Abstract

Title of Dissertation: Compiler-Assisted Scheduling for Real-Time Applications: A Static Alternative to Low-Level Tuning

Seongsoo Hong, Doctor of Philosophy, 1994

Dissertation directed by: Assistant Professor Richard Gerber
Department of Computer Science

Developing a real-time system requires finding a balance between the timing constraints and the functional requirements. Achieving this balance often requires last-minute, low-level intervention in the code modules – via intensive hardware-based instrumentation and manual program optimizations. In this dissertation we present an automated, static alternative to this kind of human-intensive work. Our approach is motivated by recent advances in compiler technologies, which we extend to two specific issues on real-time programming, that is, feasibility and schedulability.

A task is infeasible if its execution time stretches over its deadline. To eliminate such faults, we have developed a synthesis method that (1) inspects all infeasible paths, and then (2) moves instructions out of those paths to shorten the execution time.

On the other hand, schedulability of a task set denotes an ability to guarantee the deadlines of all tasks in the application. This property is affected by interactions between the tasks, as well as their individual execution times and deadlines. To address the schedulability problem, we have developed a task transformation method based on program slicing. The method decomposes a task into two subthreads: the IO-handler component that must meet the original deadline, and the state-update component that can be postponed past the deadline. This delayed-deadline approach contributes to the schedulability of the overall application. We also present a new fixed-priority preemptive scheduling strategy, which yields both a feasible priority ordering and a feasible task-slicing metric.

Compiler-Assisted Scheduling for Real-Time Applications: A Static Alternative to Low-Level Tuning

by

Seongsoo Hong

Dissertation submitted to the Faculty of the Graduate School
of The University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1994

Advisory Committee:

Assistant Professor Richard Gerber, Chair/Advisor
Professor Ashok K. Agrawala
Professor John Gannon
Professor Virgil Gligor
Associate Professor William Pugh
Associate Professor Udaya A. Shankar

Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Motivation 3

1.2 Compiler Transformations 6

1.2.1 Feasible Code Synthesis 6

1.2.2 Real-Time Task Slicing 7

1.3 Summary of Contributions 8

1.4 Outline of the Dissertation 9

2 Related Work 10

2.1 Formal Methods 10

2.2 Real-Time Programming Languages 11

2.3 Real-Time Compiler Tools 12

3 The Language of Time-Constrained Events 15

3.1 Design Goals 15

3.2 Timing Constructs and Their Semantics 16

3.3 Example TCEL Code 18

4 Basic Notations **21**

4.1 Flow Graphs 21

4.2 Basic Compiler Definitions 23

5 Transformation 1: Feasible Code Synthesis **27**

5.1 The Problem and Our Solution 28

5.1.1 The Problem of Feasible Code Synthesis 29

5.1.2 Solution Strategy 32

5.2 Section Decomposition 33

5.2.1 Determining Section Boundaries 33

5.2.2 Deriving Code-Based Timing Constraints 35

5.3 Code Scheduling 38

5.3.1 The Top-Level Algorithm 39

5.3.2 Subroutine `Schedule_Section`(S,D,$DUR(\mathbf{S})$,V_{bar}) 40

5.4 Summary 49

6 Transformation 2: Real-Time Task Slicing **50**

6.1 Background 51

6.1.1 Characterization of Discrete Control Software 52

6.1.2 Fixed-Priority Preemptive Scheduling 54

6.1.3 Scheduling with Compiler Transformations 56

6.2 Automatic Task Decomposition by Program Slicing 59

6.2.1 The Program Slicing Algorithm 60

6.2.2 Assigning Times to Subtasks 62

6.3 Scheduling Alternatives and Their Analyses 64

6.4 Priority Ordering with Task Slicing 67

6.4.1 Feasibility Test 68

6.4.2 The Algorithm 68

6.4.3 A Larger Example 69

6.5 Summary 70

7 Practical Considerations and Prototype Implementation **73**

7.1 Practical Considerations. 73

7.1.1 Limits of Data-Flow Analysis 74

7.1.2 Limits of Timing Analysis 74

7.1.3 User Interaction 75

7.2 TimeWare/SLICE: the Prototype Implementation 75

7.2.1 TimeWare/SLICE Tool Screens 76

7.2.2 Commands 76

7.2.3 Implementation 79

7.3 Summary 80

8 Conclusions **81**

8.1 Future Directions 83

List of Tables

5.1 Timing Constraints of S3 and S4 38

6.1 Example Task Set 71

6.2 Priority Assignment with Program Slicing 72

List of Figures

1.1 Software Development Process: Two Alternatives 2

1.2 Event-Based Specification of Sensor-Controller System 4

1.3 Two TCEL Programs with the Same Semantics 5

1.4 Run-Time Behavior of a TCEL Periodic Task 8

3.1 Flow Graph of **do** Construct 17

3.2 Behavior of Periodic Timing Construct 18

3.3 (A) TCEL Source Code for the Flight Controller and (B) Corresponding Flow Graph 20

5.1 Software Development Process: Revisited for Feasible Code Synthesis 28

5.2 Flow Graph of **do** Construct and its Section Division 34

5.3 Flight Controller Program: After Section Decomposition 36

5.4 Top-Level Algorithm for Code Synthesis 39

5.5 (A) Source Code in TCEL, (B) Corresponding Intermediate Code in GSA Form, (C) Intermediate Code after Bookkeeping-Free Transformations, and (D) Intermediate Code after Transformations with Bookkeeping 41

5.6 Speculative Code Motion: (A) Original Code, (B) Corresponding GSA Code and (C) Speculatively Transformed Code 43

5.7 The Section Scheduling Algorithm (*Schedule_Section*) 46

5.8 The Section Scheduling Algorithm (*Sched*) 47

5.9 Flight Controller Program: After Code Scheduling 48

6.1 Software Development Process: Revisited for Real-Time Task Slicing 51

6.2 Generalized Slicing Method for Schedulability Tuning 52

6.3 Task Instance at the k^{th} Period 53

6.4 Dynamic Behavior of Periodically Implemented Control-Loop 53

6.5 TCEL Program for Task τ_2 56

6.6 Simulated Time Line for the Example Task Set 57

6.7 Decomposed Task at the k^{th} Period 58

6.8 Two Decomposed Subtasks of Task τ_2 58

6.9 Program Dependence Graph 61

6.10 Slice with respect to Criterion $\langle \mathrm{L9}, \mathtt{cmd} \rangle$ 62

6.11 Slicing a Conditional 63

6.12 Algorithm for Priority Ordering with Slicing Decision 70

7.1 Tool Screens of TimeWare/SLICE 77

7.2 Output of TimeWare/SLICE 78

8.1 Software Development Process: Two Alternatives 82

Chapter 1

Introduction

A real-time application is characterized by the existence of two competing factors: its functional specification and its temporal requirements. Functional specifications define valid translations from inputs into outputs. As such they are realized by a set of programs, which *consume* CPU time. Temporal requirements, on the other hand, place upper and lower bounds between *occurrences of events* [9, 24]. An example is *the robot arm must receive a next-position update every 10ms.* Such a constraint arises from the system's requirements, or from a detailed analysis of the application environment. Temporal requirements implicitly *limit* the time that can be provided by the system's resources.

Figure 1.1 illustrates the real-time software development process from a high-level requirements specification to a low-level implementation. First, high-level requirements are translated into real-time programs. Then the programs are compiled into real-time tasks, and timing constraints are imposed on these tasks. After the compilation, the programmers must assure that each of the tasks in the application will meet its timing constraints on the underlying hardware platform. This process is called a *feasibility test.*

The programmers must also assure that the whole application will adhere to the timing constraints, even in the presence of direct and indirect interactions among the tasks, such as task blocking and preemption. This process is called a *schedulability test.* If the programmers cannot guarantee either of the tests, the result is careful refinement of the implementation. As shown in Figure 1.1, we can think of two alternatives as a means for the low-level system refinement: system-tuning and compiler-based task optimization.

Low-level tuning of the application can be a time-consuming process. It often involves using expensive hardware monitors (e.g., in-circuit emulators, logic analyzers, etc.), manually counting instruction-cycle times, hand-optimizing the code, and experimenting with various orderings of

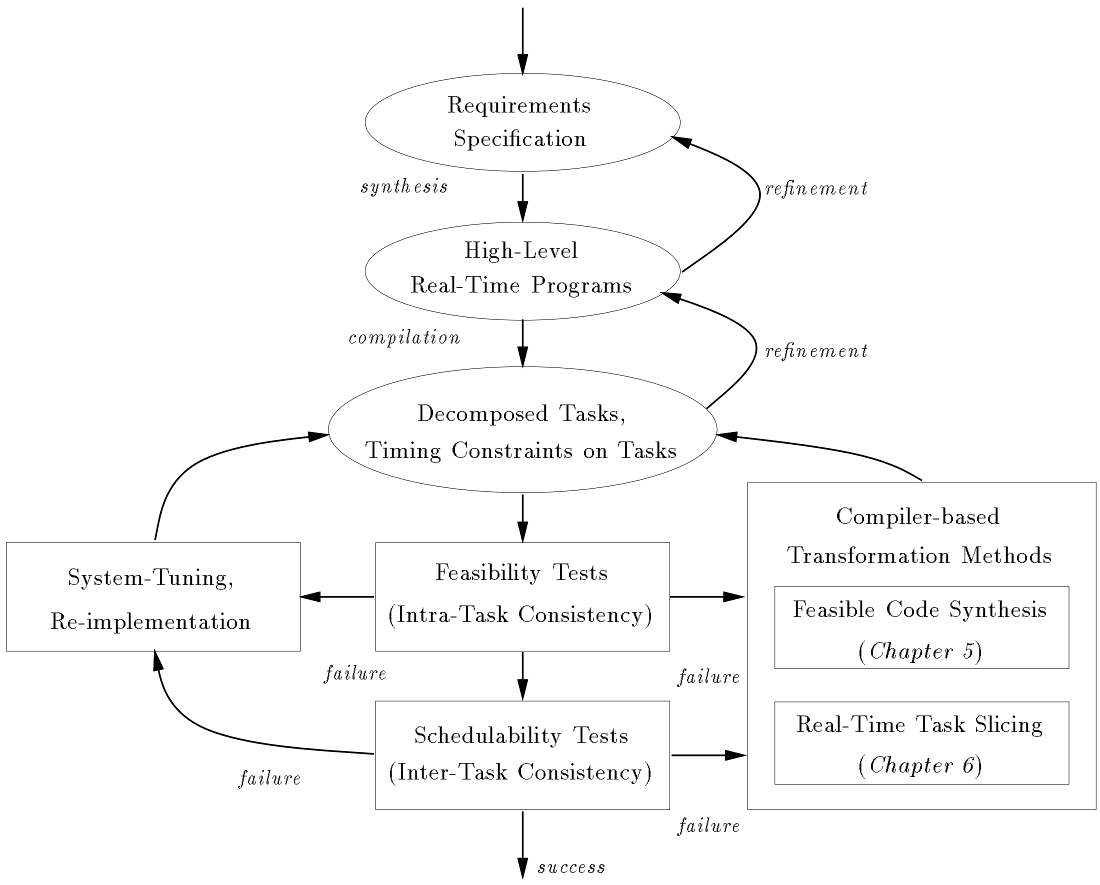

Figure 1.1: Software Development Process: Two Alternatives

operations. As a last resort, entire subsystems may have to be re-designed altogether.

The goal of this dissertation research is to provide programmers with a powerful alternative to low-level tuning. It is based on compiler-based transformation tools. Our approach consists of two inter-related components: a real-time programming language and static compiler transformation methods. The programming language not only provides a means for expressing timing constraints within a source program, but it also lays a foundation for sound compiler transformation methods. As shown in Figure 1.1, our compiler transformation methods include *feasible code synthesis* and *real-time task slicing*.

1.1 Motivation

Our approach is motivated by the success of static compiler transformation technologies in code parallelization and program debugging. It is a natural step to adopt these technologies to real-time domains, where developers lack powerful software engineering tools to apply at the tuning stage. This stage can consume a disproportionate amount of project's budget.

To carry out this approach, it is imperative to find a real-time programming language to provide high-level timing constructs with unambiguous semantics. This allows us to define a class of semantics-preserving transformations for them. Unfortunately, most existing real-time programming languages do not fulfill the requirement. We conceive this as a problem of "code-based specifications."

Consider experimental real-time programming languages which have been proposed in the literature [23, 27, 30, 33, 40]. They provide high-level real-time constructs such as "**within** 10ms **do** B," where the block of code "B" must be executed within a 10 millisecond time frame. These languages, while providing a convenient framework for *expressing* time in programs, have done little to ease the process of translating a real-time specification into schedulable code.

The reason is straightforward: Language constructs such as "**within** 10ms **do** B" establish constraints on *blocks of code.* However, "true" real-time properties establish constraints between the *occurrences of events* [9, 24]. While language-based constraints are very sensitive to a program's execution time, specification-based constraints must be maintained regardless of the platform's CPU characteristics, memory cycle times, bus arbitration delays, etc.

Our approach is to treat a real-time program as (1) an event-based timing specification, which represents the system's real-time requirements; and (2) a functional implementation, that is, the system's code. We carry out this approach with a real-time *syntax* quite similar to those found in the abovementioned languages. However, in our approach the interpretation is quite different. Instead of constraining *blocks of code*, the timing constructs establish constraints between the *observable events* within the code.

The language we propose is called Time-Constrained Event Language (TCEL). As an example of TCEL code, consider the following specification fragment, which is rendered pictorially in Figure 1.2:

(1) Every 25ms, an external sensor sends a message to the controller, containing physical world measurement data.

(2) The controller must receive every message.

(3) Using the sensor data and the current state, the controller computes a next-position command and sends it to an actuator.

(4) To achieve steady state, transmission of `cmd` is made no earlier than 3.5ms after receipt of `data`.

(5) To guarantee response-time threshold, transmission of `cmd` is made no later than 4.0ms after receipt of `data`.

(6) Based on the sensor-input, the controller updates its current state.

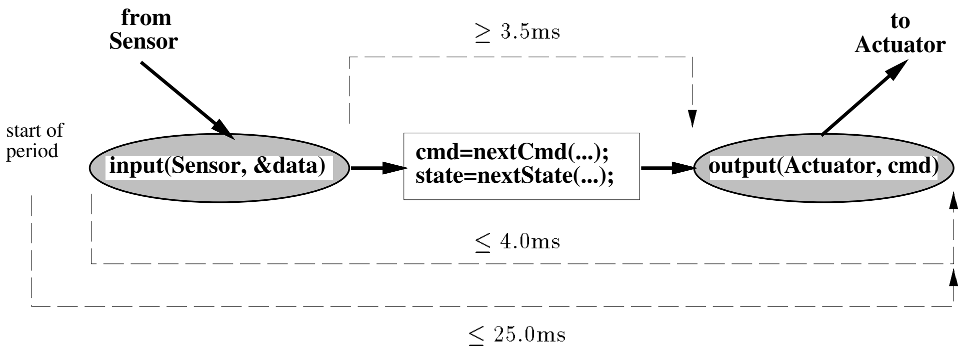

Figure 1.2: Event-Based Specification of Sensor-Controller System

Within our event-based framework, the program fragments in Figure 1.3 realize the specification. The system's only observable events occur instantaneously during the executions of the "**output**" and "**input**" operations. The "**do**" statement establishes timing constraints only between these two operations. On the other hand, the local statement "`cmd = nextCmd(data, state)`" is only constrained by the program's natural control and data dependences.

Armed with this interpretation, our compiler treats both programs as having equivalent semantics. This is quite different from the approaches mentioned above, where timing constructs establish constraints on code. In that interpretation, program A would first receive its data, then delay for 3.5ms and finally, evaluate `nextCmd` and send the result within the remaining 0.5ms. Program B would receive its data, evaluate `nextCmd`, *then* delay for 3.5ms and finally, send the result *within 4.0ms of evaluating* `nextCmd`.

Both programs may fail to implement the specification on some hardware platforms. If `nextCmd` is a CPU-intensive function (and thereby requires over 0.5ms of execution time), program A is

```
/* Program A */
every 25ms
  do
    input(Sensor, &data);
  start after 3.5ms finish within 4.0ms
  {
    cmd = nextCmd(state, data);
    state = nextState(state, data);
    output(Actuator, cmd);
  }
```

```
/* Program B */
every 25ms
  do
  {
    input(Sensor, &data);
    cmd = nextCmd(state, data);
  }
  start after 3.5ms finish within 4.0ms
  {
    state = nextState(state, data);
    output(Actuator, cmd);
  }
```

Figure 1.3: Two TCEL Programs with the Same Semantics

inherently unschedulable. On the other hand, program B establishes a constraint between the evaluation of `nextCmd` and the `nextState`, and not between the two specified events. Both programs would have to be rewritten to achieve the desired effect. The necessary corrections would include manually decomposing `nextCmd`, as well as adjusting the timing constraints. The actual changes would heavily depend on the particular characteristics of the computer, and thus, the very reason for using high-level timing constructs would be defeated.

There are several immediate benefits to our semantics for real-time constructs. First, a source program is not hardware-specific, and thus maintains the abstract, "portable" spirit of a high-level language. Since the timing constraints refer only to specification-based events, they need not be platform-specific. Second, this decoupling of timing constraints from code blocks enables a more straightforward implementation of an event-based specification.

But of most importance, some of the arduous, assembly-language level hand-tuning can now be accomplished semi-automatically – by compiler optimization techniques. In this dissertation we present two of such techniques: one that relies on Trace Scheduling, and the other based on program slicing. Traditionally, Trace Scheduling has been used in instruction scheduling, etc., and program slicing has been used in program analysis and debugging. Here, the objective is different: to achieve guaranteed real-time performance. In doing this we use the observable events as "signposts," which constrain the places where code can be moved. These events, as well as data dependences, establish the limiting constraints for the optimization algorithm.

1.2 Compiler Transformations

Armed with the event-based semantics of TCEL, we have developed two compiler transformation techniques, which we call *feasible code synthesis* and *real-time task slicing.* The objective of feasible code synthesis is to achieve internal consistency between a task's execution time and its timing constraints specified with "**do**" constructs. On the other hand, the objective of real-time task slicing is to enhance inter-task schedulability. This transformation is applicable to "**every**" periodic constructs.

1.2.1 Feasible Code Synthesis

We call a code segment *infeasible* if its execution time stretches over its specified deadline. Feasible code synthesis localizes and corrects such a fault via a two-step process. First, the compiler decomposes a "**do**" construct into a set of code blocks according to the control structure, and then automatically derives a set of dispatch equations from the language-based timing constraints. This step is necessitated by a gap between the event-based semantics model of source programs and the code-based execution model of real-time schedulers. In fact, most real-time schedulers only accept timing constraints on the start and finish times of *tasks.* The compiler thereby satisfies the scheduler's requirements by transforming event-driven source programs into constrained blocks of code, and providing the dispatch equations for the scheduler.

Next, the compiler attempts to correct feasibility faults with respect to the derived dispatch equations. This is done by a variant of Trace Scheduling, in which worst-case execution time paths of the infeasible task are selected, and unobservable code is moved to shorten their execution time.

For simple illustration, suppose that `nextCmd` in Program A of Figure 1.3 executes longer than 0.5ms and hence Program A is infeasible. As a solution, "`nextCmd(data, state)`" can be inlined, and then decomposed into two parts – one which is flow-dependent on the parameters `data` and `state`, and another which is invariant of them. This second part can be lifted out of the **do** statement altogether. We elaborate on this transformation method in Chapter 5.

1.2.2 Real-Time Task Slicing

Real-time task slicing considers a more ambitious goal – inter-task transformations for schedulability. The effect of this transformation is global, though it is individually applied to a small number of tasks selected from an unschedulable application.

The key idea behind this method is based on a simple fact, that is, an application's schedulability improves if we increase the deadlines of its constituent tasks. The same effect is achieved by allowing a task to slide past its deadline, while maintaining the original event-based semantics. We can realize this benefit by transforming a task, so that its time-sensitive component always executes within its frame, while postponing the rest of the task.

To systematically carry out the transformation, we harness a novel application of *program slicing* [41, 52, 53]. An unschedulable task is decomposed into two subthreads: one that is "time-critical" and the other "unobservable." The unobservable subthread is then appended to the end of time-critical thread, with TCEL semantics being maintained. Figure 1.4 pictorially illustrates the net effect of this transformation. The downward (upward) arrow of the k^{th} frame represents the execution of input (output) event in the same time frame.

Since the goal of the transformation comes from the scheduler component of the real-time system, the framework consists of the following ingredients.

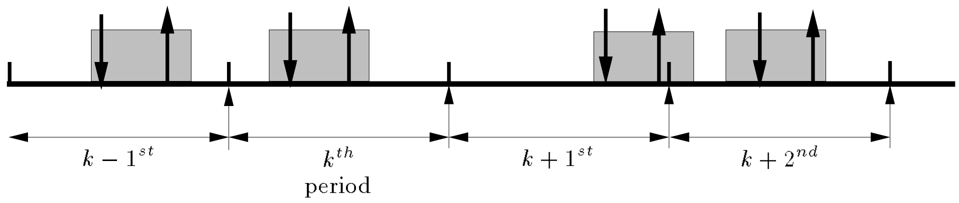

Figure 1.4: Run-Time Behavior of a TCEL Periodic Task

(1) An algorithm which uses standard fixed-priority preemptive scheduling analysis to find unschedulable tasks which need task slicing.

(2) A program slicer which decomposes a task and isolates the thread that can be postponed.

(3) An online component of the scheduler which enforces precedence constraints between task interactions.

In Chapter 6 we will discuss in detail the fixed-priority preemptive scheduling paradigm as well as the application of program slicing method.

1.3 Summary of Contributions

The major contributions of this dissertation is itemized as follows:

- The event-based abstraction is a widely used approach in the formal methods literature. We have extended it into a full-blown real-time programming language called TCEL. The event-based semantics of TCEL makes it straightforward to realize a high-level specification into a real-time program. This semantics allows a clear and unique interpretation of high-level timing constructs, and thus enables us to define safe compiler transformations for TCEL programs. Although TCEL has been implemented on top of the C programming language, it can be used with other programming languages such as Pascal or Fortran. It is a general, flexible mechanism for high-level timing annotations, and is not specific to a particular language.

