the target machine. There are a number of techniques that predict the execution times of programs written in both low-level and high-level languages.

In our analysis, it is assumed that the execution time required for any basic block on both source and target machines is computable. In Chapter IX, we discuss how to deal with instructions that have non-constant execution time.

High-level Languages

Shaw [49, 45] describes a method to predict the execution time of high-level language statements. The method takes a program written in a high-level language (C), and bounds for each loop, and predicts the upper and lower bounds on execution time of each source level construct. The method decomposes programs into basic blocks

and predicts the implementation of each basic block. The execution time of each basic block is determined for a specific target machine. The accuracy of the predicted execution time is dependent on how accurately it can predict the implementation.

Low-level languages

Most older processors have a constant execution time for each instruction. If we know the bounds of each loop, it is not difficult to compute the execution time of programs written in low-level languages such as assembly language. However, the execution time of instructions on a newer processor with pipelines and cache is not constant. Issues of computing tight worst-case execution time of instructions for these machines are discussed in [40, 19, 60].

Zhang [60] presents a method to find worst-case execution times of instructions on pipelined processors.

Caches are extensively used in most recent computer systems to improve performance of the system. While caches improve performance of the system on average, they impose significant difficulties on timing analysis. Min [40] describes an analysis technique that accurately predicts the worst-case execution times of programs in the presence of caches.

Most timing analysis techniques are machine dependent. Since machine architectures evolve rapidly, developing a retargetable timing analysis method is important. Harmon [19] describes a portable timing analysis technique called *micro-analysis*. It predicts best and worst-case bounds for point-to-point execution times, based on a pattern matching scheme that uses a machine description and a set of timing rules. This scheme is capable of taking into account the architectural characteristics of the target processors and their effect on instruction execution time.

Translation of Real-Time Programs

Translation of real-time applications must preserve both functional and temporal requirements specified in the source programs. Temporal requirements are specified either implicitly or explicitly depending on the language used in the source program. Some source programs are written in higher-level languages with real-time constructs in which case all temporal requirements are expressed explicitly. Other are written in assembly language in which cases all temporal requirements are expressed implicitly.

RT-ASLAN

RT-ASLAN [6] is a real-time programming language which allows programmers to express timing constraints explicitly in a program. The kind of systems specifiable in RT-ASLAN are loosely coupled systems communicating through formal interfaces. From RT-ASLAN specification, performance correctness conjectures are generated. These conjectures are logic statements whose proof guarantees the specification meets critical time bounds.

Real-Time Euclid

Real-Time Euclid [33] does not allow some general programming constructs, including `while()`, recursion, and recursive data structures. The schedulability analyzer of Real-Time Euclid computes the worst-case execution time of a task by assuming the execution time of each instruction is constant.

Modechart

Mok [27] describes a real-time specification language called Modechart. The semantics of Modechart is given in terms of RTL (Real Time Logic) [26] that is amenable to reasoning about the timing of events. The translation of a Modechart specification into RTL formulas results in a hierarchical organization of the resulting RTL assertions. This hierarchical organization allows filtering of assertions that concern lower levels of the abstraction.

Flex

In many hard real-time systems, obtaining an approximate result before the deadline is more desirable than obtaining an exact result after the deadline. Flex [31] is a real-time programming language that allow computations to return imprecise results. This provides the flexibility needed to guarantee all important events meet their deadlines under all circumstances.

TCEL

In most real-time programming languages, timing constraints can be specified on blocks of code. In most real-time applications, however, these timing constraints are imposed on observable i/o events. TCEL [17] allows a programmer to express timing constraints between i/o events. In TCEL, unstructured constructs such as `goto` are not allowed. The compiler takes programs written in this high-level language and generates binary executable target programs. The compiler tests if the real-time scheduler can schedule the program so that the timing constraints expressed in the program can be guaranteed. If not, the compiler decomposes the program

into *reference blocks* and *constraint blocks*. By moving code in reference blocks, the compiler improves schedulability of the program. If the program is not schedulable with a single processor, the compiler schedules it with multiple processors. If worst case timing of the control structure in the constraint blocks does not meets its timing requirements, it is not schedulable.

Since the compiler takes a program written in a high-level language and does not allow unstructured constructs, the control flow of the program is known and all control flow in constraint blocks is well structured.

Since we take binary executable programs as input, control flow of the programs, which may be unstructured must be determined. We use interval analysis to find the control flows in the program. The timing requirements also have to be found by analyzing the control flow of the program. The model we use here is more general than that of TCEL in the sense that timing requirements do not have to be expressed explicitly between constraint blocks.

TIBBIT

The TIBBIT system is a binary-to-binary translator for real-time systems. In binary source codes, timing requirements are implicitly expressed. Thus, the TIBBIT system must analyze the timing of all instructions in the binary source code and generate a target binary code which mimics its timing behavior. It assumes that execution time for an instruction on the source machine is constant.

Program Analysis

Program analysis is to facilitate optimization of programs where the meaning of optimization is subjective. To optimize programs, a compiler must perform some sort

of program transformation based on the information obtained through the analysis of source programs. Typical two analyses a compiler performs are control flow and data-flow analysis.

Control Flow Analysis

Control flow analysis [2, 1] is used to find the control structure of a program so that the information can be used for optimizing transformations and other analysis. In control flow analysis, the most important goal is finding loops. One way to find loops is to find strongly connected regions (SCR) as discussed in [53]. While this methods finds all cycles in a control flow graph, it does not reveal hierarchical structures of the program.

Another way to analyze the structure of a control flow graph is interval analysis, which finds hierarchical structures in the control flow graph. Interval analysis, as formulated by Allen and Cocke, provides a way to solve data flow equations more efficiently. There are a number of quite different proposals in defining intervals [3, 2, 20, 18, 4, 48]. In general, an interval is a special form of loops. The specific definition of interval we use is the one defined in [18].

Data-flow Analysis

Data-flow analysis [43, 30, 20, 21, 23] is used to find where variables (data) are defined and used. Data-flow information is required for many optimization techniques. We use similar analysis methods to find how the timing differences between source and target machines flow over the control flow graph.

Comparison to Previous Work

All of the binary-to-binary translators and decompilers discussed above ignore timing issues required in translation of real-time applications. Most translators for real-time programs deal with programs written in a high-level language, where timing constraints are explicitly expressed. We are dealing with binary programs where timing constraints are expressed implicitly.

The current TIBBIT [13] system provides some degree of timing equivalence in binary-to-binary translation of real-time applications. The condition required to make the TIBBIT approach work is that every basic block takes less time on the target machine. This condition is stronger than the conditions presented here for timing equivalent translation. Even with stronger conditions, the TIBBIT system does not provide timing equivalence since it uses a dynamic synchronization scheme, e.g., synchronize every 10 ms. We present necessary and sufficient conditions for timing equivalent and invariant translation even in cases where some basic blocks in target binary are slower than corresponding block in the source binary program. When the timing equivalent translation is not possible, we provide the worst case timing error.

TIBBIT Project Overview

This research has been conducted as part of the TIBBIT project. An overview of the TIBBIT project is depicted in Figure 7.

ASTRA is a program that takes the description of source and target OS's and machines and generates a TIBBIT translator. This thesis describes partially the *Timing Validation and Feedback* part of the project. New algorithms are added to test

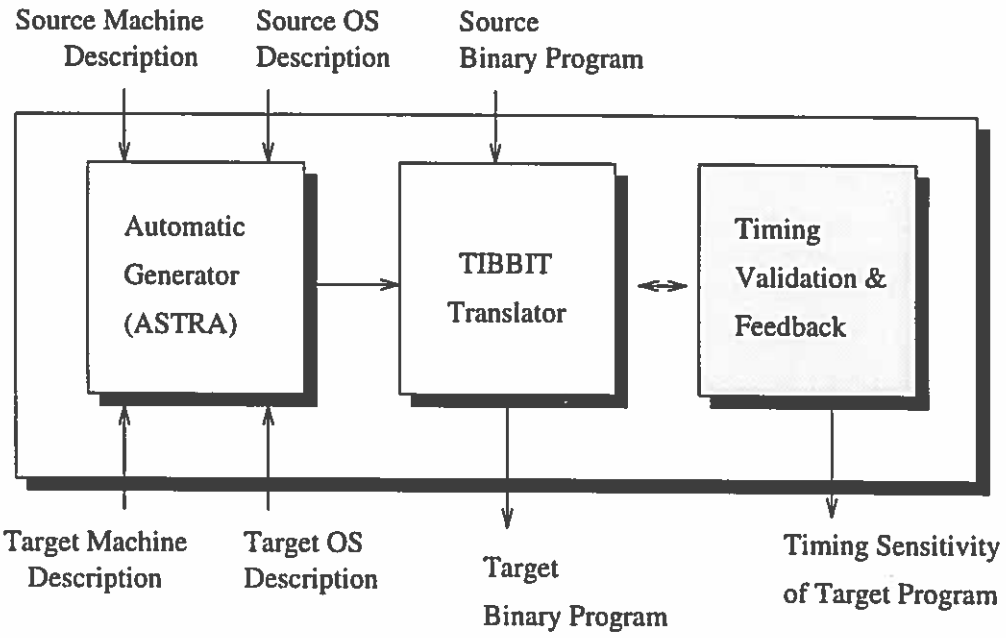


Figure 7: The overall structure of TIBBIT

if the generated target binary program is timing equivalent with respect to the source binary program. It also provides methods to reduce the maximum drift between the source and target programs.

Summary of Contributions

The goal of this thesis is to investigate timing issues in binary-to-binary translation of real-time applications. For the translation of real-time applications, preserving both semantic and timing equivalence are important, where others require preservation of semantic equivalence only. With the given source and translated target binary applications, the necessary and sufficient conditions that can guarantee timing equivalence have been found. Also, a number of optimization techniques which reduce the timing difference between the source and target applications are applied. The major

contributions of this dissertation include the following:

- A framework to analyze timing equivalence between source and target programs.
- Necessary and sufficient conditions for timing equivalent and timing invariant translations.
- Relevant timing sensitivities for "real-world" usage.
- A method to find timing sensitivities of the given target program.
- Optimization techniques for timing equivalence.

Thesis Organization

The rest of this dissertation is organized as follows. In Chapter 2, we define real-time system terminology and provide necessary background on real-time systems. In Chapter 3 we define basic terminology for compiler-related concepts to be used throughout this dissertation. In Chapter 4 we define timing equivalence between source and target systems. In Chapter 5 we present an analysis algorithm for timing equivalence testing. In Chapter 6 we present two synchronization schemes which enforce timing equivalence. In Chapter 7 we present algorithms that find timing sensitivities for the given source and target programs. In Chapter 8 we present algorithms that minimize the timing difference between the source and target programs. In Chapter 9 we discuss a number of implementation issues. Finally, in Chapter 10 we summarize our result and provide future research directions.

CHAPTER II

BACKGROUND

This chapter presents necessary background on real-time systems. We first define terminology related to time and events. We then present issues on real-time languages, programs and translators of them.

Timelines and Events

A *timeline* is a progression of *time* from the past to the future. An *event*, E, is an occurrence at a point in time, i.e., a happening at a cut of the timeline, which itself does not take any time. These terms are borrowed from Koepet and Ochsenreiter [36]. The *time value* of an event(E), denoted by $TV(E)$, is the value of the time at the event E. The *time duration* of two events E_i and E_{i+1}, denoted by $TD(E_i, E_{i+1})$, is the time interval between these two events, i.e., the section of the timeline between the two events. $TD(E_i, E_{i+1})$ is defined as Equation II.1. Figure 8 depicts timeline, events and time duration between two events.

$$TD(E_i, E_{i+1}) = \mid TV(E_{i+1}) - TV(E_i) \mid \tag{II.1}$$

Real-Time Systems

An *application* is a program running on a specific machine. A *real-time application* is an application that interacts with the external world in a way that involves

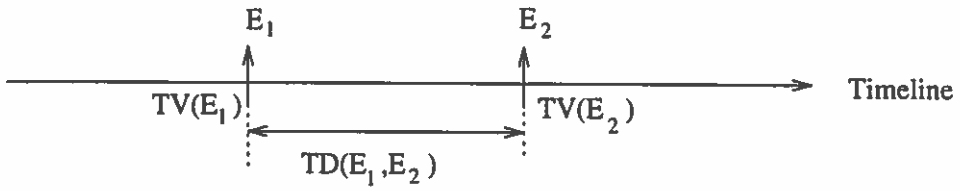


Figure 8: Timeline, events and time duration between two events.

time-related conditions. A *real-time system* is a more general term to refer to a combination of software and hardware that deals with outside events that have time constraints. The main characteristic of real-time programs is the existence of *temporal (or timing) requirements* in addition to *functional requirements.* Temporal requirements specify timing constraints for sequences of events, while functional requirements specify required transformations of inputs to produce outputs of the system. Typical real-time systems are control systems (manufacturing systems, robotics), monitoring systems (patient monitoring, air traffic), and communication systems. The potentially high cost associated with incorrect operations of these systems has created a demand for rigorous testing and implementation for both functional and temporal requirements.

I/O Events

An application generates *i/o events* to communicate with outside systems, i.e., receives inputs and generates outputs. All observable events from outside the system are i/o events. Most systems generate these i/o events by reading and writing data from/to one or more of the system i/o ports. Since i/o events are generated by a computer, every i/o event takes some time to complete. Thus, every i/o event *io* has two events associated with it, *start*(io_s) and *finish*(io_f). The *time duration* of an i/o

event io, denoted by $TD(io_s, io_f)$ or $TD(io)$ for short, is the time interval between io_s and io_f. An i/o event is represented by its *start* event. Thus, the time duration of two i/o events, io_i and io_{i+1}, is the time interval between *start* events of io_i and io_{i+1}. An application P has two special events, *Program Start*(P_S) and *Program Finish*(P_F) events, which are not i/o events but we treat as i/o events. Both P_S and P_F events take no time to complete. Figure 9 shows two i/o events io_i and io_{i+1} with program start and finish events.

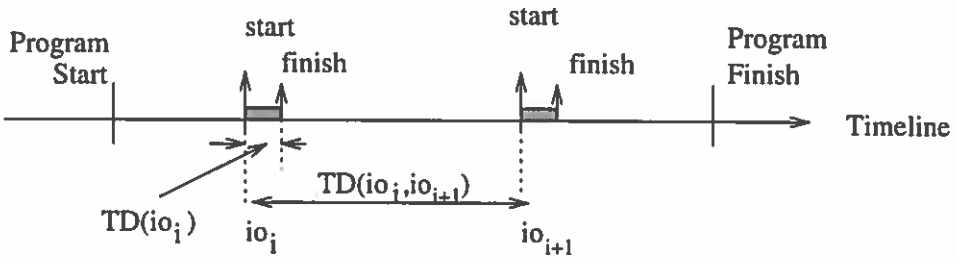


Figure 9: Timeline, i/o events and duration between two i/o events.

Suppose an application PS is running on a machine M_S. The *application time value (ATV)* of an i/o event(PSE) in PS, denoted by $ATV(PSE, M_S)$, is a function that returns the time duration from the point of the *Program Start*(PS_S) to PSE. The $ATV(PS_S, M_S)$ is zero and $ATV(PS_F, M_S)$ is equal to the total execution time of PS on M_S. The $ATV(PS_F, M_S)$ is ∞ if the application never finishes. Now, suppose the application PS is translated into PT so that it can be executed on a machine M_T. The $ATV(PT_S, M_T)$ is zero and $ATV(PT_F, M_T)$ is equal to the total execution time of PT on M_T. Here, $ATV(PS_S, M_S) = ATV(PT_S, M_T) = 0$ but $ATV(PS_F, M_S)$ is not necessary equal to $ATV(PT_F, M_T)$.

Machine Models

Suppose a real-time application (source application) running on a specific machine (source machine) is translated into another application (target application) so that it can run on another machine (target machine) as shown in Figure 10.

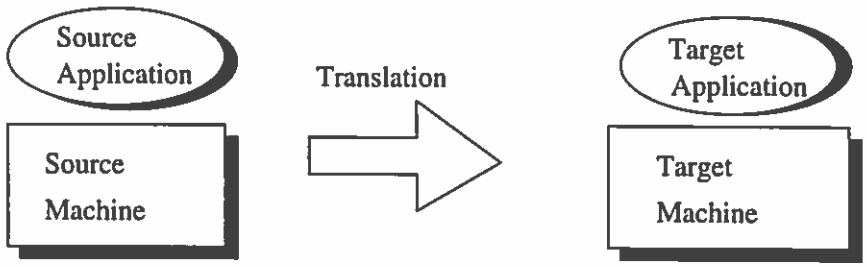


Figure 10: A source application for source machine and a translated target application for target machine.

To provide timing equivalence on the translated target application, the target machine must be at least as fast as the source machine. However, it is not so clear what we mean by "a machine is faster than the other." We thus define the meaning of faster and slower machine more clearly.

Let the sets I_T and I_S be instruction sets of machines M_T and M_S, respectively. Let $OCIS$ be an ordered combination of instructions, where every element $ocis \in OCIS$ is in I_S. Also, let $OCIT$ be an ordered combination of instructions, that is functionally equivalent to $OCIS$, where every element $ocit \in OCIT$ is in I_T. The execution time required for an ordered combination of instructions OCI on machine M is written as $ExecTime(OCI, M)$. We say machine M_T is *definitely faster* than machine M_S if for all $OCIS$, there exists a functionally equivalent set of instructions

$OCIT$ such that $OCIT$ takes less time than $OCIS$, i.e.,

$$(\forall OCIS)[\exists OCIT \ni ExecTime(OCIT, M_T) \leq ExecTime(OCIS, M_S)]. \quad \text{(II.2)}$$

Machine M_T is said to be *average faster* than machine M_S if for any large n,

$$\frac{\sum_{i=1}^{n} ExecTime(OCIT_i, M_T)}{n} \leq \frac{\sum_{i=1}^{n} ExecTime(OCIS_i, M_S)}{n}. \quad \text{(II.3)}$$

Synchronization

Suppose source and equivalent target applications are running on source and target machines, respectively. The goal of *synchronization* is the elimination of timing differences between the two systems. Synchronization of the target application with respect to the source application can be always achieved by delaying the execution of the target application when the target machine is *definitely faster* than the source machine.

While synchronization can be performed at any time/place in the target application, the timing of i/o events is significant for real-time applications. The reason is that only i/o events are observable from the outside. Thus, we are only concerned with timing differences between corresponding i/o events on source and target applications. A target application that is semantically equivalent to the source application said to be *timing equivalent* if all i/o events in the target application are synchronized with respect to the corresponding i/o events in the source application.

Figure 11 shows such an example. An i/o event is represented as $\uparrow$. There are two i/o events, SE_1 and SE_2, in the source application. The application time value of these events on the source machine are $ATV(SE_1, M_S) = 6$ and $ATV(SE_2, M_S) =$

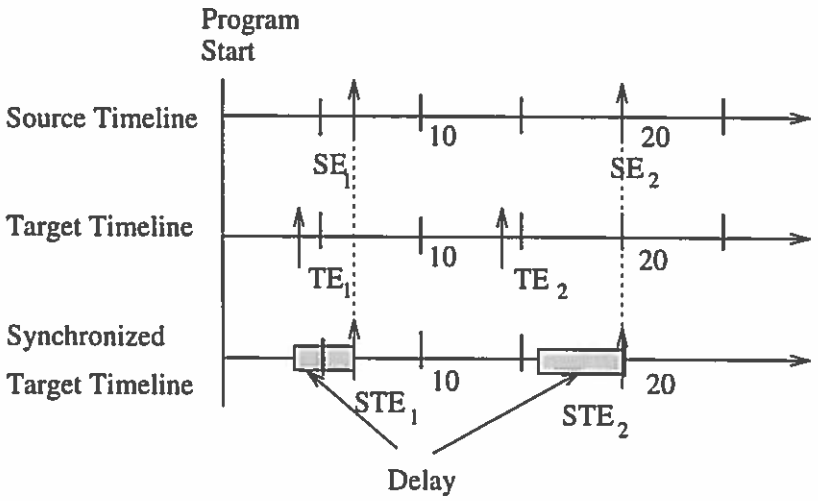


Figure 11: Synchronization of corresponding i/o events

20. The application time value of corresponding events, TE_1 and TE_2, on the target application are $ATV(TE_1, M_T) = 4$ and $ATV(TE_2, M_T) = 14$. Through synchronization, these two applications are timing equivalent, i.e., $ATV(STE_1, M_T) = 6$ and $ATV(STE_2, M_T) = 20$.

The same synchronization also can be viewed as shown in Figure 12. Synchronization can be performed either statically or dynamically. Dynamic synchronization methods decide synchronization points at run-time. The TIBBIT system [14] uses a dynamic synchronization method. In TIBBIT, the target application is synchronized at regular time intervals, every 10ms for example. Static synchronization schemes decide synchronization points statically over the control flow graph so that synchronization is performed every time the particular point in the control flow graph is executed. One simple static synchronization method is synchronization of all i/o nodes.

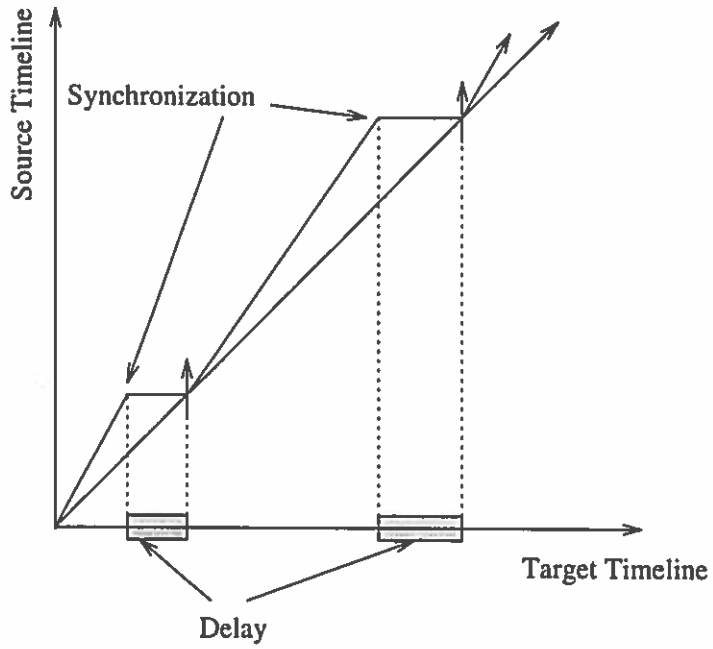


Figure 12: Synchronization of source and target programs

Real-Time Languages and Programs

In real-time programs, temporal requirements must be expressed in addition to functional requirements. These temporal requirements can be expressed either explicitly or implicitly in the program. Most older real-time programs are written in assembly languages because of lack of suitable real-time programming languages and compilers. Timing requirements in this case are expressed implicitly in the program. Programmers must embed timing requirements in the program implicitly by inserting some delay code. More recently, real-time languages are proposed and developed to allow programmers to express timing constraints explicitly in the program by providing timing constructs in the programming language. Language constructs to express timing requirements are discussed in [15, 33, 6].

Figure 13 shows an example of a real-time program written in a high-level

real-time programming language. This example has three i/o nodes, i.e., one input (`receive()`) and two outputs (`send()`). In this program, the timing requirements are explicitly specified in the program with special timing constructs, `every (stime) do`. In this specification, timing constraints are imposed on blocks of code.

```
every 10ms do
   receive(Sensor, &data)
   cmd1 = nextCmd(state, data, s1);
   cmd2 = nextCmd(state, data, s2);
   state = nextState(state,data);
   send(Actuator1, cmd1);
   send(Actuator2, cmd2);
enddo
```

Figure 13: A simple real-time program

In most real-time applications, however, these timing constraints are imposed on observable i/o events. More advanced real-time programming languages called *event-based real-time programming languages* are proposed to allow the programmer to specify timing constraints between observable events. One such real-time programming language is *Time-Constrained Event Language* (TCEL)[17]. Figure 14 shows the same example of real-time program written in $TCEL$ (also see related work section in Chapter I). In this program, the timing constraint are specified between i/o events.

Once a program is written, it is the compiler's duty to translate the real-time program into target machine code which preserves both temporal and functional requirements specified in the source program.

```
every 10.0 ms do
   receive(Sensor, &data)
   start after 1.0 ms finish within 5.0 ms
      begin
         cmd1 = nextCmd(state, data, s1);
         cmd2 = nextCmd(state, data, s2);
         state = nextState(state,data);
         send(Actuator1, cmd1);
         send(Actuator2, cmd2);
      end
enddo
```

Figure 14: A simple real-time program with event-based timing constructs

Program Translation

In general, program translation can be viewed as a process that converts a program executable on one machine model to an equivalent program that is executable on another machine model. Consider a program written in a higher-level programming language. This program can be viewed as a program that is executable on the conceptual machine, which is described in the language specification. A translator (or compiler) converts this program into another program that is executable on a specific target machine. A translator takes source program and source and target machine descriptions and generates a target program which is an executable on the target machine. Figure 15 depicts such a translator.

A program translation can be formulated as a function as shown in Equation II.4, where TP is target program, SP is source program, SM is source machine, TM is target machine and $\mathcal{F}$ is a translation function or translator.

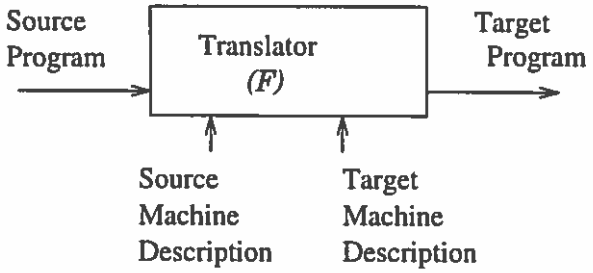


Figure 15: Translator

$$\mathtt{TP} = \mathcal{F}(\mathtt{SP}, \mathtt{SM}, \mathtt{TM}). \tag{II.4}$$

The characteristics of the translator depend on a number of categories: equivalence between source and target programs, "optimality" of the generated target program, and input and output languages.

Equivalence

While the equivalence test of two programs in general is an undecidable problem, two programs are said to be "equivalent" if all transformations performed by the translator are sound and complete within the boundary of "equivalence." Every translator must provide some degree of equivalence between source and target programs. We consider two different categories for the equivalence: functional (semantic) and temporal (timing) equivalence. For most systems, only semantic equivalence is required.

Definition 2.1

A translator $\mathcal{F}$ preserves *semantic equivalence*, denoted by $\mathtt{TP} \overset{SE}{=} \mathtt{SP}$, if SP and TP are *semantically equivalent.* □

In this case, all functional requirements specified in the source program (SP) are preserved in the target program (TP). All compilers or translators must preserve the semantic equivalence. For real-time applications, however, preserving timing equivalence is also important.

Definition 2.2

A translator $\mathcal{F}$ preserves timing equivalence, denoted by TP $\stackrel{TE}{=}$ SP, if SP and TP are semantically equivalent and for all corresponding i/o events io_s and io_t, $ATV(io_t, M_T) = ATV(io_s, M_S)$. □

In this case, both functional and temporal requirements specified in the source program (SP) are preserved in the target program (TP).

Optimality

The goal of optimization may vary depending upon system requirements. In most cases, the goal of optimization is to reduce the total execution time using fewer resources (performance optimization). Performance optimization has a long history [1, 3, 39]. However, the major goal of optimization for real-time systems is to meet the timing requirements specified in the source program, i.e., providing timing equivalence between two programs, source and target. One such example is *TCEL* [17, 16, 24]. The *TCEL compiler* optimizes target programs to comply with the timing constraints specified in programs.

Input and Output Languages

Both input and output can be anything from binary object code to programs written in high-level languages such as C or Prolog. In *TCEL*, like most other com-

pilers, input programs are written in a higher-level programming language and the output is binary executable code. *Decompilers* [8] takes binary executable codes as input and generates high-level codes such as C or Prolog. *A binary-to-binary translator* [51] takes binary executable code for a machine and generates another binary code which is executable on another machine.

CHAPTER III

THEORETICAL TERMINOLOGY AND NOTATION

Most compilers use a form of control flow graph, an intermediate representation of programs, to analyze and optimize programs. This chapter explains our control flow graph used in timing analysis. We first provide some mathematical notations to define control flow graphs. We then define the control flow graph and spanning and dominator trees for it. A table providing a summary of notations is given at the end of this chapter.

Set

A *set* S is a collection of distinct objects. An element s is a member of set S, written $s \in S$, if s is an object that is in the set S. The cardinality of a set $|S|$ is the number of elements in the set S. A set is *empty*, written $\emptyset$, if $|S| = 0$. A set is an *infinite* set if $|S| = \infty$. A set S is a *subset* of T, $S \subseteq T$, if every element of S is also an element in T. A set S can be *partitioned* into k *nonempty* disjoint subsets whose union is equal to S.

Directed Graph, Paths and Trees

A *directed graph* $G = \langle V, E \rangle$ consists of a set of *nodes* V and a set of *edges* E, where each *edge* $e \in E$ is an ordered pair of nodes, written $\langle v_1, v_2 \rangle$, and $v_1, v_2 \in V$. An edge e also can be denoted as $v_1 \rightarrow v_2$. If the edge $v_1 \rightarrow v_2 \in E$, the node v_1

is called *tail* and v_2 is called *head* of the edge. If $v_1 \rightarrow v_2$ is an edge of G then v_1 is the *predecessor* of v_2 and v_2 is the *successor* of v_1. The set $SUCC(v)$ is the set of all successors of v and the set $PRED(v)$ is the set of all predecessors of v. The *in-degree* of a node v is the number of edges of the form $u \rightarrow v$, and the *out-degree* of v is the number of edges of the form $v \rightarrow w$. A node is a *source* if its in-degree is zero and a *sink* if its out-degree is zero. A graph H is a sub-graph of G if $V_H \subseteq V_G$ and $E_H \subseteq E_G$ such that an edge in E_H has both the tail and head nodes in V_H.

A *path* p of length k is a sequence of nodes $\langle a_0, \ldots, a_k \rangle$ such that there is an edge $\langle a_i, a_{i+1} \rangle$ for all i = 0,..., k-1. We say a node v *supports* a path p if v appears in p. A path $\langle a_0, \ldots, a_k \rangle$ can be divided into smaller paths called sub-paths. A particular sub-path $\langle a_0, \ldots, a_{k-1} \rangle$ of p is denoted by p°. We say a sub-path $\langle a_i, \ldots, a_j \rangle$ of $\langle a_0, \ldots, a_k \rangle$ *supports* the path $\langle a_0, \ldots, a_k \rangle$. A *trivial path* is a path of length zero, i.e., a single node. A path is *simple* if all nodes in the path are distinct. A non-trivial path p from x to y is a *cycle* if $x = y$ and is a *simple cycle* if it is a *cycle* and p° is *simple*. An edge $\langle v_1, v_2 \rangle$ is called a *self looping edge* if $v_1 = v_2$. For a given two nodes $x, y \in G$, there may exist many distinct paths which are denoted by the set $\rho(x, y)$. The set $\rho(x, y)$ can be *infinite* if there exist *cycles* between two nodes. The set of all simple paths from x to y is denoted by $\sigma(x, y)$.

A *directed acyclic graph (DAG)* is a directed graph with no cycles, i.e., any path between two nodes in a *DAG* is *simple*. A directed graph is *cyclic* if it is not a *DAG*. A *tree* of a directed graph is a *DAG* with the following three properties: There is a single source node, called *root* which has zero in-degree; every node in the *tree* except the root has in-degree of one; and for every node in the *tree*, there exists a simple path from the node *root*. If there is a path from u to v in the *tree* i.e., $u \stackrel{*}{\rightarrow} v$, then

u is an *ancestor* of v and v is *descendant* of u. The node u is *proper ancestor* of v and v is a *proper descendant* of u if $u \neq v$, i.e., $u \xrightarrow{+} v$. The node u is an *immediate ancestor* of v and v is an *immediate descendant* of u if the length of the path is one, i.e., $u \rightarrow v$.

Control Flow Graphs

Our *control flow graph* is a directed graph G with a set of *nodes* and *edges*. A node represents a basic block, where a basic block can be either an *i/o basic block* (denoted by a rectangle) or *computation basic block* (denoted by an oval). A computation basic block contains a sequence of instructions in which the flow of control enters at the beginning and leaves at the end, without halt or possibility of branching except at the end. If any instruction in a basic block is executed, all instructions in that basic block will be executed. An i/o basic block contains an i/o function call but nothing else.

Assume M_S is the source machine and M_T is the target machine. For each node $v \in V$, the execution time on source and target for v, denoted by $EST(v)$, is an ordered set $(ExecTime(v, M_S), ExecTime(v, M_T))$. This ordered set is denoted as $(ExecTime(v, M_S)/ExecTime(v, M_T))$ in the graph. We assume that our control flow graph is obtained from the translated target program, where it may contain some nodes that are not in the control flow graph obtained from the source code. If $v \in V$ is not in the control flow graph obtained from the source code, $ExecTime(v, M_S) = 0$. The required execution time difference between source and target machine for a node $v \in V$, $STD(v)$, is defined as $ExecTime(v, M_T) - ExecTime(v, M_S)$.

An edge represents potential flow of control between basic blocks. Conditional branches are represented by nodes with two successors. A control flow graph G can

be represented as a triple $G = \langle V, E, Entry \rangle$, where

- V is a set of nodes, where each node can be either an i/o node or a computation node;
- E is a set of edges which is subset of $V \times V$;
- The $Entry$ is a node in V with zero in-degree (source).
- $(\forall v \in V)[Entry \xrightarrow{*} v]$.

A node $v \in V$ with zero out-degree is called an *exit* node. There may exist multiple *exit* nodes in G. The set of all i/o nodes in G is denoted by $IO(G)$.

A *complete path*(*cp*) of G is a path from the $Entry$ to an *exit* node. The set of all complete paths of G is denoted by $CP(G)$. A *cp* is a *complete simple path*(*csp*) if it is *simple*. The set of all complete simple paths in the graph G is denoted by $CSP(G)$. A $cp \in CP(G)$ that is not a member of $CSP(G)$ contains a $csp \in CSP(G)$. The set $CP(G)$ is roughly equivalent to all execution instances of the program represented by the graph G. A node $v \in V$ may support a $cp \in CP(G)$ many times if it is in cycles in G. The set $INST(v, cp) = \{iv_1, \ldots, iv_n\}$ represents instances of v on *cp*, where iv_i is an execution instance of v in *cp*. We denote a complete path *cp* over a specific control flow graph G as $\{iEntry_1, \ldots, iExit_1\}$, where $iEntry_1$ abd $iExit_1$ are the only instances of $Entry$ and $Exit$ nodes, respectively.

A complete path $cp \in CP(G)$ is denoted by *cps* if it is running on the source machine. The set of all complete paths on the source machine is denoted by $CPS(G)$. A complete path $cp \in CP(G)$ is denoted by *cpt* if it is running on the target machine, while set of all complete paths on the target machine is denoted by $CPT(G)$. An instance cpv_i on *cp* has corresponding instances on both *cps* and *cpt*, which are $cpsv_i$

on *cps* and $cptv_i$ on *cpt*. Here, *cp* is a conceptual path defined in the graph G while *cpt* is the execution of *cp* on target machine and *cps* is the execution of *cp* on source machine. Also, the path *cpt* can be viewed as a translated target path from *cps* by a translator.

The set $SC(G)$ is the set of all *simple cycles*(*sc*) in G. A path is *repeatable* if it is in a *cycle* and *non-repeatable* if it not in a cycle. The number of executions for a repeatable path is not generally known statically while the number of executions for a non-repeatable path is at most one. A repeatable path is simple if it is in a simple cycle. Any repeatable path can be sub-divided into simple repeatable paths. Suppose a simple cycle *sc* is given. The *maximum simple repeatable path* of *sc*, denoted by *msrp*(*sc*), is sc°.

The set of all maximum simple repeatable paths in G is denoted by $MSRP(G)$. Any $cp \in CP(G)$ is decomposable with a $csp \in CSP(G)$ and multiple instances of $msrp \in MSRP(G)$. In a *cp*, nodes that support a *csp* or *msrp* may not be in consecutive order.

Example 3.1

Figure 16 shows a control flow graph G. In this graph, the Entry node is labeled with a. Each node $v \in V$ has another oval which is labeled with $EST(v)$. The node g is the only *exit* node in G and the set $IO(G)$ is $\{a, d, g\}$. There are many complete paths in G including;

$\langle ia_1, ib_1, c_1, ie_1, ig_1 \rangle$,

$\langle ia_1, ib_1, c_1, if_1, ib_2, ic_2, ie_1, ig_1 \rangle$,

$\langle ia_1, ib_1, c_1, if_1, ib_2, ic_2, if_2, ib_3, ic_3, ie_1, ig_1 \rangle$, etc...

