Synchronous Reactive Communication: Generalization, Implementation, and Optimization

Guoqiang Wang

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-178
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-178.html

December 19, 2008

Synchronous Reactive Communication: Generalization, Implementation, and Optimization

by

Guoqiang Wang

B.E. (Xi'an Jiaotong University, China) 1995
M.E. (Xi'an Jiaotong University, China) 1998
M.S. (University of California, Berkeley) 2007

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering Science - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Alberto Sangiovanni-Vincentelli, Chair
Professor Robert K. Brayton
Professor Zuojun Shen

Fall 2008

The dissertation of Guoqiang Wang is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2008

Synchronous Reactive Communication: Generalization, Implementation, and Optimization

Guoqiang Wang

Abstract

Synchronous Reactive Communication: Generalization, Implementation, and Optimization

by

Guoqiang Wang

Doctor of Philosophy in Engineering Science - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

A fundamental asset of a model-based development process is the capability of providing an automatic implementation of the model that preserves its semantics, and at the same time makes efficient use of the execution platform resources. Synchronous Reactive (SR) models are increasingly used in model-based design flows for the development of embedded control applications.

The implementation of communication links between functional blocks in an SR model requires buffering schemes and access procedures implemented at the kernel level. Platform-based design methodology is introduced to synthesize a real-time operating system when implementing SR models. Previous research has proposed two methods for sizing the communication buffer. This dissertation demonstrates how it is possible to improve on the

state of the art, providing not only tighter bounds by leveraging task timing information, but also an approach that is capable of dealing with a more general model and implementation platform configuration.

To achieve rigorous model semantics, this dissertation presents semantics preserving implementations of SR communication for multi-rate systems on single processor architectures. The implemented protocols define the assignment of indices of shared buffers to writer and reader tasks at activation time, rather than at execution time. Two constant-time portable solutions are developed in the C language and with the automotive OSEK OS standard. Run-time complexity and memory requirements are discussed for the two protocol implementations, and tradeoffs are analyzed. This dissertation completes the SR model-based design flow by supporting automatic code generation for the double buffer and the dynamic buffering protocols. To support software portability and reusability, the ePICos18, compliant to the OSEK OS standard, is used. The generated code is validated by emulation on the PIC18F452 microcontroller through the MPLAB IDE simulator.

An implementation of communication links with a minimum buffer size is often desirable, but it may require a longer access time and it may also lead to the violation of deadline constraints in real-time applications. This dissertation demonstrates the feasibility of an MILP-based optimization approach that provides the minimum memory implementation of a set of communication channels within the deadline constraints of the tasks.

Professor Alberto Sangiovanni-Vincentelli
Dissertation Committee Chair

To My Daughter, My Wife, and My Parents.

Contents

List of Figures

List of Tables

Acknowledgments

At the very first, I would like to thank my parents for bringing me up and affording steadfast support for my study. They have been providing considerable opportunities for me to pursue my dreams. Without their eternal love, long-term spiritual as well as financial supports, and great encouragement, I would not be able to complete my advanced studies.

I would like to express my greatest gratitude to my advisor, Professor Alberto Sangiovanni-Vincentelli, who guided me into the field of Electrical Engineering and Computer Sciences and offered me this invaluable opportunity to learn interdisciplinary expertise. I would possibly fail to accomplish what I have achieved without his generous support and guidance.

I am deeply indebted to Professor Robert K. Brayton. Bob served on my dissertation committee and was the Chair of my qualifying examination committee. He is always willing to discuss with me. Bob led me into the area of sequential synthesis and offered me an opportunity to collaborate. Besides, he shared his views on being a prominent researcher in academia and industry.

I wish to express my sincere appreciation to Professor Zuojun Max Shen from IEOR and Professor Sanjit A. Seshia for serving on my qualifying examination committee. In addition, Max was also on my dissertation committee.

Special gratitude extends to Professor Marco Di Natale from Scuola Superiore Sant'Anna, Pisa, Italy. Marco helped with defining and mentoring my dissertation project. He shared a wealth of hands-on experiences on being an eminent researcher in general.

