Model-driven Timing Analysis of Embedded Software

LEI JU

(B.Eng (HONS), National University of Singapore, Singapore)

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE
2010

Acknowledgements

This thesis would not have been possible without the help, support and patience of my supervisors, Prof. Abhik Roychoudhury and Prof. Samarjit Chakraborty. Prof. Abhik Roychoudhury has been my supervisor for over six years, since I was doing my Final Year Project as an undergraduate student in School of Computing, National University of Singapore. During the five years of my graduate study, I have received immense support both in academics and life from Prof. Abhik Roychoudhury and Prof. Samarjit Chakraborty. Their profound knowledge and professional guidance have been of great value to me in my past research work presented in this thesis, and future career in life.

I wish to express my warm and sincere thanks to Prof. Wong Weng Fai and Prof. Chin Wei Ngan as my thesis committee members. They have given me many insightful comments and advices. I have been benefitted a lot from Prof. Tulika Mitra through research collaborations, as well as her distinguished courses on embedded systems. I would also like to thank Prof. Björn Lisper for taking time out of his schedule and agreeing to be my thesis external examiner.

It is an honor for me to join IBM Research - Tokyo as a student intern during my graduate study. I have gained valuable experiences by being exposed to industry-

oriented research work, under the supervision of Dr. Arquimedes Canedo, Dr. Takeo Yoshizawa, and Dr. Hideaki Komatsu.

I dedicate this thesis to my parents that have brought me so much love and encouragement throughout my life. They have been always supportive of me in pursuing my dreams and help me become the person I am today.

I would also like to express my special thanks to Huynh Bach Khoa and Liang Yun, who are great friends in daily life and excellent partners in research collaborations. Besides, I really appreciate the support and friendship from my fiends inside and outside the university, including my lab mates Wang Tao, Guo Liang, Ankit Goel, Vivy Suhendra, Qi Dawei, and Wang Chundong. I thank my basketball team members Prof. Ooi Beng Chin, Yang Fei, Bao Zhifeng, Wu Sai, Zhang Zhenjie, Cao Yu, Zhang Dongxiang, just to name a few. Doing sports with them is of huge fun and has made me refreshed after the tiredness and stress of work.

The work presented in this thesis was partially supported by National University of Singapore research projects R252-000-286-112 and R252-000-321-112. They are gratefully acknowledged.

Contents

Abstract

In recent years, model-based design has become an industrial standard to address problems associated with designing complex embedded software. For hard real-time system domains including avionics and automobiles, static timing analysis is of paramount importance. To reinforce the advantages of model-based design approach, timing analysis must be seamlessly coupled to provide designers with temporal behavior of the system at early design stages. In this thesis, we study various models (applicable at different design levels) and corresponding timing analysis techniques. We show that to achieve correct and accurate timing estimates in model-driven embedded software design, both model-level and micro-architectural information need to be considered in the timing analysis.

Code-level WCET analysis determines worst-case timing behavior of a program on a micro-architecture for all possible inputs. In a model-based design framework, executable code is automatically generated from a high-level model. We show that accurate code-level timing estimates can be achieved by taking into account the high-level information in the timing analysis. We discuss our model-driven WCET analysis in the context of Esterel, a representative synchronous programming model. Our proposed

timing analysis utilizes model-level information to help determining program path and context in the WCET analysis of generated C code from Esterel specification. In addition to strengthening existing WCET analysis approaches for sequential programs with our model-driven techniques, we also propose a framework for timing analysis of multiprocessor execution of Esterel specifications. Experimental results show that our analysis substantially reduces WCET over-estimation.

In system-level schedulability analysis, WCET of each individual task is provided as input parameters, which captures the worst-case *intra-task* timing behavior for the task. Traditional task graph-based system models and their schedulability analysis essentially concern with independent tasks and single-processor execution. We propose schedulability analysis for standard Message Sequence Chart (MSC) based system models, which are widely used for describing interaction scenarios between the components of a distributed system. We also capture the timing effects of the shared bus for inter-task communication in our proposed analysis. We illustrate the details of our analysis using a setup from the automotive electronics domain, which consist of two real-life application programs (that are naturally modeled using MSCs) running on a platform consisting of multiple electronic control units (ECUs) connected via a FlexRay bus.

Related Publications

1. L. Ju. Model-driven Timing Analysis of Embedded Software. *13th ACM SIGDA PhD Forum at the Design Automation Conference (*DAC*)*, 2010.

2. L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Timing Analysis of Esterel Programs on General-purpose Multiprocessors. *ACM Design Automation Conference (**DAC**)*, 2010.

3. L. Ju, B. K. Huynh, S. Chakraborty, and A. Roychoudhury. Context-Sensitive Timing Analysis of Esterel Programs. *ACM Design Automation Conference (**DAC**)*, 2009.

4. L. Ju, B. K. Huynh, A. Roychoudhury and S. Chakraborty. A Systematic Classification and Detection of Infeasible Paths for Accurate WCET Analysis of Esterel Programs. *Singaporean-French IPAL Symposium (**SinFra**)*, 2009

5. L. Ju, B. K. Huynh, A. Roychoudhury and S. Chakraborty. Performance Debugging of Esterel Specifications. *ACM Intl. Conference on Hardware/Software Codesign and System Synthesis (**CODES+ISSS***), 2008.

6. L. Ju, A. Roychoudhury and S. Chakraborty. Schedulability Analysis of MSC-based System Models. *IEEE Real-Time and Embedded Technology and Applications Symposium (**RTAS**)*, 2008.

7. L. Ju, S. Chakraborty and A. Roychoudhury. Accounting for Cache-related Preemption Delay in Dynamic Priority Schedulability Analysis. *Design Automation and Test in Europe (**DATE**)*, 2007.

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, model-based design has become an industrial standard to address problems associated with designing complex embedded software. It provides an efficient and cost-effective way to support various stages in the development cycle, including requirement engineering, design reuse, model-based testing, simulation and verification. Mature commercial tools have been built and are successfully adopted in different application domains, including the Unified Modeling Language (UML) [49], MATLAB Simulink [103] and SCADE Suite [99]. In the model-based design flows, the entire system description is usually developed as high-level models and final hardware/software deployment can be automatically generated from these models (also referred to as model-driven engineering [45]).

Lots of methodologies and tool support have been built for model-driven testing and verification (e.g., [111, 30, 105, 32, 73]). However, a significant portion of the works focus on functionality analysis (such as verification of safety and liveness properties).

On the other hand, very limited effort has been invested to support quantitative/timing analysis in model-based design. Existing model-level software performance predictive analyses (e.g., [6]) are based on high-level performance models (e.g., the UML Profile for Scheduling, Performance, and Time [48], and timed automata [2]), where timing information are given and annotated with the model elements. However, such analyses are usually ignorant of the underlying architecture platforms where generated software implementations are executed (which may lead to loose or even unsafe analysis results). Furthermore, a systematic design process for automatic calculation of platform-specific timing information of model elements is missing.

Timing analysis plays an very important role in real-time and embedded system design. Simulation based timing analysis techniques (e.g., [79]) are expensive, and the observed execution time may be an under-estimation of the *real* worst case scenario. In hard real-time domains (e.g., avionics, automobiles and medical embedded devices), guaranteed upper bounds of the worst-case timing behaviors must be provided via static timing analysis to ensure the correctness and safety of a system. Two well-studied static software timing analysis approaches in embedded system design are:

- Code-level worst case execution time (WCET) analysis. WCET analysis computes the maximum execution time of a program on a micro-architecture for all possible inputs. Accuracy of the estimated WCET depends on both program path information and timing effect of the micro-architecture. Thus, a typical WCET analysis involves code level flow analysis (e.g., [66, 53]) and micro-architectural modeling (e.g., pipelines [36, 70], caches [43, 26], and branch predictors [29]).

- System-level schedulability analysis. A schedulability analysis (or feasibility analysis) decides that given a set of tasks and a certain scheduling policy, whether all constraints(usually the deadlines) associated with each task could be satisfied. Various schedulability analysis techniques have been proposed for different task models on single-processor (e.g., [74, 9, 82, 8, 7]) or multiprocessor/distributed (e.g., [110, 113, 87, 18]) execution.

Motivation of this dissertation: The motivation of this dissertation is to provide seamless timing analysis support for modern model-based design framework of real-time embedded systems. Traditional schedulability analysis techniques are applicable to system models that are essentially based on the concept of task graphs (e.g., [74, 7]). However, such task graph-based models only provide *local* or *processor-centric* views of a system, and are not very suitable for specifying the interactions between the multiple entities of system. Comparing to high-level behavioral modeling languages used in model-based design frameworks(e.g. message sequence charts MSC [58]), such task graph-based specifications are too abstract and lack of expressive power to model all possible behaviors (e.g., data communication, conditional execution) for complex system functionalities.

On the other hand, one significant challenge for static WCET analysis is to reduce the overestimation between estimated WCET and real WCET, due to dynamic program behavior and complexity of underlying architecture. State-of-the-art WCET analysis techniques (e.g, [112]) try to achieve accurate timing estimates, by tightly coupling

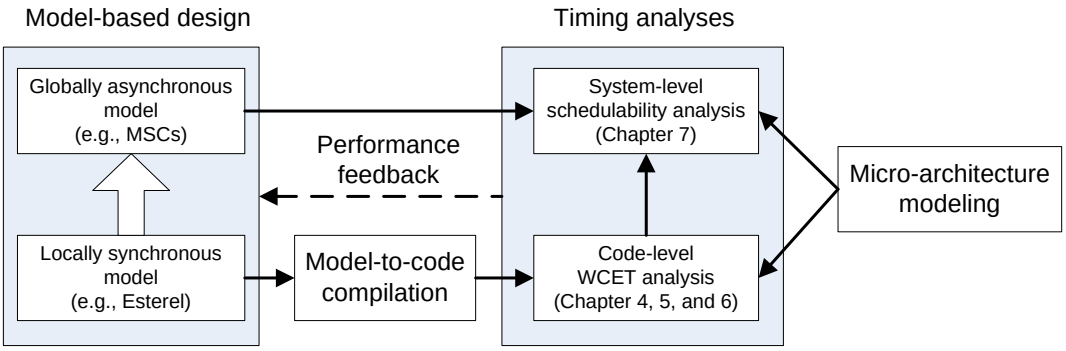


Figure 1.1: Overview of our model-driven timing analysis framework

program path analysis (at source/binary code level) and micro-architectural modeling. However, for model-based design framework where source programs are automatically generated from high level models, blindly analyzing the generated code without taking model-level information into consideration may lead to additional overestimation.

Contributions of this dissertation: To achieve our goal, we propose methodologies of both system-level schedulability analysis and low-level WCET analysis for model-based design frameworks. Figure 1.1 presents an overview of our proposed timing analysis for model-driven embedded system design. In this thesis, we consider a fairly general model hierarchy called the Globally-Asynchronous Locally-Synchronous (GALS) model [25, 83]. Each local task is designed in a synchronous model (e.g., the synchronous language Esterel [21]), where all computation and communication for a set of given inputs and outputs are assumed to react instantaneously. Software implementations (e.g., C programs) can be automatically generated from the synchronous model. The global system is represented with an asynchronous model (e.g., MSCs), which describes relations between individual local tasks of various applications in the system,

including the control/data dependencies and communications. The global system is asynchronous such that (i) reaction time of each local task, as viewed by other tasks is finite and non-zero; and (ii) communication time between local tasks are finite and non-zero.

For timing analysis of the above-mentioned setting, code-level WCET analysis can be performed on code generated from the locally synchronous model of each task. The system-level schedulability analysis determines the satisfaction of timing constraints (e.g., deadlines) annotated on the globally asynchronous model, given the estimated WCET values of individual tasks, as well as other properties including task periods, task to processing element (PE) mapping, and the architecture configuration. The main contributions of this dissertation are summarized below.

- We propose an accurate WCET analysis framework for C programs generated from Esterel specifications, which have been widely adopted for designing reactive kernels in safety-critical domains such as avionics [11]. Automatically generated code from high-level control-intensive models like Esterel usually contains massive number of infeasible paths, compared to human-written programs. In our WCET analysis, we can efficiently and effectively identify and remove infeasible paths in the generated code by exploiting the semantics and compilation information of the source Esterel specification. Thus, tighter WCET estimate of a single Esterel tick execution can be obtained [62].

- We show that bi-direction traceability can be automatically built between high-level model and low-level timing analysis [62]. By applying the maintained

model-to-code mapping on the calculated WCET path, we are able to identify parts of the model specification which might pose as timing/performance bottlenecks with respect to the underlying platform. This not only allows a designer to optimize or simplify Esterel specifications, but also choose/configure suitable implementation platforms.

- In [61], we further extend our timing analysis for Esterel specification to capture context information between tick executions. We show that program control flow as well as architecture contexts can be used to rule out certain execution paths and architecture states in the code to be executed within a tick. Our experimental results with realistic case studies show 40% tighter timing estimates when program control flow and inter-tick cache context information is taken into account.

- Following this line of work, we propose a scheme for generating efficient code from Esterel specifications for a multiprocessor execution. Furthermore, we achieve tight timing estimation on the generated multiprocessor C code, by considering inter-processor infeasible program flow and modeling the timing effect of the shared bus [63].

- We propose a general schedulability analysis for distributed system modeled in a globally asynchronous message sequence chart (MSC) based specification [64]. MSC graphs (MSGs) (or high-level message sequence charts HMSCs) can be very convenient for describing interactions among a number of agents, and are therefore a natural choice for modeling and specifying distributed real-time and

embedded systems. Given a system description in MSGs, along with the scheduling/arbitration policies at the different resources (e.g., PEs and shared buses), our analysis can be used to compute upper bounds on the end-to-end delays associated with different event (and/or message) sequences. We illustrate the details of our analysis using a setup from the automotive electronics domain, where two real-life applications running on multiple electronic control units (ECUs) connected via a FlexRay bus. We show that compared to existing timing analysis techniques for distributed real-time systems, our proposed analysis gives tighter results, which immediately translate to better system design and improved resource dimensioning.

Organization of the Chapters: The rest of the thesis is organized as follows. The next two chapters discuss background and related work on system design models and timing analysis. In order to systematically obtain WCET estimation for individual tasks in a system specification, we propose a model-driven WCET analysis for tasks designed with Esterel specification in Chapter 4, 5, and 6. In particular, Chapter 4 considers the WCET estimation for a single Esterel clock tick execution, with automatical and light-weight infeasible path detection and elimination. We also discuss how to maintain and utilize a bi-directional traceability between Esterel model specification and the generated C programs for performance feedback and further WCET refinement. Chapter 5 shows how to incorporate program control flow and architecture contexts into timing analysis of task computation that spans multiple consecutive clock ticks. Chapter 5

extends the our timing analysis techniques to multiprocessor platforms. In Chapter 7, we present our proposed system-level schedulability analysis for MSC-based globally asynchronous models. Finally, Chapter 8 presents the concluding remarks along with extensions and directions for future research.

Chapter 2

Background

2.1 Design Models

Synchronous models [10] provide a clear formalism for programming reactive systems, which exhibit high degree of concurrency but call for deterministic and predictable execution. Commonly used synchronous models in embedded system design include UML StateCharts [57], MATLAB Simulink/Stateflow [103], and synchronous languages (Esterel [21], Lustre [54] and Signal [12]). Use of synchronous models simplifies the task of programming and makes such specifications amenable to formal verification/certification. Generating implementations directly from synchronous language specifications is widely practiced in safety-critical domains such as avionics where certification of the generated implementation is essential.

On the other hand, large-scale distributed computer systems are usually implemented by asynchronously composing several synchronous components, where each

component has its own clock. In such asynchronous model, reaction time of each local task and communication time between tasks are viewed by other tasks as finite and non-zero. It relaxes the behavior of the system, and allows the designer to refine one local task at a time.

In this thesis, we consider a fairly general system description with the Globally Asynchronous Locally Synchronous (GALS) model [25]. In particular, we adopt the synchronous language Esterel and asynchronous message sequence charts (MSCs) to illustrate our model-driven timing analysis techniques.

2.1.1 The Synchronous Language Esterel

Synchronous languages like Esterel have been widely adopted for designing reactive systems in safety-critical domains such as avionics and automobiles (e.g., [28]). Esterel is an imperative concurrent language. Specifications written in Esterel are based on the underlying "synchrony hypothesis", where all computation and communication, unless explicitly paused (using a `pause` statement), happen instantaneously. A run of a program typically consists of steps or *reactions* in response to *ticks* of a global clock. With each clock tick, a reaction computes the values of output *signals* and a new state from the input *signals* and the current state of the program. Such a reaction completes (in zero time) if it does not contain any `pause`, or else it delays the instructions following the `pause` until the next clock tick.

For example, the program "`emit A; emit B; pause; emit C; pause; emit D`" *emits* the signals `A` and `B` at the first tick, `C` at the second tick, and `D` at the

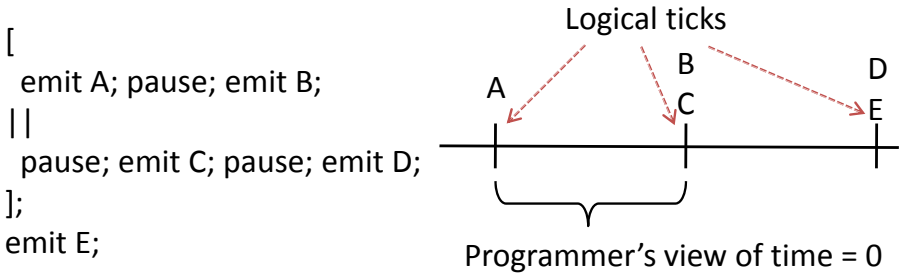


Figure 2.1: An example Esterel program

third tick. If `p` and `q` are Esterel statements, then `p || q` is the parallel composition where `p` and `q` are executed concurrently with signals between `p` and `q` being transmitted instantaneously. Thus, the Esterel program shown in Figure 2.1 will emit signal `A` at the first tick, `B` and `C` at the second tick, followed by `D` and `E` in the third tick. Further details of the syntax and semantics of Esterel may be found in [21] (or from the references in [11]).

Compiling Esterel. Esterel programs can be compiled into C programs to be simulated/executed on general processor architectures. In principle, the generated C code should preserve the semantics of original Esterel program by

- implementing a *tick function*, such that one complete execution of the function (between its entry and exit) represents Esterel computation and communication required to be instantaneously executed within one clock tick. The tick function is loop-free, since Esterel allows no loops within a clock tick.

- encoding the automata of tick transitions within the tick function, which preserves the context information of clock tick, and determines the path to be executed in the tick function.

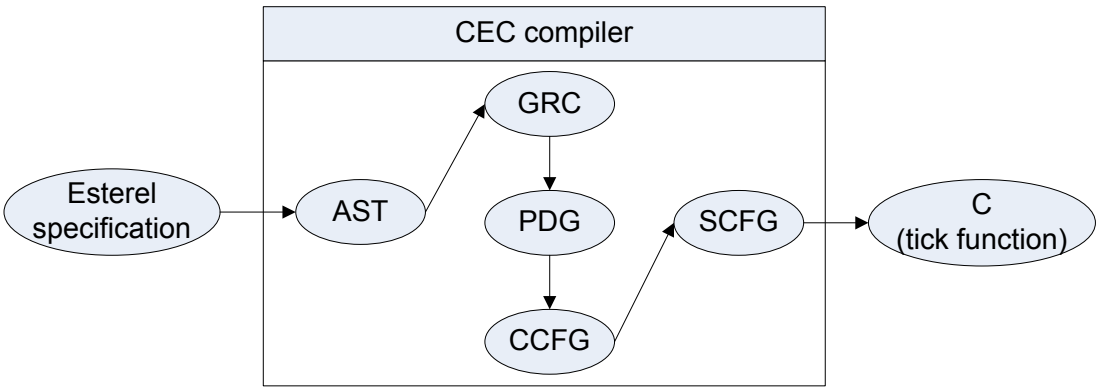


Figure 2.2: CEC compiler and the intermediate representations

- sequentializing the concurrent execution within a tick, based on the control dependencies (e.g., clock tick boundary, preemption) and communication dependencies (between set and test of signals) defined in the Esterel program.

Various techniques exist for compiling Esterel into sequential or distributed C programs (e.g., [90, 47]). Based on the intermediate representation used, they can be categorized into automata-based, netlist-based, and control flow graph-based approaches.

An automata based compiler (e.g., Berrys V3 [15]) exhaustively searches entire state space of the program and builds a product automata that captures all computation and communication in each clock tick. A separate branch is generated for each state in the automaton (representing a possible clock tick). The generated code is very fast to run, with very small overhead to determine the state to be executed. However, the size of generated code grows exponentially with the number of concurrent threads in the specification.

Netlist based approaches (e.g., Berry's V4 and V5 [90]) translate each Esterel statement into a netlist of boolean logic gates. No statement duplication is required in the

generated code, which leads to much more compact code compared to the automata based compilation. However, the main drawback is the significant increase in execution time. This is due to all code in the source specification will get executed in each clock tick, even though some of them are not required to run [90].

In this thesis, we will focus our discussion on the control flow graph-based Esterel compilation, which normally produces fast and small C code. In particular, we have integrated our work into the control flow graph-based code generation of the Columbia Esterel Compiler(CEC) [34]. Figure 2.2 presents an overview of the CEC compiler and the intermediate representations used during Esterel-to-C compilation. CEC first parses an Esterel specification to build an abstract syntax tree (AST), which is then used to generate a variant of the so-called Graph Code (GRC) [90] through a syntax directed translation. GRC represents a concurrent structure of the desired cycle function and uses a selection tree to encode the transition between cycles. It is an elegant way to represent the Esterel program, which allows optimizations to be performed prior to C code generation. The GRC is then transformed into a sequential control flow graph (SCFG), via a set of intermediate representations like a program dependence graph (PDG), and a concurrent control flow graph (CCFG). In CEC, these intermediate steps ensure that the concurrent control flow in GRC is sequentialized with the minimum number of context switches, while obeying the control/communication dependencies in the original Esterel program. Finally, sequential C code can be directly generated from the SCFG.

2.1.2 Message Sequence Charts

Message Sequence Charts (MSCs) or Sequence Diagrams are widely used by requirements engineers in the early stages of reactive system design [60, 94, 4]. MSCs can be very convenient for describing asynchronous interactions between a number of locally synchronous agents, e.g., a bus protocol between a bus controller and a number of processing elements trying to negotiate access to the bus. MSCs are therefore a natural choice for modeling and specifying distributed real-time and embedded systems.

Definition 1 (Message Sequence Chart) *An MSC is a labeled poset of the form* $Ch = (L, \bigcup_{l \in L} E_l, \preceq, \lambda)$*, where*

- L *is the set of processes (also called lifelines) appearing in the chart as vertical lines.*
- E_l *is the set of events that the lifeline* l *takes part in during the execution of* Ch*.*
- $\preceq$ *is the partial ordering relation over the occurrences of the events in* $\bigcup_{l \in L} E_l$*. The relation* $\preceq$ *or* $\preceq^{Ch}$ *(we put* Ch *as the superscript when necessary to highlight that the partial order belongs to chart* Ch*) is defined as follows.*
  - $\preceq_l^{Ch}$ *is the linear ordering of events in* E_l*, which are ordered top-down along the lifeline* l*.* $\preceq_l^{Ch}$ *is restricted to events on the same lifeline* l*, where* $\preceq_L^{Ch}$ *is the collection of* $\preceq_l^{Ch}$ *for all lifeline* $l \in L$*.*
  - $\preceq_{sm}^{Ch}$ *is an ordering on message send/receive events in* $\bigcup_{l \in L} E_l$*. If* e_s *is a send of message* m *by process* p *to process* q*, and the corresponding re-*

ceive event is e_r (the receipt of the same message by process q), we have $e_s \preceq_{sm}^{Ch} e_r$. In rest of this thesis, we also refer the ordering on message send/receive events as communication dependency *between the sender and receiver events.*

– *$\preceq^{Ch}$ is the transitive closure of $\preceq_L^{Ch} = \bigcup_{l \in L} \preceq_l$ and $\preceq_{sm}$, i.e.*

$$\preceq^{Ch} = (\preceq_L^{Ch} \cup \preceq_{sm}^{Ch})^\star$$

- *λ is the labeling function, with a suitable range of labels, which describes (a) the messages exchanged by the lifelines and (b) the internal computational steps during the execution of the chart Ch.*

For example in the MSC $msc1$ in Figure 2.3, we have $E_{1,1} \preceq_{P1}^{msc1} E_s^{m1} \preceq_{P1}^{msc1} E_{1,3}$ on the lifeline $P1$. For sending and receiving message $m1$ between $P1$ and $P2$, we have the ordering $E_s^{m1} \preceq_{sm}^{msc1} E_r^{m1}$. The transitive closure $(\preceq_{P1}^{msc1} \cup \preceq_{P2}^{msc1} \cup \preceq_{sm}^{msc1})^\star$ defines the following ordering

$$E_{1,1} \preceq_{P1}^{msc1} E_s^{m1} \preceq_{sm}^{msc1} E_r^{m1} \preceq_{P2}^{msc1} E_{1,2}$$

However, no ordering is imposed between $E_{1,2}$ and $E_{1,3}$ in $msc1$. Thus, an MSC defines a partial ordering relation over the events in the chart.

The preceding definition of MSC is an abstract one, and does not clarify the events appearing in an MSC. The complete MSC language [60] includes several types of events: message sends and receives, local actions, lost and found messages, instance creation and termination etc. However, for simplicity of exposition, we assume that

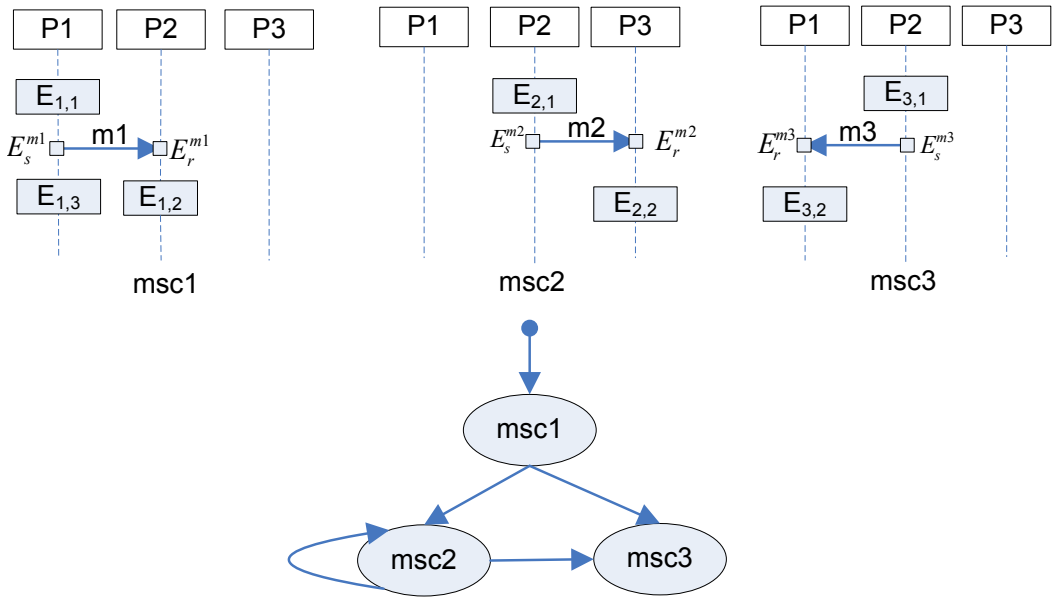


Figure 2.3: An example MSG

the events inside an MSC is of one of the following forms — *sends, receives and local events*. A local event can denote any terminating computation within a process, i.e., a terminating sequential program.

Each MSC in a system specification only denotes a *scenario* and captures the partial ordering between various computation and communication tasks/events constituting this scenario. Multiple such MSCs can be combined hierarchically to form high-level MSCs (HMSCs) [60] , which involves choice, concatenation and iteration operations over a finite set of basic MSCs. HMSCs based specification is capable of capturing all possible system behaviors. In this thesis, we consider flattened HMSCs, which are known as message sequence graphs (MSGs) [3, 58], which describe the control flow (conditional execution) *between* MSCs.

Definition 2 (Message Sequence Graph) *An MSG can be defined as a directed graph* $MSG = (N, E, \circ)$*, where*

- $N = \{MSC\} \cup \{\nabla\} \cup \{\triangle\}$ *is the set of nodes in the MSG, where each node is either a basic MSC, or a special node* ∇ *(*$\triangle$*) which denotes the unique initial (final) node respectively.*

- E *is the set of edges in the MSG, which represent the natural operation of chart concatenation between two nodes* $N_1 \rightarrow N_2$*. Two outgoing edges from a single node represent non-deterministic choice, so that exactly one of the two successor charts will be executed in an execution.*

- $\circ$ *denotes the concatenation method between two nodes. We consider the so-called* synchronous concatenation *(not to be confused with synchronous models), where for a concatenation of two charts* $Ch \circ Ch'$ *— all events in* Ch' *start only after chart* Ch *is finished.*

Example of a simple MSG is shown in Figure 2.3. In the following we consider acyclic MSGs where there are no loops between initial state (∇) to the final state ($\triangle$). An execution trace is defined to be a path from the initial state (∇) to the final state ($\triangle$) in the MSG and concatenates the sequence of MSCs encountered on the way. Of course, there is always an outer loop from final state ($\triangle$) to initial state (∇) denoting periodic behavior repeated forever. Our analysis can be extended to allow arbitrary loops in between the initial state (∇) to the final state ($\triangle$), provided these (inner) loops are bounded.

2.2 Timing Analysis

Reliable timing analysis is of significant importance for safety-critical real-time system design, where the correctness of system depends on satisfaction of both functional and timing properties. To formally verify timing constraints, extensive studies have been proposed on static timing analysis methodologies. In this section, we provide an overview of two well-known categorizations of timing analysis approaches.

2.2.1 WCET Analysis

Static worst-case execution time (WCET) analysis computes the maximum execution time of a program on a micro-architecture for all possible inputs. WCET analysis of a program involves finding the "longest" execution trace in the program's control flow graph (CFG). Recall that the nodes of a CFG are the basic blocks (maximal code fragments which are executed without control transfer), and the edges denote control transfer between basic blocks. Thus, a *path* in a control flow graph is simply a sequence of basic blocks, and an *execution trace* is a path executed for some program input. WCET analysis tries to find the maximum time the program takes to execute for any input.

Finding the weighted longest execution trace in a program can be done by running all possible inputs. However, this is not practical since (a) the number of inputs may be large, and (b) the program execution time for the same input may be different on different processors. WCET analysis methods typically solve this problem by developing a *static analysis* framework which takes as inputs (i) the program P being analyzed and (ii) a processor platform description $Proc$, and produces as output an *overestimate* of

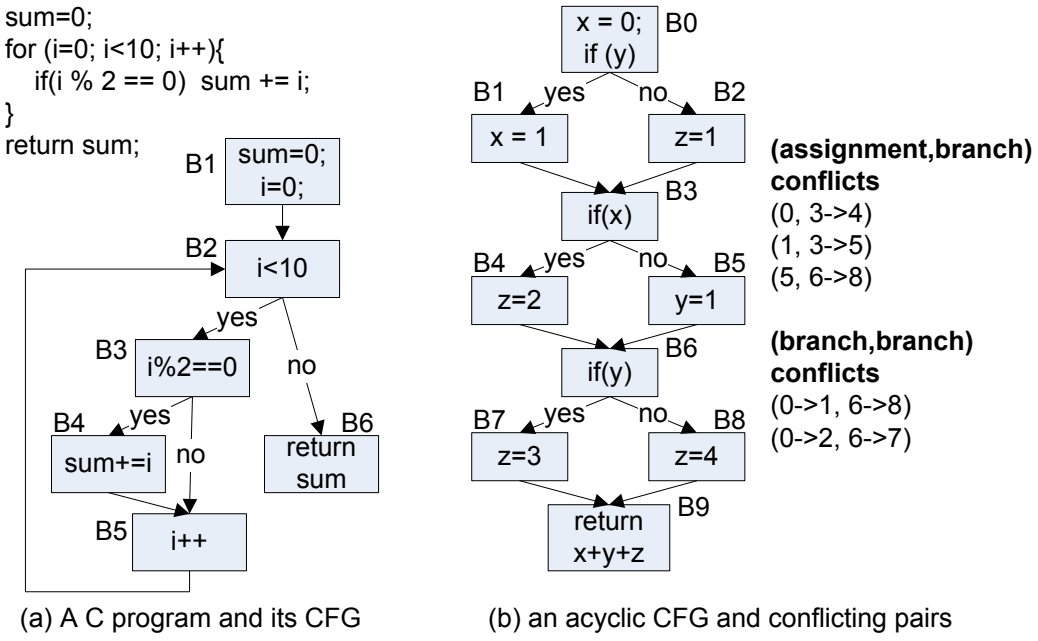


Figure 2.4: Example C programs and control flow graphs.

the WCET of program P on processor $Proc$.