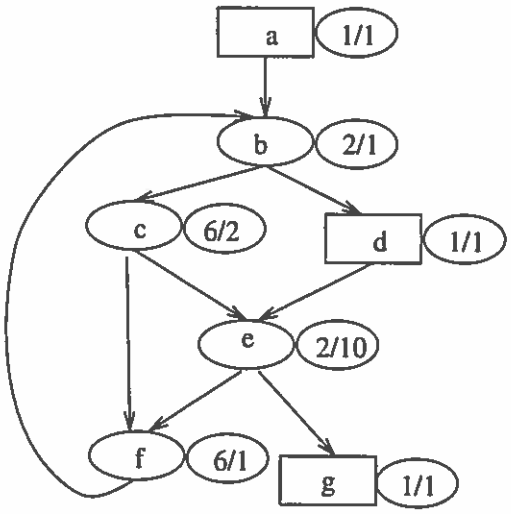


Figure 16: An example of a control flow graph

The set $CP(G)$, in this case, is *infinite.*

The set $CSP(G)$ is $\{\langle ia_1, ib_1, ic_1, ie_1, ig_1\rangle, \langle ia_1, ib_1, id_1, ie_1, ig_1\rangle\}$.

The set $SC(G)$ is $\{\langle b, c, f, b\rangle, \langle b, c, e, f, b\rangle, \langle b, d, e, f, b\rangle\}$. Thus, the set $MSRP(G)$ is $\{\langle b, c, f\rangle, \langle b, c, e, f\rangle, \langle b, d, e, f\rangle\}$. Consider a complete path $\langle ia_1, ib_1, ic_1, if_1, ib_2, ic_2, if_2, ib_3, ic_3, ie_1, ig_1\rangle$ over the graph G. The first node ia_1 is an instance of node a and is a part of a *csp*. The maximum simple repeatable path $\langle b, c, f\rangle$ is repeated twice after ia_1. The rest of the complete simple path $\langle ib_3, ic_3, ie_1, ig_1\rangle$ finishes the *cp*. □

Suppose a path $p = \langle a_1, \ldots, a_{k-1}, a_k\rangle$ in G is given. The execution time required for a path p on a machine M is represented as $ExecTime(p, M)$ as defined in Equation III.5.

$$ExecTime(p, M) = \sum_{i=1}^{k}(ExecTime(a_i, M)) \qquad \text{(III.5)}$$

For the given path p, $EST(p)$ is $((ExecTime(p, M_S), ExecTime(p, M_T))$. The

execution time difference between source and target machine for the path p, $STD(p)$, is defined as $ExecTime(p, M_T) - ExecTime(p, M_S)$. Consider two nodes $x, y \in V$. A $\sigma_i \in \sigma(x, y)$ is a simple path from x to y. The function $MaxEST(x, y)$ returns the $EST(\sigma_p)$ such that

$$STD(\sigma_p) = \max_{\sigma_i \in \sigma(x,y)} (STD(\sigma_i(x, y))). \tag{III.6}$$

The function $MinEST(x, y)$ returns the $EST(\sigma_q)$ such that

$$STD(\sigma_q) = \min_{\sigma_i \in \sigma(x,y)} (STD(\sigma_i(x, y))). \tag{III.7}$$

Example 3.2

Consider the control flow graph given in Figure 16, again. The set $\sigma(b, f)$ is $\{\langle b, c, f\rangle, \langle b, c, e, f\rangle, \langle b, d, e, f\rangle\}$.

$EST(\langle b, c, f\rangle)$ = (14,4), $EST(\langle b, c, e, f\rangle)$ = (16,14), and $EST(\langle b, d, e, f\rangle)$ = (11,13).

$STD(\langle b, c, f\rangle)$ = -10, $STD(\langle b, c, e, f\rangle)$ = -2, and $STD(\langle b, d, e, f\rangle)$ = 2.

Thus, $MaxEST(b, f)$ returns (11,13) and $MinEST(b, f)$ returns (14,4).

□

Spanning Trees and Edges

A *spanning tree* of $G = \langle V_G, E_G, Entry_G, \rangle$ is a tree $ST = \langle V_T, E_T, Root_T \rangle$ such that $V_T = V_G$, $E_T \subseteq E_G$ and $Root_T = Entry_G$. In general, ST of a G is not unique by definition. Two different spanning trees of G have two different subsets of edges from E_G. Given a spanning tree ST of a graph G, edges in G are partitioned into four groups.

- *Tree edges* are edges $\langle v_1, v_2 \rangle$ in G that are also edges in ST.
- *Forward edges* are edges $\langle v_1, v_2 \rangle$ in G that are not in ST but v_2 is a descendant of v_1 in ST.
- *Retreating edges* are edges $\langle v_2, v_1 \rangle$ in G such that $v_2 = v_1$ or v_1 is an ancestor of v_2 in ST.
- *Cross edges* are edges $\langle v_1, v_2 \rangle$ in G such that v_2 is neither an ancestor nor a descendant of v_1 in ST.

Example 3.3

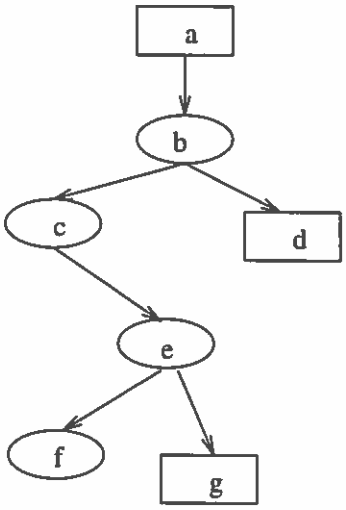


Figure 17: A spanning tree of the CFG shown in Figure 16

Figure 17 shows a spanning tree ST of the control flow graph G given in Figure 16. For the given ST, $d \rightarrow e$ is a *cross edge*, $c \rightarrow f$ is an *advancing edge* and $f \rightarrow b$ is a *back edge*. □

Dominators and The Dominator Tree

A node x dominates node y, denoted by $x \overset{DOM}{\rightarrow} y$, if every path from the *Entry* to y includes x. If $x \overset{DOM}{\rightarrow} y$, then any path $Entry \rightarrow y$ can be split into two parts: $Entry \overset{*}{\rightarrow} x$ and $x \overset{*}{\rightarrow} y$. A node x strictly dominates y, denoted by $x \overset{SDOM}{\rightarrow} y$, if $x \neq y$ and $x \overset{DOM}{\rightarrow} y$. A node x immediately dominates y, denoted by $x \overset{IDOM}{\rightarrow} y$, if $x \overset{SDOM}{\rightarrow} y$ and there is no node z such that $x \overset{DOM}{\rightarrow} z$ and $z \overset{DOM}{\rightarrow} y$. It is easy to see that each node has a unique immediate dominator if it has any [41]. The *dominator tree (DOM tree in short)* of a control flow graph $G = \langle V_G, E_G, Entry_G \rangle$ is a tree $DT = \langle V_T, E_T, Root_T \rangle$, where $V_T = V_G$, $Root_T = Entry_G$ and E_T is the set of edges of the form $x \overset{IDOM}{\rightarrow} y$. The DOM tree of a G is unique since every node has a unique immediate dominator [41]. A node u is a *proper ancestor* of v if $u \overset{SDOM}{\rightarrow} v$. An algorithm to find dominator relationship can be found in [41, 54]. A more efficient algorithm is shown in [38]. A *back edge* in G is an edge $\langle v_1, v_2 \rangle$ such that v_2 is a predecessor of v_1 in the DOM tree. A back edge also defines a loop which is different than that defined by a retreating edge in a spanning tree.

Example 3.4

Figure 18 gives the DOM tree of the control flow graph shown in Figure 16. There is only one back edge in G, which is $f \rightarrow b$.

□

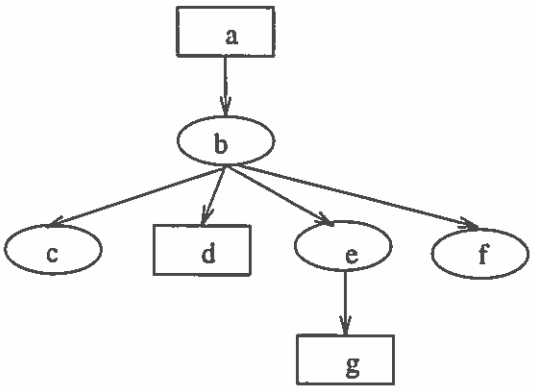


Figure 18: The dominator tree of the CFG shown in Figure 16

Summary

In this chapter, we provided some mathematical notations. Using these notations, we defined our control flow graph. The definitions of spanning trees and the dominator tree of a control flow graph are also provided. A summary of notations is shown in Table 1.

Definition	Description
p°	A sub-path $\langle a_0, \ldots, a_{k-1} \rangle$ of $p = \langle a_0, \ldots, a_k \rangle$
$\rho(x, y)$	Set of all paths from x to y
$\sigma(x, y)$	Set of all simple paths from x to y
$EST(v)$	$(ExecTime(v, M_S), ExecTime(v, M_T))$
$STD(v)$	$ExecTime(v, M_T) - ExecTime(v, M_S)$
$CP(G)$	Set of all complete paths in G
$SC(G)$	Set of all simple cycles in G
$MSRP(G)$	Set of all maximum simple repeatable paths in G
$MaxEST(x, y)$	$EST(\sigma_p(x, y))$ such that $STD(\sigma_p(x, y)) = \max_{\sigma_i(x,y) \in \sigma(x,y)}(STD(\sigma_i(x, y)))$.
$MinEST(x, y)$	$EST(\sigma_q(x, y))$ such that $STD(\sigma_q(x, y)) = \min_{\sigma_i(x,y) \in \sigma(x,y)}(STD(\sigma_i(x, y)))$.

Table 1: A summary of notations

CHAPTER IV

TIMING EQUIVALENT TRANSLATION

A real-time translator must preserve temporal equivalence as well as functional equivalence. This chapter defines temporal equivalence of the translated target program. We first define two timing sensitivities, absolute and relative timing sensitivities, of a given complete path. These timing sensitivities extend to the program. Using these concepts of timing sensitivity, three different levels of temporal equivalence, *equivalence, invariance and divergence,* are defined. The problem of deciding timing equivalence for a target program is discussed in Chapter V.

Execution Instances and Complete Paths

In Chapter III, we introduced an abstract model to represent possible flow of control in a program called a control flow graph. The control flow graph abstraction does not exactly model the flow of control in a program. Consider the program given in Figure 13. The control flow graph of this program is shown in Figure 19. There is only one execution instance for the given program which executes the loop `every (10ms) do` forever. However, there are an infinite number of complete paths in the control flow graph but every given complete path is of finite length. In fact, any control flow graph that has cycles has an infinite number of complete paths.

In general, this abstraction is acceptable since for many loops the number of iterations to be executed is not known statically. Since the control flow graph does

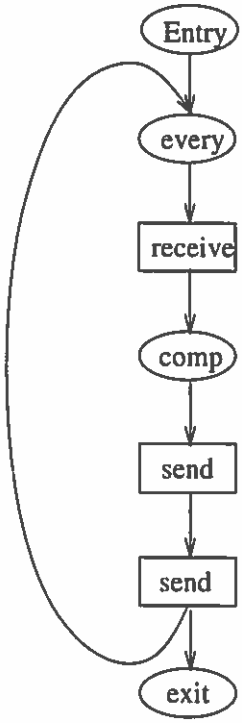


Figure 19: The control flow graph for the program shown in Figure 13

not contain this information, it is assumed that any loop (cycle) may execute infinitely, and thus can generate infinite many complete paths. One advantage of this assumption in our timing equivalence analysis is that it is easier to distinguish timing differences accumulated by the repeated execution of a loop. These accumulated timing differences by loops are too big to be accepted for most real-time systems even though the difference is limited by a constant. For any loop where the number of iterations is not known, it is covered by the infinite number of complete paths for the graph G. Thus, for our timing equivalence analysis, complete paths of a control flow graph are used instead of an application which is an execution instance of the program.

Global Virtual Clocks and Synchronizations

Two global virtual clocks, source and target, are maintained throughout the analysis. These global clocks are used to keep track of execution times of complete paths. The source clock, *accumulated source time* ($ASTime$), and the target clock, *accumulated target time* ($ATTime$), are increased as the path progresses along nodes in the control flow graph by the time required for these nodes on the source and target machine, respectively. The source and target clocks are obtained by inserting the following two statements on every $v \in V - \{Entry\}$. Both $ASTime$ and $ATTime$ are initialized to zero at the $Entry$ node.

$$ATTime \Leftarrow ATTime + ExecTime(v, M_T)$$
$$ASTime \Leftarrow ASTime + ExecTime(v, M_S)$$

Consider a $cp = \langle cpv_1, cpv_2, \ldots, cpv_p, cpv_q, \ldots, cpv_k \rangle$. The $ASTime$ at cpv_q is $\sum_{i=0}^{p}(ExecTime(cpv_i, M_S))$ and $ATTime$ at cpv_q is $\sum_{i=0}^{p}(ExecTime(cpv_i, M_T))$.

In a complete path $cp \in CP(G)$, there may exist many instances of a node $v \in V$, denoted by iv_i for some i, if the node v supports cycles in G. For a given cp, the function $AST(iv_i)$ returns $ASTime$ at iv_i and the function $ATT(iv_i)$ returns $ATTime$ at iv_i. Here, $AST(iv_i)$ is the same as $ATV(iv_i, M_S)$ and $ATT(iv_i)$ is the same as $ATV(iv_i, M_T)$ except $AST(iv_i)$ and $AST(iv_i)$ are defined over complete paths and thus are computed, but $ATV(iv_i, M_T)$ and $ATV(iv_i, M_S)$ are defined over execution instances and thus are measured.

Since we use static synchronization methods, the synchronization is performed over the control flow graph by inserting synchronization functions on either nodes or edges. When the synchronization function is inserted on edge $\langle v, w \rangle$, every instance of the edges on every $cp \in CP(G)$ are to be synchronized. The synchronization function we use is shown in Figure 20.

```
Sync(⟨v, w⟩) {
    if ATT(iw_i) < AST(iw_i)
        Delay(AST(iw_i) − ATT(iw_i))
}
```

Figure 20: Synchronization algorithm

In this algorithm, synchronization of the edge is *valid* only when $ATT(iw_i) < AST(iw_i)$. The basic function to delay program execution on the target machine is $Delay(dtime)$ which postpones program execution for $dtime$. We assume the target system provides functionality to implement $Delay(dtime)$.

Example 4.1

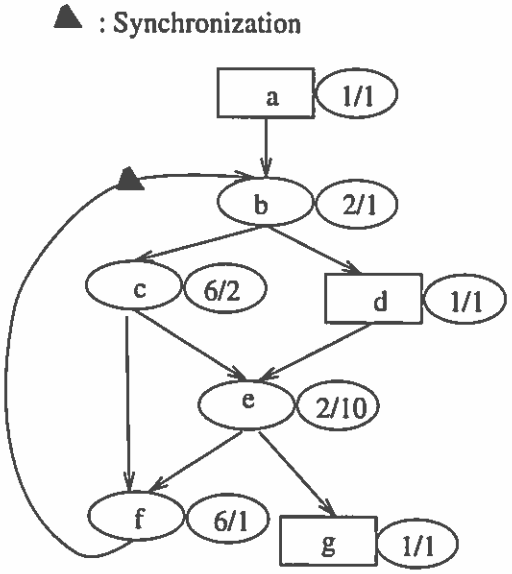


Figure 21: A control flow graph with a synchronized edge

Figure 21 shows the control flow graph shown in Figure 16 with one synchronized edge. Consider the complete path

$$\langle ia_1, ib_1, ic_1, if_1, ib_2, ic_2, if_2, ib_3, ic_3, ie_1, ig_1 \rangle$$

over the graph G. Since the edge $\langle f, b \rangle$ is synchronized in G every instance of the edges in the *cp* must be synchronized. For the given *cp* there are two instances of this edge, i.e., $\langle if_1, ib_2 \rangle$ and $\langle if_2, ib_3 \rangle$. In this *cp*, $AST(ib_2) = 15$, $ATT(ib_2) = 5$, $AST(ib_3) = 29$ and $ATT(ib_3) = 9$. Thus, after the execution of if_1, the execution must be delayed for 10 which delays all following instances of nodes in the *cp*. The *cp* resumes its execution of ib_2

when $ATTime = ASTime = 15$. Because of the delay after if_1, $ATT(ib_3)$ now becomes 19. After the execution of if_2, the execution is delayed for 10 to synchronize ib_3 at $ASTime = ATTime = 29$. □

Timing Sensitivity

Timing sensitivity is used to represent temporal differences between source and target programs. With a given control flow graph G, two different timing sensitivities, i.e., absolute timing sensitivity ($\Delta(G)$) and relative timing sensitivity ($\Psi(G)$) are to be defined.

Absolute Timing Sensitivity

One intuitive concept of timing sensitivity is the maximum timing difference between corresponding i/o events in source and target programs. These differences are critical for many real-time systems. Absolute timing sensitivity addresses these timing differences between corresponding i/o events on source and target programs.

Suppose a sub-path p of a $cp \in CP(G)$ with an i/o event pe is given. The path p is denoted by ps if it is running on the source machine and pt if it is running on the target machine. The i/o events pe on p become pse on ps and pte on pt. The absolute timing sensitivity of pe, denoted by $\Delta(pe)$, is defined in Equation IV.8. Figure 22 depicts absolute timing sensitivity.

$$\Delta(pe) = \mid ATT(pe) - AST(pe) \mid \qquad \text{(IV.8)}$$

The absolute timing sensitivity of cp with n i/o events, denoted by $\Delta(cp)$, is defined in Equation IV.9.

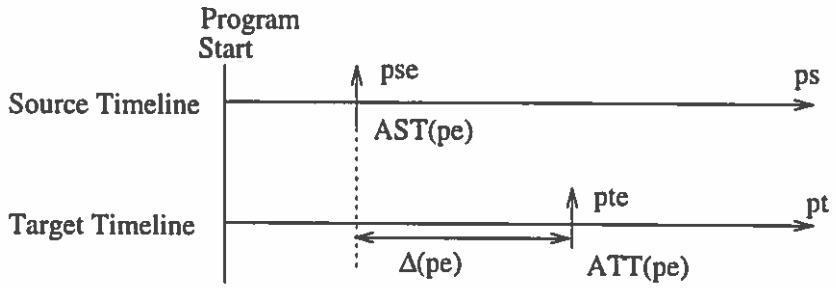


Figure 22: Absolute timing sensitivity

$$\Delta(cp) = \max_{i=1}^{n} \Delta(cpe_i) \tag{IV.9}$$

Example 4.2

Consider the example shown in Figure 23. The source execution path *cps* has four i/o events, $cpse_1$ at $AST(cpe_1) = 10$, $cpse_2$ at $AST(cpe_2) = 31$, $cpse_3$ at $AST(cpe_3) = 48$ and $cpse_4$ at $AST(cpe_4) = 67$. The total execution time of the path is 84. Total execution time of the translated execution path *cpt* is 77. The corresponding target i/o events $cpte_i$ executed at $cpte_1$ at $ATT(cpe_1) = 6$, $cpte_2$ at $ATT(cpe_2) = 22$, $cpte_3$ at $ATT(cpe_3) = 54$ and $cpte_4$ at $ATT(cpe_4) = 66$.

The absolute timing sensitivity of *cp* ($\Delta(cp)$) is the maximum value of $\Delta(cpe_1) = 4$, $\Delta(cpe_2) = 9$, $\Delta(cpe_3) = 6$, and $\Delta(cpe_4) = 1$, which is 9. □

The absolute timing sensitivity of G, denoted by $\Delta(G)$, is defined in Equation IV.10.

$$\Delta(G) = \max_{cp \in CP(G)} \Delta(cp) \tag{IV.10}$$

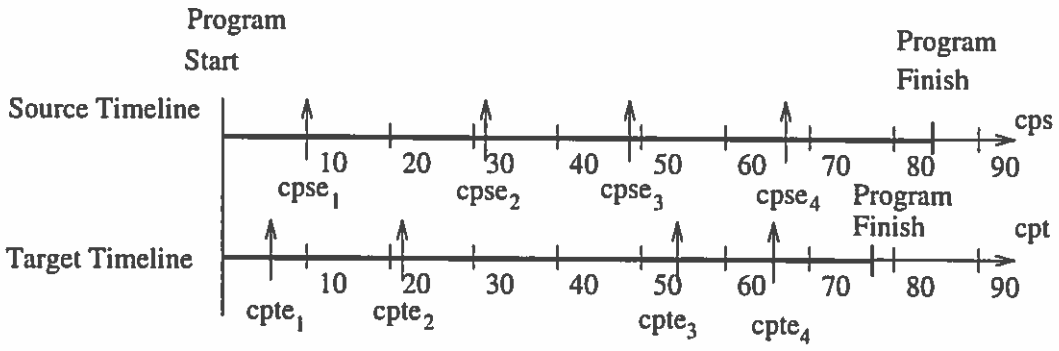


Figure 23: A complete path on the source and target machine

Relative Timing Sensitivity

In many real-time applications, maintaining the execution time between two consecutive i/o events is also critical. Relative timing sensitivity addresses this issue.

Suppose a sub-path p of $cp \in CP(G)$ with two i/o events pe_i and pe_{i+1} is given. The path p is denoted by ps if it is running on the source machine and pt if it is running on the target machine. Two i/o events pe_i and pe_{i+1} on p becomes pse_i and pse_{i+1} on ps and pte_i and pte_{i+1} on pt. The relative timing sensitivity between two i/o events, denoted by $\Psi(cpe_i, cpe_{i+1})$, is defined in Equation IV.11. Figure 24 depicts the relative timing sensitivity between two i/o events.

$$\Psi(pe_i, pe_{i+1}) = \mid TD(pte_i, pte_{i+1}) - TD(pse_i, pse_{i+1}) \mid \qquad \text{(IV.11)}$$

The relative timing sensitivity of the path cp with n i/o events, denoted by $\Psi(cp)$, is defined in Equation IV.12.

$$\Psi(cp) = \max_{i=0}^{n+1} \Psi(cpe_i, cpe_{i+1}) \qquad \text{(IV.12)}$$

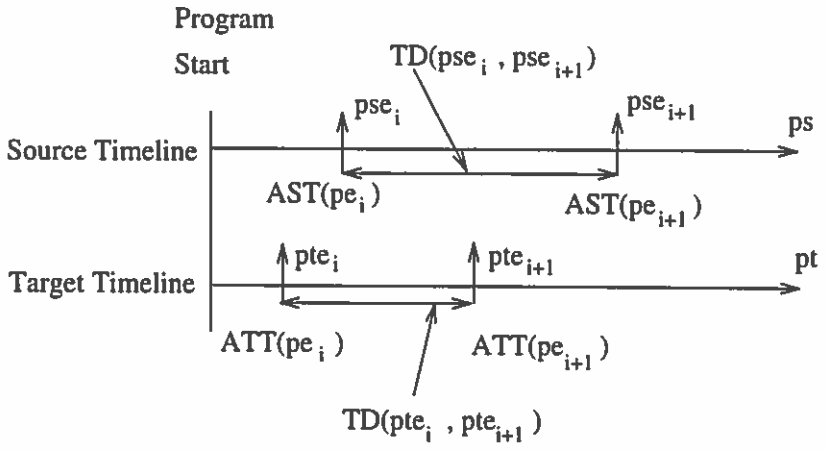


Figure 24: Relative timing sensitivity

In Equation IV.12, the event pe_0 is the *Program Start* event (E_s) and pe_{n+1} is *Program Finish* event (E_f).

Example 4.3

Consider the example shown in Figure 23 again. The relative timing sensitivity, $\Psi(cp)$, of the execution path cp is the maximum of $\Psi_{(cpe_0,cpe_1)}$ $= \mid 6 - 10 \mid = 4$, $\Psi_{(cpe_1,cpe_2)} = \mid 16 - 21 \mid = 5$, $\Psi_{(cpe_2,cpe_3)} = \mid 32 - 17 \mid = 15$, $\Psi_{(cpe_3,cpe_4)} = \mid 12 - 19 \mid = 7$, and $\Psi_{(cpe_4,cpe_5)} = \mid 11 - 17 \mid = 6$, which is 15. □

Theorem 4.1

$$0 \leq \Psi(cp) \leq 2 \times \Delta(cp)$$

□

Proof

$0 \leq \Psi(cp)$ part is obvious. For any i/o event cpe_i on cp, $\Delta(cpe_i) \leq \Delta(cp)$,

by definition. Two cases where $\Psi(cp) = 2 \times \Delta(cp)$ are shown in Figure 25. It is easy to see $\Psi(cp) < 2 \times \Delta(cp)$ for all other cases.

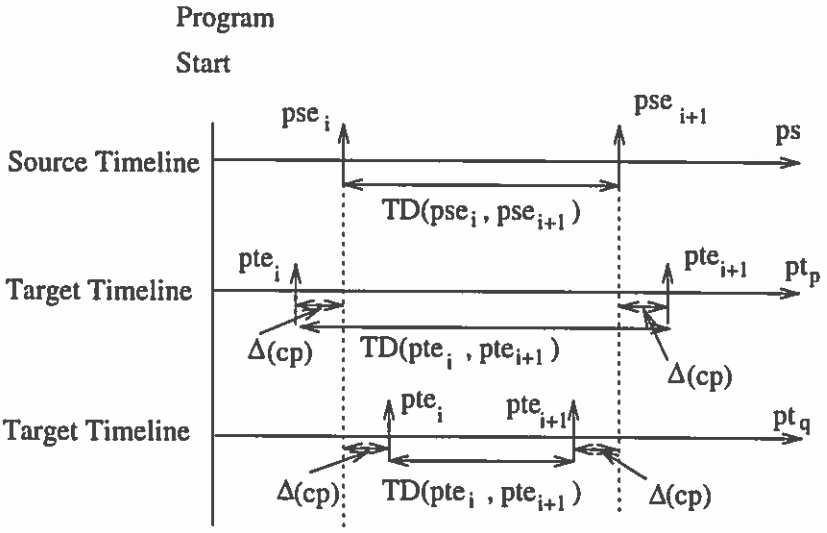


Figure 25: Two cases where $\Psi(cp) = 2 \times \Delta(cp)$

□

The relative timing sensitivity of G, $\Psi(G)$, is defined in Equation IV.13.

$$\Delta(G) = \max_{cp \in CP(G)} \Psi(cp) \tag{IV.13}$$

Time Deadline

Another definition we use for a path is *time deadline*. Again, suppose a sub-path p of a $cp \in CP(G)$ is given. We say p meets its *time deadline* if $ExecTime(p, M_T) \leq ExecTime(p, M_S)$ or $STD(cp) \leq 0$.

Timing Equivalence, Invariance and Divergence

Using the definition of absolute timing sensitivity, we define three different levels of temporal equivalence (timing equivalence, invariance and divergence) of the program represented by G. We start defining them with a complete path.

Timing Equivalence of a Complete Path

The translated target path *cpt* is *timing equivalent* with respect to *cps*, denoted by $cpt \stackrel{TE}{=} cps$, if $\Delta(cp) = 0$. The translated target path *cpt* is *executable with timing equivalence* if there exist synchronization methods which guarantee the timing equivalence of *cpt*. If the translated target path *cpt* is executable with timing equivalence, it is timing equivalent when proper synchronizations are inserted.

Theorem 4.2

Assume a complete path *cp* has n i/o events. The complete path *cp* is executable with timing equivalence if and only if $TD(cpte_i, cpte_{i+1}) \leq TD(cpse_i, cpse_{i+1})$, $0 \leq i \leq n$. □

Proof

(If part)

It is easy to see that inserting a synchronization at every i/o event provides timing equivalence.

(Only if part)

Suppose there exist two i/o events cpe_i and cpe_{i+1} on *cp* such that

$$TD(cpte_i, cpte_{i+1}) > TD(cpse_i, cpse_{i+1}).$$

To be timing equivalent, synchronization of both i/o events must be valid. Assume we synchronized the i/o event cpe_i, i.e, $ATT(cpe_i) = AST(cpe_i,)$. In this case, synchronization of the i/o event cpe_{i+1} is invalid since $ATT(cpe_{i+1}) > AST(cpe_{i+1})$. Thus, $\Delta(cpe_i) = 0$ but $0 < \Delta(cpe_{i+1}) = ATT(cpe_{i+1}) - AST(cpe_{i+1})$. □

For any given target complete path *cpt*, there exists a constant C such that $\Delta(cp) \leq C$ since both $ATT(cp)$ and $AST(cp)$ are constant. Thus, any given *cpt* that is not executable with timing equivalence is timing invariant. However, there are infinite number of complete paths for a given graph G.

Timing Equivalence of a Target Program

Since there may exist infinite number of complete paths for a given graph G, three different levels of temporal equivalence are defined as follows.

Definition 4.1

The program represented by G is said to be *timing equivalent* if

$$(\forall cp \in CP(G))[\Delta(cp) = 0]. \qquad \text{(IV.14)}$$

□

Definition 4.2

The program represented by G is said to be *timing invariant* if there exist a constant C such that

$$(\forall cp \in CP(G))[\Delta(cp) \leq C]. \qquad \text{(IV.15)}$$

□

Definition 4.3

The program represented by G is said to be *timing divergent* if there is no constant C such that

$$(\forall cp \in CP(G))[\Delta(cp) \leq C]. \qquad \text{(IV.16)}$$

□

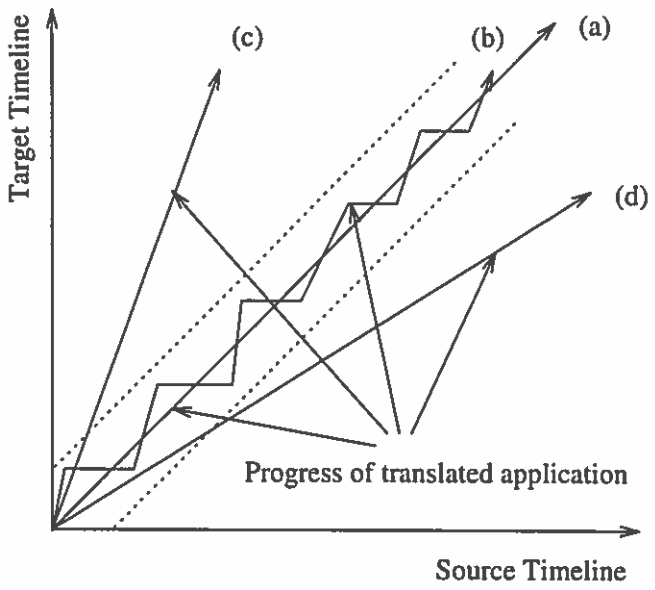


Figure 26: Timing equivalence, invariance and divergence of paths

Figure 26 depicts the difference between timing equivalence, invariance and divergence. The path (a) is timing equivalent, path (b) is timing invariant, and paths (c) and (d) are timing divergent.

However, some timing divergent paths, such as path (c), can be converted to timing equivalent or invariant ones by inserting synchronization, while other paths,

such as path (d), can not be converted to timing equivalent or invariant ones by inserting synchronizations. To distinguish these two cases, we define following:

Definition 4.4

The graph G is *executable with timing equivalence* if all $cp \in CP(G)$ are executable with timing equivalence. □

Definition 4.5

The graph G is *executable with timing invariance* if there exist a synchronization method that provides the timing invariance for all possible paths on G. □

Theorem 4.3

$\Delta(G) \leq C$ implies that there exists some constant C' such that $\Psi(G) \leq C'$, but $\Psi(G) \leq C'$ does not imply $\Delta(G) \leq C$. □

Proof

$\Delta(G) \leq C$ implies $\Psi(G) \leq 2C \leq C'$ by Theorem 4.1. There are cases such that $\Psi(G) \leq C'$ does not imply $\Delta(G) \leq C$. Consider a complete path cp given in Figure 27. Assume the cp contains multiple instances of a $msrp(sc)$. Since the set $CP(G)$ is *infinite*, there always exist another $cp \in CP(G)$ such that it has one more $msrp(sc)$ on it. Thus, even though $\Psi(G)$ is 10 and thus limited by a constant, $\Delta(G)$ is not limited by any constant. □

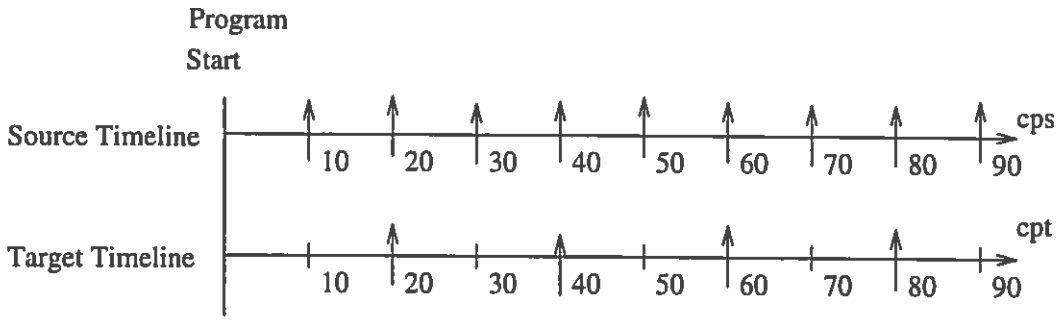


Figure 27: A complete path

Summary

In this chapter, we defined two timing sensitivities, i.e., absolute and relative timing sensitivity. Using these timing sensitivities, we defined three different levels of temporal equivalence (timing equivalence, invariance and divergence) of target programs. In the following chapter, we describe our approach to test timing equivalence of target programs.

CHAPTER V

TIMING EQUIVALENCE ANALYSIS

This chapter discusses issues in testing timing equivalence or invariance of a target program represented by a control flow graph. We first develop a method to test if a target program is executable with timing invariance. This problem is converted into a data-flow problem which can be solved using intervals. If a target program is executable with timing invariance, we also test if it is executable with timing equivalence. A program that is executable with timing equivalence can be converted to an equivalent one by inserting synchronization on every i/o node.

If a target program is executable with timing invariance but not with timing equivalence, the maximum timing error of the target program depends on how and where it is synchronized. Synchronization schemes which reduce the maximum timing error are presented in Chapter VI.

Static methods which measure timing sensitivities of target programs are presented in Chapter VII.

Problem Statement

The problem is how to test if the given graph G is executable with timing equivalence or invariance. If the graph G is executable with timing invariance, synchronizations must be inserted on G to convert it into a timing invariant one.

We defined timing invariance of G in terms of $CP(G)$. In most cases, however,

the set $CP(G)$ is *infinite*. Thus, it is not possible to check all $cp \in CP(G)$. Here, we present an algorithm which tests and provides timing invariance of G without testing all $cp \in CP(G)$.

A complete path $cp \in CP(G)$ is composed of simple paths, repeatable and non-repeatable, over the graph G. If all repeatable paths meet their time deadline, synchronization of these repeatable paths provides timing invariance. It is obvious that any repeatable path is a sub-path of a cycle since any path that is not a sub-path of cycle is non-repeatable. All repeatable paths in G can be found by finding all *maximum simple repeatable paths*($MSRP(G)$) of G. The set $MSRP(G)$ can be found by finding the set $SC(G)$.

The absolute timing sensitivity of G, defined in Equation IV.10, is the maximum timing difference between corresponding i/o events on source and target paths,where an i/o event is an execution instance of an i/o node in G. Consider an i/o node $io \in IO(G)$. The node io may have many instances in a given $cp \in CP$ if it supports cycles. The set $INST(io, cp) = \{iio_1, \ldots, iio_n\}$ is the set of execution instances of io in cp. The absolute timing sensitivity of io on cp, denoted by $\Delta(io, cp)$, can be found as in Equation V.17.

$$\Delta(io, cp) = \max_{inst \in INST(io,cp)} \Delta(inst), \tag{V.17}$$

where $\Delta(inst) = \mid ATT(inst) - AST(inst) \mid$.