I have furthermore to thank Professor Andreas Kuehlmann. Andreas guided me

into the area of logic synthesis and verification. Afterwards he provided me an opportunity for research collaboration and mentored my project on symmetry detection.

I am highly thankful to Professor Randy H. Katz. Randy was of great help for reading my master's degree thesis and offering comments on experimental setup for performance analysis. He taught me my very first lesson in digital system design.

I am very much obliged to Professor Edward A. Lee for bringing me to the palace of the models of computation. Knowledge on computation models provided me a solid theoretical foundation and facilitated my dissertation project. I was honored to be hooded by Edward.

I owe special thanks to Dr. Alex Kondratyev and Dr. Yosinori Watanabe from Cadence Research Labs for exposing me to hardware scheduling and behavioral synthesis as well as mentoring my research project.

Sincere thanks are also due to Professors James Demmel, Anthony D. Joseph, Richard M. Karp, George Necula, Christos Papadimitriou, and Pravin Varaiya for spending time uncovering my confusion.

I would also like to say thanks a lot to Ms. Ruth Gjerde and Ms. La Shana Porlaris for taking good care of the graduate students in the Department. They have really done a great logistics job for us.

I wish to extend my thanks to GSRC and CHESS for their sponsorship of my research projects. During the period of my study at Berkeley, I also received enormous help from friends as well as colleagues and staff members in the DOP Center and the BWRC. I genuinely appreciate this great environment and thank them for their help and support.

I am deeply grateful to my wife, Jia Yang, for her patient love and careful consideration for the family. She has been taking more responsibility on raising our daughter, Hannah. Meanwhile, she has been doing most of the housework. This enabled me to fully concentrate on my research work. I also want to express my gratitude to Hannah for the few time I spent with her. Fortunately, both my parents and my parents-in-law offered tremendous help with raising Hannah, which made it possible for Jia and I to attend graduate school. I greatly appreciate their help.

For all those who helped me in one way or another, may God Bless you!

Part I

Background and Problem Statement

Chapter 1

Introduction

1.1 Embedded Systems and Design Methodology

In this section, embedded computing is first introduced with examples and its common characteristics are described. Then embedded system design requirements and challenges are presented. Finally, to cope with the challenges, system-level design methodology, the enabler of ubiquitous computing, is discussed in details.

1.1.1 Embedded Systems

Since the first computer was invented in 1930s, it has been experiencing significant development: programmable computers, personal computers, workstations, laptop computers, etc, all of which share the same common characteristics: general-purpose computing. Another type of computation, called embedded computing, abounds in people's daily life. Any computing systems other than a computer can be considered as an embedded system. Nowadays, in electronic industry, there are more embedded system products in the mar-

ket than personal computer products. According to ERCIM news [110], more than 98% of processors applied in 2003 are in embedded systems. This explicitly indicates that it has already been in the post-PC era. There are many applications for embedded systems and they mainly fall into two categories, depending whether or not they are safety-critical. Typical examples of safety-critical embedded systems include transportation systems (e.g., aircrafts, automobiles, and railways), military systems (e.g., missiles and radars), medical diagnostic systems (e.g., robotic surgeons), identity checking equipments (e.g., fingerprint identifiers, card-key readers), etc. The other category is very broad: examples used for communication include bluetooth devices, cell phones, fax machines, modems, pagers, routers, etc; examples used for entertainment include camcorders, cameras, CD players, DVD players, MP3 players, radios, set-top boxes, stereo systems, televisions, VCRs, etc; examples of office use are printers, scanners, etc; examples for kitchen use are dishwashers, microwaves, etc.

Unlike general-purpose computers, embedded systems share the following common characteristics. They usually function for a particular purpose only and are based on programmable components such as microcontrollers and Digital Signal Processors (DSP). Embedded systems usually do not have direct interaction or control from end users. However, they engage the physical world via directly interacting with sensors and actuators. Reactive embedded real-time systems react to changes in the external environment usually through performing computation based on data from sensors and producing results to control actuators in a timely manner. Ideally, the close interaction with the environment never terminates. The wide range of examples of embedded systems listed above reveals that there

exist both small and large scale systems in the embedded application arena. Embedded system functionality is very heterogeneous. They may need to meet stringent constraints such as time, power, and weight/size as well as have a higher level of dependability such as reliability, robustness, fault-tolerance, and availability.