Static analysis based WCET estimation proceeds by finding the longest path in the program's control flow graph, satisfying certain loop bounds (*e.g.*, in the example of Figure 2.4(a) the loop bound for the only loop is 10). The execution time estimate of each basic block is found by micro-architectural modeling where timing models of the processor micro-architecture (*e.g.*, pipeline, cache, branch prediction) are developed to find the WCET of a sequence of instructions. Note that the WCET estimate of the instruction sequence corresponding to a basic block B is an upper bound on the execution time of B under all possible execution contexts.

With the knowledge of WCET of the basic blocks, finding the WCET of the whole program is reduced to an optimization problem. Here, we maximize the program execution time without enumerating the execution traces. This is done by expressing linear

constraints on the execution counts of any node/edge of the control flow graph. We then maximize an objective function representing the program execution time subject to these linear constraints. Since the execution counts of control flow graph nodes/edges are integers, we can employ integer linear programming (ILP) technology. Formally, let $\mathcal{B}$ be the set of basic blocks of a program. The program's WCET is given as:

$$\text{maximize} \sum_{B \in \mathcal{B}} N_B * c_B$$

where N_B is an ILP variable denoting the execution count of basic block B and c_B is a constant denoting the WCET estimate of basic block B. The linear constraints on N_B are developed from the flow equations based on the control flow graph. Thus for basic block B,

$$\sum_{B' \to B} E_{B' \to B} = N_B = \sum_{B \to B''} E_{B \to B''}$$

where $E_{B' \to B}$ ($E_{B \to B''}$) is an ILP variable denoting the number of times control flows through the CFG edge $B' \to B$ ($B \to B''$). Additional linear constraints capture the loop-bounds (*e.g.*, in Figure 2.4(a) we need to add the constraint $E_{5 \to 2} \leq 10$).

Infeasible path detection. The core WCET estimation method outlined in the preceding may not be accurate. The cause of imprecision comes from the fact that many paths in the control flow graph might be *infeasible*, that is not appearing in the execution trace for any input. For example in the acyclic CFG shown in Figure 2.4(b), the execution path ($B0 \to B2 \to B3 \to B4$) cannot be taken for any program input, due to conflict between the assignment $x = 0$ (in $B0$) and the conditional branch $B3 \to B4$ (which can be taken only if $x \neq 0$). It is clear that undue WCET overestimation is introduced if an infeasible path is considered to be the longest path in WCET analysis.

Many techniques have been proposed to detect and eliminate infeasible paths at source/assembly code level for WCET analysis (e.g, [78, 37, 107, 53]). In this thesis, we adopt a light-weight infeasible path detection technique based on the notion of *conflicting pairs* [107] — pairs of (assignment, branch) or (branch, branch) statements which may not appear together in an execution trace. Simply put, an assignment a on a variable x conflicts with a branch edge e (a branch edge refers to a branch condition being evaluated to either true or false) testing the same variable x if and only if (i) the test on x in e never succeeds with the value assigned in a, and (ii) there exists at least one path in the control flow graph between a and e which does not modify variable x. Similarly, a branch edge $e1$ testing a variable x conflicts with another branch edge $e2$ testing the same variable x if and only if (i) the conditions on x in $e1$ and $e2$ can never succeed together, and (ii) there exists at least one path in the control flow graph between $e1$ and $e2$ which does not modify variable x. Note that infeasible paths spanning across loop iterations are not captured by the definition of conflicting pair. Thus, [107] considers the control flow graph (CFG) to be a directed acyclic graph (DAG), representing the body of a loop. However, as we have discussed in Section 2.1.1, code generated from Esterel specification (the tick function) contains no loop within execution of a single clock tick. Thus, we do not detect infeasible paths spanning across loop iterations.

The notion of conflicting pair is extensively used in our model-driven timing analysis for infeasible path detection of a synchronous model specification. To help readers have a better understanding the concept, we borrow the formal definition of conflicting pairs from [107].

Definition 3 (Effect constraint) *The effect constraint of an assignment* $var := expression$ *is* $var == expression$*. The effect constraint of a branch-edge* e *in the CFG for a branch condition* c *is* c *(*$\neg c$*) if* e *denotes that the branch is taken (not taken).*

Definition 4 (Conflicting pair) *A branch-edge (or assignment)* x *has (branch, branch) (or (assignment, branch)) conflict with a subsequent*[1] *branch-edge* e *if and only if*

- *Conjunction of the effect constraints of* x *and* e *is unsatisfiable, and*
- *There exists at least one path from* x *to* e *in the CFG that does not modify the variables appearing in their effect constraints.*

In Figure 2.4(b), we list the (assignment, branch) and (branch, branch) conflicting pairs in the example acyclic CFG. Conflicting pairs capture only pairwise conflicts, which cannot detect (and exploit) arbitrary infeasible path information. However, we will show that conflicting pair based infeasible path detection technique is efficient and effective for analyzing compiler generated code from high-level control-intensive models like Esterel.

2.2.2 Schedulability Analysis

Schedulability analysis decides for a given set of tasks under certain scheduling policy, whether all deadline requirement associated with each task can be satisfied. In order to perform schedulability test for a set of tasks, one has to first characterize their (timing)

[1] *Subsequent* in the sense of the topological order of the control flow DAG.

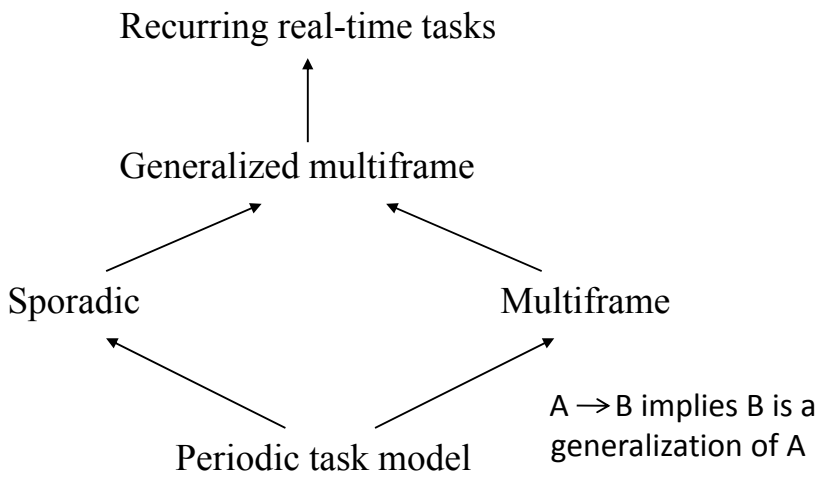


Figure 2.5: Relationship between the various task graph based models [7].

behaviors into certain task models. The well-known periodic task model by Liu and Layland [74] makes following assumptions on each task in the task set.

1. Every task τ_i is periodic, i.e. activated within a constant interval T_i
2. Deadline of each instance of task τ_i is the same and equal to its period T_i
3. Each instance of task τ_i has the same worst-case execution time C_i
4. All tasks are independent, i.e. requests for a certain task do not depend on the initiation or the completion of requests for other tasks.

Based on periodic task model, many new task graph based models and corresponding schedulability analysis techniques have been proposed in recent works. In these models, one or more assumptions made in the periodic task model are relaxed, so that more complex system behaviors can be modeled. For example, the sporadic task model [9] allows minimum separation time to be specified between two consecutive task releases, instead of the restricted constant period required in periodic task model. In

multi-frame and generalized multi-frame task model [82, 8], execution time and deadline can be varied between different task instances (a.k.a. frames). Finally, the recurring real-time task model [7] allows conditional execution between task instances. Figure 2.5 adopted from [7] summarizes the relationships between the various task models.

When a set of tasks are running on shared resources, we need a scheduling policy to decide which task should be allowed to proceed when several of them are ready. Scheduling involves the allocation of resources to task in such a way that certain performance constraints are met. Different kinds of scheduling policies have been proposed [91], including the well-known Rate Monotonic scheduling (RMS) and Earliest Deadline First (EDF) scheduling.

There are two standard categorizations of approaches for schedulability analysis of of real-time systems — worst case response time (WCRT) analysis-based techniques (WCRT) [22, 55, 68], and the processor demand criteria-based analysis [8, 23].

WCRT analysis can be performed to test feasibility of a task set under a static priority policy. The response time of a task is defined as the time interval between it releases and finishes execution. The response time of task τ_i can be calculated by the following recursive equation

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \lceil \frac{w_i^n}{T_j} \rceil \times C_j$$

where B_i is the blocking time of τ_i by lower priority tasks(due to resource contention), and $hp(i)$ contains the set of tasks whose priorities are higher than τ_i. The recursive equation converges when $w_i^{n+1} = w_i^n$ and this final value of w_i^n is the response time

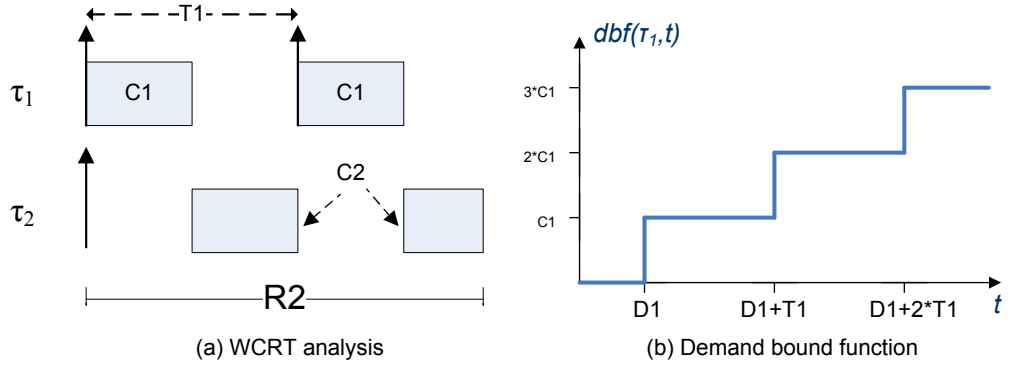


Figure 2.6: Examples of Schedulability analysis approaches.

(R_i) of τ_i. Then the feasibility test for a task set S is just to check that

$$\forall \tau_i \in S, R_i \leq D_i$$

where D_i is the explicit deadline of task τ_i. Figure 2.6(a) shows an example of WCRT calculation. The worst case response time of τ_2 (labeled as $R2$ in the Figure) is equal to the summation of its worst case execution time $C2$ and $2 \times C1$, the time spent for execution of a higher priority task τ_1 within $R2$.

Another more general technique is called the processor demand criterion[23]. The processor demand criterion quantifies the maximum amount of processor time that all the jobs generated by the tasks can require in an interval of specified size, and attempts to determine whether there is an interval-size for which the processor demand criteria summed over all tasks in the system exceeds the processor capacity. Schedulability analysis using the processor demand criterion will check that

$$\forall t \geq 0, t \geq \sum_{i=1}^{n} (\lfloor \frac{t - D_i}{T_i} \rfloor + 1) C_i$$

In other words, within any point of time t, the processor demand generated by all tasks in a task set must be smaller than or equal to the total available processing time (i.e., t). [23] shows that this checking only needs to be tested for a bounded value of t(the least common multiple LCM of all the task's period in the system). Figure 2.6(b) shows the processor demand of a task τ_1 over time t, given as the demand bound function $dbf(\tau_1, t)$. For example, no computation workload is required by τ_1 before its first deadline at time $D1$. While the processor must allocate $C1$ time unit to τ_1, for the time period between $[D1, D1 + T1)$, i.e., before the deadline due for the second arrival of τ_1. If a task set satisfies the processor demand criterion, there is an optimal uniprocessor scheduling algorithm(such as EDF) that manages to schedule it without missing any deadlines.

Chapter 3

Related Work

In this chapter, we present an overview of related work on timing analysis approaches for model based design framework.

3.1 WCET Analysis for Synchronous models

A real-life implementation generated from high-level synchronous models can be said to follow the synchrony hypothesis if all events that are logically assumed to be processed instantaneously are processed before the next set of events arrive. Verifying the synchrony hypothesis when a synchronous model is compiled into hardware is relatively straightforward. As a result, compiling synchronous language specifications directly into hardware is currently the most popular design flow [14].

On the other hand, when synchronous languages are compiled into software – e.g., into a sequential C code – validation of the synchrony hypothesis is more complicated

and depends on both the generated code, as well as the micro-architecture of the platform executing this code. For the synchrony hypothesis to hold, the estimated Worst-case Execution Time (WCET) associated with the processing of events should be less than the minimum separation time between the arrival of sets of events (that are assumed to be processed instantaneously).

3.1.1 High-level WCET analysis

Architecture independent high-level timing analysis of Esterel (or Esterel-like) programs have been studied in [16, 102, 75]. Given a synchronous language specification, the task of a high-level WCET analysis is to compute the worst case computation time for a particular input event (or all allowed inputs), in terms of number of clock ticks required. This is usually done by translating the synchronous specification into a finite-state machine whose transitions correspond to clock ticks in the model [100]. In other words, the *high-level* WCET analysis problem is concerned with the number of Esterel clock ticks, rather than the execution time of code within a clock tick. The timing analysis problem where the states of the automata have been annotated with WCET estimates has been discussed in [77]. Again, the focus here was not to obtain tight WCET estimates, but to analyze high-level timing properties of an Esterel specification, assuming that platform-level WCET estimates are already available.

[51] proposes an early stage WCET analysis to derive "approximate" WCET estimates at early stages of the software development process. Instead of performing executable code level WCET analysis on a target architecture, the early stage WCET anal-

ysis tries to construct a high-level timing model that contains approximate execution time information (i.e., a not guaranteed bound) on basic code constructs. Flow analysis is then performed on the timing model to find the worst case execution trace. Since the timing model can be built at any of the implementation levels between high-level model specification and executable code, early stage WCET analysis can be achieved. However, the analysis does not guarantee a safe upper bound on execution time, and its accuracy (how close to the real WCET) depends on many issues including (i) nature of the model specification (e.g., sequential or concurrent, imperative or declarative); (ii) model-to-executable compilation technique; (iii) and complexity of the target platform.

3.1.2 Code-level WCET analysis

Code-level WCET analysis for synchronous models aims to find architecture dependent execution time for computation within a single clock tick in the generated low-level executable code (e.g., C or assembly). [76] performs the low level WCET analysis for synchronous language Quartz (an Esterel-variation, see [100]) by building a formal transition model on the statements in the generated executable code. Transitions are labeled with the physical execution time of corresponding statements, and symbolic model checking is applied to search the "longest" WCET path. However, the presented technique is only applicable to automata-based code generation, which does not scale well for large Esterel programs.

The problem addressed in [95] is the closest to what we study in this thesis. Here, the problem of infeasible paths in the generated code is mentioned and timing anal-

ysis of the whole Esterel program is studied. Though the work can also be used for estimating the maximum computation in a clock tick, the methodology is restricted, since it requires two separate codes to be generated from the synchronous program — one on which the WCET analysis is performed, and one which guides the analysis. The approach in [95] is only feasible for the generation of circuit code, which tends to be slow for large-scale application specifications. Furthermore, the problem of bidirectional traceability or performance debugging of Esterel specifications – even though mentioned – was not studied on non-trivial Esterel benchmarks by including traceability links in an Esterel compiler.

Code-level WCET analysis techniques for other synchronous models, e.g., StateCharts [57] and MATLAB Simulink [103], have been also studied (e.g., [38, 65]). Recent advances in WCET analysis techniques and the availability of industry-strength tools (e.g., [42, 112]) has renewed the interest in synchronous language-based design flows targeting general-purpose platforms. [106] reports practical experiment results on performing WCET analysis for avionics programs, including a program that is automatically generated from a SCADE specification. A WCET analysis framework that integrates the synchronous language SCADE [99] from Esterel Technologies with the aiT WCET analyzer from AbsInt GmbH [1], targeting general-purpose processors, was presented in [59]. In [59], the WCET analysis is ignorant of the fact that the executable code is compiled from a high-level modeling language. On the other hand, our proposed model-driven timing analysis framework (discussed in Chapter 4, 5, and 6) automatically utilizes model-level information in low-level WCET analysis, which leads

to more efficient and tighter WCET estimation. Very recently, [108] proposes a technique to improve timing analysis for MATLAB Simulink/Stateflow model, by incorporating model-level flow information into WCET analysis. However, the model-level flow constraints are manually identified and translated into code-level flow constraints for WCET calculation.

3.1.3 Timing analysis for special-purpose architecture

Special-purpose reactive processors have been developed to support concurrent execution of Esterel specification, where instead of compiling into C code, the Esterel specification is mapped to a concurrent reactive processing ISA. Example architectures include EMPEROR [114], STARPro [115] and Kiel Esterel processor (KEP) [69, 71].

Timing analysis for execution of Esterel specifications on the special-purpose reactive processors have been studied in [69, 114, 19, 81]. Recently, similar approaches have also been followed for timing analysis of a synchronous version of C, called PRET-C, to be implemented on special precision timed or PRET architectures [96]. Compared to timing analysis of general-purpose processors, micro-architectural modeling for such special-purpose processors is simplified to a large extent. Furthermore, special-purpose reactive processors implement hardware supports for handling concurrency and other Esterel semantics (e.g, preemption and event broadcasting). As a result, timing analysis designed for these architectures can not be applied to the general-purpose processor setting. In first part of the thesis, we focus on timing analysis of Esterel specifications on general-purpose processor architectures.

3.2 Schedulability Analysis for Distributed System

Tindell et al. [110] propose a holistic schedulability analysis for distributed real-time systems, which bounds the worst case delays of both local computations and inter-processor communications. However, their analysis assumes only a simple static TDMA protocol for the bus communication, and the communication dependencies are not taken into consideration (thereby leading to coarse analysis and pessimistic results). In [89], timing analysis for distributed system connected via a shared FlexRay bus [44] is presented. The analysis focuses on bounding the messages transmission time in both the static and the dynamic segments of a FlexRay communication cycle. However, above-mentioned schedulability analysis techniques are applicable to task models with only communication dependencies, where control dependencies (e.g., conditional execution of tasks) are ignored. Recently, schedulability test has been proposed for *independent* sporadic tasks executed on uniform multiprocessors with global EDF scheduler.

Although the task graph based models introduced in Section 2.2.2 naturally represent periodically or sporadically executing applications [35, 86], they only provide *local* or *processor-centric* views of a distributed system. More specifically, the structuring mechanism used revolves around specifying all the tasks that execute on any given processing (or communication) element. As a result, they are not very suitable for specifying the *interactions* between the multiple entities of a distributed system – which is often a more natural way of specifying such systems.

There are two standard approaches for schedulability analysis of task graph-based specifications of real-time systems — worst-case response time analysis-based tech-

niques [22, 55, 68], and the processor demand bound criteria-based analysis [8, 23] (refer to Section 2.2.2 for a brief discussion). Recently, the response time analysis has been extended for schedulability analysis of periodic/sporadic task sets (of independent tasks) on multiprocessor platform [17, 50]. However, it turns out that neither of these approaches can be applied to our setting in a straightforward manner. This is primarily because in traditional task graph-based specifications, all the vertices are mapped onto a single resource, whereas in our case each globally asynchronous MSG (in fact even a vertex of an MSG denoting an MSC) involves multiple computation and communication resources. Hence, the semantics of MSGs are fundamentally different from the task graph based models that have been studied in the real-time systems literature.

Our proposed schedulability analysis for MSC-based system model is motivated by the response time calculation algorithm presented in [113], which can handle system specifications with multiple computation and communication elements. We have adapted this algorithm to the specific context of MSCs, and in particular proposed two new extensions.

- The algorithm in [113] is based on a response time analysis framework, which iteratively computes tighter estimates on the response times of various computation and communication tasks. However, it cannot handle conditional or non-deterministic branches which exist in MSGs. We get around this problem by combining the response time analysis-based technique in [113] with a demand bound criteria-based technique that was proposed in [7] to handle conditional branches in a different task model.

- Compared to [113], we also obtain tighter bounds on the response times of tasks by accounting for the dependencies in the preempting tasks/applications, by calculating request bound from higher priority tasks during the response time of the preempted task (event).

The main novelty of our work stems from the combination of response time analysis and demand bound criteria-based techniques, which is not commonly seen in the real-time systems literature.

Finally, we would like to point out that there have been a few previous attempts towards developing schedulability analysis techniques for MSC-based system models [98, 104]. However, they either do not fully exploit the communication dependencies within an MSC, or are restricted to the analysis of a single MSC (as opposed to a complete system model).

Chapter 4

Performance Analysis and Debugging of Esterel

In this chapter, we propose a timing analysis technique for WCET estimation of a single Esterel tick. Such estimates can validate Esterel-level assumptions on the instantaneous processing of signals or events that occur together (Section 2.1.1). More importantly, with our automatically built traceability between Esterel specification and generated C code, they can be used to identify parts of the specification which might pose as timing/performance bottlenecks with respect to the underlying platform. We show the results of our WCET analysis on a set of standard Esterel benchmarks and illustrate the utility of our model-to-code traceability technique using an Esterel specification of a reflex game application.

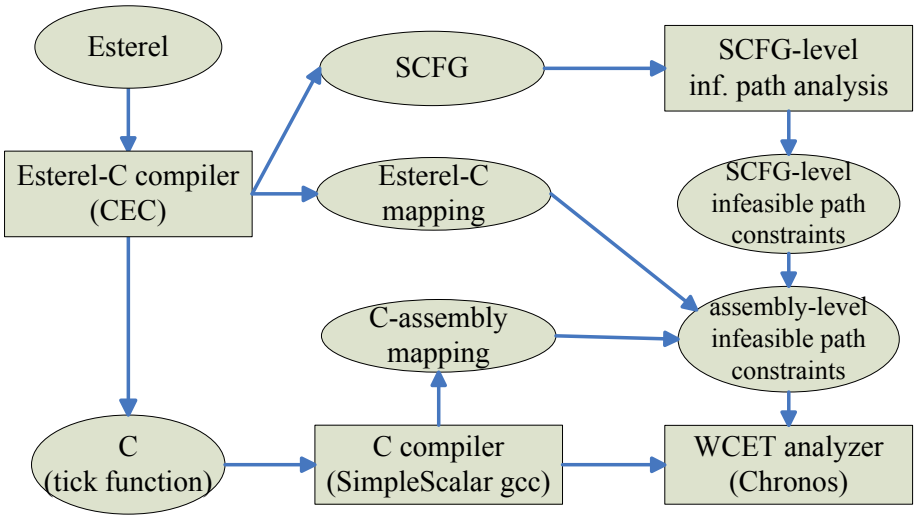


Figure 4.1: WCET analysis of a single Esterel tick.

4.1 Overview

Figure 4.1 gives an overview of our WCET analysis framework. We use the Columbia Esterel Compiler [34]) to compile a given Esterel program into C, and calculate the WCET of the C code via an ILP-based platform-aware WCET analyzer. The generated C program is in the form of a tick function (refer to Section 2.1.1), where one complete execution of the tick function (between its entry and exit) represents all computation and communication required to be processed instantaneously within one clock tick. As a result, we can validate that the synchrony hypothesis assumed at model level indeed holds in the real implementation, if WCET of the generated tick function is guaranteed to be less than the minimum separation time between the arrival of sets of input events.

For C code generated from Esterel specifications, the user can largely avoid the problems related to automation of the WCET analysis (refer to Section 2.2.1). In particular, since the tick function is loop-free (Esterel allows no loops within a clock tick),

this leads to an acyclic control flow graph and hence there is no need to provide loop bounds to the WCET analyzer. Since each basic block is executed at most once in one execution of the tick function, an ILP-based WCET analysis produces a 0-1 assignment for the execution count of each basic block. However, compiler generated programs, especially from high-level control-intensive specifications, usually contain a huge number of infeasible execution paths compared to hand-written code. As a result, WCET overestimation due to infeasible execution paths is largely amplified in timing analysis of compiler generated programs.

We propose a comprehensive and light-weight infeasible path detection and elimination technique for WCET estimation of programs generated from Esterel specification. Our proposed infeasible path detection is performed at sequential control-flow graph (SCFG) level, which is a standard intermediate representation used in control-flow graph-based Esterel compilation. The computed SCFG-level infeasible path constraints are translated into assembly-level infeasible path constraints, via our Esterel-C mapping (obtained by instrumenting the CEC compiler) and the C-assembly mapping (obtained by disassembling the compiled C code). Finally, we integrate the assembly-level infeasible path constraints into the ILP formulation generated by Chronos [41], an ILP-based WCET analyzer. The WCET value and corresponding critical path for a single tick execution of the Esterel specification on a specific platform can be obtained by solving the resulted ILP formulation. In summary, we use the pattern of the generated code to identify infeasible path patterns, which are then taken into account during the timing analysis.

4.2 Infeasible Path Patterns

We observe that the automatically generated C code (from Esterel) often contains certain infeasible path patterns which may be less frequent in hand-written C code. Thus, low-overhead automatic methods for detecting/exploiting infeasible path information can substantially reduce the WCET of such automatically generated C code. Based on our study of C programs generated via Esterel compilation to sequential control flow graphs, we found the following four common sources of infeasible paths. For each of these four sources, in Figure 4.2 we show example Esterel program fragments and the corresponding C code (labeled with line numbers) generated by the default code generation mechanism in the Columbia Esterel Compiler [34]. We adopt the notion of conflicting pairs which has been presented in Section 2.2.1 in our discussion of infeasible path patterns. The four infeasible path pattern categorizations are as follows.

1. Emit and test signals. The corresponding infeasible paths are also presented at the C level, *e.g.*, the conflicts due to assignment and test on signal A ($L1$ and $L2 \rightarrow L4$) in the first program fragment shown in Figure 4.2. Besides, in an Esterel clock tick, the same signal may be tested in different concurrent threads. As a result, in the generated C program, multiple identical tests on the same signal variable will result in paths with (branch, branch) conflicts (refer to Section 2.2.1). For example, the two conditional tests on signal A at $L2$ and $L5$ in the first program fragment in Figure 4.2 introduce two conflicting pairs, $(L2 \rightarrow L3, L5 \rightarrow L7)$ and $(L2 \rightarrow L4, L5 \rightarrow L6)$.

Type 1	Generated C code	Type 2	Generated C code

emit A;
present A
then emit B
else emit C;
end present
||
present A
then emit D;
else emit E;
end present
	L1: **A = 1;**
	L2: **if (A)**
	L3: B = 1;
	else
	L4: C = 1;
	L5: **if (A)**
	L6: D = 1;
	else
	L7: E = 1;
		emit A;
		present B
		then …
		end present
		||
		present A
		then
		emit B;
		else emit D;
		end present
			L1: A = 1;
			L2: **_DPSCUT_VAR2 = 0;**
			L3: if (A) B = 1;
			L4: else D = 1;
			L5: **if (_DPSCUT_VAR2)** {
			L6: …
			} else {
			L7: if (B) {…}
			}

Type 3	**Generated C code**	**Type 4**	**Generated C code**

trap T in
[
exit T;
pause;
…
||
emit B;
pause;
…
]
emit C
	/*exit T */
	L1: **_term_17 &= -(1 << 2)**;
	L2: B = 1;
	/*pause */
	L3: **_term_17 &= -(1 << 1)**;
	…
	L4: **switch (~_term_17) {**
	L5: case 0: … break;
	L6: **case 1**: … break;
	//pause
	L7: **case 3**: …
	L8: C = 1; break; //exit T
		loop
		emit A0;
		pause;
		emit A1;
		pause;
		||
		emit B0;
		pause;
		emit B1;
		pause;
		end
			L1: **if (_state_3)** {
			L2: A0 = 1; **_state_3 = 0;**
			} else {
			L3: A1 = 1; **_state_3 = 1;**
			}
			L4: **if (_state_6) {**
			L5: B0 = 1; **_state_6 = 0;**
			} else {
			L6: B1 = 1; **_state_6 = 1;**
			}

Figure 4.2: Example infeasible path patterns in generated C code.

2. Sequentialization of concurrency in a tick. To generate sequential C code from a concurrent Esterel program, communication dependencies (between emit and test of a signal) and context switches between concurrent threads must be captured. In CEC, this is handled by inserting new control variables and corresponding test nodes in the generated C code, when the concurrent control flow graph CCFG is translated into sequential control flow graph SCFG (refer to Figure 2.2). In the second program fragment (Figure 4.2), the variable $\_DPSCUT\_VAR2$ captures the *state* of the first thread before a context switch (by setting its value to 0 at $L2$), and is used as a conditional guard when the thread resumes execution at $L5$. Such assignments and tests (may be at multiple places in the same clock tick) on the guard will introduce possible infeasible paths. In our example code fragment, the

(assignment, branch) conflicts (refer to Section 2.2.1) between $L2$ and $L5 \rightarrow L7$ introduces infeasible paths in the generated C code.

3. Termination and preemption. The multi-threaded Esterel program follows the "wait for all threads to terminate" and "winner takes all" behaviors for thread completion and thrown exceptions [34]. In the C code generated from CEC, this is handled by setting and testing the values of newly introduced guard variables (e.g. variable $\_term\_17$ as in the third example in Figure 4.2). These guard variables are assigned to non-negative integer values during the execution of each thread (0 for thread terminating, 1 for pausing, 2 and higher for throwing an exception). Such assignments and the tests on these guard variables (e.g., $(L1, L4 \rightarrow L6)$ on value of $\_term\_17$) introduce possible infeasible paths.

4. Encoding tick transitions. In Esterel, a global automaton is defined on the sequence of ticks to be executed in each thread, via the use of "pause" and "await" statements. In the generated C code, this automata is encoded through a set of state variables. Setting and testing these state variables introduce infeasible paths since certain combinations of states are not allowed in the automata. For example, in the last program fragment, given the initial value [0,0], the combinations of values of ([$\_state\_3$, $\_state\_6$]) can *only* be [0,0] or [1,1] — which prevents the paths corresponding to evaluation of [$\_state\_3$, $\_state\_6$] to be [0,1] or [1,0]. In particular, the path contains branches $L1 \rightarrow L2$ and $L4 \rightarrow L6$ cannot appear in any feasible execution trace of the program.

4.3 SCFG-level Infeasible Path Detection

In traditional WCET analysis, infeasible path detection is usually done via flow analysis at assembly code level [107, 52]. In our previous work [62], we perform infeasible path elimination on C code generated from Esterel by capturing conflicting pairs at assembly code level. However, our previous experimental results in [62] show that due to large number of infeasible paths in the generated C code (several thousands of detected conflicting pairs), the analysis is complex and takes up to 15 minutes for the *mca200* benchmark from Estbench Esterel Benchmark Suite [33].

In this section, we propose a light-weight infeasible path detection at higher level during Esterel compilation. By maintaining a traceability link between model-level statements and compiled executables (the details will be discussed in Section 4.5), we automatically translate the high-level infeasible path information captured via conflicting pairs into assembly code level ILP constraints, which can be used in code-level WCET analysis to obtain tighter estimation results.

There are many levels of intermediate representations (IRs) while compiling an Esterel specification into assembly code for WCET estimate. For example, the control-flow graph based CEC compilation produces IRs including AST, GRC, PDG, CCFG and SCFG (refer to Figure 2.2). In our work, we perform our infeasible path detection at SCFG level because of the following reasons.

- Any intermediate representations at higher level than SCFG (refer to Figure 2.2) does not contain all the infeasible path patterns. For example, the second type of

infeasible path pattern due to sequentialization is only introduced when CCFG is translated into SCFG.

- C (assembly) code level analysis is incomplete without additional instrumented code. For example, the third type of infeasible path patterns will often produce many *switch-cases* constructs in the generated C code. the *switch* is translated into a *register indirect jump (jr)* in some ISAs [24], i.e. the branch target can not be determined statically.