- We have developed an automatic method of translating a high-level TCEL program into schedulable units of code. When implemented within a TCEL compiler, this method decomposes a TCEL program into tasks, and then derives code-based timing constraints from the language-based timing constraints. This method automatically refines the original timing constraints to account for the effects caused by programs' control structures.

- We have developed an automatic compiler method that is used to transform an infeasible task into a feasible one. This problem is an intractable one, as will be proved. Thus we have invented an approximation approach based on Trace Scheduling.

- We have developed another compiler transformation method that is used to enhance the real-time schedulability of an input task set. This method automatically decomposes a task so that the real-time scheduling method can guarantee timely execution of observable operations, while local operations need not lead to scheduling overrun.

- We have developed a priority ordering algorithm that not only assigns feasible priorities, but also selects a subset of tasks to be transformed. The guiding metric is schedulability.

1.4 Outline of the Dissertation

This dissertation is organized as follows. In Chapter 2 we survey background and related work. In Chapter 3 we introduce TCEL with an example program. In Chapter 4 we introduce basic notations for flow graphs and standard compiler technologies which are used throughout the dissertation. In Chapter 5 we present the first transformation method, *feasible code synthesis*, along with a motivating example. Then, in Chapter 6 we present the second transformation method, *real-time task slicing*, and we discuss its underlying theory of fixed-priority, preemptive scheduling. In Chapter 7 we consider the practical limitations of our compiler-based approach, and propose solutions to overcome these problems. We also briefly show TimeWare/SLICE, our prototype implementation of the real-time task slicer, which reflects our philosophy on tool-based system development. Finally, in Chapter 8 we conclude this dissertation with future research directions.

Chapter 2

Related Work

In this chapter we survey related work in real-time programming. First, we review some of formal methods that have laid the foundation for our event-based semantics. We also study real-time programming languages, since they have influenced the design of TCEL's timing constructs. Finally, we survey compiler-based real-time tools, and compare them with our approach.

2.1 Formal Methods

TCEL's semantics was inspired by a principle commonly applied in formal methods. That is, when reasoning about a real-time concurrent system it is often useful to consider only "events of interest," and to abstract away local-state information. Indeed, almost all formal models ease this process by making some distinction between an "event" and a corresponding "action." In this section we survey four such methodologies: Real-Time Logic [24], RTRL [9], Timed IO Automata [37] and ACSR [29].

RTL. Real-Time Temporal Logic possesses an underlying event-action model. It captures the temporal ordering between an application's actions and its events. An action in RTL is the execution of an operation which consumes a certain amount of system resources. The effects of actions are revealed by events that are generated before and after they execute. The occurrence of an event is defined to be instantaneous, while the execution of an action takes non-zero time. Thus a timing constraint is an assertion about temporal relationships between certain events. The conjunction of these timing assertions are "the system specification," and they must imply that key safety properties are maintained.

RTRL. RTRL (Real-Time Requirements Language) is a formal specification language developed for use in modeling telephone switching systems. In RTRL, a real-time system is viewed as a finite-state machine, in which a response at any instance is determined by the system's state and the external inputs. Hence the timing constraints are established between external inputs and responses. Unlike the events in RTL, however, these external inputs and responses denote signals (of a switching system) rather than points in time, and thus have duration times.

Timed I/O Automata. A timed I/O automaton is essentially a state transition system consisting of state variables and events, where each of the events has an enabling condition and an action. A timed I/O automaton has a set of timing assumptions, each of which specifies an event set and an interval during which the event set can be continuously enabled since its last occurrence. Thus the timing assumptions place constraints on the event-firing times [46]. A "behavioral abstraction" of a timed I/O automaton allows reasoning about only the event sequences, since local state information can be ignored in the abstraction.

ACSR. The computation model of ACSR (Algebra of Communicating Shared Resources) addresses two key issues in modeling a real-time system: concurrency and communication. An application specified in ACSR consists of a set of communicating processes that use shared resources for execution and synchronization with one another. A timed action in ACSR takes unit time to execute and consumes a set of resources during that time. Synchronization between actions is supported by events that are instantaneous, and consume no resources.

Impact of Formal Methods. Strongly motivated by the event-action models, we have extended the event-action abstraction into to a "full-blown" real-time programming language, in which the "events" correspond to actual IO operations within C code. A logical consequence of the event-action model is the ability to exploit this looser semantics, and to use compiler transformations to move unobservable instructions out of over-constrained code blocks.

2.2 Real-Time Programming Languages

Most other real-time languages do not make such a distinction, and instead place constraints on the boundaries of code blocks. Two paradigms are used in these languages: either constraints are expressed directly in the program itself (as in [33, 30, 54]), or they are postulated in a separate interface, and then passed to the scheduler as directives. A common language-based approach (first presented in [30]) is to provide constructs such as "**within** t **do** $\{\ldots\}$," "**at** t **do** $\{\ldots\}$" and "**after** t **do** $\{\ldots\}$." An alternative, taken in [33], is to set up linear constraint expressions on the the start times and deadlines of code blocks. We have borrowed from both approaches: in the TCEL source we use the higher-level constructs, while in our intermediate code we make use of the constraint representation. But in TCEL the semantics is quite different, as it establishes constraints between the *observable events* within the code, and not on the code's textual boundaries.

2.3 Real-Time Compiler Tools

There have been many other compiler-based approaches to real-time programming, most of which address different real-time programming problems and rely on different techniques. However, they share a common goal, namely, enhancing predictability and schedulability of real-time applications. In this section we survey tools, and then show where we can place our tools in the realm of real-time programming.

Schedulability Analyzer for Real-Time Euclid. Among the early approaches was the schedulability analyzer [49], specifically developed for the timing and scheduling analyses of programs written in Real-Time Euclid [27]. The schedulability analyzer consists of a front-end and a back-end. The front-end is incorporated into the code generator of the Real-Time Euclid compiler, and it produces a segment tree that represents compilation units such as modules, monitors or routines. Using a segment tree, the back-end computes the execution time profile of each segment considering synchronization, phasing, and IO between segments, as well as execution times of segments' instructions. We believe that such a technique is infeasible in a practical sense, because it presents an intractable circularity between the compiler-based analyzer and the scheduler. The result of the analysis affects that of scheduling, and vice versa.

Compiler-Assisted Adaptive Scheduling. Gopinath and Gupta introduced a technique called compiler-assisted adaptive scheduling in [18]. In their work, the compiler indexes a piece of code into four classes on the basis of predictability and monotonicity. Then it rearranges the code to support adaptive run-time scheduling. Unlike our approach, which is entirely static in terms of both program analysis and scheduling, their approach was developed to aid in dynamic runtime scheduling. However, their work still demonstrates that a successful interplay between the compiler and scheduler is possible without introducing a circularity between them.

Partial Evaluation. Partial evaluation may conceptually be understood as compile-time evaluation of constant expressions. Partial evaluation of a source program not only reduces constant expressions in the program into simple values, but also simplifies some control structures such as loops and conditionals. In [39] Nirke and Pugh applied the technique of partial evaluation to real-time programming to produce residual code that is both more optimized and more deterministic.

Safe Real-Time Code Optimizations. Conventional compiler optimizations are designed to reduce the expected or average execution time of programs without taking in account the exact timing behavior of programs. On the other hand, optimizations for hard real-time must meet stringent timing constraints. In [36] Marlowe and Masticola examine a large class of conventional code transformations, and then classify them for application in real-time programming by the notion of safe real-time code transformations. To do so, they assume time-critical statements (or events) in the underlying programming language, and interpret deadlines as relationships between the executions of time-critical statements. This approach comes closest to our work, in that the timing behavior of real-time programs is described in terms of events or the executions of time-critical statements. Unlike the semantics for TCEL, however, the execution times of non-time-critical statements are not explicitly decoupled from timing constraints imposed on the events. This is because the start of a process or start of a critical section is still considered "time-critical." As a result, the applicability of some transformations may be unnecessarily restricted.

Timing Tools. Predicting the worst-case execution time of a program is a fundamental requirement to build a real-time application. Indeed, the results of our compiler-based methods rely on timing tools, since both the feasibility and schedulability of tasks are a function of predicted worst-case executions of the transformed tasks.

The technique reported in [42] is based on a simple source-level timing schema, and it is fairly straightforward to implement in a tool. In [19] another approach for more accurate timing was proposed. Since the resulting tool is able to analyze micro-instruction streams using machine-description rules, it is retargetable to various architectures. On the other hand, both approaches do not address the problem of predicting architecture-specific timing behaviors due to various latencies in the memory hierarchy and pipelines. Recently, several approaches have been developed to account for this timing variance. In [55] Zhang et al. presented a timing analyzer based on a mathematical model of the pipelined Intel 80C188 processor. This analysis method is able to take into account the overlap between instruction execution and fetching, which is an improvement over schemes where instruction executions are treated individually. In [44] Arnold et al. developed a timing prediction method called *static cache simulation* to statically analyze memory and cache reference patterns. A similar but more advanced approach was reported in [32]. While the latter approach is able to predict pipeline stalls as well, both approaches essentially rely on attribute grammars [2] to propagate cache hit information backward in a flow graph.

However, no static timing tool is precise enough to be used with complete confidence for developing production-quality software. Moreover, even sophisticated timing analysis methods such as [32, 44] are not appropriate for fine-grained instruction timing. In Chapter 7 we explain how we can effectively use these tools in spite of the limitations, by also taking advantage of software profiling, as well as static timing prediction. Specifically, our slicing technique does not *require* any static analyzer: it can be used to first transform the program, with the timing carried out later by a runtime profiler.

Chapter 3

The Language of Time-Constrained Events

In this chapter we present TCEL's timing constructs to express timing constraints within a high-level program. These high-level timing constructs make explicit reference to *observable events* as well as *time*, so that the specification-level timing constraints can be extracted from the source code, and then conveyed to real-time schedulers.

3.1 Design Goals

The objectives of the TCEL design are (1) to lay a semantic foundation for developing safe real-time compiler transformation techniques; and (2) to provide high-level timing constructs that enable a compiler to automatically translate a high-level real-time specification into executable code. In addition to these, the design of the timing constructs is motivated by the following practical goals:

- *To add a minimal set of features to existing languages.* To this end, we embedded timing constructs in C, and then extended it as a real-time programming language. Our constructs are syntactic descendents of the *temporal scope* in [30].
- *To keep the definition of an observable event as general as possible.* The notion of an observable event is a relative term depending on the application system. We support a generalizable mechanism in that observable events are classified by the programmer, and the resulting event specification is externally provided for a TCEL compiler. Such a specification may include input and output operations, message passing primitives and memory accesses to shared variables. For simplicity, in the sequel we consider all "**input**" and "**output**" operations to be observable.

3.2 Timing Constructs and Their Semantics

We first elaborate the "**do**" construct which establishes several types of relative timing constraints. Its general form is as follows.

> **do**
>
> ⟨reference block⟩
>
> [**start after** t_{min}] [**start before** t_{max1}] [**finish within** t_{max2}]
>
> ⟨constraint block⟩

The reference block (RB) and the constraint block (CB) are simply C statements, or alternatively, timing constructs themselves. The "**do**" construct induces the following timing constraints:

- **start after** t_{min}: There is a minimum delay of t_{min} between the last event executed in the RB, and the first event executed in the CB.
- **start before** t_{max1}: There is a maximum delay of t_{max1} between the last event executed in the RB, and the first event executed in the CB.
- **finish within** t_{max2}: There is a maximum delay of t_{max2} between the last event executed in the RB, and the last event executed in the CB.

Since either block may contain conditionals, depending on the program's state there may be several such events executed either "first" or "last." For example, consider the fragment from a typical flow graph in Figure 3.1.

Depending on the path taken, the last event executed in the reference block may be either E1 or E2. Similarly, the first event in the constraint block will be E3 or E4, while the last event will be either E4 or E5. To denote such possibilities, we introduce two mappings *FIRST* and *LAST* from code blocks to sets of events. That is, $LAST(RB) = \{E1, E2\}$, $FIRST(CB) = \{E3, E4\}$ and $LAST(CB) = \{E4, E5\}$. Thus, the "**do**" construct introduces two potential constraints between an executed event from *LAST*(RB) and another from *FIRST*(CB), as well as one constraint between two executed events from *LAST*(RB) and *LAST*(CB) each.

The second real-time construct denotes a statement with cyclic behavior of a positive periodicity:

> **every** p [**while** ⟨condition⟩]
>
> [**start after** t_{min}] [**start before** t_{max1}] [**finish within** t_{max2}]
>
> ⟨constraint block⟩

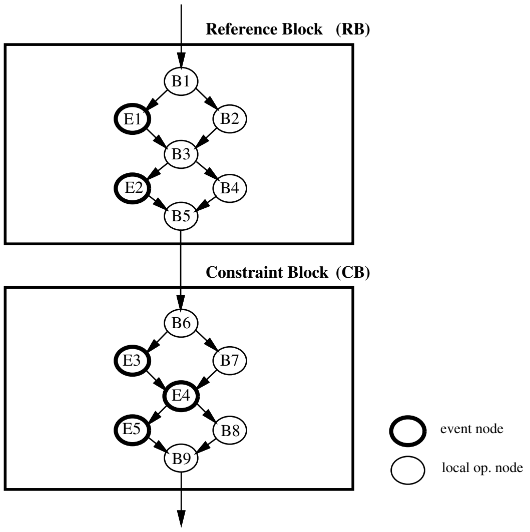

Figure 3.1: Flow Graph of **do** Construct

As long as the "**while**" condition is true, the observable events in the constraint block execute every p time units. Akin to an untimed **while**-loop, when the condition evaluates to *false* the statement terminates. However, unlike the untimed counterpart, event operations cannot be part of the condition. In its real-time behavior, the interpretation of the "**every**" construct is similar to that of "**do**." For example, assume that the statement is first scheduled at time t, and that the "**while**" condition is true for periods 0 through i. The periodic constraints established by this statement are depicted in Figure 3.2, where the time-line shows the first two instances of the statement.

Examining the time-line, we see that the **every** statement is released at time t, and that within the first frame, the first observable event (denoted by an arrow) occurs between $t + t_{min}$ and $t + t_{max1}$. Similarly, the first frame's last event occurs before $t + t_{max2}$. Generalizing, the following constraints are induced for period i where $i \geq 0$:

- **start after** t_{min} : The first event executed in the CB occurs after $t + i \cdot p + t_{min}$.
- **start before** t_{max1} : The first event executed in the CB occurs before $t + i \cdot p + t_{max1}$.
- **finish within** t_{max2} : The last event executed in the CB occurs before $t + i \cdot p + t_{max2}$.

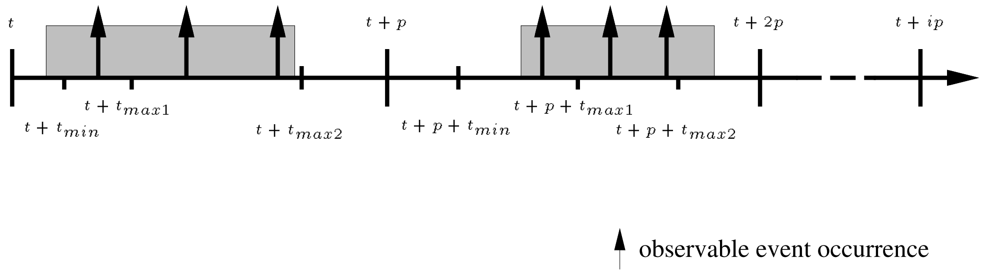

Figure 3.2: Behavior of Periodic Timing Construct

3.3 Example TCEL Code

As we have stated, timing constructs may be arbitrarily nested. Consider the program in Figure 3.3(A), which is a (very gross) 2-dimensional abstraction of an aircraft navigation/control loop. A set of route coordinates are maintained in the array "`GOAL`," which is managed by another module. These coordinates are accessed using index variable "`i`," which is initialized before the periodic loop, and updated within the loop body. The TCEL program's role is to (1) sample the aircraft's current coordinates, its (true) heading, roll, and its ground speed; (2) get the next route coordinate to visit; (3) compute the relative attitude between the heading and the coordinate; and (4) adjust the course by updating throttle and roll. Adjustments are made in discrete increments, and are contingent on the *current* roll and velocity, as well as the amount that the course must be changed.

The timing constraints are induced as follows:

(1) Control updates are made periodically, with rate 50/second.

(2) In order to give the actuators time to get updated (and for the craft to respond accordingly), all updates must be made within the first 5ms of each period.

(3) Velocity (ground speed) is obtained via a "request-response" protocol from an external unit; the response arrives with maximum latency of 0.75ms.

(4) To correlate ground speed with outputs, all throttle and flap updates must be made within of 3.1ms of *actual* ground speed sample. In the best case this may be made upon issuing the request.

If the specification mandated additional timing constraints, clearly we could employ further levels of nesting to achieve them. For example, suppose we desired to add a fifth timing requirement to the four listed above:

(5) The final two outputs must be correlated within 0.5ms of each other. (This is not unrealistic, since the two outputs control are coordinated to effect the angular adjustment.)

To accomplish this we would replace the sequential composition with an additional nested TCEL statement:

do
    output(THROT, throttle);
finish within 0.5ms
    output(FLAP_Cntrl, wflap);

The net runtime effect would simply be a refinement of the potential behaviors; i.e., the time-event relationships exhibited by the altered program would be a subset of those in the original version.

```
every 20ms finish within 5ms
  do
  {
    /* Get current position, heading & roll */
    input(GPS, &x, &y);
    input(NAV, &theta);
    input(IMU, &roll);

    /* Request current velocity */
    output(Cntrl_out, REQ_VEL);
  }
  start after 0.75ms finish within 3.1ms
  {
    /* Get current velocity */
    input(Cntrl_in, &vel);

    /* Update target position */
    if (GOAL[i].passed) {
      gx = GOAL[i].x;
      gy = GOAL[i].y;
      i = (i+1) % NCOORD;
    }

    /* Using relative attitude w.r.t target, */
    /* compute angular adjustment.          */
    rtheta = compRelAtt(theta, x, y, gx, gy);
    if (|rtheta| < EPS)
      dtheta = 0.0;
    else {
      if (vel < VHIGH)
        dtheta = rtheta;
      else
        dtheta = safeDtheta(rtheta, roll);
    }

    /* Adjust flap and throttle for heading. */
    wflap = compFlapw(roll, vel, dtheta);
    throttle = compThrot(roll, vel, dtheta);
    output(THROT, throttle);
    output(FLAP_Cntrl, wflap);
  }
```

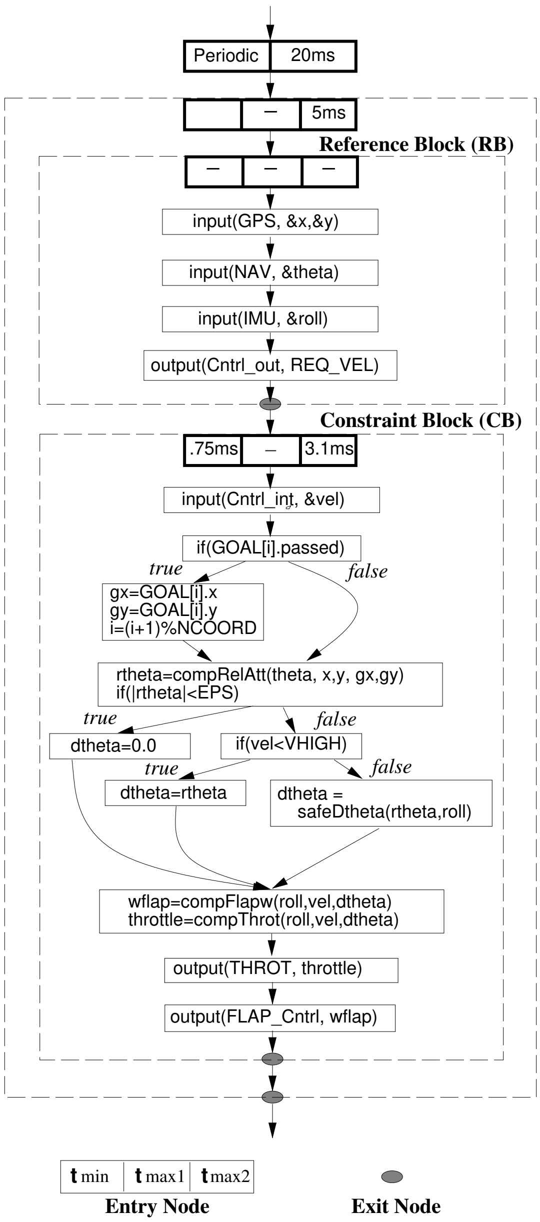

Figure 3.3: (A) TCEL Source Code for the Flight Controller and (B) Corresponding Flow Graph

Chapter 4

Basic Notations

The output of the TCEL compiler's machine-independent pass is the code in an intermediate representation, encapsulated in a *flow graph* [2]. We extend the format of a flow graph to hold the original timing information. For example, Figure 3.3(B) shows the extended flow graph for our flight controller program in Figure 3.3(A), where for the sake of brevity we have left the code in its original C form. In this chapter we first introduce two types of flow graphs, and then define basic notations of standard compiler technologies. These notations include a data structure called a *program dependence graph*, and a code representation called *static single assignment* form. We will use these notations throughout this dissertation.

4.1 Flow Graphs

A flow graph is a natural representation of an intermediate program from which other important information (such as data and control dependences) is extracted by standard static analysis techniques. In order to help compilers deal with nested program structures we define a hierarchical flow graph as well as a standard flow graph.

Flow Graph. A flow graph is a standard, flattened representation of a program, in which a node denotes a fragment of straight-line code and an edge specifies potential flow of control from one node to another.

Definition 4.1 A *flow graph* $FG(\mathrm{B})$ of code block B is a directed graph

$$(V, E, entry(\mathrm{B}), exit(\mathrm{B}))$$

where V is a set of basic blocks of B, and the two distinguished nodes $entry(\mathrm{B})$ and $exit(\mathrm{B})$ that denote the unique entry and exit of B, respectively. E is a set of edges representing potential control flow. □

Hierarchical Flow Graph. It is sometimes necessary to explicitly specify the program's original hierarchical levels of scoping in a flow graph. We call this structure a *hierarchical flow graph.*

Definition 4.2 A *hierarchical flow graph* $HFG(\mathrm{B})$ of code block B is a directed graph

$$(V, E, entry(\mathrm{B}), exit(\mathrm{B}))$$

where V is a set of nodes, and E is a set of edges representing potential control flow between nodes. A vertex $n \in V$, however, may be either a basic block of B, an entry node, an exit node, or another hierarchical flow graph $HFG(\mathrm{B}')$. □

Thus all nested constructs, including loops, are reduced into single nodes in a hierarchical flow graph. In our work we do not lift code out of loops, and thus we treat *HFG*'s as acyclic graphs, since we can ignore all back edges.