Embedded systems are usually implemented as Systems-On-a-Chip (SOC). Designing embedded electronic products is a complex process, which involves modeling, implementation, testing, and manufacturing. Functionality of most embedded systems is mapped into both hardware and software. However, embedded systems are increasingly more software driven and functionality has steadily shifted from hardware to software, i.e., more and more functionalities are achieved via embedded software [67].

Ubiquitous embedded computing has been demanded by accident-free automotive, free and safe aerospace, continuously seamless connectivity, etc. This pervasive computing demand has been enabled by new technologies, intelligent sensors/actuators, new methodologies for embedded system design. Among these enablers, design methodologies are particularly important, which are the focus of the next subsection.

1.1.2 Design Methodology

Steadily increasing networked embedded devices are usually small, mobile, adaptable, and configurable. They need to provide strong guarantees of availability and performance. Demands for more elaborate application functionalities further increase embedded software complexity. Development efforts increase greatly due to the rising design complexity. The dynamics of electronic system market demands for shorter and shorter development time. Synchronization, concurrency, and heterogeneity are the essential aspects of reactive

embedded systems. Among them, heterogeneity is the fundamental problem that embedded system designers have to face. Composition of essential properties such as safety and liveness is the key to characterize heterogeneous systems. It is essential for the system designers to be able to specify, manage, and verify the functionality and performance of concurrent behavior.

There exist multiple design metrics [117]: unit manufacturing cost, Non-Recurring Engineering (NRE) cost (the one-time cost of designing the system, e.g., from tooling of systems), physical size, performance (latency and throughput), power consumption, flexibility for functionality adaption, time-to-market (if outside of the market window, the revenues may drop dramatically), system maintainability, etc. The biggest challenge for embedded system design is to simultaneously optimize numerous design metrics. This may be very difficult because some of the metrics compete against each other. Therefore, in practice, the goal of electronic embedded system design is to balance costs from development time as well as product performance and functionality.

Key embedded system technologies are discussed in [117]. There are three types of architectures of the computing engine used to perform system functionality: general-purpose, application-specific, and single-purpose. Unlike single-purpose processors, general-purpose and application-specific processors have program memory. Designs with general-purpose processors may achieve low time-to-market, low NRE cost, and high flexibility while designs with application-specific and single-purpose processors are featured with good performance, small size, and low power. Application Specific Instruction Set Processors (ASIP) [103][42] are a cost-effective design style due to the flexible support for cost or

power reduction and the increased functionality complexity in newer generations of design. Similarly, there are three types of Integrated Circuit (IC) technologies to support embedded hardware: full-custom, semi-custom, and programmable logic device (e.g., Field-Programmable Gate Array (FPGA)). From full-custom to programmable logic, performance, time-to-market, and NRE cost decrease while size and power consumption increase.

Embedded system manufacturing cost, area, power consumption, and weight mainly depend on its hardware components while functional correctness and timeliness are mostly dependant upon its software components. Embedded software is hard to design [67] because it requires domain-specific expertise. It is important to deliver low-cost and efficient implementations in both hardware and software.

Traditional design methods are usually based on Register Transfer Level (RTL) Hardware Description Languages (HDL) or programming languages such as C/assembly. There are problems with past electronic design methods: e.g., lack of unified hardware-software representation which makes specification revision difficult; partitions into hardware and software are defined a priori so that it cannot verify the entire system and is difficult to find incompatibilities across the hardware-software boundary; lack of well-defined design flow which leads to a time-to-market problem. On the other hand, traditional general software development techniques cannot be directly applied for embedded software design due to the poor product quality, long development time, and low development productivity. In addition, monolithic execution platform dependent implementations are unfriendly to port, upgrade, or customize, thus they have limited chances for reuse. Furthermore, traditional design methodology cannot model concurrency due to the lack of enabling semantic

constructs.