Many i/o nodes may support a cp. Let the set $CPIO(cp) = \{io_1, \ldots, io_{nio}\}$ be the set of all i/o nodes that support cp. The absolute timing sensitivity of cp can be

found as in Equation V.18.

$$\Delta(cp) = \max_{cpio \in CPIO} \Delta(cpio, cp) \tag{V.18}$$

Also, an io may support many $cp \in CP(G)$. Let the set $IOCP(io)$ be a subset of $CP(G)$ such that io supports the $iocp \in IOCP(G)$ at least once. The absolute timing sensitivity of io, denoted by $\Delta(io)$ can be found as in Equation V.19.

$$\Delta(io) = \max_{iocp \in IOCP(G)} \Delta(io, iocp) \tag{V.19}$$

The absolute timing sensitivity of G can be found as in as in Equation V.20.

$$\Delta(G) = \max_{io \in IO(G)} \max_{iocp \in IOCP(G)} \Delta(io, iocp) = \max_{iocp \in IOCP(G)} \max_{io \in IO(G)} \Delta(io, iocp) \tag{V.20}$$

Theorem 5.1

A control flow graph G is executable with timing invariance [Definition 4.5] if and only if every maximum simple repeatable path meets its time deadline, i.e.,

$$(\forall sc \in SC(G))[STD(msrp(sc)) \leq 0]. \tag{V.21}$$

□

Proof

(If Part)

We prove that inserting a $Sync()$ on an edge of every $sc \in SC(G)$ provides timing invariance to G if the Equation V.21 is satisfied. By Equation V.20, the $\Delta(G)$ is limited by a constant if for all $io \in IO(G)$, $\Delta(io)$ is not ∞.

Consider an $io \in IO(G)$ and the set $IOCP(io)$. If io supports cycles, the set $IOCP(io)$ is an infinite set. If for every $iocp \in IOCP(io)$, $ATT(INST(io, iocp)) - AST(INST(io, iocp))$ is limited by a constant C, then $\Delta(io)$ is limited by C. Let the set $IOMSRP(io)$ be a sub-set of $MSRP(G)$ such that io supports $iomsrp \in IOMSRP(io)$ and let $MIOMSRP(io)$ be the set $MSRP(G) - IOMSRP(io)$. Consider an $iocp_1 \in IOCP(io)$ with $\Delta(iocp) = C_1$. Consider a longer path $iocp_2 \in IOCP(io)$ which is composed of $iocp_1$ and a $msrp_1 \in MSRP(G)$. The absolute timing sensitivity $\Delta(iocp_2)$ does not decrease since $msrp_1 \in MSRP(G)$ is synchronized. Thus, $\Delta(iocp_2)$ is the maximum of $\Delta(iocp_2)$ and $\Delta(msrp)$ unless $msrp_1$ is inserted in the middle of an instance of $msrp_2 \in MSRP(G)$ over the $iocp_1$.

When $msrp_1$ is inserted in the middle of an instance of $msrp_2 \in MSRP(G)$ over the $iocp_1$, synchronization of $msrp_1$ may cause $ASTime < ATTime$ at the synchronization point of $msrp_2$ which makes the synchronization *invalid*. An example of this is shown in Example 5.2. Once this happens, any synchronization after this are invalid until $ATTime < ASTime$. However, since every $msrp$ meets its time deadline, $ATTime$ will be smaller than $ASTime$ by repeated execution of $msrp$.

The maximum of $ATTime - ASTime$ for any i/o nodes can be decided statically by examination of all possible paths from all previous synchronization points. Thus, $ATT(io) - AST(io)$ does not increase indefinitely.

(Only If Part)

Conceptually, if any $msrp \in MSRP(G)$ does not meet its time deadline, $\Delta(G)$ depends on how many times this path is executed in a $cp \in CP(G)$. However, the number of executions for the path is not decidable statically since we only use control flow graph. Thus, the maximum value of $ATTime - ASTime$ for any instance of an $io \in IO(G)$ can not be decided statically. More formally, assume $\Delta(iocp_i) = C$ for a $iocp_i$ and $msrp_i$ does not meet its time deadline. There always exists another $iocp_j \in IOCP(io)$ which is composed of $iocp_1$ and a $msrp_i \in MSRP(G)$. The $\Delta(iocp_j)$ is bigger than the $\Delta(iocp_i)$ since $0 \leq STD(msrp_i)$. In this case, any strategy that uses $Delay()$ on the target machine will fail to provide timing invariance. Thus, the condition Equation V.21 is necessary to provide timing invariance.

□

Example 5.1

Consider the graph G shown in Figure 21 again. There are two complete simple paths in G, i.e., $\langle a, b, c, e, g \rangle$ and $\langle a, b, d, e, g \rangle$. The set $MSRP(G)$ is $\{mrsp_1 = \langle b, c, f \rangle, mrsp_2 = \langle b, c, e, f \rangle, mrsp_3 = \langle b, d, e, f \rangle\}$.

By inserting a synchronization on the edge $\langle f, b \rangle$, all $sc \in SC(G)$ are synchronized. The execution time of these paths on the source and target machine are described in Table 2.

Since there exists a $msrp \in MSRP(G)$ that does not meet its time deadline, the graph G is not executable with timing invariance. Consider a

$msrp_i$	$ExecTime(msrp(sc_i), M_S)$	$ExecTime(msrp(sc_i), M_T)$
$msrp_1$	14	4
$msrp_2$	16	14
$msrp_3$	11	13

Table 2: Execution Time on Source and Target Machines for All Simple Paths in Figure 21

complete path $iocp \in IOCP(d)$,

$$\langle ia_1, ib_1, ic_1, if_1, ib_2, id_1, ie_1, if_2, ib_3, id_2, ie_2, if_3, ib_4, ic_2, ie_3, ig_1 \rangle$$

over the graph G. The given $iocp$ is composed of one csp ($\langle a, b, c, e, g \rangle$) and two $msrp$ ($\langle b, c, f \rangle$ and $\langle b, d, e, f \rangle$). Since the edge $\langle f, b \rangle$ is synchronized, all instances of the edges on the $iocp$ are to be synchronized. Three instances of the edge $\langle f, b \rangle$ are $\langle if_1, ib_2 \rangle$, $\langle if_2, ib_3 \rangle$ and $\langle if_3, ib_4 \rangle$.

Since $AST(ib_2) = 15$ and $ATT(ib_2) = 5$, the execution after if_1 is delayed for 10 to synchronize at ib_2.

The $\Delta(d_1, iocp) =\mid 16-17 \mid= 1$. The next synchronization at ib_3 is invalid, since $AST(ib_3) = 26$ and $ATT(ib_3) = 28$. Here the $\Delta(d_2, iocp) =\mid 27 - 16 \mid= 1$. The next synchronization at ib_4 is also invalid, since $AST(ib_4) = 37$ and $ATT(ib_4) = 41$. Thus, $\Delta(d, iocp) = 1$. It is also easy to see that for any $iocp_i \in IOCP(io)$ there exists $iocp_j \in IOCP(io)$ such that $\Delta(io, iocp_i) < \Delta(io, iocp_j)$ by having another $iomsrp \in IOMSRP(io)$ over $iocp_i$.

□

Example 5.2

Consider the example given in Figure 28. There are four complete simple paths in G, i.e., $\langle a, d, e, f, h, k\rangle$, $\langle a, d, e, g, h, k\rangle$, $\langle a, b, c, d, e, f, h, k\rangle$ and $\langle a, b, c, d, e, g, h, k\rangle$. There are four simple cycles in G, i.e., $sc_1 = \langle d, e, f, i, d\rangle$, $sc_{21} = \langle e, g, j, e\rangle$, $sc_{22} = \langle e, f, h, e\rangle$ and $sc_{23} = \langle e, g, h, e\rangle$. The maximum simple repeatable path of each $sc \in SC(G)$ are $msrp(sc_1) = \langle d, e, f, i\rangle$, $msrp(sc_{21}) = \langle e, g, j\rangle$, $msrp(sc_{22}) = \langle e, f, h\rangle$ and $msrp(sc_{23}) = \langle e, g, h\rangle$.

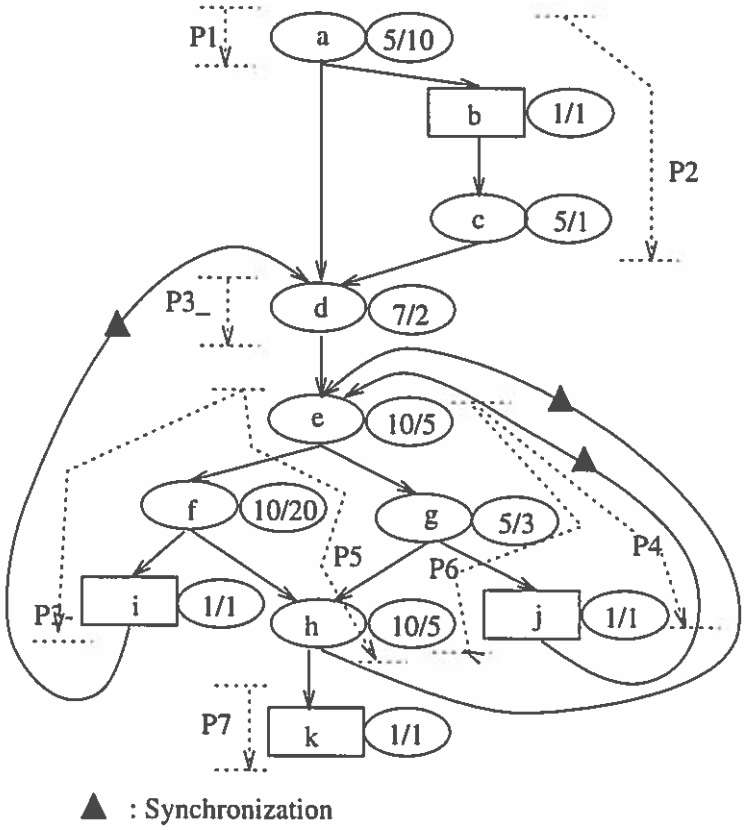


Figure 28: An Example of Control Flow Graph

By inserting a synchronization on the edges $\langle i, d\rangle$, $\langle j, e\rangle$ and $\langle h, e\rangle$, all

$sc \in SC(G)$ are synchronized. The execution time on source and target machine of these paths are as in Table 3.

$msrp(sc_i)$	$ExecTime(msrp(sc_i), M_S)$	$ExecTime(msrp(sc_i), M_T)$
$msrp(sc_1)$	28	28
$msrp(sc_{21})$	16	9
$msrp(sc_{22})$	30	30
$msrp(sc_{23})$	25	13

Table 3: Execution Time on Source and Target Machine for All Simple Paths in Figure 28

Since all $msrp \in MSRP(G)$ meet their time deadlines, the graph G is executable with timing invariance. Consider a complete path cp

$$\langle ia_1, id_1, ie_1, ig_1, ij_1, ie_2, if_1, ii_1, id_2, ie_3, ig_2, ih_1, ie_4, ig_3, ih_2, ik_1 \rangle$$

The given cp is composed of one csp ($\langle a, d, e, g, h, k \rangle$) and three $msrp$ ($\langle e, g, j \rangle$, $\langle e, g, h \rangle$ and $\langle d, e, f, i \rangle$). Note that an instance of $\langle e, g, j \rangle$, $\langle ie_1, ig_1, ij_1 \rangle$ is inserted in the middle of an instance of $\langle d, e, f, i \rangle$, $\langle id_{,1}, ie_2, if_1, ii_1 \rangle$. Since three edges $\langle i, d \rangle$, $\langle j, e \rangle$ and $\langle h, e \rangle$ are synchronized, all instances of these edges on the cp are to be synchronized. Three instances of these edges are $\langle ij_1, ie_2 \rangle$, $\langle ii_1, id_2 \rangle$ and $\langle ih_1, ie_4 \rangle$.

Since $AST(ie_2) = 28$ and $ATT(ie_2) = 21$, the execution after ij_1 is delayed for 7 to synchronize at ie_2. By synchronizing the edge $\langle ij_1, ie_2 \rangle$, the next synchronization point id_2 is invalid since $AST(id_2) = 49$ and $ATT(id_2) = 54$ and thus $0 < ATTime - ASTime$. However, since all $msrp \in MSRP(G)$ meet their time deadline, the value of $ATTime -$

$ASTime$ decreases until it becomes negative as we will see. The last synchronization point is $\langle ih_1, ie_4 \rangle$. Since $AST(ie_4) = 81$ and $ATT(ie_4) = 69$, the execution after ih_1 is delayed for 12 to synchronize at ie_4. Since $\Delta(j, cp) =| 20 - 17 |) = 7$, the $\Delta(i, cp) =| 53 - 58 |= 5$ and $\Delta(k, cp) =| 94 - 106 |= 12$, the absolute timing sensitivity of the cp, $\Delta(cp) = 12$. This cp on the source and target machine is shown in Figure 29.

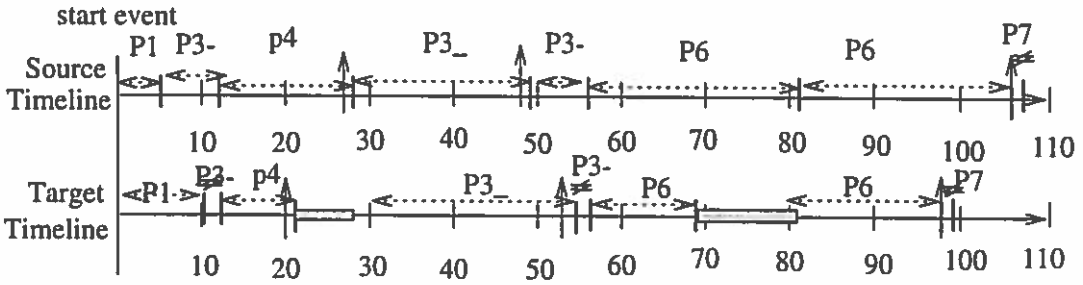


Figure 29: A complete path divided into simple paths

□

If G is executable with timing invariance, $Sync()$ must be inserted in the graph G to provide timing invariance. Let SG be a synchronized version of G. Where and how G is synchronized affects timing sensitivities of SG significantly. We will discuss more on this issue later. If $\Delta(SG) = 0$, SG is timing equivalent and G is executable with timing equivalence.

The problem now is how to find all cycles in a control flow graph. There exist different approaches in finding cycles in a control flow graph. A program decomposition method, *strongly connected components (SCC)* discussed in [53], finds all cycles in a control flow graph. This algorithm decomposes a control flow graph into strongly connected components. A *strongly connected component* is a maximal strongly con-

nected region, where a *strongly connected region* is a set of nodes S such that there is a path $S_1 \xrightarrow{*} S_2$ for any two nodes $S_1, S_2 \in S$. A cycle in a control flow graph forms a strongly connected region. Strongly connected components cover all form of loops, i.e., it is the most general form of loops. Strongly connected components uniquely partition the nodes of any directed graph. Once strongly connected components are found in a control flow graph, we can test timing invariance of the program by testing each strongly connected region. A target program is timing invariant if all strongly connected components in the control flow graph are timing invariant. A strongly connected component is timing invariant if all cyclic paths of strongly connected regions meet their time deadline. However, *strongly connected regions* in a *strongly connected component* may not be uniquely defined. Also, there may be multiple *Entry* and *Exit* points for a strongly connected component which may hinder our timing analysis.

An alternative method is using *natural loops*, discussed in [1]. A natural loop is a strongly connected region with a unique *Entry* called a *header* which dominates all nodes in the region. Even though natural loops provide a better solution in finding all repeatable paths in a control flow graph, they are not general enough to cover all kinds of loop structures found in programs. In the next section we shall discuss a more powerful decomposition method, called interval analysis.

Interval Analysis

Interval analysis, first invented by Allen and Cocke [4], has been used to facilitate data flow analysis. It finds hierarchical structures in a control flow graph by dividing it into intervals, which helps in solving data-flow equations efficiently.

An interval is defined with respect to the dominator tree of a control flow graph. A number of quite different definitions of interval have been proposed for better per-

formance and applicability [4, 56, 55, 18]. Ryder and Paul [48] present a comparison study between different definitions of interval. The definition of interval we use is essentially the same as *S-set* defined by Graham and Wegman [18].

Definition of Interval

For a given control flow graph G, there exists a unique dominator tree, denoted by $DT(G)$ [41]. Let $BE(h)$ is the set of dominator tree back edges in G, whose head node is h. A dominator tree back edge $be \in BE(h)$ defines a strongly connected region. An interval $I(h)$ is the union of strongly connected regions defined by the set $BE(h)$, where h is the *header* of the interval $I(h)$.

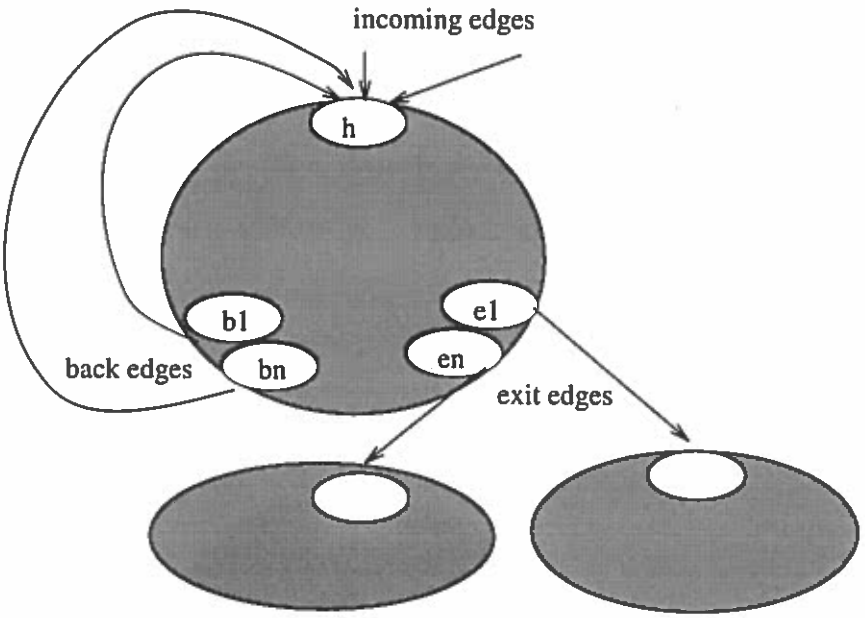
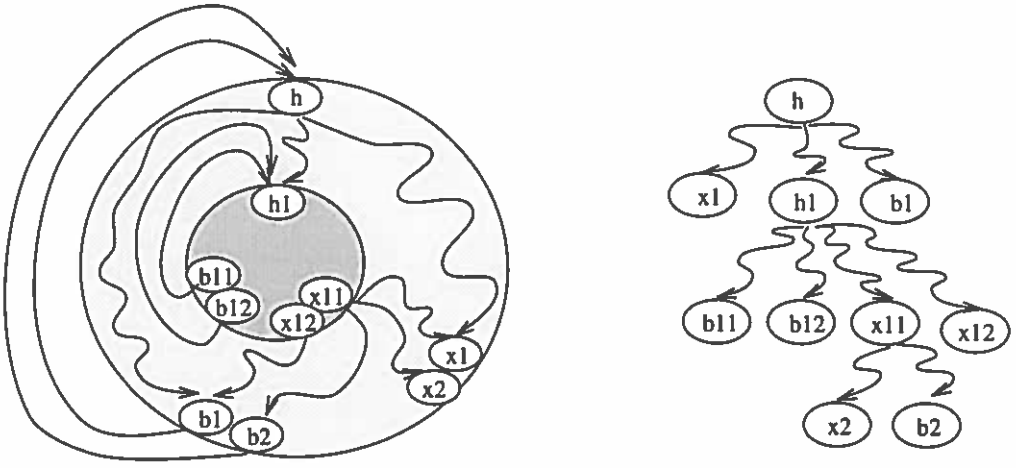


Figure 30: The structure of an interval

The structure of an interval is shown in Figure 30. Given an interval $I(h)$, edges are partitioned into four groups.

- *Exit edges* are edges $\langle v_1, v_2 \rangle$ such that $v_1 \in I(h)$ and $v_2 \notin I(h)$. We call v_1 an *exit node.* The set of all exit nodes for the interval $I(h)$ is denoted by $XN(I(h))$
- *Incoming edges* are edges $\langle v_1, v_2 \rangle$ such that $v_1 \notin I(h)$ and $v_2 \in I(h)$. The head of all incoming edges are the same node, i.e., the header node h, since there is only one entry for the interval $I(h)$.
- *Back edges* are edges $\langle v_1, v_2 \rangle$ such that $v_1 \in I(h)$ and $v_2 = h$. We call v_1 a *back node.* The set of all back node of the interval $I(h)$ is denoted by $BN(I(h))$.
- *Interval edges* are edges $\langle v_1, v_2 \rangle$ such that $v_1, v_2 \in I(h)$ and $v_2 \neq h$.

Intervals may be nested. An interval $I(h_i)$ is a sub-interval of $I(h_j)$ if (1) h_j dominates h_i and (2) every *exit node* of $I(h_i)$ has a path to either an *exit* or *back node* or both of $I(h_j)$ without passing h_j.



(a) A Compound Interval

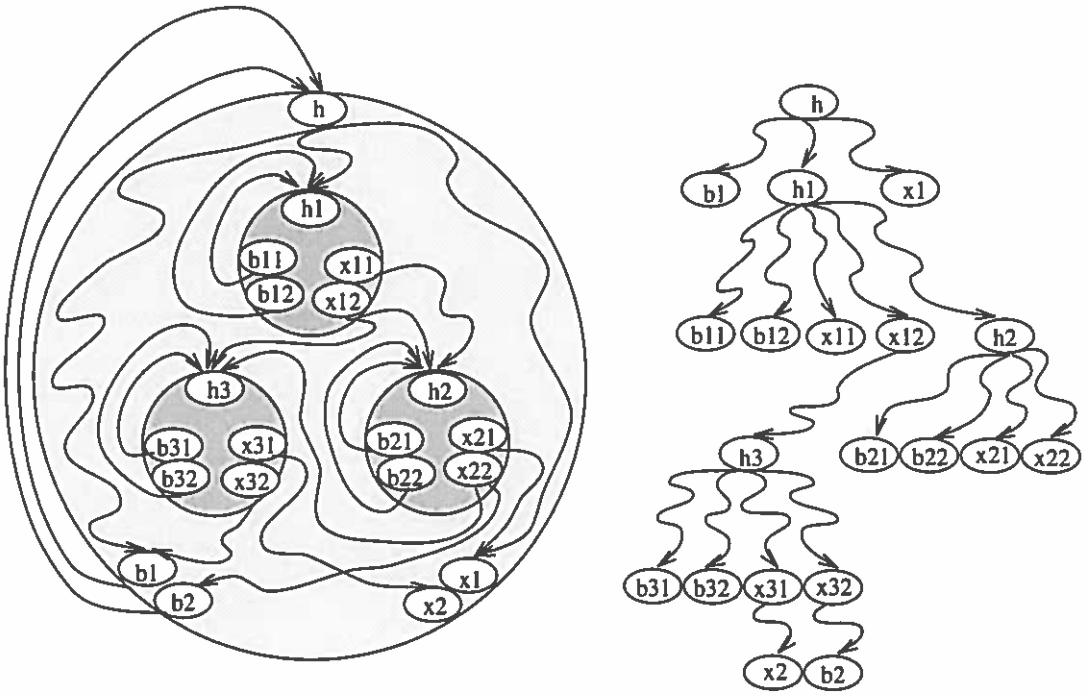
(b) DOM Tree for (a)

Figure 31: A nested interval and its dominator tree

Figure 31 shows a nested interval and its dominator tree, where the wiggly lines denote node disjoint (except for the endpoints) paths. The compound interval $I(h)$

has one sub-interval, two back nodes and two exit nodes. The sub-interval $I(h_1)$ of $I(h)$ is a simple interval which has two back nodes and two exit nodes.

An interval may contain arbitrarily many sub-intervals. An *outer-most interval* is an interval that is not a sub-interval of any other interval. The nesting level of a graph G is the maximum nesting level of the sub-intervals of the outer-most interval of G. An *inner-most or simple interval* is an interval that does not contain any subinterval inside. A *compound interval* is an interval that contains sub-intervals.



(a) A General Compound Interval

(b) DOM Tree for (a)

Figure 32: A general compound interval and its dominator tree

Figure 32 shows a more general compound interval. The interval $I(h1)$ is sub-interval of $I(h)$ since $h1$ is dominated by h and each exit node of $I(h1)$ has paths to exits and/or back nodes of $I(h)$. For similar reasons, both $I(h2)$ and $I(h3)$ are

sub-intervals of $I(h)$.

For a given control flow graph G, all intervals can be found using back edges in the dominator tree of G. The set $H(G)$ is all *header* nodes of G. A header node $h \in H(G)$ has at least one back edge and it is the only *Entry* of the interval $I(h)$. The *Immediate header* of v in G is the *header* of the inner-most interval that contains v. For a given interval $I(h)$, we define two simple paths, *head-to-back* and *head-to-exit* paths.

Definition 5.1

A *head-to-back* path of an interval $I(h)$ is a *simple path* from h to a $bn \in BN(I(h))$. The set of all head-to-back paths of an interval $I(h)$ is denoted by $HBP(I(h))$. The set $HBP(I(h))$ can be found as in Equation V.22.

□

$$HBP(I(h)) = \bigcup_{bn \in BN(I(h))} \sigma(h, bn) \tag{V.22}$$

Definition 5.2

A *head-to-exit* path of an interval $I(h)$ is a *simple path* from h to an $xn \in XN(I(h))$. The set of all head-to-exit paths of an interval $I(h)$ is denoted by $HXP(I(h))$. The set $HXP(I(h))$ can be found as in Equation V.23.

□

$$HXP(h) = \bigcup_{xn \in XN(I(h))} \sigma(h, xn) \tag{V.23}$$

Consider a compound interval $I(h_j)$ with a sub-interval $I(h_i)$. A $sc \in SC(I(h_j))$ may or may not include nodes in $I(h_i)$. If it does, it must include both h_i and an $xn \in XN(I(h_i))$, i.e., a *head-to-exit* path of $I(h_i)$.

Example 5.3

Consider the control flow graph G shown in Figure 16 again. The dominator tree of G is given in Figure 18. The only *back edge* in G is $\langle f, b \rangle$. Thus, there is only one simple interval, $I(b)$. The node e is the only *exit* and f is the only *back* node of $I(b)$. Three head-to-back paths in $I(b)$ are $\langle b, c, f \rangle$, $\langle b, c, e, f \rangle$ and $\langle b, d, e, f \rangle$. Two head-to-exit paths in $I(b)$ are $\langle b, c, e \rangle$ and $\langle b, d, e \rangle$.

□

Example 5.4

Consider the control flow graph G shown in Figure 28. The dominator tree of the graph is shown in Figure 33. Three back edges in G are $\langle i, d \rangle$, $\langle j, e \rangle$, and $\langle h, e \rangle$. Since $\langle j, e \rangle$, and $\langle h, e \rangle$ have the same head, they together define an interval, $I(e)$. The edge $\langle i, d \rangle$ defines a compound interval $I(d)$ that has $I(e)$ as sub-interval.

In $I(e)$, there are two *back nodes* (h and j) and two *exit nodes*(f and h). Three *head-to-back* paths in $I(e)$ are $\langle e, f, h \rangle$, $\langle e, g, h \rangle$, and $\langle e, g, j \rangle$. Three *head-to-exit* paths in $I(e)$ are $\langle e, f \rangle$, $\langle e, f, h \rangle$, and $\langle e, g, h \rangle$.

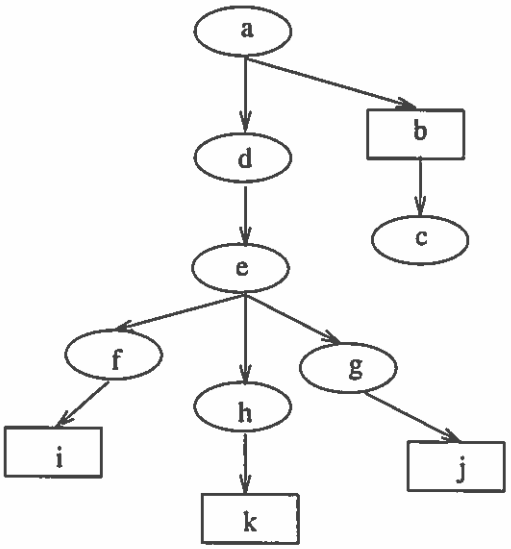


Figure 33: The dominator tree of the G shown in Figure 28.

In $I(d)$, there is one *back node*(i) and one *exit node*(h). The only *head-to-back* paths is $\langle d, e, f, i\rangle$ and two *head-to-exit* paths are $\langle d, e, f, h\rangle$ and $\langle d, e, g, h\rangle$). Note that the edge $\langle h, k\rangle$ is exit edge for both $I(e)$ and $I(d)$.

□

Reduction of Control Flow Graph

Many program analysis and optimization algorithms are designed for *reducible* control flow graphs, which occur quite often in practice as stated in [22]. If a graph is irreducible, the graph can be converted into a reducible one via a node splitting transformation as discussed in [1, 4]. In our discussion, we assume the control flow graph is reducible.

We present an algorithm which reduces a reducible graph into a single node by finding and reducing intervals. For our timing analysis, a slightly modified control flow graph G_M is used. We use two transformations to reduce the modified graph G_M.

By repeated application of these transformations, a reducible graph G_M is reduced into a single node.

Modification of Control Flow Graph

The modified control flow graph $G_M = \langle G, TE \rangle$, where TE is the set of all technical edges. A technical edge $te \in TE$ is an edge from an $xn \in XN(G)$ to the *Entry*. By adding technical edges, we make every complete simple path *repeatable*. The justification of this is that the program represented by G may be a part of a larger program so the main program may call this unit multiple times. In this case, the program implicitly has a loop outside the program. Another advantage of having these technical edges is that it simplifies the reducibility test of G by removing the necessity of T_3 transformation discussed in [18].

Transformations

We now describe the two transformations, T_1 and T_2 used to reduce the modified control flow graph G_M. The transformation T_2 is applied until there exists a unique $h \in V - \{v\}$ such that $\langle h, v \rangle \in E$, at which point T_1 is applied.

Definition 5.3

Let G_M be a modified control flow graph. If for some $v \in V$ there exists an edge $e \in E$ such that $e = \langle v, v \rangle$ and there exists a unique $h \in V - \{v\}$ such that $\langle h, v \rangle \in E$ then the transformation T_1 is given by

$$T_1(G_M, \langle v, v \rangle) = G_M - \{\langle v, v \rangle\}).$$

□

The transformation T_1 removes a self looping edge. Figure 34 shows a graphical representation of T_1 transformation.

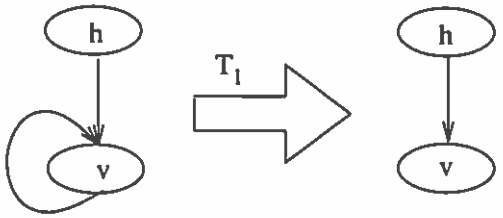


Figure 34: Graphical representation of T_1 transformation

Definition 5.4

Let G_M be a modified control flow graph. If for a node $v \in N - \{Entry\}$ there exists a unique $h \in N - \{v\}$ such that $\langle h, v\rangle$ is in E and there exist any $e = \langle v, w\rangle$ in E, where $v \neq w$, the transformation T_2 is given by

$$T_2(G_M, h, v, w) = (N', E', Entry),$$

where if v has no immediate successors other than w, then $N' = N - \{v\}$, $E' = E \cup \{\langle h, w\rangle\} - \{\langle h, v\rangle, \langle v, w\rangle\}$ and otherwise $N' = N$, $E' = E \cup \{\langle h, w\rangle\} - \{\langle v, w\rangle\}$. □

The transformation T_2 creates another graph, which is $(N', E', Entry)$. We say both node v and edge $\langle v, w\rangle$ are consumed by $T_2(h, v, w)$. Figure 35 shows a graphical representation of the T_2 transformation.

These two transformations are the same as T_1' and T_2' in [18] and T_1 and T_2 in [1]. Using these two transformations, we present an algorithm that reduces a reducible graph G_M into a single node.

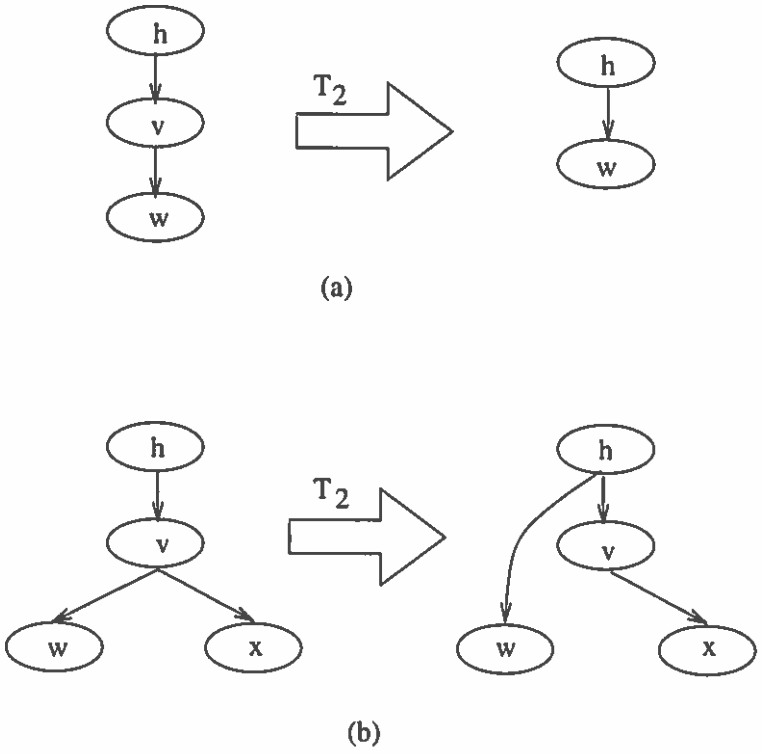


Figure 35: Graphical representation of T_2 transformation

Interval Algorithm

The algorithm shown in Figure 36 is a portion of the algorithm in [18]. In Figure 36, the transformation T_3 has been removed since the modified control flow graph G_M becomes an interval by itself through the addition of technical edges over G. The complexity of our algorithm is the same as the algorithm in [18] since T_3 is a constant factor.

The process of reduction provides hierarchy for the intervals since the reduction process starts from the inner-most and moves to the outer-most intervals.

Characteristics of Reducible Graphs

As mentioned, there are a number of quite different definitions of an interval [48]. Despite the differences in the definition of intervals, the reducibility of G remains the same [22]. Reducible graphs have many desirable properties that help in program analysis and optimization. The properties of reducible control flow graphs has been studied by Hecht and Ullman [22]. Two of them are given in Lemma 5.1 and Lemma 5.2.

Lemma 5.1

A graph G_M is reducible if and only if its DAG is unique. □

Proof

See Theorem 5 in [22]. □

Lemma 5.2

A graph G_M is reducible if and only if for each simple cycle $sc \in SC(G_M)$

```
Algorithm: ReduceInterval
N: nodes in G_M
E: edges in G_M

T1(v is node of self looping edge)
{   E = E - {(v,v)} }
T2(h,v,w: nodes of edges (h,v) and (v,w) in E)
{   E = E ∪ {(h,w)} - {(v,w)}
    if (v has no immediate successor in G) {
       N = N - {v}; E = E - {(h,v)}
    }
}
Reduce(I: set of nodes, h: node in I)
{   while (exists ⟨h,v⟩,⟨v,w⟩ ∈ E with v ∈ I − {h}, w ∈ I, v ≠ w,
       such that if ⟨u,v⟩ ∈ E then either u = h or u = v) {
       choose any such (v,w)
       if ((v,v) ∈ E) {
          T1(v)
       }
       T2(h,v,w)
    }
}
main()
{   T = {u | ⟨v,u⟩ is a dom tree back edge in G_M}
    while (T is not empty) {
          h = a node in T not dominating any other node in T
          I = { v ∈ N | h dominates v and there is a path p from v
             to h such that all nodes on p are dominated by h }
          Reduce(I,h)
          T = T - {h}
       }
}
```

Figure 36: An interval algorithm

there exists an entry node h of sc which dominates all other nodes in the sc. □

Proof

See Theorem 7 in [22]. □

Theorem 5.2

If G_M is reducible,

$$SC(G_M) = \bigcup_{h \in H(G_M)} SC(I(h)), \tag{V.24}$$

where $SC(I(h))$ is the set of all simple cycles in $I(h)$ and $H(G_M)$ is the set of all interval headers in G_M. □

Proof

By Lemma 5.2, the set $H(G_M)$ is uniquely determined, where $h \in H(G_M)$ is an element of a simple cycle $sc \in SC(G_M)$ and dominates all nodes in such simple cycles. Thus, the set $SC(G_M)$ is the union of all simple cycles for all intervals in G_M. □

Theorem 5.3

If G_M is reducible,

$$MSRP(G_M) = \bigcup_{h \in H(G_M)} MSRP(I(h)), \tag{V.25}$$

where $MSRP(I(h))$ is the set all *maximum simple repeatable paths* in $I(h)$. □

Proof

Immediate from Theorem 5.2 and the definition of *maximum simple repeatable path* for a simple cycle. □

Timing Invariance Analysis Using Intervals

We now present a method to perform timing analysis using intervals.