Of course our hierarchical structure assumes that the underlying programming language is "perfectly structured." That is, any two statements S_1 and S_2 in the program are in one of the following forms:

1. S_1 is contained in S_2;
2. S_2 is contained in S_1; or
3. S_1 and S_2 are disjoint.

Since many real-time programming languages allow only "structured" programs without unrestricted **goto**s[1], we assume that our programs possess this property.

Since nodes in both a flow graph and a hierarchical flow graph have the same interface and structure, all notations defined for a flow graph are used in a hierarchical flow graph. We can easily extend traditional compiler terminology to *HFG*'s as well. In the following sections we define basic compiler notations for $HFG(\mathrm{B}) = (V, E, entry(\mathrm{B}), exit(\mathrm{B}))$.

[1]We disallow **break** statement as well, since it is a special instance of **goto**.

4.2 Basic Compiler Definitions

Paths. A path (or trace) between $n_1, n_2 \in V$ is denoted by "$n_1 \rightarrow^b n_2$," where b is the sequence of nodes traversed between (and including) n_1 and n_2. When b includes a node $m \in V$, we denote this by overloading the set membership operator, i.e., as "$m \in b$."

We also use the path relation, but omit the actual path, to denote the existence of *some* path between two nodes, i.e.,

$$n_1 \rightarrow n_2 \stackrel{\text{def}}{=} \exists\ b :: n_1 \rightarrow^b n_2$$

We assume that for any node $n \in V, n \rightarrow n$.

Dominator and Postdominator. A node d is called a *dominator* of node n, if every path from *entry*(B) to n goes through d. Similarly, a node p is a *postdominator* of node n, if every path from n to *exit*(B) goes through d [2].

Data dependence. Let $Def(n)$ and $Use(n)$ be sets of variables defined and used by node n in B, respectively. For instructions $s_1, s_2 \in$ B, s_2 is *data dependent* on s_1 (denoted "$s_1 \xrightarrow{d} s_2$") iff there is a path b such that $s_1 \rightarrow^b s_2$ and

$$\exists v \in Def(s_1) \cap Use(s_2) :: (\forall s \in \text{B} :: (s \in b) \wedge v \in Def(s)) \Rightarrow s = s_1)$$

For instructions s_1 and s_2 in B, we say that s_2 is *transitively* data dependent on s_1 (denoted "$s_1 \xrightarrow{d}_+ s_2$") iff there is a path

$$s_1 \xrightarrow{d} s_1' \xrightarrow{d} s_1'' \xrightarrow{d} \ldots \xrightarrow{d} s_2$$

We extend the notion of data dependence for nodes of *HFG*. For nodes $n_1, n_2 \in V$, n_2 is *data dependent* on n_1 (denoted "$n_1 \xrightarrow{d} n_2$") iff

$$\exists s_1 \in n_1, s_2 \in n_2 :: s_1 \xrightarrow{d} s_2.$$

Control Dependence. For nodes $n_1, n_2 \in V$, n_2 is *control dependent* on n_1 (denoted "$n_1 \xrightarrow{c} n_2$") if one of the following holds:

1. n_1 is an entry vertex and n_2 is not nested within any loop or conditional; or

2. n_1 represents a control predicate and n_2 is immediately nested within the loop or the conditional whose predicate is represented by n_1.

Our definition of control dependence is simpler than that found in [11], since it covers a restricted language possessing only structured program constructs.

Reaching Definitions. For a node $n \in V$ and an expression e in B, we define reaching definitions $RD(n, e)$ as a set of nodes m such that

$$\begin{aligned}\exists b :: (m \rightarrow^{b} n \ \wedge \exists v \in \mathit{Def}(m) \cap \mathit{Use}(e) ::\\ \forall n' \in V :: ((n' \in b) \wedge v \in \mathit{Def}(n')) \Rightarrow n' = m).\end{aligned}$$

In other words, $RD(n, e)$ is a set of nodes whose definitions of some variables in $\mathit{Use}(e)$ are available at node n.

Dependence Closure (Static Backward Program Slice). The *dependence closure* for node n in the block B, denoted by "$DC(n, \mathrm{B})$," contains n and all nodes m that reach n via zero or more control or data dependence edges. It is inductively defined by the following least fix-point operation:

$$\begin{aligned}DC(n, \mathrm{B}) &= \mathit{fix}\, F(\{n\}), \text{ where}\\ F(S) &= \{m \in V \mid \exists\, n' \in S :: m \xrightarrow{d} n' \ \vee m \xrightarrow{c} n'\} \ \cup S\end{aligned}$$

When we are concerned only with data dependences, we make use of *data dependence closure*, "$DDC(n, \mathrm{B})$" defined as below.

$$DDC(n, \mathrm{B}) = \{m \in V \mid m \xrightarrow{d}_{+} n\} \ \cup \{n\}$$

In fact, a dependence closure and a data dependence closure are equivalent to a static backward program slice and a static data slice, respectively [52].

Program Dependence Graph. A program dependence graph is an intermediate program representation that stores both the data and control dependences for each operation in a program.

Formally, the program dependence graph is a directed graph $PDG(\mathrm{B}) = (U, W)$, where

- The vertex set U is equal to V in $HFG(\mathrm{B})$.
- The edges W are of two sorts: either a control dependence between $n1$ and n_2 such that $n_1 \xrightarrow{c} n_2$, or a data dependence between n_1 and n_2 such that $n_1 \xrightarrow{d} n_2$.

In addition, we define "$m \Rightarrow_{*} n$" to mean that there is a path starting from node n and ending at node q in $PDG(\mathrm{B})$.

Static Single Assignment Form. The static single assignment (SSA) form of a program can be considered not only as a sparse representation of flow data dependences, but also as a notation where the spurious data dependences such as output and anti-dependences are eliminated [7, 8].

A program is defined to be in SSA form if each use of a variable is reached by exactly one assignment to it [8]. Thus, a program's SSA representation can be obtained iteratively applying the following process: For each variable in the program, (1) unique names are given to all of its appearances on the left-hand-side of an assignment; and (2) all of the uses reached by that assignment are renamed to correspond to the new name. The following examples demonstrate this process. In the straight line code, each assignment to a variable is given a subscripted name, and all of its uses are then renamed as well.

```
v = f();              v1 = f();
a = v + 1;     ==>    a = v1 + 1;
v = g();              v2 = g();
b = v + 2;            b = v2 + 2;
```

Conditional statements require a bit more work in achieving the SSA form. At confluence points in the CFG, merge functions called ϕ-functions are introduced. A ϕ-function for a variable merges its possible values from distinct incoming control flow paths, and produces one argument for each control flow predecessor.

```
                          P = cond;
if cond                   if P
    then v = f();             then v1 = f();
    else v = g();   ==>       else v2 = g();
a = v;                    v3 = φ(v1, v2);
                          a = v3;
```

In addition to a confluence point immediately following a conditional, another type of confluence point exists at every loop header. For each variable v defined in a loop body, two values of v, namely one assigned v before the loop, the other within the loop, merge at the confluence point of the loop header. Thus a ϕ-function for v is introduced there.

```
v = 10;                v0 = 10;
do {                   do {
    v = v-1;    ==>        v1 = φ(v0, v2);
} while(v > 0)             v2 = v1-1;
                       } while( v2 > 0)
```

Here we denote the value initialized before the loop by $\mathtt{v}_0$ (subscript 0).

Gated Single Assignment Form. Although SSA form allows for efficient representation of data flows in a program, it possesses a weakness in terms of program transformability. For example, in the above code the ϕ-function is structurally bound to the conditional "`if P`," since the ϕ-function "$\mathtt{v}_3$=$\phi(\mathtt{v}_1, \mathtt{v}_2)$" can be executed only after the outcome of "`P`" is known. Such a coupling will be avoidable, if we parameterize the ϕ-function with the associated predicate.

Gated single assignment (GSA) form solves this problem [20]. GSA form is an extension of SSA form with new functions that encode control over value assignment. The following code fragments show translation from a conditional into the code in GSA form.

```
if cond                          P = cond;
    then v = f();                if P
    else v = g();     ==>            then v1 = f();
a = v;                               else v2 = g();
                                 v3 = γ(P, v1, v2);
                                 a = v3;
```

The γ-function denotes the extension. The first argument of the γ-function is the predicate associated with the original ϕ-function and its remaining arguments are the same as the ϕ-function. Having our intermediate programs in GSA form, we can treat γ-functions in the same way as we deal with any other statement.

In GSA, loops need a special form of pseudo-assignment function other than a γ-function. The reason is the confluence points at loop headers are not associated with particular conditional predicates. Thus for each variable v defined in a loop body, a definition $v' = \mu(v_{init}, v_{iter})$ is inserted at the loop header, where v_{init} is the initial value of v reaching the header from outside and v_{iter} is the iterative value reaching along the back-edge of the loop [20]. In our example below we denote v_{init} and v' by v_0 and v_1, respectively.

```
v = 10;                 v0 = 10;
do {                    do {
    v = v-1;     ==>        v1 = μ(v0, v2);
} while(v > 0)              v2 = v1-1;
                        } while( v2 > 0)
```

Chapter 5

Transformation 1: Feasible Code Synthesis

In this chapter we address translating a TCEL program into a low-level representation eligible for real-time scheduling. To give a clear understanding of the problem domain, consider Figure 5.1. Our problem domain at hand – denoted by the components with bold lines – actually contains two problems. The first problem lies in compiling a TCEL program into the schedulable units, and the second problem deals with attaining the feasibility of an infeasible task.

We approach the first problem with a compiler-based decomposition method. This method enables a TCEL compiler to automatically translate a TCEL source program into a set of decomposed tasks, and it derives a set of timing equations for them. Later, at runtime, a scheduler benefits from this decomposition method, since it can efficiently dispatch the tasks according to their time schedules (or priorities) based on both their execution times and associated timing equations.

The second problem arises when any task is determined to have an execution time conflicting with the derived timing constraints. In this case, it is impossible for any scheduler to generate a feasible schedule for the task set. We develop a compiler transformation method to automatically correct such feasibility faults based on a variant of Trace Scheduling [12]. We name this method *feasible code synthesis*. In this method, the compiler picks the worst-case execution time paths of an infeasible task and moves unobservable code to eliminate the overload.

The remainder of this chapter is organized as follows. In Section 5.1 we formally state the feasible code synthesis problem, and show that it is essentially an intractable problem. Then we briefly discuss our solution strategy. In Sections 5.2 and 5.3 we explain the transformation strategy.

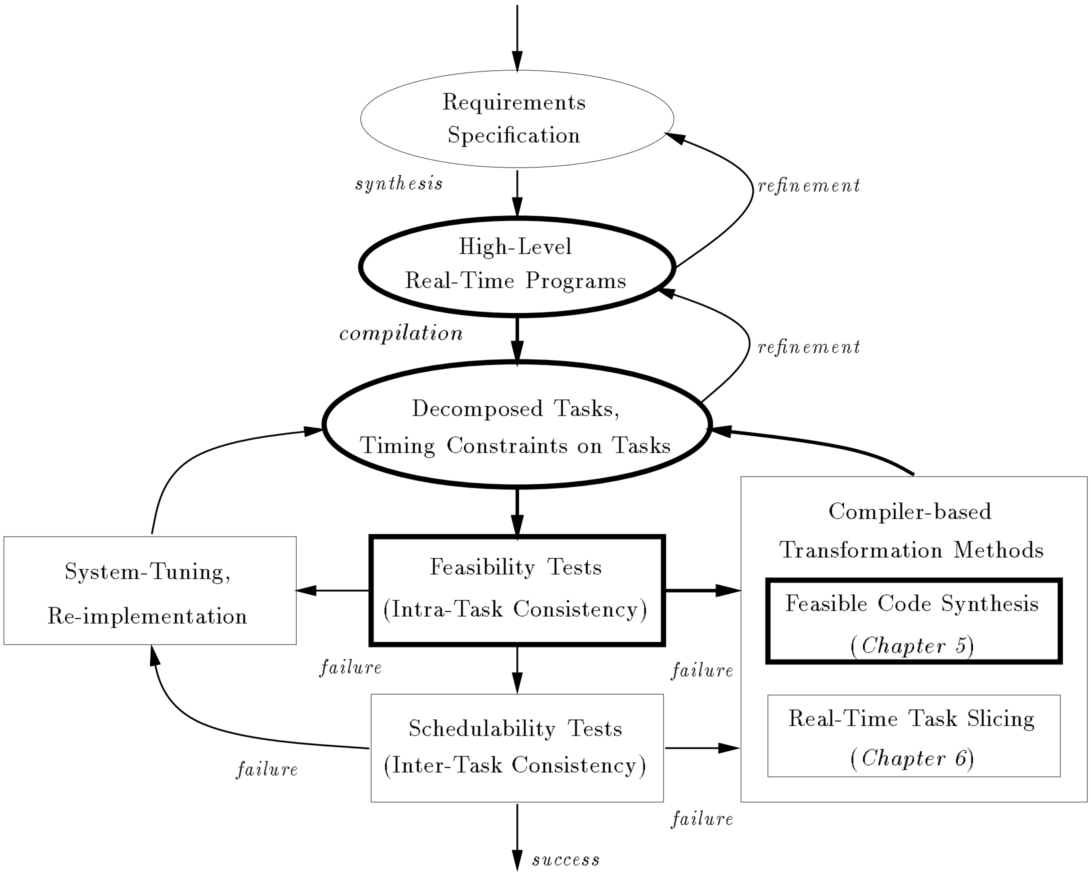

Figure 5.1: Software Development Process: Revisited for Feasible Code Synthesis

5.1 The Problem and Our Solution

Compiling a TCEL program into schedulable code raises two fundamental issues, which are endemic to all real-time systems.

1. A program may not be *feasible*, i.e., a single process's execution time may conflict with its own real-time constraints.

2. While the program may be feasible, it may not be *schedulable* under any tractable real-time scheduling algorithm.

As for the feasibility problem, consider the following TCEL fragment:

```
do
    input(P, &m);
start after 10ms finish within 20ms
{
    input(Q, &x);
    S;                                  [20ms]
    output(R, y);
}
```

The code's timing constraints mandate a 10ms latency between the events generated by "**input**(P, &m)" and "**input**(Q, &x)," as well as a 20ms deadline between the events generated by "**input**(P, &m)" and "**output**(R, y)." Meanwhile, the bracketed "20ms" denotes that the unobservable statement S requires a maximum of 20ms to execute, a bound obtained by a timing analysis tool (e.g., [19, 32, 42, 47, 55]). Consequently, the program possesses an inherent conflict, since S requires 20ms to execute while it is only allowed 10ms.

We address this problem by an approach we call *feasible code synthesis*. In our example this would involve decomposing S and, if possible, moving instructions not dependent on "**x**" out of the overloaded section. However merely achieving feasibility may be of little help, since the transformed code may still not be schedulable under any known methods. For example, when a program contains if-then-else branches, then the actual execution paths (and the events executed) are determined dynamically. But since schedulers must provide guarantees, they do not have the flexibility to instantaneously, dynamically reschedule a task set whenever an event is triggered. Indeed, while an event-based semantics makes conceptual sense at the source-program level, most real-time schedulers only accept timing constraints on the start and finish times of *tasks*.

Since the strategies used to achieve feasibility have a profound affect on the ultimate task structure, we solve these two problems together. Thus the role of the TCEL compiler is to partition event-driven source programs into time-constrained blocks of code, in which all of the blocks are feasible.

5.1.1 The Problem of Feasible Code Synthesis

Even without the code block partitioning component, achieving feasibility is a nontrivial problem. This is obviously the case if we allowed potentially unbounded loops (or conditional jumps), which would render the problem undecidable. But since real-time programs must be amenable to worst-case timing estimates, we assume that upper bounds can be obtained for execution times. Formally, we let "$wt(S)$" denote a statement's worst-case execution time, where we assume that $wt(S) \in [0, \infty)$ for any statement S. Obviously "wt" is implementation dependent, and its tightness is determined by the quality of the analysis tool used to generate the bound.[1]

While the feasibility problem may be decidable for our domain, it is not necessarily trivial. Even when a program is structured like our example above, and where S is a *basic block*, simply deciding whether the program can be made feasible is still NP-hard.

The problem can be stated as follows: given a TCEL timing construct

do RB **start after** t_{min} **start before** t_{max1} **finish within** t_{max2} CB

is it possible to transform RB and CB to meet the following constraints?

(1) On any execution path, the original ordering of observable events is maintained.

(2) The original data and control dependences are not violated between instructions and events.

(3) The code's execution time does not conflict with the timing constraints between the events.

Clauses (1)-(2) imply that the transformed code must be functionally correct: events may not be reordered, and the original relationships between input and output data must be maintained. Clause (3) means that the new code is feasible.

This problem is NP-hard, due to the existence of immovable operations and data dependences.

Theorem 5.1 *Feasible code synthesis is NP-hard.*

Proof: The proof follows by a straightforward transformation from "Partition[SP12]" [13] to feasible code synthesis. Consider an instance (A, s) of Partition, where $A = \{a_1, a_2, \ldots, a_n\}$ is a set of elements, and where $s : A \mapsto \mathbb{N}$ is the cost of each element. Letting $\sum_{i=1}^{n} s(a_i) = 2T$, a partition of A is some $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = T = \sum_{b \in A - A'} s(b)$. Determining whether such an A' exists is equivalent to determining whether the following TCEL program can be made feasible:

[1] We return to this issue in Chapter 7.

```
         do
E1:          input(P, &x);
         start after T start before T finish within 2T
         {
E2:          output(Q, g(x));
L1:          x1=f1(x);                          [s(a1)]
L2:          x2=f2(x);                          [s(a2)]
               ⋮                                  ⋮
Ln:          xn=fn(x);                          [s(an)]
E3:          output(R, h(x1,...,xn));
         }
```

Here $E_1 - E_3$ generate events, $L_1 - L_n$ are unobservable instructions, and each line is considered atomic (that is, a line must be relocated as a single entity). Then by the construction,

(1) E_2, $L_1 - L_n$ are mutually data independent.

(2) E_2, $L_1 - L_n$ are data dependent on E_1.

(3) E_3 is data dependent on $L_1 - L_n$.

If we assume that $wt(L_i) = s(a_i)$ for $1 \leq i \leq n$, and that $wt(E_j) = 0$ for $1 \leq j \leq 3$, then there exists a partition of our original set A if and only if there is a feasible transformation for the program.

As for the "only if" part, assume there is a partition $A' \subseteq A$. Then for all $a_i \in A'$, moving the corresponding instruction L_i between the events E_1 and E_2 ensures feasibility: exactly T execution time is consumed between E_1 and E_2, and another T is used between E_2 and E_3.

As for the "if" part, assume there is a feasible transformation. Then, since $2T$ execution time is used overall, the constraints mandate that exactly T of it be used between E_1 and E_2, and the rest between E_2 and E_3. Thus we must move some set of instructions $L \subseteq \{L_1, \ldots L_n\}$ between E_1 and E_2, where $\sum_{L_j \in L} wt(L_j) = T$. But then the corresponding elements in A form a partition. □

When both the constraint block and reference block consist of straight-line code, the problem is obviously in NP as well. A feasible ordering can always be "guessed" and then verified, which consequently yields the following corollary.

Corollary: Feasible code synthesis is NP-complete for straight-line code. □

5.1.2 Solution Strategy

In proving Theorem 5.1 we used the simplest possible TCEL program, which possessed just two basic blocks. In this case a feasible transformation would simply reorder the instructions, while keeping the program's fundamental structure intact.

But the situation gets significantly more complicated when the program possesses branches, and when the events that get executed are determined at runtime. Since attaining feasibility mandates that we statically guarantee the timing constraints along *all* execution paths, reordering the instructions along a *single path* may not be sufficient. In the worst of all cases, all paths of a multi-branching program would have to be "expanded," and then each one reordered individually – potentially requiring time *and* space exponential in the original number of instructions. Clearly this is not an attractive solution approach. Moreover, it would render the problem of *schedulable* task decomposition – our second objective – next to impossible.

Thus we take the following alternative approach, in which feasible tasks are synthesized in a two-step process – *section decomposition* (Section 5.2) and *code scheduling* (Section 5.3).

Section Decomposition. First the code is translated into its gated single assignment (GSA) form [8, 20]. This representation serves two purposes: (1) it yields a compact means of representing the data-dependence relation discussed in Chapter 4; and (2) GSA's convention of uniquely naming each variable assignment is precisely what we require in the code transformations phase.

Next, the timing construct is decomposed into *sections*, which represent the natural schedulable "task" units. This step involves determining the section boundaries, as well as generating a set of dispatch equations that constrain the start and finish times of each section.

Code Scheduling. In this step the dispatch equations are checked for their consistency with the code's execution time. If there is an inconsistency, program transformations are used to relocate unobservable code across section boundaries.

The algorithm is a greedy approximation, in that each section is processed locally, with the goal of attaining consistency for an entire program. The following strategy is used: if a given section is determined to be over-constrained, code is moved out of the section and "up" on the *HFG*. After the section at hand is determined consistent, another section can be processed in reverse topological order. Thus the net result is an "upward migration" of unobservable instructions, which terminates either when the program achieves consistency, or it is determined to be inconsistent.

An analogous strategy is used for programs with nested timing constructs. In this case the *HFG* is scheduled in a "bottom-up" manner. That is, the innermost nodes in the *HFG* are scheduled first, with the goal of satisfying their local constraints. Then, the surrounding node is handled. If *this* level is found inconsistent, the inner nodes are "opened up" once again, and more aggressive optimization is carried out.

The actual transformations are similar to those used in Trace Scheduling [12]. As the name implies, the Trace Scheduling algorithm works on specific traces: it selects a path (or trace) from a given code block, and then selects instructions on that path to move. Such an approach is well suited to our purpose, because it can focus on the traces that have the maximum execution time among all traces within a given code block.