- It improves the efficiency of conflicting pair detection. There are much less number of nodes in SCFG comparing to the number of assembly instructions. Moreover, no register/memory tracing or pointer analysis is required. Thus, the infeasible path analysis takes much less time at SCFG-level.

- As we will discuss in detail later, the state automata construction for detecting the fourth type of infeasible path is obviously easier to perform at SCFG level other than at other lower level representations (C or assembly). Furthermore, a preliminary SCFG level conflicting pair detection allows us to build an automaton to capture program execution context for tighter timing estimation for execution of multiple consecutive ticks (see Chapter 5).

Figure 4.3 shows an example Esterel program, its SCFG generated by CEC (partial), and some conflicting pairs in the SCFG with our infeasible path pattern categorization in Section 4.2.

```
trap FIN in
[  loop
      present IN then   emit A; end present;
      pause;
      present B then  emit C; end present;
      pause;
   end loop
||
   loop
      emit  D;
      pause;
      emit B;
      present C then  exit FIN; end present;
      pause;
end loop ]
```

Infeasible paths

Type 1: Emit and Test signals
emit B (n3) <> not present B (n6->n7)
Type 2: Sequentialization of concurrency
VAR2= 0 (n2) <> VAR2 ==1 (n7->n8)
Type 3: Termination and preemption
Termination code = 2 (n9)
<> Termination code == 1 (n10->n11)
Type 4: Encoding tick transition
value combination of state variables
[s1, s9,s6] can not be [1, 0, 1]

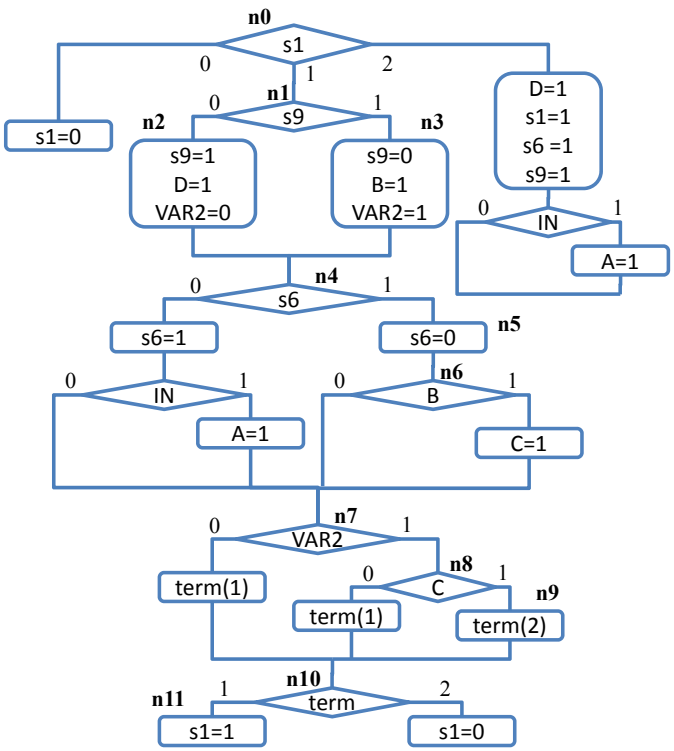


Figure 4.3: Conflicting pairs in SCFG of an Esterel Program

4.3.1 Detection of Infeasible Paths Type 1-3

The first three categorizations of infeasible paths listed in Section 4.2 are between pairwise assignments and branches. To capture them in the SCFG, we adopt the notion of *conflicting pairs* — pairs of (assignment, branch) or (branch, branch) statements which may not appear together in an execution trace (refer to Section 2.2.1). For example in Figure 4.3, node $n3$ (which assigns $B = 1$) and branch $n6 \rightarrow n7$ (to be taken when $B == 0$) is an (assignment, branch) conflict.

Conflicting pairs are easy-to-compute at SCFG-level. In Esterel semantics, a signal is either emitted or absent throughout any tick execution, corresponding to value 1 or 0 in the generated C code. Furthermore, domains of all compiler-introduced variables are statically known. We use the following notations in our discussion:

- DM_x is the domain of all possible values a variable x can be assigned to.
- $AS_{x,v}$ is the set of all nodes n in SCFG that contain assignment of x to value v.
- $BR_{x,v}$ be the set of all branch edges e in SCFG that are taken when $x == v$.

The (assignment, branch) and (branch, branch) conflicts on value v of variable x are represented as 4-tuple (n, e, x, v) and $(e1, e2, x, v)$. We compute the set of all conflicting pairs as follows.

$$\begin{aligned} AB &= \cup_{\forall v,v' \in DM_x} \{(n, n1 \rightarrow n2, x, v) | n \in AS_{x,v} \wedge (n1 \rightarrow n2) \in BR_{x,v'} \\ &\qquad \wedge reach(n, n1) \wedge v \neq v'\} \\ BB &= \cup_{\forall v,v' \in DM_x} \{(n1 \rightarrow n2, n3 \rightarrow n4, x, v) | (n1 \rightarrow n2) \in BR_{x,v} \\ &\qquad \wedge n3 \rightarrow n4 \in BR_{x,v'} \wedge reach(n2, n3) \wedge v \neq v'\} \end{aligned} \tag{4.1}$$

where $reach(n1, n2)$ is true if and only if there is a path from node $n1$ to $n2$ in the SCFG. For each (assignment, branch) conflict $(n, n1 \rightarrow n2, x, v)$ in AB and (branch, branch) conflict $(n1 \rightarrow n2, n3 \rightarrow n4, x, v)$ in BB, we also compute set of nodes in SCFG that may invalidate the conflicting pair through assigning a different value to x. In other words, if x is assigned to a different value in between, the conflict between the assignment (test) on x in n ($n1 \rightarrow n2$) and the test on x in $n1 \rightarrow n2$ ($n3 \rightarrow n4$) is no longer valid. We compute the set of nodes that invalidate each conflicting pairs as follows.

$$\begin{aligned} invalid(n, n1 \rightarrow n2, x, v) &= \{n' | reach(n, n') \wedge reach(n', n1) \\ &\qquad \wedge n' \in AS_{x,v'} \wedge v \neq v'\} \\ invalid(n1 \rightarrow n2, n3 \rightarrow n4, x, v) &= \{n' | reach(n2, n') \wedge reach(n', n3) \\ &\qquad \wedge n' \in AS_{x,v'} \wedge v \neq v'\} \end{aligned} \tag{4.2}$$

Computation of the above-mentioned sets AS, AB and BB can be done in a single DFS traversal of the SCFG in $O(|N|+|E|)$ time. For each conflicting pair in AB or BB on variable x, finding the set of nodes that may invalidate it requires time $O(|N| \times |E|)$ to perform a reachability test for each node that updates value of x (captured in set AS) The set of conflicting pairs and corresponding "invalid" nodes will be used to generate ILP constraints for infeasible path elimination (refer to Section 4.4).

4.3.2 Detection of Infeasible Paths Type 4

When Esterel specification is compiled into C code, a set of state variables are introduced to encode the program execution context. Let's define a global state as a value

State	State variable values	Next State(s)
st_0	$[s1 == 2, s6 == \perp, s9 == \perp]$	st_1
st_1	$[s1 == 1, s6 == 1, s9 == 1]$	st_2, st_3
st_2	$[s1 == 1, s6 == 0, s9 == 0]$	st_1
st_3	$[s1 == 0, s6 == 1, s9 == 1]$	

Table 4.1: Feasible States of the example SCFG shown in Figure 4.3.

combination of all the state variables. The number of state variables introduced in a control-flow graph based Esterel compilation usually depends on the number of concurrent threads in the Esterel specification, where each state variable captures the current tick of its corresponding thread (by having different values for each tick). For example, three state variables $[s1, s6, s9]$ are used in the SCFG shown in Figure 4.3.

Given the initial state and allowed finite state transitions defined by Esterel, certain value combinations of the state variables are *not* reachable. Our type 4 infeasible path pattern due to such unreachable combinations cannot be simply captured by above-mentioned conflicting pairs, since (i) an infeasible path of this kind consists of many branches on state variable tests; and (ii) this type of the infeasible path patterns is "context-sensitive", i.e., a state variable's value in the current tick depends on the execution in the previous tick.

Given a SCFG $scfg$ and its initial state st_0, Algorithm 1 shows how to find an *overestimated* set of all feasible states FS (i.e., the set contains *all* feasible states with possibly some infeasible ones). For a feasible state st, we compute feasible states st'

Algorithm 1 $computeFeasibleState(scfg, st_0)$ — Compute set of all feasible states FS, where st_0 is the known initial state for SCFG $scfg$.

1: $FS.add(st_0)$; $Queue.insert(st_0)$;
2: **while** !$Queue.empty()$ **do**
3: $st = Queue.remove()$;
4: **for** each feasible path $p \in scfg$ **do**
5: $st' = st$;
6: $curFLAG = true$;
7: **for** each branch e on p **do**
8: **if** $e \in BR_{s_i,v} \wedge st[i] \neq v$ **then**
9: $curFlag = false$; /*path p is not in current state st*/
10: $break$;
11: **end if**
12: **end for**
13: **if** !$curFlag$ **then**
14: $break$; /*search next path*/
15: **end if**
16: **for** each assignment node n on p **do**
17: **if** $n \in AS_{s_i,v} \wedge st[i] \neq v$ **then**
18: $st'[i] = v$;
19: **end if**
20: **end for**
21: **if** $st' \notin FS$ **then**
22: $FS.add(st')$;
23: $Queue.insert(st')$;
24: **end if**
25: **end for**
26: **end while**

that execute in the next tick after st. We consider all feasible paths in the SCFG by excluding the infeasible paths captured by conflicting pairs as calculated in Section 4.3.1 (line 4). For a feasible path p, we first test whether p corresponds to execution of current analyzing state st (line 6-15). If any branch on a state variable s_i that can be taken when $s_i == v$ for a different value v from the value of s_i in st (line 8), the path p will not be considered when searching st's next states (line 13 -14).

Otherwise, for each assignment that updates state variable s_i to a new value v (in the set $AS_{s_i,v}$) on p, we set the value in st' correspondingly (line 16-20). Finally, if the newly computed feasible state st' has not been visited before, we add it into the set of feasible state FS, as well as the workspace $Queue$ so that states reachable from it will be computed in the future (line 21-24). Table 4.1 shows the list of all feasible states that are reachable from the initial state $[s1 == 2, s6 == \perp, s9 == \perp]$ (where $s6$ and $s9$ are undefined) of the example SCFG shown in Figure 4.3.

Let S be the set of all possible value combinations on all the state variables $< s_1, \ldots, s_n >$,

$$S = \prod_{i \in \{1,\ldots,n\}} DM_{s_i}$$

The set of all infeasible state IS can be obtained by $IS = S - FS$, where FS contains the set of all value combinations for reachable states as calculated above. Note that we perform a conservative static simulation for reachable state calculation, which reports a superset of the "real" reachable states. We utilize the computed reachable states to obtain a safe subset of unreachable state variables' value combinations, and generate ILP constraints to eliminate the corresponding infeasible paths (see Section 4.4).

4.4 Infeasible Path Elimination

We now discuss methods to eliminate infeasible paths given the conflicting pairs and feasible state variables's value combinations. We generate an ILP constraint for each conflicting pair and infeasible value combination detected at SCFG level. As shown in Figure 4.1, we maintain an Esterel-C mapping which provides traceability between corresponding Esterel statements, SCFG nodes, and generated C statements. Together with the C-assembly mapping between C statements and basic blocks/edges in the CFG of generated C code, we can *automatically* translate the SCFG-level ILP constraints into assembly code level ILP constraints. These assembly code level ILP constraints are utilized in an ILP-based WCET analyzer to prevent infeasible paths from being considered as the WCET critical path (thereby leading to a tighter WCET estimate).

Even after the conflicting pairs are detected, we cannot directly use them in our ILP-based WCET analysis. Suppose we find that an assignment to a variable x in SCFG node n_i conflicts with a branch edge $e : n_j \rightarrow n_k$ (edge e between node n_j and n_k) on the same variable x with effect constraint $x == v$. A straightforward encoding of this conflicting pair as a linear constraint would be $N_i + E_{j \rightarrow k} \leq 1$ where N_i ($E_{j \rightarrow k}$) is the 0-1 execution count of node n_i (edge $n_j \rightarrow n_k$). The above constraint means that node n_i and edge $n_j \rightarrow n_k$ cannot be executed together. However, a conflicting pair captures a pair of statements which cannot be executed together *provided the variable resulting in the conflict is not modified in between the execution of these two statements*. For example, in Figure 2.4, the branch edge $B0 \rightarrow B2$ (corresponding to $y == 0$) conflicts with $B6 \rightarrow B7$ (corresponding to $y \neq 0$). However, this (branch, branch) conflicting

pair cannot appear in a real execution trace if y is not modified in between. In other words, for this conflicting pair to be a valid one, node $B5$ (which modifies the value of y) must not be executed in between. This leads to the constraint

$$E_{0\to 2} + E_{6\to 7} - N_5 \leq 1$$

Let $(n_i, n_j \to n_k) \in AB$ (or $(n_i \to n_j, n_k \to n_l, x, v) \in BB$) on variable x and its value v be a conflicting pair (Equation 4.1), and $invalid(n_i, n_j \to n_k, x, v)$ (or $invalid(n_i \to n_j, n_k \to n_l, x, v)$) be the set of nodes that invalidate it (Equation 4.2). Formally, we can encode this conflicting pair as a linear constraint

$$\begin{aligned} N_i + E_{j\to k} - \textstyle\sum_{\forall n_p \in invalid(n_i, n_j \to n_k, x, v)} N_p &\leq 1 \\ E_{i\to j} + E_{k\to l} - \textstyle\sum_{\forall n_p \in invalid(n_i \to n_j, n_k \to n_l, x, v)} N_p &\leq 1 \end{aligned} \tag{4.3}$$

where N_i ($E_{j\to k}$) is the 0-1 execution count of SCFG node n_i (edge $n_j \to n_k$).

For the fourth type of infeasible path pattern, we generate following ILP constraints for each infeasible state $st : [s_1 == v_1, \ldots, s_n == v_n]$ in IS (refer to Section 4.3.2).

$$E_1 + \ldots + E_n < n, \forall e_i \in BR_{s_i, v_i}$$

where E_i is the 0-1 execution count of edge e_i, for all $e_i \in BR_{s_i, v_i}$ that can be taken if state variable $s_i == v_i$ in the infeasible state st. In other words, we prevent the longest path to contain state variable evaluation corresponding to an infeasible state st. Hence, not all branches that can be taken in the infeasible state st are allowed to execute in a single tick.

Given the example in Figure 4.3, the ILP constraint

$$E_{0\to 1} + E_{1\to 2} + E_{4\to 5} < 3$$

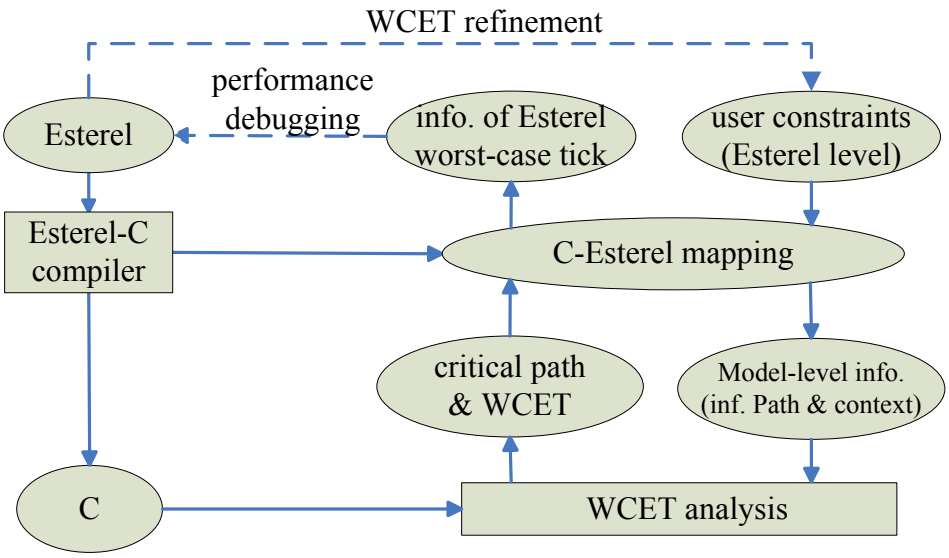


Figure 4.4: Performance debugging framework for Esterel specifications.

will be generated to eliminate the infeasible path containing edges $n0 \rightarrow n1$, $n1 \rightarrow n2$, and $n4 \rightarrow n5$, which corresponds to an infeasible control state $[s1 == 1, s9 == 0, s6 == 1]$. Given the one-to-one mapping between nodes/edges in SCFG and basic blocks/edges in assembly code level CFG, Above constraints can be automatically translated into assembly code level ILP constraints for infeasible path elimination in WCET analysis.

4.5 Performance Debugging and WCET Refinement

In this section, we discuss how to achieve the bi-directional traceability and use it for performance debugging and WCET refinement of Esterel specifications, as shown in Figure 4.4. If the WCET estimate produced for the C-level tick function is greater than a pre-defined clock tick length, we have a violation of the synchrony hypothesis. It

is then useful to show the programmer the Esterel statements executed corresponding to the WCET estimate. To provide such backwards traceability, a C-Esterel mapping is built during code compilation (Figure 4.4). This mapping is used to generate the Esterel-level critical path (statements executed when the WCET is realized) from the C-level critical path produced by the WCET analyzer. By visualizing these Esterel statements, the programmer can perform optimization/modification of the Esterel specification.

Assembly to C mapping State-of-the-art WCET analysis tools typically perform the analysis on assembly code (which obtained by disassembling the program binary) rather than source code. This is to take into account the effect of compiler optimizations for accurate timing estimation. For an ILP-based WCET analyzer, the WCET estimate is given via basic block counts, where each basic block is a sequence of assembly instructions. Our first step towards maintaining backwards traceability is to provide a mapping from assembly to C code. This can be easily achieved by disassembling the C object file using the *objdump* command, which produces the link between assembly instructions and the corresponding C code.

C to Esterel mapping To enable a mapping from the C-level WCET path back to the Esterel level, we maintain traceability links while compiling Esterel to C. In order to impose minimum overheads on the Esterel to C compilation, we only need to maintain the C to Esterel mapping for a subset of Esterel statements. We only trace the Esterel statements that are eventually translated into C statements (such as data and signal processing, conditional statement, preemption statements, etc.) and affect the execution

time of the generated C program. For Esterel statements that only affect the control flow of the C code and produce no explicit execution costs, we do not need to monitor them during the compilation process. In particular, we classify the Esterel statements into following four categories.

- Data and signal processing statements (e.g. expressions, assign, procedural call, emit). These statements need to be traced, because they are directly translated into C statements, and will explicitly affect the execution time of both the Esterel and C programs.

- Conditional and preemption statements (e.g. if-then-else, present-then-else, abort-when, trap-exit). We trace the predicate signal/expressions for these statements, which are translated into conditional tests in the C code. This is to reflect the time taken to evaluate and test these predicates. Furthermore, tracing these predicates helps to automatically generate constraints in WCET analysis, based on the programmer's annotation given at Esterel level. We will discuss the details in Section 4.6.3.

- Other control flow statements (e.g. ||, ;, loop, pause). These statements are translated into control flow in the generated C program. There is no explicit C statements corresponds to them. As a result, we do not trace these statements.

- Variable/signal declaration statements (e.g. signal, var, input, output). These statements are not traced since they are compiled into variable declarations in the C program.

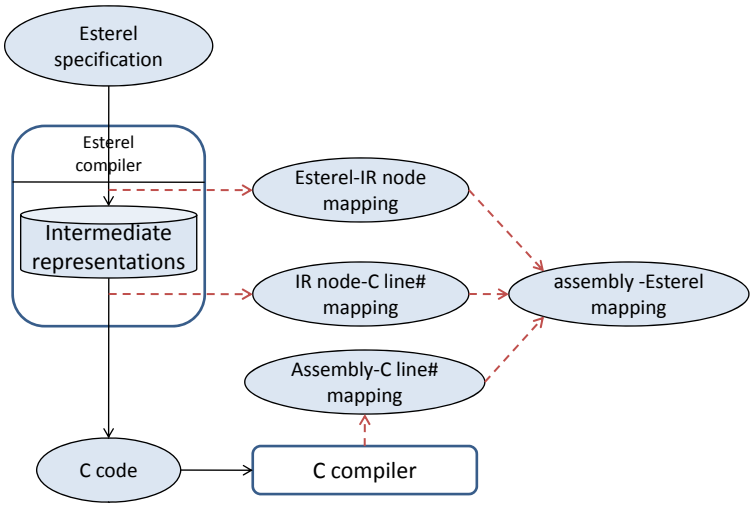


Figure 4.5: Construction of the assembly-Esterel mapping in Figure 4.4.

When an Esterel program is compiled to C, it is first translated to an intermediate representation (IR), *e.g.*, the abstract syntax tree (AST). The list of IRs used in CEC compilation is shown in Figure 2.2. During the AST construction, we maintain a mapping from Esterel line numbers to the IR node ids. Subsequently the AST is transformed into sequential control flow graph (SCFG) which sequentialize Esterel's concurrency. However, the computation/predicate nodes of the AST that we trace are retained in the SCFG. Hence, we can map the AST nodes to SCFG nodes. A mapping between IR node ids and C line number is created when SCFG is translated into C program. As shown in Figure 4.5, by combing all above-mentioned mappings, we can construct a mapping between the assembly code and Esterel specification.

Mapping back the longest path Recall that the ILP-based WCET analysis (as discussed in the preceding section) only reports the WCET estimate; it does not produce the corresponding longest path (also called the *critical path*). However, the control flow

graph of the Esterel tick function is a directed acyclic graph or DAG and each basic block is executed at most once. C statements executed in the critical path of the tick function can be reconstructed easily from the 0-1 assignments of the basic block counts via the assembly to C mapping (*any C statement appearing in a basic block with execution count 1 must lie on the critical path*). Finally, via our C to Esterel mapping, the Esterel statements corresponding to the WCET can be obtained. However, since infeasible path detection methods are incomplete, the reported critical path may, in principle, still be an infeasible path. Hence, we allow the programmer to provide infeasible path annotations at the Esterel level. These are automatically translated into ILP constraints on the execution counts of the C program's basic blocks via our traceability links between Esterel, C and the assembly code.

What kind of infeasible path annotations can be provided at the Esterel level? Esterel allows the programmer to explicitly define $\#$ (exclusion) and $=>$ (implication) relations on signals. These are constraints on the environment of the Esterel specification (*e.g.*, signals x and y never happen in the same tick) which are automatically translated to ILP constraints for tighter WCET analysis. We have also extended the $\#$, $=>$ relations to Esterel statements and predicates. In particular, we have defined two relational operators, $\#\#$ (*conflict*) and $<=>$ (*coexist*), between Esterel statements/predicates (represented using their line numbers) that we trace when building the C-Esterel mapping. These annotations can be automatically translated into ILP constraints as follows. A *conflict* annotation $A\#\#B$ is translated into the linear constraint $N_A + N_B \leq 1$ and a *coexist* annotation $A <=> B$ is translated into the linear constraint $N_A = N_B$, where

$N_A(N_B)$ is the execution count of the basic block that contains $A(B)$ if $A(B)$ is a statement, or the execution count of the corresponding branch edge (evaluating to true) if $A(B)$ is a predicate.

4.6 Experimental Results

4.6.1 Experiment Setup

We now present some implementation details and experimental results to evaluate our proposed WCET analysis for a single Esterel tick execution. We compiled Esterel programs into C using the default (control-flow graph based) code generation mechanism in the Columbia Esterel Compiler (CEC) [34]. We instrumented CEC so that during the compilation a C-Esterel mapping is created. We used Chronos [41], an ILP-based WCET analyzer, to calculate the WCET of the tick function in the generated C code. For the WCET analysis, the default architectural configuration of the tool was used, which assumes a direct mapped L1 instruction cache with 8-byte block size, dynamic 2-level branch predictor, 5-staged pipeline, and an instruction dispatch queue size of 4. We assume no data cache and the instruction cache miss penalty is 30 cycles.

We used benchmarks from Estbench Esterel Benchmark Suite [33], including a runner's behavioral description (*runner*), a simple combination lock (*abcd*), a shock absorber (*mca200*), and a *wristwatch* example.

Benchmark	# of lines		conflicting	WCET (cycle)			sim.	overest
	Esterel	C	pairs	w/o inf.	w/ inf.	reduction	(cycle)	
runner	55	253	12	3905	3781	3.2%	3589	5.4%
reflex	96	378	41	5197	4971	4.4%	4649	6.9%
abcd	101	827	437	10335	8463	18.1%	8099	4.5%
mejia	555	2598	3359	25983	23343	10.2%	18834	23.9%
tcint	687	3031	8319	13497	10949	18.9%	8869	23.5%
wristwatch	1088	2831	3307	40862	27773	32%	22300	24.5%
mca200	7269	10894	2993	129038	99396	23%	89541	11%

Table 4.2: WCET analysis results.

4.6.2 WCET Analysis Results

Table 4.2 summarizes our WCET analysis results. For each program, we show the code size of the Esterel specification and the generated C program. The number of conflicting pairs automatically detected by our infeasible path detection algorithm are also listed. Comparing to normal hand-written programs, huge number of conflicts (infeasible paths) exist in automatically generated code. The analysis time spent on finding these conflicting pairs takes less than 1 second for each the benchmarks. The calculated WCET values without and with the integration of SCFG level infeasible path constraints for each benchmark are presented in column "w/o inf." and "w/ inf.". We can see 3.2% to 32% tighter WCET estimates can be obtained with our automatical infeasible path elimination technique.. A tighter WCET value improves the accuracy of the synchrony hypothesis validation, and provides system engineers with more flexibility in terms of design choices. Finally, we compare our WCET estimation (in "w/ inf") with the Sim-

Benchmark	asm-level analysis in Chronos V3		our SCFG-level analysis		
	WCET (cycle)	analysis time	WCET (cycle)	WCET reduction	analysis time
runner	3905	6.9s	3781	3.2%	0.008s
reflex	5197	11.8s	4971	4.4%	0.003s
abcd	8463	41.2s	8463	0%	0.02s
mejia	25238	2m15s	23343	7.5%	0.2s
tcint	13057	3m15s	10949	16.1%	0.5s
wristwatch	31278	2m34s	27773	11.2%	0.12s
mca200	113519	15m18s	99396	12.4%	0.82s

Table 4.3: Comparison with assembly code level infeasible path detection.

pleScalar [5] simulation results shown in column "sim." using the same architecture configuration. The potential WCET overestimation is presented in "overest". However, the simulation results are usually an under-estimation of the *real* WCET values. C programs generated from Esterel specifications are control-intensive programs that handle many concurrent input events in a single tick execution and have complex internal control states. Hence, worst-case inputs and program control flow contexts are difficult to identify in the simulation. As a result, the presented ratio only serves as an upper bound of the overestimation between our estimated WCET and the real WCET.

Assembly-level infeasible path detection We have also compared our SCFG-level infeasible path detection with the build-in assembly code level infeasible detection in Chronos V3 (which is also based on the notion of conflicting pairs). The WCET results and infeasible path analysis time are shown under "SCFG-level" and "asm-level" in

Table 4.3, respectively. As one can see, our proposed SCFG-level analysis outperforms the assembly code level infeasible path detection in terms of both accuracy and analysis time.

Compiler optimization Modern C compilers support optimizations to increase the performance of compiled code. For example, the `gcc` *-O3* option performs optimization techniques including dead code elimination, code-block reordering, function inlining, and register renaming. However, such optimizations may potentially increase the binary code size.

The above-mentioned experiments are based on WCET analysis of non-optimized assembly code (i.e., with *-O0* option in `gcc` compilation). The control-flow structure of a non-optimized assembly code is very close to that of SCFG in Esterel compilation. On the other hand, if the C code is compiled with optimizations, there may be large differences in the control-flow structures between the optimized assembly code and SCFG. As a result, (i) infeasible paths identified at SCFG level may not exist in the optimized assembly code, and (ii) the optimization may introduce additional infeasible paths which are not presented at SCFG level. However, any infeasible path detection must be sound, but may be incomplete. In other words, by simply discarding the SCFG-level infeasible path constraints that cannot be matched in assembly code level, we are still able to obtain a safe (but probably less accurate) WCET estimation.

Table 4.4 shows our experimental results with C compiler optimization. We show the number of infeasible path constraints obtained at SCFG level. We discard SCFG

Benchmark	# of infeasible path constraints			-*O3* WCET (cycle)		
	SCFG	asm(-*O0*)	asm(-*O3*)	w/o inf.	w/ inf.	sim
runner	12	12	7	2681	2557	2304
reflex	41	28	13	4329	4171	3054
abcd	437	384	114	7605	6189	5395
wristwatch	2831	2278	512	28476	21925	13631

Table 4.4: WCET results with C compiler optimization.

level constraints that cannot be matched at assembly code level (due to missing statements and/or branches). The number of "survived" constraints are shown in Table 4.4 for both -*O0* and -*O3* optimizations. In particular, the "asm(-*O0*)" column shows the number of SCFG-level infeasible path constraints that are successfully translated into assembly code level constraints when no optimization is used in C compilation. These constraints are used to produce the WCET results with SCFG level infeasible path constraints as shown in Table 4.2 and Table 4.3. On the other hand, the "asm(-*O3*)" column shows the number of SCFG-level constraints that can be translated into assembly code level constraints when -*O3* is used for gcc compilation. Finally, we present the WCET estimations for -*O3* optimized assembly code, without and with the SCFG level infeasible path constraints (as shown in in column "asm(-*O3*)"), as well as the simulated WCET for each benchmark.

```
1   module reflex_game
    …
7   relation ..., READY # STOP
    …
14  every COIN do
    …
22    [
23      copymodule AVERAGE
24          ||
          …
38          trap END_MEASURE in
39          [
40            every READY do
41              emit RING_BELL
42            end
43            ||
        …
52      do
53        do
54          every MS do
55              TIME:=TIME+1
56          end
57        upto STOP;
58        emit DISPLAY;
59        emit INC_AVE(TIME)
60      watching LIMIT_TIME MS
61      time out exit ERROR end;
62      emit GO_OFF;
63      exit END_MEASURE
64    ] %trap END_MEASURE
        ...

87  module AVERAGE
    …
91    every immediate INC_AVE do
92      TOTAL := TOTAL + ?INC_AVE
93      NUM := NUM +1;
94      emit AVE_VALUE (TOTAL/NUM)
95    end
    …
```

Figure 4.6: The reflex game Esterel specification and highlighted critical path.

4.6.3 Case Study in Performance Debugging

We illustrate our timing analysis framework using the well-known reflex game example [13]. A user can start a game session by inserting one COIN to a machine. To test his reflex time, once the user is ready (by pressing the $READY$ button), he needs to press $STOP$ as quickly as possible after the machine generates a GO_ON signal to turn on a light. This is repeated three times and finally the average reflex time will be calculated and displayed before game is over. Figure 4.6 shows an Esterel fragment of the game controller. The complete Esterel specification of the game can be found in [13] (game version 1).