Lemma 5.3

Let $I(h)$ be a simple interval. The set $MSRP(I(h))$ can be found as in Equation V.26.

$$MSRP(I(h)) = HBP(I(h)) \tag{V.26}$$

□

Proof

Immediate from Lemma 5.2 and by the definition of an interval. All $sc \in SC(I(h))$ are of the form $h \xrightarrow{*} bn \rightarrow h$, where $bn \in BN(I(h))$. □

Lemma 5.4

A simple interval, $I(h)$, is executable with timing invariance if

$$(\forall hbp \in HBP(I(h)))[STD(hbp) \leq 0]. \tag{V.27}$$

□

Proof

Immediate from Lemma 5.3 and Theorem 5.1. □

Lemma 5.5

A compound interval $I(h)$ is executable with timing invariance if (1) $(\forall hbp \in HBP(I(h)))[STD(hbp) \leq 0]$ and (2) all its sub-intervals are executable with timing invariance. □

Proof

Immediate from Theorem 5.1 and 5.3. Here, a sub-interval can be either simple or compound. □

Theorem 5.4

The graph G_M is executable with timing invariance if the outer-most interval of G_M is executable with timing invariance □

Proof

The outer-most interval of G_M is either a simple or compound interval.

□

The algorithm shown in Figure 36 reduces from the inner-most intervals to the outer-most ones. In the following examples, we show how to use the reduction algorithm of Figure 36 in testing timing invariance of modified control flow graphs.

Example 5.5

Figure 37 shows the modified control flow graph G_M of the control flow graph shown in Figure 21.

There are two intervals, $I(a)$ and $I(b)$, in G_M. The algorithm starts with $I(b)$, the inner-most interval. The execution times required on the source

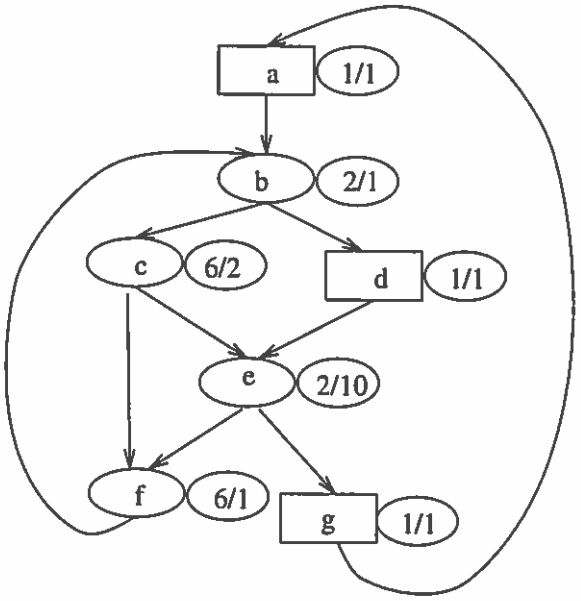


Figure 37: The modified control flow graph G_M of G given in Figure 16.

and target machines for each $hbp \in HBP(I(b))$ are $EST(\langle b, c, f \rangle) = (14, 4)$, $EST(\langle b, c, e, f \rangle) = (16, 14)$ and $EST(\langle b, d, e, f \rangle) = (11, 13)$.

Since there exists a $hbp_i \in HBP(I(b))$ which does not meet its time deadline, the interval $I(b)$ is not executable with timing invariance. The algorithm must stop at $I(b)$.

□

Example 5.6

Figure 38 shows the modified control flow graph G_M of the control flow graph shown in Figure 28. There are three intervals $I(a)$, $I(d)$ and $I(e)$ in the graph.

The algorithm starts with $I(e)$.

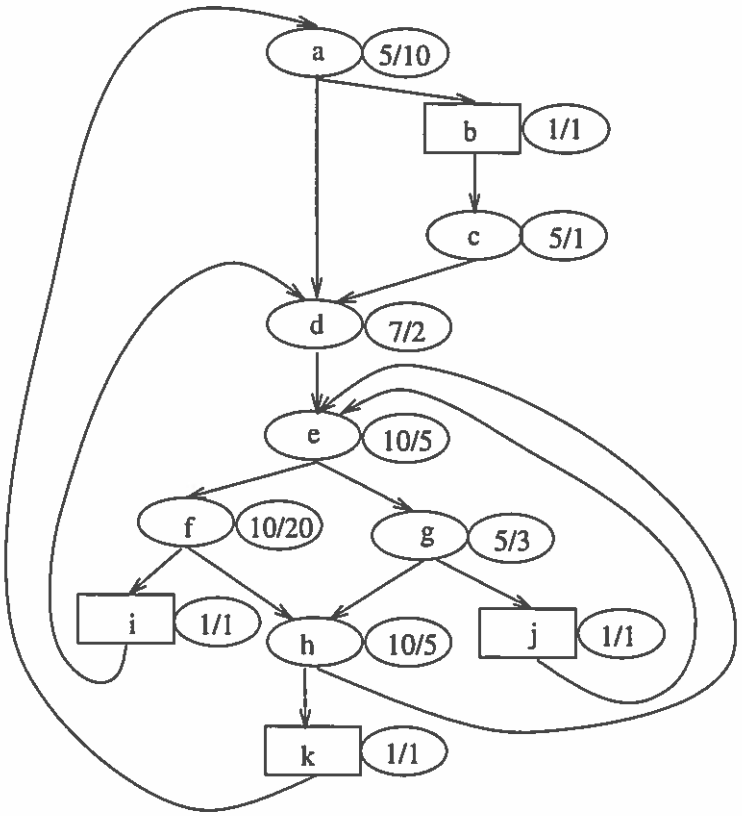


Figure 38: The modified control flow graph G_M of G given in Figure 28.

The execution times required on the source and target machines for each $hbp \in HBP(I(e))$ are $EST(\langle e, f, h\rangle) = (30, 30)$, $EST(\langle e, g, h\rangle) = (25, 13)$ and $EST(\langle e, g, j\rangle) = (16, 9)$. Thus, the interval $I(d)$ is executable with timing invariance.

Next, the algorithm reduces $I(d)$. This interval is also executable with timing invariance since the only *head-to-back* path $\langle d, e, f, i\rangle$ meets its time deadline and the only sub-interval $I(e)$ is executable with timing invariance. The head-to-back path $\langle d, e, f, i\rangle$ includes a *head-to-exit* path of the sub-interval $I(e)$.

The last stage of the algorithm examines $I(a)$.

The node k is the only *back node* in the interval. Four *head-to-back* paths in $I(a)$ are $\langle a, d, e, f, h, k\rangle$, $\langle a, d, e, g, h, k\rangle$, $\langle a, b, c, d, e, f, h, k\rangle$ and $\langle a, b, c, d, e, g, h, k\rangle$. The execution times required on the source and target machines for each $hbp \in HBP(I(e))$ are $EST(\langle a, d, e, f, h, k\rangle) = (43, 43)$, $EST(\langle a, d, e, g, h, k\rangle) = (38, 26)$, $EST(\langle a, b, c, d, e, f, h, k\rangle) = (49, 45)$ and $EST(\langle a, b, c, d, e, g, h, k\rangle) = (44, 28)$. The interval $I(a)$ is executable with timing invariance since all head-to-back paths meet their time deadline and the only sub-interval $I(d)$ is executable with timing invariance.

Thus, the modified control flow graph G_M is executable with timing invariance. □

The problem here is how to find the set $HBP(I(h))$ for each interval $I(h)$ efficiently. One can find the set $HBP(I(h))$ using super-blocking techniques discussed in [9]. However, it can be very expensive by having many copies of the same nodes in

the interval. In the next section, we show how to use a data-flow approach to solve this problem.

Data-Flow Approach

A data-flow analysis technique is a compile-time method used to reason about the run-time flow of data in a program. Data-flow information is essential for program optimization [1, 30]. The systematic study of data-flow analysis techniques begins with Allen and Cocke [2, 3, 4], although various data-flow analysis methods were in use. Since the Allen and Cocke papers, numerous research efforts have been devoted to finding and solving data-flow problems [32, 20, 42, 23, 28, 29, 48]. Kennedy's survey paper [30] gives a list of data-flow problems and optimizing transformations that use data-flow information. We use a data-flow analysis technique to find timing information for *head-to-back* and/or *head-to-exit* paths in intervals.

For the timing invariance test it is not necessary to find all simple cycles in the graph. Lemma 5.4 and 5.5 can be re-stated as Theorem 5.5.

Theorem 5.5

An interval, simple or compound, $I(h)$ is executable with timing invariance if and only if all sub-intervals are executable with timing invariance and

$$hbpstd_{max}(I(h)) \leq 0,$$

where

$$hbpstd_{max}(I(h)) = \max_{hbp \in HBP(I(h))} STD(hbp).$$

□

Proof

If $hbpstd_{max}(I(h)) \leq 0$ then $(\forall hbp \in HBP(I(h)))[STD(hbp) \leq 0]$. □

Theorem 5.6

The condition $hbpstd_{max}(I(h)) \leq 0$ is satisfied if and only if

$$(\forall bn \in BN(I(h)))[hbnstd_{max}(h, bn) \leq 0],$$

where

$$hbnstd_{max}(h, bn) = STD(MaxEST(\sigma(h, bn))).$$

□

Proof

Immediate from Lemma 5.3. □

We use a data-flow approach to find $hbnstd_{max}(h, bn)$ for each interval, where $bn \in BN(I(h))$.

Data-flow equations

Solving data-flow problems involves the computation of information about the flow of data along execution paths in the control flow graph. Data-flow analysis sets up and solves a set of equations that relates information at various points in a program. Here, a point in the program is a spot in the control flow graph between two nodes. For each node in the control flow graph, some basic attributes are defined that can be determined unambiguously from an analysis of the problem. Then, inherited and synthesized attributes are defined in a set of data-flow equations to be solved.

The data-flow equations for our timing analysis can be stated as follows:

$$In(v) = \begin{cases} (0,0), & v = header, \\ MSTD(v), & otherwise, \end{cases} \tag{V.28}$$

$$Out(v) = \begin{cases} LT(v), & v = header, \\ In(v) + LT(v), & otherwise, \end{cases} \tag{V.29}$$

where

- $In(v)$ is the inherited attribute for the node v,
- $Out(v)$ is the synthesized attribute for the node v,
- $LT(v)$ is the basic attribute for the node v (equal to $EST(v)$) and
- The function $MSTD(v)$ returns an $Out(p)$ such that

$$STD(Out(p)) = \max_{p \in PRED(v)} STD(Out(p)).$$

In the above data-flow equations, we associate information with nodes. One also can associate the information with edges instead of with nodes. This allows information to propagate between intervals more easily. If e is an edge $\langle v, w \rangle$, then let $eloc(e)$ be $LT(v)$. The data flow equations now are as follows:

$$In(v).hvmaxest = \begin{cases} (0,0), & v = header, \\ MAXEST(v), & otherwise, \end{cases} \tag{V.30}$$

$$ehvmaxest(\langle v,w\rangle) = \begin{cases} eloc(\langle v,w\rangle), & v = header, \\ MA(\langle v,w\rangle), & otherwise, \end{cases} \tag{V.31}$$

where

- The function $MAXEST(v)$ returns the $ehvmaxest(e)$ such that

$$MaxSTD(h,v) = \max_{e \in InEdge(v)} STD(ehvmaxest(e))$$

- The set $InEdge(v)$ is the set of all edges of the form $\langle w,v\rangle$ and
- $MA(\langle v,w\rangle) = In(v).hvmaxest + eloc(\langle v,w\rangle)$

Using these data-flow equations and the interval algorithm, we develop an algorithm which finds $hbnstd_{max}(h,bn)$ for all intervals in G_M, where $bn \in BN(I(h))$.

Algorithm

We modify the interval algorithm shown in Figure 36 so that it can test timing invariance for each interval. We use the data-flow equations, defined above, to find timing required for paths from the header node.

The algorithm starts with one of the inner-most intervals, which is a simple interval by definition, in the graph. The algorithm stops as soon as it finds an interval that is not executable with timing invariance. A compound interval is tested if all sub-intervals are tested as executable with timing invariance. If the outer-most interval is executable with timing invariance, the graph G_M is executable with timing invariance.

We start our modification with the `main()` function. The `main()` function

initializes $In(h).hvmaxest$ and $ehvmaxest(\langle h, w\rangle)$, where h is the header node of the interval under testing (reducing). Also, $In(s).hvmaxest$ for all $s \in SUCC(h)$ are initialized. Figure 39 shows the modified `main()` function.

```
main()
{
(1)  T = {u | ⟨v,u⟩ is a dom tree back edge in G};
(2)  while (T is not empty) {
(3)        h = a node in T not dominating any other node in T;
(4)        I = { v ∈ N | h dominates v and there is a path p from v
                 to h such that all nodes on p are dominated by h };
(5)        In(h).hvmaxest = (0,0);
(6)        forall (s ∈ SUCC(h)) {
(7)            ehvmaxest(⟨h,s⟩) = eloc(⟨h,s⟩);
(8)            In(s).hvmaxest = ehvmaxest(⟨h,s⟩);
           }
(9)        Mark h visited ;
(10)       Reduce(I,h);
(11)       T = T − {h};
      }
(12) msg(''timing invariant'');
}
```

Figure 39: Algorithm for timing analysis

The function *Reduce*() is modified so that an interval is reduced if all $hbp \in HBP(I(h))$ meet their time deadline. The modified version of *Reduce*() is shown in Figure 40. In *Reduce*(), the header node h consumes all nodes and edges in the interval, except *exit nodes*, by calling $T_2()$. As soon as it finds a *hbp* that does not meet its time deadline, the algorithm stops. If all $hbp \in HBP(I(h))$ meet their time deadline, the interval is reduced.

Once reduced, after the `while()` loop, these nodes need to be initialized. Line 14

- 15 in *Reduce()* propagates timing information of *head-to-exit* to the outer interval.

```
Reduce(I: set of nodes, h: node in I)
{
(1)  while (exists ⟨h,v⟩,⟨v,w⟩ ∈ E with v ∈ I − {h}, w ∈ I, v ≠ w,
        such that if ⟨u,v⟩ ∈ E then either u = h or u = v) {
(2)     choose any such (v,w)
(3)     if (⟨v,v⟩ ∈ E) {
(4)        T1(v);
        }
(5)     T2(h,v,w);
     }
(6)  Clear h visited;
(7)  In(h).hvmaxest = (0,0);
(8)  forall (xn ∈ XN(I(h))) {
(9)     if (xn ≠ h) {
(10)       Clear xn visited;
(11)       In(xn).hvmaxest = (0,0);
(12)       ehvmaxest(⟨h,xn⟩) = (0,0);
(13)       eloc(⟨h,xn⟩) = (0,0);
        }
(14)    forall(ss ∈ SUCC(xn)) {
(15)       eloc(⟨xn,ss⟩) = ehvmaxest(⟨xn,ss⟩);
(16)       ehvmaxest(⟨xn,ss⟩) = (0,0);
        }
     }
}
```

Figure 40: Algorithm for modified *Reduce*()

The transformation $T_2()$ consumes nodes and edges in the interval. When a node v is visited, at which time $T_2(h, v, SUCC(v))$ is called, timing information of v is propagated to all successors of v by 1 - 3 in T_2. When $bn \in BN(I(h))$ is visited, at which $T_2(h, v, h)$ is called, the $hbnstd_{max}$ for the back edge bn is found. If $hbnstd_{max} \leq 0$, all paths from h to bn meet their time deadline. Two functions used

in $T_2()$ are defined as follows:

- `InMaxEST`(est_1, est_2) returns

$$\begin{cases} est_1, & if\ STD(est_2) < STD(est_1)\ or\ est_2 = (null, null), \\ est_2, & otherwise. \end{cases}$$

- `MPlus`$((a_s, a_t), (b_s, b_t))$ returns $(a_s + b_s, a_t + b_t)$.

```
T2( h,v,w:  nodes of edges ⟨h,v⟩ and ⟨v,w⟩ in E)
{
(1)  if (v not visited) {
(2)     forall (s ∈ SUCC(v)) {
(3)        ehvmaxest(⟨v,s⟩) = MPlus(In(v).hvmaxest, eloc(⟨v,s⟩));
        }
(4)     Mark v visited ;
     }
(5)  In(w).hvmaxest = InMaxEST(ehvmaxest(⟨v,w⟩), In(w).hvmaxest);
(6)  if (h = w) {
(7)     if (0 < STD(In(w).hvmaxest)) {
(8)        Msg(''timing error:  In(w).hvmaxest'');
(9)        Exit();
        }
     }
(10) E = E ∪ {⟨h,w⟩};
(11) ehvmaxest(⟨h,w⟩) = In(w).hvmaxest;
(12) E = E − {⟨v,w⟩};
(13) if (v has no immediate successor in G) {
(14)    N = N − {v};
(15)    E = E − {⟨h,v⟩};
     }
}
```

Figure 41: Algorithm for $T_2()$ transformation

Consider a node v in $I(h)$ with out-degree of m. To delete v, the node has to be consumed m times. If the node v is not deleted after the `while()` loop in `Reduce()`, the node v is consumed less than m times. This happened when some out-edges of v are not in $I(h)$. Such a node v is an *exit node* by definition.

Lemma 5.6

The first time a node v is consumed, the first time $T_2(h, v, SUCC(v)$ is called, $ehvmaxest(\langle v, SUCC(v)\rangle) = MaxEST(\sigma(h, v))$. □

Proof

At line 5 in `main()`, $In(h).hvmaxest$ is initialized to $(0, 0)$.

At line 6 - 8 in `main()`, $(\forall s \in SUCC(h))$ $[ehvmaxest(\langle h, s\rangle) = EST(v)$ and $In(s).hvmaxest = EST(v)]$. By Line 1 in `Reduce()`, a node v is consumed only when there is no u such that $u \rightarrow v$ and u is neither h nor v. Thus, to consume v, all $InEdges(v)$ in $I(h) - \{h\}$ must be consumed. If all $InEdges(v)$ in $I(h) - \{h\}$ are consumed, $In(v).hvmaxest$ has $MaxEST(\sigma(h, v))$ because of Line 5 in $T_2()$.

The first time $T_2(h, v, SUCC(v))$ is called, where v is consumed the first time, $ehvmaxest(\langle v, SUCC(v)\rangle)$ has $MaxEST(\sigma(h, v))$ by line 2 - 3 in $T_2()$.

□

Lemma 5.7

The function `Reduce`(I, h) shown in Figure 40 tests if a simple interval $I(h)$ is executable with timing invariance. □

Proof

Since an interval is reducible graph by itself, all nodes in $I(h)$ are visited at least once. Since all nodes in $I(h)$ are visited at least once, all $bn \in BN(I(h))$ are also visited at least once. If a $bn \in BN(I(h))$ is visited, $hbpstd_{max}$ for the bn is found by Lemma 5.6. By Line 7 - 9 in $T_2()$, the condition $hbnstd_{max}(h, bn) \leq 0$ is examined. □

Theorem 5.7

The algorithm shown in Figure 39 tests if a reducible G_M is executable with timing invariance. □

Proof

The modified graph G_M is an interval by itself. By line 3 in the `main()`, the inner-most interval, a simple interval by definition, is reduced first. Once an interval is reduced, only header and exit nodes remains. At this stage, *ehvmaxest* of $OutEdges(xn)$ contains $MaxEST(\rho(h, xn))$ by line 14 - 15 in `Reduce()`. The $MaxEST(\rho(h, xn))$ and $MaxEST(\rho(h, bn))$ of the outer interval can be found by having $MaxEST(\rho(h, xn))$ of all sub-intervals. No compound interval is reduced until all its sub-intervals are reduced by line 3 in `main()`. A compound interval becomes a simple interval when all its sub-intervals are reduced. When a compound interval is reduced all of its sub-intervals have been tested as executable with timing invariance. Thus, a compound interval is executable with timing invariance if all of its *head-to-back paths* meet their time deadline. By Lemma 5.7, this is tested by `Reduce()`. □

Example 5.7

Consider the example shown in Figure 37 again.

The outer-most interval $I(a)$ has a sub-interval $I(b)$. The algorithm starts with the inner-most interval, $I(b)$. In the following, we show snapshots of the process that reduces the inner-most interval $I(b)$.

In $main()$:

T = { a,b }; $h = b$

I = { b,c,d,e,f }; $In(b).hvmaxest = (0,0)$

$ehvmaxest(\langle b,c \rangle) = (2,1)$; $In(c).hvmaxest = (2,1)$

$ehvmaxest(\langle b,d \rangle) = (2,1)$; $In(b).hvmaxest = (2,1)$

$Reduce(I,b)$

Assume the consumption order in $Reduce(I,b)$ is as follows.

$T_2(b,c,f)$: c not deleted

$T_2(b,c,e)$: c deleted

$T_2(b,d,e)$: d deleted

$T_2(b,e,f)$: e deleted

$T_2(b,f,b)$: `timing error: In(b).hvmaxest`

The order of consumption shown above is not required except that $T_2(b,e,f)$ must be after $T_2(b,d,e)$ and $T_2(b,c,e)$; $T_2(b,f,b)$ must be last, since f can not be consumed until all predecessors of it are consumed. Now, we show the snapshot of $T_2(b,c,f)$ and $T_2(b,f,b)$.

$T_2(b, d, e)$

$ehvmaxest(\langle c, f\rangle) = MPlus(In(c).hvmaxest, eloc(\langle c, f\rangle)) = (8, 3)$

$ehvmaxest(\langle c, e\rangle) = MPlus(In(c).hvmaxest, eloc(\langle c, e\rangle)) = (8, 3)$

$In(f).hvmaxest = (8, 3)$

$E = E \cup \{\langle b, f\rangle)\}$

$ehvmaxest(\langle b, f\rangle) = In(f).hvmaxest = (8, 3)$

$T_2(b, f, b)$

$ehvmaxest(\langle f, b\rangle) = (11, 13).$

$In(f).hvmaxest = (11, 13).$

`timing error: In(b).hvmaxest`

At the function call $T_2(b, f, b)$, $ehvmaxest(\langle f, b\rangle)$ has $MaxEST(\sigma(b, f))$ and $b \in BN(I(h))$. Thus, the only $hbnstd_{max}(h, bn)$, i.e., $hbnstd_{max}(b, f)$, is found which is $STD(MaxEST(\sigma(b, f))$. Since $0 < hbnstd_{max}(b, f) = 2$, the interval $I(b)$ is not executable with timing invariance. The following table shows the content of *In(v).hvmaxest* and $ehvmaxest(\langle v, SUCC(v)\rangle)$ during the reduction process of the interval $I(b)$.

Node	$In(v).hvmaxest$	$ehvmaxest(\langle v, SUCC(v)\rangle)$
b	(0,0)	(2,1)
c	(2,1)	(8,3)(8,3)
d	(2,1)	(3,2)
e	(8,3)(3,2)	(5,12)(5,12)
f	(8,3)(5,12)	(11,13)

□

Example 5.8

Consider the example shown in Figure 28 again. The outer-most interval $I(a)$ has a sub-interval $I(d)$ which also has a sub-interval $I(e)$. The algorithm starts with the most-inner interval $I(e)$.

In $main()$:

$T = \{a, d, e\}$; $h = e$

$I = \{e, f, g, h, j\}$; $In(e).hvmaxest = (0, 0)$

$ehvmaxest(\langle e, f \rangle) = (10, 5)$; $In(f).hvmaxest = (10, 5)$

$ehvmaxest(\langle e, g \rangle) = (10, 5)$; $In(g).hvmaxest = (10, 5)$

$Reduce(I, e)$

Assume the consumption order in $Reduce()$ is as follows.

$T_2(e, f, h)$: f not deleted

$T_2(e, g, h)$: g not deleted

$T_2(e, h, e)$: h not deleted

$T_2(e, g, j)$: g deleted

$T_2(e, j, e)$: j deleted

Again, this is not the only computation order. Timing is checked when both $T_2(e, h, e)$ and $T_2(e, j, e)$ are called.

Since both $hbnstd_{max}(e, h)$ and $hbnstd_{max}(e, j)$ meet their time deadline, the interval $I(e)$ is executable with timing invariance. The following table shows the content of $In(v).hvmaxest$ and $ehvmaxest(\langle v, SUCC(v) \rangle)$ during the reduction process of the interval $I(e)$.

$Node$	$In(v).hvmaxest$	$ehvmaxest(\langle v, SUCC(v)\rangle)$
e	(0,0)	(10,5)
f	(10,5)	(20,25)
g	(10,5)	(15,8)
h	((20,25)(15,8))	(30,30)
j	(15,8)	(16,9)

The graph after this step is shown in Figure 42.

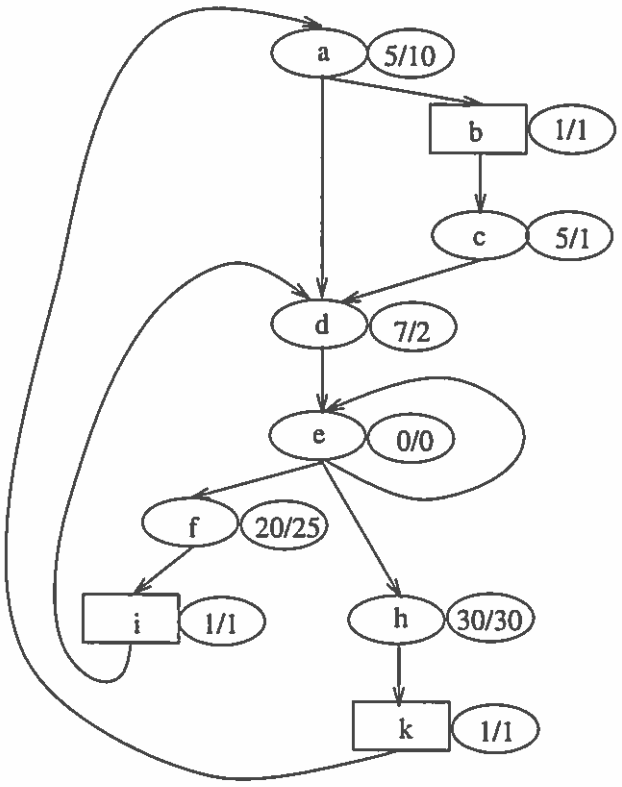


Figure 42: After $I(e)$ is reduced

Since the interval $I(e)$ is executable with timing invariance, the algorithm can examine the outer interval of it. Before it starts the next interval, it

cleans up the header node and exit nodes of $I(e)$. Each exit node $xn \in XN(I(h))$ has $MaxEST(e, xn)$, which is saved on $eloc(\langle xn, SUCC(xn)\rangle)$. Now, the algorithm starts to reduce the interval $I(d)$.

In *main*():

T = { a,d}; $h = d$

I = { d,e,f,i }; $In(d).hvmaxest = (0, 0)$

$ehvmaxest(\langle d, e\rangle) = (7, 2)$; $In(e).hvmaxest = (7, 2)$

$Reduce(I, d)$

The reduction order in *Reduce*() for the interval $I(d)$ is as follows.

$T_1(e)$: self-looping edge $\langle e, e\rangle$ deleted

$T_2(d, e, f)$: e not deleted

$T_2(d, f, i)$: f deleted

$T_2(d, i, d)$: i deleted

Timing is checked when $T_2(d, i, d)$ is called. Since $hbnstd_{max}(d, i)$ meet its time deadline, the interval $I(d)$ is executable with timing invariance. The node e is not deleted since it becomes the only exit node for the interval $I(d)$. The following table shows the content of $In(v).hvmaxest$ and $ehvmaxest(\langle v, SUCC(v)\rangle)$ during the reduction process of the interval $I(d)$.

$Node$	$In(v).hvmaxest$	$ehvmaxest(\langle v, SUCC(v)\rangle)$
d	(0,0)	(7,2)
e	(7,2)	(7,2)
f	(7,2)	(27,27)
i	(27,27)	(28,28)

The graph after this stage is shown in Figure 43.

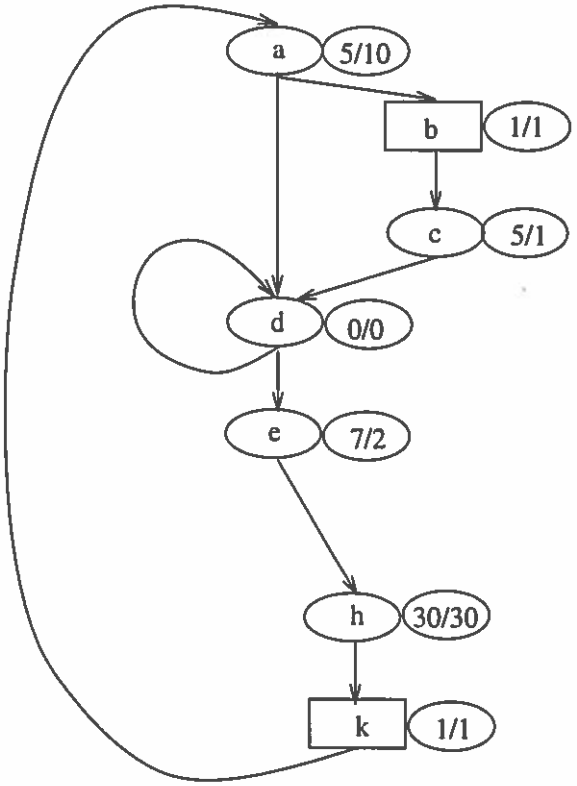


Figure 43: After $I(d)$ is reduced

Since the interval $I(d)$ is executable with timing invariance, the algorithm can examine the outer interval of it. Again, it cleans up the header node and exit nodes of $I(d)$. Each exit node e has $MaxEST(d, e)$, which is

saved on $eloc(\langle e, h \rangle)$. Now, the algorithm start to reduce the interval $I(a)$. The rest of the reduction process for the graph G_M is straightforward. The resulting graph is a single node a with a self-looping edge. The graph G_M is executable with timing invariance since the outer-most interval $I(a)$ is also executable with timing invariance.

□

Timing Equivalence Analysis

If a target program is executable with timing invariance, we test if it is executable with timing equivalence. It is easy to see that insertion of a synchronization on each i/o node provides timing equivalence if for all $v \in V$, $ExecTime(vM_T) \leq ExecTime(v, M_S)$. The precise conditions for timing equivalence are stated in Theorem 5.8.

Theorem 5.8

Let $PIOIO(G_M)$ be the set of all simple paths from an i/o node in $IO(G_M)$ to another i/o node in $IO(G_M)$ that does not contains any other i/o node in it. The graph G_M is executable with timing equivalence if and only if it is executable with timing invariance and

$$(\forall pioio \in PIOIO(G_M))[ExecTime(pioio, M_T) \leq ExecTime(pioio, M_S)] \quad \text{(V.32)}$$

□

Proof

(If Part)

For any $cp \in CP(G_M)$, no two i/o events cpe_i and cpe_{i+1} exist on the cp such that $TD(cpte_i, cpte_{i+1}) \leq TD(cpse_i, cpse_{i+1})$ if Equation V.32 is true.

(Only If Part)

There exist at least one $cp \in CP(G_M)$ that contains two i/o events such that $TD(cpte_i, cpte_{i+1}) > TD(cpse_i, cpse_{i+1})$ if Equation V.32 is not true.

Thus, the theorem is obvious by Definition 4.4 and Theorem 4.2.

□

If the graph G_M is executable with timing equivalence, it is easy to see that the insertion of a synchronization on each i/o node in the graph provides timing equivalence. If G_M is timing equivalent, both $\Delta(G_M)$ and $\Psi(G_M)$ are zero. The set $PIOIO(G_M)$ is easy to find using a data-flow approach.

Summary

This chapter derived necessary and sufficient conditions to provide timing equivalence and invariance for target programs. We used a program decomposition method, called interval analysis, to test the conditions. The problem of timing equivalence testing is converted into a data-flow analysis problem which uses intervals. When a control flow graph is executable with timing equivalence or invariance, we must convert it into an invariant or equivalent one. Chapter VI presents methods to provide timing invariance or equivalence for the graph that is executable with timing invariance or equivalence.

CHAPTER VI

ENFORCEMENT OF TIMING EQUIVALENCE

Chapter V presented a method to test if a target program represented by G_M is executable with timing equivalence or invariance. If it is executable with timing equivalence, inserting synchronization on i/o node provides timing equivalence. If it is executable with timing invariance but not with timing equivalence, the timing sensitivity of the target program depends on how and where the target program is synchronized. This chapter presents two different synchronization schemes, global and local, to improve timing sensitivity of the target program.

Chapter VII presents a method to find both timing sensitivities, $\Delta(G_M)$ and $\Psi(G_M)$. Chapter VIII presents methods to optimize these timing sensitivities.

Problem Statement

We are to convert the target program into a timing invariant one if it is executable with timing invariance. If all simple cycles are synchronized, the graph is timing invariant by Theorem 5.1. Synchronization of all simple cycles can be achieved by inserting synchronizations on all back edges in the graph G_M, since all simple cycles in a reducible graph must include back edges. If the graph G_M is executable with timing invariance, but not with timing equivalence, the quality of the target program depends on timing sensitivity it has, i.e., $\Delta(G_M)$ and $\Psi(G_M)$. These timing sensitivities depend on where and how the graph is synchronized. This chapter discusses

how to add synchronization to improve timing sensitivities of the target program. So far, we have used a global synchronization scheme which uses a single global clock for each system, source and target. In this chapter, we first discuss problems with the global synchronization scheme and then present another synchronization scheme called "local synchronization," which uses multiple clocks for each system. With the local synchronization scheme, it is easier to find the timing sensitivities of the graph and the timing sensitivities are reduced.

Global Synchronization

The synchronization method discussed in Chapter IV is a global synchronization scheme which synchronizes the global target clock to the global source clock at synchronization points (only on back edges, so far). Even though this global synchronization scheme provides timing invariance whenever possible, there are a number of problems with it. With the global synchronization scheme, synchronization of an edge in the graph may invalidate other synchronizations as seen in Example 5.2. Thus, finding timing sensitivities of the graph G_M with this scheme requires consideration of all paths from all possible synchronization points that may reach each an i/o node. This is not a desirable characteristic for timing sensitivity analysis. This also makes the timing sensitivities of G_M worse than necessary as we will see in Example 6.1.

Example 6.1

Consider the example given in Figure 44. Four back edges are synchronized with the global clock. The set $IO(G_M)$ is $\{d, j, l\}$.

Consider a following complete path $cp \in CP(G_M)$.

$$\langle ia_1, ib_1, if_1, ih_1, ii_1, ij_1, ih_2, ii_2, ij_2, ih_3, ii_3, ij_3, ih_4, ik_1, il_1,$$

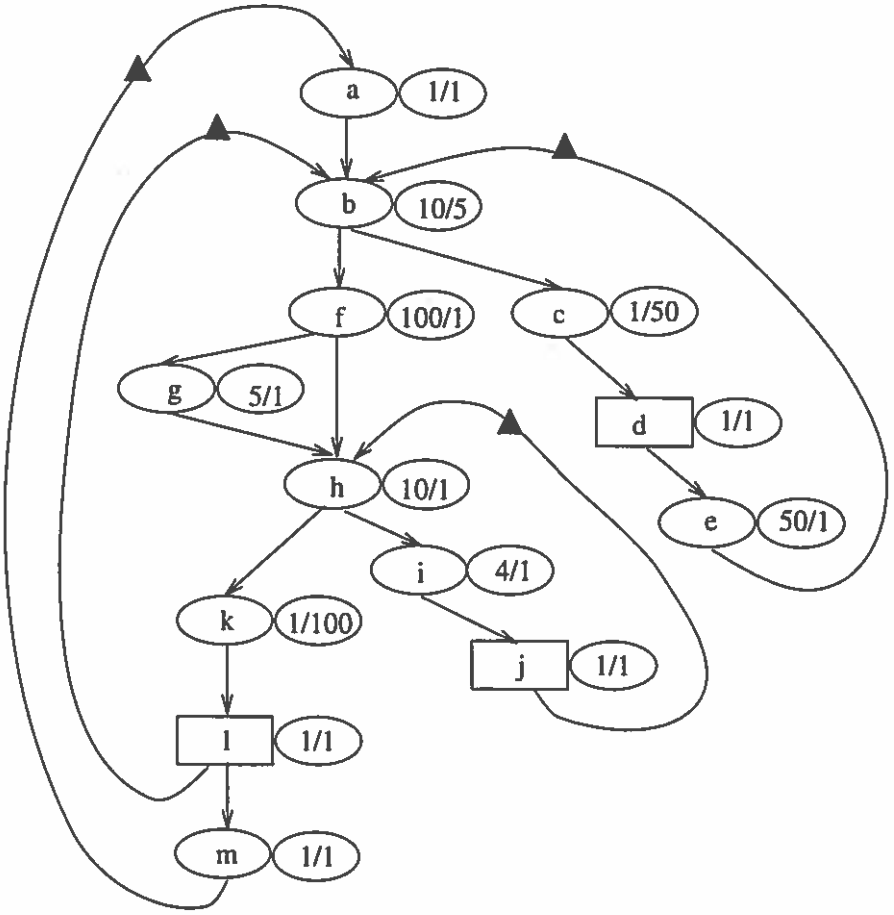


Figure 44: An example of control flow graph for global synchronization

$$ib_2, ic_1, id_1, ie_1, ib_3, if_2, ih_5, ik_2, il_2, im_1\rangle$$

In the *cp* given above, there are three instances of j (j_1, j_2 and j_3), two instances of node l (l_1 and l_2) and one instance of node d (d_1). The global source and target times at each instance of i/o nodes ($ASTime$ and $ATTime$ at iio_i) are given in the following table.