5.2 Section Decomposition

Section decomposition is the process of decomposing the program into a set of "code blocks" (or *sections*). The input is the original *HFG* (e.g, that portrayed in Figure 3.3(B)), with the output being a slightly different *HFG*, which is more amenable to real-time dispatching. This involves dividing a timing construct into five code sections, as portrayed in Figure 5.2. As can be seen, the reference block is decomposed into three sub-blocks. The unobservable code before the first observable statement becomes an interface section (S1). The code containing the observable statements becomes the reference section (S2). The unobservable code after the observable statements becomes the first part of the delay section (S3). Consequently, the topmost unobservable code of the constraint block becomes the second part of S3, and so on.

5.2.1 Determining Section Boundaries

Recall the discussion of the $FIRST$ and $LAST$ functions in Chapter 3. Since a code block may contain complicated control structures, we require a convenient means of defining the boundaries of S2 and S4 – the sections that contain observable events. We accomplish this by inserting "markers" in the flow graph, which consume no time and are not visible. The following marker definitions guarantee that there are unique boundaries into and out of the sections containing observable events.

- begin_S2: This marker is inserted directly after the unobservable instruction most closely dominating $LAST(\mathrm{RB}) \cup \{exit(\mathrm{RB})\}$.

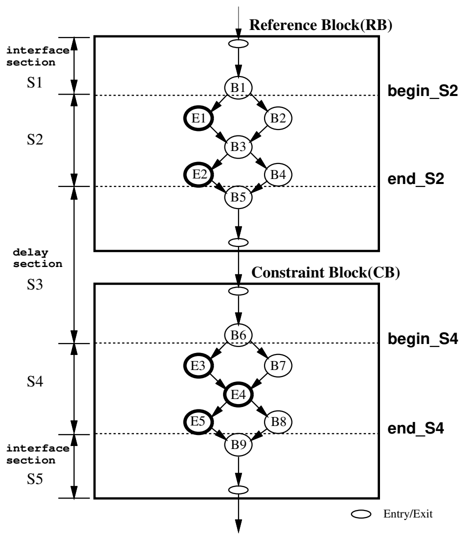

Figure 5.2: Flow Graph of **do** Construct and its Section Division

- end_S2: This marker is inserted directly before the unobservable instruction most closely post-dominating $LAST(\text{RB}) \cup \{entry(\text{RB})\}$.
- begin_S4: This marker is inserted directly after the unobservable instruction most closely dominating $FIRST(\text{CB}) \cup \{exit(\text{CB})\}$.
- end_S4: This marker is inserted directly before the unobservable instruction most closely post-dominating $LAST(\text{CB}) \cup \{entry(\text{CB})\}$.

For example, consider the constraint block in Figure 5.2. The unobservable node B9 post-dominates $LAST$(CB) and the entry node. Thus, its logical place is in the interface section S5, which is not subject to the construct's timing constraints. Hence we need the marker end_S4, which is the unique exit point for the constrained section S4.

Now, let the variable S2.start correspond to the actual time that the marker begin_S2 is "executed" (that is, the dispatch time of section S2), and let S2.finish correspond to the time that the section ends. Similarly, let S4.start and S4.finish represent the start and finish times of section S4. Using these variables we can represent the section decomposition of a TCEL construct in a manner similar to that found in the Flex language [25].

Recall the flight controller program from Figure 3.3. Figure 5.3 illustrates its constituent sections. The constraint-expression for S6 corresponds to the program's outer, periodic loop. As the program is in GSA form, γ-functions appear at confluence points where different values of the same variable in the original program merge. For example, "`dtheta4=`γ`(c3,dtheta2,dtheta3)`" is inserted at the place where two different values of `dtheta` merge. As a result, each use of a variable is reached by a unique assignment. Also, μ-functions for variables `gx`, `gy` and `i` are inserted at the periodic loop header so that the initialized values are merged with the iterative values. Again, the bracketed numbers denote the maximum execution times of the corresponding operations on the targeted CPU. On modern architectures, fine-grained operations like simple assignments possess minuscule execution times, and cannot be measured (in isolation) by any timing tool. For the sake of presentation we assume that such instructions take zero time, and concentrate on larger-grained function calls and the like.[2]

5.2.2 Deriving Code-Based Timing Constraints

As seen in Figure 5.3, the code-based timing constraints can be expressed as conjunctions of linear inequalities between start-times and finish-times of different sections. However, note the difference between the code-based constraints and the TCEL source-level constraints: In Figure 3.3 the "**finish within**" deadline is 3.1ms, while in Figure 5.3 it is tightened to 3.0ms. There is good reason for this – the new code-based timing constraints must be strong enough to guarantee the original semantics of the event-based constraints. That is, they must take into account the program's execution-time characteristics. In general, consider the TCEL construct such as

do RB **start after** t_{min} **start before** t_{max1} **finish within** t_{max2} CB

Obviously, the original TCEL parameters are not tight enough to guarantee the correctness of the code-based constraints. For example, if we wish to maintain the "t_{max1}" requirement, it is not sufficient to simply mandate that S4 starts within a maximum delay of t_{max1} after S2 ends (though this is certainly necessary). We can see in Figure 5.2 that the event actually *executed* in $LAST$(S2) may be E1, while the event executed in $FIRST$(S4) may be E4. Thus the naive strategy fails to factor in the execution times of B3 and B4.

However, the event-based semantics is clear: the time between the *executed* event in $LAST$(S2) and the *executed* event in $FIRST$(S4) is at most t_{max1}. To guarantee that this occurs, we must account for *all* possible execution scenarios. Specifically, we must tighten the constraints, allowing for the maximum amount of time between an event in $LAST$(S2) and end_S2, as well as the

[2]We revisit this issue in Chapter 7.

```
S6: (S6.start[p]≥p×20ms, S6.finish[p]≤p×20ms+5ms)
    {
S1:   {
        gx1 = μ(gx0, gx3);
        gy1 = μ(gy0, gy3);
        i1 = μ(i0, i3);

        input(GPS, &x1, &y1);                                   [.1ms]
        input(NAV, &theta1);                                    [.1ms]
        input(IMU, &roll1);                                     [.1ms]
      }
S2:   {
        output(Cntrl_out, REQ_VEL);                             [.1ms]
      }
S3:   { /* Null */ }
S4:   (S4.start≥S2.finish+0.75ms, S4.finish≤S2.finish+3.0ms)
      {
        input(Cntrl_in, &vel1);                                 [.1ms]
        c1 = GOAL[i1].passed
        if (c1) {
          gx2 = GOAL[i1].x;
          gy2 = GOAL[i1].y;
          i2 = (i1+1) % NCOORD;
        }
        gx3 = γ(c1, gx2, gx1);
        gy3 = γ(c1, gy2, gy1);
        i3 = γ(c1, i2, i1);

        rtheta1 = compRelAtt(theta1, x1, y1, gx3, gy3);         [.25ms]
        c2 = |rtheta1| < EPS;
        if (c2)
          dtheta1 = 0.0;
        else{
          c3 = vel1 < VHIGH;
          if (c3)
            dtheta2 = rtheta1;
          else
            dtheta3 = safeDtheta(rtheta1, roll1);               [.43ms]
          dtheta4 = γ(c3, dtheta2, dtheta3);
        }
        dtheta5 = γ(c2, dtheta1, dtheta4);

        wflap = compFlapw(roll1, vel1, dtheta5);                [.95ms]
        throttle1 = compThrot(roll1, vel1, dtheta5);            [.89ms]
        output(THROT, throttle1);                               [.1ms]
        output(FLAP_Cntrl, wflap);                              [.1ms]
      }
S5:   { /* null */ }
    }
```

Figure 5.3: Flight Controller Program: After Section Decomposition

maximum amount of execution time between begin_S4 and an event in *FIRST*(S4). We must similarly adjust t_{max2}. To do this, we make the following definitions:

- $\Delta_{\text{S2}} \stackrel{\text{def}}{=} max\{wt(p) \mid e \in LAST(S2),\ e \rightarrow^p \text{end\_S2}\}$.[3]
- $\Delta_{\text{S4}} \stackrel{\text{def}}{=} max\{wt(p) \mid e \in FIRST(S4),\ \text{begin\_S4} \rightarrow^p e\}$.

Note that Δ_{S2} and Δ_{S4} are sensitive to not only code's execution time characteristics, but also changes made to some paths between events and markers during program translation. For example, changes to paths between end_S2 and a node in *LAST*(S2) might require re-evaluation of Δ_{S2}.

Now the code-based timing constraints can be postulated as follows:

$$
\begin{array}{llcll}
(1) & \text{S4.start} & \geq & \text{S2.finish} + T_{min} & (\text{where } T_{min} = t_{min}) \\
(2) & \text{S4.start} & \leq & \text{S2.finish} + T_{max1} & (\text{where } T_{max1} = t_{max1} - \Delta_{\text{S2}} - \Delta_{\text{S4}}) \\
(3) & \text{S4.finish} & \leq & \text{S2.finish} + T_{max2} & (\text{where } T_{max2} = t_{max2} - \Delta_{\text{S2}})
\end{array}
$$

These timing constraints are strong enough to guarantee the original event-based timing constraints. (By convention, if the "**start after**" constraint is omitted, we consider t_{min} to be 0. Similarly, when either the "**start before**" or "**finish within**" constraints are missing, we consider $t_{max1} = \infty$ or $t_{max2} = \infty$, respectively.) Returning to Figure 5.3, we can see that equation (3) indeed mandates tightening the original 3.1ms to 3.0ms.

Now we wish to determine when (1)-(3) can be met. That is, what do these equations infer about the program's allowable worst-case execution-time behavior? This can easily be derived if we add precedence constraints reflecting the natural flow of the program; *i.e.*, that S4 executes after S3, which executes after S2:

$$
\begin{array}{lrcl}
(4) & \text{S2.finish} + wt(\text{S3}) & \leq & \text{S4.start} \\
(5) & \text{S4.start} + wt(\text{S4}) & \leq & \text{S4.finish}
\end{array}
$$

Eliminating S2.finish, S4.start and S4.finish from (1)-(5), we end up with:

$$
\begin{array}{lrcl}
(a) & T_{min} & \leq & T_{max1} \\
(b) & wt(\text{S3}) & \leq & T_{max1} \\
(c) & wt(\text{S3}) + wt(\text{S4}) & \leq & T_{max2} \\
(d) & wt(\text{S4}) & \leq & T_{max2} - T_{min}
\end{array}
$$

[3] We overload the *wt*-function for paths, since a path can be thought of as a sequential composition of statements.

Section	Duration Constraint (DUR(S))
S3	$min\{T_{max1}, T_{max2} - wt(\text{S4})\}$
S4	$T_{max2} - T_{min}$

Table 5.1: Timing Constraints of S3 and S4

Obviously, (a) had better be true in order for the TCEL construct to make any sense. For the purposes of our algorithm we combine (b) and (c), yielding the following two constraints on execution times:

$$\begin{array}{lrcl} (*) & wt(\text{S3}) & \leq & min\{T_{max1}, T_{max2} - wt(\text{S4})\} \\ (**) & wt(\text{S4}) & \leq & T_{max2} - T_{min} \end{array}$$

These are the necessary and sufficient conditions to achieve feasibility, and they are summarized in Table 5.1.

Returning to our example, we find that section S4 violates its duration constraint ($DUR(\text{S4}) =$ 2.25ms),[4] since $wt(\text{S4})$ by far exceeds it. (Adding up the time annotations yields $wt(\text{S4}) = 2.82ms$ along the worst-case execution time path.) In the next section we discuss our code-scheduling techniques which handle cases such as this, in which the duration constraints fail to hold.

5.3 Code Scheduling

The code scheduling algorithm is inspired by a common compiler strategy used for VLIW and superscalar architectures [3, 10, 12, 14, 38, 48]. In such domains, an optimizing compiler exploits a program's inherent fine-grained parallelism, and "packs" its computations into as many functional units as possible. Thus the objective is to keep each unit busy, and to achieve better overall throughput.

Our problem context has an entirely different goal, and it cannot be solved by *directly* applying well-known techniques such as Trace Scheduling [12] or Percolation Scheduling [3, 10]. We are concerned not with enhancing average-case performance, but instead with ensuring feasibility. In fact, we will be satisfied with even *increasing* the program's overall execution time – as long as the timing constraints are met.

[4] $T_{max_2} - T_{min} = 3.0\text{ms} - 0.75\text{ms} = 2.25\text{ms}$

algorithm *Code_Scheduling*(T) /* T is a timing construct */
input: the ordered set of sections $\{S1, S2, \ldots, S5\}$ in T
begin
    $d = T_{max2} - T_{min}$;
    /* Schedule code from S4 into S3 */
    call *Schedule_Section*(S4, S3, d, $\emptyset$);
    recompute T_{max1}; /* to reflect the change in Δ_{S4} */
    if (wt(S3) $\leq T_{min}$) **then** exit("No scheduling needed for S3.");
    else $d = \min\{T_{max1}, T_{max2} - wt(S4)\}$;
    /* Schedule code from S3 into S1 */
    call *Schedule_Section*(S3, S1, d, *Def*(S2));
end

Figure 5.4: Top-Level Algorithm for Code Synthesis

5.3.1 The Top-Level Algorithm

Our approach to code scheduling is a greedy approximation, and it attempts to attain the desired feasibility of a timing construct in a section-by-section manner. It inspects sections S4 and S3 (in reverse topological order), and checks whether they satisfy their duration constraints. If S4 violates its constraint, the algorithm attempts to reduce its surplus execution time by moving nodes into section S3. In turn it processes section S3, which may now contain newly moved code.

To perform greedy code motion, we have adapted a technique from the approach to Trace Scheduling in [12], and we use it as a component of the code scheduling algorithm. In our approach, nodes lying on paths that exceed their section's duration constraint are considered for code motion. We distinguish such paths as *critical traces*. Formally, a critical trace p of section S is defined as a path

$$entry(\mathrm{S}) \rightarrow^{p} exit(\mathrm{S})$$

such that $wt(p) > DUR(\mathrm{S})$. The reason we use the trace-based approach is straightforward: optimizing to avoid hard real-time exceptions demands scheduling only the critical traces, and no others.

Figure 5.4 sketches the algorithm. Note that T_{max1} is recomputed after scheduling S4 and before scheduling S3. This is mandatory, since Δ_{S4} may be changed during the scheduling. Also observe that the code of S3 is moved into S1, while that of S4 is moved into S3. We disallow code from moving into S2 because it could potentially change the value of Δ_{S2}, which would in turn invalidate our assumptions about DUR(S4). In order to complete the procedure in a single pass, we assume that Δ_{S2} remains constant. In reality this restriction does not seriously limit the approach: from our experience, events in the RB typically lie in straight-line code (and thus S2 contains a single instruction, as in our example).

The top-level algorithm calls subroutine "*Schedule_Section*," which then schedules the overloaded section at hand. Note that when code is scheduled from S3 into S1, the variables defined in S2 are passed to the subroutine, which ensures the dependences from S2 to S3 are maintained. In the following section we discuss this subroutine, and the strategies it uses to solve the scheduling problem.

5.3.2 Subroutine `Schedule_Section`(S,D,DUR(S),V_{bar})

For the flow graphs of source section S and destination section D, the critical trace scheduling problem is to construct new flow graphs for S and D such that:

(1) The observable nodes of S remain in S and keep their relative order on all paths in S.

(2) $wt(p) \leq DUR(\mathrm{S})$ holds for all traces p in S.

(3) Execution ordering established by code's original data dependences is preserved.

When code is scheduled from S3 into S1 the parameter V_{bar} – containing the variables defined in S2 – is required to maintain property (3).

As we have stated, the trace-based approach is attractive precisely because it allows us to concentrate on paths which violate the duration constraints. However, a direct application of Trace Scheduling induces a severe liability – extra code must be inserted to preserve the program's semantic integrity. In the parlance of instruction-scheduling, this is typically called *bookkeeping code*.

Consider the GSA program in Figure 5.5(B) to see why Trace Scheduling requires bookkeeping. Since the instruction "`z1=F(x)`" is free of a data dependence on the variable"`y`," it *may* be eligible to be moved into S3. (This transformation – which we develop in the sequel – is called *speculative*, since it breaks a control dependence.) As shown in Figure 5.5(C), moving the instruction requires no additional code to maintain the program's semantics; GSA's naming conventions maintain the correctness.

(A) Code in TCEL	(B) Code in GSA	(C) Our Approach	(D) Bookkeeping Approach

do
  input(P, &x);
start after t_1
finish within t_2
{
  input(Q, &y);
  c1 = p(y);
  if (c1) {
    a = E(y);
    z = F(x);
  } **else**
    z = G(y);
  c2 = q(y);
  if (c2)
    r = H(z);
  else
    r = K(z);
  s = I(r);
    ⋮
}
	⋮
	S2:
	{
	  input(P, &x);
	}
	S3: { /* Null */ }
	S4: (S4.start$\geq$ ...,
	    S4.finish$\leq$ ...)
	{
	  input(Q, &y);
	  c1 = p(y);
	  if (c1) {
	    a1 = E(y);
	    z1 = F(x);
	  } **else**
	    z2 = G(y);
	  a2 = γ(c1, a1, a0);
	  z3 = γ(c1, z1, z2);
	  c2 = q(y);
	  if (c2)
	    r1 = H(z3);
	  else
	    r2 = K(z3);
	  r3 = γ(c2, r1, r2);
	  s = I(r3);
	   ⋮
	}
		⋮
		S2: {
		    input(P, &x);
		  }
		S3: {
		    z1=F(x);
		  }
		S4: (S4.start$\geq$ ...,
		    S4.finish$\leq$ ...)
		  {
		    input(Q, &y);
		    c1 = p(y);
		    if (c1)
		      a1 = E(y);
		    else
		      z2 = G(y);
		    a2 = γ(c1, a1, a0);
		    z3 = γ(c1, z1, z2);
		    c2 = q(y);
		    if (c2)
		      r1 = H(z3);
		    else
		      r2 = K(z3);
		    r3 = γ(c2, r1, r2);
		    s = I(r3);
		     ⋮
		  }
			⋮
			S2: {
			    input(P, &x);
			  }
			S3: {
			    z1 = F(x);
			    r1 = H(z1);
			    r2 = K(z1);
			  }
			S4: (S4.start$\geq$ ...,
			    S4.finish$\leq$ ...)
			  {
			    input(Q, &y);
			    c1 = p(y);
			    if (c1) {
			      a1 = E(y);
			      c3 = q(y);
			      r3 = γ(c3, r1, r2);
			    } **else** {
			      z2 = G(y);
			      c4 = q(y);
			      if (c4)
			        r4 = H(z2);
			      else
			        r5 = K(z2);
			      r6 = γ(c4, r4, r5);
			    }
			    a2 = γ(c1, a1, a0);
			    z3 = γ(c1, z1, z2);
			    r7 = γ(c1, r3, r6);
			    s = I(r7);
			     ⋮
			  }

Figure 5.5: (A) Source Code in TCEL, (B) Corresponding Intermediate Code in GSA Form, (C) Intermediate Code after Bookkeeping-Free Transformations, and (D) Intermediate Code after Transformations with Bookkeeping

However a more aggressive policy could be carried out, which is shown in Figure 5.5(D). Note that additional code may be moved without breaking data dependences; even the variable `r` may be split into movable parts (i.e., `r1` and `r2`) and the parts that depend on `y` (i.e., `r3` and `r5`). However, the price we pay is the additional bookkeeping code required to maintain correctness.

One obvious problem with bookkeeping is that it induces a significant amount of extra code – indeed, if carried to extremes, the transformations in Figure 5.5(D) may result in an exponential blow-up. And in our problem context bookkeeping may have an additional, "fatal" effect:

> *Scheduling a critical trace may insert bookkeeping code on other, non-critical traces, and thereby increase their execution times. Hence a non-critical trace may become critical.*

To avoid this drawback of Trace Scheduling, we use the type of transformation depicted in Figure 5.5(C), and we use it as aggressively as possible. The strategy involves repetitively applying the following three steps: (i) finding a critical trace p; (ii) identifying a node n which can be moved into the destination section D; and (iii) moving n into D, along with n's ancestor nodes required to maintain the program's semantics. Since our objective is to keep the amount of new code to a minimum, step (iii) translates into the following rules for moving n into D:

(1) Node n's data dependence predecessors are moved along with n; *i.e.*, the nodes on which n is transitively data dependent.

(2) The control-dependence predecessors (i.e., the **if-then-else**'s guarding n) are treated as follows:

  (a) If possible they are copied into D, so that they still guard the execution of n.

  (b) Otherwise (as in Figure 5.5(C)), n will now be unguarded in its destination section D.

The end result of code scheduling appears as if large-grained control structures were rearranged, and hence we name the strategy *structural code motion*. Yet code scheduling is still trace-based, since it is driven by worst-case paths.

Consider a node n to be moved into the destination section. When *all* of n's dependence predecessors (both data and control) are moved (or copied) along with n, the new execution ordering is guaranteed to maintain the program's original semantics.

But what are the ramifications of case 2(b) above, i.e., where control dependences are broken? Consider Figure 5.6(A), where we assume that the condition variable "`c`" is dependent on an input event. Examining the source code, assume that we wish to move the node x=a+3 above if c.

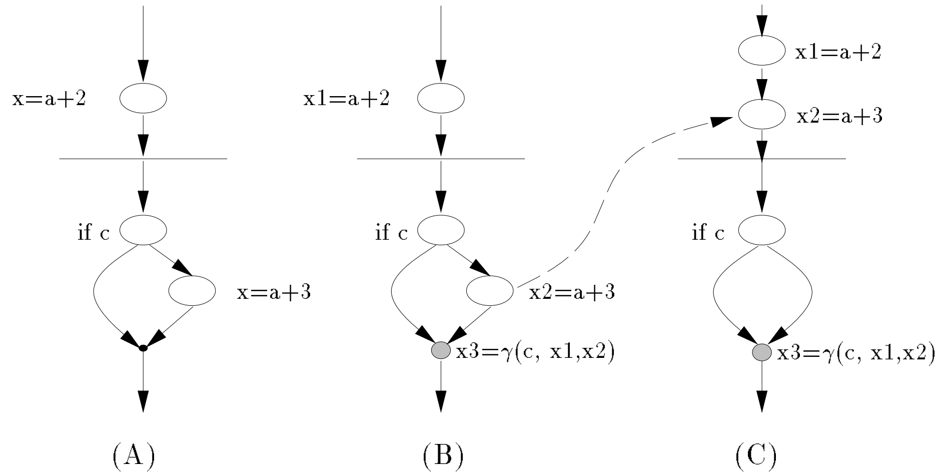

Figure 5.6: Speculative Code Motion: (A) Original Code, (B) Corresponding GSA Code and (C) Speculatively Transformed Code

The moved instruction is executed regardless of the control-predicate's outcome; hence the name "speculative transformation."