We used CEC to compile the reflex game program. We instrumented CEC to produce a C-Esterel mapping as discussed earlier. Automated infeasible path detection and ILP-based WCET analysis were performed on the generated C code. Once the critical path was computed at the C level, we identified the critical path at the Esterel level via

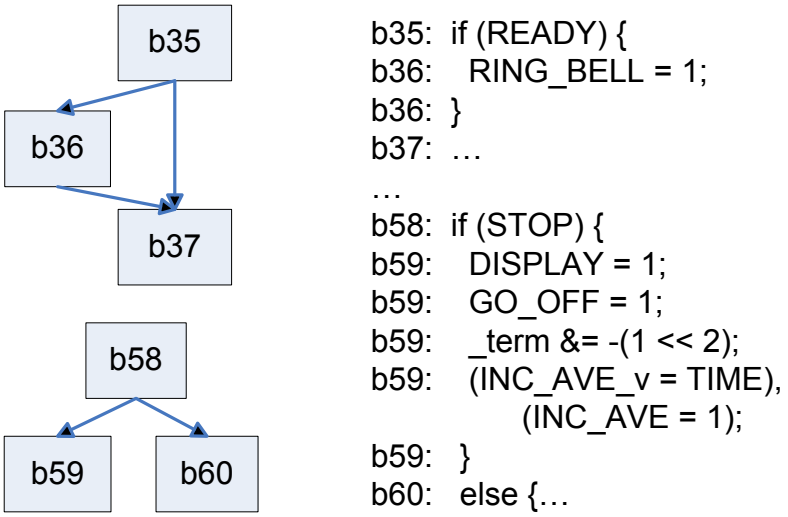


Figure 4.7: C-level critical path of the reflex game.

backwards traceability. Figure 4.7 shows a CFG fragment of the reflex game example, where assembly code level basic blocks in the critical path (blocks which have an execution count of 1) computed by our WCET analyzer are highlighted. Figure 4.7 also shows the corresponding C code fragment, through the assembly to C mapping. Finally, via the C to Esterel mapping, the corresponding Esterel statements executed in the worst case path are obtained. Thus, the C code fragment shown in Figure 4.7 corresponds to Esterel lines 40-42, 58-63 in Figure 4.6.

The entire Esterel level critical path is highlighted using shaded lines in Figure 4.6. It corresponds to the user pressing $READY$ and $STOP$ buttons simultaneously after the machine generates a GO_ON signal. In such a case, the machine rings a bell (*emit RING_BELL*) to indicate that the $READY$ button is pressed wrongly (it should only be pressed before each time the user wants to start a reflex time measurement). At the same time, to handle the $STOP$ button, the machine calculates and displays the average

reflex time, generates a GO_OFF signal to turn off the light, exits from the current measurement and enters the next measurement (or finishes after three runs). Now, in the Esterel specification, we find the user annotation that the input signals $READY$ and $STOP$ cannot happen within the same tick (line 7). Hence our reported critical path is not a feasible one. Using the mechanism discussed in this section, such user annotations are automatically converted to (branch, branch) conflicting pair information, i.e., tests on $READY$ and $STOP$ cannot both be true. Naturally, this yields a tighter WCET estimate.

4.7 Summary

We have presented our model-driven timing analysis framework to compute the single tick WCET of an Esterel specification in this chapter. It is an important design process for model based design using synchronous models. Our analysis obtains tight WCET estimates by exploiting model-level information in the code-level WCET analysis. Our framework also provides timing feedbacks to the system designer for performance debugging and WCET refinement via a bi-directional traceability link between high-level model and generated executable code. By analyzing the results from WCET analysis, potential performance bottlenecks are identified and better resource dimensioning can be achieved. We showed the results of our WCET analysis on a set of standard Esterel benchmarks and illustrated the utility of our model-code traceability technique using an Esterel specification of a reflex game application.

Chapter 5

Context-sensitive Timing Analysis of Esterel

The WCET analysis technique presented in Chapter 4 finds a safe upper bound on the worst-case computation time within any single Esterel tick. In the real execution, different ticks may have different execution time. Furthermore, previously executed ticks may influence the execution time of current tick (e.g., by changing the micro-architecture states). In this chapter, we extend our proposed model-driven timing analysis framework to consider the program flow and micro-architecture contexts across ticks. We show our context-sensitive timing analysis yields tighter estimation when the response time of particular input event(s), whose computation spans across multiple ticks, is of concern.

5.1 Overview

In a real application modeled in synchronous language, it is common for the response of an event to span across multiple clock ticks. Consider an event that takes n ticks between its arrival and completion, and e_{tick} is our estimated WCET for any single tick (as in Chapter 4). A safe worst-case estimation of its processing time is $n \times e_{tick}$. Clearly, this leads to an overestimation because the execution path corresponding to e_{tick} is typically not exercised during all the n consecutive ticks. In this chapter, we address this issue by ruling out – using *program flow context* information – certain program paths that are not executed during specific ticks. As a result, we obtain WCETs, $e_1, \ldots, e_n$, for each of the individual n clock ticks and estimate the processing time of the event to be $e_1 + \ldots + e_n$ (instead of $n \times e_{tick}$). Further, we capture the *micro-architecture context* (e.g., possible instruction cache states) at the start of each clock tick and use this while estimating the WCET associated with this tick. Clearly, this requires us to model how the cache states evolve from one tick to the next, the details of which are presented later in this chapter.

Figure 5.1 shows our context-sensitive timing analysis framework for worst-case response time (WCRT) calculation of events whose processing spans multiple clock ticks. Central to our approach is the use of a finite state automata (described in Section 5.2) to capture the execution context of each clock tick. To accurately estimate the worst-case response time between the arrival of an input event *IN* and its corresponding output *OUT*, our first step is to estimate the WCET of each tick between *IN* and *OUT*. Our tick transition automata (TTA) captures program as well as micro-architecture contexts

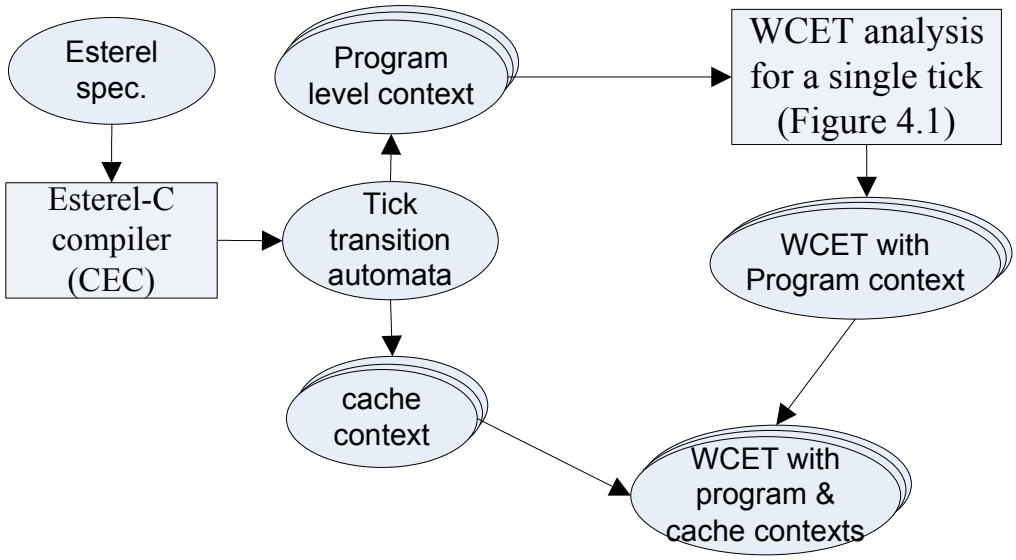


Figure 5.1: Context-sensitive timing analysis framework

for each tick execution, which can be utilized in the timing analysis to provide a tight estimate for each specific tick execution.

5.2 Tick Transition Automata

The Esterel language is finite state in nature, that is, a finite-state automaton can capture the behavior of an Esterel program. The full automaton corresponding to an Esterel program has many uses, such as in compilation and/or program property verification [20]. However, the combinatorial explosion in the number of states of the full automaton is well-known. Instead, for our response-time calculation, we construct a smaller automaton called the tick transition automaton (TTA for convenience, but not to be confused with Time triggered architectures). The states of this automaton capture only the global control flow of an Esterel program — data variable values do not appear in the states.

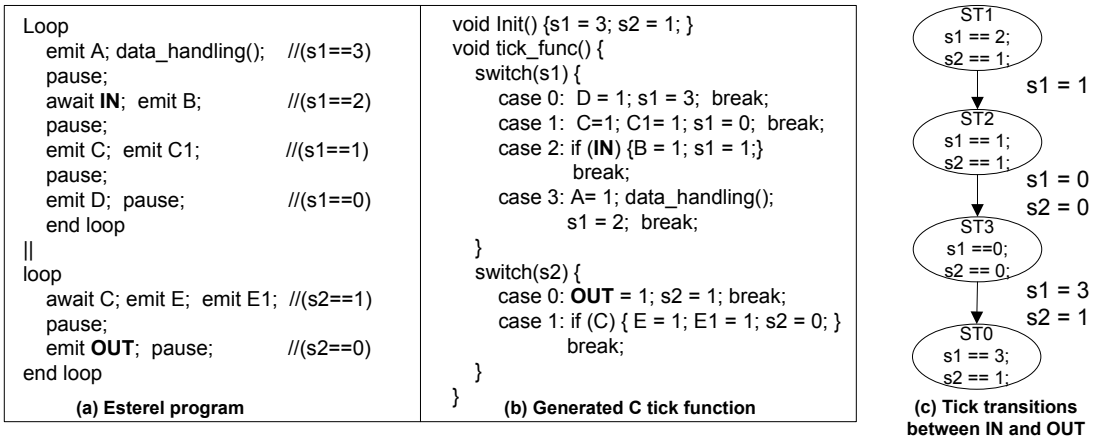


Figure 5.2: An Esterel program, compiled C tick function, and tick transitions

To give a detailed explanation of the Tick Transition Automata, we dwell a bit on the Esterel-to-C compilation process that has been briefly discussed in Section 2.1.1. Typically such a compilation generates a tick function which captures all possible Esterel executions in a single tick. In other words, the Esterel program execution corresponds to repeated execution of the tick function. Naturally, the tick function needs auxiliary variables to capture the progress in control flow in each process (or thread) of the Esterel program. For each thread T_i, a state variable s_i is introduced. Different values of a state variable s_i correspond to the different ticks that thread T_i could execute. Figure 5.2(a) shows a toy Esterel program which will be used as an illustrative example in this chapter. It consists of two concurrently running threads. The input event IN is consumed by the first thread in the second logical tick, and the corresponding output event OUT is emitted by the second tread in the fourth tick. Thus, it takes three logical ticks to produce the output. The compiled C code is shown in Figure 5.2(b).

5.2.1 Formal Definition

The states of a tick transition automaton correspond to valuations of the s_i variables. A transition in the tick transition automata corresponds to assignment of one or more s_i variables. In the example shown in Figure 5.2, two state variables $s1$ and $s2$ are introduced to encode the tick transition information of the Esterel program. The states of the tick transition automaton correspond to the valuations of $[s1, s2]$. We call a valuation of the s_i variables as a *global control state* since it captures the progress in control flow of all the threads of an Esterel program. The individual s_i variables will be called *control state variables* (or *state variables* for short).

Formally, a tick transition automaton identifies all the paths in the Esterel program that can be executed between an input event *IN* and its output *OUT*. It can be defined as a 5-tuple $\langle Q, \Sigma, \delta, Q_0, F \rangle$, where

- Q is the set of all TTA states. A TTA state is a global control state capturing the progress in control flow (tick transitions) in all the concurrent processes of an Esterel specification.

- Σ is a finite set of symbols, where each symbol represents a value assignment on one or more state variables.

- δ is the transition function, such that $\delta : Q \times \Sigma \rightarrow Q$. Each transition in the automata represents an execution of a tick in the Esterel program. In our TTA, we need only label the modifications of state variables (assignments) in the corresponding tick.

- Q_0, F are the set of initial/final states of the TTA. An initial state is a global control state of the Esterel program where the input signal *IN* is ready to be consumed. A final state is one where the output signal *OUT* is produced.

Figure 5.2(c) shows the tick transitions between the input event *IN* and its output *OUT* for the example Esterel program. In between the initial state $ST1$ and final state $ST0$, there is only one possible execution path which consists of three ticks.

TTA is succinct. Note that a Tick Transition Automata captures only the paths in an Esterel program between a given input event *IN*, and output event *OUT*. Moreover, for each such path only the global control flow in maintained — values of data/signals in the Esterel program do not appear. Hence the Tick Transition Automata is typically several orders of magnitude smaller than the full automata for a given Esterel program.

In general, the tick transition automata is much smaller than a full Esterel automata used in automata-based code generation or program verification, because

- The number of state variables used to defined the TTA (one per a thread) is normally much less than the number of signals and program variables used.

- One control state in TTA may represent a set of data states. For example, one Esterel program variable may have many different values at a single control location (finish of a tick).

- The TTA is built only for part of the entire program, between a particular input event *IN* and its output *OUT*. Furthermore, in a big Esterel program consisting

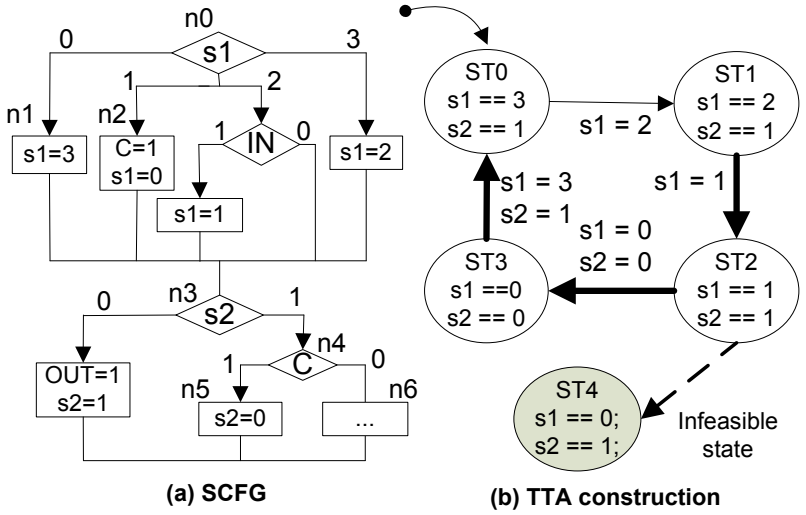


Figure 5.3: SCFG and TTA construction for the program in Figure 5.2

of many concurrent processes, only a subset of the processes are involved in processing IN. The TTA's state space Q and transition relations are built on only the state variables for this subset of processes.

5.2.2 Construction of TTA

The tick transition automaton is built by analyzing the state variables' evaluation across different iterations of tick function execution. The analysis can be done at various levels of program representations during the Esterel-C-assembly compilation. In our implementation, we construct TTA by analyzing the sequential control flow graph (SCFG) produced by the Columbia Esterel Compiler (CEC). As an example, the SCFG of the Esterel program in Figure 5.2(a) is shown in Figure 5.3(b).

Prior to building TTA from the SCFG, an infeasible path analysis (refer to Section 4.3) is performed on the SCFG to eliminate infeasible paths resulting from both the

Esterel program itself and Esterel-to-C compilation (Section 4.3). Infeasible paths are taken into account while constructing the TTA — we do not construct edges in the TTA which correspond to infeasible control flows in the SCFG generated from Esterel.

If the investigated input event is tested in a feasible path of the tick function, the current state is set to be a possible initial state of the TTA. Similarly, if the output event is present in a feasible path of the tick function which corresponds to a tick transition from state ST_i to ST_j, ST_j is marked as a final state of the TTA.

Construction of TTA at SCFG-level is similar to finding all feasible control states in the SCFG as discussed in Algorithm 1 in Chapter 4. In our example Esterel program SCFG shown in Figure 5.3, the initial state is $ST0$ where $s1$ and $s2$ equal to 3 and 1, respectively. From a feasible state, the TTA construction traverses all feasible paths to find next possible combinations of state variables' values. Assume the current analyzing state is $ST2$ ($[s1 == 1, s2 == 1]$), since the path $(n0, n2, n3, n4, n6)$ is infeasible (due to conflict between assignment $C = 1$ in $n2$ and branches $n4 \rightarrow n6$), we can not reach a state where $[s1 == 0, s2 == 1]$ (labeled as $ST4$ in Figure 5.3(b).

By taking the conflicting pair information into consideration, infeasible states such as $ST4$ will not be visited. The ticks executed between input IN and its output OUT in the example Esterel program are highlighted with bold arrows in Figure 5.3(b). The grey edge $ST0 \rightarrow ST4$ corresponds to an infeasible control flow in the C code generated from Esterel. We detect and leave out such edges while constructing the TTA. Clearly, any infeasible path detection method is sound but incomplete, so we may in principle fail to detect certain infeasible paths/states.

5.3 Inter-tick Control Flow Context

Given our automatically constructed TTA, we want to estimate the WCET of a tick $ST_i \rightarrow ST_j$ (where ST_i and ST_j are states of the TTA) by considering the program states in which the tick is executed. The program-level context with which $ST_i \rightarrow ST_j$ is executed is captured by ST_i. We describe how such context information can be integrated into our WCET estimation for a single tick.

As mentioned in Section 5.1, we can assume that each tick between input *IN* and output *OUT* takes time e_{tick}, the maximum WCET obtained for any single tick execution (Chapter 4). However, this clearly leads to a gross over-estimation. To accurately estimate the worst-case response time between *IN* and *OUT*, our first step is to accurately estimate the WCET of each individual tick between *IN* and *OUT*. Thus, we accurately estimate the WCET of each transition in the TTA. This is achieved by automatically generating additional constraints for each specific transition, and integrating them with the tick function's WCET constraints to build a new ILP formulation for a particular tick. Solving the tick-specific ILP will produce the accurate WCET estimate of the tick in question.

We now explain how the additional ILP constraints for a specific tick transition $ST_i \rightarrow ST_j$ are generated. The key difficulty here is that the ILP constraints refer to occurrences of code fragments in the generated C code – they *do not* refer to occurrences of specific ticks at the Esterel program level. Hence, constraints resulting from the occurrence of a specific tick $ST_i \rightarrow ST_j$ need to be expressed in terms of *occurrences of nodes in the Sequential Control flow graph (SCFG) of the code generated from Esterel.*

Recall that ST_i and ST_j correspond to valuations of control state variables $s_1, \ldots, s_n$ where n is the number of threads in the Esterel program. Let $BR_{x,v}$ be the set of all branch edges $e : n1 \rightarrow n2$ in SCFG that are taken when $x == v$. Now, assume the value of a state variable s_k in state ST_i be v. The following set of ILP constraints are introduced

$$\{E_{i \rightarrow j} = 0 | (n_i \rightarrow n_j) \in BR_{s_k, v_x} \wedge v_x \neq v\}$$

where $E_{i \rightarrow j}$ is the number of times control flows through the SCFG edge $n_i \rightarrow n_j$. These path constraints ensure that the tick execution that corresponds to $ST_i \rightarrow ST_j$ takes only the SCFG path where "$s_k == v$" whenever state variable s_k is tested, for each state variable s_k in the TTA state ST_i.

Moreover, there can be multiple outgoing tick transitions from a TTA state ST_i. Suppose the control state variable s_k is assigned to a new value v' ("$v \neq v'$") in the tick transition $ST_i \rightarrow ST_j$. To calculate the WCET for a particular tick from $ST_i \rightarrow ST_j$, we also introduce ILP constraints to ensure that state variable s_k is assigned to v' during the tick execution. Let $AS_{x,v}$ be the set of all nodes n in SCFG that contain assignment of x to value v, we have

$$\sum_{n_i \in AS_{s_k, v'}} N_i > 0$$

where N_i is the execution count of a node n_i. In other words, at least one of the assignment $s_k = v'$ must be executed (for s_k's value to be v' in state ST_j).

Referring back to our example in Figure 5.2 and 5.3, to compute the WCET for tick transition $ST1 \rightarrow ST2$, execution counts of the outgoing edges $\{0, 1, 3\}$ from the SCFG node that tests $s1$ and outgoing edge $\{0\}$ from the node that tests $s2$ are set to

0; execution count of the node that contains assignment "$s1 = 1$" is set to 1. Similar to our SCFG-level infeasible path constraints in Section 4.4, these SCFG-level program context constraints can be translated to corresponding basic block-level ILP constraints by maintaining an Esterel-C code association during Esterel compilation.

5.4 Inter-tick Micro-architectural Contexts

We now show how the micro-architectural state at which a given tick is executed can also be taken into account into the WCET estimation. Note that state-of-art WCET analyzers take into consideration of the underlying processor micro-architecture for an accurate WCET calculation. Thus, even the vanilla WCET analysis in Chapter 4 will consider the intra-tick micro-architectural states (such as cache states) while estimating the WCET of a single tick. However, for tight WCET estimation of a given tick, we also need to consider the inter-tick micro-architectural states — the micro-architectural state at the beginning of a tick's execution. In the following, we study how such inter-tick micro-architectural states can be captured in the WCET estimation of a given tick. We consider one particular micro-architectural feature namely the instruction cache. We note that a very similar modeling can be used to capture the inter-tick timing effects of data cache.

The tick transition automata (TTA) defines the sequencing of tick transitions from the consumption of input signal *IN* to production of output signal *OUT*. In the framework shown in Figure 4.1, the WCET of each tick t is estimated independently, assum-

ing that the tick t starts execution in a micro-architectural state which will result in the worst case timing behavior for t. However, execution of previous ticks may load certain instructions of t into the instruction cache, and the execution of t may re-use these "pre-loaded" instructions in the cache, leading to cache hits. Thus, by considering the cache state resulted from previously executed tick transitions, a tighter WCET of t can be obtained.

In our example shown in Figure 5.2, even though there is no instruction reuse across different ticks at the Esterel program level, same instructions of the generated C level tick function may be reused in different ticks. Since the execution of different Esterel ticks is accomplished by several executions of the C-level tick function, this results in non-trivial instruction reuse across ticks. For example, in the two consecutive tick transitions $ST1 \rightarrow ST2$ and $ST2 \rightarrow ST3$, although different paths of the C level tick function are executed, they use the same instructions for testing $s1$ ("switch(s1)"), $s2$ ("switch(s1)"), and signal C ("if (C)"). In a processor containing instruction cache, execution of these common instructions in the subsequent tick transition (i.e., $ST2 \rightarrow ST3$) will result in cache hits, and hence a smaller WCET.

Timing effects of cache sharing by different program fragments have been well-studied in the literature on cache-related preemption delay [40]. Given a preempted task T and a preempting task T', the cache-related preemption delay is an upper bound on the delay due to additional cache misses caused by preemption of T by T'. Our problem is somewhat different. Instead of computing cache states at each possible preemption point, we only need to compute the cache states at the beginning and end of the tick

function. Moreover, we want to estimate the cache reuse (*i.e.* gain in execution time) due to prior execution of other program fragments. In comparison, the works on CRPD analysis estimate the cache pollution (*i.e.* loss in execution time) due to prior execution of other program fragments.

Let a (instruction) cache state be the mapping between instructions and cache blocks at a certain program point. Using the terminology of [84], we can say that we compute two sets of cache states for each tick transition $ST_i \rightarrow ST_j$.

1. $RCS(ST_i \rightarrow ST_j)$. The ***Reaching Cache States*** is the set of possible cache states when the tick function finishes executing the tick $ST_i \rightarrow ST_j$.

2. $LCS(ST_i \rightarrow ST_j)$. The ***Live Cache States*** is the set of the possible first memory references to cache blocks during the execution of the tick $ST_i \rightarrow ST_j$.

The RCS and LCS of a particular tick transition $ST_i \rightarrow ST_j$ are computed via program path analysis of the code corresponding to $ST_i \rightarrow ST_j$. If the code contains loops, such a computation is iterative and is guaranteed to terminate by converging to a least fixed point. Furthermore, the RCS of a state ST_i in a tick transition automata T is defined as:

$$RCS(ST_i) = \cup\{RCS(ST_k \rightarrow ST_i) | (ST_k \rightarrow ST_i) \in T\}$$

The guaranteed cache reuse (cache hits) due to inter-tick cache behavior in the execution of a tick transition $ST_i \rightarrow ST_j$ is now summarized as follows. We define:

$$reuse(ST_i \rightarrow ST_j) = min\{match(R, L) | \\ R \in RCS(ST_i), L \in LCS(ST_i \rightarrow ST_j)\}$$

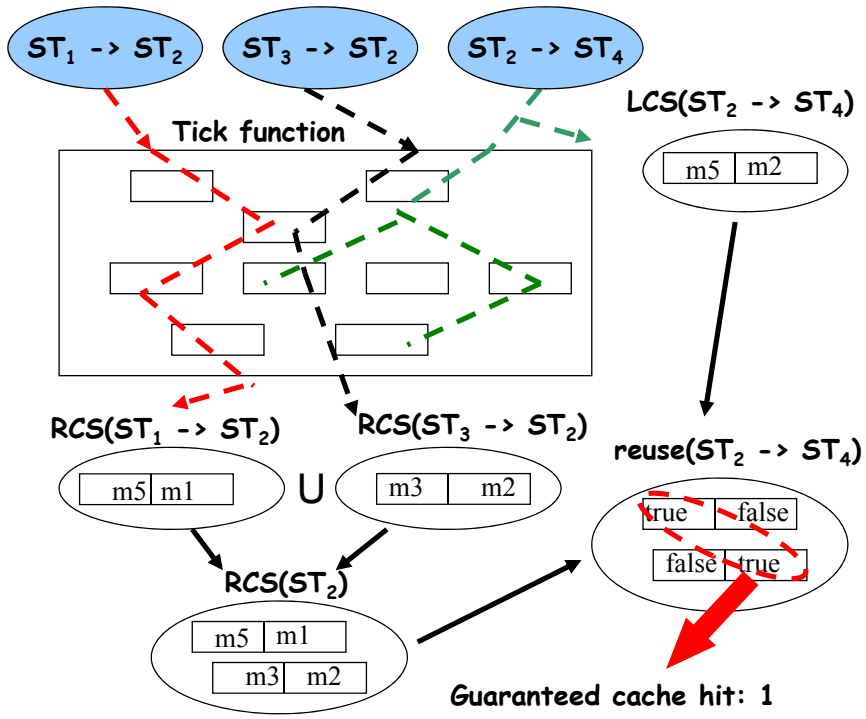


Figure 5.4: Example of inter-tick cache reuse analysis

where $match(R, L)$ returns the number of cache blocks with same contents in cache states R and L.

Finally, the WCET of tick transition $ST_i \rightarrow ST_j$ can takes into account the captured inter-tick cache behavior. Assume the processor architecture has no timing anomalies [93], the overestimated cache misses for each tick can be simply removed from the WCET estimate as follows.

$$wcet'_{i,j} = wcet_{i,j} - reuse(ST_i \rightarrow ST_j) \times penalty \tag{5.1}$$

where $wcet_{i,j}$ is the WCET value for the tick execution corresponding to tick transition $ST_i \rightarrow ST_j$ without inter-tick cache modeling (calculated as in Section 5.3), and $penalty$ is the time penalty for a cache miss.

On the other hand, for processor architecture with possible timing anomalies, the calculated cache states in $RCS(ST_i)$ can be used as initial cache states when calculating WCET for a tick $ST_i \rightarrow ST_j$. By restricting the possible initial cache states, tighter WCET estimation can be obtained.

Figure 5.4 shows an example of inter-tick cache reuse analysis for WCET calculation of tick transition $ST_2 \rightarrow ST_4$. Assume ST_2 is reachable only from two other states ST_1 and ST_3. Possible cache states before execution of $ST_2 \rightarrow ST_4$ is captured in $RCS(ST_2)$. By checking all possible combinations between $RCS(ST_2)$ and $LCS(ST_2 \rightarrow ST_4)$, we can guaranteed at least 1 additional cache reuse, compared to the context-insensitive WCET analysis where execution of $ST_2 \rightarrow ST_4$ are assumed to start with an empty cache state (for processor architecture without timing anomalies).

5.5 WCRT Estimation

A TTA $\mathcal{T}$ captures all execution paths between the consumption of a given input *IN* and the production of an output *OUT*. Given a TTA and tight WCET values for tick transitions in it, we now need to compute the worst case response time (WCRT) between an input signal *IN* and output signal *OUT*.

Since the execution count of each tick transition is an integer, we employ an Integer Linear Programming (ILP) approach to compute the WCRT. We solve the following ILP optimization problem. This problem uses the WCET values of the individual ticks as constants.

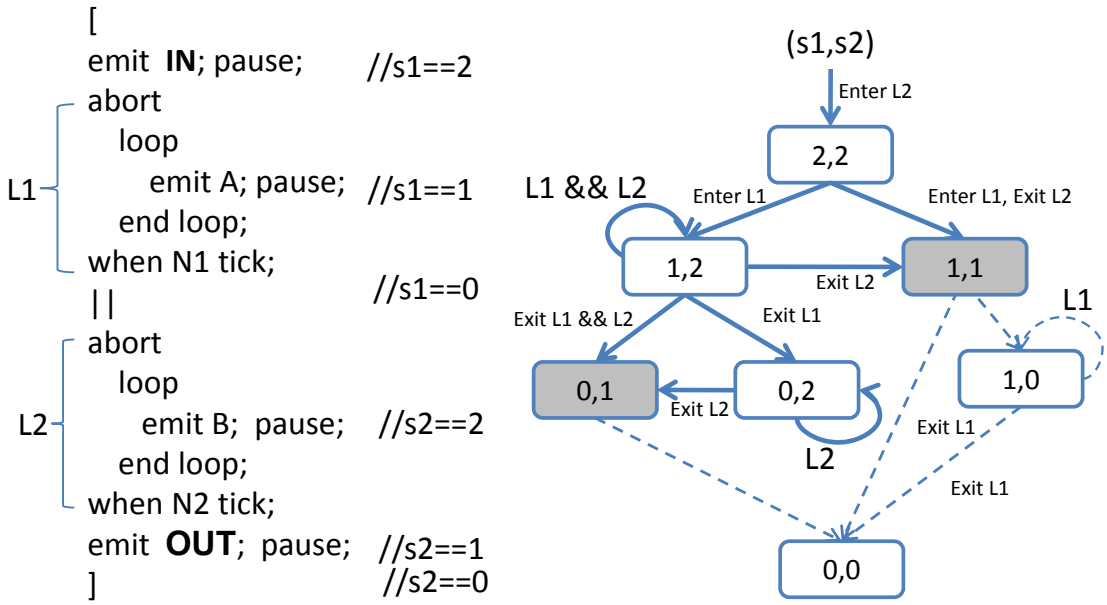


Figure 5.5: An Esterel program containing loops and its TTA

$$\text{maximize} \sum_{ST_i \to ST_j \in \mathcal{T}} Cnt_{i,j} \times wcet'_{i,j}$$

where $Cnt_{i,j}$ and $wcet'_{i,j}$ are the execution count and WCET (see Equation 5.1) of tick transition $ST_i \to ST_j$ in TTA $\mathcal{T}$.

The linear constraints on $Cnt_{i,j}$ are developed from the TTA's control flow. Since we are calculating the WCRT between an input signal and its output, only one of the initial transitions is allowed to take place. This is captured by the constraints

$$\sum_{ST_i \to ST_j \in \mathcal{T} \wedge ST_i \in Q_0} Cnt_{i,j} = 1$$

where Q_0 is the set of initial states of TTA $\mathcal{T}$. Furthermore, for each state ST_i in the TTA, the aggregate execution counts of all incoming tick transitions should be equal to

that of the outgoing transitions. Thus,

$$\sum_{ST_j \to ST_i \in \mathcal{T}} Cnt_{j,i} - \sum_{ST_i \to ST_k \in \mathcal{T}} Cnt_{i,k} = 0$$

Over and above the constraints given in the preceding, we need to bound the number of iterations of every cycle in the TTA. This is a difficult task. The Esterel program may contain loops in one or more processes in the computation between the input signal *IN* and output *OUT*. Even if we know the loop bounds of these Esterel level loops, we cannot directly use them to bound the cycles of the TTA. While constructing the TTA from the Esterel program, an Esterel level loop is often partially unrolled, or fused with other loops. The execution counts for ticks inside a loop depend on the loop bound. The loop bound can be automatically detected for some simple cases, otherwise it needs to be provided by the programmer. When constructing the global states in the tick transition automata, the repeated states in one process (as defined by the loop) is often partially unrolled or combined with the control states of other concurrent processes.

Figure 5.5 shows an example Esterel program that has two processes each containing a single-tick loop, bounded by $N1$ and $N2$ respectively. In the constructed TTA, different paths exist between the initial state (2,2) and the final states (0,1) and (1,1) (in gray boxes), and there is no one-to-one correspondence between the Esterel level loops and TTA cycles. For example in state (1,2), it goes back to itself in a cycle if both loops keep iterating; or it may go to state (0,2) if loop $L1$ in the first process exits before $L2$ in the other process. In general, the execution count of a cycle in TTA may be different from the bound of corresponding loop(s) in Esterel program, which makes it complex to design a path enumeration based algorithm to compute the end-to-end delay. How-

ever, we show that it is much easier to generate ILP constraints on aggregate execution counts of certain tick transitions to satisfy the given Esterel-level loop bound.