$INST(io)$	$AST(iio_i)$	$ATT(iio_i)$
ij_1	125	9
ij_2	140	12
ij_3	155	15
il_1	167	117
id_1	179	173
il_2	351	282

Synchronizations are performed after instances of nodes ij_1, ij_2, ij_3, il_1, ie_1, and im_1. These synchronizations are to match $ATTime$ to $ASTime$ at 126, 141, 156, 168, 230, and 353.

With the global synchronization scheme, the following will happen when the program is running on the target machine: since no synchronization was performed before the i/o event ij_1, ij_1 occurs at 9. After ij_1, the program execution is delayed for 116 from 10, which makes $ATTime = 126$ (synchronized). The ij_2 event occurs at 128 and the program execution is delayed for 12 from 129, which makes $ATTime = 141$ (synchronized). The ij_3 event occurs at 143 and the program execution is delayed for 12 from 144, which makes $ATTime = 156$ (synchronized). After delaying for

101, the il_1 event is executed at $ATTime = 257$. At 258, synchronization is invalid, since $ASTime < ATTime$. The id_i event is executed at 313 and the synchronization at 316 is also invalid. The i/o event il_2 is executed at 422 and the last synchronization is invalid also. Figure 45 shows the execution time of the given *cp* on source and target machine with global synchronization scheme.

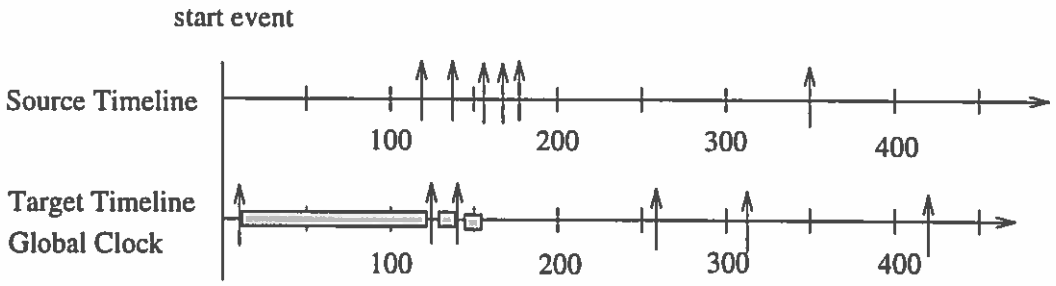


Figure 45: With global synchronization scheme

The absolute timing sensitivity for the *cp*, $\Delta(cp)$, with global synchronization is 134, since the $\Delta(INST(io))$ for each $INST(io)$ is computed as in the following table.

$INST(io)$	$AST(iio_i)$	$ATT(iio_i)$	$\Delta(iio_i)$
ij_1	125	9	$\mid -116 \mid$
ij_2	140	128	$\mid -12 \mid$
ij_3	155	143	$\mid -12 \mid$
il_1	167	257	$\mid 90 \mid$
id_1	179	313	$\mid 134 \mid$
il_2	351	422	$\mid 71 \mid$

□

In this example, $\Delta(G_M)$ is unnecessary big, since the synchronization on $ij_1 \rightarrow ih_2$ delayed the program execution too long (over commitment). This over commitment invalidate the synchronization on $il_1 \rightarrow ib_2$. When this happened, all other synchronizations after this event become invalid until $ATTime < ASTime$. This over commitment also hinders analysis and optimization of timing sensitivities. In the next section, we present another synchronization scheme which reduces the value of timing sensitivities and make timing sensitivity analysis easier.

Local Synchronization

To resolve problems with the global synchronization scheme, a local synchronization scheme is proposed. Our local synchronization scheme uses multiple local clocks instead of the single global clock for each system. In the local synchronization scheme, each interval $I(h)$ maintains both source and target clocks, $LASTime(h)$ and $LATTime(h)$. Synchronization of any instance of edges in $I(h)$ is performed with its own local clocks. In the global sense, this local synchronization is not a synchronization, but is a *delay* which removes the timing error introduced by the interval.

The source and target clocks for each interval $I(h)$ are obtained by inserting the following two statements for every v in $I(h) - \{h\}$.

$$LATTime(h) \Leftarrow LATTime(h) + ExecTime(v, M_T)$$
$$LASTime(h) \Leftarrow LASTime(h) + ExecTime(v, M_S)$$

At each header node $h \in H(G_M)$, its own local clocks are initialized to zero, i.e., $LATTime(h) = LASTime(h) = 0$.

For a given *cp*, the function $LAST(iv_i, h)$ returns $LASTime(h)$ at the iv_i and the function $LATT(iv_i, h)$ returns $LATTime(h)$ at the iv_i.

Consider a compound interval $I(hi)$ with a sub-interval $I(hj)$. Every node v in $I(hj)$ also maintains local clocks of $I(hi)$. When the flow of control enters an interval $I(hj)$ from an outer interval $I(hi)$ (through hj), the local clock of the interval $I(hi)$ are kept running and local clocks of $I(hj)$ are initialized to zero, i.e., $LATTime(hj) = LASTime(hj) = 0$. When there is a local synchronization of $I(hj)$, the program execution is delayed for $LASTime(hj) - LATTime(hj)$ at the synchronization point. When the flow of control exits from interval $I(hj)$ to $I(hi)$ (through a $xn \in XN(I(hj))$ of course), local clocks of $I(hj)$ are no longer available.

Since every interval in G_M is a sub-interval of the outer-most interval, the local clock of the outer-most interval is the same as the global clock. The local synchronization function we use is shown in Figure 46.

```
LSync(⟨v, w⟩, h);
{
      if (LATT(iw_i, h) ≤ LAST(iw_i, h)) {
          Delay(LAST(iw_i, h) − LATT(iw_i, h))
      }
}
```

Figure 46: Algorithm for local synchronization

Here, an instance of local synchronization is *valid* if $0 \leq LAST(iw_i, h) - LATT(iw_i, h)$.

Lemma 6.1

If all back edges in G_M are locally synchronized and if the graph G_M is executable with timing invariance, all instances of these synchronizations on all $cp \in CP(G_M)$ are *valid.* □

Proof

Obvious, since all $msrp \in MSRP(G_M)$ meet their time deadline and inserting local synchronization on all back edges removes only local timing error introduced by the cycle defined by the back edge. Notice that no synchronization is inserted on head-to-exit paths, which is a part of the outer-interval's head-to-back paths. □

Theorem 6.1

The local synchronization scheme provides timing invariance to the G_M if it is executable with timing invariance. □

Proof

For all sub-intervals, any timing difference caused by the sub-interval is removed by local synchronization of all back edges. Thus, all timing differences in an interval are local, i.e., timing difference from the header node to the current node. Since all local timing differences are determined statically, and since the graph G_M itself is an interval, the timing difference for the graph G_M is determined statically. If the timing difference is determined statically, the graph G_M is executable with timing invariance by definition. □

If every simple cycle in G_M takes the same amount of time on both source and target machines, the graph is timing invariant. If all back edges in the graph

are locally synchronized and if the graph is executable with timing invariance, all simple cycles takes the same amount of time on source and target machine. By Lemma 5.2, any simple cycle in a reducible graph includes an interval header node. Thus, synchronization of all back edges in the graph enforces timing invariance on a graph that is executable with timing invariance.

Example 6.2

Consider Example 6.1 with the local synchronization scheme. In the interval $I(h)$, there is only one back edge, $\langle j, h\rangle$. Thus, the local synchronizations of $I(h)$, $\texttt{LSync}(\langle j, h\rangle, h)$ are performed after ij_1, ij_2, and ij_3 on the given *cp*.

There are two back edges in $I(b)$, i.e, $\texttt{LSync}(\langle l, b\rangle, b)$ and $\texttt{LSync}(\langle e, b\rangle, b)$. The local synchronizations of $I(b)$ are performed after il_1 and id_1 on the given *cp*. There is one technical back edge $m \rightarrow a$ which forms the outermost interval $I(a)$.

Every *cp* starts with *Entry* and every *cp* has only one instance of *Entry* which generates the *Program Start* event. At the event ia_1, the local clocks of $I(a)$ are started by initializing them to zero, i.e., $LASTime(a) = LATTime(a) = 0$. When the flow enters to b, local clocks of the interval $I(b)$ are initialized. When the flow enters to h, before ih_1 on the *cp*, $LAST(b) = 110$, $LATTime(b) = 6$, $LASTime(a) = 111$ and $LATTime(a) = 7$. Also, local clocks of $I(h)$ are started by initializing them, i.e., $LASTime(h) = LATTime(h) = 0$. Since no synchronization occurs before ij_1, the ij_1 event occurs at $ATTime(a) = 9$. After ij_1, the pro-

gram execution is delayed for 12 ($LAST(ig_2, h) - LATT(ig_2, h) = 15$ - 3) from $ATTime(a) = 10$, which makes $ATTime(a) = 22$. When the flow reaches to the header node h, local clocks of $I(h)$ are initialized to zero. The ij_2 event occurs at $ATTime(a) = 24$ and is delayed for 12 (again, $LAST(ig_2, h) - LATT(ig_2, h) = 15$ - 3) from $ATTime(a) = 25$, which makes $ATTime(a) = 37$. The ij_3 event occurs at $ATTime(a) = 39$ and is delayed for 12 from $ATTime(a) = 39$ which makes $ATTime(a) = 52$. When the flow reaches to the header node h, local clocks of $I(h)$ are initialized to zero, again. When the flow exits from interval $I(h)$ to $I(b)$ through h, local clocks of $I(h)$ are not available anymore. At this point $LAST(ik_1, a) = 166$, $LATT(ik_1, a) = 53$, $LAST(ik_1, b) = 165$ and $LATT(ik_1, b) = 52$.

The il_1 event is executed at $ATTime(a) = 153$ which is 100 from 53. The program execution is delayed for 14 ($LAST(ia_2, b) - LATT(ia_2, b) = 167$ - 153) from $ATTime(a) = 154$ which makes $ATTime(a) = 168$. When the flow reaches to the header node b, local clocks of $I(b)$ are re-initialized to zero, i.e., $LASTime(b) = LATTime(b) = 0$.

The id_1 event occurs at $ATTime(a) = 223$ and after ie_1 the program execution is delayed for 5 ($LAST(ib_3, b) - LATT(ib_3, b) = 62$-57) from $ATTime(a) = 225$, which makes $ASTime(a) = ATTime(a) = 230$.

When the flow reaches to the header node b, local clocks of $I(b)$ are initialized to zero, again.

The il_2 event occurs at $ASTime(a) = 351$ and $ATTime(a) = 337$. The given *cp* finishes at $ASTime(a) = 353$ and $ATTime(a) = 339$, but delayed

for 14 for the synchronization of the technical edge.

Figure 47 depicts the local synchronization scheme for the given *cp*.

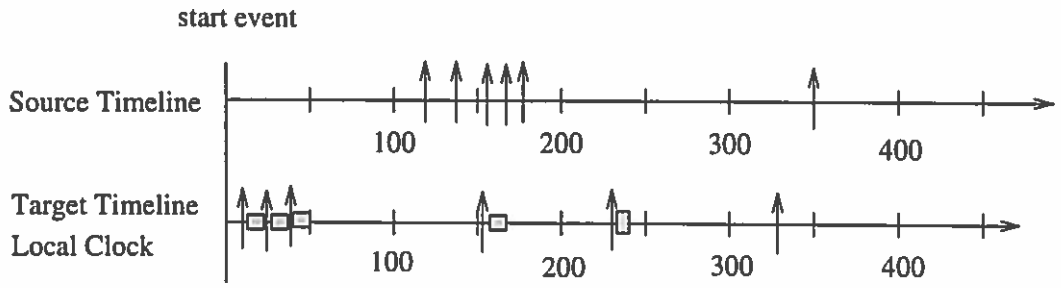


Figure 47: With local synchronization scheme

The absolute timing sensitivity of the given *cp*, $\Delta(cp)$, with local synchronization is 116. □

By using the local synchronization scheme, the absolute timing sensitivity is reduced from 134 to 116. However, there are problems with the pure local synchronization scheme also. By having only a local view on synchronization, some possible synchronizations which improve timing sensitivity may be omitted.

Example 6.3

Consider the example given in Figure 48 which is the same as in Figure 44, but with different $EST(k)$. Consider the same *cp* as in Example 6.1 which is:

$$\langle ia_1, ib_1, if_1, ih_1, ii_1, ij_1, ih_2, ii_2, ij_2, ih_3, ii_3, ij_3, ih_4, ik_1, il_1,$$
$$ib_2, ic_1, id_1, ie_1, ib_3, if_2, ih_5, ik_2, il_2, im_1\rangle$$

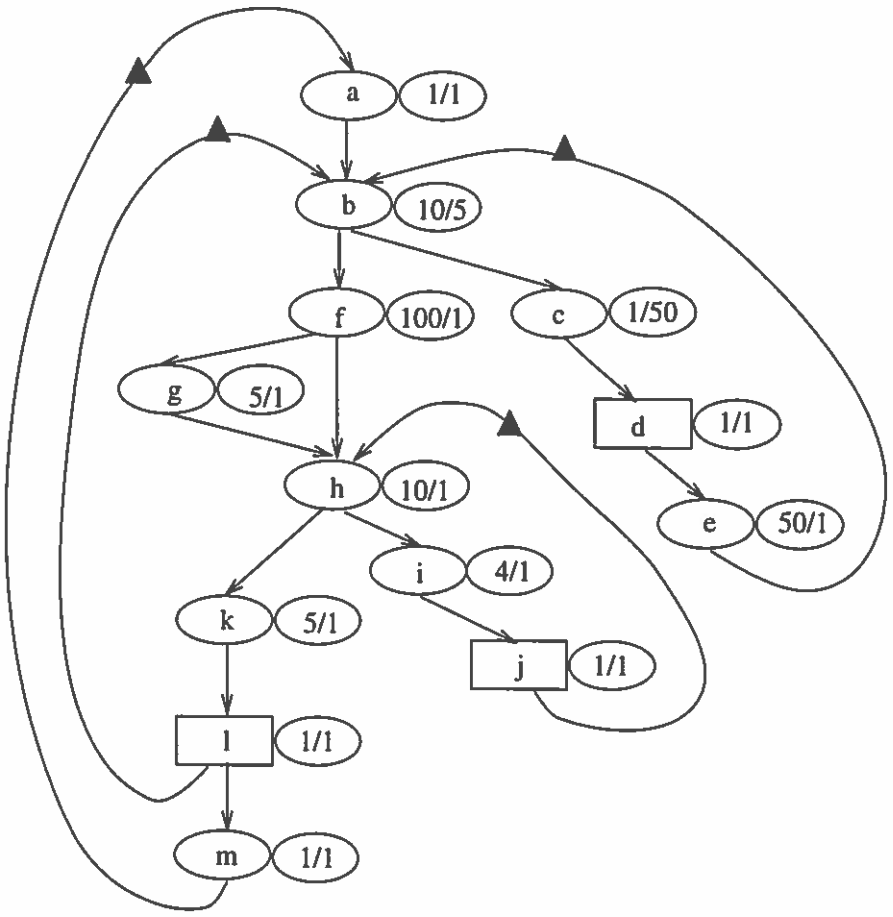


Figure 48: An example of control flow graph for local synchronization scheme

With no synchronization, $ASTime$ and $ATTime$ of $io \in IO(G_M)$ are as shown in the following table.

$INST(io)$	$AST(iio_i)$	$ATT(iio_i)$
ij_1	125	9
ij_2	140	12
ij_3	155	15
il_1	171	18
id_1	183	74
il_2	359	84

With the global synchronization scheme, the instances of $io \in IO(G_M)$ occur at 9, 128, 143, 158, 227 and 242. The absolute timing sensitivity of the given cp, $\Delta(cp)$, with global synchronization scheme is 117. Figure 49 depicts the execution of cp with global synchronization.

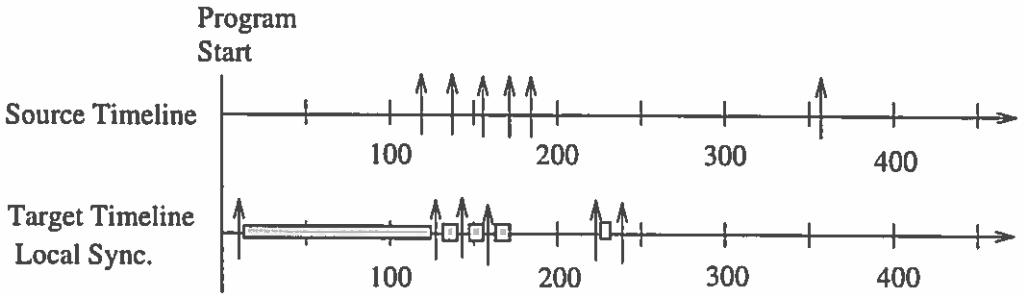


Figure 49: With global synchronization scheme

With the local synchronization scheme, instances of $io \in IO(G_M)$ occur at 9, 24, 39, 54, 227 and 241. The absolute timing sensitivity of the given cp, $\Delta(cp)$, with local synchronization is also 117. Figure 50 depicts the execution cp with local synchronization.

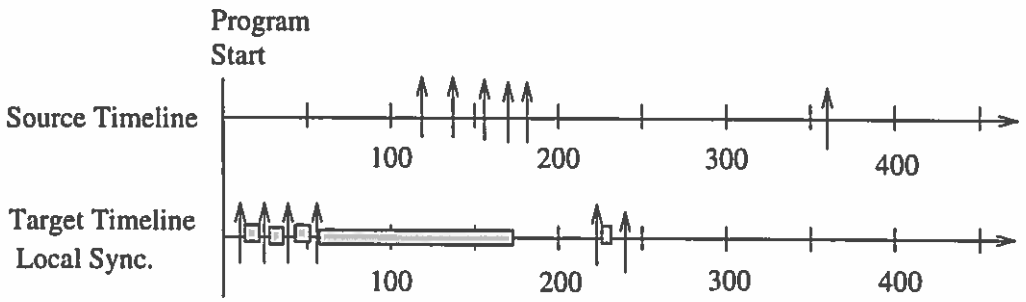


Figure 50: With local synchronization scheme

□

In this example, local synchronization does not improve timing sensitivities of the given path *cp* over the global synchronization scheme. Here, timing sensitivities of the *cp* can be improved dramatically by inserting synchronizations on incoming edges of intervals as shown in Example 6.4.

Example 6.4

Consider the example shown in Figure 51, which is the same graph as in Figure 48, but synchronizations on all incoming edges of h.

Consider the same *cp* as in previous Example which is:

$$\langle ia_1, ib_1, if_1, ih_1, ii_1, ij_1, ih_2, ii_2, ij_2, ih_3, ii_3, ij_3, ih_4, ik_1, il_1,$$
$$ib_2, ic_1, id_1, ie_1, ib_3, if_2, ih_5, ik_2, il_2, im_1\rangle$$

Synchronizations are performed after the following instances: ia_1, if_1, ij_1, ij_2, ij_3, il_1, ie_1 and if_2. With synchronizations on all incoming edges of intervals, i/o events occurs at 113, 127, 143, 158, 226 and 345. With this scheme, $\Delta(cp)$ is 44. Figure 52 depicts the execution of *cp* with optimized local synchronization.

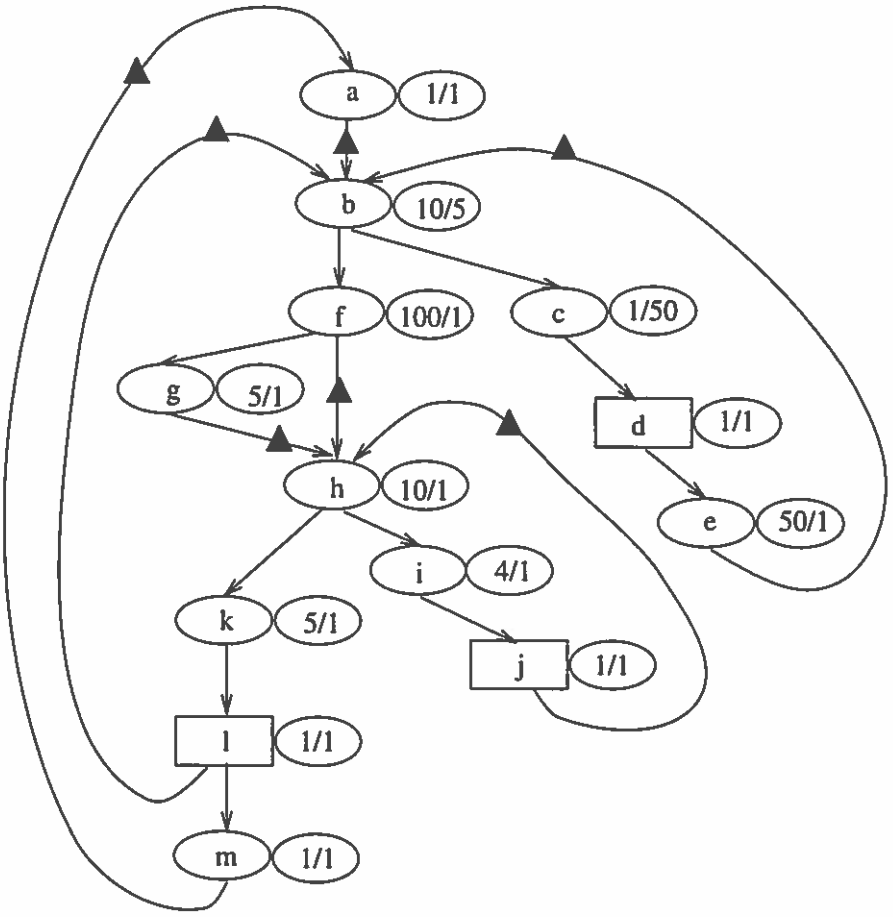


Figure 51: Adding additional synchronization to optimize local synchronization

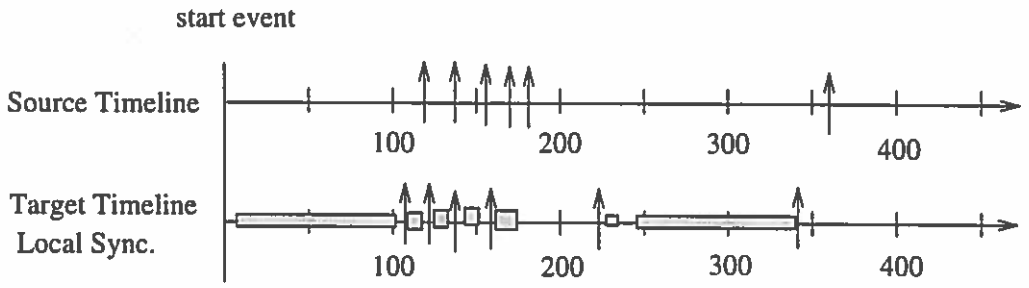


Figure 52: With optimized local synchronization scheme

□

By inserting synchronizations on all incoming edges of intervals, the absolute timing sensitivities are reduced from 117 to 44. However, inserting synchronization on these edges does not always help in providing better timing sensitivities as in Example 6.1. Chapter VIII discusses issues on optimizing timing sensitivities by inserting more synchronizations.

Summary

When a program is executable with timing equivalence or invariance, it must be enforced by a synchronization scheme to be timing equivalent or invariant. This chapter described two synchronization schemes, global and local synchronization, which provide timing invariance to a graph that is executable with timing invariance. A main advantage of using the local synchronization scheme is that it reduces timing sensitivity of the target program by avoiding over commitment that occurs in the global synchronization scheme. Another advantage is that it simplifies timing sensitivity analysis which we will discuss in the next chapter. The under commitment problem in the local synchronization scheme is solved by inserting additional local synchronizations. Chapter VIII presents timing optimization algorithms by inserting additional synchronization.

CHAPTER VII

TIMING SENSITIVITY ANALYSIS

Chapter V presented methods to test if a given target program represented by G is executable with timing equivalence or invariance. Chapter VI presented methods to enforce timing invariance or equivalence.

This chapter presents static algorithms that find two timing sensitivities of a target program defined in Chapter IV. These timing sensitivities are used to judge how closely the target program mimics the source program. We assume every back edge is synchronized with local clocks for our timing sensitivity analysis.

Methods that optimize these timing sensitivities are presented in Chapter VIII.

Problem Statement

In Chapter IV, two timing sensitivities of a target program represented by G_M, $\Delta(G_M)$ and $\Psi(G_M)$, are defined in terms of the set $CP(G_M)$. In Chapter V, the absolute timing sensitivity of an i/o node io in G_M, denoted by $\Delta(io)$, is defined in Equation V.19. To find $\Delta(io)$, all complete paths that contains at least an instance of io must be considered. Similarly, the relative timing sensitivity of io can be defined. The program represented by G_M is timing invariant if for all $io \in IO(G_M)$, $\Delta(io)$ is limited by a constant. Both absolute and relative timing sensitivities of G_M can be found when the timing sensitivities of all $io \in IO(G_M)$ are found. In this chapter, we develop methods which find these timing sensitivities for all i/o nodes.

Absolute Timing Sensitivity

To compute absolute timing sensitivity of a graph G_M, we must find the absolute timing sensitivity for each i/o node $io \in IO(G_M)$. Once the absolute timing sensitivity of each i/o node is known, the absolute timing sensitivity of G_M is the maximum of $\Delta(io)$ for all $io \in IO(G_M)$.

If an edge $\langle v, w \rangle$ is globally synchronized, $ASTime = ATTime$ at any instance of w that comes after an instance of v for any $cp \in CP(G_M)$. The node w is said to be a possible global synchronization node. If all $InEdges(w)$ are globally synchronized, w is a global synchronization node. The *Entry* is always a global synchronization node.

Suppose *psn* is a globally synchronized node and there exist paths from *psn* to an i/o node *io* not passing through another global synchronization point. Such a node *psn* is called a previous synchronization node of *io*. With the pure local synchronization scheme, the only global synchronization point for all $io \in IO(G_M)$ is the *Entry*. Figure 53 shows a control flow graph with two header nodes (h_i and h_j), one i/o node (*io*) and a global synchronization node (*psn*). The node *psn* is the only previous synchronization node of *io*. The header node h_i has one back edge and h_j has two back edges with local synchronization.

For a given previous synchronization node *psn* of *io*, $MaxSTD(psn, io)$ and $MinSTD(psn, io)$ are defined as in Equation VII.33 and Equation VII.34.

$$MaxSTD(psn, io) = \max_{\rho_i \in \rho(psn, io)} STD(\rho_i^o) \qquad \text{(VII.33)}$$

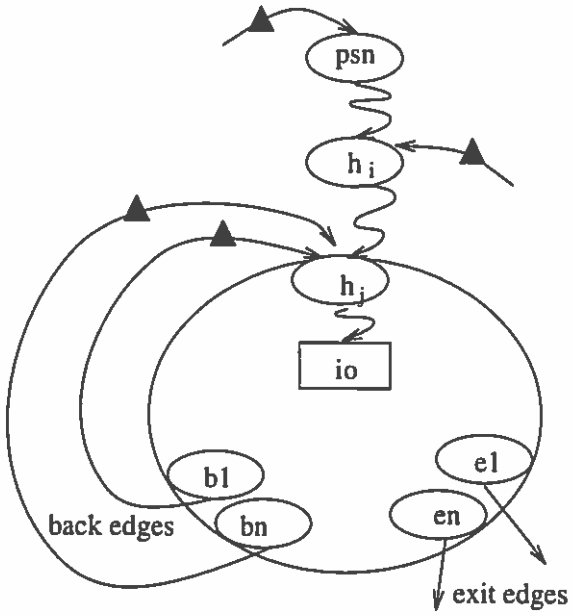


Figure 53: Previous synchronization node of an i/o node

$$MinSTD(psn, io) = \min_{\rho_i \in \rho(psn,io)} STD(\rho_i^\circ) \qquad \text{(VII.34)}$$

The absolute timing sensitivity of io from a previous synchronization node psn, denoted by $\Delta(psn, io)$, can be found as in Equation VII.35.

$$\Delta(psn, io) = Max(\mid MaxSTD(psn, io) \mid, \mid MinSTD(psn, io) \mid) \qquad \text{(VII.35)}$$

In Equation VII.33 and Equation VII.34, the set $\rho(psn, io)$ may be infinite. With the local synchronization scheme, however, it is not necessary to consider all $\rho_i \in \rho(psn, io)$ to find $\Delta(psn, io)$.

Theorem 7.1

With local synchronization,

$$\max_{\rho_i \in \rho(psn,io)} STD(\rho_i^\circ) = \max_{\sigma_i \in \sigma(psn,io)} STD(\sigma_i^\circ)$$

and

$$\min_{\rho_i \in \rho(psn,io)} STD(\rho_i^\circ) = \min_{\sigma_i \in \sigma(psn,io)} STD(\sigma_i^\circ).$$

□

Proof

With local synchronization on every back edge, $ExecTime(msrp, M_S) = ExecTime(msrp, M_T)$ for any instance of any $msrp \in MSRP(G_M)$. If any path $\rho_i \in \rho(psn, io)$ is not in $\sigma(psn, io)$, it has to have at least a $msrp$ in it by definition.

The simple cycle sc that defines $msrp$ starts from the header node in $\sigma(psn, io)$ and finishes at the header node and $ExecTime(msrp, M_S) = ExecTime(msrp, M_T)$ for all $msrp \in MSRP(G_M)$. Thus, any $msrp$ in ρ_i can be ignored in finding $STD(\rho_i)$. If all $msrp$ in ρ_i are ignored, the path ρ_i is in $\sigma(psn, io)$. □

By Theorem 7.1, $MaxSTD(psn, io)$ and $MinSTD(psn, io)$ can be obtained as in Equation VII.36 and Equation VII.37.

$$MaxSTD(psn, io) = \max_{\sigma_i \in \sigma(psn,io)} STD(\sigma_i^\circ) \qquad \text{(VII.36)}$$

$$MinSTD(psn, io) = \min_{\sigma_i \in \sigma(psn,io)} STD(\sigma_i^\circ) \qquad \text{(VII.37)}$$

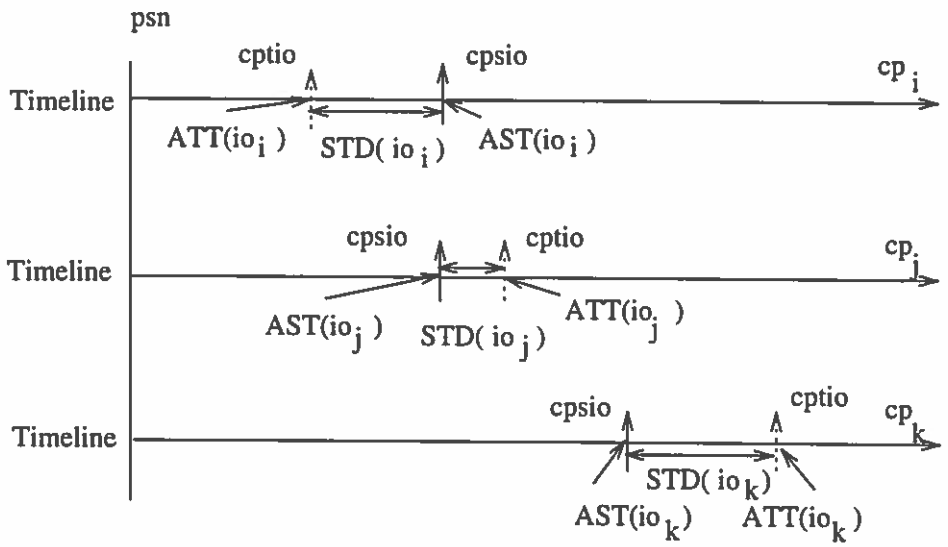


Figure 54: Simple paths from a previous synchronization node

Figure 54 shows three simple paths from psn to io. Suppose these three paths are the set of all simple paths from psn to io, i.e, $\sigma(psn, io)$. By ignoring all instances of all $msrp \in MSRP(G_M)$ that occur between an instance of psn and instances of io, all $iocp \in IOCP(io)$ can be viewed as one of these three paths.

We only need $MaxSTD(psn, io)$ and $MaxSTD(psn, io)$ to find $\Delta(psn, io)$ for the previous synchronization node psn. In the example above, $MaxSTD(psn, io)$ is $STD(io_k)$ and $MinSTD(psn, io)$ is $STD(io_i)$. Thus, $\Delta(psn, io)$ is $STD(io_k)$. If psn is the only previous synchronization node of io, the absolute timing sensitivity of io, $\Delta(io)$, is $\Delta(psn, io)$ and thus Equation VII.38.

$$\Delta(io) = Max(\mid MaxSTD(psn, io) \mid, \mid MinSTD(psn, io) \mid). \qquad \text{(VII.38)}$$

Since we only need $MaxSTD(psn, io)$ and $MinSTD(psn, io)$ to find $\Delta(psn, io)$, we use a simplified method to represent this information as in Figure 55.

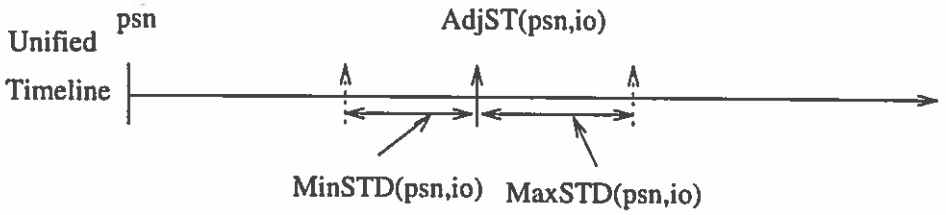


Figure 55: Adjusted source timing of io for a previous synchronization node psn

In Figure 55, three timelines are unified into one timeline and $AdjST(psn, io)$ is used to represent three source times, where the value of $AdjST(psn, io)$ itself is not significant. The values of $MaxSTD(psn, io)$ and $MinSTD(psn, io)$ are the only significant information.

We may insert synchronizations on the outer-most interval for optimization purpose as we will discuss in Chapter VIII. In this case, there may exist multiple previous synchronization nodes for a given i/o node $io \in IO(G_M)$. Let $PSE(io)$ be the set of all edges such that a $pse \in PSE(io)$ is a previous global synchronization edge of io. If an edge $\langle v, w \rangle$ is in $PSE(io)$, w is called a possible previous synchronization node of io. Let $PSN(io)$ be the set of all possible previous synchronization nodes of io. Any path from $Entry$ to io includes an edge $pse \in PSE(io)$ and there is at least one simple path from a $psn \in PSN(io)$ to io without passing through an edge $pse \in PSE(io)$. For the given set $PSN(io)$, $MaxD(io)$ and $MinD(io)$ are defined as

in Equation VII.39 and Equation VII.40.

$$MaxD(io) = \max_{psn \in PSN(io)} MaxSTD(psn, io) \tag{VII.39}$$

$$MinD(io) = \min_{psn \in PSN(io)} MinSTD(psn, io) \tag{VII.40}$$

Suppose the set $PSN(io)$ is $\{p, q, r\}$. Figure 56 shows unified timelines for each $psn \in PSN(io)$ with $MaxSTD(psn, io)$ and $MinSTD(psn, io)$. In this example, $MaxD(io) = MaxSTD(r, io)$ and $MinD(io) = MinSTD(q, io)$.