Carrying out speculative transformations raises three critical issues: variable naming, execution time and safety.

Naming: Consider what would happen if the transformation shown in Figure 5.6(C) were performed at the source level. Since `x` would always end up defined as `a+3`, one branch of the conditional would result in an incorrect state. Fortunately, the GSA form of the program ensures that multiply defined source variables – and their corresponding γ-functions – maintain the original semantics, regardless of where assignments are moved. Examining Figure 5.6(C), we see that `x1` and `x2` are defined sequentially. By GSA's naming conventions, the node $x3 = \gamma(c, x1, x2)$ ensures that x3 always carries the assignment corresponding to the original source variable `x`.

Timing: Figure 5.6 shows how speculative transformations can easily *increase* the execution time of the destination section D. This is not necessarily harmful, since D may in fact possess sufficient slack for both instructions to execute. (Indeed, D may contain an explicit delay.) But if D itself exceeds its own duration constraint, excessive speculative transformations could make matters worse. Thus we take the following approach: the algorithm performs speculative code motion only when feasibility cannot be achieved with the non-speculative variety.

Safety: Perhaps the most critical issue is the correctness of the transformed program. After all, the source code is written by a *human* programmer. When an instruction appears within the body of a conditional (but is free of a transitive dependence on it), one should still assume that the programmer had a good *reason* for putting it there. Often the reason stems from a personal coding style, or perhaps for readability. Also, splitting variable definitions in the style of GSA is a rather unnatural practice at the source level.

Referring back to Figure 5.5(C), we note that the "eager" execution of "`F`" should be safe if: (1) it contains no observable events, (2) it induces no global side-effects, and (3) it does not cause an exception. We can assume that (1)-(2) hold – otherwise "`F`" would not have been moved in the first place. However, verifying property (3) may be difficult, since there may be an invariant relationship between "`p`" and "`F`" that only the programmer understands.

While this seems to argue against speculative transformation, recall that our objective is to assist programmers in tuning faulty code. And production real-time programmers will find this type of code reordering sadly familiar, since it is usually carried out by hand, and often under the pressure of an approaching release deadline. Our technique can help in this effort, since it helps automate this process by (1) identifying the "good target" instructions to move, (2) by transferring them to their "correct" places, and (3) by analyzing the results. Nonetheless, we do believe that this should be an interactive process (perhaps driven by a graphical front-end), in which the programmer visually checks each transformation.[5]

For the sake of brevity, however, we present the algorithm in a fully automatic form. Thus we assume that any node n that can be speculatively executed is "pre-checked" and is denoted by the condition $spec(n)$.

Unconditional and Speculative Movability. The preceding discussion leads to three classes of instructions: those that can be unconditionally moved, those that can be moved to execute speculatively, and those which cannot be moved at all. The following definitions distinguish between these cases.

Definition 5.1 (Unconditional Movability) $Mu(\mathrm{S}, V_{bar})$ is the set of nodes in S that do not trigger events, and do not use any variables in V_{bar}; i.e.

$$Mu(\mathrm{S}, V_{bar}) = \{m \in \mathrm{S} \mid m \text{ is not an event} \wedge\ Use(m) \cap V_{bar} = \emptyset\}$$

[5]We discuss this issue in Chapter 7.

Then $Umove(n)$ denotes that we can unconditionally schedule node n from S into D:

$$Umove(n) \;\equiv\; DC(n, \mathrm{S}) \subseteq Mu(\mathrm{S}, V_{bar})$$

That is, all of n's data and control dependence ancestors are also unconditionally movable – and when n is moved, they will be moved (or copied) as well. □

Definition 5.2 (Speculative Movability) Additional considerations come into play when a node is speculatively executed. Consider the set $Ms(\mathrm{S}, V_{bar})$:

$$Ms(\mathrm{S}, V_{bar}) = \{m \in \; Mu(S, V_{bar}) \,|\, spec(m)\}$$

If a node n is in $Ms(\mathrm{S}, V_{bar})$ then (1) it uses no variables in V_{bar}, (2) it triggers no events, and (3) the programmer has checked that it does not cause a local exception. Then $Smove(n)$ denotes that we can speculatively move node n:

$$Smove(n) \equiv DDC(n, \mathrm{S}) \subseteq Ms(\mathrm{S}, V_{bar})$$

That is, when $Smove(n)$ holds true, all of n's data-dependent ancestors can be moved too, without mandating bookkeeping. □

The Algorithm. The code scheduling algorithm is presented in Figures 5.7 and 5.8. It is composed of three stages: pre-processing, marking/deleting and post-processing. In the pre-processing stage, S's flow graph is traversed in topological order, during which the conditions *Umove* and *Smove* are evaluated. A topological traversal ensures that whenever a node n is visited, all ancestor nodes in $DC(n, \mathrm{S})$ have already been processed; hence a single traversal is sufficient to evaluate these conditions. Then a "clone" S′ of S is created, which is used to hold the part of the flow graph to be transferred to D.

Next the algorithm searches for a critical trace, and if one exists it invokes subroutine "*Sched.*" *Sched* makes use of array "`mark`," each of whose entries corresponds to a node in S′. Whenever "`mark[`m, S′`] =` *true*," it means that node m will be "moved" into the destination section. *Sched* examines the critical trace in topological order, looking for node n such that $Umove(n)$ is *true*. If such a node n exists, then closure $DC(n, \mathrm{S})$ is generated; its non-predicate members are deleted from S, while all corresponding nodes in S′ are marked. If no unconditionally movable instruction exists, then a transformation of the speculative variety is attempted. And if *no* movable node is present, the program is forced to exit.

subroutine *Schedule_Section*(S, D, d, V_{bar})
input: source section S, destination section D, duration constraint d, and
a set V_{bar} of variables defined in sections lying between S and D.
begin
/* Pre-processing Stage */
foreach node n in S in topological order **do**
evaluate $Umove(n)$ and $Smove(n)$;
/* Marking/Deleting Stage */
make a copy S′ of S;
compute p in S such that $wt(p) = max\{wt(p') \mid entry(\mathrm{S}) \rightarrow^{p'} exit(\mathrm{S})\}$;
while $(wt(p) > d)$
call *Sched*$(p, \mathrm{S}, \mathrm{S}')$;
recompute p in S such that $wt(p) = max\{wt(p') \mid entry(\mathrm{S}) \rightarrow^{p'} exit(\mathrm{S})\}$;
end
/* Post-processing Stage */
delete all unmarked nodes from S′;
delete all predicate nodes guarding null code from S;
append S′ to end of D;
end

Figure 5.7: The Section Scheduling Algorithm (*Schedule_Section*)

At the end, if all critical traces were scheduled, the algorithm proceeds to a post-processing stage. If speculative transformation was carried out then S′ will, by definition, contain branching structures with empty predicate nodes. In this case, the nodes on the different branches of the predicate node are merged into a single block.

Finally, S′ is attached to the end of the destination section D. Cleaning up, the algorithm deletes control-predicates which guard empty nodes in section S – i.e., the "**if**" nodes whose corresponding bodies and γ-functions were completely transferred to S′.

Example Revisited. We return to our original flight controller example from Figure 5.3, and subject it to the code scheduling algorithm. The end-result appears in Figure 5.9. In scheduling S4, "*Sched*" unconditionally moves the function call `compRelAtt`, as well as the other nodes in its dependence closure. This reduces $wt(S4)$ by 0.25ms, which now stands at 2.57ms. Since $DUR(\mathrm{S4}) =$

subroutine $Sched(p, \mathrm{S}, \mathrm{S}')$
input: critical trace p, source section S, and a clone S$'$ of S.
begin
    if there is some $n \in p$ such that $Umove(n)$ holds **then**
      begin
          Select first such $n \in p$ such that $Umove(n)$ holds;
          foreach node $m \in DC(n, \mathrm{S})$ **do**
            mark$[m, \mathrm{S}']$:= *true*;
            if m is not a control-predicate **then** Delete m from S;
          end
      end
    else if there is some $n \in p$ such that $Smove(n)$ holds **then**
      begin
          Select first such $n \in p$ such that $Smove(n)$ holds;
          foreach node $m \in DDC(n, \mathrm{S})$ **do**
            mark$[m, \mathrm{S}']$:= *true*;
            if m is not a control-predicate **then** Delete m from S;
          end
      end
    else exit("Unable to synthesize.");
end

Figure 5.8: The Section Scheduling Algorithm (*Sched*)

2.25ms), further reductions are made by entering the speculative transformation phase; this results in moving one conditional branch and the function call `safeDtheta` beyond the immovable control-predicate `if (c3)`.

After the transformation, the implementation satisfies the necessary condition for consistency, since the body of S4 requires 2.14ms in the worst-case. Since $wt(\mathrm{S3}) = 0.68$ms is still less than $DUR(\mathrm{S3})$, the code scheduling successfully terminates without further scheduling S3.

In addition to such an instant benefit, the transformation converts possibly wasteful delay into useful computation time, since the new code in S3 can be scheduled within the delay interval between S2 and S4.

```
S6: (S6.start[p]≥p×20ms, S6.finish[p]≤p×20ms+5ms)
    {
S1:   {
        gx1 = μ(gx0, gx3);
        gy1 = μ(gy0, gy3);
        i1 = μ(i0, i3);

        input(GPS, &x1, &y1);                              [.1ms]
        input(NAV, &theta1);                               [.1ms]
        input(IMU, &roll1);                                [.1ms]
      }
S2:   {
        output(Cntrl_out, REQ_VEL);                        [.1ms]
      }
S3:   {
        c1 = GOAL[i1].passed;
        if (c1) {
          gx2 = GOAL[i1].x;
          gy2 = GOAL[i1].y;
          i2 = (i1+1)% NCOORD;
        }
        gx3 = γ(c1, gx2, gx1);
        gy3 = γ(c1, gy2, gy1);
        i3 = γ(c1, i2, i1);

        rtheta1 = compRelAtt(theta1, x1, y1, gx3, gy3);    [.25ms]
        c2 = |rtheta1| < EPS;
        if (c2)
          /* null */
        else
          dtheta3 = safeDtheta(rtheta1, roll1);            [.43ms]
      }
S4:   (S4.start≥S2.finish+0.75ms, S4.finish≤S2.finish+3.0ms)
      {
        input(Cntrl_in, &vel1);                            [.1ms]

        if (c2)
          dtheta1 = 0.0;
        else{
          c3 = vel1 < VHIGH;
          if (c3)
            dtheta2 = rtheta1;
          else
            /* null */
          dtheta4 = γ(c3, dtheta2, dtheta3);
        }
        dtheta5 = γ(c2, dtheta1, dtheta4);

        wflap = compFlapw(roll1, vel1, dtheta5);           [.95ms]
        throttle1 = compThrot(roll1, vel1, dtheta5);       [.89ms]
        output(THROT, throttle1);                          [.1ms]
        output(FLAP_Cntrl, wflap);                         [.1ms]
      }
S5:   { /* null */ }
    }
```

Figure 5.9: Flight Controller Program: After Code Scheduling

5.4 Summary

In this chapter we have addressed the problem of feasible code synthesis. We approach the problem via a two-step process, in that (1) an event-based TCEL program is decomposed into code-based tasks; and (2) unobservable code is scheduled to avoid feasibility faults. We attack the intractability of the code scheduling problem with a greedy approximation approach, based on Trace Scheduling.

Chapter 6

Transformation 2: Real-Time Task Slicing

After tasks have been successfully synthesized into feasible code, the next challenge lies in achieving schedulability of the entire application. Consider Figure 6.1. The two components at the bottom of the diagram correspond to the major elements involved in the tuning problem. "Schedulability" is, by definition, a key metric that drives our compiler-based task transformation tools. Thus close interactions between the real-time scheduler and the compiler tool are necessary. But this presents two competing demands: (1) it is desirable to maintain the traditional separation of concerns between compilation and scheduling; and (2) schedulability depends on complex task interactions that are often exposed at runtime.

In this chapter we present a compiler transformation method which satisfies these demands. We name the technique *real-time task slicing.* This transformation is specifically intended for periodic tasks in the domain of periodic, discrete control systems.

Ideally real-time task slicing should be complementary to feasible code synthesis, since both address two distinct system-tuning problems. Each must be solved before we get a correct real-time system. In practice, however, it is possible that their successive applications to the same task will produce a conflicting result. If we were to truly aim for optimal performance, feasible code synthesis would have to be geared for the specific schedulers. However we have already shown that the synthesis problem is, by itself, highly complex. This would add an additional layer of complexity.

In this thesis we treat the two phases as orthogonal. In doing so, we restrict ourselves to applying the real-time task slicing method only to simple "**every**" constructs with no nested "**do**" constructs within their bodies. Fortunately, many applications from discrete control domains possess this type of periodicity requirement.

To take a closer look at the problem of schedulability tuning, we open up the process diagram of

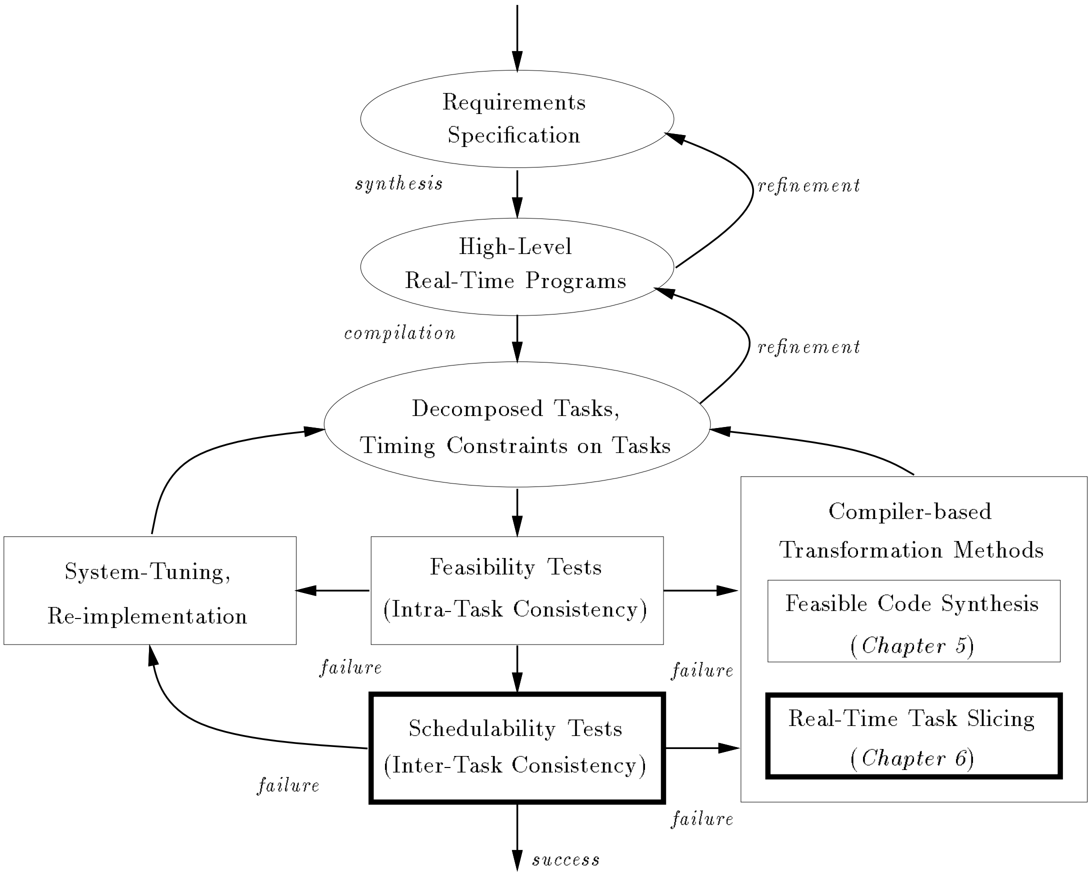

Figure 6.1: Software Development Process: Revisited for Real-Time Task Slicing

Figure 6.1, and re-draw the related components in Figure 6.2. The generalized task transformation method consists of three interrelated components: (1) schedulability tests, (2) a priority ordering algorithm, and (3) a real-time task slicer. In this chapter we explain these components, as well as the interactions. First, in Section 6.1 we discuss the characteristics of control domain software and fixed-priority preemptive scheduling algorithms. We also present the key idea of the generalized slicing method, and discuss the safety of the transformation method. We devote the remaining sections to the technical discussion of these three components.

6.1 Background

In this section we study two essential factors related to the real-time task slicing method, namely the characteristics of discrete control software and fixed priority preemptive scheduling. Since discrete

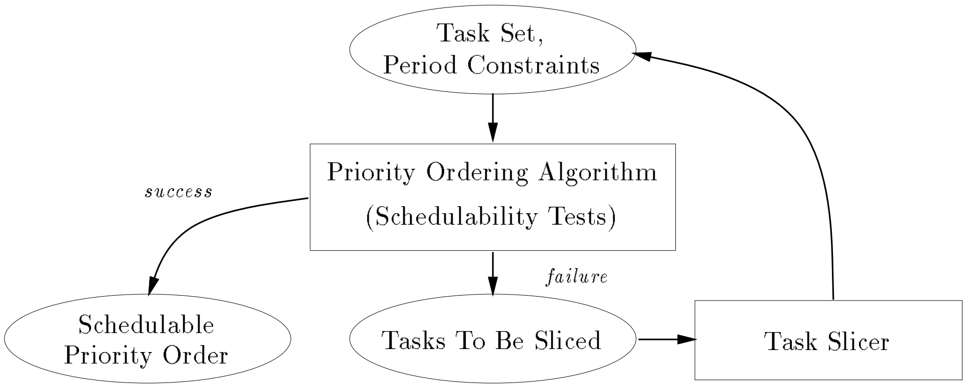

Figure 6.2: Generalized Slicing Method for Schedulability Tuning

control software possesses many representative properties that can be found in other applications (e.g., multimedia, vision, etc.), our approach can be easily adopted to other types of real-time systems.

6.1.1 Characterization of Discrete Control Software

Many discrete control algorithms possess computations that fit a fixed-rate algorithm paradigm [28], i.e., control-loops which execute repetitively with fixed periods. During each period, the physical world measurement data is sampled, and then actuator commands are computed. Meanwhile, a set of states is updated based on the current state and the sampled data.

The dynamic behavior of discrete control systems can often be expressed by the following equations.

$$\begin{aligned} Output_k &= g(State_k, Input_k) \\ State_{k+1} &= h(State_k, Input_k) \end{aligned}$$

In these equations, $Input_k$, $State_k$, and $Output_k$ respectively represent the input, current state, and output of the k^{th} period, while g is an output generation function and h is a state evolution function.

Since control equations are thought of as simultaneous relationships (and not as a computation procedure), there are usually many valid computational orderings. The usual practice is to choose a single ordering, and then to code it up as a cyclic control-loop whose k^{th} iteration is rendered

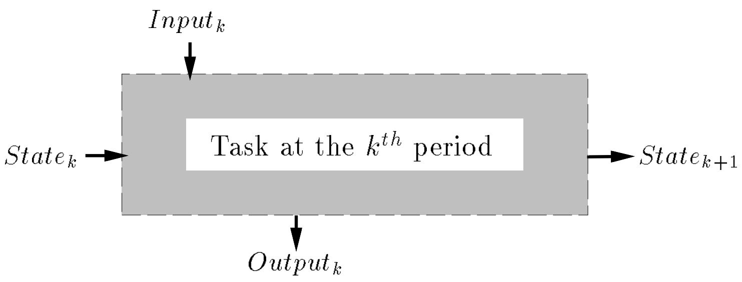

Figure 6.3: Task Instance at the k^{th} Period

in Figure 6.3. The actual loop structure is often driven by one's personal programming style, or perhaps the availability of generic code modules. But regardless of the choice (unless the underlying control laws are stateless), g and h mandate key precedence constraints, denoted by "$\prec$":

$$\begin{aligned}
Input_k &\prec Output_k \\
State_k &\prec Output_k \\
Input_k &\prec State_{k+1} \\
State_k &\prec State_{k+1}
\end{aligned}$$

The typical way to enforce these constraints is to use the "code-based" semantics, and ensure that each iteration of the control-loop completes by the end of its period. This means that the $k+1^{st}$ iteration starts only after the k^{th} iteration ends. Figure 6.4 illustrates the effect.

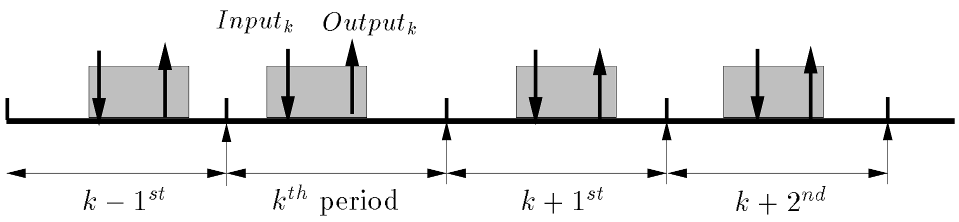

Figure 6.4: Dynamic Behavior of Periodically Implemented Control-Loop

6.1.2 Fixed-Priority Preemptive Scheduling

In any nontrivial system, there are usually many such tasks that must share the CPU and other resources. Thus they must be scheduled in a way that allows each of them to adhere to their timing constraints. Fixed-priority, preemptive scheduling algorithms are well-suited for control domain applications, not only because they possess periodic behavior, but also because efficient schedulability tests can be applied. Rate-monotonic scheduling, originally developed by Lui and Layland, is the first well-known algorithm of this kind. In their seminal paper [34] they proposed a priority assignment algorithm, in which a task with the shorter period is assigned the higher priority (hence the name rate-monotonic scheduling (RMS)). They also showed that such priority assignment is optimal in a sense that whenever it fails to find a feasible priority ordering, neither can any other static priority algorithm. However, their algorithm is applicable only to the periodic task model where tasks have fixed periods, deadlines are equal to periods, and tasks are totally independent of each other.