We bound the number of executions of each TTA cycle as follows. Recall that each state in the TTA is a valuation of control state variables $s_1, \ldots, s_n$ – each variable s_i corresponds to a thread or process in the Esterel program. Now, for each loop L in the Esterel program, we first find the control state variable s_k that captures the control flow of the process p containing L. Since the loop defines repeated execution of certain local control states of the process p, and the variable s_k simply encodes the progress in control flow of p – we can always find a value v of s_k that appears exactly once in each iteration of the loop L. Such a value v corresponds to a control state of p lying inside the loop L. We now generate the following ILP constraints to incorporate the loop bound B_L for each loop L in the Esterel program

$$\sum_{ST_i \rightarrow ST_j \in \mathcal{T} \wedge (s_k == v) \in ST_i} Cnt_{i,j} <= B_L$$

where s_k is the control state variable for the process containing loop L, v is a value of s_k that holds once in each iteration of L.

5.6 Case Study

We illustrate our response time estimation using the TURBOchannel Interface (TcInt) benchmark from the Estbench Esterel Benchmark Suite [33], which contains 687 lines in Esterel source file and 3031 lines of compiled C code. We adopt the same architecture configuration as presented in Section 4.6, including a direct mapped L1 instruction

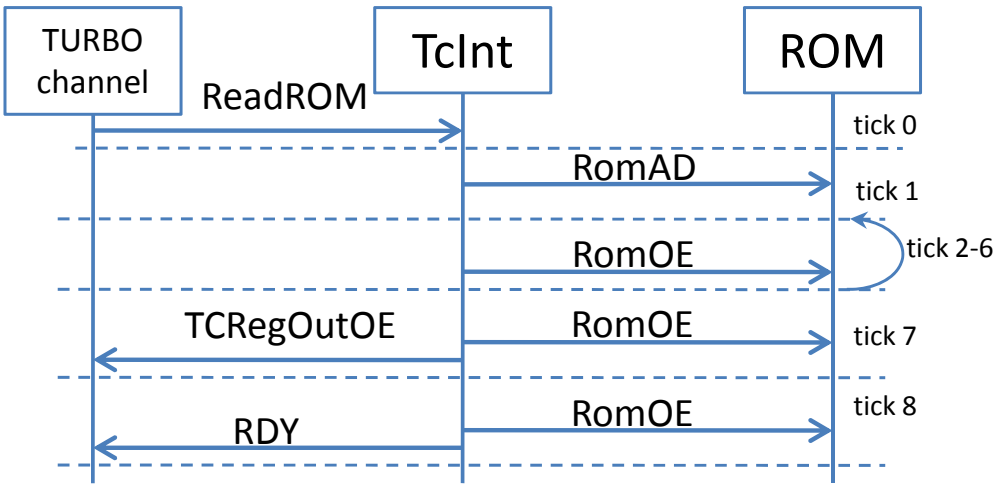


Figure 5.6: ROM read operation in TURBOchannel interface program

cache with 8-byte block size, LRU replacement policy, and 30 cycles miss penalty. We assume there is no data cache. We also assume the processor architecture has no timing anomaly effects.

TURBOchannel is an I/O interconnect that allows several I/O options to connect to one system [85]. The TURBO channel Interface helps guarantee correct TURBOchannel implementation and provides additional features such as DMA operations. In the following, we show the response time calculation of a ReadROM operation. Figure 5.6 shows the event communications between TcInt program and the environment during each tick in the ReadROM operation. There are in total 9 ticks between the input event "ReadRom" and the final output "RDY", which informs the TURBOchannel that the operation is completed. A single-tick loop iterates five times between tick 2 to 6, which corresponds to the ROM read delay period. We do not show all internal signals used for communicating between processes within the TcInt program (e.g., RROM, DRIVER,

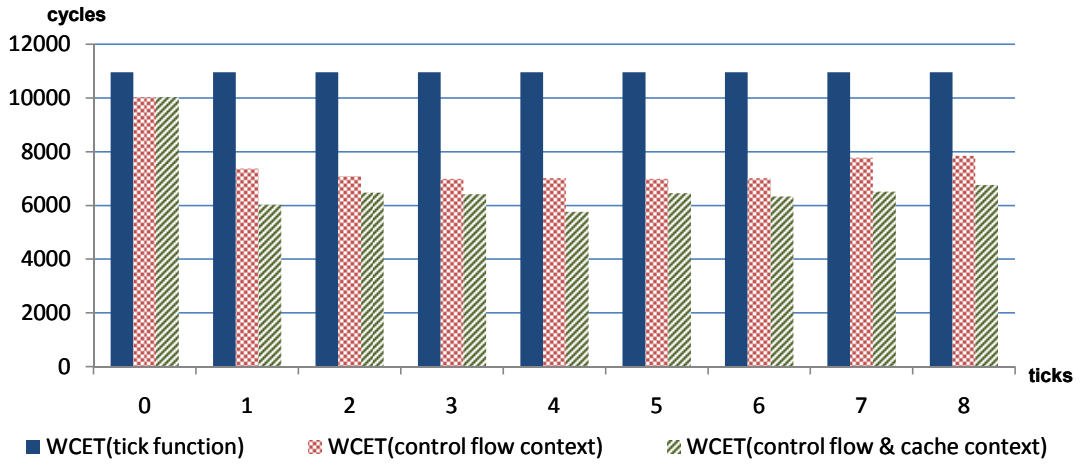


Figure 5.7: Tick WCET results from different calculation approaches

and DELAY processes) in the figure.

The tick transition automaton (TTA) corresponding to the ROM read operation contains 45 states and 91 transitions. On the other hand, a full automaton consists of around 3000 states and 20000 transitions – clearly showing the succinctness of the TTA.

The WCET of the generated tick function is 10949 cycles without considering any tick path information from TTA. Since the operation takes maximum 9 ticks to finish, a pessimistic estimation of the response time is $9 \times 10949 = 98541$ clock cycles. One the other hand, by utilizing program path contexts, the calculated response time is 68103 clock cycles, which gives a 30.9% reduction over the previous result. Furthermore, after modeling the inter-tick cache behavior to capture the additional guaranteed cache hits across tick function iterations, we obtain an even tighter response time of 60753 clock cycles. This amounts to a reduction of $(98541 - 60753)/98541$, that is, almost 40% reduction by taking into account program control flow as well as micro-architectural contexts. Finally, Figure 5.7 compares the WCET values calculated for the individual

ticks which appear in the longest path of the Tick Transition Automata (TTA) — ticks responsible for the worst-case response time of a ReadROM operation. For each tick, (i) the leftmost bar shows the WCET value without any context information, (ii) the middle bar shows the WCET value considering only program level contexts and (iii) the rightmost bar shows the WCET value considering program level as well as micro-architectural contexts.

5.7 Summary

In this chapter, we have shown a context-sensitive timing analysis for Esterel programs. This is useful for tightly estimating the response time of input events. We consider the program flow and micro-architecture contexts at the beginning of each Esterel clock tick to deliver tight response time estimates. Such tighter estimates immediately translate into more cost-effective implementations. Our experimental results with a realistic case study show up to 40% reduction in timing estimates when context information is taken into account.

Chapter 6

Multiprocessor Execution of Esterel

In this chapter, we further extend our model-driven timing analysis of Esterel specification for multiprocessor execution. Towards this goal, we propose a scheme for generating efficient code from Esterel specifications for a multiprocessor platform, followed by timing analysis of the generated code. Due to dependencies across program fragments mapped onto different processors, traditional WCET analysis techniques for sequential programs cannot be applied to this setting. Our Worst-Case Response Time (WCRT) analysis technique is tailored to capture such inter-processor dependencies. Our main novelty stems from how we detect and remove infeasible paths arising from a multiprocessor implementation, along with a shared bus modeling in order to obtain tight estimates on the WCRT. Furthermore, we integrate a shared bus modeling for simple round-robbin TDMA bus schedule into our timing analysis framework to capture the inter-processor architectural behavior. We illustrate our techniques using a number of standard Esterel benchmarks, which show that ignoring inter-processor bus and infea-

sible path modeling in a multiprocessor setup may lead to up to 133% over-estimation of the WCRT thereby leading to resource over-dimensioning and poor design.

6.1 Overview

Worst case response time (WCRT) analysis for concurrent Esterel execution is proposed in [19]. The targeted platform is a special-purpose single processor (the Kiel Esterel Processor) with hardware support to schedule concurrent Esterel threads. Techniques that compile synchronous languages for distributed execution are summarized in [47]. In [114], the distributed executable code generated from Esterel specification targets special-purpose reactive processors, where a hardware scheduler is designed for concurrent Esterel threads running on the same processor. Moreover, the issue of timing analysis is not discussed for the generated distributed program.

In this chapter we propose a scheme for generating C code from Esterel specifications for general-purpose multiprocessor platforms. Our platform architecture is fairly general and consists of multiple processors each with a private L1 cache; they communicate via shared memory and are connected to a shared communication bus supporting time division multiple access (TDMA) (see Figure 6.1). Given a multi-threaded Esterel program, we assume that the thread to processor mapping is given. Communication between threads mapped onto different processing elements (PEs) is implemented via shared memory objects. The shared bus is used only for loading instruction/data from shared main memory to individual PEs (not for explicit message exchanges between

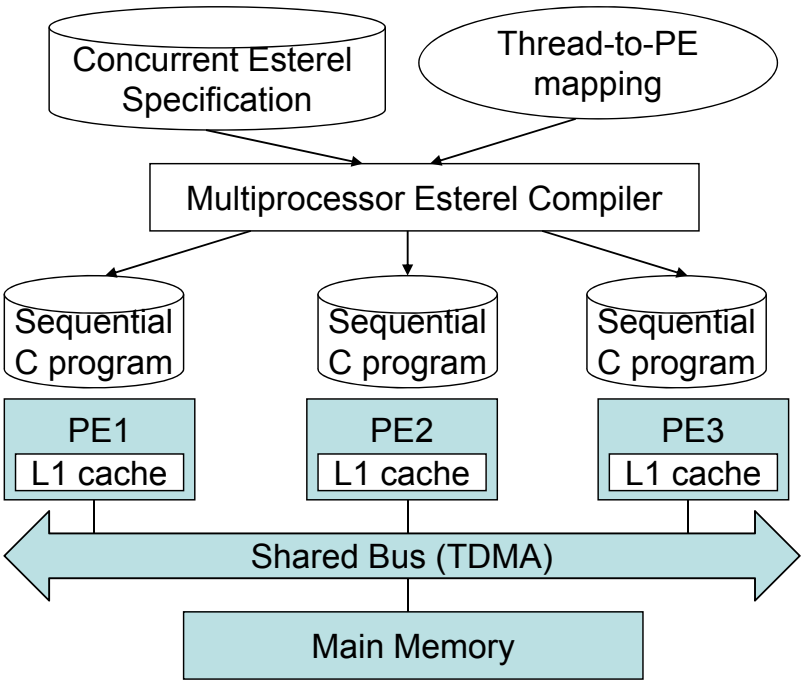
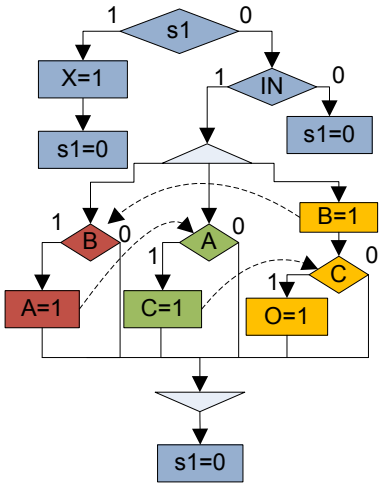


Figure 6.1: Multiprocessor execution of Esterel Specification.

PEs). For multiple threads mapped onto the same processor, a valid sequentialization – as followed by Esterel compilers for uni-processors – is assumed (refer to Section 2.1.1). The resulting sequential code for each processor needs to be augmented with partial code replication, as well as additional communication dependencies from other processors to resolve control flow decisions. We discuss our code generation technique in Section 6.2, and propose a timing analysis framework that captures the inter-processor program flow and architectural behaviors in Section 6.3.

```
loop
   X=1;                          //T1
   await IN;
   [
   present B then emit A;
   end present                   //T2
   ||
   present A then emit C;
   end present                   //T3
   ||
   emit B;
   present C then emit O;
   end present                   //T4
   ]
   pause;
end loop
```

(a) Example Esterel program



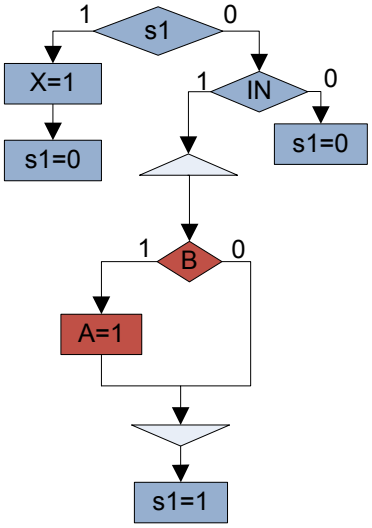
(b) Global CCFG

Figure 6.2: Example Esterel specification and it concurrent control flow graph (CCFG).

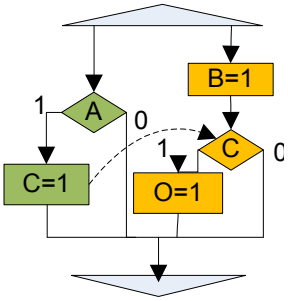
6.2 Code Generation

Our multiprocessor Esterel code generation is built on top of the Columbia Esterel Compiler (CEC), which produce single-thread C program for uniprocessor execution (Section 2.1.1). We present our code generation technique using the example Esterel specification shown in Figure 6.2(a). We assume that threads $T1$ (the root thread) and $T2$ are mapped onto PE1, and threads $T3$ and $T4$ are mapped onto PE2. The corresponding concurrent control flow graph (CCFG) generated by CEC is (partially) given in Figure 6.2(b). To preserve Esterel semantics on instantaneous signal broadcasting, a signal test is executed only after the signal emit in the same clock tick in the generated C code.

In a multiprocessor execution environment, compiling partitioned Esterel threads *individually* for each PE does not produce a correct implementation, due to the inter-processor control and communication dependencies. In our example, simply partition-



(a) CCFG1 on PE1



(b) CCFG2 on PE2

```
L1:  If (s1) {
L2:      X = 1;
L3:      s1 = 0;
L4:  }
L5:  else {
L6:      if (IN) {
L7:          if (B)  A = 1;
L8:          s1 = 1;
L9:      }
L10:     else {
L11:         s1 = 0;
L12:     }
L13: }
```

(c) C code on PE1

```
L1:  if (A) C = 1;
L2:  B = 1;
L3:  if (C) O = 1;
L4:  s1 = 1;
```

(d) C code on PE2

Figure 6.3: Incorrect multiprocessor code generation.

ing the global CCFG into two CCFGs (each containing the corresponding threads' statements mapped onto a PE), results in the wrong implementation of CCFGs and C programs shown in Figure 6.3. In particular, the following issues must be corrected in the multiprocessor code generation.

- Control dependencies. For example, in the generated multiprocessor C tick functions in Figure 6.3(c) (for PE1) and Figure 6.3(d) (for PE2), it is possible for the code fragments $X = 1$ (line L2) and $if(A)\ldots$ (line L1) to be executed in a same global clock tick. However, it is not allowed in the original Esterel specification. In order for the generated C programs to execute on multiprocessors in globally synchronized locksteps, control dependencies between threads mapped onto different PEs (captured by the use of state variables) must be retained locally in the C program on each PE.

- Communication dependencies. The generated C programs do *not* ensure that the test of signal A on PE2 (line L1) must be executed after emit of A on PE1 (line L7). Since the signal A is represented as a shared memory object in our setting, it is possible that the test of A returns false while A is emitted (set to 1) later in the same clock tick.

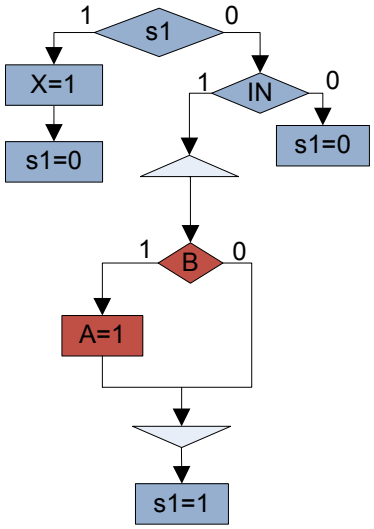
- Even if the communication dependencies are ensured between emit and test of a signal across different processors (i.e., test of a signal can be executed only after possible emit of the signal in a same clock tick), the generated two programs may still result in deadlock during runtime. In particular, test of signal A on PE2 (line

L1) is waiting for emit of A on PE1 (line L7), while the test of B on PE1 (line L7) is waiting for emit of B on PE2 (line L2). Neither program is allowed to proceed.

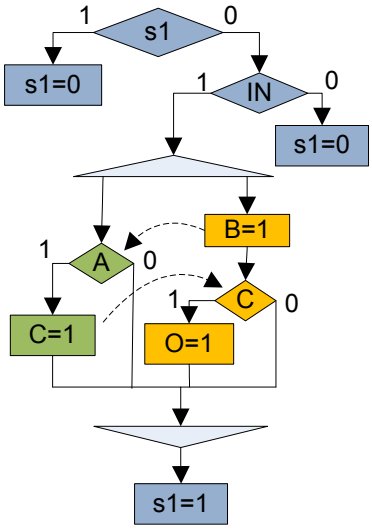
In our code generation, we extend the methodologies in [114] for handling such dependencies (refer to Section 6.2.1 and Section 6.2.2). In [114], one sequential program is generated for each concurrent thread in the Esterel specification. For threads mapped onto each PE, the corresponding programs are executed in a round-robin fashion by a scheduler. On the other hand, we produce one program for all threads mapped onto each PE by statically sequentializing their execution (Section 6.2.3). Thus, we do not require any hardware supported scheduler to ensure the correctness of our multiprocessor Esterel execution. Furthermore, as we will discuss in this section (and also show in the experimental results refer Section 6.4), our code generation produces efficient code with less overhead for handling control and communication dependencies across different threads. Finally, to resolve the possible inter-processor deadlock during execution, we propose a technique that takes into account the global communication dependencies when performing sequentialization for threads mapped onto the same PE (Section 6.2.3). The correct partitioned CCFGs and generated C programs of our example Esterel program is shown in Figure 6.4.

6.2.1 Replicating Control-flow

In order to execute a thread on a different PE from its parent threads, the control-flow context of its parents (up to the root thread) needs to be replicated. In particular, the creation and execution of a child thread may be affected by the control states of its



(a) CCFG1 on PE1



(b) CCFG2 on PE2

```
L1:  If (s1) {
L2:      X = 1;
L3:      s1 = 0;
L4:  }
L5:  else {
L6:      if (IN) {
L7:          /*guard B*/
L8:          while(!(RN_B[2]);
L9:          if (B)  A = 1;
L10:         /*resolve A*/
L11:         RN_A[1] = 1;
L12:         s1 = 1;
L13:     }
L14:     else {
L15:         /*resolve A*/
L16:         RN_A[1] = 1;
L17:         s1 = 0;
L18:     }
L19: }
L20: barrierSyn();
```

(c) C code on PE1

```
L1:    If (s1) {
L2:        s1 = 0;
L3:    }
L4:    else {
L5:        if (IN) {
L6:            B = 1;
L7:            /*resolve B*/
L8:            RN_B[2] = 1;
L9:            /*guard A*/
L10:           while(!(RN_A[1]);
L11:           if (A) C = 1;
L12:           if (C) O = 1;
L13:           s1 = 1;
L14:       }
L15:       else {
L16:           /*resolve B*/
L17:           RN_B[2] = 1;
L18:           s1 = 0;
L10:       }
L20:   }
L21:   barrierSyn();
```

(d) C code on PE2

Figure 6.4: Correct multiprocessor code generation.

parent threads. In [114], the following two types of nodes (in the compiler intermediate representations) are distinguished when Esterel is compiled into C code,

Control nodes: Nodes affect the control flow, including conditional test nodes (for signals and normal data variables), and Esterel specific control nodes. (for manipulation of compiler-introduced control state variables, signal guard, fork/join, terminate/preemption, etc).

Assignment nodes: Nodes that contain signal emission and normal data handling (they correspond to the assignments that are visible to other PEs via updating on shared memory objects in the generated C code).

In order to produce a correct multiprocessor execution of Esterel, for each thread's implementation, all its parent threads' control nodes must be replicated. In Figure 6.4(d), the control nodes of $T1$ are replicated in CCFG for PE2, including test and assignment on control variable $s1$ and test of signal IN. Note that "X=1" is an assignment node in $T1$, thus no replication is needed. As a result, the generated programs on all PEs execute in globally synchronized locksteps.

In [114], each thread is compiled into a separate program. Thus, parent threads' control nodes have to be always replicated in all child threads' code regardless of the thread-to-PE mapping. As a result, [114] replicates $T1$'s control nodes three times for each of its child threads in our example. On the other hand, our code generation requires to replicate a thread's control nodes at most once on each processor. In particular, a thread T's control nodes are replicated on processor PE_i only if T is not mapped to PE_i and at least one child thread of T is mapped to PE_i.

6.2.2 Handling Signal Communication

Esterel requires that in a single clock tick, the test of a signal must be executed after the signal emission (if any). In our multiprocessor code generation, we further classify the following two scenarios of signal communication between concurrent threads.

Local within a PE: If all threads that emit and test a signal are mapped onto a same PE, the compiler statically schedules the execution order between them (refer to Section 6.2.3). No inter-processor signal communication is required.

Across different PEs: Otherwise, we adopt the method of inserting *signal resolution/guard nodes* proposed by [114] to guarantee the correct sequentialization of signal communication across PEs.

We insert resolution/guard nodes at the sequential control flow graph (SCFG) level during code generation of each PE. Let RN_A^i be set of resolution nodes for signal A inserted in SCFG for PE_i. Execution of RN_A^i is used to notify other PEs that PE_i has determined the value of signal A (either presence or absence) in its current clock tick. In general, a resolution node for presence of signal A is inserted immediately after emit of A in the SCFG (e.g., L11 in Figure 6.4(c)); or in case signal A's emission is absent in a path, the resolution node for A is inserted at the earliest possible control location from which no emission of A is reachable (e.g., L16 in Figure 6.4(c)). Correspondingly, a signal guard node for A is inserted *before* each test of A (e.g., L10 in Figure 6.4(d)). The guard node allows test of A to take place only if for each PE_i that may potential affect the value of A, a resolution node in RN_A^i has been executed. In our current implementation, this is done via a busy-waiting `while` loop testing all the shared memory

objects of signal resolutions. In the correct multiprocessor C programs shown in Figure 6.4(c) and Figure 6.4(d), resolution/guard nodes for signal A and B are inserted.

Note that we classify signal communication at PE level, such that resolution/guard nodes are not inserted for local communication within a PE (e.g., in Figure 6.4(d), no resolution/guard nodes required for signal C on PE2). Our multiprocessor Esterel execution encounters less communication overhead compared to [114], where resolution/guard nodes are required for any *inter-thread* signal communication.

Global Synchronization In our model of multiprocessor execution, an Esterel clock tick completes when all concurrent programs finish execution of current tick. Barrier synchronization is performed at the end of each generated C tick function, such that the current execution of a tick function completes only if all programs reach the barrier. In Figure 6.4(c) and Figure 6.4(d), barrier synchronization $barrierSyn()$ is invoked at L20 an L21, respectively. The barrier synchronization can be implemented similarly to shared signal handling as described above, by introducing a shared "sync" signal in all generated programs. Only when all programs emit the "sync" signal, each program can proceed to execute next clock tick.

6.2.3 Sequentializing Concurrent Threads

In order to generate one single sequential C program for all threads mapped on a PE, concurrent execution of these threads needs to be sequentialized based on signal communication dependencies. To ensure that Esterel semantics is followed in the generated

code, a test on signal A can be checked only after the decision on emit of A has been made (possibly in other threads). Figure 6.2(b) shows (part of) the concurrent control-flow graph (CCFG) produced by [34] for the example program, where the communication dependencies are denoted with dashed directed edges.

In our code generation, we build one local CCFG for all threads executed on each PE (e.g., Figure 6.4(a) and Figure 6.4(b)). However, we cannot simply compute the communication dependency on each local CCFG *individually*. In our example, "present A" in thread $T3$ depends on "emit A" in thread $T2$, and "present B" in thread $T2$ depends on "emit B" in thread $T4$. We assume threads $T1$ (the root thread) and $T2$ are mapped onto PE1, while threads $T3$ and $T4$ are mapped onto PE2. When generating a sequential C program for threads $T3$ and $T4$ to be executed on PE2, if only the local dependencies between $T3$ and $T4$ are considered (between emission and test of signal C as shown in the incorrect implementation in Figure 6.3(b)), the compiler may decide to schedule the test of A in $T3$ prior to the emit of B in $T4$. The resulted incorrect sequentialization (shown in Figure 6.3(d)) leads to a deadlock situation (circular wait) between the emit/test of A and B signals across the two processors PE1, PE2 as we have previously discussed.

We solve this inter-processor dependency problem by adding an *indirect* dependency between emit of B and test of A in the CCFG for PE2 (as shown in Figure 6.4(b)). The compiler ensures that "emit B" will be executed before "present A" in the generated sequential C code. Indirect dependency can be deduced from the global CCFG that contains all signal dependencies between threads in the Esterel program (as

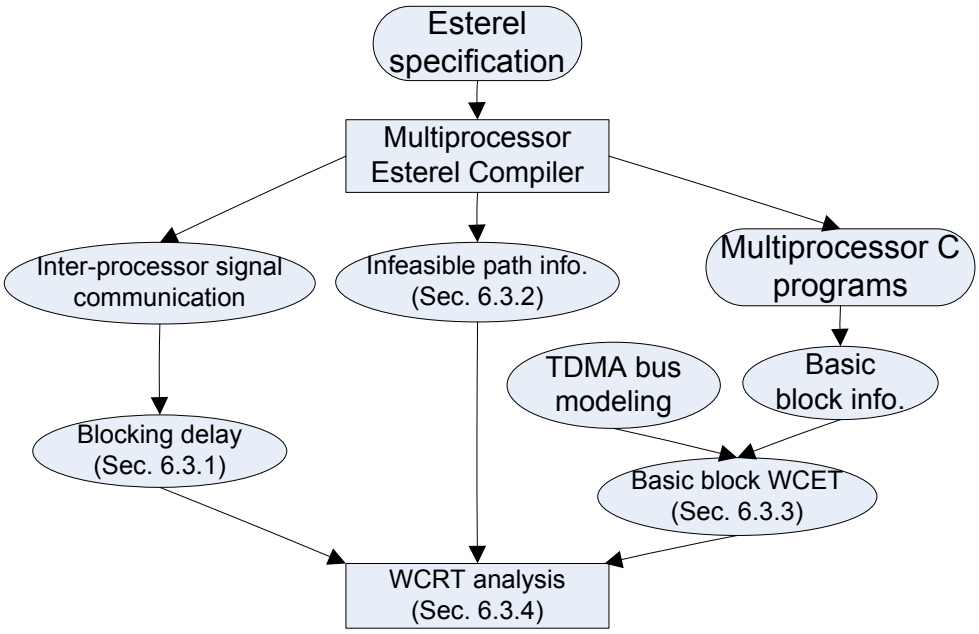


Figure 6.5: Overview of timing analysis framework for multiprocessor execution of Esterel.

shown in Figure 6.2(b)).

6.3 Timing Analysis

Figure 6.5 presents an overview of our proposed timing analysis framework. During the multiprocessor Esterel compilation, we generate information on inter-processor signal communication as well as infeasible paths. In our work, we assume signal communications are implemented via shared memory objects among generated multiprocessor programs (no explicit message passing across processors). As a result, a signal guard node blocks test on a signal to be executed until the corresponding signal resolution nodes finish execution. Such blocking time must be considered during the timing analysis (Section 6.3.1). Furthermore, by utilizing infeasible path information, our timing

analysis ignores a subset of infeasible paths and control states in the generated programs, resulting in tighter timing estimates (Section 6.3.2).

We compute the WCET of each basic block in the assembly code level control flow graphs (CFGs) of the generated C programs. Tighter WCET estimates can be obtained by micro-architectural modeling of the execution platform. In particular, we integrate a shared bus modeling for TDMA based bus schedule (Section 6.3.3).

The compiler generated information on infeasible paths and signal communication can be propagated into assembly code level CFGs by maintaining a mapping between the CFG basic blocks and various intermediate representations (IRs) produced during Esterel compilation (as discussed in Chapter 4). Finally, our timing analysis works on assembly code level CFGs of the generated multiprocessor programs (Section 6.3.4). By utilizing infeasible path information, communication blocking time, and WCET of basic blocks, it determines WCRT of the generated programs in any single Esterel clock tick.

6.3.1 Computing Start Times

Recall that in our multiprocessor execution model of Esterel specification, one single sequential C program is generated and executed on each PE. However, we cannot directly apply traditional WCET analysis for sequential programs to estimate the execution time of each individual program due to the dependencies across programs on different PE. In particular, the signal guard nodes introduced to guarantee the correctness of communication dependency act as blocking delay during programs' execution.

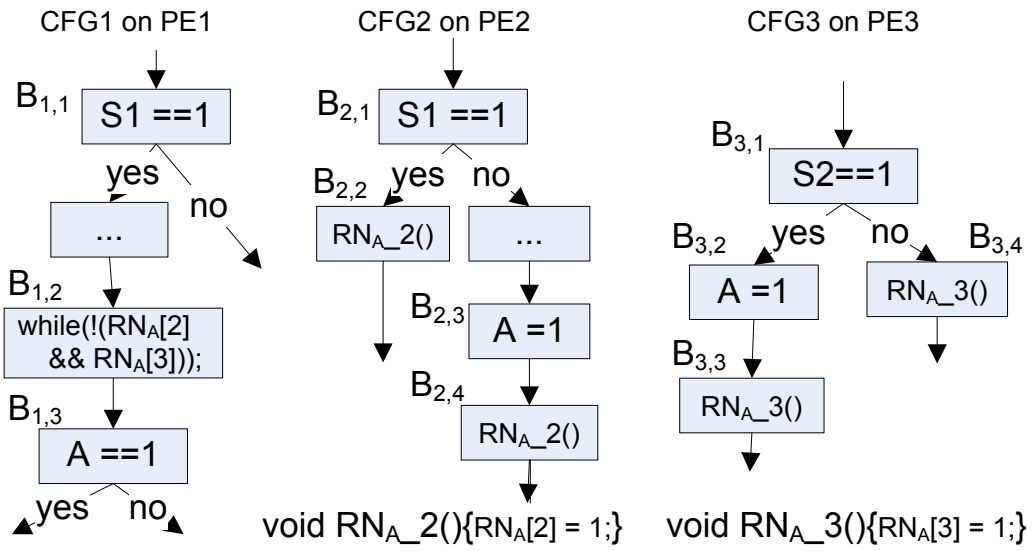


Figure 6.6: Blocking delay due to signal communication.

To compute WCRT of the generated parallel program in any Esterel clock tick, we need to incorporate the blocking delay time within the program's execution time.

Consider a system consisting of three processors, each having a generated sequential program to be executed on. Figure 6.6 shows part of the three program's assembly code level CFGs, related to communication of a shared signal A. In $CFG1$, basic block $B_{1,2}$ is the signal guard for test of A which appears in $B_{1,3}$). The emit statements for signal A appear in the other two PEs, namely PE2 and PE3; the corresponding signal resolution statements in PE2 and PE3 are $B_{2,2}$, $B_{2,4}$, $B_{3,3}$ and $B_{3,4}$. For example in $CFG2$, if state variable $S1 == 1$, it can be determined that signal A cannot be emitted by the second program in current clock tick. Thus, a resolution node $B_{2,2}$ is inserted immediately after. Otherwise, a resolution node ($B_{2,4}$) is inserted after signal A is emitted ($B_{2,3}$) on the other path. It is clear that in the worst case, the time at which $B_{1,3}$ begins to execute depends on which of the four corresponding resolution nodes completes last.