With more and more design challenges emerging, the winning strategies for embedded system design are design methodology, testing, and automation. The next generation of tool chain for modeling, design, validation, and implementation should be capable of producing embedded hardware/software systems that are cost-efficient, performance-optimal, ultra-stable, and highly dependable with reduced time-to-market.

To accomplish these, it may require knowledge from multi-disciplines. Verification and validation of system properties and functionality to meet design constraints should be supported as early as possible during the development process. To effectively reduce the cost and development time, design reuse in all shapes and forms should be well supported. A system-level design approach is the key for successful development of hardware and software products characterized as real-time, distributed, parallel, and reactive. Electronic System-Level (ESL) [43] design methodology starts with a high level of abstraction. Moving to a higher level of abstraction makes designs in hardware and software indistinguishable so that it enables to take best advantage of design freedom and prevents errors from the interaction between software and hardware designs. Hence, it may effectively reduce design time. This technique has been successfully used several times [39]: design methodology from simulation-based at transistor level (prior to early 1980s) to simulation-based at gate level (in 1980s), and then to synthesis-based at register-transfer level (between late 1980s and late 1990s). Finally, it comes to refinement-based design methodology at system level (from early 2000s). It is important for system-level design methodology to separate/orthogonalize concerns [60], for example, to separate the various design aspects: functionality versus architecture and

computation versus communication. This enables more effective exploration of alternative solutions, better design complexity management, and component reuse.

Embedded real-time software design must take into consideration of the laws of physics to control the environment. The design framework should have properties that better match the application domain and define a model of computation that governs the interaction of components. The requirements for embedded software development tools are compilation retargetability, multilevel simulation, source-level debugging tool with links to In-Circuit Emulation (ICE), and computer-aided exploration of processor architectures.

To cope with the increasing SOC design complexity, increased system design productivity, as well as decreased time-to-market, system-level methodology is the key. System-level design methodologies have already be around for about a decade. Fundamentals with object-oriented design methods for development of formal executable models at system level are discussed in [118]. The synthesis/refinement-based ESL approach is the most appealing because it reduces the risk of making mistakes and uses powerful optimization techniques to reduce cost (e.g., power, performance, and area) and time of designs, and verifies designs a lot faster than traditional methodologies. System-level design languages [32] may reduce the semantics gap between system-level specifications and Intellectual Property (IP) level implementation decisions.

System-level hardware/software co-design provides a unified design environment. It consists of three parts at four different abstraction levels. To support reuse, an IP library incorporates pre-designed implementations of hardware/software/operating systems, cores, RT components, and gate cells at the system, behavioral, register, and gate levels, respec-

tively. Synthesis/compilation tools (system synthesis, behavior synthesis, RT synthesis, and logic synthesis) automatically explore the design space and insert low level implementation details in the best way with respect to design constraints. Verification tools (model checking, hardware/software co-simulator, HDL simulator, and gate simulator) ensure correctness at each level.

Basic principles on system-level methodology are presented in [39]. A system-level design methodology starts with a well-defined executable specification model that serves as the golden reference. The most important for functional specification is the underlying mathematical model of computation. Well-defined (clear/unambiguous) formal mathematical model semantics for system-level methodology may bridge the gap between system specification and implementation. Models should be abstract enough and defined at each level of abstraction. Model refinements from high to low level give the best implementation. Model equivalence based on simulation semantics specifies that two models are equivalent if they have the same simulation result. The behavior of a correct implementation should be consistent with that of the abstract model. Details about model-based design are presented in next section.

1.2 Model-Based Design

In this section, model-based design, a particular instance of system-level design methodology is discussed. First of all, commonly used models of computation are introduced, and then some representative model implementation tools are presented.

1.2.1 Models of Computation

Model-based development of embedded real-time software aims at improving quality by enabling design-time verification as well as simulation and fostering reuse. This subsection introduces those commonly used models of computation. There are excellent academic articles for each of them. In addition, there are great courses offered in universities addressing models of computation. Two typical examples are EE249 [97] and EE290N [98], which are graduate-level classes offered at the University of California, Berkeley.