Recent research has made significant enhancements to this model which relaxes the original restrictions. In [31] Leung and Merrill showed that deadline monotonic priority assignment is also optimal where deadlines are shorter than periods. In [45] Sha *et al.* presented two protocols which enable tasks to interact via shared resources, while still guaranteeing the tasks' deadlines. Most recently, a group of researchers at the University of York developed a set of analytical techniques which can provide schedulability tests for broad classes of tasks, including those whose deadlines are greater than their periods. [5, 51, 50].

In this dissertation work we choose the York model, mainly because of its generality. The following notation is used in the remainder of this chapter.

- $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denotes a set of n tasks to be scheduled.
- T_i denotes the period of task τ_i.
- D_i denotes the deadline of task τ_i.
- c_i denotes the worst-case execution time of τ_i.

The fixed priority scheduling theory of York is based on response time analysis. The response time of task τ_i is defined as the time interval between when a request for τ_i arrives, and when τ_i finishes its execution servicing the request. If we can confirm that the maximum response time of τ_i is no greater than D_i, we can guarantee that τ_i will meet its deadline even in the worst-case. We first consider the simpler case where $D_i \leq T_i$.

Let R_i denote the maximum response time of task τ_i. Then R_i is computed as shown below.

$$R_i = c_i + \sum_{\tau_j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil c_j \qquad \textit{(Eq6.1)}$$

where $hp(i)$ is the set of higher priority tasks than τ_i. Observe that R_i is composed of two components, namely execution time c_i and interference. The interference, the second term of *Eq 6.1*, is the amount of time during which τ_i is preempted by the higher priority tasks in $hp(i)$ since the arrival of its request. As *Eq 6.1* is a recurrence equation on R_i, an iterative algorithm computes R_i by initially assigning it c_i, and then getting a new value until it converges on a fixpoint.

On the other hand, where there exists a task τ_i such that $D_i > T_i$, *Eq 6.1* is not sufficient. This is because uncompleted iterations of τ_i can now interfere with the current one. In this case the following general equation is used instead as discussed in [51].

$$R_i = \max_{q=0,1,2,\ldots} \{r_{i,q} - q \cdot T_i\} \qquad \textit{(Eq6.2)}$$

where

$$r_{i,q} = (q+1)c_i + \sum_{\tau_j \in hp(i)} \lceil \frac{r_{i,q}}{T_j} \rceil c_j$$

Consider the case of three periodic tasks, where the source of task τ_2 is given in Figure 6.5.

Task	Execution Time	Period	Deadline
τ_1	$c_1 = 400$	$T_1 = 1000$	$D_1 = 1000$
τ_2	$c_2 = 400$	$T_2 = 1600$	$D_2 = 1600$
τ_3	$c_3 = 570$	$T_3 = 2500$	$D_3 = 2500$

Since the periods are equal to the deadlines, rate-monotonic priority assignment is a natural choice. In the above table the row order corresponds to the priority order; *i.e.*, τ_1 is assigned the highest priority. We can carry out the response time analysis for these tasks using *Eq 6.1* as follows:

For τ_1:	R_1	$=$	400	$<$	$D_1 = 1000$
For τ_2:	R_2	$=$	$400 + \lceil 800/1000 \rceil 400 = 800$	$<$	$D_2 = 1600$
For τ_3:	R_3	$=$	$570 + \lceil 2570/1000 \rceil 400 + \lceil 2570/1600 \rceil 400 = 2570$	$>$	$D_3 = 2500$

We observe that the two high priority tasks τ_1 and τ_2 are schedulable, while τ_3 is not. (R_3 is greater than D_3 when τ_3 runs at priority 3.) In an effort to make the task set schedulable, we might try some hacking: e.g., by promoting τ_3 to the highest priority level. Although this makes τ_3 schedulable, it does not achieve the desired schedulability, since τ_2 will now be unschedulable.

```
        every 16ms
        {
L1:       input(Sensor, &data);
L2:       if (!null(data))
          {
L3:          t1 = F1(state);
L4:          t2 = F2(state);
L5:          t3 = F3(data);
L6:          t4 = F4(data);
L7:          state = F5(t1, t2, t3);
L8:          cmd = F6(t1, t3, t4);
L9:          output(Actuator, cmd);
          }
L10:      status_dump("logfile", cmd, state);
        }
```

Figure 6.5: TCEL Program for Task τ_2

Indeed, since the rate-monotonic assignment is optimal, no fixed priority assignment will suffice here – *unless the code-based semantics is abandoned!*

The reason the above task set is unschedulable is obvious: the computation demands of τ_3 exceed the available time. The simulated time line given in Figure 6.6 pictorially illustrates an unschedulable instance of τ_3. Thus, to make the task set schedulable, we must be able to postpone some computation out of the overloaded time frame. This transformation helps achieve schedulability, since we can utilize "idle" time intervals outside the congested period.

6.1.3 Scheduling with Compiler Transformations

When the task is found unschedulable, current engineering practice forces programmers to manually pick some critical tasks from the task set, and then to hand-optimize them. Such system-tuning is often repeated many times, until the entire task set finally gets schedulable. We aim to ease this process by providing a semi-automatic task transformation method, *real-time task slicing*.

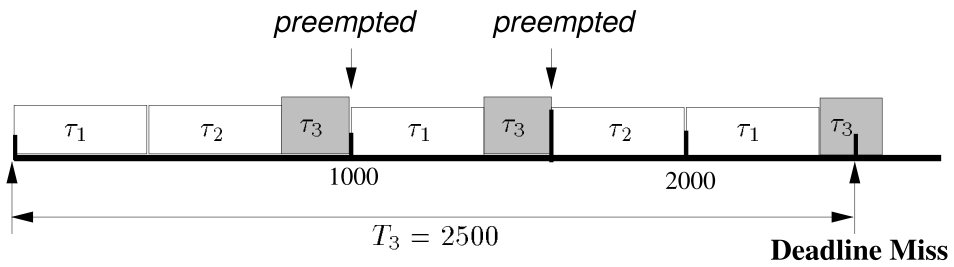

Figure 6.6: Simulated Time Line for the Example Task Set

Real-time task slicing is based on two simple observations; that is, (1) the current practice is to assume that an entire control-loop must finish by its deadline, but (2) the high-level TCEL semantics mandates only the observable event operations be finished within the task's time frame.

The key idea of the task slicing method is as follows. We decompose a task τ into two subtasks: one containing all observable event operations, and the other all remaining local operations. We call the former the *IO-hander* and the latter the *state-update* component, and we denote them by τ^{IO} and τ^{State}, respectively. Figure 6.7 demonstrates the decomposition of the control-loop task originally shown in Figure 6.3.

After the decomposition, we ensure that the IO-handler subtask will execute within its allowable time frame. On the other hand, we may postpone the execution of the state-update subtask under the worst-case task phasing. Finally, we maintain precedence constraints between τ^{IO} and τ^{State}, originally induced by the task's data and control dependences.

The task decomposition itself is carried out by the program slicing technique. As we stressed above, we put the greatest emphasis on preserving the timing behavior of observable events and the precedence constraints derived in Subsection 6.1.1. Before presenting the systematic slicing procedure we describe our approach using our example task set.

Assume that slicing τ_2 yields the greatest benefit in schedulability. We decompose τ_2's code into IO-handling τ_2^{IO} and state-update τ_2^{State}, as shown in Figure 6.8. Their computation times are separately calculated as follows:

$$c_2^{IO} = 2.2\text{ms}, \quad c_2^{State} = 1.9\text{ms}$$

Note that the sum of the two execution times is slightly greater than the original execution time

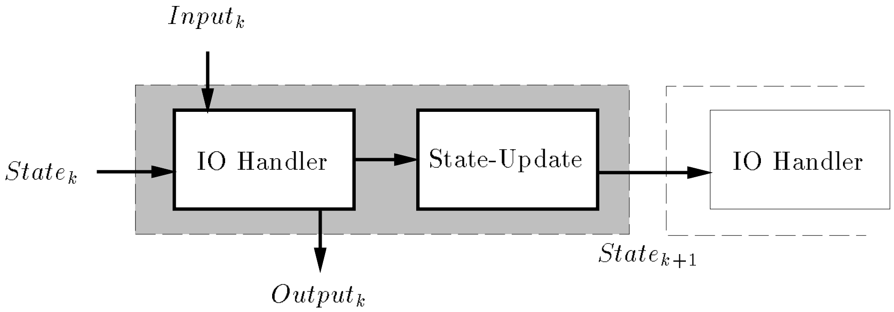

Figure 6.7: Decomposed Task at the k^{th} Period

```
/* Subtask τ2^IO */
  input(Sensor, &data);
  c = !null(data);
  if (c)
  {
    t1 = F1(state);
    t3 = F3(data);
    t4 = F4(data);
    cmd = F6(t1, t3, t4);
    output(Actuator, cmd);
  }
```

```
/* Subtask τ2^State */
  if (c)
  {
    t2 = F2(state);
    state = F5(t1, t2, t3);
  }
  status_dump("logfile", cmd, state);
```

Figure 6.8: Two Decomposed Subtasks of Task τ_2

4.0ms of τ_2.

To enforce that the precedence constraints between the subtasks, we rewrite them for τ_2: (1) the i^{th} instance of τ_2^{IO} must finish before the i^{th} instance of τ_2^{State}; and (2) the k^{th} instance of τ_2^{State} must finish before the $k+1^{st}$ instance of τ_2^{IO} starts. Thus, the scheduling and dispatching subsystems must guarantee the following execution behavior of τ_2.

$$\tau_2^{IO} \rightarrow \tau_2^{State} \rightarrow \tau_2^{IO} \rightarrow \tau_2^{State} \rightarrow \ldots$$

The net result of the transformation of τ_2 is as follows: within the worst-case time interval of $2T_2$, we may only see one whole instance of τ_2^{State}.

Remainder of This Chapter. Throughout the remainder of this chapter we discuss the generalized slicing method for schedulability tuning, explaining the details of the three components. In Section 6.2 we present the program slicing algorithm which is the crux of our transformation. In Section 6.3 we discuss new schedulability tests, and then show how they are used, by way of a motivating example. In Section 6.4 we introduce a priority ordering algorithm that is capable of making feasible slicing decisions, as well as finding a feasible priority order for a given task set. To demonstrate the effectiveness of this algorithm we show the result of an experiment we conducted on a task set drawn from an avionics platform.

6.2 Automatic Task Decomposition by Program Slicing

The idea behind the task decomposition is, as discussed in Subsection 6.1.3, to accept a task and then generate its two code components: one that corresponds to a subthread triggering all observable events, and the other that corresponds to a subthread computing the next-state update. Straightforward as it may look, the decomposition can be a very complex compiler problem. Many factors make this the case, among which are intertwined threads of control, nested control structures, complex data dependences between statements, procedure calls in the task code, etc. To cope with these problems in a systematic manner, we harness a novel application of *program slicing* [41, 52, 53]. For the sake of brevity, we assume the following:

- Function calls are inlined.
- Loops are unrolled.
- The intermediate code of programs is translated into static single assignment form [8, 21, 16].

The first assumption allows us to avoid *interprocedural* slicing [22]. The next two assumptions simplify problems induced by spurious data dependences such as anti-dependences and output dependences [4]. However, we can partially alleviate the restrictions, relying on dependence breaking transformations, such as scalar expansion [4]. Static single assignment is one such transformation. However, such methods will unfortunately lead to overly-conservative analysis – a limiting factor in the algorithm.

6.2.1 The Program Slicing Algorithm

Informally a *slice* of program P with respect to program point p and expression e consists of P's statements and control predicates that may affect the value of e at point p. We call a pair $\langle p, e\rangle$ a *slicing criterion*, and denote its associated slice by $P/\langle p, e\rangle$. The result is that we can execute the slice $P/\langle p, e\rangle$ to obtain the value of e at location p. Recall our periodic controller task τ_2 of Figure 6.5. The following fragment is the slice $\tau_2/\langle \text{L9}, \texttt{cmd}\rangle$.

```
L1:   input(Sensor, &data);
L2:   if (!null(data))
      {
L3:     t1 = F1(state);
L5:     t3 = F3(data);
L6:     t4 = F4(data);
L8:     cmd = F6(t1, t3, t4);
L9:     output(Actuator, cmd);
      }
```

Statements L1, L3, L5, L6 and L8 are included in the slice, because variable "`cmd`" depends on their computations (this is called *data dependence*). Also, statement L9 is included because it generates an observable event.[1] Finally, the predicate on line L2 is included, because the execution of statements L3, L5, L6, L8 and L9 (hence the value of "`cmd`") depends on the boolean outcome of the predicate (this is called *control dependence*).

Thus the computation of slices is based on data dependence as well as control dependence. In this regard, using a *program dependence graph* [11, 22, 41] is ideal, since it represents both types of dependences in a single graph. (See Section 4.2 for definition.)

The program dependence graph *PDG* of our controller task τ_2 is shown in Figure 6.9.

[1]We intentionally include L9, as will be discussed in Algorithm 6.1.

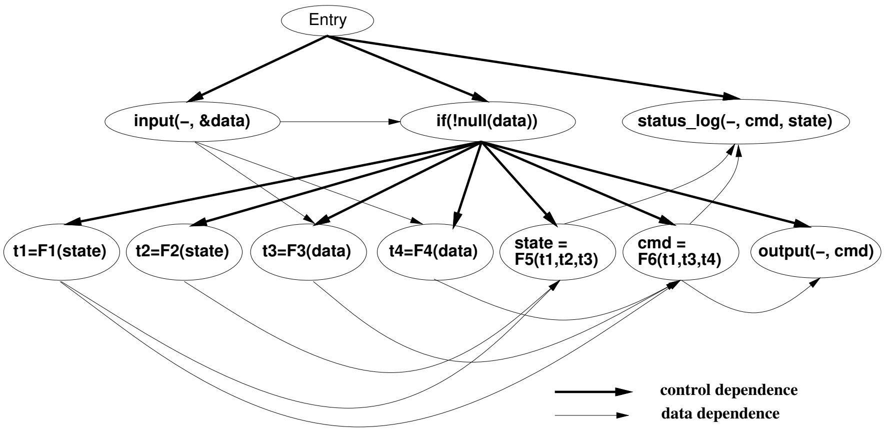

Figure 6.9: Program Dependence Graph

The slice of program P with respect to program point p and expression e (i.e., $P/\langle p, e\rangle$) can be obtained through a traversal of P's program dependence graph. We can extend the definition of a program slice for a set of slicing criteria C in a way that $P/C = \bigcup_{\langle p,e\rangle \in C} P/\langle p, e\rangle$. A simple algorithm to compute the slice is given below. In the algorithm the program point p corresponds to a vertex of PDG.

Algorithm 6.1 *Computes the slice $P/\langle p, e\rangle$:*

Step 1 Compute reaching definitions $RD(p, e)$.

Step 2 Compute the slice by a backward traversal of PDG such that

$$P/\langle p, e\rangle = \{m \mid \exists n \in RD(p, e) \,:\, m \Rightarrow_* n\} \cup \{p\}.$$

Figure 6.10 shows the graph that results from taking a slice of the program dependence graph in Figure 6.9 with respect to criterion $\langle \mathrm{L9}, \mathtt{cmd}\rangle$.

One of the essential points in using our task decomposition algorithm is providing right slicing criteria for the algorithm, so that the computed I/O slice of a task "covers" all the observable behaviors of the original task. Criteria selection can be automated by means of the observable event specification, or it can be manually performed by way of graphical user interface.

Let $C_{IO}(\tau)$ be a set of slicing criteria for I/O slice of task τ. Then the task decomposition

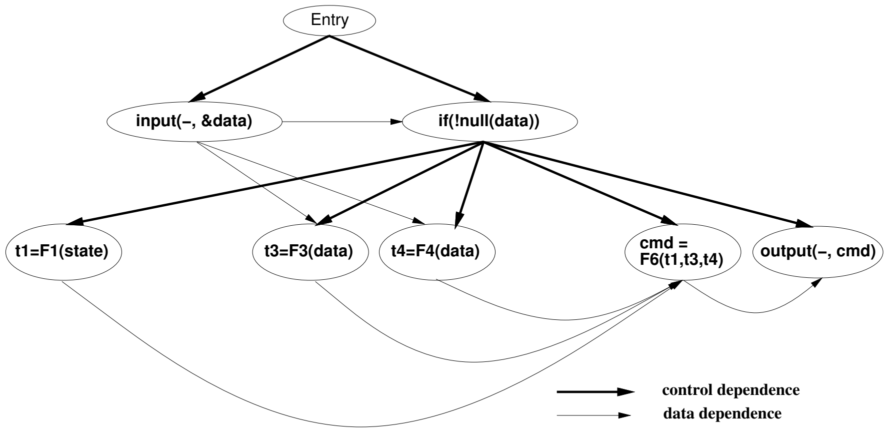

Figure 6.10: Slice with respect to Criterion $\langle$L9, `cmd`$\rangle$

algorithm is given below:

Algorithm 6.2 *Decompose task τ into τ^{IO} and τ^{State}:*

Step 1 Compute the slice of τ with respect to $C_{IO}(\tau)$ using Algorithm 6.1. The generated slice $\tau / C_{IO}(\tau)$ becomes τ^{IO}.

Step 2 Delete from τ all repeated statements of τ^{IO} except for the conditional statements. The remaining code becomes τ^{State}.

Figure 6.8 shows the two subtasks τ_2^{IO} and τ_2^{State} of τ_2 computed by Algorithm 6.2 with slicing criteria $C_{IO}(\tau) = \{\langle L1, data\rangle, \langle L9, cmd\rangle\}$.

6.2.2 Assigning Times to Subtasks

Program slicing may well increase worst-case execution times of tasks for a number of reasons: (1) control structures are replicated and will be executed twice; (2) splitting a basic block may increase the number of register load and store operations [2]; and (3) worst-case execution time paths of the two resultant subtasks may be incorrectly derived. We take a close look at the last factor, since it tends to take up the greatest portion of the increase, though it is not a genuine cause of the increase, but an artifact of overly-conservative timing prediction.

After a conditional of a task is sliced and then spliced, the worst-case execution time of the new task may be increased unless we carefully correlate the duplicated condition predicates. Figure 6.11 pictorially depicts this case. The original task τ consists of one conditional, one branch of which is IO-generating code "`IO`" and the other is state-update code "`ST`." The worst-case execution time of τ is:

$$wt(\tau) = wt(\mathtt{c}) + \max\{wt(\mathtt{IO}), wt(\mathtt{ST})\}.$$

In Figure 6.11 τ is sliced into two subtasks τ^{IO} and τ^{State}. Their worst-case execution times are also given below.

$$\begin{aligned} wt(\tau^{IO}) &= wt(\mathtt{c}) + wt(\mathtt{IO}) \\ wt(\tau^{State}) &= wt(\mathtt{c}) + wt(\mathtt{ST}). \end{aligned}$$

Consequently, the worst case execution time of the transformed task τ' ($\equiv \tau^{IO}; \tau^{State}$) may be measured as:

$$wt(\tau') = 2 \cdot wt(\mathtt{c}) + wt(\mathtt{IO}) + wt(\mathtt{ST}),$$

which is much larger than $wt(\tau)$.

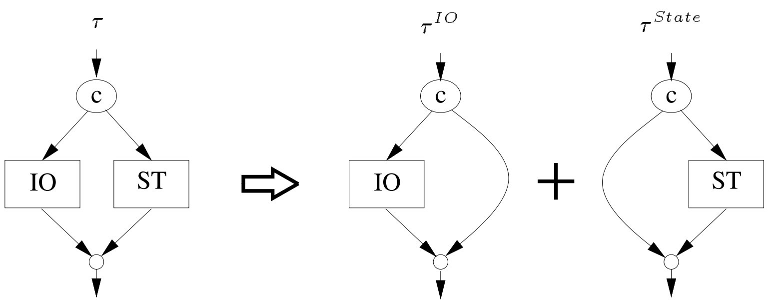

Figure 6.11: Slicing a Conditional

However, tighter worst-case execution time can be easily obtained by correlating the conditional predicate of the subtask τ^{IO} with that of the subtask τ^{State}. For example, in Figure 6.11 if "`IO`" is executed in subtask τ^{IO}, then we know that the empty left branch will be executed in subtask τ^{State}. Thus $wt(\tau')$ can be refined as follows:

$$wt(\tau') = 2 \cdot wt(\mathtt{c}) + \max\{wt(\mathtt{IO}), wt(\mathtt{ST})\}$$

For the given two subtasks of τ_i, we carry out the following simple steps which are based on the notion of predicate correlation to compute tight worst-case execution times of subtasks τ_i^{IO} and τ_i^{State}.

Step 1 Calculate c_i' by running a timing tool with the code of τ_i'.

Step 2 Calculate c_i^{IO} by running a timing tool with the code of τ_i^{IO}.

Step 3 Calculate c_i^{State} such that $c_i^{State} = c_i' - c_i^{IO}$.

This will serve as a good rough estimate for the transformed task code. Then we can use a profiler to account for the two other factors that incur timing overhead.

6.3 Scheduling Alternatives and Their Analyses

We now sketch a simple, high-level procedure that uses slicing to transform an unschedulable task set into a schedulable one. The input is set Γ of n tasks, processed in order from τ_n to τ_1. If the task set is found unschedulable, the slicer algorithm is invoked to decompose τ_n into its two constituent threads, which then replace τ_n in Γ. If the updated set is still deemed unschedulable, the procedure goes to work on τ_{n-1}, and so on.

In [15] we present a detailed alternative to this approach, in which tasks are processed from τ_1 to τ_n; i.e., the first task found unschedulable is selected for slicing. One can imagine other alternatives as well.

However, any such scheme is critically dependent on two elements:

(1) A scheduling policy that can exploit our task model; i.e. while the τ^{State} threads can miss their original deadlines, the precedence constraints between instances τ^{IO} and τ^{State} must be maintained.

(2) An offline schedulability analyzer for the given scheduling policy.

In this section we present an analytic approach that systematically addresses (1) and (2).

A static priority approach: In [15] we present a RMS based method which enjoys a simple priority assignment rule and analysis test for its dual-priority scheme. This is certainly one of its strengths. Its principal weakness is that the online component lacks the simplicity found in pure, static priority scheduling due to its semi-dynamic dual priority assignment.