In our timing analysis, the blocking delay is handled by setting the start time of a signal guard basic block to be the latest finish time among all its control-dependent predecessors (in the same program's CFG) and data-dependent predecessors (corresponding signal resolution basic blocks in other programs' CFGs). Let RN_A be the set of resolution basic blocks of signal A in all CFGs running on all PEs, and $start_{b_i}/finish_{b_i}$ be the start/finish time of basic block b_i in the worst case. We have

$$start_{b_i} = \begin{cases} \max\{\max\{finish_{b_j} | \forall b_j \in Pred(b_i)\}, block_A\} & \\ & \text{if } b_i \text{ is guard of signal A,} \\ \max\{finish_{b_j} | \forall b_j \in Pred(b_i)\} & \text{Otherwise.} \end{cases} \quad (6.1)$$

where $block_A = \max\{finish_{b_k} | \forall b_k \in RN_A\}$ is the blocking delay due to shared signal A, and $Pred(b_i)$ is set of b_i's control-dependent predecessor basic blocks in the same CFG.

6.3.2 Inter-processor Infeasible Paths

Infeasible paths introduce substantial over-estimation in the static timing analysis, especially for generated programs from high-level modeling languages. Intra-processor infeasible path elimination for a single sequential program generated from Esterel specification has been studied in Chapter 4. In this section, we show how to exploit inter-processor infeasible paths to obtain tighter WCRT estimate.

As discussed in Section 6.2.1, if child threads of thread T are mapped onto different PEs, control nodes of T are replicated on those PEs. In any clock tick execution, these replicated control variables must take the same value. In the CFGs shown in Figure 6.6,

control variable $s1$ is replicated in both $CFG1$ and $CFG2$. Thus, $B_{1,2}$ in CFG1 for PE1 (executed when "s1==1") and $B_{2,4}$ in CFG2 for PE2 (executed when "s1==1" is false) are on two conflicting inter-processor paths. To calculate the blocking time for $B_{1,2}$, we only need to check the finish time of the other three resolution blocks, which gives a tighter result on start time of $block_A$ and overall WCRT. Note that such kind of inter-processor infeasibility does not exist in uniprocessor code generation, because $B_{1,2}$ and $B_{2,4}$ appear on two mutually exclusive paths. However, in multiprocessor code generation, the timing analysis might report a global WCRT path containing both $B_{1,2}$ and $B_{2,4}$, if no global infeasible path detection is applied.

Our timing analysis eliminates such inter-processor infeasible paths by constructing all feasible value combinations of the programs control variables $[s_1, \ldots, s_n]$ during compilation. We adopt the methodology of computing feasible control states for uniprocessor Esterel code generation discussed in Algorithm 1, Section 4.3.2. It is clear to see that in our multiprocessor code generation, we do not introduce any new infeasible global control states (value combinations of state variables) via the thread partition and control node replication. For each feasible control state $[v_1, \ldots, v_n]$, we perform WCRT analysis. In the WCRT analysis, we ignore paths containing any conditional branch of the form $s_i \neq v_i$. The final WCRT of the overall multiprocessor program will be the largest WCRT obtained for all the feasible control states. For example in Figure 6.6, if the only feasible global control states are $[s1 == 1, s2 == 1]$ and $[s1 == 0, s2 == 0]$, our WCRT analysis needs to consider only two path combinations (instead of all the eight possible path combinations across the three CFGs).

6.3.3 WCET Calculation of a Basic Block

The tightness of the proposed high-level timing analysis depends also on the accuracy of calculating WCET for each basic block, which requires low-level modeling of micro-architectural features. Micro-architecture modeling for timing analysis has been studied for single task uniprocessor execution (works on WCET analysis). In our work, we assume there is only private L1 instruction cache in each PE. For each L1 cache miss, main memory is accessed via the shared bus. We adopt existing instruction cache modeling for uniprocessor architecture to determine the L1 cache hit/miss for each instruction access [109]. In the case of a shared bus is used for main memory access, a bus modeling is required for accurate timing analysis. Otherwise, penalty for every L1 cache miss has to incorporate the worst case bus delay for a safe analysis. In order to compute WCET of each basic block for a multiprocessor architecture, we adopt a round-robin TDMA bus modeling proposed in [27] and integrate it into our framework. Compared with other existing shared bus modeling techniques, for example [97], [27] requires no virtual loop unrolling, which leads to a light-weight timing analysis.

Let L be the slot length assigned to each processor, and $B = L \times n$ be the TDMA schedule period for a system of n PEs. Algorithm 2 describes how to compute the WCET of a basic block b_i, by keeping track of its start time $start_{b_i}$ and latest bus slot $slot_{b_i}$ available to it. The computed value $wcet_{b_i}$ is the estimated WCET of the basic block b_i. For each instruction in b_i, if the instruction is a L1 cache hit, L1 hit latency (HIT_{L1}) and the instruction's execution cost ($cost_{inst}$) is added to current $wcet_{b_i}$. Otherwise, we compute the distance (Δ) between previous instruction's finish time and the

Algorithm 2 computeWCET[b_i,$start_{b_i}$,$slot_{b_i}$]

1: $wcet_{b_i} := 0$;
2: $inst :=$ first instruction in b_i;
3: **while** $inst \neq NULL$ **do**
4: **if** $inst$ hits in L1 cache **then**
5: $wcet_{b_i} = wcet_{b_i} + HIT_{L1} + cost_{inst}$;
6: **else**
7: $\Delta = (slot_{b_i} + L) - (start_{b_i} + wcet_{b_i})$;
8: **if** $(\Delta >= LAT)$ **then**
9: /*$inst$ can be loaded within current bus slot*/
10: $wcet_{b_i} = wcet_{b_i} + LAT + cost_{inst}$;
11: **else**
12: $wcet_{b_i} = wcet_{b_i} + \Delta + (n-1) \times L + LAT + cost_{inst}$;
13: **end if**
14: **end if**
15: $inst :=$ next instruction in b_i
16: **end while**
17: return $wcet_{b_i}$;

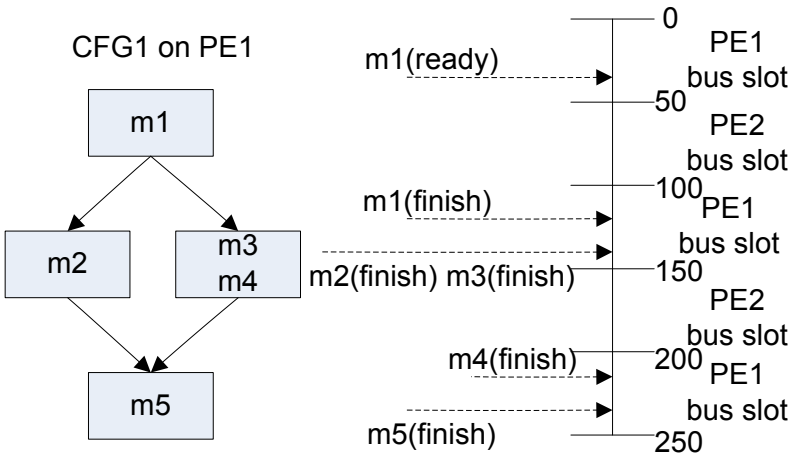


Figure 6.7: Shared TDMA bus modeling.

end of current bus slot. If Δ is no less than the main memory access latency (LAT), the instruction can be loaded within current bus slot. $wcet_{b_i}$ after execution of this instruction is computed at Line 10. Otherwise, the instruction can be loaded only at next bus slot available. Thus, it needs to wait for $(\Delta + (n-1) \times L)$ cycles for a TDMA-scheduled bus shared between n PEs (line 12).

Figure 6.7 shows an example of memory access behavior of a code fragment on PE1 via a TDMA bus shared by two PEs. The bus slot length is $L = 50$ cycles, and the main memory latency is $LAT = 20$ cycles. For the illustration, we assume all 5 cache accesses $m1, \ldots, m5$ encounter L1 cache misses, and we ignore instruction execution time. In the worst case, the ready time of $m0$ is at cycle $L - LAT + 1 = 31$ (the access is ready when bus is available, but the access can not finish in the remaining bus slot). The starting time of $m0$ is at cycle 100 (at the next time the bus is available to PE1). If the right branch is taken, $m4$ can not be loaded within the second bus slot for PE1. Thus, it will wait till next assigned bus slot (start at cycle 200). Finally, $m5$ can be guaranteed

to complete within the third bus slot for PE2 in the worst case (at cycle 240). Thus, the total time elapsed between $m0$ gets ready and $m5$ finishes is $240 - 31 = 209$ cycles. On the other hand, without the bus modeling, the analysis assumes each miss with the worst case delay (which corresponds to the delay of $m1$ in our example). For $m1$, the ready time is at 31, start time is at 100 and finish time is at 120 — so the delay between ready and finish time is $120 - 31 = 89$ cycles. Thus, the WCET estimate of the example program fragment without bus modeling is $4 \times 89 = 356$ cycles (at most four accesses on any program path).

6.3.4 WCRT Analysis

Our timing analysis framework is described in Algorithm 3. We first compute all feasible control states of the generated multiprocessor programs in order to remove inter-processor infeasible paths as discussed in Section 6.3.2. For each feasible control state, basic blocks from all the programs are visited in topological order (line 4). Note that for a signal guard node, corresponding signal resolution nodes are considered as its predecessors and must be visited before the signal guard node. Esterel language prohibits the use of loops within a clock tick (except for loops in external procedure calls). Thus, the traversal in line 4 of Algorithm 3 is acyclic, where each basic block will be analyzed at most once.

We set certain blocks (paths) to be infeasible according to the current control state values in each analysis iteration (line 8-14). For a test block b_i on control variable s_k, its successor block b_j is set to be infeasible if the conditional branch $b_i \rightarrow b_j$ is taken when

Algorithm 3 WCRT analysis for multiprocessor Esterel execution.

1: $WCRT = 0$;
2: getGlobalFeasibleState(); /*Section 6.3.2*/
3: **for** each feasible control state $\langle v_1, \ldots, v_n \rangle$ **do**
4: **for** all basic block b_i of all programs P in topological order **do**
5: **if** $reachable_{b_j} == false$ **then**
6: $continue$;
7: **end if**
8: **if** b_i is a test node on state variable s_k **then**
9: **for** $\forall b_j \in Succ(b_i)$ **do**
10: **if** branch $b_i \rightarrow b_j$ is taken when $s_k \neq v_k$ **then**
11: $reachable_{b_j} := false$;
12: **end if**
13: **end for**
14: **end if**
15: **if** b_i is a source node of any program **then**
16: /*Assume start at the next available bus slot for worst case*/
17: $start_{b_i} := B$; $slot_{b_i} := B$;
18: $reachable_{b_i} := true$;
19: **else**
20: $reachable_{b_i} := \vee\{reachable_{b_j} | b_j \in Pred(b_i)\}$;
21: **if** $reachable_{b_i} == false$ /*Infeasible path, refer Section 6.3.2*/ **then**
22: $continue$; /*no need WCET computation of this basic block*/
23: **end if**
24: $start_{b_i} := computeStartTime(b_i)$; /*Equation 6.1 used here*/
25: $slot_{b_i} := \lfloor \frac{start_{b_i}}{B} \rfloor \times B$;
26: **end if**
27: $finish_{b_i} := start_{b_i} + computeWCET(b_i, start_{b_i}, slot_{b_i})$;
28: **end for**
29: $tmpWCRT := \max\{finish_{b_i} | b_i \text{ is a sink node of any program}\}$;
30: $WCRT = \max\{WCRT, tmpWCRT\}$;
31: **end for**

"$s_k \neq v_k$". If a block is infeasible (not reachable) in current control state, the current analysis iteration ignores it (line 5-7). Furthermore, for a basic block b_i, it is reachable in current control state only if at least one of its predecessors ($Pred(b_i)$) is reachable (line 20). Otherwise, if all predecessors of a basic block b_i is unreachable, we ignore this basic block (and subsequent paths from it) in current analysis iteration (line 21-23).

For any source node b_i in any of the generated sequential programs, since we do not know the exact starting time of b_i, we consider the worst-case scenario by adding the "maximum initial delay" to b_i's start time. The "maximum initial delay" is defined as the distance between b_i gets ready and it acquires the bus slot to execute, which is always less than the bus period B (line 16-18). For any other reachable basic block b_i, we compute its start time as shown in Equation 6.1 in Section 6.3.1 (line 24). The latest bus slot available to b_i is computed at line 25. The finish time of b_i is obtained using our bus-aware WCET calculation as presented in Algorithm 2 (line 27). We perform WCRT analysis to find a local WCRT value for each feasible control state, and the final global WCRT is set to be the maximum among them (line 29-30).

6.4 Experimental Results

We extend the Columbia Esterel Compiler (CEC) for our multiprocessor code generation. Concurrent control flow graph (CCFG) representation of the Esterel program produced by CEC is first duplicated into multiple copies, one for each PE. Based on a given thread-to-PE mapping, for each PE, we remove CCFG nodes/edges that should

Benchmark	# of Esterel lines	# of lines in generated C programs		
		1 PE (CEC)	4 PEs	
			(our approach)	[114]'s approach
abcd	101	827	1190	1255
mejia	555	2598	3464	5231
wristwatch	1088	1755	2560	3494
elevator	324	1241	2340	3272

Table 6.1: Esterel benchmarks and generated C programs.

not be executed on it, except for nodes that need to be replicated (Section 6.2.1). Each CCFG is converted into SCFG by adding signal guard/resolution nodes as described in Section 6.2.2. Additional indirect communication dependencies are then added to ensure correctness of sequentialization (Section 6.2.3). Finally, a sequential C program is dumped from each SCFG, where signals and guard/resolution variables are implemented as shared memory objects.

Table 6.1 lists the benchmarks we used in our experiment, where the "elevator" is taken from Esterel Studio [39] and the other three programs are from Estbench Esterel benchmark suite [33]. For each Esterel program we show Esterel code size and the compiled C programs' total size for uniprocessor (CEC), 4 processors (our approach with a random thread-to-PE mapping), and the approach of [114]. In [114], code size of each generated program (for each concurrent thread) is related to neither number of PEs nor thread-to-PE mapping. In general, code size increases in the multiprocessor

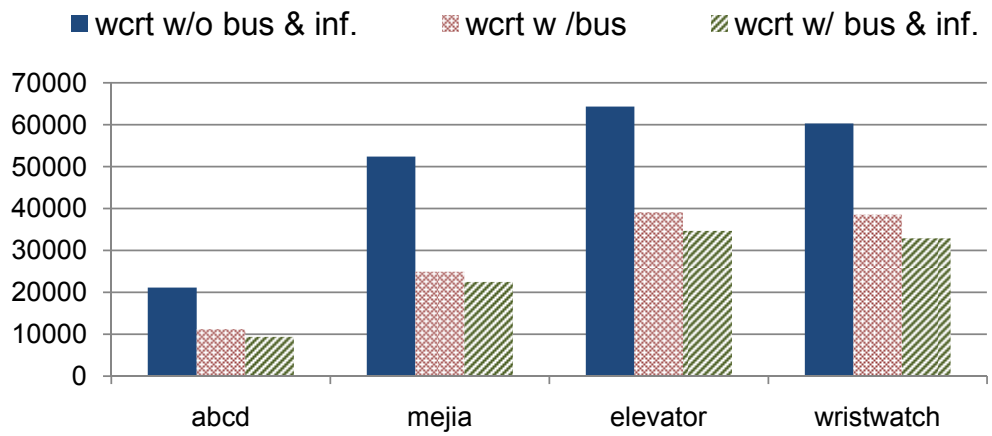


Figure 6.8: WCRT analysis results.

generation due to control-flow node replication and inserted resolution/guard nodes for handling signal communication. Table 6.1 shows that our code generation produces 5.2% to 33.8% smaller code compared to the approach of [114]. Further, our code generation for any of the benchmarks takes *less than 1 second*.

Figure 6.8 shows the results from WCRT analysis of the generated multi-processor code on a platform with 4 processors. We consider a direct mapped L1 instruction cache with 256 cache blocks, where each block's size is 8 bytes. The 4 processors are connected by a shared bus running static bus scheduling (TDMA). We set the TDMA bus slot length assigned to each PE to be $L = 50$ cycles, and memory access latency to be $LAT = 20$ cycles. For each of the benchmarks, we compute: (i) WCRT without bus modeling[1] or infeasible path elimination ("*wcrt w/o bus & inf.*"), (ii) WCRT with bus modeling only ("*wcrt w/ bus*"), and (iii) WCRT with both bus modeling and infeasible path elimination ("*wcrt w/ bus & inf.*").

[1] Assuming each memory access encounters worst case bus delay.

The experimental results show that by employing our shared bus modeling we get a reduction of 36.1% to 52.5% in the WCRT estimate. If we combine bus modeling with infeasible path elimination, we get an overall reduction of 45.4% to 57.1% in the WCRT estimates. In other words, WCRT estimation without bus modeling and infeasible path elimination *would have resulted in* 83.3% *to* 133.2% *overestimation*.

6.5 Summary

In this chapter, we propose a scheme to compile Esterel specification for general-purpose multiprocessor execution. A comprehensive timing analysis framework is presented for WCRT estimation of a single Esterel clock tick in the generated C programs. We capture the blocking delay of signal guard blocks due to inter-processor communication, which is crucial for safe multi-processor timing estimation. Furthermore, we eliminate inter-processor infeasible paths (control states) and model the shared bus behavior (with TDMA-based bus schedule) to produce tighter WCRT results.

Chapter 7

Schedulability Analysis for MSG Model

An embedded system containing only one task can be entirely modeled in a synchronous specification. In previous chapters, we have studied how to perform model-driven WCET analysis on a single task modeled in Esterel, for both single processor and multi-processor execution. However, the globally synchronous model is usually too strict for a complex multi-tasking distributed system. In particular, forcing different (and possibly independent) tasks/applications to execute with a globally synchronized clock is too restrictive (i.e., changing specification of one task may need modifications in many other tasks). Furthermore, the computation and communication time in such systems are inherently asynchronous. To consider a fairly general model for multi-tasking and distributed systems, we adopt the globally asynchronous locally synchronous (GALS) model specification (Section 2.1).

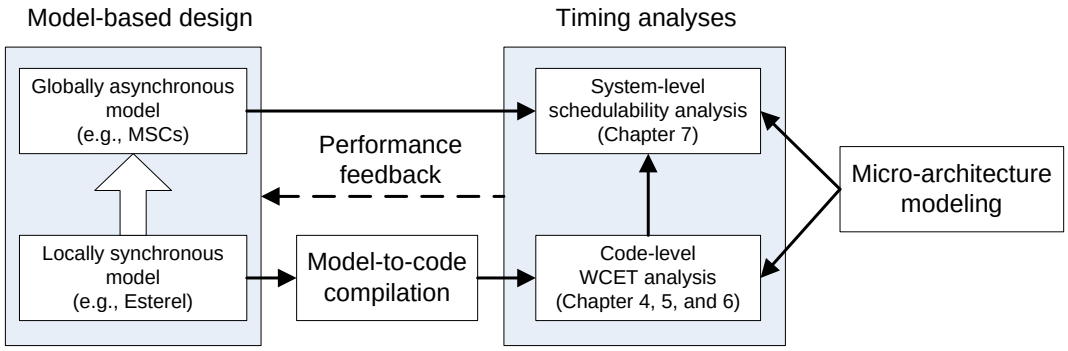


Figure 7.1: Overview of our model-driven timing analysis framework (from Figure 1.1)

Figure 7.1 presents overview of our model-driven timing analysis framework, which is reproduced from Figure 1.1 in Chapter 1. We use MSC graph (MSG) based specification as the globally asynchronous model for describing interaction scenarios between components of a distributed system. In this chapter, we propose a schedulability analysis on an MSG based model, which captures (i) event partial ordering as defined in individual basic MSCs; (ii) asynchronous message communication between events; and (iii) tight preemption cost between events mapped onto a single processing element (PE) across different MSGs (that model different applications). We can assume the local tasks (events) of the MSG specification are designed using synchronous models, e.g., Esterel, from which executable code can be automatically generated. WCET estimates of these local tasks can be obtained via our model-driven timing analysis techniques discussed in previous chapters, and fed into the schedulability analysis as input parameters.

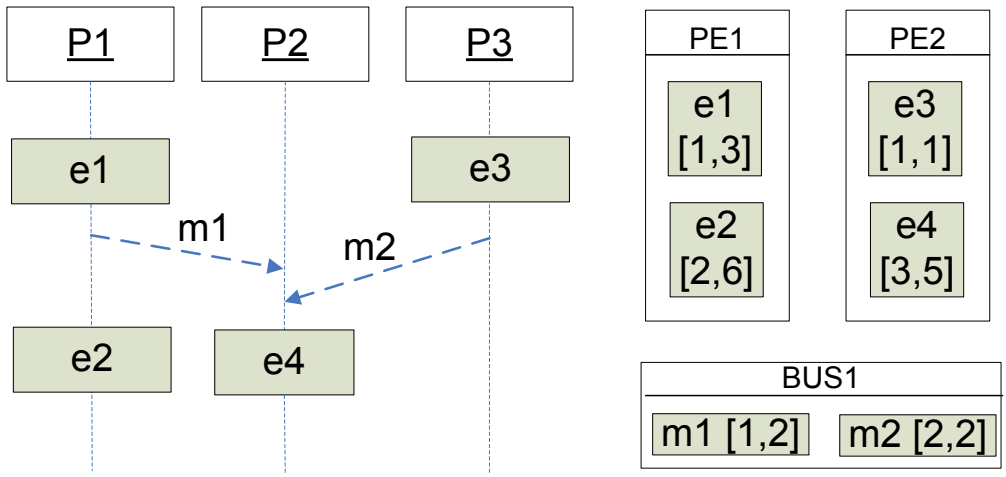


Figure 7.2: A basic MSC and timing annotations

7.1 Overview

Currently there exists a large gap between the quantitative performance analysis techniques that exist in the real-time systems literature, and the modeling/specification techniques that are advocated by the formal methods community. As a result, although a number of schedulability analysis techniques are known for a variety of task graph-based models, it is not clear if they can be used to effectively analyze standard specification formalisms such as MSCs. In this chapter we make an attempt to bridge this gap by proposing a schedulability analysis technique for MSG based global system specifications.

The standard MSC-specific terminologies have been explained in Section 2.1.2. In our setting, a complete system specification consists of a set of MSC graphs (MSGs) denoting concurrently running applications that share common resources, e.g., processing elements (PEs) and shared buses. We extend the system specification by mapping the

different *lifelines* in a basic MSC to different processing elements and their associated *messages* to different communication resources (e.g. buses). Further, we annotate the *events* and the *messages* constituting the different lifelines with lower and upper bounds on their execution/communication times. Figure 7.2 shows a basic MSC that contains 3 processes (lifelines), 4 local events, and 2 messages. For simplicity of the presentation, we do not show the events that correspond to send and receive messages. The messages $m1$ and $m2$ are represented in dashed lines with downward-slope, which indicate the messages are asynchronously transmitted over a shared bus. Event-to-PE and message-to-bus mappings are also shown in the figure, associated with lower and upper bounds on their execution/communication times. Such execution/communication times do not involve blocking times arising out of resource contentions, which is accounted for by our schedulability analysis.

Given the above-mentioned system description, along with the scheduling/arbitration policies at the different resources, our analysis can be used to compute upper bounds on the end-to-end delays associated with various event (and/or message) sequences, which can then be checked against pre-specified deadlines. Examples for such sequences might start with data arriving via a sensor, getting processed on several PEs which also involves multiple transmissions over one or more buses, and then finally ending at an actuator.

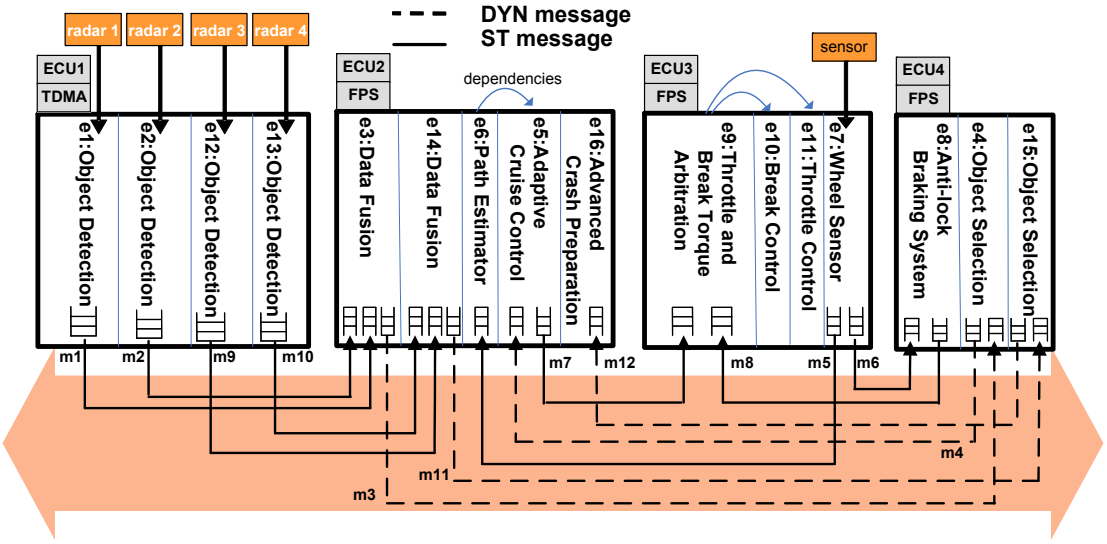
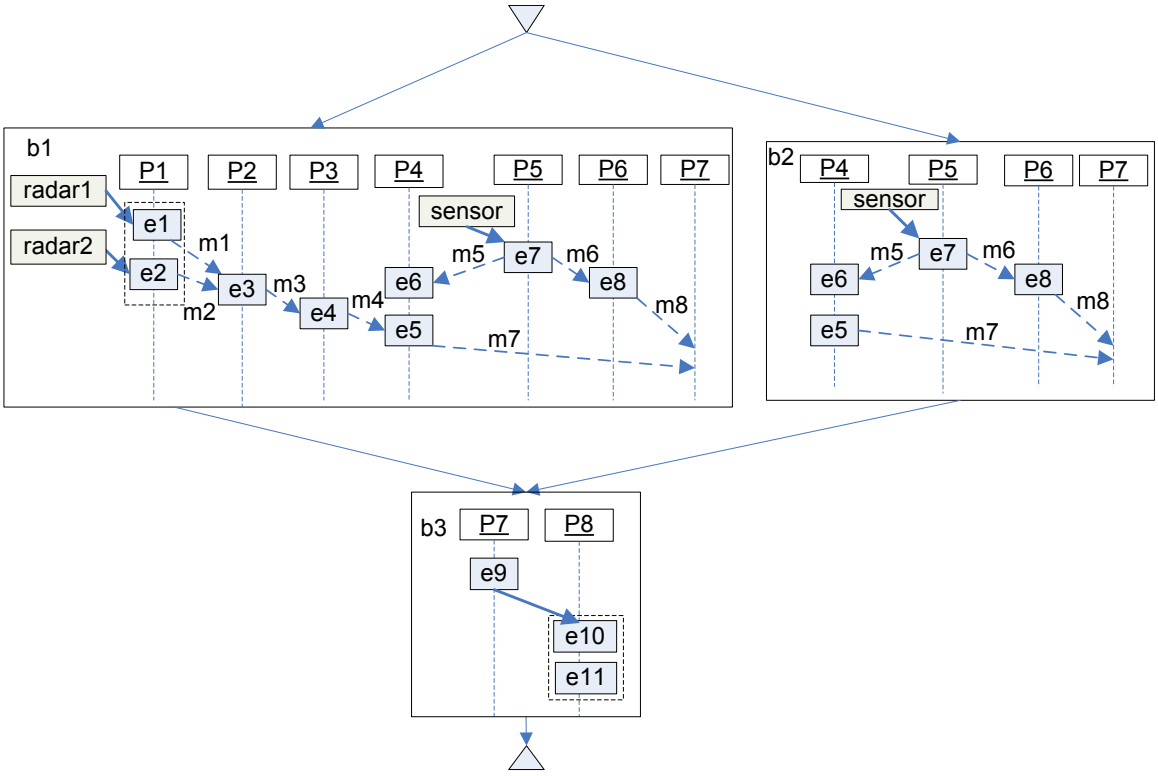


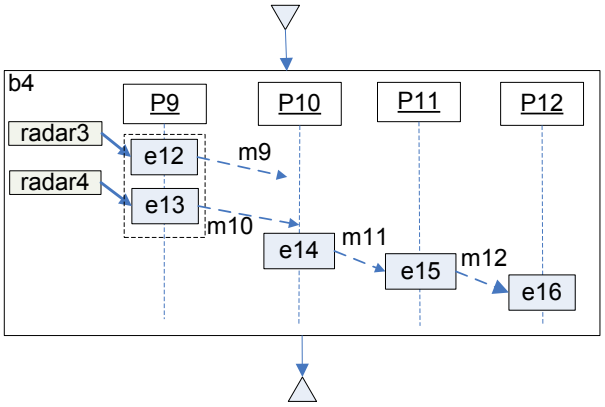
Figure 7.3: A FlexRay-based ECU network.

7.1.1 Running Example

A distributed system has a number of processing elements (PEs) which are connected by shared buses. A typical distributed application consists of a collection of local computations that run on different PEs and communicate with each other through message exchanging via buses. As an example, Figure 7.3 shows a distributed FlexRay [44] based electronic control unit (ECU) network from the automotive electronics domain. There are four PEs (ECUs) and one shared FlexRay bus in the system. Two concurrently running applications, an Adaptive Cruise Controller (ACC) [80] and an Advanced Crash Preparation (ACP) system [31], are shown in the example. The ACC application contains local computations $e1$ to $e11$ and messages $m1$ to $m8$, while the rest belong to the ACP system. Control dependency relations between local computations on the same ECU are also shown, e.g., $e10$ and $e11$ can start only after $e9$ finishes execution.



(a) Adaptive Cruise Control



(b) Advanced Crash Preparation

Figure 7.4: MSG model of the ACC and ACP applications.

Figure 7.4 shows the MSG modeling of the ACC and ACP applications. Each local computation is mapped to one local event on a lifeline (process) in a basic MSC. Note that a lifeline can represent a piece of software program which handles its corresponding event(s), or a hardware functional unit. Thus, the mapping of events onto processes can be easily obtained from the given system specification. Several processes can share a single PE, which implements its own scheduling policy (e.g. the fixed priority preemptive scheduling for processes $P5$, $P7$, and $P8$ on ECU3). It may be noted here that our analysis is flexible enough to handle different scheduling policies, specified at both MSG and PE/bus level. In fact, the example shown in Figure 7.3 has a TDMA scheduling implemented on ECU1, fixed-priority scheduling implemented on the remaining ECUs and a FlexRay protocol implemented on the bus.

Communication between processes in an MSC can be modeled using message passing. Communication may take place via a shared bus (across PEs) or between processes running on the same PE. If the communication is done via a shared bus, we label it with a message name (e.g., $m1$ and $m2$ in MSC $b1$). We will only consider *asynchronous message passing* in our MSG modeling/analysis. Synchronous message passing, where the message sender and message receiver handshake, can be obtained as a special case of our framework. Finally, a "coregion" (denoted by a dashed-line box) is used to relax the strict ordering of local events along a lifeline, e.g. events $e10$ and $e11$ on process $P8$ of MSC $b3$ can be executed in any order (decided by the scheduler of $ECU3$).

In our example, the ACC application has three external triggers, namely radar1, radar2 and sensor. We assume the sensor receives input from environment twice faster

than the two radars. Consequently, in the start-up stage of a complete run of the ACC application, either it receives input data from both two radars and the sensor, which corresponds to the scenario described as in MSC $b1$; or it receives only the sensor's input which triggers the scenario in MSC $b2$, and uses the old output value from the "object selection". The different system behaviors due to environment input are modeled using the indeterministic choice operation from the start of the application.