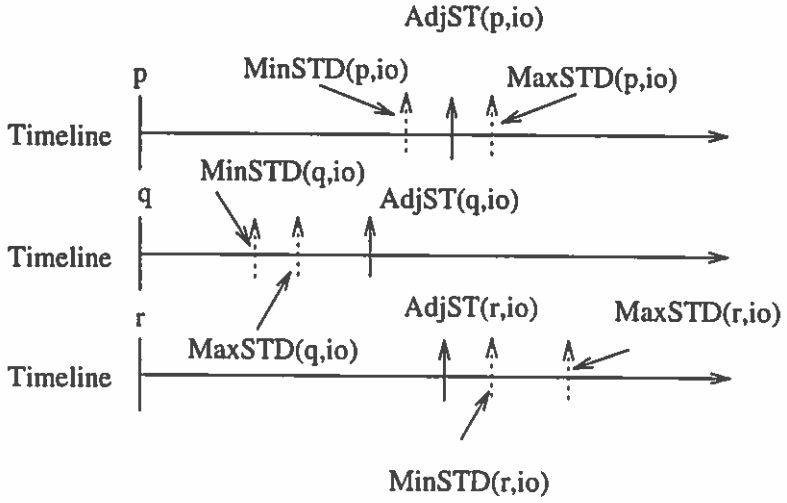


Figure 56: Multiple previous synchronization nodes

The only information we need is the maximum of $MaxSTD(psn, io)$ and minimum of $MinSTD(psn, io)$, denoted by $MaxD(io)$ and $MinD(io)$, for all $psn \in PSN(io)$. Here, another unification of timelines can be done as in Figure 57.

Once the $MaxD(io)$ and $MinD(io)$ are known, the absolute timing sensitivity of the io can be found as in Equation VII.41.

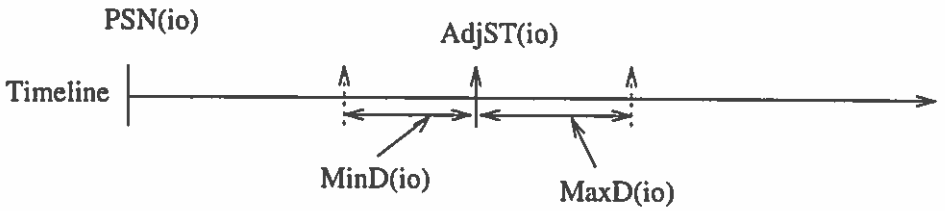


Figure 57: Unification of timelines for multiple previous synchronization nodes

$$\Delta(io) = Max(\mid MaxD(io) \mid, \mid MinD(io) \mid) \tag{VII.41}$$

When $\Delta(io)$ for all $io \in IO(G_M)$ are known, the absolute timing sensitivity of a target program represented by G_M can be found as in Equation VII.42

$$\Delta(G_M) = \max_{io \in IO(G_M)} \Delta(io) \tag{VII.42}$$

Example 7.1

Consider the example given in Figure 58 which is the same graph as in Figure 38, but with a synchronization on each back edge. We assume the pure local synchronization scheme is used. There are four i/o nodes in G_M and there is only one global synchronization node which is a. To find $\Delta(G_M)$, we have to find the $\Delta(io)$ for all $io \in IO(G_M)$.

There is only one simple path from a to b. The absolute timing sensitivity for b is 5, since $MaxD(b) = MinD(b) = 5$. There are two simple paths from a to i which are $\langle a, d, e, f, i \rangle$ and $\langle a, b, c, d, e, f, i \rangle$. The absolute timing sensitivity of i is 5 since $MaxD(i) = MaxSTD(a, i)$

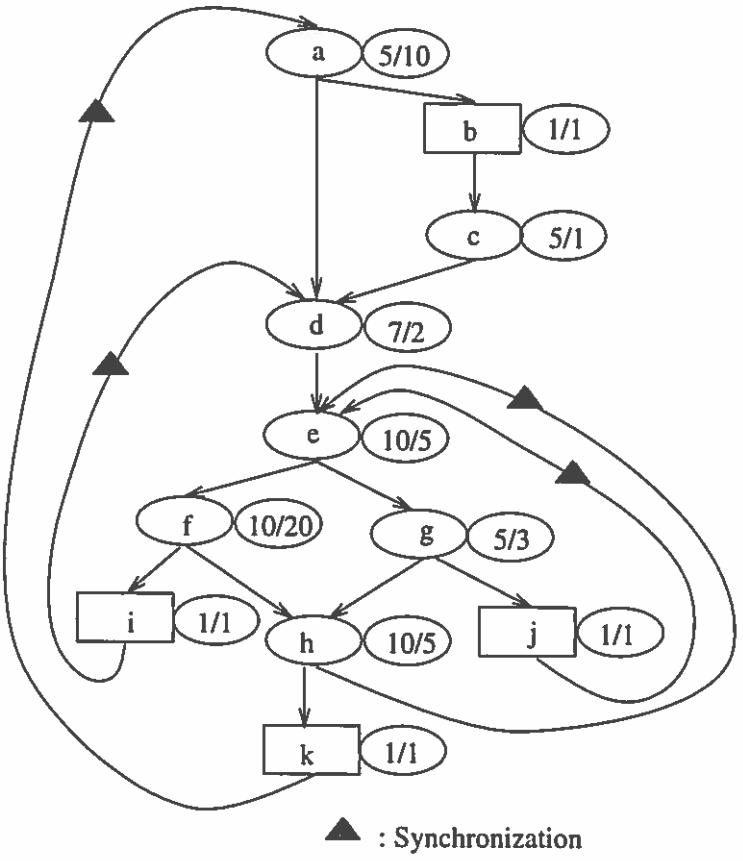


Figure 58: An example of control flow graph for timing sensitivities

$= STD(\langle a, d, e, f\rangle) = STD(32, 37) = 5$ and $MinD(i) = MinSTD(a, i)$ $= STD(\langle a, b, c, d, e, f\rangle) = STD(38, 39) = 1$. There are also two simple paths from a to j which are $\langle a, d, e, g, j\rangle$ and $\langle a, b, c, d, e, g, j\rangle$. The absolute timing sensitivity of j is 11 since $MaxD(j) = MaxSTD(a, j) = STD(\langle a, d, e, g\rangle) = STD(27, 20) =$ -7 and $MinD(j) = MinSTD(a, j) = STD(\langle a, b, c, d, e, g\rangle) = STD(33, 22) =$ -11. There are four simple paths from a to k which are $\langle a, d, e, f, h, k\rangle$, $\langle a, d, e, g, h, k\rangle$ $\langle a, b, c, d, e, f, h, k\rangle$ and $\langle a, b, c, d, e, g, h, k\rangle$. The absolute timing sensitivity of k is 16 since $MaxD(k) = MaxSTD(a, k) = STD(\langle a, d, e, f, h\rangle) = STD(42, 42) = 0$ and $MinD(k) = MinSTD(a, k) = STD(\langle a, b, c, d, e, g, h\rangle) = STD(33, 22)$ $=$ -16.

Thus, the absolute timing sensitivity of G_M, $\Delta(G_M)$, is 16. □

Relative Timing Sensitivity

The absolute timing sensitivity of an i/o node $io \in IO(G_M)$ is determined by previous synchronization points of io. However, the relative timing sensitivity of an i/o node $io \in IO(G_M)$ is determined by previous i/o nodes of io.

Let $PIO(io)$ be the set of all possible previous i/o nodes that have at least a simple path to io without passing through another i/o node. Some $pio \in PIO(io)$ is in $\sigma(Entry, io)$, but others are not in $\sigma(Entry, io)$. If a pio is not in $\sigma(Entry, io)$, every simple path from pio to io includes an edge $\langle v, w\rangle$ such that the edge is a back edge in G_M and w supports a path in $\sigma(Entry, io)$. Figure 59 shows an i/o node io and its previous i/o nodes. A $pio \in PIO(io)$ may be inside the same interval as io which reaches through one of the back edges or outside of the interval which reaches through one of the incoming edges of the interval. For each node $io \in IO(G_M)$, we already

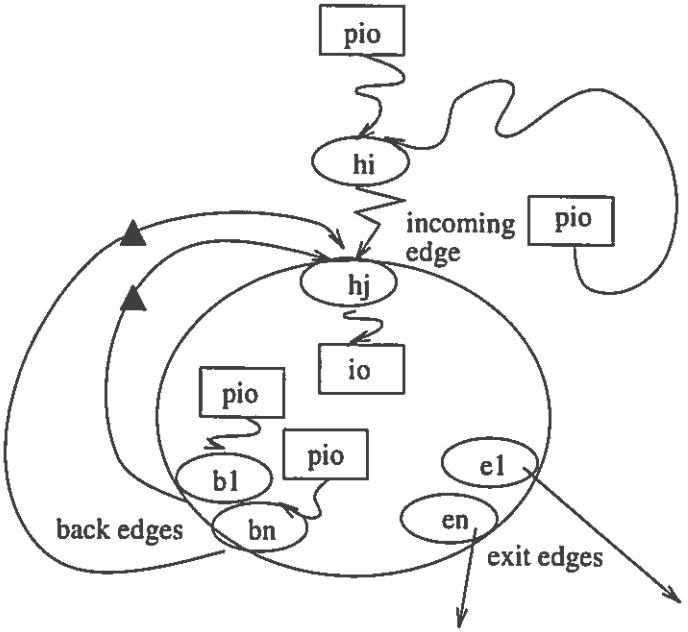


Figure 59: Previous i/o nodes for io

computed $MaxD(io)$ and $MinD(io)$ to compute the absolute timing sensitivity of io. Using $MaxD(io)$ and $MinD(io)$, the maximum relative timing sensitivity between two adjacent i/o nodes pio and io can be found as in Equation VII.43.

$$\Psi(pio, io) = Max(\mid MaxD(pio) - MinD(io) \mid, \mid MaxD(io) - MinD(pio) \mid) \quad \text{(VII.43)}$$

For the given set $PIO(io)$, we define $MaxPIO(io)$ and $MinPIO(io)$ as in Equation VII.44 and Equation VII.45.

$$MaxPIO(io) = \max_{pio \in PIO(io)} MaxD(pio) \quad \text{(VII.44)}$$

$$MinPIO(io) = \min_{pio \in PIO(io)} MinD(pio) \tag{VII.45}$$

Figure 60 shows two possible previous i/o nodes of io, pio_i and pio_j. In this figure, we can see $MaxPIO(io) = MaxD(pio_j)$ and $MinPIO(io) = MinD(pio_i)$.

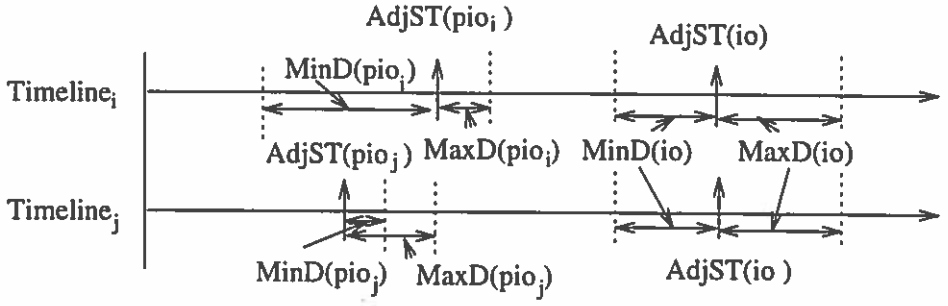


Figure 60: $MaxPIO(io)$ and $MinPIO(io)$

Again, we can unify timelines for all $pio \in PIO(io)$ as seen in Figure 61.

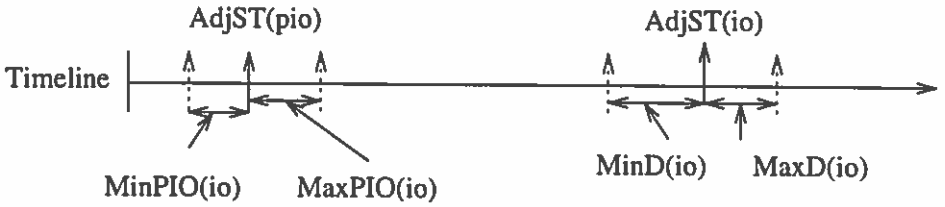


Figure 61: $MaxMin(io)$ and $MinMax(io)$

With $MaxPIO(io)$ and $MinPIO(io)$, we define $MaxMin(io)$ and $MinMax(io)$ as in Equation VII.46 and Equation VII.47.

$$MaxMin(io) = MaxPIO(io) - MinD(io) \tag{VII.46}$$

$$MinMax(io) = MaxD(io) - MinPIO(io) \tag{VII.47}$$

It is easy to see $\mid MaxMax(io) \mid$ and $\mid MinMin(io) \mid$ are smaller than either $\mid MaxMin(io) \mid$ or $\mid MinMax(io) \mid$. Once $MaxMin(io)$ and $MinMax(io)$ are computed, the relative timing sensitivity of io can be computed as in Equation VII.48.

$$\Psi(io) = Max(\mid MaxMin(io) \mid, \mid MinMax(io) \mid) \tag{VII.48}$$

Once the relative timing sensitivity of all $io \in IO(G_M)$ is computed, the relative timing sensitivity of the program G_M can be computed as in Equation VII.49.

$$\Psi(G_M) = \max_{io \in IO(G_M)} \Psi(io) \tag{VII.49}$$

Example 7.2

Consider the example given in Figure 58, again. There are four i/o nodes, $\{b, i, j, k\}$. The *Entry* nodes a is also treated as an i/o node which has always ($MaxD(io) = 0$, $MinD(io) = 0$). In the following, ($MaxD(io)$, $MinD(io)$) for each i/o nodes are given.

$$(MaxD(a), MinD(a)) = (0, 0)$$
$$(MaxD(b), MinD(b)) = (5, 5)$$
$$(MaxD(i), MinD(i)) = (5, 1)$$
$$(MaxD(j), MinD(j)) = (-7, -11)$$
$$(MaxD(k), MinD(k)) = (0, -16)$$

For each i/o node io, $MaxPIO(io)$ and $MinPIO(io)$ can be found. With $MaxPIO(io)$ and $MinPIO(io)$, $MaxMin(io)$ and $MinMax(io)$ can be also found. The following table summarizes the results.

io	$(MaxPIO(io), MinPIO(io))$	$(MaxMin(io), MinMax(io))$
b	(0,0)	(0-5,5-0)
i	(5,-11)	(5-1,5+11)
j	(5,-11)	(5+11,-7+11)
k	(5,-11)	(5+16,0+11)

The relative timing sensitivity of G_M is 21 since the relative timing sensitivity of each i/o node is computed as following;

$$\Psi(b) = Max(|\ -5\ |, |\ 5\ |) = 5,$$
$$\Psi(i) = Max(|\ 4\ |, |\ 16\ |) = 16,$$
$$\Psi(j) = Max(|\ 16\ |, |\ 4\ |) = 16 \text{ and}$$
$$\Psi(k) = Max(|\ 21\ |, |\ 11\ |) = 21.$$

□

Algorithm

To find $\Delta(io)$, the values of $MaxD(io)$ and $MinD(io)$ are needed. Since no timing difference comes from back edges, we can find $\Delta(io)$ for all i/o node $io \in IO(G_M)$ using the DAG of G_M. By Lemma 5.1, the DAG of G_M is unique. The DAG of G_M is easy to find by deleting all edges in the set $BE(G_M)$.

To find $\Psi(io)$, we also need the values of $MaxPIO(io)$ and $MinPIO(io)$. To find $MaxPIO(io)$ and $MinPIO(io)$, it is necessary to consider all possible previous

i/o nodes of io, i.e., $PIO(io)$. Since some previous i/o nodes of io are reached through back edges, we have to scan the DAG twice. At the first scan, all header nodes $h \in H(G_M)$ can find timing information, $MaxD(pio)$ and $MinD(pio)$, of any pio that reaches through back edges of $I(h)$ from the *Entry*. At the second scan, all timing information of previous i/o nodes for io can be reached to io.

We solve these two problems together in two phases. The first phase is to find $\Delta(G_M)$ by finding $MaxD(io)$ and $MinD(io)$ for all $io \in IO(G_M)$. Also, for all header nodes $h \in H(G_M)$ find the maximum of $MaxD(hio)$ and minimum of $MinD(pio)$ for all $hio \in HIO(I(h))$, where $HIO(I(h))$ is the set of all i/o nodes that reach through a back edge of $I(h)$ without passing through another i/o node from the *Entry*.

The second phase is to find $\Psi(G_M)$ by finding $MinPIO(io)$ and $MaxPIO(io)$ for all $io \in IO(G_M)$. These problems can be solved using a data-flow approach also.

Data-flow Equations

We setup data-flow equations as in the following:

$$In(v).psmaxest = \begin{cases} (0,0), & v \text{ is globally synchronized}, \\ MAXEST(v), & otherwise. \end{cases} \tag{VII.50}$$

$$In(v).psminest = \begin{cases} (0,0), & v \text{ is globally synchronized}, \\ MINEST(v), & otherwise. \end{cases} \tag{VII.51}$$

$$epsmaxest(\langle v,w\rangle) = \begin{cases} eloc(\langle v,w\rangle), & v\ is\ globally\ synchronized, \\ PMA(\langle v,w\rangle), & otherwise. \end{cases} \quad \text{(VII.52)}$$

$$epsminest(\langle v,w\rangle) = \begin{cases} eloc(\langle v,w\rangle), & v\ is\ globally\ synchronized, \\ PMI(\langle v,w\rangle), & otherwise. \end{cases} \quad \text{(VII.53)}$$

$$In(v).piomaxstd(v) = \begin{cases} 0, & v = header, \\ MaxSTD_{e\in InEdge(v)} epiomaxstd(e), & otherwise. \end{cases} \quad \text{(VII.54)}$$

$$In(v).piominstd(v) = \begin{cases} 0, & v = header, \\ MinSTD_{e\in InEdge(v)} epiomaxstd(e), & otherwise. \end{cases} \quad \text{(VII.55)}$$

$$epiomaxstd(\langle v,w\rangle) = \begin{cases} MaxD(v), & v \in IO(G_M) \\ In(v).piomaxstd, & otherwise. \end{cases} \quad \text{(VII.56)}$$

$$epiominstd(\langle v,w\rangle) = \begin{cases} MinD(v), & v \in IO(G_M) \\ In(v).piominstd, & otherwise, \end{cases} \quad \text{(VII.57)}$$

where

- $PMAXEST(v)$ returns the $epsmaxest(e)$ such that

$$MaxSTD(h,v) = max_{e \in InEdge(v)} STD(epsmaxest(e))$$

- $PMINEST(v)$ returns the $epsmaxest(e)$ such that

$$MinSTD(h,v) = min_{e \in InEdge(v)} STD(epsminest(e))$$

- $PMA(\langle v,w \rangle) = In(v).psmaxest + eloc(\langle v,w \rangle)$ and
- $PMI(\langle v,w \rangle) = In(v).psminest + eloc(\langle v,w \rangle)$.

Phase 1

The first phase of the algorithm, shown in Figure 62 and 63, is to find $MaxD(v)$ and $MinD(v)$ for all $v \in V$ using the data-flow equations shown in Equation VII.50-VII.53. The algorithm also finds $MaxD(hio)$ and $MinD(hio)$ for all $h \in H(G_M)$ using the data-flow equations shown in Equation VII.54-VII.57. The function $Phase_1()$, shown in Figure 62, is similar to the function *Reduce*() and the function $P_1T_2()$, shown in Figure 63, is similar to the function $T_2()$. The function $Phase_1()$ repeatedly calls $P_1T_2()$ to reduce the graph into a single node, after initialization of the *Entry* node and edges of it. The node *Entry* is treated as an i/o node, since it generates the event *Program Start.* Here, the calling order of $P_1T_2()$ is not fixed by the algorithm. Every call of $P_1T_2()$ consumes an edge in G_M. Thus, the complexity of this algorithm is $O(\mid E \mid)$, where $\mid E \mid$ is the number of edges in G_M.

```
Phase_1(G_M);
{
(1)  h = Entry;
(2)  In(h).psmaxest = (0,0);
(3)  In(h).psminest = (0,0);
(4)  In(h).piomaxstd = 0;
(5)  In(h).piominstd = 0;
(6)  MaxD(h) = 0;
(7)  MinD(h) = 0;
(8)  UpdateDelta(0);
(9)  forall (s ∈ SUCC(h)) {
(10)     epsmaxest(⟨h,s⟩) = eloc(⟨h,s⟩);
(11)     epsminest(⟨h,s⟩) = eloc(⟨h,s⟩);
(12)     epiomaxstd(⟨h,s⟩) = In(h).piomaxstd;
(13)     epiominstd(⟨h,s⟩) = In(h).piominstd;
(14)     In(s).psmaxest = epsmaxest(⟨h,s⟩);
(15)     In(s).psminest = epsminest(⟨h,s⟩);
(16)     In(s).piomaxstd = epiomaxstd(⟨h,s⟩);
(17)     In(s).piominstd = epiominstd(⟨h,s⟩);
     }
(18) while (there exist an edge ⟨v,w⟩ ∈ E such that
        v ∈ SUCC(h) and if ⟨u,v⟩ ∈ E then either u = h or
        u ∈ BN(G_M)) {
(10)     choose any such (v,w)
(20)     P_1T_2(h,v,w);
     }
}
```

Figure 62: Phase 1: finds $\Delta(G_M)$ and $MaxPIO(v)$ and $MinPIO(v)$ for all $v \in V$

```
P1T2( h,v,w:  nodes of edges (h,v) and (v,w) in E)
{
(1)  if (v ∈ IO(G_M) ) {
(2)     MaxD(v) = STD(In(v).psmaxest);
(3)     MinD(v) = STD(In(v).psminest);
(4)     UpdateDelta(Max(| MaxD(v) |,| MinD(v) |));
     }
(5)  if (v not visited) {
(6)     forall (s ∈ SUCC(v)) {
(7)        epsmaxest(⟨v,s⟩) = MPlus(In(v).psmaxest, eloc(⟨v,s⟩));
(8)        epsminest(⟨v,s⟩) = MPlus(In(v).psminest, eloc(⟨v,s⟩));
(9)        if (v ∈ IO(G_M) ) {
(10)          epiomaxstd(⟨v,w⟩) = MaxD(v);
(11)          epiominstd(⟨v,w⟩) = MinD(v);
(12)       } else {
(13)          epiomaxstd(⟨v,w⟩) = In(v).piomaxstd;
(14)          epiominstd(⟨v,w⟩) = In(v).piominstd;
           }
        }
(15)    Mark v visited ;
     }
(16) In(w).psmaxest = InMaxEST(epsmaxest(⟨v,w⟩), In(w).psmaxest);
(17) In(w).psminest = InMinESTf(epsminest(⟨v,w⟩), In(w).psminest);
(18) In(w).piomaxstd = InMaxD(epiomaxstd(⟨v,w⟩), In(w).piomaxstd);
(19) In(w).piominstd = InMinD(epiominstd(⟨v,w⟩), In(w).piominstd);
(20) if (⟨v,w⟩ ∉ BE(G_M)) {
(21)    E = E ∪ {⟨h,w⟩};
(22)    epsmaxest(⟨h,w⟩) = In(w).psmaxest;
(23)    epsminest(⟨h,w⟩) = In(w).psminest;
(24)    epiomaxstd(⟨h,w⟩) = In(w).piomaxstd;
(25)    epiominstd(⟨h,w⟩) = In(w).piominstd;
(26) } elseif ((InDegree(w) = 1)&&(w ≠ Entry)) { N = N − {w}; }
(27) E = E − {⟨v,w⟩};
(28) if (v has no immediate successor in G_M) {
(29)    if (v ∉ H(G_M)) { N = N − {v}; }
(30)    E = E − {⟨h,v⟩};
     }
}
```

Figure 63: Algorithm for $P_1T_2()$

Functions used in $P1T_2()$ are defined as follows:

- `InMinEST`(est_1, est_2) returns

$$\begin{cases} est_1, & if\ STD(est_1) < STD(est_2)\ or\ est_2 = (null, null), \\ est_2, & otherwise. \end{cases}$$

- `InMaxD`(max_1, max_2) returns

$$\begin{cases} max_1, & if\ max_2 < max_1\ or\ max_2 = null, \\ max_2, & otherwise. \end{cases}$$

- `InMinD`(min_1, min_2) returns

$$\begin{cases} min_1, & if\ min_1 < min_2\ or\ min_2 = null, \\ min_2, & otherwise. \end{cases}$$

- `UpdateDelta`$(Delta)$ updates $\Delta(G_M)$ with $Delta$ if $\Delta(G_M) < Delta$.

Example 7.3

Consider the example shown in Figure 58, again. Figure 64 shows the first part of the graph after the initialization at $Phase_1()$. In the graph, every edge is denoted with $(epiomaxstd(\langle a, b\rangle), epiominstd(\langle a, b\rangle))$ and every node is denoted with $(In(d).psmaxest, In(d).psminest)$.

At the function call $P_1T_2(a, b, c)$, $\Delta(G_M)$ is updated to 5 by line 2-3, since it is an i/o node and $MaxD(b) = MinD(b) = 5$. Also, $epiomaxstd(\langle b, c\rangle)$ and $epiominstd(\langle b, c\rangle)$ are replaced by $MaxD(b)$ and $MinD(b)$ by line

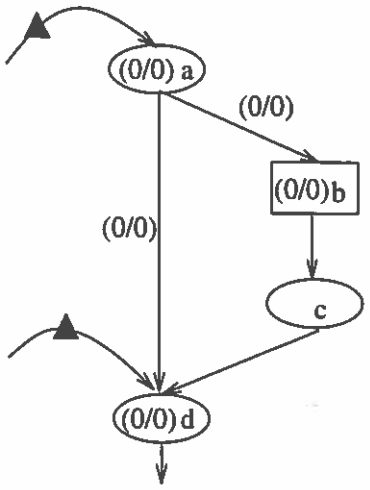


Figure 64: After initialization in $Phase_1()$

10-11. By line 18 -19, $In(c).piomaxstd = 5$ and $In(c).piominstd = 5$ and the node b is deleted by line 29.

At the function call $P_1T_2(a, c, d)$, $In(d).piomaxstd = InMaxD(5, 0) = 5$ and $In(d).piominstd = InMinD(5, 0) = 0$ by line 18-19 and the node c is deleted by line 29. Figure 65 shows graphs before and after the call $P_1T_2(a, c, d)$.

At the function call $P_1T_2(a, d, e)$, d is not deleted by line 29, since it is a header node. Figure 66 shows before and after the function call $P_1T_2(a, d, e)$.

At the function call $P_1T_2(a, e, f)$, the node e is not deleted, since it is both a header node and it has another successor node other than f, which is g. At the function call $P_1T_2(a, f, i)$, the node f is not deleted, since it has another successor node which is h. At the function call $P_1T_2(a, i, d)$, the $\Delta(G_M)$ remains the same, since $MaxD(i) = 5$ and $MinD(i) = 1$. By line

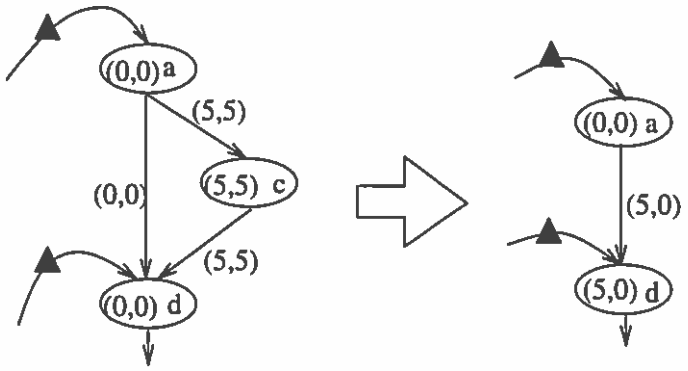


Figure 65: Before and after $P_1T_2(a, c, d)$

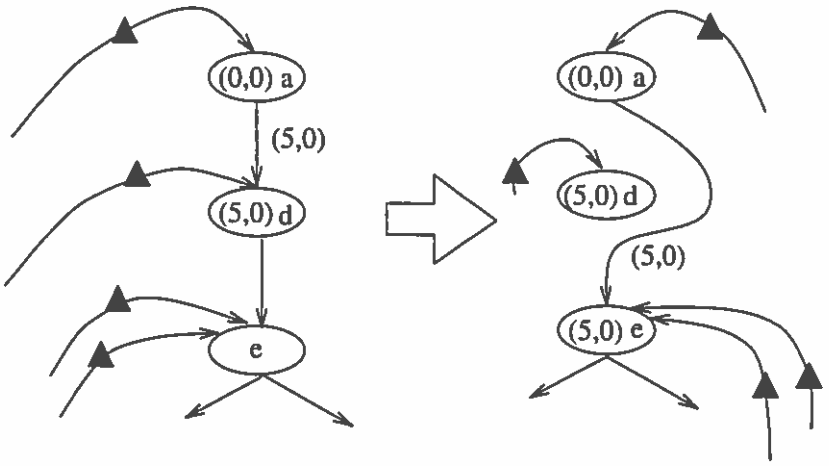


Figure 66: Before and after $P_1T_2(a, d, e)$

18-19, $In(d).piomaxstd = 5$ and $In(d).piominstd = 0$. Both i and d are deleted after $P_1T_2(a, i, d)$ by line 26 and 29. Figure 67 shows before and after the function call $P_1T_2(a, i, d)$.

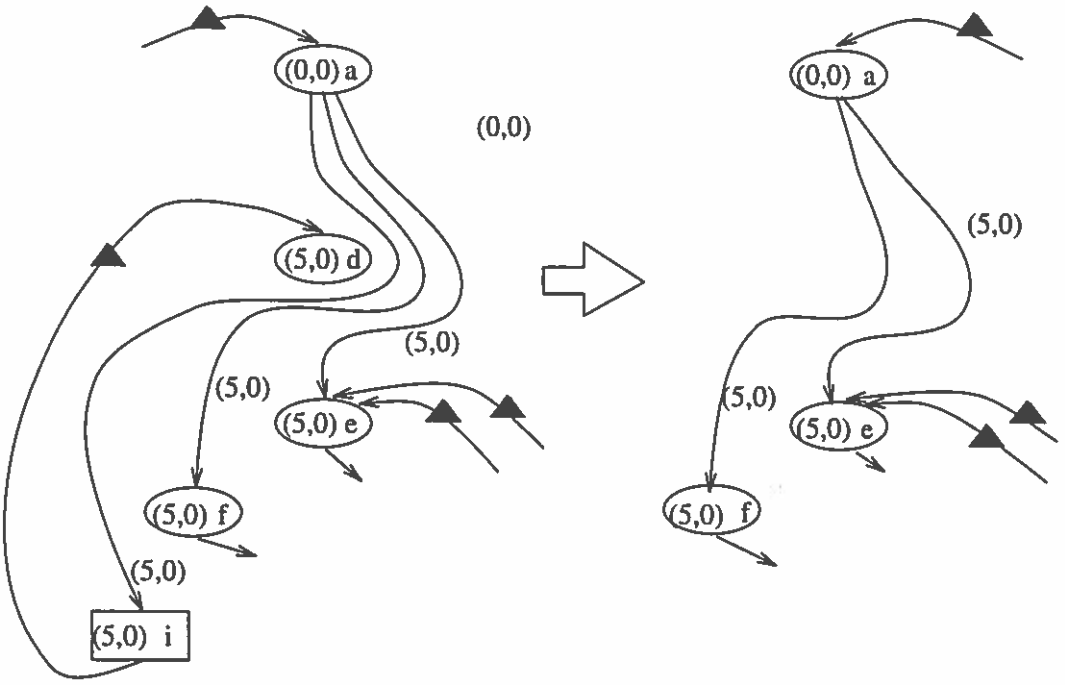


Figure 67: Before and after $P_1T_2(a, i, d)$

At the call $P_1T_2(a, f, h)$, the node f is deleted and $In(h).piomaxstd = 5$ and $In(h).piominstd = 0$. At the function call $P_1T_2(a, e, g)$, the node e is not deleted, since it is a header node. At the function call $P_1T_2(a, g, j)$, the node g is not deleted, since it has another successor which is h. At the function call $P_1T_2(a, j, e)$, the $\Delta(G_M)$ is updated to 11, since $MaxD(j)$ = -7 and $MinD(j)$ = -11. Similarly, $In(e).piomaxstd = InMaxD(-7, 5)$ = 5 and $In(e).piominstd = InMinD(-11, 0)$ = -11. After this stage, the node j is deleted but e is not since $InDegree(e)$ is 2.

At the function call $P_1T_2(a, g, h)$, the node g is deleted. At the function call $P_1T_2(a, h, e)$, the node h is not deleted, since there is another successor node which is k, but e is deleted. The value of $In(e).piomaxstd$ and $In(e).piominstd$ remains the same, since $In(e).piomaxstd = 5$ and $In(e).piominstd$ = -11. Figure 68 shows before and after the function call $P_1T_2(a, h, e)$.

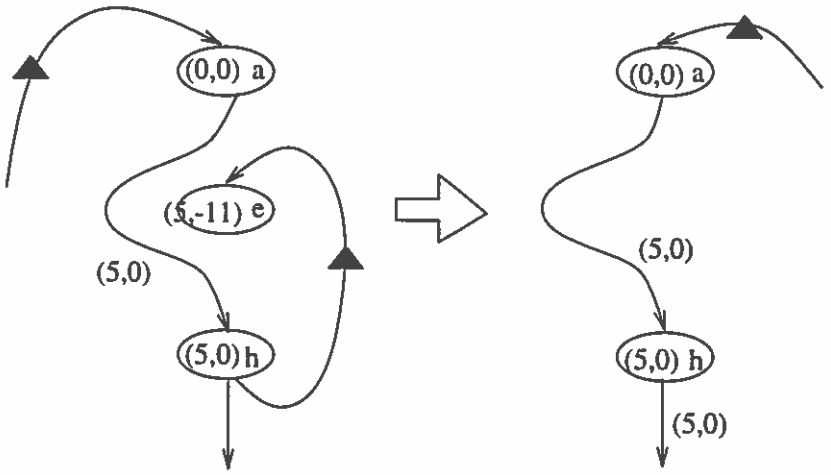


Figure 68: Before and after $P_1T_2(a, h, e)$

At the function call $P_1T_2(a, h, k)$, the node h is deleted. At the function call $P_1T_2(a, k, a)$, the $\Delta(G_M)$ is updated to 16, since $MaxD(k) = 0$ and $MinD(k)$ = -16. For the same reason, $In(a).piomaxstd = InMaxD(0, 0)$ = 0 and $In(a).piominstd = InMinD(-16, 0)$ = -16. Finally, the node k is deleted and the graph G_M is reduced into a node. □

Phase 2

The second phase of the algorithm, shown in Figure 69 and 70, finds the relative timing sensitivity for each i/o node $\Psi(io)$ using the data-flow equations shown

in Equation VII.54-VII.57. The first phase of the algorithm finds $MaxD(io)$ and $MinD(io)$ for all $io \in IO(G_M)$ and $MaxD(hio)$ and $MinD(hio)$ for all header nodes $h \in H(G_M)$.

The second phase of the algorithm find $MaxPIO(io)$ and $MinPIO(io)$ by propagating $MaxD(hio)$ and $MinD(hio)$ through the DAG. These can be obtained by an algorithm similar to that of $Phase_1()$. When the value of $MaxPIO(io)$ and $MinPIO(io)$ are available for io, $MaxMin(io)$ and $MinMax(io)$ are found. The complexity of this algorithm is $O(\mid E \mid)$ as well. Again, the node *Entry* is treated as an i/o node since it generates the event *Program Start*. Since any incoming edge to the *Entry* node is a technical edge, the value of $In(Entry).piomaxstd$ and $In(Entry).piominstd$ are ignored.

Example 7.4

Figure 71 shows the result of Example 7.3. In the graph, each header node $h \in H(G_M)$ has ($In(h).piomaxstd$,$In(h).piominstd$) inside it and each i/o node $io \in IO(G_M)$ has $(MaxD(io), MinD(io))$ beside it. The *Entry* is again treated as an i/o node. The value of *In(Entry).piomaxstd* and *In(Entry).piominstd* are ignored by line 2-3 and 6-7.