Thus the following question arises: when can a set of transformed TCEL tasks be scheduled under a fully preemptive, static priority scheme? Burns [6] provides an answer to this question after identifying a simple, but essential fact about the TCEL task model. That is, whenever we let a task's deadline be greater than its period, this represents a relaxation of the classical rate-monotonic restrictions put forth in [34]. Thus the rate-monotonic priority assignment may not be the optimal one.

Given set Γ' of transformed TCEL tasks

$$\begin{aligned} \tau_1' &= \tau_1^{IO}; \tau_1^{State} \\ \tau_2' &= \tau_2^{IO}; \tau_2^{State} \\ &\vdots \\ \tau_n' &= \tau_n^{IO}; \tau_n^{State} \end{aligned}$$

it turns out the appropriate priority assignment is not only dependent on the deadlines (as in the pure deadline-monotonic model), but also on the respective execution times of each IO-handler and state-update component. In [6] Burns presents a search algorithm to generate the feasible static-priority order – or to detect when no such order exists. Thus the approach includes the following components.

Online Scheduler: This is a simple, preemptive dispatching mechanism, in which priority "ties" are broken in favor of the task dispatched first. Thus, for example, a task's current iteration will finish before the next one starts.

Offline Analyzer: The analyzer is *constructive*, in that it produces a feasible priority assignment if one exists. If no such assignment exists, perhaps the programmer may have to go back to the system design step and reply on more aggressive system-tuning.

For given task set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ Burns' priority assignment algorithm accepts the pre-processed task set $\Gamma' = \{\tau_1', \tau_2', \ldots, \tau_n'\}$ as its input, where τ_i' is a sliced version of τ_i. It then begins looking for a task that can run at the lowest priority (level n)[2]. After such a task, say, τ_k' is found, the algorithm proceeds to search the new task set $\Gamma' - \{\tau_k'\}$ for the second lowest priority task, and so on. There is an important fact that leads to the optimality of this algorithm: while a task is being tested for priority level p, all $p - 1$ tasks whose priorities have not yet been assigned are assumed to run at a higher priority. In fixed-priority, preemptive scheduling, since a lower priority task can never preempt the higher priority tasks, selections made for priority levels p or below will not affect those above p.

[2] For n tasks, n denotes the lowest priority level, and 1 the highest.

During this priority ordering, the schedulability test of τ_i' ($\equiv \tau_i^{IO}; \tau_i^{State}$) for priority p yields the following two conditions:

(1) Whether τ_i^{IO} can always run within time D_i at priority p, and

(2) Whether q consecutive iterations of τ_i' can run within $q \cdot T_i + D_i$ at priority p where $q \geq 1$.

Condition (1) is required by the TCEL's semantics; condition (2) accounts for the case where at least one iteration of τ_i^{State} is delayed. The schedulability test boils down to a check to see if the maximum response time of τ_i^{IO} is no greater than D_i in either case.

The maximum response time (denoted by R_i^{IO}) of τ_i^{IO} with respect to $hp(i)$ is computed as below:

$$r_{i,q} = q(c_i^{IO} + c_i^{State}) + c_i^{IO} + \sum_{\tau_j \in hp(i)} \lceil \frac{r_{i,q}}{T_j} \rceil c_j$$

$$R_i^{IO} = \max_{q=0,1,2,\ldots} \{r_{i,q} - q \cdot T_i\} \qquad \textit{(Eq6.3)}$$

We must subtract $q \cdot T_i$ from $r_{i,q}$ to obtain the real response time, since $r_{i,q}$ is measured from the start of the q^{th} period prior to the current period. Although q is denoted as an unbounded number in *Eq* 6.3, it can be trivially shown that there exists bounded response time R_i^{IO} as long as utilization of tasks in $hp(i)$ and τ_i is less than 100% [51]. The value of q is bounded below by p such that $r'_{i,p} < p \cdot T_i + D_i$ where

$$r'_{i,p} = (p+1)(c_i^{IO} + c_i^{State}) + \sum_{\tau_j \in hp(i)} \lceil \frac{r'_{i,p}}{T_j} \rceil c_j.$$

The intuition behind this is the execution pattern of τ_i' repeats after the entire execution of τ_i' fits within its time frame.

Now recall the unschedulable task set we showed in Subsection 6.1.2. Suppose that only τ_2 was sliced. This requires priority rearrangement among the tasks, since RMS is no longer optimal in the transformed task model. The result of new priority ordering is as follows:

$$\tau_3 \prec \tau_1 \prec \tau_2'$$

In the next section we show how this ordering is obtained. But given that we have an ordering, we can check it using *Eq 6.3*.

For τ_3:				
R_3	$=$	570	$<$	$D_3 = 2500$
For τ_1:				
R_1	$=$	$400 + \lceil 970/2500 \rceil 570 = 970$	$<$	$D_1 = 1000$
For τ_2':				
$r'_{2,1}$	$=$	$2 \cdot 410 + \lceil 2750/2500 \rceil 570 + \lceil 2750/1000 \rceil 400 = 2750$	$<$	$T_2 + D_2 = 3200$
$r_{2,0}$	$=$	$220 + \lceil 1590/2500 \rceil 570 + \lceil 1590/1000 \rceil 400 = 1590$		
$r_{2,1}$	$=$	$410 + 220 + \lceil 2400/2500 \rceil 570 + \lceil 2400/1000 \rceil 400 = 2400$		
R_2^{IO}	$=$	$\max\{1590, 2400 - 1600\} = 1590$	$<$	$D_2 = 1600$

As a result, the task set is shown to be schedulable under the new priority assignment.

6.4 Priority Ordering with Task Slicing

As discussed in Section 6.3, Burns' priority assignment algorithm expects that all tasks in Γ are sliced before they are submitted for priority assignment. However, it is typically not desirable to slice all tasks in the application due to execution time overhead incurred by task slicing. As an example, consider a task set whose utilization is 0.96. Suppose that task slicing uniformly increases the worst-case execution times of the tasks by 5%. If we naively slice all the tasks, this will result in utilization of 1.008 and render the task set permanently unschedulable.

Moreover, since we view slicing as a means of tuning an application, it should selectively be applied to tasks which will realize the greatest benefit.

To address this problem, we present an algorithm that not only finds a feasible task priority ordering, but also picks only a small subset of tasks to slice. For a given ordered list of tasks $\Gamma = [\tau_1, \tau_2, \ldots, \tau_n]$, we make the following definitions.

- sliced(τ_i) : a boolean variable denoting whether or not τ_i is sliced.
- $c'_i = c_i^{IO} + c_i^{State}$.

We refer to a certain permutation Γ' of Γ as a *configuration*, i.e. Γ' denotes a priority ordering[3] of the tasks in Γ, and sliced(τ_i) is defined for all $\tau_i \in \Gamma'$. There are $n!$ different priority orderings, and

[3] The first task in the list has the highest priority.

2^n possible slicing choices. Thus the algorithm's job is to choose a task configuration among $2^n \cdot n!$ distinct ones in an efficient manner.

Definition 6.1 (Feasibility) For a given Γ, a configuration Γ' is said to be *feasible iff* all tasks in Γ' meet their deadlines under the priority ordering and slicing choice denoted by Γ'.

6.4.1 Feasibility Test

Since Burns' algorithm accepts a pre-sliced application, it can compute the exact amount of interference from the higher priority tasks when it considers a task for a priority. On the other hand, our problem is to slice for schedulability. (Recall that for a task τ_i, $c_i \neq c_i^{IO} + c_i^{State}$.) Thus it seems inevitable to search the entire solution space of size $2^n \cdot n!$ in order to find a feasible task configuration.

Fortunately, there are cases where we can make a slicing decision without exhaustively exploring the search space. We rely on the response time analysis summarized by equations *Eq 6.2* and *Eq 6.3* to find these cases. To be specific, we make use of the following schedulability test.

$$
\boxed{
\begin{array}{l}
\mathit{Feasible}(\mathcal{L}, \tau_k) \equiv \\
\quad \textbf{if } \neg\mathrm{sliced}(\tau_k) \textbf{ then } \max_{q=0,1,2\ldots}\{r_{k,q} - q \cdot T_k\} \leq D_k \\
\quad \textbf{else } \max_{q=0,1,2\ldots}\{r'_{k,q} - q \cdot T_k\} \leq D_k \\
\text{where} \\
\quad S = \{\tau_i \in \mathcal{L} \mid \mathrm{sliced}(\tau_i)\}, \\
\quad r_{k,q} = (q+1)c_k + \sum_{\tau_j \in \mathcal{L}-S} \lceil \frac{r_{k,q}}{T_j} \rceil c_j + \sum_{\tau_j \in S} \lceil \frac{r_{k,q}}{T_j} \rceil c'_j, \text{ and} \\
\quad r'_{k,q} = q \cdot c'_k + c_k^{IO} + \sum_{\tau_j \in \mathcal{L}-S} \lceil \frac{r'_{k,q}}{T_j} \rceil c_j + \sum_{\tau_j \in S} \lceil \frac{r'_{k,q}}{T_j} \rceil c'_j.
\end{array}
}
$$

"$\neg\mathrm{sliced}(\tau_k) \wedge \mathit{Feasible}(\mathcal{L}, \tau_k)$" denotes that the unsliced τ_k is schedulable with tasks in $\mathcal{L}$ running at higher priorities. Similarly, "$\mathrm{sliced}(\tau_k) \wedge \mathit{Feasible}(\mathcal{L}, \tau_k)$" means that τ_k, when being sliced, is schedulable with tasks in $\mathcal{L}$.

6.4.2 The Algorithm

We now present the priority ordering algorithm in Figure 6.12. We describe below variables we use in the algorithm.

- $\Gamma = [\tau_1, \tau_2, \ldots, \tau_n]$ is the input task list which is initially ordered in nondecreasing order of the deadlines. Such a deadline monotonic ordering is desirable as a starting point, since most tasks, except for a small number of tasks to be sliced will be consistent with the ordering.

- The two parameters $\mathcal{L}_1$ and $\mathcal{L}_2$ of function *Search* collectively hold the list of tasks to be priority-ordered. In every invocation, function *Search*($\mathcal{L}_1, \mathcal{L}_2$) returns either the priority-ordered list of the tasks or *false*, if it cannot find any feasible ordering among them.

We use operator "@" to denote *list append* operation.

In every invocation, function *Search* attempts to assign the last task in $\mathcal{L}_1$ (τ in Figure 6.12) priority level $|\mathcal{L}_1|+|\mathcal{L}_2|$. The condition on line (1) denotes that the algorithm has already generated a complete task configuration of Γ.

The condition on line (2) means that the algorithm has checked all tasks in lists $\mathcal{L}_1$ and $\mathcal{L}_2$ for priority level $|\mathcal{L}_1| + |\mathcal{L}_2|$, but it can assign none of them that priority. Thus *false* is returned.

When the condition on line (3) holds, the algorithm checks if τ is feasible with current high priority tasks. Before doing this, the algorithm attempts to find a feasible priority ordering among the higher priority tasks (on line (4)). If it cannot do so, τ is infeasible in the current priority level, and thus the algorithm tries other task for the current priority level, invoking tail recursion on line (12).

On the other hand, if the higher priority tasks are schedulable, the algorithm checks if τ is feasible. If so, it returns a new ordering $L@[\tau]$. Otherwise, the algorithm slices τ and sees if τ' is feasible. If τ' is deemed feasible, it returns $L@[\tau']$.

6.4.3 A Larger Example

We constructed a task set consisting of eighteen periodic tasks, based on the avionics application described in [35, 51]. We added tighter deadlines to the original task set, and modified the execution times of some of the tasks. As a result, the task set had utilization of 0.836, and it was unschedulable. The resultant timing specification of our task set is given in Table 6.1 where the time unit is 1 microseconds.

We make the following assumptions for the task set, which we have found representative.

1. Only small portion of a task – no more than 25 % of the original task code in terms of the worst case execution time – can be sliced.

2. Slicing incurs no more that 5 % increase in a task's worst-case execution time.

```
        algorithm PriAssign(Γ)
        begin
          return(Search(Γ, []));
        end

        list function Search(L1, L2)
        case
(1)       when L1 = L2 = []: return([]);
(2)       when L1 = [], L2 ≠ []: return(false);
(3)       when L1 = L1'@[τ]:
(4)           L = Search(L1'@L2, []);
(5)           if L ≠ false then
(6)             if Feasible(L, τ) then
(7)               return(L@[τ]);
(8)             else
(9)               τ' = Slice(τ);
(10)              if Feasible(L, τ') then
(11)                return(L@[τ']);
                end
              end
(12)          return(Search(L1', [τ]@L2));
        end
```

Figure 6.12: Algorithm for Priority Ordering with Slicing Decision

When we ran the priority ordering algorithm with the task set in Table 6.1, it chose to slice tasks τ_4, τ_7 and τ_{16}, and made the task set schedulable. The utilization grew slightly to 0.844. We show the result in Table 6.2, where "R" denotes the maximum response time of an unsliced task, and R^{IO} and R^{State} respectively represent the maximum response times of the two components of an sliced task.

6.5 Summary

In this chapter we addressed the schedulability tuning problem for real-time applications in discrete control domains. We solved this problem with a two-tier approach. At one end was a semantics-based compiler transformation method that safely segregates two subthreads from given task code; at the other end was an extended schedulability analysis test, and its associated feasible priority ordering algorithm. While the compiler-based tool had to be carefully guided by the scheduling

	T	D	c	c^{IO}	c^{State}
τ_1	1000	1000	51	51	0
τ_2	25000	5000	2000	1600	500
τ_3	25000	5000	1000	800	250
τ_4	40000	5000	2000	1600	500
τ_5	50000	20000	3000	2400	750
τ_6	200000	20000	3000	2400	750
τ_7	50000	25000	5000	4000	1250
τ_8	59000	25000	8000	6400	2000
τ_9	80000	80000	9000	7200	2250
τ_{10}	80000	80000	2000	1600	500
τ_{11}	100000	80000	8000	6400	2000
τ_{12}	100000	100000	5000	4000	1250
τ_{13}	200000	100000	3000	2400	750
τ_{14}	200000	100000	1000	800	250
τ_{15}	200000	120000	1000	800	250
τ_{16}	200000	140000	2000	1600	500
τ_{17}	1000000	1000000	1000	800	250
τ_{18}	1000000	1000000	1000	800	250

Table 6.1: Example Task Set

component in our approach, the conventional separation of concerns between compilation and scheduling could still be maintained.

	T	D	c	c'	R	R^{IO}	R^{State}	Sliced?
τ_1	1000	1000	51	51	51	0	0	n
τ_2	25000	5000	2000	2100	2153	0	0	n
τ_3	25000	5000	1000	1050	3204	0	0	n
τ_4	40000	5000	2000	2100	0	4855	5406	y
τ_6	200000	20000	3000	3150	8559	0	0	n
τ_5	50000	20000	3000	3150	11712	0	0	n
τ_8	59000	25000	8000	8400	20171	0	0	n
τ_7	50000	25000	5000	5250	0	24375	28829	y
τ_9	80000	80000	9000	9450	38339	0	0	n
τ_{10}	80000	80000	2000	2100	42643	0	0	n
τ_{11}	100000	80000	8000	8400	71372	0	0	n
τ_{12}	100000	100000	5000	5250	79780	0	0	n
τ_{13}	200000	100000	3000	3150	96747	0	0	n
τ_{14}	200000	100000	1000	1050	97798	0	0	n
τ_{15}	200000	120000	1000	1050	98849	0	0	n
τ_{16}	200000	140000	2000	2100	0	139890	140441	y
τ_{17}	1000000	1000000	1000	1050	141492	0	0	n
τ_{18}	1000000	1000000	1000	1050	142543	0	0	n

Table 6.2: Priority Assignment with Program Slicing

Chapter 7

Practical Considerations and Prototype Implementation

The TCEL paradigm helps incorporate a higher level of abstraction into real-time domains. As we have shown, TCEL's event-based semantics constrains only those operations that are critical to real-time operation; i.e., the events denoted in the specification or those derived from it. As such, a source program is an appropriate representation of the designer's intentions, and it need not overburden the system with unnecessary constraints. Moreover, the event-based semantics enables our compiler tools to transform the program, and helps resolve conflicts between the timing constraints and the code's actual execution time. Since this is exactly the type of dirty work that compilers do best, a human programmer's time is probably better spent elsewhere.

On the other hand, the success of our compiler tools is contingent upon the limitations of static program analysis techniques. In this chapter we consider such limitations, and show how we can work around the problems associated with them.

We faced these problems in our implementation of the real-time slicer tool which uses an existing data and control dependence analyzer. We finish this chapter with a discussion of the prototype implementation which we call TimeWare/SLICE.

7.1 Practical Considerations.

For the sake of brevity we presented the code scheduler and the program slicer in a rather idealized form, abstracting out some implementation-related considerations. For example, we assume that a given program is in its perfect GSA form so that all spurious dependences are effectively removed. We also assume that available timing tools provide fairly tight execution time bounds for small code segments. During our research these implementation-related factors revealed themselves via three sources: (1) experiences in building our prototypes, (2) experiences in dealing with large programs, and (3) discussions with colleagues who design and build production-quality real-time systems. In the following subsections we briefly summarize several of these considerations.

7.1.1 Limits of Data-Flow Analysis

Both of the code scheduler and the program slicer heavily rely on contemporary compiler methods, including intra- and inter-procedural data and control analysis. And as with all program transformation algorithms, the limitations of this enabling technology become a constraining factor of our approach. For example, current static data-flow analysis is incapable of disambiguating all pointer aliases (which at worst is an undecidable problem). Thus we cannot always translate the TCEL source into its corresponding "perfect" GSA form. We partially assuage the problem by adopting techniques such as (1) inlining procedures to avoid inter-procedural aliases; (2) rendering in GSA form only those assignments that contain statically analyzable variables; and (3) unrolling loop bodies. Of course these and similar methods will degrade the code scheduler's performance, either by increasing the amount of code, or by decreasing its efficacy. However, dependence analyzers are improving at a rapid rate, and our algorithm will improve along with them. For example, if we incorporate the recent advances in loop dependence analyses such as those in the *Omega Test* [43], we may not have to unroll loops to slice a real-time task. We can obtain better slices for loops using techniques like *loop distribution.*

7.1.2 Limits of Timing Analysis

Another limiting factor is the difficulty of achieving accurate, static timing analysis in the face of more complicated architectures. Quite simply, it has become incredibly difficult to use vendor-supplied benchmarks, and to model the interplay between pipelines, hierarchical caches, shared memories, register windows, etc. Thus with an approach like ours, it seems meaningless to predict the execution time of a single instruction (or even a small block). First, the CPU time will probably be too small to make a difference in achieving feasibility, and second, the "noise" in the prediction will be too large.

Thus we have adopted a hierarchical abstraction approach to deal with time predictions. For example, in the program of Figure 5.3 we accounted only for the CPU-intensive function calls that performed complex operations, while ignoring the execution time of finer-grained instructions. The same approach can be used on larger-grained structures. Our experience shows the compiler should usually hunt for the "big-game targets," and forget about the smaller ones.

However after code scheduling or program slicing is completed, it becomes imperative to verify the result with a more sophisticated timing tool; for example, a good profiler. Performing such re-timing is especially important in a cached memory structure, where code scheduling will always change the instruction alignment. We note that all modern RISC compilers re-order instructions to some degree; thus the efficacy of *any* source-level timing analysis is diminishing.

7.1.3 User Interaction

The above two factors argue against the fully automated code synthesis and program slicing tools. There is also a third factor, which we discussed in reference to speculative code motion. That is, programmers of production-quality, real-time systems will simply not accept a compiler technology that "outsmarts" them, and possibly "disobeys" their intentions. They will, however, accept a tool that helps tune their systems, but not at the price of sacrificing traceability to their original programs. A simple example illustrates the importance of this. Consider what might happen if an instruction that interacts with the environment fails to be annotated as an event (which could easily happen with memory-mapped IO). If the instruction is relocated outside of its source section, debugging the transformed program could become a nightmare. Even worse, a fatal timing fault may accrue at runtime, since the program slicer may place it in the state thread, which can be delayed past the original deadline.

All of these considerations argue for a front-end that permits the programmer to interact with the tool during system-tuning. With our transformation engine as its foundation, a graphical interface allows a programmer to selectively apply the transformations – and also remain informed of the results. We have implemented such a tool for program slicing. In this tool, programmers are asked to pick a slicing criterion from the tool's program source screen, rather than the program slicer doing it automatically. In this way, programmers can selectively control the application of program slicing.

7.2 TimeWare/SLICE: the Prototype Implementation

As a proof of concept, we have implemented a program slicer tool we named TimeWare/SLICE. The key features of the tool are as follows:

- It computes a program slice with respect to a given slicing criterion.

- It works on source code.
- It allows users to save a computed slice, and to carry out operations between the current and the saved slices. These operations include intersection between slices, union of slices and subtracting one slice from another. These commands are used to segregate special threads from given task code.
- It provides a graphical user interface.
- It generates the transformed task code.

In this section we discuss the implementation of TimeWare/SLICE.

7.2.1 TimeWare/SLICE Tool Screens

A source program is displayed on the two tool windows: one called the *primary window* and the other the *secondary window*. Figure 7.1 demonstrates a possible layout on the tool screen of a SUN Sparc station equipped with a 17 inches display.

The primary window provides users with a work place where they can pick a slicing criterion and get the slicing result. The result is shown as a set of highlighted source lines on the window. On the other hand, the secondary window provides with a buffer space where a user can temporarily store a pre-computed slice. When a user carries out operations between two slices (one on the primary window and the other on the secondary window), the primary window works as if it were an accumulator. That is, the primary window provides the first operand and gets the result. We refer to the slice on the primary window as a *current* slice, and the one on the secondary window as a *saved* slice.

7.2.2 Commands

Figure 7.2 shows a screen dump of the primary window of TimeWare/SLICE. The highlighted source lines correspond to a computed slice. Now we describe the command buttons that are located below the text window.

Picking a Slicing Criterion. A slicing criterion consists of a source line and a variable name. By pressing the leftmost button of a mouse and then dragging the pointer on the primary window, the user can select a variable name. Then the selected string is highlighted.

Similarly, by moving the pointer and clicking the middle button of a mouse, the user can mark a source line. At that point, an arrow sign appears at the beginning of the line.