Given the system architecture and MSG based modeling of applications, our goal is to perform schedulability analysis by checking whether the worst-case response time (WCRT) for each application meets its deadline. Towards this, we first need to extend the standard MSG formalism with real-time annotations. Each MSG depicting an application is associated with the application's activation period P and deadline D. Each event is associated with the best-case and worst-case execution time (BCET/WCET) of its corresponding local computation. We assume the intra-processor communication (e.g., from $e9$ to $e10$ and $e11$ in MSC $b3$ of Figure 7.4(a)), as well as the local events of sending/receiving a message (implicitly denoted by the start/end of a message arrow), take zero time. Messages are labeled with their transmission time, while the actual communication time (including blocking time due to possible bus contentions) will be calculated by our analysis.

7.1.2 Issues in Analyzing the Model

Before proceeding to present our schedulability analysis method, let us examine the inherent difficulties in finding the end-to-end delay of such an MSG model of a dis-

tributed application. In order to obtain a tight analysis, we need to consider the effect of resource contention, event dependencies (e.g., partial order in an MSC, message communication), as well as conditional execution of MSCs in an MSG specification.

The possible contentions and data dependencies bring the *timing anomaly* phenomenon [46] when the execution times of events are not constant. In such cases, the local WCET of an event may not lead to the global worst case end-to-end delay of the application. Thus, the worst-case delay of an application cannot be simply obtained by simulating the system using the WCETs of each individual events, over the LCM of all applications' periods.

Existing works on schedulability analysis of MSC-based specifications of distributed systems (e.g. [104] and [98]) compute the local worst-case response time for each individual event in a critical instance, which assumes all events are independent. The global worst-case delay is then obtained by summing up these local worst-case response times. However, the dependencies between set of preempting events and preempted events restrict the possible preemption scenarios, which results in the critical/optimal instance assumed for worst/best case response time analysis for set of independent tasks to be too pessimistic/optimistic. For example, suppose events e_i and e_j belong to different applications in a system, and they are mapped to the same PE where e_i has a higher priority than e_j. If e_i and e_j are ready at the same time (e_i imposes the maximum interference on response time of e_j), we have the following.

- Dependency between preempting events: the successor of e_i (say e_k) cannot be ready at the same time as e_j, resulting in e_k preempting e_j fewer number of times

than it could have preempted in the worst case scenario (where e_k is ready at the same time as e_j).

- Dependency between preempted events: subsequent releases of e_i may not be ready at the same time as the successor of e_j (say e_p) which also mapped on the same PE, results in less number of preemptions from e_i on e_p.

[113] proposes a schedulability analysis based on a task (precedence) graph model, which captures the dependency between preempted tasks by capturing phase adjustment between a preempting task and preempted tasks. To retain a conservative analysis, the distance between two preempted tasks must be relatively small to preserve phase adjustment. However, in a bus-based distributed application, it is common to have computations mapped to other PEs as well as bus communications between two preempted events (e.g. e_j and e_p in the above-mentioned example). Such gaps in many cases counteract the usefulness of the phase adjustment.

We adopt the analysis framework from [113] and extend it to consider (a) the dependencies between preempting events, and (b) control flow, in particular non-deterministic branches, among the MSCs in an MSG. In our case, an event e in an application A can be preempted by events in a different application A'. Conditional executions of events in A' should be exploited to avoid a gross overestimation of the preemption cost of e. This is done in our analysis by adapting the idea from the recurring real-time task model in real-time systems literature [7], which allows for conditional branches.

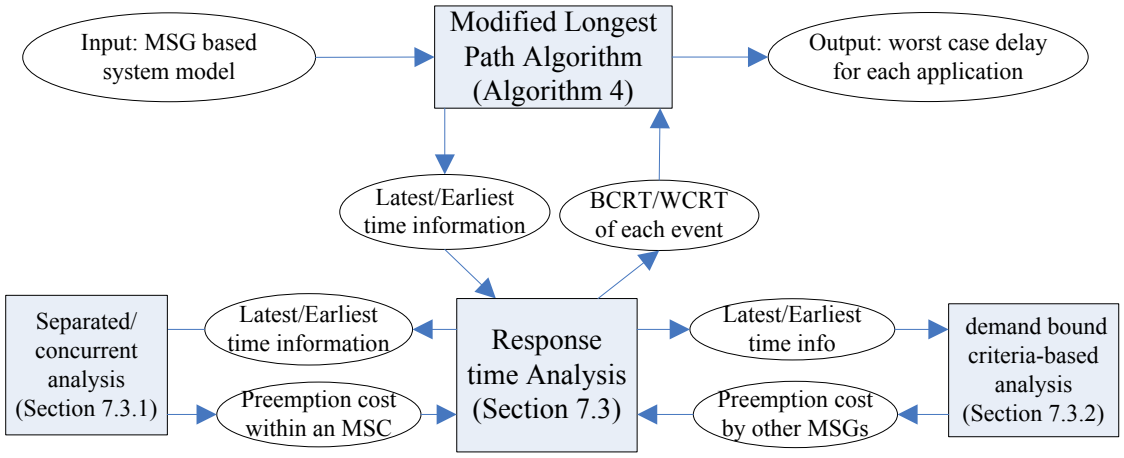


Figure 7.5: Overview of our schedulability analysis framework.

7.2 Schedulability Analysis Framework

Figure 7.5 shows the overview of our feasibility analysis framework for MSG-based system models. Given a set of MSGs each representing a real-time distributed application and annotated with required timing information, our analysis will return an upper bound on the end-to-end delay for each MSG. We present our analysis method in two levels. In this section, we present the top-level analysis for computing end-to-end delay of MSG-based distributed real-time applications, which is a modified longest path algorithm adopted from [113] with necessary modifications to handle MSC concatenation and conditional branching in the MSG model. In the next section, we will present response time analysis of individual events.

To facilitate the analysis, four time instances are defined for each event e and MSC M in an MSG.

• earliest ready time ($earliest[e^r]$, $earliest[M^r]$)

• latest ready time ($latest[e^r]$, $latest[M^r]$)

• earliest finish time ($earliest[e^f]$, $earliest[M^f]$), and

• latest finish time ($latest[e^f]$, $latest[M^f]$).

Algorithm 4 presents the top-level iterative algorithm for computing worst case end-to-end delay ($wcrt[MSG_i]$) for each application MSG_i. The end-to-end delay of MSG_i is defined as the time elapsed between release of its dummy initial node ∇_i (denoted as ∇_i^r) and finish of its dummy end node $\triangle_i$ (denoted as $\triangle_i^f$). The algorithm terminates when (a) no time instance for any of the events is changed (the fixed point is reached), or (b) the maximum number of iteration steps (determined by the system designer) are executed (line 5). It uses information of individual events' response times to generate latest and earliest time instances (line 8-9), which in turn will be used to refine the results of the response time analysis in the next iteration (line 10-11). The top level framework captures the control dependencies between individual MSCs. Since exactly one of the conditional edges are taken for each branch, the earliest ready time of an MSC is set to be the minimum value of the earliest finish times of its predecessors, while the latest ready time of an MSC is set to be the maximum of the latest finish times of its predecessors (line 12-15). The algorithm begins with a very coarse approximation for the start and completion times of the events, and the worst/best case delay it may suffer. The results are refined in each iteration based on the information computed in last iteration. The algorithm is safe in the sense that it never produces under-estimation for the worst case delays or over-estimation for the best case. For an application MSG_i with deadline D_i, our analysis considers it schedulable if $wcrt[MSG_i] \leq D_i$.

Algorithm 4 $computeDelay(SYS)$ — Compute worst case end-to-end delay of each application of the system SYS modeled in MSG specifications.

1: $step = 0$; /*number of iterations*/
2: **for** each application $MSG_i \in SYS$ **do**
3: $latest[\nabla_i^r] = earliest[\nabla_i^r] = 0$; /*initialization*/
4: **end for**
5: **while** any time instance changed and $step < limit$ **do**
6: **for** each application $MSG_i \in SYS$ **do**
7: **for** each MSC M_j of MSG_i in topologically sorted order **do**
8: LatestTimes(M_j); /*Algorithm 5*/
9: EarliestTimes(M_j);
10: $latest[M_j^f] = \max_{e \in M_j}\{latest[e^f]\}$;
11: $earliest[M_j^f] = \max_{e \in M_j}\{earliest[e^f]\}$;
12: **for** each successor MSC M_k of M_j **do**
13: $latest[M_k^r] = \max(latest[M_j^f], latest[M_k^r])$;
14: $earliest[M_k^r] = \min(earliest[M_j^f], earliest[M_k^r])$;
15: **end for**
16: **end for**
17: /*worst case delay of MSG_i*/
18: $wcrt[A_i] = latest[\triangle_i^f]$;
19: **end for**
20: step++;
21: **end while**

Algorithm 5 $LatestTimes(MSC)$ — Compute $latest[e_i^r]$ and $latest[e_i^f]$ for all e_i in MSC.

1: **for** each event e_i in MSC **do**

2: $latest[e_i^r] = latest[MSC^r]$; /*initialize*/

3: **end for**

4: **for** each event e_i respecting the partial order $\preceq^{MSC}$ **do**

5: $w_i = computeWCRT(e_i)$; /* **See Equation 7.3 in Section 7.3 for details** */

6: $latest[e_i^f]$ = $latest[e_i^r]$ + w_i;

7: **for** each immediate successor e_k of e_i **do**

8: **if** $latest[e_k^r] < latest[e_i^f]$ **then**

9: $latest[e_k^r] = latest[e_i^f]$;

10: **end if**

11: **end for**

12: **end for**

The `LatestTimes` calculation invoked at line 8 in Algorithm 4, is shown in Algorithm 5. It is similar to the `LatestTimes` algorithm in [113]. Basically, the algorithm uses a modified longest-path algorithm to take into account partial order and message communication dependencies within a single MSC. Based on dependencies between events of the MSC being analyzed, the main purpose of the algorithm is to update the latest ready and finish times for each event (line 6-10). This updating is independent of the resource scheduling policies on the PEs. The scheduling policy is only taken into account in the calculation of the WCRT of an event (line 5); this calculation will be elaborated in the next section. Finally, the `LatestTimes` algorithm can be easily transformed into the `EarliestTimes` algorithm (invoked at line 9 in Algorithm 4), which updates the earliest ready and finish times by calculating the best-case response time for each event.

7.3 Response Time Calculation

The procedure for computing the earliest/latest ready and finish times of MSC events, as discussed so far, only provides an algorithmic framework. In particular, it depends on worst case and best case response time (WCRT/BCRT) estimates of individual events inside MSCs. We now elaborate the WCRT/BCRT calculation of MSC events. Clearly, this will require us to consider the scheduling policy inside the PEs on which these events are executed. We use fixed priority preemptive scheduling for our response time calculation in this section.

The standard WCRT calculation for fixed-priority scheduling of independent peri-

odic tasks is discussed in Section 2.2.2. To briefly recap, the calculation is given by the following recursive equation:

$$w_i^{n+1} = c_i + \sum_{t_j \in hp(t_i)} c_j \cdot \lceil \frac{w_i^n}{P_j} \rceil \tag{7.1}$$

Here w_i, c_i, and P_i are the response time, computation time, and period for task t_i respectively. The set $hp(t_i)$ denotes the set of higher priority tasks mapped to the same PE as t_i. The fixed point computation starts with $w_i^0 = c_i$, and terminates when the response time calculated in $n+1$th iteration (w_i^{n+1}) equals to the value in previous iteration (w_i^n). Equation 7.1 computes the WCRT of a task t_i in its *critical time instance* (i.e. all higher priority tasks are ready when t_i is ready).

The BCRT calculation is proposed in [92] as

$$b_i^{n+1} = c_i + \sum_{t_j \in hp(t_i)} c_j \cdot (\lceil \frac{b_i^n}{P_j} \rceil - 1) \tag{7.2}$$

for the same setting. It is based on the best case phasing (or optimal instance) where t_i finishes simultaneously with the release of all its higher priority tasks.

However, in our distributed MSC-based system model, we can obtain much more accurate WCRT/BCRT estimates by taking into consideration the dependencies between preempting events as discussed in Section 7.1.2. We divide the worst and best preemption cost on the execution of any event e_i as follows — (a) preemption on e_i by other events in the same application (denoted as WS_i and BS_i), and (b) preemption on e_i by events from other applications (denoted as WD_i and BD_i), respectively. Thus, our WCRT and BCRT equations are given as follows.

$$w_i^{n+1} = c_i + WS_i^n + WD_i^n \tag{7.3}$$

$$b_i^{n+1} = c_i + BS_i^n + BD_i^n \tag{7.4}$$

In the following sections, we elaborate the calculation of these four quantities — WS_i, BS_i, WD_i, BD_i.

7.3.1 Preemption within an MSC

Equation 7.1 and Equation 7.2 assume deadline less than or equal to period for all tasks ($D \leq P$). This guarantees that, for a schedulable task set, a task instance will not get delayed by any its previous instances. In our analysis, we also assume that the deadline is less than or equal to period for all the applications being analyzed. Thus, to show that application MSG is schedulable ($wcrt[MSG] \leq D$), interference from events in previous instances of MSG need not be considered for the critical and optimal time instances. Suppose e_i and e_j are events in the same application MSG, and there is no dependency between them (neither $e_i \preceq e_j$ nor $e_j \preceq e_i$). For e_j to possibly preempt e_i, the events e_i, e_j cannot be events in different MSCs of the MSG model of MSG, since MSCs in an MSG are synchronously concatenated. Moreover, e_j may preempt e_i at most once owing to assumption that deadline is less than or equal to period for all the applications.

Furthermore, for an event e_j to preempt event e_i in a same MSC M, there must be an overlap between their execution time intervals. Let event $NCP(i, j)$ be the nearest common predecessor event for e_i and e_j in M. If such a predecessor event does not exist, we set $NCP(i, j)$ to be the start of M. We define the following quantities.

- smallest time interval between $NCP(i,j)$ finishing and e_i becoming ready

$$SFR_i^{NCP(i,j)} = earliest[e_i^r] - earliest[NCP(i,j)^f]$$

  which corresponds to the scenario that all events on path from $NCP(i,j)$ to e_i execute in their BCRT.

- largest time interval between $NCP(i,j)$ finishing and e_i becoming ready

$$LFR_i^{NCP(i,j)} = latest[e_i^r] - latest[NCP(i,j)^f]$$

  which corresponds to the scenario that all events on path from $NCP(i,j)$ to e_i execute in their WCRT.

- smallest time interval between $NCP(i,j)$ finishing and e_i finishing,

$$SFF_i^{NCP(i,j)} = earliest[e_i^f] - earliest[NCP(i,j)^f]$$

- largest time interval between $NCP(i,j)$ finishing and e_i finishing,

$$LFF_i^{NCP(i,j)} = latest[e_i^f] - latest[NCP(i,j)^f]$$

Executions of two events e_i and e_j from the same vertex are guaranteed to be separated in one execution of the MSG if and only if

$$\begin{aligned} separated(i,j) = e_i \preceq e_j \ \vee \ e_j \preceq e_i \ \vee (LFF_i^{NCP(i,j)} \leq \\ SFR_j^{NCP(i,j)}) \vee \ (LFF_j^{NCP(i,j)} \leq SFR_i^{NCP(i,j)}) \end{aligned} \tag{7.5}$$

evaluates to true, i.e. either there is a dependency between e_i and e_j (as per the partial order for the MSC), or e_i always finishes before e_j releases, or vice versa. Note that the instances of e_i and e_j involved in the preemption belong to the same run of the MSG.

Thus, the above intervals should be measured w.r.t their nearest common predecessor event (instead of start of the MSG ∇), which gives a much more accurate estimation.

Finally, the worst case preemption cost imposed on event e_i by events from same application can be calculated as follows: let c_j^u be the WCET of event e_j.

$$WS_i = \sum\{\, c_j^u \mid contend(j,i) \wedge \neg separated(i,j)\}$$

where $contend(j,i)$ is true if and only if the events e_j and e_i are mapped to the same PE and e_j has higher priority than e_i (as per the scheduling policy of the PE).

For the BCRT calculation of e_i, we find the events e_j that are guaranteed to be ready during e_i's execution.

$$\begin{aligned} concurrent(i,j) = \neg(e_i \preceq e_j) \wedge \neg(e_j \preceq e_i) \wedge (LFR_i^{NCP(i,j)} \\ \leq SFR_j^{NCP(i,j)}) \wedge\ (LFR_j^{NCP(i,j)} \leq SFF_i^{NCP(i,j)}) \end{aligned} \tag{7.6}$$

The best case preemption cost imposed on event e_i by events from same application can be calculated as follows: let c_j^l be the BCET for event e_j.

$$BS_i = \sum\{\, c_j^l \mid contend(j,i) \wedge concurrent(i,j)\}$$

7.3.2 Preemption by a Single MSC

WD_i and BD_i in Equation 7.3 and Equation 7.4 denote worst (best) case preemption cost on event e_i from other applications in system. Before computing the preemption cost between full-fledged applications modeled in MSGs, let's first consider a simpler scenario, where an event $e1$ in application $MSG1$ get preempted by a single MSC $M2$ in another independent application $MSG2$.

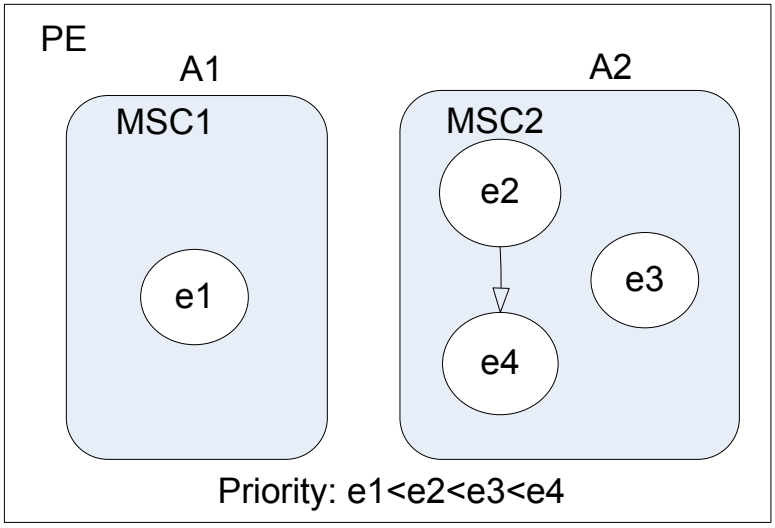


Figure 7.6: Projection of Events on same PE.

Figure 7.6 gives the projection of the events in $M2$ executed by the same PE as $e1$, including dependencies and priority assignments for the fixed-priority preemptive scheduling on PE. The directed edges in Figure 7.6 denote event dependencies. Thus, a directed edge from event e_i to event e_j indicates $e_i \preceq e_j$, as per the partial order $\preceq$ of the MSC in which e_i, e_j reside. Note that there might be events in between e_i and e_j, which are executed on other PEs. Assume that the set of events within an MSC M that can preempt an event e_i is denoted as $ps_{e_i}^M$. For instance, in the example given in Figure 7.6 we have $ps_{e1}^{M2} = \{e2, e3, e4\}$.

Our goal is to find the worst-case preemption scenario for e_i, when e_i is preempted by events in $ps_{e_i}^M$. Existing works calculate the WCRT of e_i by assuming that all events in $ps_{e_i}^M$ release at the critical instance as e_i is ready (see equation 7.1). Clearly, the critical instance assumption will introduce over-estimation on the WCRT of e_i when dependencies exist between preempting events. In our example, $e4$ can only be ready

some time after $e2$ finishes. Thus, if $e2$ is released at the critical time instance (when $e1$ releases), the worst case number of times $e4$ can preempt $e1$ may become less compared to the number assumed in equation 7.1. In our proposed analysis, we will use the latest/earliest ready and finish times of each event in $ps_{e_i}^M$, calculated in each iteration of the delay estimation algorithm shown in Algorithm 4, to tighten the worst-case preemption cost on e_i.

Preempting events in $ps_{e_i}^M$ may either have dependencies (e.g. $e2$ and $e4$) or execute concurrently with other events (e.g. $e3$). In order to explore number of preemptions they may impose on a preempted event, we first construct a *preemption chain* to capture the possible release times of events in $ps_{e_i}^M$. A *preemption chain* $PC_{e_i}^M = \{\hat{N}, \hat{E}\}$, is a sequence of nodes $n \in \hat{N}$, and each directed edge $E(n_1, n_2) \in \hat{E}$ is labeled with weight $W(n_1, n_2)$ representing the minimum time interval between request times of nodes n_1 and n_2. A node n contains a set of events from $ps_{e_i}^M$. Similar to our handling of events in Section 7.3.1, four time instances $earliest[n^r]$, $latest[n^r]$, $earliest[n^f]$, $latest[n^f]$ are defined for each node n in the preemption chain. The upper and lower bound computation time of a node n are denoted as c_n^u and c_n^l respectively; these estimates are obtained from summing up the WCET/BCET of the events in node n.

The algorithm to construct *preemption chain* is shown in Algorithm 6, which takes $ps_{e_i}^M$ as input. Events that may execute concurrently are grouped into one node in the constructed preemption chain PC - the release of any of these events may cause all of them to preempt e_i in the worst case. For each event $e_j \in ps_{e_i}^M$, we insert it into PC in the sequence of the partial order defined by M (line 2). If a newly created

Algorithm 6 $ConstructPC(ps_{e_i}^M)$ — Construct a preemption chain that contains events from MSC M mapped onto the same ECU with e_i

1: $PC = empty$; /*initialize the preemption chain*/
2: **for** each e_j in $ps_{e_i}^M$ as the partial order $\preceq^M$ of MSC M **do**
3: create a new node n containing e_j;
4: /*insert n into PC*/
5: **if** PC is empty **then**
6: PC.insert(n);
7: **else if** $\neg separated(n, source(PC)$ **then**
8: `merge`(n, $source(PC)$); /*merge n into $source(PC)$, Algorithm 7*/
9: **else if** $earliest[n^r] > earliest[source(PC)^r]$ **then**
10: `insertAfter`(n,$source(PC), PC$); /*insert n after $source(PC)$, Algorithm 8*/
11: **else**
12: insert n as the source node of PC; /*n is ready before $source(PC)$*/
13: **end if**
14: **for** each edge $E(n, n_1)$ in PC **do**
15: $W(n, n_1) = earliest[n_1^r] - earliest[n^r]$;
16: **end for**
17: $W(sink(PC), source(PC)) = P(M) - latest[sink(PC)^r] + earliest[source(PC)^r]$
18: **end for**

Algorithm 7 $merge(n, n_1)$ — Merge node n into n_1.

1: $C^u(n_1) = C^u(n_1) + C^u(n)$; /*update computation time*/

2: $earliest(n_1^r) = \min\{earliest(n_1^r), earliest(n^r)\}$;

3: $latest(n_1^r) = \min\{latest(n_1^r), latest(n^r)\}$;

4: $earliest(n_1^f) = \max\{earliest(n_1^f), earliest(n^f)\}$;

5: $latest(n_1^f) = \max\{latest(n_1^f), latest(n^f)\}$;

Algorithm 8 $insertAfter(n, n_1, PC)$ — Insert node n after n_1 in the generated preemption chain.

1: **if** $succ(n_1)$ not exist **then**

2: insert n as the sink node of PC;

3: **else if** $\neg separated(n, succ(n_1)$ **then**

4: `merge`(n,$succ(n_1)$); /*Algorithm 7*/

5: **else if** $earliest[n^r] > earliest[succ(n_1)^r]$ **then**

6: `insertAfter`(n,$succ(n_1), PC$);

7: **else**

8: insert n between n_1 and $succ(n_1)$;

9: **end if**

10: /*$pred(n)$/$succ(n)$ denote immediate predecessor, and successor of n in the preemption chain.*/

node for an event e_j (line 3) has overlapping execution time with the current source node $source(PC)$ of PC (Equation 7.5 evaluated to false), they are group into a single node since they may cause same number of preemptions on e_i (line 7-8). This is done by invoking the `merge` procedure in Algorithm 6, which decides the earliest/latest times for the merged node. Note that such a node is ready when any of its events is ready (see line 2 of the `merge` procedure in Algorithm 7). Otherwise, we look for the "correct" position of the newly created node in PC (line 9-10), via the procedure presented in Algorithm 8. The distance between two nodes will be the minimum time elapsed between their ready time (line 14-16, Algorithm 6). And the distance between the sink node and the next occurrence of the source node is computed based on the period $P(M)$ of the preempting MSC M.

In our example shown in Figure 7.6, suppose $e2$ executes between time interval $[3, 6]$, and $e3$ executes between [4,7]. Then every time $e2$ preempts $e1$, it is also possibly for $e3$ to preempt $e1$ before $e1$ resume its execution. Thus, when considering the worst-case preemption scenario, we can group $e2$ and $e3$ into a single node n_1, which has an earliest ready time of 3, and execution time of $c_2^u + c_3^u$. On other hand, suppose $e4$'s earliest ready time is 10. In this case, e_1 could finish its execution in the interval between e_2 and e_3 finish execution to $e4$ gets released. Thus, number of preemptions caused by node n_1 and the node containing e_4 could be different.

Given the preemption chain $PC_{e_i}^M$ as defined in the preceding, we need to find the maximum preemption cost it imposes on e_i during the worst case response time w_i of e_i. This is equivalent to the problem of finding the *request bound function* of a recurring

real-time task within a time interval t which is discussed by Baruah in [7] (which has been briefly discussed in Section 2.2.2). The **request bound function**, $PC_{e_i}^M.rbf(t)$, accepts a non-negative real number t, and returns the maximum cumulative execution requirement by releasing of nodes in $PC_{e_i}^M$ that have their ready times within any time interval of duration t. We will discuss how the request bound can be calculated when we present our analysis for the full-fledged MSG where conditional branches are added.

Given the request bound of $PC_{e_i}^M.rbf(t)$, the worst case preemption cost imposed on an event e_i by the execution of an independent MSC M within time interval t (quantity WD in Eq. 7.3) is

$$WD_i^n = PC_{e_i}^M.rbf(w_i^n)$$

The calculation for the best case preemption cost is similar modulo the following changes:

- Events are grouped into a node of the preemption chain only if they are guaranteed to execute simultaneously, i.e. replace the condition check $\neg separated(n, n_1)$ by $concurrent(n, n_1)$ (line 7 in Equation 7.6) when constructing the *preemption chain*.

- The computation requirement of a node is replaced with the summation of the lower bound computation times.

- The distance between two connected nodes is modified to represent the *maximum* time interval between ready times of the two nodes.

- The request bound function for $PC_{e_i}^{M}.rbf(t)$ is modified to return the minimum cumulative execution requirement.

The best case preemption cost imposed on an event e_i via execution of other MSC M (quantity BD in Eq. 7.4) is

$$BD_i^n = PC_{e_i}^{M}.rbf(b_i^n)$$

7.3.3 Preemption by MSGs

An MSG modeled application may contain multiple MSC connected by conditional branches, describing its reactions to different environment input (*e.g.*, the packet types obtained as input in an MPEG decoder). To calculate the worst case preemption cost imposed on event e_i by a complete run of application A_i modeled in MSG MSG_i, we first construct a *preemption graph* $PG_{e_i}^{MSG_i}$ capturing the dependencies between the events in MSG_i that can preempt e_i's execution. This is done via the following steps.

1. We construct the preemption chain $PC_{e_i}^{M}$ for each MSC M in MSG_i, based on the algorithm in Algorithm 6.

2. If M' is a successor MSC of M in the MSG for application MSG_i, we create a directed edge $E(M, M')$ from $sink(PC_{e_i}^{M})$ to $source(PC_{e_i}^{M'})$ with weight of

$$earliest[source(PC_{e_i}^{M'})^r] - earliest[sink(PC_{e_i}^{M})^r]$$

that is, the minimum distance between ready time of $sink(PC_{e_i}^{M_i})$ and $source(PC_{e_i}^{M_j})$.

3. Finally, we create a unique dummy source node for the preemption graph, denoted as $source(PG_{e_i}^{MSG_i})$, representing the start time of application MSG_i. The

dummy node is set as the immediate predecessor of each preemption chain $PC_{e_i}^M$ which have no predecessor, with the weight of these edges (from the dummy node) being $earliest[source(PC_{e_i}^M)^r]$.

The above *preemption graph* $PG_{e_i}^{MSG_i}$ captures the release information as well as path information of events in application MSG_i that preempt event e_i. Thus, the WCRT of e_i can be found by computing the preemption cost from each of such applications over e_i's response time. Our *preemption graph* is similar to the graphical representation of a recurring real-time task in [7], where (i) each node is labeled with its execution requirement; (ii) edges are weighted with the minimum triggering-times between two nodes; (iii) two out-going edges from a node represent conditional choice; (iv) and the unique source node is triggered periodically. Thus, our problem of finding $PG_{e_i}^{MSG_i}.rbf(w_i^n)$, the maximum cumulative execution requirement by releasing of nodes in $PG_{e_i}^{MSG_i}$ over e_i's $n-th$ iteration response time w_i^n, can be converted to the problem of computing the *request bound function* of a recurring real-time task over a given time interval. Note that in recurring real-time tasks, each node also has a deadline. However, this deadline information will not be used when calculating the request bound function. In section, we briefly discuss how the *request bound function* can be calculated given the *preemption graph*. The full-detailed computation for *request bound function* can be found in [7].

Consider an event e_1 of response time $w_1 = 50$ preempted by events in an application MSG_i with period of 20, as shown in Fig 7.7. The cost of MSG_i's preemption on e_i within e_i's response time can be divided into two parts — (a) preemption on e_1

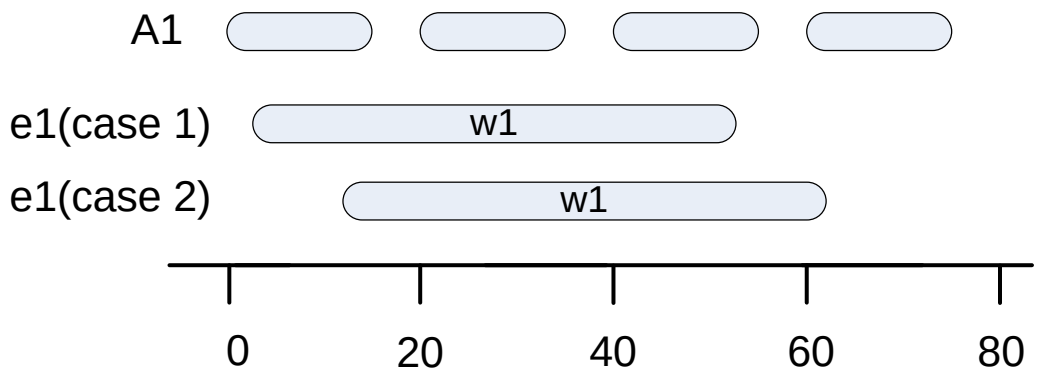


Figure 7.7: Preemption from other applications.

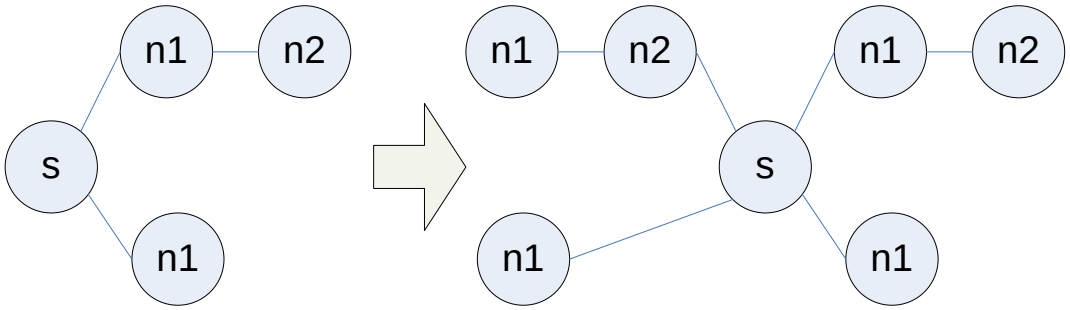


Figure 7.8: Constructing a super preemption graph.

by several complete runs of MSG_i, and (b) preemption on e_1 by possible incomplete runs of MSG_i at the beginning and end of e_1's response time. Hence, the number of complete runs of MSG_i within the response time w_i of e_i may be either $\lfloor \frac{w_i}{P} \rfloor - 1$ as shown in case 1 of Figure 7.7, or $\lfloor \frac{w_i}{P_i} \rfloor$ as shown in case 2 of Figure 7.7.