Models of computation provide a high level of abstraction for behavioral design. They need to be expressive, general, simple, compilable/synthesizable, and verifiable. A model of computation is a framework in which to express what sequence of actions must be taken to complete a computation, e.g., Finite State Machine, Turing machine, continuous time models (differential equation), etc. There exist many different models of computation and different models signify different properties. Turing-complete models might be too powerful and some properties may be undecidable. A good model of computation needs to be powerful enough for the application domain and has appropriate synthesis and validation algorithms. The model of a design needs to have a precise, unambiguous semantics. Models of computation provide a formal framework for reasoning about certain aspects/properties of an object in embedded systems. In the following, commonly used models of computation are briefly discussed. Note that the following list is never attempted to be complete.

Process Networks Process Networks (PN) are a model of computation that is based on asynchronous message passing, where the writer does not need to wait for the reader to be ready for communication. The blocking read and non-blocking write mechanism

guarantees the model determinacy. Under the blocking read mechanism, processes cannot test input queues for emptiness. Typical examples of process networks are the Kahn's process networks [57][58], where First-In-First-Out (FIFO) queues with an infinite capacity are used to model communication between processes. Process networks are Turing-complete. It is very expressive but key system properties such as deadlock and memory boundedness are undecidable.

Finite State Machine The Finite State Machine (FSM) model of computation is like a graphical language. It consists of states and transitions of a system. In terms of reactiveness, FSMs can be categorized as either Moore or Mealy machine, while in terms of communication mechanism, FSMs can be either synchronous or asynchronous. There exist powerful algorithms for software/hardware synthesis and verification. However, FSM models may have problems such as over-specification of implementation, incompact specification of numerical computation, and state explosion. To solve these problems, FSMs can be used together with other models of computation. A representative example is the Statecharts formalism [46], which extends conventional FSMs to have an instantaneous broadcast synchronous communication mechanism. Another example is the Codesign Finite State Machine (CFSM) model of computation [8]. A CFSM is an FSM extended with the support for data handling and asynchronous communication. In CFSM models, synchronicity (zero and infinite time) and asynchronicity (non-zero, finite, and bounded time) are combined in a globally-asynchronous and locally-synchronous way. A CFSM model preserves formality and efficiency in implementation.

Synchronous Reactive In the Synchronous Reactive (SR) [11] model of computation, time is abstracted as global discrete ticks. All events in an SR model are synchronized with ticks, i.e., computation is triggered by a global clock. Conceptually, an SR system takes zero time to react to events, i.e., computation is simultaneous and instantaneous, which is the underlying synchrony assumption. Unlike discrete time discussed later, a signal in an SR model does not need to have a value at every clock tick. Based on the SR model of computation, synchronous languages have been designed: Esterel [14][13], Lustre [45][19], Signal [64][44], Statecharts [46]. SR models have strong formal properties and many key questions are decidable. One of the nice properties is that buffer memory is bounded. The fixed point semantics of the SR model of computation only addresses the behavior at a clock tick. The behavior across clock ticks requires a clock calculus [96]. SR models are suitable for designing applications with concurrent and complex control algorithms.

Discrete Event The Discrete Event (DE) model of computation has an explicit notion of global time and communication is based on events consisting of a pair of value and time stamp. Typical examples that use DE are HDLs: Verilog [114] and VHDL [95]. In DE models, a global event queue is usually maintained according to the event stamps, which unfortunately leads to tight coordination. The discrete event semantics needs to deal with simultaneous events and Zeno conditions. Simultaneous events lead to a nondeterministic behavior and some simulators use delta delay to prevent nondeterminacy. For example, in Verilog, simultaneous events are handled nondeterministically while in VHDL, the notion of delta time is introduced to separate a sequence of two simultaneous distinct events so that they can be processed deterministically. A Zeno condition means that there may be

an infinite number of time stamps between two finite time stamps. In [66], it is shown that delta-causality guarantees the absence of Zeno conditions. The problem with DE models is that it may be difficult to maintain a globally consistent notion of time. To solve this problem, distributed discrete event modeling is presented in [88].