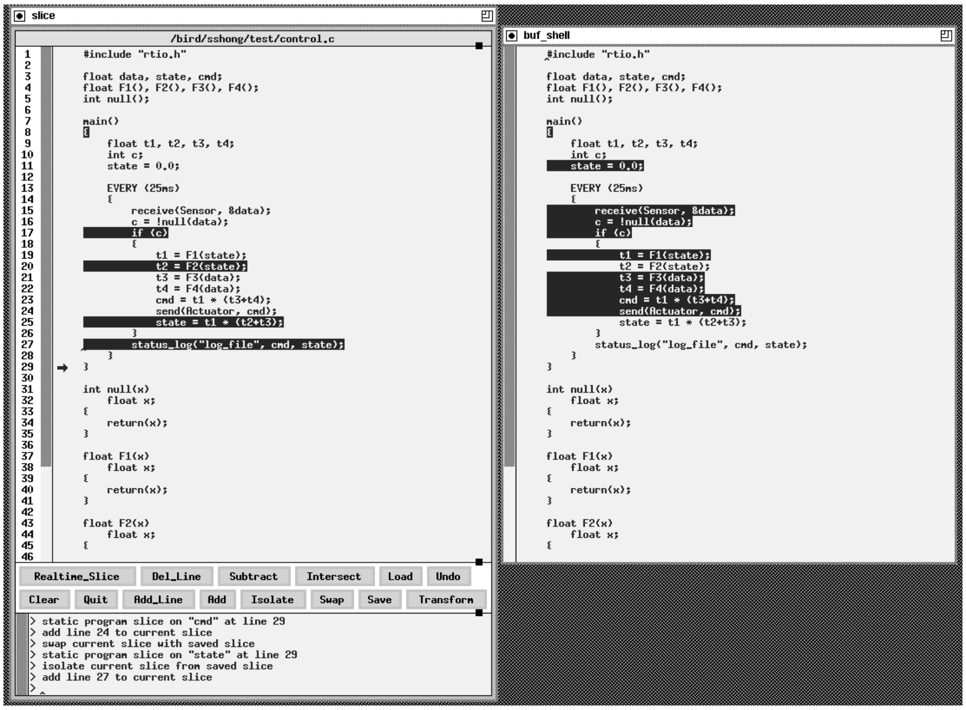

Figure 7.1: Tool Screens of TimeWare/SLICE

Realtime Slice Command. The $\boxed{\text{Realtime\_Slice}}$ function computes a static program slice of the function one of whose source lines are picked as part of the slicing criterion. Figure 7.2 shows the slice with respect to the slicing criterion ⟨ line 29, `state` ⟩.

Add and Delete Line Commands. When $\boxed{\text{Add\_Line}}$ (or $\boxed{\text{Del\_Line}}$) is pressed, TimeWare/SLICE adds (or deletes) a current line. These commands are useful to manually edit a slice. If the current slice is empty then $\boxed{\text{Del\_Line}}$ has no effect.

Subtract Command. The $\boxed{\text{Subtract}}$ function subtracts the saved slice from the current slice, and then displays the result on the primary window. As a result, all source lines common to both slices are removed from the current slice.

Add Command. The $\boxed{\text{Add}}$ function adds the saved slice to the current slice, and then displays the result on the primary window.

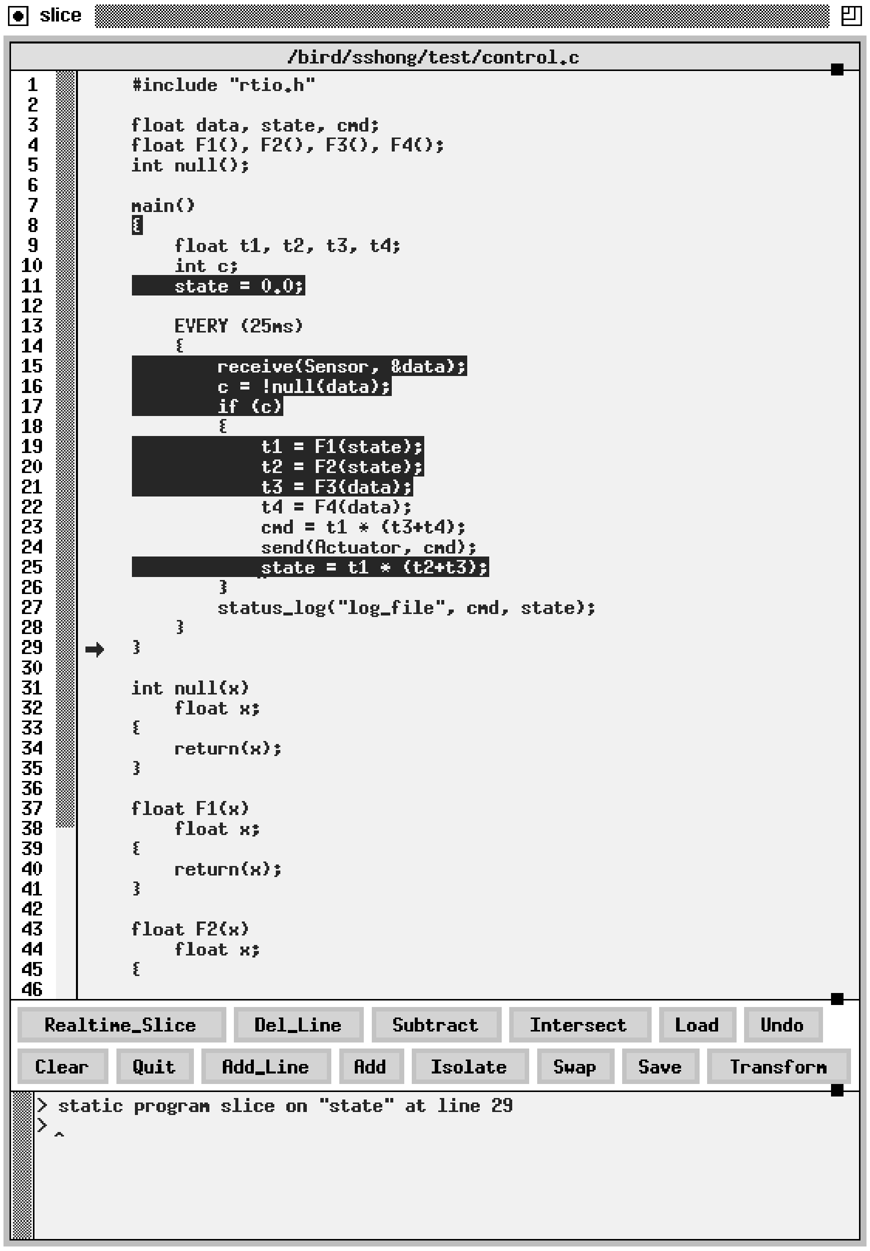

Figure 7.2: Output of TimeWare/SLICE

Intersect Command. The $\boxed{\text{Intersect}}$ function computes the intersection of the saved slice with the current slice, and then displays the result on the primary window.

Isolate Command. The $\boxed{\text{Isolate}}$ function deletes all repeated lines from the current slice, but leaves the control predicates of the current slice undeleted. The $\boxed{\text{Isolate}}$ command is used to isolate the state-update slice from the IO slice.

Transform Command. The $\boxed{\text{Transform}}$ command produces transformed code by attaching the current slice to a user-specified target line. The result is copied into a file but the original code remains intact.

Save, Load and Swap Commands. These buttons are used to save the current slice onto the secondary window, load the saved slice onto the primary window, and swap the two slices.

7.2.3 Implementation

The prototype implementation of TimeWare/SLICE is based on a dynamic program slicing tool SPYDER developed at Purdue University [1]. SPYDER is originally a program debugging tool relying on dynamic slicing, and it consists of two components: a modified version of GCC (GNU C compiler) and GDB (GNU symbolic debugger). The role of the modified GCC is to produce the program dependence graph for an input program as well as the object code. SPYDER traverses the graph to compute a static program slice.

We had to tailor the implementation of SPYDER due to the following limitations.

(1) It does not allow users to pick a general slicing criterion. Instead, it limits the criterion to a variable name.

(2) It is a program analysis tool where our implementation actually transforms the program.

(3) Its static slicer is not complete, which results in incorrect slices being produced. For example, the static data flow analyzer of the modified GCC does not detect redefinitions of a global variable within a function, but SPYDER does not take into account of such limitation. We had to retool TimeWare/SLICE to conquer this problem.

We have added several features to the TimeWare/SLICE implementation, as we described in the previous subsection. We made a strong assumption that every function call has a potential to redefine every global variable. This is due to SPYDER's limited interprocedural analysis. While our assumptions may result in too a large slice, users of TimeWare/SLICE can still modify the automatically generated slice using the editing facilities. In such a case it is always better to be safe than sorry.

The implementation of TimeWare/SLICE enjoys all the benefits of GNU-based software. For instance, it can easily be compiled and run on various hardware platforms.

7.3 Summary

In this chapter we considered two fundamental technologies that enabled our tool-based methodology. Unfortunately, these technologies still impose practical limitations and they impede the development of fully automated compiler tools. Thus it is desirable that programmers should be able to closely guide and trace tool applications. Keeping this philosophy as a principle, we developed a program slicer tool that allowed user interactions and traceability of transformations through a graphical front-end.

Chapter 8

Conclusions

Real-time computing, once the realm of small, hard-coded embedded systems, has evolved into a major computing discipline. Many practical techniques and formal theories have been developed for real-time computing, and they are now widely available for various computer applications. The goal of our work is to provide engineers with software support where we believe they need it the most: in system-tuning. We identified three problems in this area and then presented a comprehensive solution to them.

The first problem arises well before system-tuning, but it results in many of the inconsistencies that only tuning can remedy. That is, a high-level real-time program must be translated into both scheduler-oriented tasks and associated timing constraints on the tasks. The second problem is in reconciling the difference between task execution characteristics and the timing constraints. The last problem is attaining the desired real-time schedulability of the task set.

Our alternative to system-tuning is based on semi-automatic, compiler-based transformation methods as shown in Figure 8.1. Our approach consists of three ingredients: a new real-time programming language and two novel compiler transformation methods.

The TCEL Language. We designed the TCEL language, which possesses high-level timing constructs for deadlines, periodicity, etc. Unlike other languages that came before it, TCEL has a semantics based on time-constrained relationships between observable events. The semantics yields a clear interpretation of timing behavior.

Feasible Code Synthesis. Armed with the event-based semantics, we developed a program transformation called feasible code synthesis, to help synthesize feasible task code from a TCEL program. The objective is to correct intra-task feasibility errors.

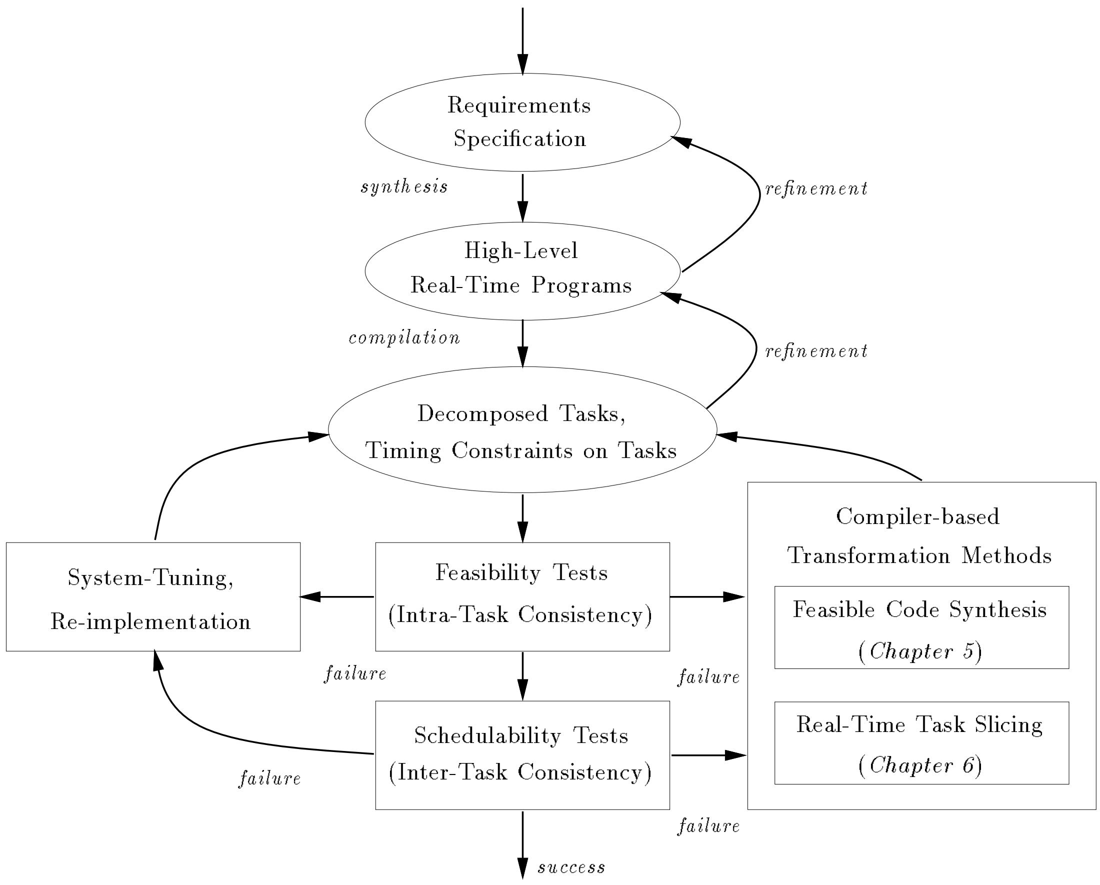

Figure 8.1: Software Development Process: Two Alternatives

Real-Time Task Slicing. After the feasibility of each task is achieved the entire task set may still be unschedulable. We developed a transformation method to address this problem, and we call it real-time task slicing. While the optimization is completely local, the improvement is realized globally, for the entire task set.

Alan Burns at the University of York independently developed a fixed-priority scheduling scheme to support the TCEL model. His analysis, in turn, led us to make a further improvement in the slicing tool; specifically, we developed a priority ordering algorithm that derives only a small subset of tasks to be sliced for schedulability.

8.1 Future Directions

We originally envisioned TCEL as a stand-alone real-time programming language. However, over time the emphasis of this work shifted from "programming languages" to "tuning." Eventually we realized that this tuning approach was applicable to many different languages where the minimal requirement is an annotation mechanism that can both distinguish observable events in a real-time program, and establish timing relationships between the events. Indeed, perhaps a graphical language may prove better suited to this purpose. Though their application domain is different from ours, several recent multimedia tools have been developed, which allow users to specify temporal relationships between several continuous media [26]. We plan to borrow some of the ideas used in these languages to graphically describe timing relationships in a real-time program.

When a real-time application is written in such a graphical language, the compiler has more freedom to interleave operations because code blocks are only constrained by the specified precedences and data dependences. We are investigating this research direction, since it accords with our thrust in a graphical user interface. Moreover, we believe a wider spectrum of compiler transformations will be applicable to the program.

There is another interesting research direction which is closely related to our tool-based approach to real-time programming. The compiler-based approach we presented in this dissertation can help developers tune applications. However, when the design is flawed, no amount of tuning can help achieve either feasibility or schedulability. Flaws are often the result of translating a high-level specification into a set of schedulable "tasks." For example, real-time programmers manually break a complex real-time design into tasks, while trying to maintain the functional correctness, so that the real-time schedulers can tractably guarantee the system's requirements. Consequently, this decomposition introduces a new class of intermediate timing constraints, which are artifacts used to realize the original high-level requirements. If our tuning tools cannot help the programmers achieve an implementation consistent with the intermediate constraints, then the whole manual decomposition may have to be repeated.

An important research direction is to automate this process with a comprehensive design methodology. Although our compiler-based approach provides a partial solution by means of a new programming language, we plan to push it into a higher level in the design hierarchy. We recently laid the foundation to achieve this goal, via an automated design methodology that works hand-in-hand with real-time design tools [17]. We call this end-to-end design.

The methodology links two more phases in the design hierarchy, i.e., a high-level component derives intermediate constraints using the low-level tuning tools. The result is a way to aid programmers in decomposing tasks and deriving the intermediate rate constraints.

We believe that this type of strategy can significantly streamline the design process, since it supports a variety of low-level resource-specific considerations early on in the life-cycle. We are currently investigating a full-scale version of this method in order to incorporate network communication, as well as the load on each processor.

Bibliography

[1] H. Agrawal, R. DeMillo, and E. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):590–616, June 1993.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley Publishing Company, 1986.

[3] A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, pages 584–594, May 1988.

[4] F. Allen, B. Rosen, and K. Zadeck. *the forthcoming Optimization in Compilers.* Addison Wesley Publishing Company, 1992.

[5] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS 164, Department of Computer Science, University of York, England, December 1991.

[6] A. Burns. Fixed priority scheduling with deadlines prior to completion. Technical Report YCS 212 (1993), Department of Computer Science, University of York, England, October 1993.

[7] J.-D. Choi, R. Cytron, and J. Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Transactions on Software Engineering*, 20(2):105–114, February 1994.

[8] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, October 1991.

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

[10] K. Ebcioglu and A. Nicolau. A global resource-constrained parallelization technique. In *International Conference on Supercomputing*, pages 154–163. ACM Press, June 1989.

[11] J. Ferrante and K. Ottenstein. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–345, July 1987.

[12] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30:478–490, July 1981.

[13] M. Garey and D. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[14] F. Gasperoni. Compilation techniques for VLIW architectures. Technical Report RC 14915(#66741), IBM T. J. Watson Research Center, September 1989.

[15] R. Gerber and S. Hong. Semantics-based compiler transformations for enhanced schedulability. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 232–242. IEEE Computer Society Press, December 1993.

[16] R. Gerber and S. Hong. Compiling real-time programs with timing constraint refinement and structural code motion. *IEEE Transactions on Software Engineering*, 1995. To Appear.

[17] R. Gerber, S. Hong, and M. Saksena. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 192–203. IEEE Computer Society Press, December 1994. Also to appear in *IEEE Transactions on Software Engineering*.

[18] P. Gopinath and R. Gupta. Applying compiler techniques to scheduling in real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 247–256. IEEE Computer Society Press, December 1990.

[19] M. Harmon, T. Baker, and D. Whalley. A retargetable technique for predicting execution time. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 68–77. IEEE Computer Society Press, December 1992.

[20] P. Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Department of Computer Science, Rice University, May 1994.

[21] S. Hong and R. Gerber. Compiling real-time programs into schedulable code. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. ACM Press, June 1993. *SIGPLAN Notices, 28(6):166-176.*

[22] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graph. *ACM Transactions on Programming Languages and Systems*, 12:26–60, January 1990.

[23] Y. Ishikawa, H. Tokuda, and C. Mercer. Object-oriented real-time language design: Constructs for timing constraints. In *Proceedings of OOPSLA-90*, pages 289–298, October 1990.

[24] F. Jahanian and A. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, September 1986.

[25] K. Kenny and K. Lin. Building flexible real-time systems using the Flex language. *IEEE Computer*, pages 70–78, May 1991.

[26] M. Kim. Hyperstories: Combining time, space and asynchrony in multimedia documents. Technical Report RC 19277 (#83726), IBM T. J. Watson Research Center, October 1993.

[27] E. Kligerman and A. Stoyenko. Real-Time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12:941–949, September 1986.

[28] J. Krause. GN&C domain modeling: Functionality requirements for fixed rate algorithms. Technical Report (DRAFT) version 0.2, Honeywell Systems and Research Center, December 1991.

[29] I. Lee, P. Brémond-Grégoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. *IEEE Proceedings*, 82(1), January 1994.

[30] I. Lee and V. Gehlot. Language constructs for real-time programming. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 57–66. IEEE Computer Society Press, 1985.

[31] J. Leung and M. Merill. A note on the preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, November 1980.

[32] S. Lim, Y. Bae, C. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim. An accurate worst case timing analysis for risc processors. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 97–108. IEEE Computer Society Press, December 1994.

[33] K. Lin and S. Natarajan. Expressing and maintaining timing constraints in FLEX. In *Proceedings of IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1988.

[34] C. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[35] C. Locke, D. Vogel, and T Mesler. Building a predictable avionics platform in ada: A case study. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 181–189. IEEE Computer Society Press, December 1991.

[36] T. Marlowe and S. Masticola. Safe optimization for hard real-time programming. In *Second International Conference on Systems Integration*, pages 438–446, June 1992.

[37] M. Merritt, F. Modungo, and M. Tuttle. Time-Constrained Automata. In *CONCUR '91*, August 1991.

[38] A. Nicolau. *Parallelism, Memory Anti-aliasing and Correctness Issues for a Trace Scheduling Compiler.* PhD thesis, Yale University, June 1984.

[39] V. Nirkhe. *Application of Partial Evaluation to Hard Real-Time Programming.* PhD thesis, Department of Computer Science, University of Maryland at College Park, May 1992.

[40] V. Nirkhe and W. Pugh. Partial evaluator for the Maruti hard real-time system. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 64–73. IEEE Computer Society Press, December 1991.

[41] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, May 1984.

[42] C. Park and A. Shaw. Experimenting with a program timing tool based on source-level timing schema. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 72–81. IEEE Computer Society Press, December 1990.

[43] W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.* ACM Press, June 1992.

[44] D. Whalley R. Arnold, F. Mueller. Bounding worst-case instruction cache performance. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 172–181. IEEE Computer Society Press, December 1994.

[45] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Software Engineering*, 39:1175–1185, September 1990.

[46] U. Shankar. A simple assertional proof system for real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 167–176. IEEE Computer Society Press, December 1992.

[47] A. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, pages 875–889, July 1989.

[48] M. Smith, M. Horowitz, and M. Lam. Efficient superscalar performance through boosting. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248–259. ACM Press, October 1992.

[49] A. Stoyenko. A schedulability analyzer for Real-Time Euclid. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 218–227. IEEE Computer Society Press, December 1987.

[50] K. Tindell. Using offset information to analyse static priority pre-emptively scheduled task sets. Technical Report YCS 182 (1992), Department of Computer Science, University of York, England, August 1992.

[51] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *The Journal of Real-Time Systems*, 6(2):133–152, March 1994.

[52] G. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[53] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.

[54] V. Wolfe, S. Davidson, and I. Lee. RTC: Language support for real-time concurrency. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 43–52. IEEE Computer Society Press, December 1991.

[55] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *The Journal of Real-Time Systems*, 5(4), October 1993.