Clearly, the worst-case preemption cost on e_i from a complete run of MSG_i — denoted as $C(PG_{e_i}^{MSG_i})$ — is the maximum cumulative execution requirement of all nodes along any path in $PG_{e_i}^{MSG_i}$ from the source node to any of the sink nodes.

Finally, we calculate the preemption cost imposed on an event e_i from incomplete runs of application MSG_i. As in [7], we construct a super preemption graph $SPG_{e_i}^{MSG_i}$

by connecting two copies of $PG_{e_i}^{MSG_i}$. One edge is added between each sink node (n_{sink}) of the first copy and the dummy source node of the second copy, with weight of $P - latest[n_{sink}^r]$ where P is the period of application MSG_i. Then the dummy source node of the first copy as well as all its outgoing edges are removed from the super graph. Fig 7.8 shows the super graph construction with an example.

The super graph depicts all possible preemption behaviors for the *two possible incomplete runs of MSG_i at the beginning and end of the response time of* e_i, ignoring the complete runs of MSG_i in between. If the response time w_i of e_i is less than the period P of MSG_i, the *request-critical trace* can be found from all node sequences in $SPG_{e_i}^{MSG_i}$ (the dummy source node need not be included). However, if $w_i \geq P$, the dummy source nodes must be included in the *request-critical trace* for the two incomplete runs of MSG_i within event e_i's response time, that is, they must span over two releases of MSG_i. Let us denote the maximum cumulative execution requirement of nodes in $SPG_{e_i}^{MSG_i}$ over time interval t to be $SPG_{e_i}^{MSG_i}.rbf(t)$ if the corresponding *request-critical trace* includes the dummy source node, or $SPG_{e_i}^{MSG_i}.rbf'(t)$ otherwise. The worst-case preemption cost on e_i from other applications (the quantity WD in equation 7.3 can be expressed as follows.

$$WD_i^n = \sum_{MSG_j} \begin{cases} SPG_{e_i}^{MSG_j}.rbf'(w_i^n), & \text{if } w_i^n < P_j \text{ ;} \\ \max\{\lfloor \frac{w_i^n}{P_j} \rfloor \cdot C(PG_{e_i}^{MSS_j}) + SPG_{e_i}^{MSG_j}.rbf(w_i^n \bmod P_j), & \\ \quad (\lfloor \frac{w_i^n}{P_j} \rfloor - 1) \cdot C(PG_{e_i}^{MSG_j}) + & \\ \quad\quad SPG_{e_i}^{MSG_j}.rbf(P_j + w_i^n \bmod P_j)\}, & \text{otherwise.} \end{cases} \tag{7.7}$$

Similarly, to find the best-case preemption cost, we need to construct the preemption graph and super preemption graph with the weight of edges showing the *maximum* time interval between two connected nodes, and find the minimum cumulative execution requirement over given time interval in the graph. Similar to Equation 7.7, we obtain:

$$BD_i^n = \sum_{MSG_j} \begin{cases} SPG_{e_i}^{MSG_j}.rbf'(b_i^n), & \text{if } b_i^n < P_j \text{ ;} \\ \min\{\lfloor \frac{b_i^n}{P_j} \rfloor \cdot C(PG_{e_i}^{MSG_j}) + SPG_{e_i}^{MSG_j}.rbf(b_i^n \bmod P_j), \\ \quad (\lfloor \frac{b_i^n}{P_j} \rfloor - 1) \cdot C(PG_{e_i}^{MSG_j}) + \\ \quad\quad SPG_{e_i}^{MSG_j}.rbf(P_j + b_i^n \bmod P_j)\}, & \text{otherwise.} \end{cases} \tag{7.8}$$

7.4 Case Study

7.4.1 Experimental Setup

In this section, we illustrate our analysis method by applying it to a setup from the automotive electronics domain. The system architecture of a FlexRay-based ECU network and two distributed applications (ACC and ACP) were presented in Section 7.1.1. The underlying system architecture consists of four ECUs communicating via a shared FlexRay bus, as shown in Figure 7.3. We assume ECU1 implements a Time Division Multiple Access (TDMA) scheduler, while the remaining three ECUs use preemptive fixed-priority scheduling.

Communication on the FlexRay bus takes place in periodic cycles (or bus cycles), where each cycle is partitioned into a static (ST) and a dynamic (DYN) segment. The ST segment is divided into several fixed static slots, and messages can only be sent during their allocated slots. The DYN segment implements an event-triggered bus pro-

tocol based on fixed priority scheduling. Further details of the FlexRay communication protocol can be found in [44, 88]. We compute the best and worst response times for each FlexRay message between its ready time (generated by the sender) and finish time (available to the receiver). For a ST message m_i with a transmission time of C_i, we have

$$b_i = C_i; \;\; w_i^{n+1} = C_i + T + St(w_i^n) \times T;$$

where T is the length of the bus communication cycle, and $St(w_i^n)$ is the number of occurrences of higher priority ST messages using the same ST slot as m_i, within a time interval of length w_i^n. For a DYN message m_i, the response time is calculated as

$$b_i = C_i; \;\; w_i^{n+1} = C_i + T + Dyn(w_i^n) \times T;$$

where $Dyn(w_i^n)$ is the number of occurrences of higher priority DYN messages m_j within w_i^n time units, such that m_j and m_i are not allowed to be transmitted in the same bus cycle (due to size restriction of the DYN segment).

The two applications receive data periodically from the external environment (i.e. radars and sensors), and are required to complete before the next arrival of their input data (i.e. deadlines are equal to periods). We assume input data received by the four radars and the sensor every 100 ms and 50 ms respectively. Thus, the period/deadline of the ACC and ACP applications are 50 ms and 100 ms respectively. Furthermore, we assume the FlexRay bus has a communication cycle of 5 ms.

	ACC	ACP
Proposed analysis	48 ms	95 ms
Saksena and Karvelas [98]	60 ms	110 ms

Table 7.1: End-to-end delay (from sensor/radar to actuator) for the ACC and ACP applications shown in Figure 7.3.

7.4.2 Experimental Results

In this section we present the results obtained by analyzing the setup described above using our proposed analysis technique. Further, we compare these results with those obtained from response time analysis techniques for UML-based system models of multi-threaded implementations of objects/processes [98], where the dependency between events are not considered. Our proposed analysis as well as the one in [98] are *safe* (*i.e.*, if analysis returns "schedulable" then it is guaranteed to be so).

Table 7.1 shows the results obtained using the two techniques when all the ECUs run at a clock frequency of 500 MHz. Note that while our analysis returns a "schedulable" result (i.e. the end-to-end delays of the two applications are lower than the sampling periods of the radars/sensors that feed data into them), the analysis proposed in [98] returns "not schedulable".

Figure 7.9 shows the estimated end-to-end delays of the ACP application using the two analysis techniques when the clock frequencies of ECU2 and ECU4 are chosen between 400 to 700 MHz at a scale of 100 MHz, with the execution times of the associated tasks being scaled accordingly. The frequencies of the remaining ECUs are kept at 500 MHz. Clearly, the delay estimates obtained using our technique are con-

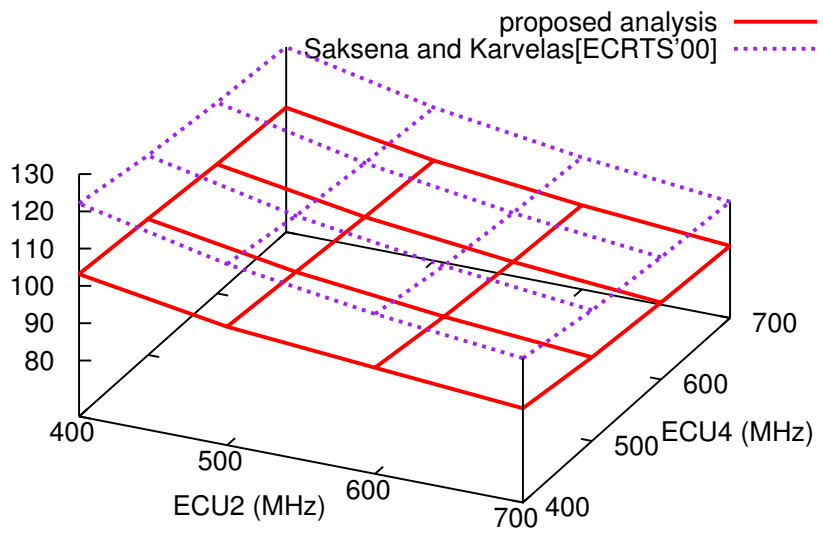


Figure 7.9: Delay bound for ACP obtained using our proposed analysis and the technique presented in [98].

siderably tighter than those obtained using [98] (12% to 16% improvements). Such tighter estimates immediately translate into better resource dimensioning and system design. In Figure 7.9 the clock frequencies are scaled in steps of 100 MHz. It may be noted that our analysis returns "not schedulable" only for two different combinations of frequency settings, viz. (ECU2:400 MHz, ECU4: 500 MHz) and (ECU2:400 MHz, ECU4: 400 MHz), from our underlying design space. On the other hand, the analysis proposed in [98] marks a much larger portion of the design space as "not schedulable". In particular, only (ECU2:700 MHz, ECU4: 600 MHz) and (ECU2:700 MHz, ECU4: 700 MHz), are estimated to be feasible clock frequencies.

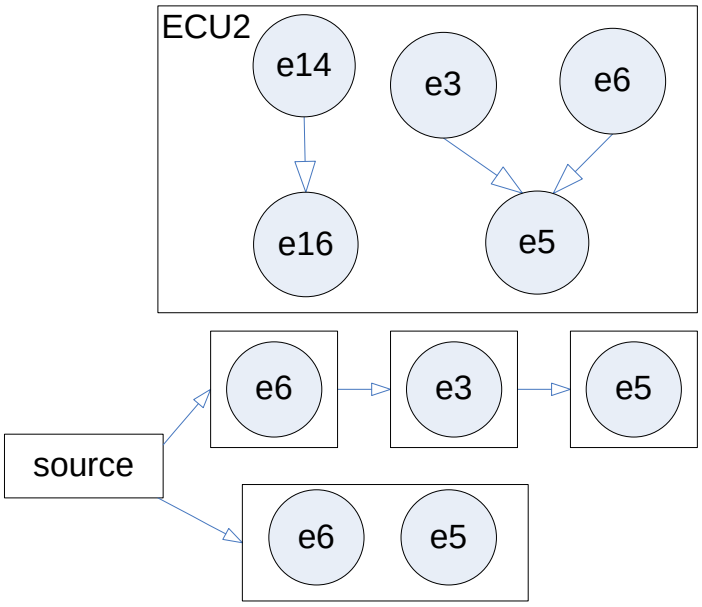


Figure 7.10: Preemption graph for $e14$ by events from the ACC application.

7.4.3 Discussion

There are a number of reasons behind the tighter estimates on the end-to-end delays resulting from our proposed analysis. We discuss some of them below.

- On ECU2: $e16$ is dependent on $e14$. Hence, execution of $e16$ will never get preempted by $e14$ in the same iteration of the ACP application.

- On ECU4: The execution intervals of $e4$ and $e8$ of the ACC application are not interleaving. It can be guaranteed that one execution of $e15$ will not be preempted by both $e4$ and $e8$.

- On ECU2: The set of events in ACC that can possibly contend with $e14$ are $ps_{e14}^{b1} = \{e6, e3, e5\}$ if the MSC $b1$ in Figure 7.4(a) is executed; or $ps_{e14}^{b2} = \{e6, e5\}$ if MSC $b2$ is executed. The dependency information and the resulting

preemption graph as discussed in Section 7.3.3 is shown in Figure 7.10. By computing the request bound function of this preemption graph over the response time of $e14$, our analysis estimates that it is not possible for all three events from ACC application ($e3$, $e5$, and $e6$) to preempt a single execution of $e14$. The worst case actually happens when $e3$ preempts $e14$ first, followed by $e5$, which also holds for the event $e16$.

None of the above scenarios can be taken into account in the technique presented in [98] which, as shown above, leads to pessimistic bounds on end-to-end delay estimates.

7.5 Summary

In this chapter, we have presented a schedulability analysis technique for MSG-based modeling of distributed real-time systems. This makes schedulability analysis techniques accessible to formal system specifications such as MSCs which have long been studied in the context of the Unified Modeling Language (UML). We show the utility of our modeling and response-time analysis with real-life applications from the automotive electronics domain. Our experiments show that our method can consider the event dependencies as prescribed by an MSC partial order as well as sequencing and branching between MSCs in an MSG to produce tight response time estimates of MSG-based system models.

Chapter 8

Conclusion and Future Work

8.1 Thesis Contributions

Worst case timing analysis is of paramount importance for hard real-time system design, by providing *safe* guarantees on the system timing behavior. Two fundamental problems in static worst case timing analysis, WCET analysis and schedulability analysis, have been well-studied for decades. With the increasing complexity of embedded software, model-based design methodologies have become industrial standard. It not only provides an efficient and effective design environment, but also reduces possible design flaws by automatically generating executable code from high-level models. In this thesis, we have made attempts to extend existing timing analysis techniques and seamlessly integrate them into the model-based design framework.

In order to illustrate our model-driven timing analysis, we consider a fairly general model hierarchy called the Globally-Asynchronous Locally-Synchronous (GALS)

model. For large-scale distributed systems, the globally asynchronous model provides system designer with flexibility to relax interaction behaviors between subsystems, and allows the designer to refine one local task at a time. On the other hand, the locally synchronous models have deterministic behaviors which enables formal verification and automatic code generation. In this thesis, we adopt Esterel as the locally synchronous model and message sequence chart (MSC) as the globally asynchronous model.

Although the existing code-level WCET analysis can be directly applied to the generated executable code in a model-based design framework, it usually leads to significant overestimation due to unawareness of the fact that the code is compiled from a high-level model. The overestimated WCET estimates may result in resource over-dimensioning and poor design. Traditional task graph-based system models and their schedulability analyses concern with independent tasks, which are lack of expressive power to model all possible control and data dependencies for complex system functionalities. However, a full-fledged message sequence graph based specification allows designer to model all possible control and data dependencies (e.g., time/event triggering and conditional execution) in a distributed execution platform.

The main contributions of our proposed model-driven timing analysis for the above-mentioned GALS model are:

- We have proposed a comprehensive and accurate model-driven WCET analysis framework for C programs generated from Esterel specification. Our analysis is capable of finding the WCET of a single Esterel tick, or WCRT of multiple consecutive ticks, on both single processor or multiprocessor platforms.

- Our proposed WCET analysis efficiently and effectively identifies and removes (inter- and intra processor) infeasible paths in the generated code by exploiting the semantics and compilation information of the source Esterel specification. We also captures inter-tick architecture contexts when computation of event(s) spans multiple ticks.

- We automatically build a bi-directional traceability between Esterel specification and generated executable code. It helps the designer to identify the performance bottleneck, and refine the current design by optimizing Esterel specification or choose/configure the architecture platform.

- We have proposed a general schedulability analysis for distributed system modeled in a globally asynchronous MSC based specification. Our analysis is able to capture both control and data dependencies in the model specification, via a combination of WCRT based analysis and the demand bound approach.

8.2 Future Work

We have identified the following directions to be pursued in the future.

Model-driven multi-core architectural modeling. In our timing analysis methods proposed for multiprocessor and distributed architecture (Chapter 6 and Chapter 7), we have been mainly focus on the inter-processor control and communication dependencies. Multiprocessor/multi-core architectures is gaining increasing popularity in both

general-purpose computers and embedded systems. Meanwhile, providing tight and complete architectural modeling for such systems is still a difficult problem, due to complexity of modeling shared resources as well as inter-task interferences.

Multi-core architectural modeling has been recently studied for shared instruction cache [72, 56], and shared bus [101, 27]. We believe our model-driven timing analysis can be extended to produce tight timing models of shared resources, with the help from model-level flow information that restricts the possible timing behavior of the component.

Timing analysis of cyber-physical systems. Cyber-physical systems (CPSs) [67] capture the interaction between networked computing systems and the complex physical world. They will dominate a large segment of the computing landscape in the future. High-level models are of significant importance for design and verification of large-scale CPSs. While many CPSs operate under real-time constraints, static timing analysis of CPSs becomes a challenging research problem. In the future, we plan to design a general timing analysis framework for typical CPSs, which consists of distributed execution, heterogeneous microprocessor architectures, and inter-component communications.

Timing analysis for industrial standard models. Although the GALS model considered in this thesis is a fairly general system, we will consider modeling formalisms other than Esterel and MSCs. In particular, our choice of Esterel in illustrating the model-driven WCET analysis is mainly due to (i) its comprehensive and clear seman-

tics to specify concurrent reactive application; and (ii) available of good open-source compiler.

On the other hand, MATLAB Simulink/Stateflow is considered as de facto standard in automobile industry. In the future, we plan to extend our proposed model-driven timing analysis techniques for MATLAB Simulink/Stateflow models. The Simulink model is substantially different from the Esterel model in the sense that the execution time is usually dominated by data-intensive computation, compared to the control dominated systems in the Esterel model. As a result, applying our model-driven techniques to architectural modeling (e.g., for data caches) is expected to produce significant improvement in static timing estimation.

Bibliography

[1] AbsInT GmbH, http://www.absint.com/.

[2] R. Alur and D. Dill. The theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *ICALP*, 2001.

[4] R. Alur and M. Yannakakis. Model checking message sequence charts. In *CONCUR*, 1999.

[5] T. M. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[6] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, pages 295–310, 2004.

[7] S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93,128, 2003.

[8] S. Baruah, D. Chen, S. Gorinsky, and A. K. Mok. Generalized mulitframe tasks. *Real-Time Systems*, 1999.

[9] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. *Proceedings of Real-Time System Symposium*, 1990.

[10] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Readings in hardware/software co-design*, pages 147–159. Kluwer Academic Publishers, 2001.

[11] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[12] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: The SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

[13] R. Bernhard, G. Berry, F. Boussinot, G. Gonthier, A. Ressouche, J. P. Rigault, and J. M. Tanzi. Programming a Reflex game in Esterel v3. Technical Report 07/91, Rapport de Recherche, INRIA, Sophia-Antipolis, France, June 1991.

[14] G. Berry. *Mechanized reasoning and hardware design*, chapter Esterel on hardware. Prentice-Hall, 1992.

[15] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[16] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. TAXYS= Esterel+ Kronos. A tool for verifying real-time properties of embedded systems. *Proceedings of the 40th IEEE Conference on Decision and Control*, 3(4-7), 2001.

[17] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *IEEE International Real-Time Systems Symposium (RTSS)*, 2007.

[18] M. Bertogna, M. Cirinei, G. Lipari, S.S. Sant'Anna, and I. Pisa. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of 17th Euromicro Conference on Real-Time Systems*, pages 209–218, 2005.

[19] M. Boldt, C. Traulsen, and R. von Hanxleden. Worst Case Reaction Time Analysis of Concurrent Reactive Programs. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 203(4):65–79, 2008.

[20] A. Bouali. XEVE, an ESTEREL verification environment. In *Computer Aided Verification*, 1998.

[21] F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 9(79):1270–1282, 1991.

[22] A. Burns. *Advances in Real-Time Systems*, chapter Preemptive priority based scheduling: An appropriate engineering approach, pages 225 – 248. Prentice-Hall, 1994.

[23] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[24] P.Y. Chang, E. Hao, and Y.N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[25] DM Chapiro. *Globally-asynchronous locally-synchronous systems*. PhD thesis, Stanford University, 1984.

[26] S. Chattopadhyay and A. Roychoudhury. Unified Cache Modeling for WCET Analysis and Layout Optimizations. In *2009 30th IEEE Real-Time Systems Symposium*, pages 47–56. IEEE, 2009.

[27] S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2010.

[28] M. Chiodo, D. Engels, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, and A. Sangiovanni-Vincentelli. A case study in computer-aided co-design of embedded controllers. *Design Automation for Embedded Systems*, 1(1):51–67, 1996.

[29] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2):249–274, 2000.

[30] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[31] T.A. Dingus, SK Jahns, AD Horowitz, and R. Knipling. Human factors design issues for crash avoidance systems. *Human factors in intelligent transportation systems*, pages 55–93, 1998.

[32] T. Dinh-Trong, N. Kawane, S. Ghosh, R. France, and AA Andrews. A tool-supported approach to testing UML design models. In *10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 519–528, 2005.

[33] S.A. Edwards. The Estbench Esterel Benchmark Suite. *http://www1.cs.columbia.edu/ sedwards/software.html*, 2003.

[34] S.A. Edwards and J. Zeng. Code generation in the columbia Esterel compiler. *EURASIP Journal on Embedded Systems*, 2007.

[35] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. In *Design, Automation and Test in Europe (DATE)*, 1998.

[36] J. Engblom. *Processor pipelines and static worst-case execution time analysis*. PhD thesis, Uppsala University, Sweden, 2002.

[37] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *IEEE International Real-Time Systems Symposium (RTSS)*, 2000.

[38] E. Erpenbach and P. Altenbernd. Worst-case execution times and schedulability analysis of Statecharts models. *Euromicro Conference on Real-Time Systems (ECRTS*, 1999.

[39] Esterel Studio, http://www.synfora.com/products/esterelstudio.html.

[40] C.-G. Lee *et al.* Analysis of cache-related preemption delay in fixed-priority preemtive scheduling. *IEEE Trans. Computers*, 47(6):700–713, 1998.

[41] X. Li *et al.* Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007, `http://www.comp.nus.edu.sg/~rpembed/chronos`.

[42] C. Ferdinand and *et al.* Reliable and precise WCET determination for a real-life processor. In *International Conference on Embedded Software (EMSOFT)*, 2001.

[43] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In *LCTES*, 1998.

[44] The flexray communications system specifications, ver 2.1, www.flexray.com, 2005.

[45] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering*, pages 37–54, 2007.

[46] R. Gerber, W. Pugh, and M. Saksena. Parametric dispatching of hard real-time tasks. *IEEE transactions on computers*, 44(3), 1995.

[47] A. Girault. A survey of automatic distribution method for synchronous programs. In *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05*, 2005.

[48] The Object Management Group. *UML Profile for Schedulability, Performance, and Time Specification*. OMG, 2003.

[49] The Object Management Group. *UML 2.0: Superstructure Specification*. Version 2.0, OMG, formal/05-07-04, 2005.

[50] N. Guan, M. Stigge, W. Yi, and G. Yu. New Response Time Bounds for Fixed Priority Multiprocessor Scheduling. In *IEEE International Real-Time Systems Symposium (RTSS)*, 2009.

[51] J. Gustafsson, P. Altenbernd, A. Ermedahl, and B. Lisper. Approximate worst-case execution time analysis for early stage embedded systems development. In *Proc. of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009)*, 2009.

[52] J. Gustafsson, A. Ermedahl, and B. Lisper. Algorithms for infeasible path calculation. *International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[53] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *IEEE International Real-Time Systems Symposium (RTSS)*, 2006.

[54] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), 1991.

[55] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering*, 20(1):13 – 28, 1994.

[56] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[57] D. Harel. Statecharts: a visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[58] D. Harel and P.S. Thiagarajan. *UML for real: design of embedded real-time systems*, chapter Message Sequence Charts. Kluwer, 2003.

[59] R. Heckmann and *et al.* Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executables. In *4th European Congress on Embedded and Real Time Software (ERTS)*, 2008.

[60] ITU-T. 120: Message sequence chart (msc). *ITU-T, Geneva*, 1996.

[61] L. Ju, B.K. Huynh, S. Chakraborty, and A. Roychoudhury. Context-sensitive timing analysis of Esterel programs. In *Design Automation Conference (DAC)*, 2009.

[62] L. Ju, B.K. Huynh, A. Roychoudhury, and S. Chakraborty. Performance debugging of Esterel specifications. In *International Conference on Hardware-Software Codesign and System Synthesis (CODES-ISSS)*, 2008.

[63] L. Ju, B.K. Huynh, A. Roychoudhury, and S. Chakraborty. Timing analysis of Esterel programs on general-purpose multiprocessors. In *Design Automation Conference (DAC)*, 2010.

[64] L. Ju, A. Roychoudhury, and S. Chakraborty. Schedulability analysis of MSC-based system models. In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2008.

[65] R. Kirner, R. Lang, G. Freiberger, and P. Puschner. Fully automatic worst-case execution time analysis for Matlab/Simulink models. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2002.

[66] R. Kirner and P. Puschner. Transformation of path information for wcet analysis during compilation. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 29, 2001.

[67] E. A. Lee. Cyber physical systems: Design challenges. In *IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, 2008.

[68] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *IEEE International Real-Time Systems Symposium (RTSS)*, 1990.

[69] X. Li, J. Lukoschus, M. Boldt, M. Harder, and R. Von Hanxleden. An Esterel processor with full preemption support and its worst case reaction time analysis. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2005.

[70] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 34(3):195–227, 2006.

[71] X. Li and R. von Hanxleden. Multi-Threaded Reactive ProgrammingłThe Kiel Esterel Processor. *IEEE Transactions on Computers*, 2010.

[72] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[73] J. Lilius and I. Paltor. Formalising UML state machines for model checking. *Proceedings of UML'99, volume 1723 of LNCS*, pages 756–756, 1999.

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

[75] G. Logothetis and K. Schneider. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In *Design, Automation and Test in Europe (DATE)*, 2003.

[76] G. Logothetis, K. Schneider, and C. Metzler. Exact low-level runtime analysis of synchronous programs for formal verification of real-time systems. *Forum on Design Languages (FDL)*, 2003.

[77] G. Logothetis, K. Schneider, and C. Metzler. Generating gormal models for real-time verification by exact low-level runtime analysis of synchronous programs. In *IEEE International Real-Time Systems Symposium (RTSS)*, 2003.

[78] T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–15, 1998.

[79] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2):183–207, 1999.

[80] G. Marsden, M. McDonald, and M. Brackstone. Towards an understanding of adaptive cruise control. *Transportation Research Part C: Emerging Technologies*, 9(1):33–51, 2001.

[81] M. Mendler, R. von Hanxleden, and C. Traulsen. WCRT algebra and interfaces for Esterel-style synchronous processing. In *Design, Automation and Test in Europe (DATE)*, 2009.

[82] A. K. Mok and D. Chen. A mulitframe model for real-time tasks. *Proceedings of Real-Time Systems Symposium*, 1996.

[83] J. Muttersbach, T. Villiger, and W. Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 52, 2000.

[84] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*, 2003.

[85] M.J.K. Nielsen. TURBOchannel. In *Proceedings of 36th IEEE Computer Society International Conference, COMPCON*, 1991.

[86] P. Pop, P. Eles, and Z. Peng. Schedulability analysis for systems with data and control dependencies. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2000.

[87] T. Pop, P. Eles, and Z. Peng. Schedulability analysis for distributed heterogeneous time/event triggered real-time systems. In *Proceedings of 15th Euromicro Conference on Real-Time Systems*, pages 257–266, 2003.

[88] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing Analysis of the FlexRay Communication Protocol. *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 203–213, 2006.

[89] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. *Real-Time Systems*, 39(1):205–235, 2008.

[90] D. Potop-Butucaru, S.A. Edwards, and G. Berry. *Compiling ESTEREL*. Springer, 2007.

[91] K. Ramamritham and J. A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(3):55,67, 1994.

[92] O. Redell and M. Sanfridson. Exact best-case response time analysis of fixed priority scheduled tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2002.

[93] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[94] M.A. Reniers. *Message Sequence Chart: syntax and semantics*. PhD thesis, Technical University of Eindhoven, Netherlands, 1999.

[95] T. Ringler. Static worst-case execution time analysis of synchronous programs. In *5th Ada-Europe International Conference*, LNCS 1845, 2000.

[96] P.S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis for synchronous C programs. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2009.

[97] J. Rosen and *et al.* Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. *IEEE Real-Time Systems Symposium (RTSS)*, 2007.

[98] M. Saksena and P. Karvelas. Designing for schedulability: Integrating schedulability analysis with object-oriented design. *Euromicro Conference on Real-Time Systems (ECRTS)*, 2000.

[99] SCADE Suite, http://www.esterel-technologies.com/products/scade-suite/.

[100] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *International Conference on Application of Concurrency to System Design (ICACSD)*, 2001.

[101] A. Schranzhofer, J.J. Chen, and L. Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010.

[102] R. K. Shyamasundar and J. V. Aghav. Realizing real-yime systems from synchronous language specifications. In *IEEE International Real-Time Systems Symposium (RTSS), Work-in-Progress Session*, 2000.

[103] MATLAB Simulink, http://www.mathworks.com/products/simulink/.

[104] F. Slomka, J. Zant, and L. Lambert. Schedulability analysis of heterogeneous systems for performance message sequence chart. *International Conference on Hardware-Software Codesign (CODES)*, 1998.

[105] F. Soares and P.J.C. Branco. Simulation of a 6/4 switched reluctance motor based on Matlab/Simulink environment. *IEEE Transactions on Aerospace and Electronic Systems*, 37(3):989–1009, 2001.

[106] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2005.

[107] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Design Automation Conference (DAC)*, 2006.

[108] L. Tan, B. Wachter, P. Lucas, and R. Wilhelm. Improving timing analysis for Matlab Simulink/Stateflow. *MoDELS'09 ACES-MB Workshop*, 2009.

[109] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2):157–179, 2000.

[110] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and microprogramming*, 40(2-3):117–134, 1994.

[111] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2007.

[112] R. Wilhelm et al. The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008.

[113] T.Y. Yen and W. Wolf. Performance Estimation for Real-Time Distributed Embedded Systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11), 1998.

[114] L. H. Yoong, P. Roop, Z. Salcic, and F. Gruian. Compiling Esterel for distributed execution. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP)*, 2006.

[115] S. Yuan, S. Andalam, L.H. Yoong, P.S. Roop, and Z. Salcic. Starpro–a new multithreaded direct execution platform for esterel. *Electronic Notes in Theoretical Computer Science*, 238(1):37–55, 2009.

Glossary

Columbia Esterel Compiler(CEC) an open-source Esterel compiler developed in Columbia University.

control dependency an instruction B is control dependent on a preceding instruction A if the latter determines whether B should execute or not.

control state variable an integer variable introduced in generated C code by a control flow graph based Esterel compiler, which is used to keep track of the tick transitions of a concurrent Esterel process.

data dependency an instruction B is data dependent on a preceding instruction A if B reads some data written by A.

electronic control unit (ECU) in automotive electronics, ECU is a generic term for any embedded system that controls one or more of the electrical systems or subsystems in a motor vehicle.

Esterel a synchronous language with imperative programming style, which is well suited for control-dominated model designs.

Globally-Asynchronous Locally-Synchronous (GALS) a system model hierarchy where individual local tasks are described in synchronous models and communicate with each other asynchronously.

infeasible path a path in a program's control flow graph that is not appearing in the execution trace of that program for any input.

integer linear programming (ILP) a technique for calculating integer solutions that optimize a linear objective function, subject to linear equality and linear inequality constraints.

message sequence chart (MSC) an interaction diagram from the SDL family very similar to UML's sequence diagram, standardized by the International Telecommunication Union.

MSC graphs (MSGs) a hierarchical graph whose vertices are labeled by MSCs, each of which represents a single logical unit of interaction.

processing element (PE) a unit of hardware for execution of software tasks.

schedulability analysis a methodology to statically determines whether all real-time tasks can meet their deadlines under a given scheduling policy.

sequential control flow graph (SCFG) an intermediate representation used in Columbia Esterel Compiler for control flow graph-based Esterel compilation.

state variable see also control state variable.

synchronous language a high-level specification language optimized for programming real-time reactive systems.

synchrony hypothesis in synchronous languages, all computation and communication, unless explicitly separated in different logical ticks, happen instantaneously.

tick function a loop-free C function generated from an Esterel specification, where one complete execution of the function represents computation and communication required to be instantaneously executed within one Esterel clock tick..

tick transition automata (TTA) a finite state automata that captures the control state variable changes between different Esterel ticks.

Unified Modeling Language (UML) a standardized general-purpose modeling language in the field of software engineering.

WCET analysis a methodology to estimate WCET bound by statically analyzing the characteristics of the program code and the target hardware.

worst case execution time (WCET) the maximum length of time that a task could take to execute on a specific hardware platform.

worst case response time (WCRT) total time elapsed between release and completion of a computation task (possibly interfered by execution of other tasks).