At the function call $P_2T_2(a, b, c)$, $MaxMin(b)$ = -5 and $MinMax(b)$ = 5, since *In(b).piomaxstd* = 0 and *In(b).piominstd* = 0. The relative timing sensitivity of G_M, $\Psi(G_M)$, is updated to 5. At the function call $P_2T_2(a, c, d)$, the the node c is deleted. The values of *In(d).piomaxstd* and $In(d).piominstd$ remain the same, since *In(d).piomaxstd* = $MaxD(5, 5)$ = 5 and *In(b).piominstd* = $MinD(5, 0) = 0$.

```
Phase_2(G_M);
{
(1)  h = Entry;
(2)  MaxMin(h) = 0
(3)  MinMax(h) = 0
(4)  UpdatePsi(0);
(5)  Mark h visited ;
(6)  forall (s ∈ SUCC(h)) {
(7)      epiomaxstd(⟨h,s⟩) = 0;
(8)      epiominstd(⟨h,s⟩) = 0;
(9)      In(s).piomaxstd = InMaxD(0, In(s).piomaxstd);
(10)     In(s).piominstd = InMinD(0, In(s).piominstd);
     }
(11) while (there exist an edge ⟨v,w⟩ ∈ E such that v ∈ SUCC(h)
         and if ⟨u,v⟩ ∈ E then either u = h or u ∈ BN(G_M)) {
(12)     choose any such (v,w)
(13)     P_2T_2(h,v,w);
     }
}
```

Figure 69: Phase II: finds relative timing sensitivity of G_M.

```
P2T2( h,v,w:  nodes of edges (h,v) and (v,w) in E)
{
(1)  if (v ∈ IO(G_M) ) {
(2)     MaxMin(h) = In(v).piomaxstd − MinD(v)
(3)     MinMax(h) = MaxD(v) − In(v).piominstd
(4)     UpdatePsi(| MaxMin(h) |,| MaxMin(h) |);
     }
(5)  if (v not visited) {
(6)     forall (s ∈ SUCC(v)) {
(7)        if (v ∈ IO(G_M) ) {
(8)           epiomaxstd(⟨v,w⟩) = MaxD(v);
(9)           epiominstd(⟨v,w⟩) = MinD(v);
(10)       } else {
(11)          epiomaxstd(⟨v,w⟩) = In(v).piomaxstd;
(12)          epiominstd(⟨v,w⟩) = In(v).piominstd;
           }
        }
(13)    Mark v visited ;
     }
(14) In(w).piomaxstd = InMaxD(epiomaxstd(⟨v,w⟩), In(w).piomaxstd);
(15) In(w).piominstd = InMinD(epiominstd(⟨v,w⟩), In(w).piominstd);
(16) if (⟨v,w⟩ ∉ BE(G_M)) {
(17)    E = E ∪ {⟨h,w⟩};
(18)    epiomaxstd(⟨h,w⟩) = In(w).piomaxstd;
(19)    epiominstd(⟨h,w⟩) = In(w).piominstd;
(20) } elseif (InDegree(w) = 1) { N = N - { w }; }
(21) E = E − {⟨v,w⟩};
(22) if (v has no immediate successor in G_M) {
(23)    if (v ∉ H(G_M)) { N = N − {v}; }
(24)    E = E − {⟨h,v⟩};
     }
}
```

Figure 70: Algorithm for $P_2T_2()$

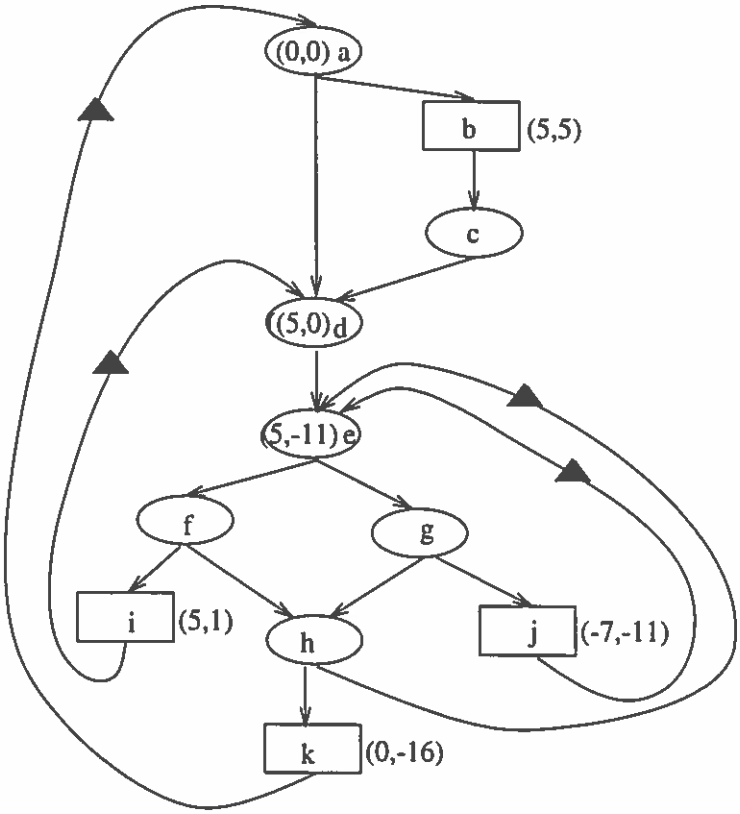


Figure 71: The control flow graph after phase I

At the function call $P_2T_2(a, d, e)$, the node d is not deleted, since it is a header node. The values of *In(e).piomaxstd* and $In(e)$*.piominstd* remain the same, since *In(d).piomaxstd* = $MaxD(5,5) = 5$ and *In(b).piominstd* = $MinD(0,-11)$ = -11.

At the function call $P_2T_2(a, e, f)$, the node e is not deleted, since it is a header node. The values of *In(f).piomaxstd* and *In(f).piominstd* are changed to 5 and -11, since *In(d).piomaxstd* = $MaxD(5,5) = 5$ and *In(b).piominstd* = $MinD(-11,0)$ = -11.

At the function call $P_2T_2(a, f, i)$, the node f is not deleted, since it has another successor node which is h. The values of *In(i).piomaxstd* and *In(i).piominstd* are changed to 5 and -11, since *In(d).piomaxstd* = $MaxD(5,5) = 5$ and *In(b).piominstd* = $MinD(-11,0)$ = -11.

At the function call $P_2T_2(a, i, d)$, both i and d are deleted. *In(d).piomaxstd* and *In(d).piominstd* does not changed, since *In(d).piomaxstd* = 5 and *In(b).piominstd* = 0. The relative timing sensitivity of G_M is updated to 16 since $MinMax(i) = 16$ and $MaxMin(i) = 4$.

At the function call $P_2T_2(a, f, h)$, the node f is deleted. The values of *In(h).piomaxstd* and *In(h).piominstd* are changed to 5 and -11, since *In(h).piomaxstd* = $MaxD(5,5) = 5$ and *In(h).piominstd* = $MinD(-11,0)$ = -11.

At the function call $P_2T_2(a, e, g)$, the node e is not deleted, since it is a header node. The values of *In(g).piomaxstd* and *In(g).piominstd* are changed to 5 and -11, since *In(g).piomaxstd* = $MaxD(5,5) = 5$ and *In(g).piominstd* = $MinD(-11,0)$ = -11.

At the function call $P_2T_2(a, g, j)$, the node g is not deleted, since it has another successor node which is h. The values of *In(j).piomaxstd* and *In(j).piominstd* are changed to 5 and -11, since *In(j).piomaxstd* = $MaxD(5, 5) = 5$ and *In(j).piominstd* = $MinD(-11, 0)$ = -11.

At the function call $P_2T_2(a, j, e)$, the node j is deleted.

The values of *In(e).piomaxstd* and *In(e).piominstd* are not changed, since *In(e).piomaxstd* = 5 and *In(e).piominstd* = -11. The relative timing sensitivity of G_M is not changed, since $MinMax(j) = 4$ and $MaxMin(j) =$ 16.

At the function call $P_2T_2(a, g, h)$, the node g is deleted. The values of *In(h).piomaxstd* and *In(h).piominstd* are not changed. At the function call $P_2T_2(a, h, e)$, the node h is not deleted, but e is deleted. The values of *In(e).piomaxstd* and *In(e).piominstd* are not changed.

At the function call $P_2T_2(a, h, k)$, the node h is deleted. The values of *In(k).piomaxstd* and *In(k).piominstd* are changed to 5 and -11, since *In(k).piomaxstd* = $MaxD(5, 5) = 5$ and *In(k).piominstd* = $MinD(-11, 0)$ = -11.

At the function call $P_2T_2(a, k, a)$, the node k is deleted, but not a. The values of *In(a).piomaxstd* and *In(a).piominstd* are changed to 0 and -16, since *In(a).piomaxstd* = $MaxD(0, 0) = 0$ and *In(a).piominstd* = $MinD(-16, 0)$ = -16. The relative timing sensitivity of G_M is changed to 21, since $MinMax(k) = 11$ and $MaxMin(j) = 21$.

□

Summary

This chapter presents an algorithm to find both timing sensitivities for a given control flow graph G_M. To find these timing sensitivities, the pure local synchronization scheme is assumed. Optimization techniques that reduce these timing sensitivities further are discussed in the next chapter. The same algorithm can be used to find these sensitivities for the optimized control flow graph.

CHAPTER VIII

OPTIMIZATION FOR TIMING EQUIVALENCE

Chapter VII presented a method to measure timing sensitivities for a given target program. This chapter presents methods that minimize the timing sensitivities for a target program. For the optimization of timing sensitivities, we assume the local synchronization scheme is used and all back edges in the graph are locally synchronized.

Problem Statement

The goals of an optimization performed by a translator vary depending on requirements of the system. In most cases, optimization is to reduce the total execution time of a program by using fewer resources. However, the optimization goal for real-time systems is to minimize the timing difference of the target program with respect to the source timing specified in the source program. In the context of binary-to-binary translation of real-time programs, optimization can be one of two cases: converting a timing divergent target program into a timing equivalent or invariant one and/or reducing timing sensitivities of a timing invariant target program.

If a target program is timing divergent, the correctness of the resultant timing is not guaranteed. It may maintain timing equivalence or invariance for a while, but will not after some point. The exact timing behavior of this target program is not known until run-time, which is dangerous by the definition of real-time systems.

Thus, conversion of such a target program into a timing invariant or equivalent one is an important optimization.

Even though the given target program is timing invariant, if the sensitivities of it are greater than the desired or required level, the target program may not be useful. Thus, the reduction of these timing sensitivities is also an important optimization.

To convert a timing divergent target program into an invariant one, we must reduce the execution time required for each node or paths in the control flow graph of the target program. The required execution time for each basic block or paths can be reduced by ordinary compiler optimization techniques such as code motion [34, 35], strength reduction [12, 59], constant propagation [47, 57, 58], elimination of redundant computations [56, 44] etc.. These optimizing transformations require analysis of codes inside basic blocks. A typical analysis used by most optimizing transformations is data-flow analysis [1, 20, 28, 30, 21, 48].

Since we do not deal with codes inside basic blocks, we only discuss timing optimization techniques that use program execution delay such as synchronization, assuming the target program is executable with timing invariance.

Synchronization can be used both in enforcing timing invariance and in minimizing timing sensitivities of the target program. The enforcement of timing invariance for the given target program was achieved by inserting synchronization on all back edges in the control flow graph as seen in Chapter VI. We also showed an example that used synchronization to reduce timing sensitivities in Example 6.4.

However, the insertion of synchronization does not always reduce timing sensitivity. This chapter discusses how to insert synchronization to reduce timing sensitivity.

We mainly focus on optimizing the absolute timing sensitivity of a given graph. We use the term "timing sensitivity" to mean the absolute timing sensitivity. Optimizing the absolute timing sensitivity does not always reduce the relative timing sensitivity. In general, the relative timing sensitivity is reduced by reducing the absolute timing sensitivity by Theorem 4.1.

Optimization Through Synchronization

After timing invariance of a target program is enforced by inserting synchronization on every back edge, optimization of timing sensitivities can be performed by inserting additional synchronizations in the graph. A synchronization can be inserted anywhere(node or edge) in the control flow graph, although inserting additional synchronization does not always reduces timing sensitivities. We now mainly focus on inserting synchronization on i/o nodes since timing sensitivities are defined with respect to i/o events, and thus instances of i/o nodes.

Header nodes are special in that local clocks for each interval start from the header node. If the sensitivity, timing difference, at the header node is high, the timing sensitivity of any i/o node in the interval is also high. The local synchronization scheme does not remove timing error caused by the outer intervals. Thus, by inserting synchronization on the header node of the interval the timing sensitivity may be reduced.

However, there are two different kind of edges in $InEdges(h)$ for a header node h , i.e., back and incoming edges. All back edges are synchronized using the interval's local clocks but incoming edges do not have access to the local clocks. It must use the outer interval's local clocks. To distinguish these two cases, we insert synchronizations on incoming edges of the interval instead of on the header node. We

showed an example that reduces timing sensitivity by inserting synchronizations on incoming edges of intervals in Example 6.4.

Insertion of Synchronizations

The insertion of a local synchronization in a graph may invalidate some instances of synchronizations on back edges. Some *head-to-back paths* in the graph may miss their time deadline by the insertion. To make the local synchronization scheme work, all instances of synchronizations on all back edges must remain valid.

Definition 8.1

An insertion of a synchronization in a control flow graph G_M is said to be *legal* if all instances of synchronizations on all back edges remain valid for all $cp \in CP(G_M)$ after the insertion. □

To examine if the insertion of a synchronization on v in $I(h)$ is legal, it is necessary to examine all head-to-back paths in the graph that v supports, all sub-intervals in $I(h)$ are already reduced. Some of these head-to-back paths may be in $HBP(I(h))$ but others may not be in it. If v supports head-to-exit paths of $I(h)$ and all head-to-back paths that v supports meet their time deadline, the next outer interval has to be examined since the head-to-exit paths of $I(h)$ is a part of head-to-back paths of the outer interval. This process continues until a head-to-back path that does not meet its time deadline is found or all head-to-back paths in the outer-most interval are examined.

Consider the control flow graph shown in Figure 72. There are three intervals: $I(Entry)$, $I(hi)$, and $I(hj)$. To examine if the insertion of a synchronization on io is legal, it is necessary to examine if all simple paths from hj to $bj1$ that contain io meet

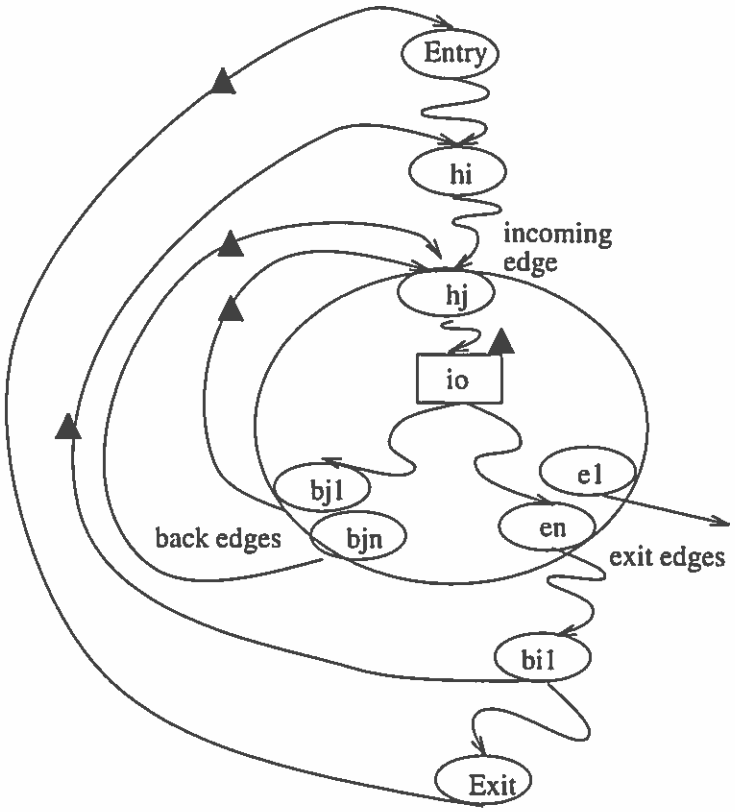


Figure 72: Insertion of synchronization at *io*

their time deadline after the insertion. If they do, the next outer interval $I(hi)$ must be examined. To examine $I(hi)$, all simple paths from hi to en that contain io have to be examined. If all those simple paths meet their time deadline, the most-outer interval $I(Entry)$ is examined. If all head-to-back paths in the outer-most interval meet their time deadline, the insertion is legal.

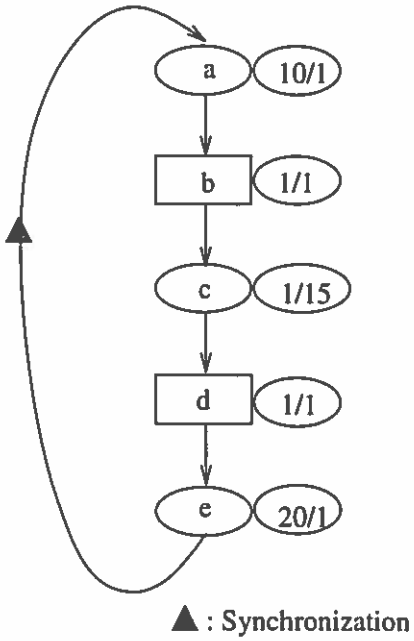


Figure 73: Legal but not positive

Even when the insertion of synchronization on v is legal, it does not always improve timing sensitivity. Consider a control flow graph shown in Figure 73. Insertion of synchronization on node b is legal but the timing sensitivity of the graph get worse from 9 to 14 after the insertion. Thus, it is required to check if the insertion of a synchronization improves timing sensitivity of the control flow graph.

Definition 8.2

An insertion of a synchronization in a control flow graph is said to be

positive if it reduces the timing sensitivity of the graph. □

To examine if the insertion of a synchronization in a graph is positive, the timing sensitivity of the graph has to be computed after the insertion.

Definition 8.3

An insertion of a synchronization in a control flow graph G_M is said to be *safe* if all instances of the synchronization for all $cp \in CP(G_M)$ are valid with local clocks. □

To be safe, the insertion must be legal and all simple paths from the header of the interval to v must meet their time deadlines. By Lemma 6.1, it is obvious that the insertion of synchronization on every back edge is valid if the graph is executable with timing invariance.

An unsafe insertion of synchronization in a graph may improve timing sensitivity if it is legal. Consider the example shown in Figure 74. The insertion of a synchronization on e is legal but not safe. In the graph, the synchronization on e is unsafe if the path $\langle a, b, d \rangle$ is taken, but safe if the path $\langle a, c, d \rangle$ is taken. By inserting synchronization on e, the timing sensitivity of the graph is reduced from 15 to 1.

In Chapter V, we found $MaxEST(h, v)$ using a data-flow approach. To allow unsafe insertions of synchronization in the graph other than back edges, we also need to find $MinEST(h, v)$ for each node in the interval. This also can be found using the same data-flow approach, in fact, both $MaxEST(h, v)$ and $MinEST(h, v)$ can be found together with the same algorithm. The data-flow equations to find $MinEST(h, v)$ are as in Equation VIII.58 and Equation VIII.59.

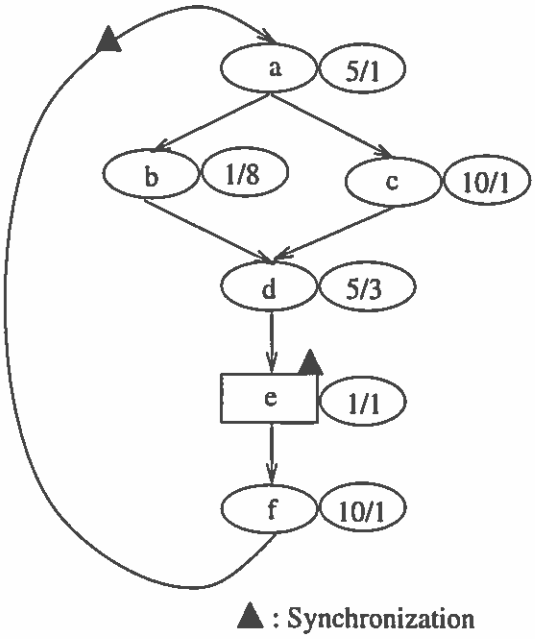


Figure 74: Unsafe but positive

$$In(v).hvminest = \begin{cases} (0,0), & v = header, \\ MINEST(v), & otherwise, \end{cases} \tag{VIII.58}$$

$$ehvminest(\langle v,w\rangle) = \begin{cases} eloc(\langle v,w\rangle), & v = header, \\ MI(\langle v,w\rangle), & otherwise, \end{cases} \tag{VIII.59}$$

where

- The function $MINEST(v)$ returns $ehvminest(e)$ such that

$$MinSTD(h,v) = \min_{e \in InEdge(v)} STD(ehvminest(e))$$

- $MI(\langle v,w\rangle) = In(v).hvminest + eloc(\langle v,w\rangle)$

Data-flow Equations with Synchronizations

To insert synchronization in the graph, we need to test if the insertion is legal and positive. Algorithm shown in Figure 39 and Figure 62 can be used to examine if the insertion is legal and positive. However, it requires a modification on data-flow equations to support these additional synchronizations.

Consider the control flow graph shown in Figure 72. Assume the insertion of a synchronization on io in $I(hj)$ is legal. Since unsafe insertion of synchronization is allowed, three cases occur after the insertion at io depending on $STD(\sigma(h,v))$ as shown in Figure 75. In case 1, the insertion is safe since both $MaxSTD(h,v)$ and $MinSTD(h,v)$ are less than zero. In case 2, the insertion is not safe but the insertion may reduces timing sensitivity of the graph since $MinSTD(h,v)$ is less than zero. In case 3, the insertion is not safe and it does not change timing sensitivity of the graph.

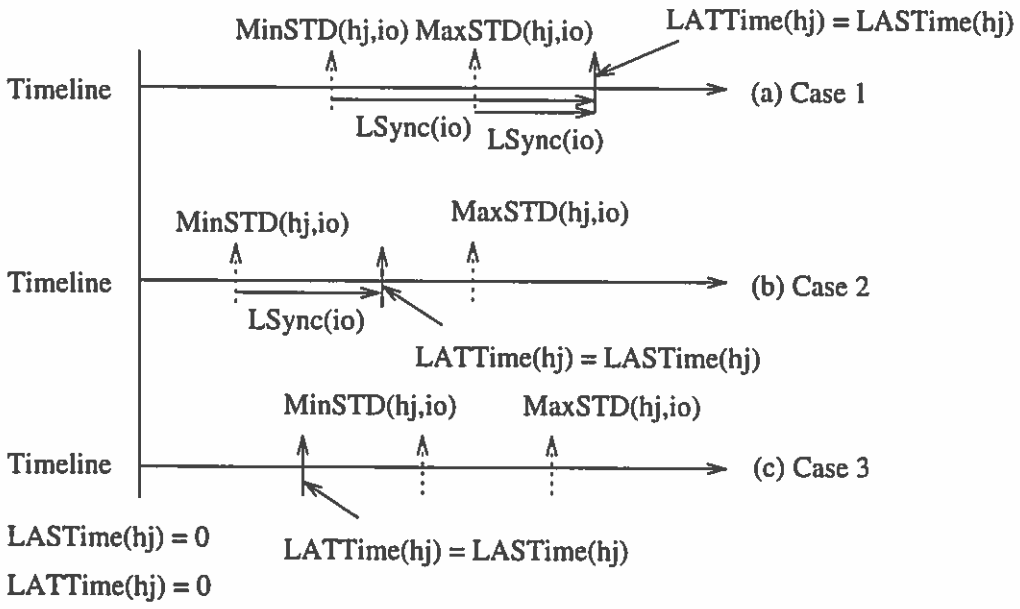


Figure 75: After the insertion of synchronization on io.

Similarly, if a synchronization is inserted on an edge, the synchronization is performed before the node w only if the control is from the edge. Considering these facts, the data-flow equations which find $MaxEST(h, bn)$ and $MinEST(h, bn)$ for all $h \in H(G_M)$ are changed as follows.

$$In(v).hvmaxest = \begin{cases} (0,0), & v = header\ or \\ & LSync(v)\ and\ STD(MAXEST(v)) < 0 \\ MAXEST(v), & otherwise, \end{cases} \tag{VIII.60}$$

$$In(v).hvminest = \begin{cases} (0,0), & v = header\ or \\ & LSync(v)\ and\ STD(MINEST(v)) < 0 \\ MINEST(v), & otherwise, \end{cases} \tag{VIII.61}$$

$$ehvmaxest(\langle v, w \rangle) = \begin{cases} (0,0), & LSync(\langle v, w \rangle)\ and\ STD(MA(\langle v, w \rangle)) < 0 \\ eloc(\langle v, w \rangle), & v = header \\ MA(\langle v, w \rangle), & otherwise, \end{cases} \tag{VIII.62}$$

$$ehvminest(\langle v,w\rangle) = \begin{cases} (0,0), & LSync(\langle v,w\rangle)\ and\ STD(MI(\langle v,w\rangle)) < 0 \\ eloc(\langle v,w\rangle), & v = header \\ MI(\langle v,w\rangle), & otherwise, \end{cases} \tag{VIII.63}$$

where

- $LSync(v)$ means the node v is locally synchronized

The algorithm that examines if an insertion of synchronization is legal can be developed by modifying the algorithm given in Figure 39. If the insertion of synchronization is legal, we examine if it is positive, i.e., reduces the timing sensitivity of the graph. The algorithm that examines if the insertion is positive can be developed by modifying the algorithm shown in Figure 62 with slight modification of the data-flow equations. The data-flow equations are changed as follows:

$$In(v).psmaxest = \begin{cases} (0,0), & GSync(v), \\ In(Header(v)).psmaxest, & LSync(v)\ and \\ & STD(In(v).hvmaxest) < 0 \\ MAXEST(v), & otherwise, \end{cases} \tag{VIII.64}$$

$$In(v).psminest = \begin{cases} (0,0), & GSync(v), \\ In(Header(v)).psminest, & LSync(v) \text{ and} \\ & STD(In(v).hvminest) < 0 \\ MINEST(v), & otherwise, \end{cases} \tag{VIII.65}$$

$$epsmaxest(\langle v,w\rangle) = \begin{cases} eloc(\langle v,w\rangle), & GSync(\langle v,w\rangle), \\ In(Header(v)).psmaxest, & LSync(\langle v,w\rangle) \text{ and} \\ & STD(In(v).hvmaxest) < 0 \\ PMA(\langle v,w\rangle), & otherwise, \end{cases} \tag{VIII.66}$$

$$epsminest(\langle v,w\rangle) = \begin{cases} eloc(\langle v,w\rangle), & GSync(\langle v,w\rangle), \\ In(Head(v)).psminest, & LSync(\langle v,w\rangle) \text{ and} \\ & STD(In(v).hvminest) < 0 \\ PMI(\langle v,w\rangle), & otherwise, \end{cases} \tag{VIII.67}$$

where

- $GSync(v)$ means the node v is globally synchronized;
- $Header(v)$ returns the intervals header of v.

Algorithm for Timing Optimization

We now present our algorithm to insert synchronizations in the graph to optimize the timing sensitivity of the graph. The algorithm finds where to insert local synchronizations in the graph to reduce the timing sensitivity. As we mentioned, we insert synchronization at every i/o node and incoming edges for all intervals if it is legal and positive. The order of inserting synchronization is also important to reduce timing sensitivity. Since the timing sensitivity is affected by previous global synchronization points and the insertion of synchronization of a node v affects all successors of v in the DAG, the algorithm starts with $Entry$. The algorithm used to find the absolute timing sensitivity can be modified to insert synchronizations as shown in Figure 76. The function $IST_2()$ inserts a local synchronization on all i/o nodes and incoming edges of intervals if it is legal and positive.

The functions used in this algorithm are defined as follows:

- The functions $Legal(v)$ and $Legal(\langle v, w \rangle)$ determine if the insertion is legal. They return $True$ if the insertion of synchronization at the node v or edge $\langle v, w \rangle$ does not invalidate any synchronization on back edges in the graph. As mentioned, these functions can be implemented using the algorithm shown in Figure 39 with the modified data-flow equations. The complexity of a naive version of this algorithm is the same as the interval algorithm, which is $O(\mid E \mid log \mid E \mid)$. [18].

- The functions $Positive(v)$ and $Positive(\langle v, w \rangle)$ determine if the insertion is positive. They return $True$ if the insertion of synchronization at the node v or edge $\langle v, w \rangle$ reduces the timing sensitivity of the graph. As mentioned, these

```
IST2(h,v,w: nodes of edges (h,v) and (v,w) in E)
     if (v ∈ IO(GM)) {
        If ( Legal(v) and Positive(v)){
           InsertLSync(v)
        }
     }
     if (⟨v,w⟩ ∉ BE(GM)) {
        if (w ∈ H(GM)) {
           if (Legal(⟨v,w⟩) and Positive(⟨v,w⟩)){
              InsertLSync(⟨v,w⟩)
           }
        }
        E = E ∪{⟨h,w⟩};
     } elseif (InDegree(w) = 1) {
        N = N - { w };
     }
     E = E -{⟨v,w⟩};
     if (v has no immediate successor in GM) {
        if (v ∉ H(GM)) {
           N = N - {v}
        }
        E = E - {⟨h,v⟩};
     }
}
InsertLSync(GM);
{
     h = Entry;
     while (there exist an edge ⟨v,w⟩ ∈ E such that v ∈ SUCC(h)
        and if ⟨u,v⟩ ∈ E then either u = h or u ∈ BN(GM)) {
        choose any such (v,w)
        IST2(h,v,w)
     }
}
```

Figure 76: Algorithm for timing optimization by inserting local synchronizations

functions can be implemented using the algorithm shown in Figure 62 with the modified data-flow equations. The complexity of a naive version of these functions is $O(log \mid E \mid)$.

Complexity of the algorithm is $O(\mid E \mid^2 log \mid E \mid)$ since each time the function $IST_2()$ is called, an edge is reduced and each reduction requires invocation of both *Legal*() and *Positive*(). Once the absolute timing sensitivity is found, finding the relative timing sensitivity is easy to find using the algorithm given in Figure 69.

Example 8.1

Consider the example shown in Figure 77. There are four intervals, $I(a)$, $I(c)$, $I(d)$ and $I(l)$, in the graph. The outer-most interval $I(a)$ has a sub-interval $I(c)$ which itself has two sub-intervals, $I(d)$ and $I(l)$. The set $IO(G_M) = \{b, e, i, k, m, r\}$ and $H(G_M) = \{a, c, d, l\}$.

The graph can be reduced by the algorithm shown in Figure 39. The algorithm starts by reducing one of the inner-most intervals. In this graph, the algorithm can start with either the interval $I(d)$ or $I(l)$. Figure 78 shows the graph after both $I(d)$ and $I(l)$ are reduced. Exit nodes (h and o) and header nodes (d and l) of these reduced intervals are not deleted. When an interval is reduced, values of $ehvmaxest(\langle xn, ss\rangle)$ are saved on exit edges, $eloc(\langle xn, ss\rangle)$, by line 15 in the algorithm shown in Figure 40. Here, xn is an exit node and ss is a $SUCC(xn)$. These values are denoted on exit edges in the reduced graph. The next stage of the algorithm starts since all $STD(ehvmaxest(\langle bn, h\rangle))$ are less than zero.

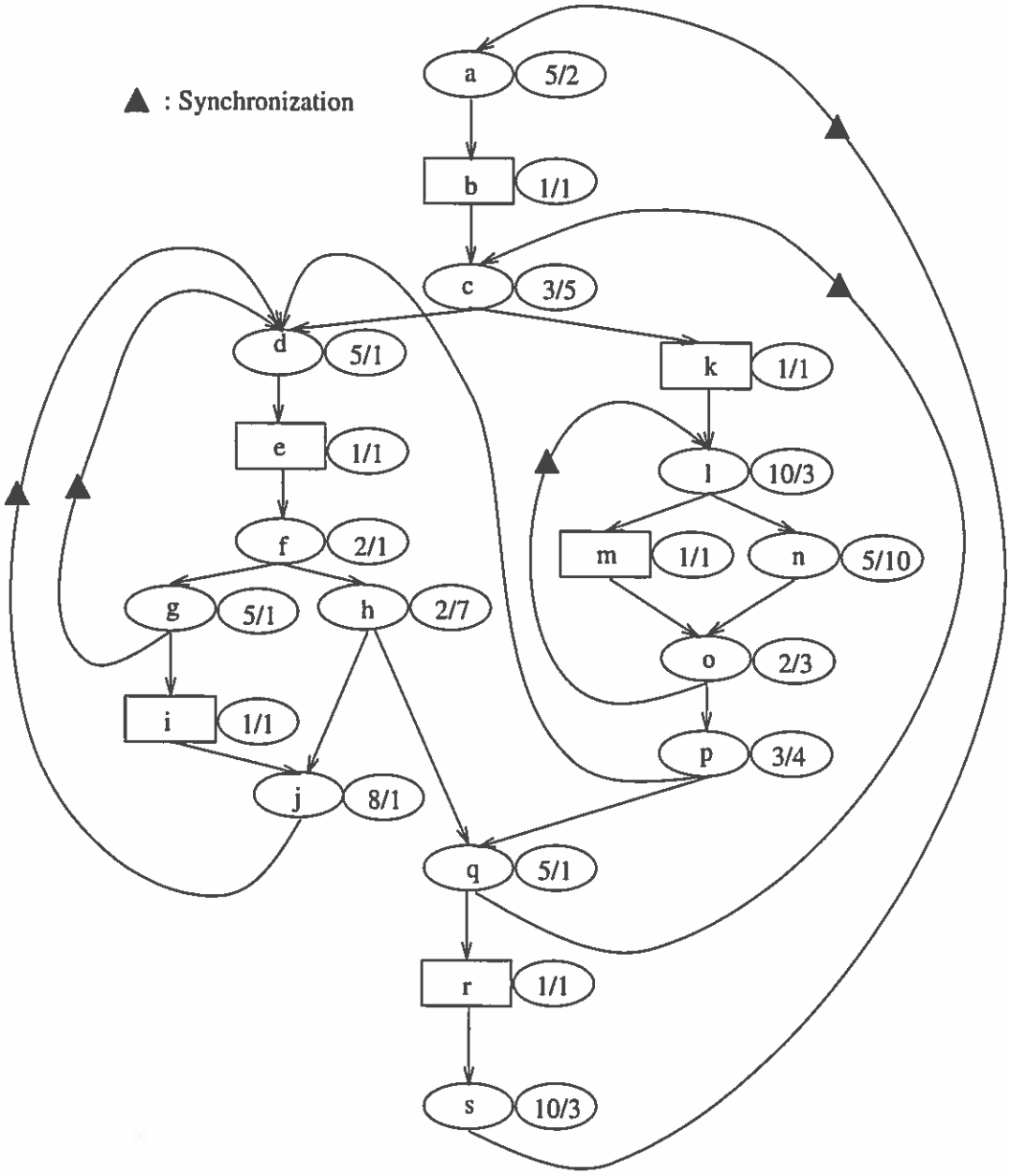


Figure 77: An example of control flow graph for optimization

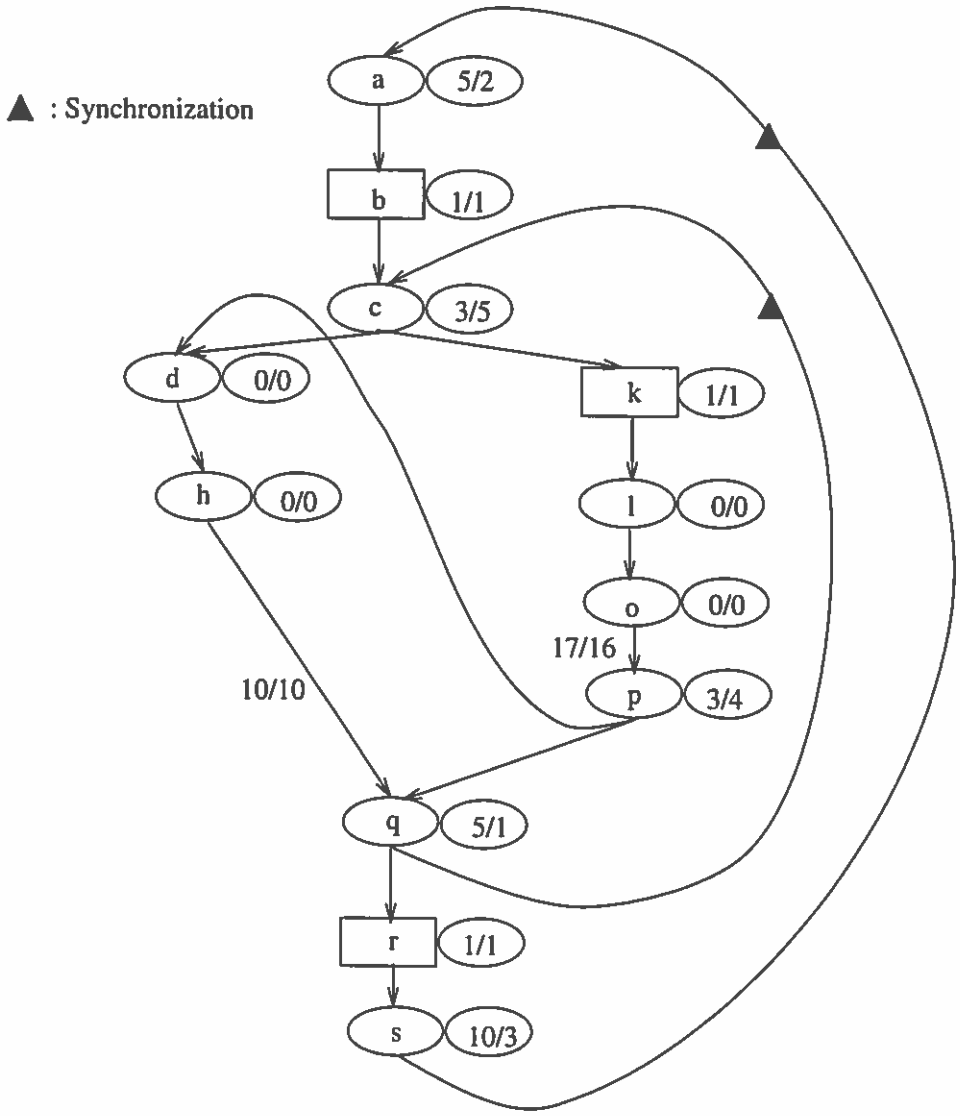


Figure 78: After the two inner-most intervals are reduced

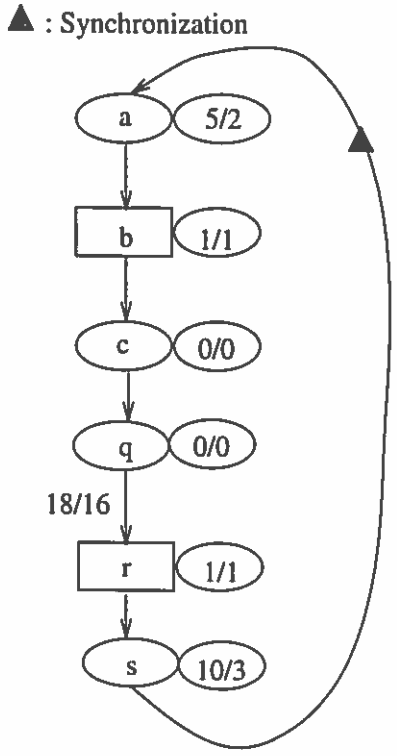


Figure 79: After all sub-intervals are reduced

The next stage of the algorithm reduces the next outer interval, $I(c)$. Figure 79 shows the graph after the interval $I(c)$ is reduced. Again, the header node c and exit node q are not deleted.