Dataflow The seminal work on Dataflow (DF) is the computation graphs given by Karp and Miller [59]. As a special case of the process networks, the dataflow model of computation [69] is based on asynchronous message passing. A dataflow network is a collection of functional nodes (commonly called actors) connected over unbounded FIFO queues. Stateless actors perform computation while unbounded FIFOs perform communication via sequences of tokens carrying values. A unique output sequence corresponds to a unique input sequence. At each time, one actor is fired. During firing, actors consume input tokens and produce output tokens. Actors can be fired only if there are enough tokens in the input queues. The key property of DF networks is that output sequences do not depend on the time of firing of actors. There are variant types of dataflow networks. Static dataflow procedure language is discussed in [30]. Synchronous (or better Static Schedulable) Dataflow (SDF) [68] [92] networks can be statically scheduled by solving the balance equations at compile time. In SDF models, an actor is executed when it is known to be fireable. There is no overhead due to the sequencing of concurrency and buffers can be sized statically. In Dynamic Dataflow (DDF) [3] models, firings are scheduled only at run time. In Boolean Dataflow (BDF) [15] and Integer Dataflow (IDF) [16] models, balance equations involving unknown production/consumption rates are solved symbolically so that data-dependent routing of tokens is supported. For all DDF, BDF, and IDF models, deadlock, memory

boundedness, and scheduling are undecidable. In summary, dataflow networks are a powerful formalism for specifying data-dominated systems. They are a partially-ordered model (no over-specification) and their execution is deterministic (independent of scheduling). DF models have explicit concurrency and may be used for simulation, scheduling, memory allocation, and code generation for digital signal processors. Similar to a graphical language, DF networks are easy to use. There exist powerful verification and synthesis algorithms. But efficient synthesis is only for restricted models and a DF model cannot describe reactive control due to blocking read.

Petri Nets Petri Nets (PN) are named after Dr. Carl Adam Petri, in honor of his pioneer work in communication with automata. As an asynchronous model, Petri nets [91] explicitly and graphically describe sequencing/causality, conflict/nondeterministic choice, and concurrency. Petri nets have nice properties: the behaviors are dependent upon the initial marking; the reachability problem with Petri nets is decidable; the behavioral property of boundedness implies that the number of tokens in any place cannot grow indefinitely; the liveness property signifies that any transition can become fireable from any marking; the conservation property means that the total number of tokens in the net is constant. Petri nets are added with the time notion in [56] to satisfy the timing constraints of communication protocol architectures for reliable conversion functions.

Continuous Time In the Continuous Time (CT) model of computation [74], communication is via continuous time signals that depend on the real numbers. A continuous time model describes Ordinary Differential Equations (ODE) and differential algebraic equa-

tions, which govern the operation of physical systems, e.g., analog circuits, chemical reaction processes, mechanical systems, etc. The ODEs may be solved by different numerical schemes such as the Trapezoidal Method, the Forward/Backward Euler Solver, the Runge-Kutta Solvers, etc.

Discrete Time Discrete Time (DT) [37] is an extension of the synchronous dataflow model of computation. It is very similar to the synchronous reactive model of computation with the exception that every signal has a value at every clock tick. On top of the desirable properties that synchronous dataflow models have, the DT model of computation is temporally causal, which requires that the outputs depending on the inputs are never produced before the time of the inputs.

Communicating Sequential Processes Hoare's Communicating Sequential Process (CSP) [50] is a rendezvous model, where concurrent processes communicate via the so-called rendezvous, a synchronous message passing mechanism. Between two communicating processes, the one that reaches the rendezvous communication point earlier will need to stall to synchronize with the other process.

Calculus of Concurrent Systems Similar to Hoare's Communicating Sequential Processes, Milner's Calculus of Concurrent Systems (CCS) [87] is another model of computation that is based on rendezvous.