The algorithm finally reduces the given graph into a single node a since $STD(ehvmaxest(\langle s, a\rangle))$ is less than zero. Thus, the graph given is executable with timing invariance. Once all back edges are locally synchronized, the graph is timing invariant.

The absolute timing sensitivity can be found by applying Algorithm 62. The absolute timing sensitivity of each i/o node are $\Delta(b) = 3$, $\Delta(e) = 10$, $\Delta(i) = 15$, $\Delta(k) = 1$, $\Delta(m) = 8$, and $\Delta(r) = 10$. Thus, the absolute timing sensitivity of the graph is 15.

Now, we optimize timing sensitivity of the graph by inserting additional synchronizations.

At the function call $IST_2(a, b, c)$, the algorithm examines if the insertions of a synchronization on both e and $\langle b, c\rangle$ are legal and positive. Consider the insertion of a synchronization on b first. The function *Legal*() examines if it is legal by checking all *header-to-back paths* that contains b. This can be performed with the reduced graph shown in Figure 79, where all sub-intervals are reduced. The insertion of synchronization on b is legal since $STD(ehvmaxest(\langle s, a\rangle)) = STD((27, 20))$ = -7 after the insertion. Since the insertion of synchronization on b is valid, $MaxEST(\langle a, b\rangle) =$ (5,2), the absolute timing sensitivity may be reduced. After the insertion of a synchronization on b, the absolute timing sensitivity of each i/o node becomes $\Delta(b) = 0$, $\Delta(e) = 7$, $\Delta(i) = 12$, $\Delta(k) = 2$, $\Delta(m) = 5$, and $\Delta(r)$

= 7. The absolute timing sensitivity of the graph is reduced from 15 to 12 and thus the insertion of synchronization on b is legal and positive. Figure 80 shows before and after $IST_2(a, b, c)$.

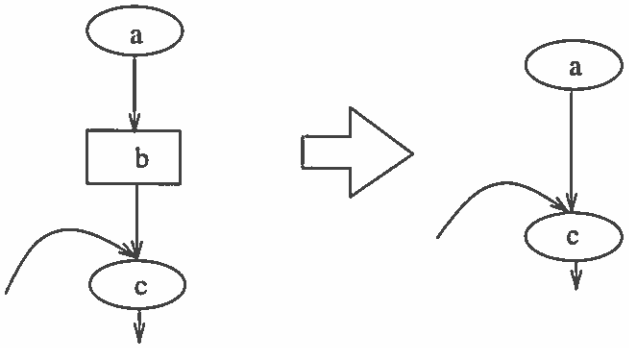


Figure 80: Before and after $IST_2(a, b, c)$

The insertion of a synchronization on $\langle b, c\rangle$ is legal and but not positive.

At the function call $IST_2(a, k, l)$, the insertion of a synchronization on k is legal but not positive. Similarly, the insertion of a synchronization on $\langle k, l\rangle$ is legal but not positive.

The insertion of synchronization on m is not legal. The insertion of synchronization on $\langle p, d\rangle$ is legal since $ehvmaxest(\langle q, c\rangle)$ is (39,37) and $ehvmaxest(\langle s, a\rangle)$ =(56,47). After the insertion of a synchronization on $\langle p, d\rangle$, the absolute timing sensitivity of each i/o node becomes $\Delta(b) = 0$, $\Delta(e) = 4$, $\Delta(i) = 9$, $\Delta(k) = 2$, $\Delta(m) = 5$, and $\Delta(r) = 7$. The absolute timing sensitivity of the graph is reduced from 12 to 9 and thus the insertion of synchronization on $\langle p, d\rangle$ is legal and positive. Figure 81 shows before and after $IST_2(a, p, d)$.

The insertion of a synchronization on $\langle c, d\rangle$ legal but not positive.

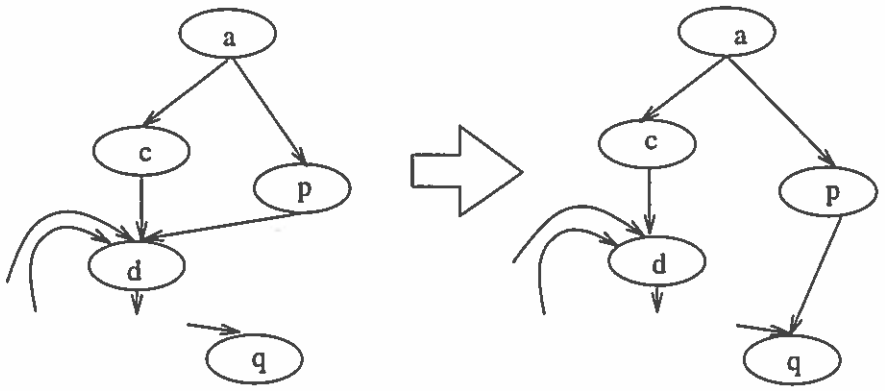


Figure 81: Before and after $IST_2(a, p, d)$

At the function call $IST_2(a, e, f)$, the algorithm examines if $Legal(e)$ returns $True$. At the function call $Legal(e)$, it first examines back edges of $I(d)$. Both $STD(ehvmaxest(\langle g, d\rangle))$ and $STD(ehvmaxest(\langle j, d\rangle))$ after the insertion are less than zero. Since e supports a *head-to-exit path* of the interval and $STD(ehvmaxest(\langle h, q\rangle))$ increases from 0 (10-10) to 4 (9-5), the algorithm examines all outer intervals.

To examine the outer interval $I(c)$, we can use the graph shown in Figure 78 with the updated execution time of *head-to-exit path* that is denoted on the exit edge $\langle h, q\rangle$. It is not legal for the outer interval $I(c)$ since $STD(ehvmaxest(\langle q, c\rangle))$ is greater than zero as seen in Figure 82. Thus, the insertion of a synchronization on e is not legal. The insertion of a synchronization on i is legal since $STD(ehvmaxest(\langle j, d\rangle))$ is less than zero and and i does not support head-to-exit paths of $I(d)$. After the insertion the timing sensitivity of i is reduced from 9 to 2 and thus the timing sensitivity of the graph is reduced from 9 to 7. Thus the insertion of synchronization on i is legal and positive.

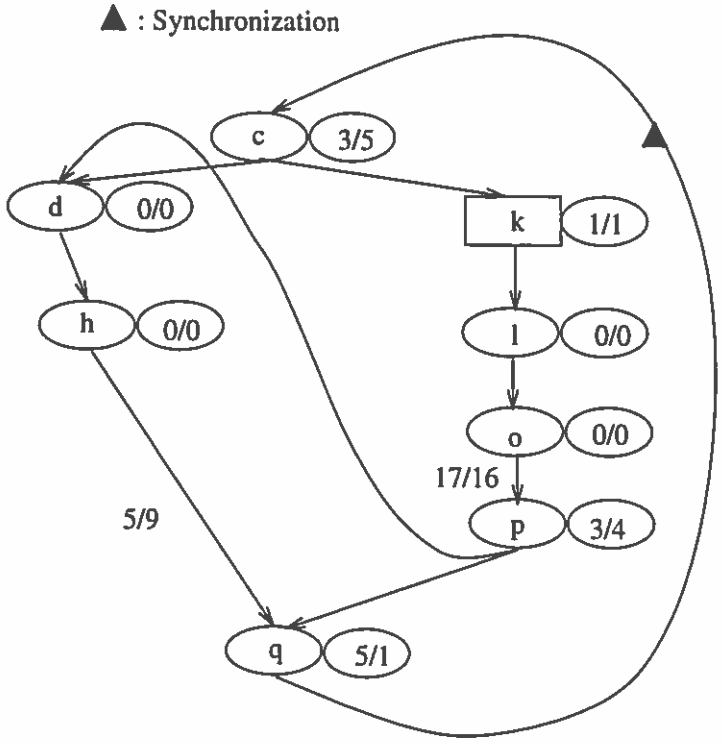


Figure 82: After the insertion of a synchronization on e

The last optimization process is to insert synchronization on r. It is easy to see it is legal and but not positive. The timing sensitivity of r is reduced from 7 to 0 but the timing sensitivity of the graph remains at 7. □

Optimization Through Lazy Synchronization

There are cases where synchronization can not be inserted since the insertion of synchronization is illegal, but the timing sensitivity can be reduced by delaying the program execution. Consider the example shown in Figure 83. Insertion of synchronization on e is not legal.

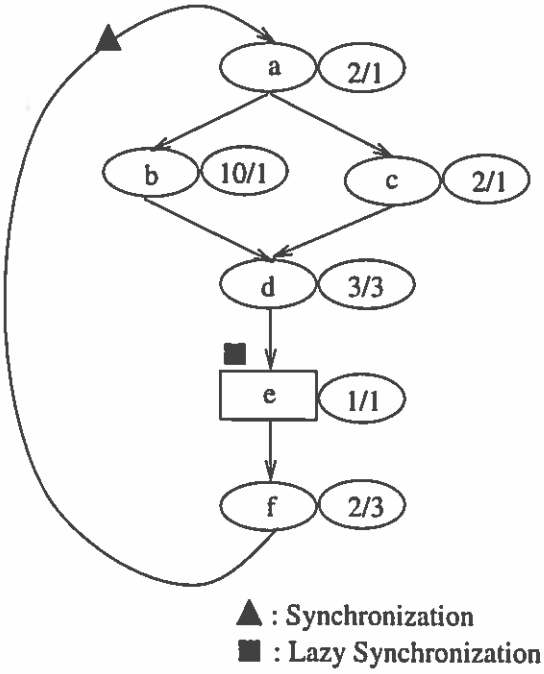


Figure 83: An example for lazy synchronization

To reduce timing sensitivity in such cases, we introduce another optimization method called *Lazy Synchronization*. Lazy synchronization reduces timing sensitivity by delaying program execution, but not completely synchronizing it.

Definition 8.4

The function $LazySync(dtime)$ at io delays program execution until

$$\begin{cases} LASTime(h) = LATTime(h) - dtime & dtime < 0, \\ LASTime(h) = LATTime(h) & otherwise, \end{cases} \quad \text{(VIII.68)}$$

where $dtime$ is $MaxSTD(h, io)$.

□

The function $LazySync(dtime)$ also can be implemented using $Delay()$ as in Figure 84.

```
LazySync(dtime);
{
    if (LATTime(h) + dtime < LASTime(h)) {
        Delay(LASTime(h) − LATTime(h) + dtime)
    }
}
```

Figure 84: Algorithm for $LazySync()$

Timing sensitivity of the control flow graph shown in Figure 83 can be reduced from 10 to 2 by inserting $LazySync()$ on e. In general, three cases occur after the insertion of a lazy synchronization in a graph as shown in Figure 85. In case 1, $MinSTD(h, v)$ becomes equal to $MaxSTD(h, v)$. In case 2, $MinSTD(h, v)$ becomes zero. In case 3, no execution delays.

Lemma 8.1

An insertion of lazy synchronization in a control flow graph is always legal

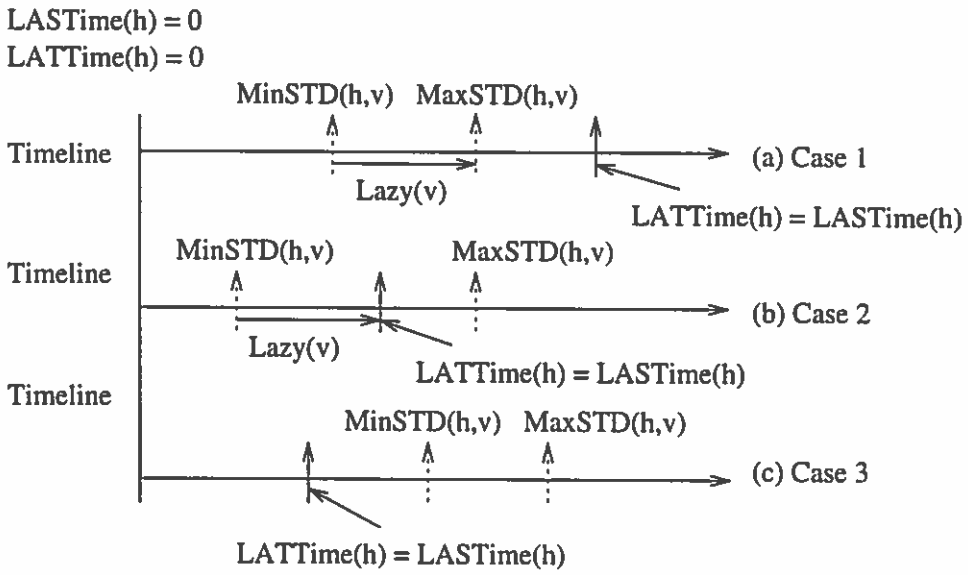


Figure 85: Timing sensitivity with lazy synchronization

and positive. □

Proof

It is easy to see any insertion of a lazy synchronization in the graph is legal since it can never change $MaxSTD(h, bn)$ for all $h \in H(G_M)$. The absolute timing sensitivity of a graph is defined with $\mid MinD(io) \mid$ and $\mid MaxD(io) \mid$. Thus, the absolute timing sensitivity of the graph can be reduced if we reduce the value of $\mid MinD(io) \mid$ or $\mid MaxD(io) \mid$. Considering the three cases shown in Figure 85, the value of $\mid MinD(io) \mid$ is always reduced by inserting $LazySync()$. □

Thus, we insert lazy synchronization for every $io \in IO(G_M)$ and all incoming edges of all intervals in the graph. Also, even lazier synchronization may be possible

by delaying even further. The local synchronization scheme works as long as all instances of synchronizations on back edges are valid. Thus, the execution on i/o nodes and incoming edges can be delayed even further as long as it it legal.

The timing sensitivity of the graph, after insertion of lazy synchronization can be found by using the same algorithm given in Figure 62 with modified data-flow equations. The modification of the data-flow equations to support lazy synchronization is trivial.

Summary

This chapter presented two algorithms which reduce the timing sensitivity of a given control flow graph. One works by inserting additional synchronization and the other by inserting lazy synchronization. An insertion of a synchronization in a graph is not always legal. Even a legal insertion of synchronization does not always improve the timing sensitivity (positive). We provide an algorithm to test if the insertion of synchronization is legal and positive. When an insertion of synchronization is not legal, we use lazy synchronization. An insertion of lazy synchronization is always legal and positive. Both synchronization and lazy synchronization use program execution delay on the target machine.

CHAPTER IX

IMPLEMENTATION ISSUES

This chapter discusses a number of issues regarding the actual implementation of algorithms discussed in this thesis.

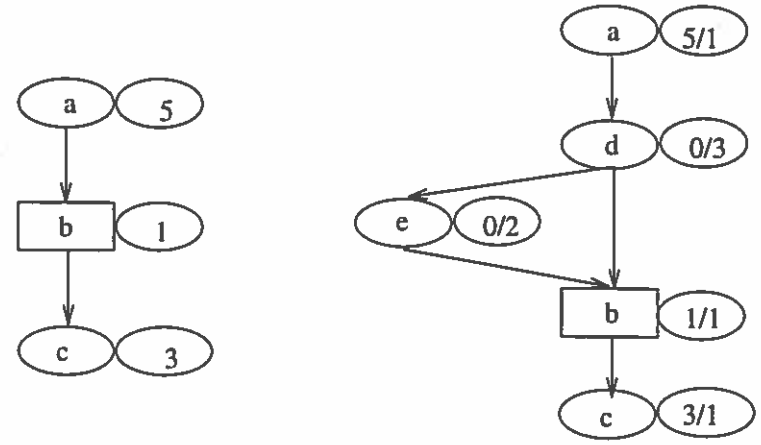
Engineering Problems

Implementation of algorithms described in this thesis is straight forward. Since timing analysis and optimization are performed by a binary-to-binary translator, we assume the control flow graph described in Chapter III is already constructed. We also assume that the execution time required on the source and target machines is computable. Implementation of the algorithm that reduces a reducible graph is described in [18]. The algorithms for timing sensitivity analysis, described in Chapter VII, and timing optimization, described in Chapter VIII, are straight forward. Here, we discuss a number of engineering problems.

Control Flow Graph

When translating a binary code to another binary code, we have to deal with architectural differences between the two machines, source and target. To deal with these architectural differences, the translator may insert conditional codes on the target binary program which change the control flow graph. Thus, we may have to deal with control flow graph differences between the source and target binary

programs. This problem can be dealt with by assigning zero execution time for source machine for those nodes. If a node in the target control flow graph does not exist on the source control flow graph its required execution time on the source machine is zero. Figure 86 shows such an example. In the figure, (a) shows the control flow graph of a source program with required execution time on the source machine and (b) shows the control flow graph of the translated target program with required execution time on both source and target machines. In the target control flow graph, d and e are added.



(a) Source CFG (b) Target CFG

Figure 86: Source and target control flow graphs

Clocks and Multiple Clocks

With the local synchronization scheme, conceptually many local clocks are maintained for inner-intervals. The maintenance cost for multiple clocks is unnecessarily high. In the implementation, however, we only need one global clock throughout the system. Each interval maintains its local clocks to find local timing error. Local tim-

ing error can be computed if we save the timing difference between two global clocks at the header node of each interval.

Consider a control flow graph shown in Figure 87. Each interval has its own local clocks and these clocks are updated at each node. When entering the interval $I(b)$, the timing difference between the two global clock is $LASTime(a) - LATTime(a)$ = 4. On the back edge $\langle c, b \rangle$, $LASTime(a) - LATTime(a)$ = 9. If we save the timing difference at the header, then the local timing error, which is 9-4 = 5, can be computed.

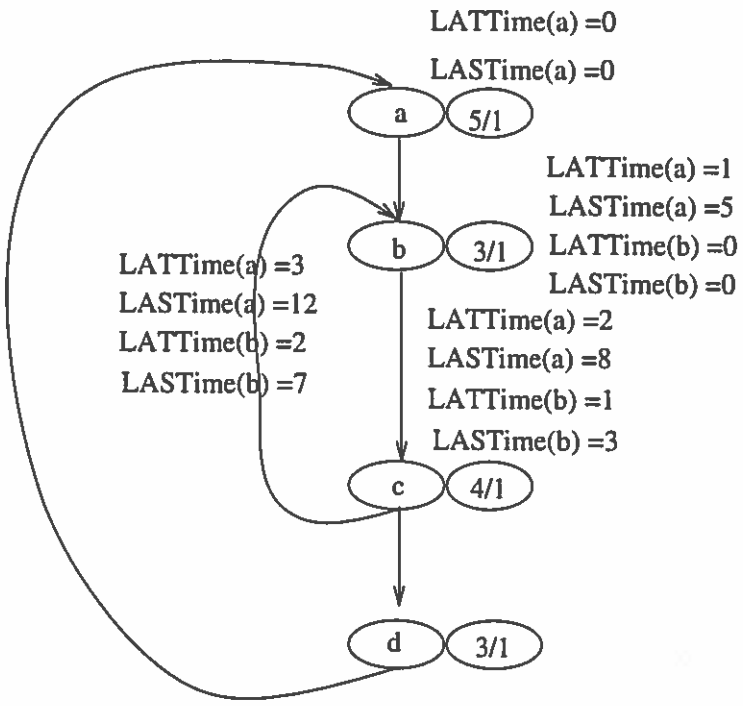


Figure 87: Multiple local clocks

In the real system, it is not necessary to maintain the target time ($LATTime$). The computation of target time is necessary only for timing analysis and optimization. Once the timing analysis and optimization are performed, target execution time can

be measured directly from the target system's clock.

Dealing With System Overhead

To enforce timing equivalence or invariance, the compiler must insert additional codes that maintain clocks and that delays the program execution. This additional code changes the required execution time for each node. However, these additional codes are a constant factor for each node. Timing analysis and optimization must be performed with consideration for these updated execution times.

Non-Engineering Problems

Tight Worst Case Execution Time

For our timing analysis and optimization, we assumed that the execution times required for each basic block on both source and target machines are given. However, it is quite difficult to compute the exact execution time for a given set of instructions on a machine with pipelines and cache. The minimum requirement to apply our results is that the exact source timing for each basic block must be available. Fortunately, for most older machines, such as the MC-68000, the execution time for each instruction is a constant and its execution time is known. If the exact execution time for each basic block is not computable on the target machine, we must use worst-case execution time. Issues in computing tight worst-case execution time are discussed in [40, 19, 60].

Environmental Changes

There are also a number of issues in dealing with the changes in the environment, such as the network and disk, where different devices may have different degrees of

speed improvement. In this thesis, we dealt with problems which happen when the CPU is replaced with a faster one. If the environments are changed, other problems may happen. For example, if data from a disk arrives too quickly and the CPU is not ready to receive that particular data, the result may be not expected. To solve this problem, we need a hardware version of TIBBIT that provides timing equivalence for environmental behaviors such as disk and network accesses.

CHAPTER X

SUMMARY AND FUTURE WORK

This chapter states the significance of our research problem, summarizes our contributions, and discusses future research directions.

Significance of The Problem

Typical real-time systems include nuclear reactors, flight-control systems, and process control systems for chemical plants. For such applications, verification of correctness of the program is particularly important since a failure may result in catastrophic destruction of property and life. The correctness of a real-time system depends not only on the output of the system, but also *when* the output is produced. Assuming the correctness of a real-time program is verified, binary-to-binary translation of such a program must provide timing equivalence as well as semantic equivalence. Failure in providing either of these equivalences may result catastrophic destruction.

The TIBBIT project addressed this problem, but the timing equivalence of programs generated by the TIBBIT system is not guaranteed. This thesis provides a method to guarantee the timing equivalence of generated target binary programs.

Summary of Contributions

We studied two important issues in binary-to-binary translation of real-time programs. The first issue is how to verify if a source binary program can be translated with timing equivalence or invariance for a given target machine. The second issue is how to reduce the timing sensitivity, the maximum timing error, on the target machine.

We assumed a binary-to-binary translation is performed. Issues in translation of binary programs are discussed elsewhere [51, 5, 50, 13].

We defined two timing sensitivities, absolute and relative, of target programs in Chapter IV. These two timing sensitivities were used to judge how closely the target program mimics the source program's timing.

Using the absolute timing sensitivity, we provided a taxonomy:

- A translated target program is timing equivalent if for any execution of the target program, the absolute timing sensitivity is zero.
- A translated target program is timing invariant if for any execution of the target program, absolute timing sensitivity is bounded by a constant.
- A translated target program is timing divergent if the absolute timing sensitivity is not bounded by a constant.

A target program is executable with timing equivalence if there exists a synchronization method that makes the target program a timing equivalent one, and with timing invariance if there exists a synchronization method that makes the target program a timing invariant one.

In Chapter V, we provided a method that examines if the target program is executable with timing equivalence or invariance with respect to the source program.

We proved that a target program represented by a graph G is executable with timing invariance if and only if all repeatable paths in G takes less time on the target machine. We also proved that a target program represented by a graph G is executable with timing invariance if and only if the graph G is executable with timing invariance and all simple paths between two i/o nodes in G takes less time on the target machine. We used interval analysis to find all simple cycles in the graph G. To examine if every simple cycle in G takes less time on the target machine, we used a data-flow approach with intervals.

If the graph G is executable with timing equivalence, inserting a synchronization on every i/o node provides timing equivalence. If the graph G is executable with timing invariance but not with timing equivalence, the timing sensitivity, the maximum timing error, is dependent on how and where the graph is synchronized. Chapter VI determines how to synchronize and Chapter VIII determines where to synchronize the graph G.

In Chapter VI, we presented two synchronization schemes: global and local. Both schemes have problems, over-commitment and under commitment, respectively. However, the under commitment problem can be overcome by inserting additional synchronization as seen in Chapter VIII.

If the target program after the addition of synchronization is timing invariant but not equivalent, the maximum timing error, timing sensitivity, has to be found so that the user can judge how closely the target program mimics the source program. Chapter VII presented methods that find the maximum timing error, the absolute

and relative timing sensitivity. We again used a data-flow approach to find both timing sensitivities.

If the timing sensitivity of the target binary program is not under the desired or required level, it is necessary to optimize the timing sensitivity. In Chapter VIII, we presented optimization techniques to minimize timing sensitivity by inserting additional synchronizations.

In summary, we:

- developed a new method to analyze the timing equivalence properties of BBT programs and provided necessary and sufficient conditions for timing equivalent and invariant translations.
- identified the timing sensitivities that can be used to judge the quality of the generated target program.
- developed timing sensitivities prediction and measuring models for target programs.
- developed a new synchronization based method for increased predictability
- developed criteria and techniques for achieving timing equivalence.

In this thesis, we created the theory and practical algorithms to give timing assurance for the process of real-time binary-to-binary translators. With the theory and algorithms, we make real-time binary-to-binary translation not only a practical method but one supported by a sound theoretical base. Since we used an intermediate representation (control flow graph), our theory and algorithms can be used for any platform. Thus, we open the way for translation with timing properties from any form to any other.

Future Work

In this thesis, we assumed the input is a binary executable program. Our results also can be applied when the input is a program written in a higher-level language. In a real-time program, timing requirements are expressed either implicitly as with binary executable programs or explicitly in programs written in a higher-level language. The duty of the real-time translator is to provide timing equivalence, as well as semantic equivalence, on the generated target program with respect to the requirements expressed in the source program. Our results can be applied in translation of programs written in a higher-level language.

Another problem for future research is the application of ordinary compiler optimization techniques to provide timing equivalence or invariance on a timing divergent target program. The motivation of optimization in this case is different from that of ordinary compiler optimizations (minimizing timing difference versus minimizing the total execution time).

BIBLIOGRAPHY

[1] AHO, A., SETHI, R., AND ULLMAN, J. *Compilers-Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] ALLEN, F. Control flow analysis. *sigplan 5*, 7 (July 1970), 1–19.

[3] ALLEN, F., AND COCKE, J. A catalog of optimizing transformations. In *Design and Optimization of Compilers*, R. Rustin, Ed. Prentice-Hall, 1971, pp. 1–30.

[4] ALLEN, F., AND COCKE, J. A program data flow analysis procedure. *Communications of the ACM 19* (1976), 137–147.

[5] ANDREWS, K., AND SAND, D. Migrating a CICS computer family onto RISC via object code translation. *5th International Conference on Architectural Support for Programming Languages and Operating Systems* (1992), 213–222.

[6] AUERNHEIMER, B., AND KEMMERER, R. RT-ASLAN: A specification language for real-time systems. *IEEE Transactions on Software Engineering 12*, 9 (Sept. 1986), 879–889.

[7] BERGH, A., MAGENHEIMER, K., AND MILLER, J. HP3000 emulation on HP Precision Architecture computers. *Hewlett-Packard Journal* (Dec. 1987).

[8] BREUER, P., AND BOWEN, J. Decompilation: The enumeration of types and grammars. *ACM Transactions on Programming Languages and Systems 16*, 5 (Sept. 1994), 1613–1647.

[9] CHANG, P., MAHLKE, S., CHEN, W., WARTER, N., AND HWU, W. Impact: An architectural framework for multiple-instruction-issue processors. In *18th International Symposium on Computer Architecture* (1991), pp. 266–275.

[10] CIFUENTES, C. Reverse compilation techniques. Ph.D. dissertation, Queensland University of Technology, School of Computing Science, July 1994.

[11] CIFUENTES, C., AND GOUGH, K. Decompilation of binary programs. *Software: Practice & Experience 25*, 7 (1995), 811–829.

[12] COCKE, J., AND KENNEDY, K. An algorithm for reduction of operator strength. *Communications of the ACM 20*, 11 (1977), 850–856.

[13] COGSWELL, B. Timing insensitive binary-to-binary translation. Ph.D. dissertation, Carnegie Mellon University, Dept. Electrical and Computer Engineering, Apr. 1995.

[14] COGSWELL, B., AND SEGALL, Z. Timing insensitive binary to binary translation. In *Workshop on Architectures for Real-Time Applications* (1994).

[15] DASARATHY, B. Timing constraints of real-time systems: Constructs for expressing them, methods for validating them. *IEEE Transactions on Software Engineering 11*, 1 (Jan. 1985), 80–86.

[16] GERBER, R., AND HONG, S. Semantics-based compiler transformations for enhanced schedulability. In *Proceedings of IEEE Real-Time Symposium* (Dec. 1993), IEEE Society Press, pp. 232–242.

[17] GERBER, R., AND HONG, S. Compiling real-time programs with timing constraint refinement and structural code motion. *IEEE Transactions on Software Engineering 21*, 5 (May 1995), 389–404.

[18] GRAHAM, S., AND WEGMAN, M. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM 23*, 1 (Jan. 1976), 172–202.

[19] HARMON, M., BAKER, T., AND WHALLEY, D. A retargetable technique for predicting execution time of code segments. *Real-time Systems 7* (1994), 159–182.

[20] HECHT, M. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

[21] HECHT, M., AND ULLMAN, J. Analysis of a simple algorithm for global flow problems. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Oct. 1973), pp. 207–217.

[22] HECHT, M., AND ULLMAN, J. Characterization of reducible flow graphs. *Journal of the ACM 21*, 3 (Mar. 1974), 367–375.

[23] HECHT, M., AND ULLMAN, J. A simple algorithm for global data flow analysis problems. *SIAM Journal of Computing 4*, 4 (Dec. 1975), 519–532.

[24] HONG, S., AND GERBER, R. Compiling real-time programs into schedulable code. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation* (1993), pp. 166–176.

[25] HUNTER, C., AND BANNING, J. DOS at RISC. *Byte Magazine* (Nov. 1989), 361–368.

[26] JAHANIAN, F., AND MOK, A. K. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering 12*, 9 (Sept. 1986), 890–904.

[27] JAHANIAN, F., AND MOK, A. K. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering 20*, 12 (Dec. 1994), 933–9947.

[28] KAM, J. B., AND ULLMAN, J. D. Global data flow analysis and iterative algorithms. *Journal of the ACM 23*, 1 (Jan. 1976), 158–171.

[29] KAM, J. B., AND ULLMAN, J. D. Monotone data flow analysis frameworks. *Acta Informatica 7* (1977), 305–317.

[30] KENNEDY, K. A survey of data flow analysis techniques. In *Program Flow Analysis*, S. Muchnick and N. Jones, Eds. Prentice-Hall, 1981, pp. 5–54.

[31] KENNY, K., AND LIN, K. Building flexible real-time systems using the flex language. *IEEE Computer* (May 1991), 70–78.

[32] KILDALL, G. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Oct. 1973), pp. 194–206.

[33] KLIGERMAN, E., AND STOYENKO, A. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering 12*, 9 (Sept. 1986), 941–949.

[34] KNOOP, J., RUTHING, O., AND STEFFEN, B. Lazy code motion. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (1992), pp. 224–234.

[35] KNOOP, J., RUTHING, O., AND STEFFEN, B. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems 16*, 4 (July 1994), 1117–1155.

[36] KOEPETZ, H., AND OCHSENREITER, W. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers 36*, 8 (Aug. 1987), 933–940.

[37] KRONENBERG, N., BENSON, T., CARDOZA, W., JAGANNATHAN, R., AND THOMAS, B. Porting OpenVMS from VAX to Alaph AXP. *Communications of the ACM 36*, 2 (Feb. 1993), 45–53.

[38] LENGAUER, T., AND TARJAN, R. E. A fast algorithm for finding dominators in a flow graph. *ACM Transactions on Programming Languages and Systems 1*, 1 (July 1979), 121–141.

[39] LEWIS, T., EL-REWINI, H., AND WITH I. KIM. *Introduction To Parallel Computing.* Prentice-Hall, 1992.

[40] LIM, S., BAE, Y., JANG, G., RHEE, B., MIN, S., PARK, C., SHIN, H., PARK, K., AND KIM, C. An accurate worst case timing analysis technique for RISC processors. In *Proceedings of IEEE Real-Time System Symposium* (1994).

[41] LORRY, E., AND MEDOCK, C. Object code optimization. *Communications of the ACM 12* (1969), 13–22.

[42] LOVEMAN, D. B. Program improvement by source-to-source transformation. *Journal of the ACM 20*, 1 (Jan. 1977), 121–145.

[43] MARLOWE, T., AND RYDER, B. Properties of data flow frameworks. *Acta Informatica 28* (1990), 121–163.

[44] MOREL, E., AND RENVOISE, C. Global optimization by suppression of partial redundancies. *Communications of the ACM 22* (1979), 96–103.

[45] PARK, C., AND SHAW, A. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer* (May 1991), 48–57.

[46] PAVEY, D., AND WINSBORROW, L. Demonstrating equivalence of source code and PROM contents. *The Computer Journal 36*, 7 (1993), 654–667.

[47] REIF, J., AND LEWIS, H. Efficient symbolic analysis of programs. *Journal of Computer and System Sciences* (June 1986), 280–313.

[48] RYDER, B., AND PAUL, M. Elimination algorithms for data flow analysis. *ACM Computing Surveys 18*, 3 (Sept. 1986), 277–316.

[49] SHAW, A. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering 15*, 7 (July 1989), 875–889.

[50] SILBERMAN, G., AND EBCIOGLU, K. An architectural framework for supporting heterogeneous instruction-set architectures. *IEEE Computer* (June 1993), 39–56.

[51] SITES, R., CHERNOFF, A., KIRK, M., MARKS, M., AND ROBINSON, S. Binary translation. *Communications of the ACM 36*, 2 (Feb. 1993), 69–81.

[52] SPECTOR, A., AND GIFFORD, D. Case study: The space shuttle primary computer system. *Communications of the ACM 27*, 9 (1984), 872–900.

[53] TARJAN, R. E. Depth-first search and linear graph algorithms. *SIAM Journal of Computing 1*, 2 (June 1972), 146–160.

[54] TARJAN, R. E. Finding dominators in directed graphs,. *SIAM Journal of Computing 3*, 1 (Jan. 1974), 62–89.

[55] TARJAN, R. E. Fast algorithms for solving path problems. *Journal of the ACM 28*, 3 (July 1981), 594–614.

[56] ULLMAN, J. D. Fast algorithms for the elimination of common subexpressions. *Acta Informatica 2* (1973), 191–213.

[57] WEGMAN, M., AND ZADECK, F. Constant propagation with conditional branches. In *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages* (Jan. 1985), pp. 291–299.

[58] WEGMAN, M., AND ZADECK, F. Constant propagation with conditional branches. *ACM Transaction on Programming Languages and Systems* (Apr. 1991), 181–210.

[59] WOLFE, M. Beyond induction variables. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (1992), pp. 162–174.

[60] ZHANG, N., BURNS, A., AND NICHOLSON, M. Pipelined processors and worst case execution times. *Real-time Systems 5* (1993), 319–343.