Giotto The Giotto [47] model of computation defines a time-triggered programming language for implementing embedded control systems of hard real-time type. In a Giotto

model, all software tasks are periodically invoked according to their respective sampling periods. Together with the FSM model of computation, Giotto is useful for implementing modal models.

The Tagged Signal Model The tagged signal model [70] provides a very general conceptual framework for comparing and reasoning about models of computation. It provides a natural model for design refinement, which offers the possibility of type-system-like formal structures that deal with dynamic behavior in addition to static structures. The tagged signal model provides a mathematical denotational framework, within which the essential properties of models of computation can be analyzed and compared. With the tagged signal model semantics, three categories of abstract semantics can be defined: process networks type, firing type, and stateful firing type. Examples of concrete semantics conforming to the process networks abstract semantics are Hoare's CSP, Milner's CCS, and Kahn's process networks; examples conforming to the firing abstract semantics include DF, DE, and Giotto; and those that conform to the stateful firing abstract semantics include SR and CT.

1.2.2 Model Implementation Tools

After introducing models of computation, in this subsection existing implementation tools are discussed for some of them. There are two big categories of the implementation tools: from either academia or industry. In the following, the first five examples are from academia, which aim at a better research and development environment. Meanwhile, industry has paid a close attention to model-based design methodology and the last six are typical representatives.

Giotto Giotto defines two virtual machine platforms to implement embedded software consisting of hard real-time periodic tasks with deterministic mode changes. The Embedded Machine (E-Machine) [48] defines conditions that enable tasks while the Schedule Machine (S-Machine) [49] defines task switching for execution. Task enabling events include external interactions a task may have. Scheduling code can describe standard scheduling policies such as rate monotonic and earliest deadline first. Giotto compiler automatically targets the E-Machine and the S-Machine. To achieve data determinism, Giotto [47] delays committing output data on every connection, which means that there is one additional unit sampling delay form input to output in any communication.

Metropolis Metropolis [9], successor of Polis [8], is developed at the University of California, Berkeley. Metropolis provides a design environment for heterogeneous systems. In the Metropolis framework, the infrastructure consists of models of communication through abstract semantics, design methodologies at different abstraction levels, and communication refinement as well as base tools used for design imports, user interface, and simulation. In Metropolis, system functions are specified via a network of processes, where a process is a sequential function plus ports. Metropolis does not commit to any particular communication semantics. Ports are interconnected by communication media, which define the communication semantics. Examples of communication media include queues, shared memory, and so on.

Modelica Modelica [102] is developed and maintained by Modelica Association [4], a nonprofit organization. Modelica consists of three parts: object-oriented language, standard

libraries, and simulation environment. The underlying model of computation of Modelica is captured by a set of differential, algebraic, and discrete equations with real-time constraints. Modelica is primarily used for modeling large, complex, and heterogenous physical systems including mechanical, electrical, hydraulic, thermal, power, and control subsystems.

Ptolemy Ptolemy [52][35] is a research project at the University of California, Berkeley. The goal of the project is to research modeling, simulation, and design of heterogenous concurrent embedded real-time systems. Ptolemy is a framework that supports design of embedded software based on well-defined models of computation that govern the interaction among concurrent components. It started with static dataflow model of computation for digital signal processing and the current version of the project, Ptolemy II, supports many of the models of computation discussed above.

The Generic Modeling Environment The Generic Modeling Environment (GME) [65][1] is developed at Vanderbilt University. The GME is a model-integrated program synthesis tool and is featured with configurable modeling, meta-modeling, and model visualization. The configuration of the GME is via metamodels that specify the modeling language of the application domain. The metamodeling language of the GME is based on the Unified Modeling Language (UML) and Object Constraint Language (OCL). The GME supports design reuse by metamodel composition. Decorator interfaces of the GME enables customization of model visualization.

LabVIEW LabVIEW [27] is developed at National Instruments Corporation and the best supported underlying model of computation is synchronous dataflow. LabVIEW can

be viewed as a high level graphical programming language (known as G code). LabVIEW combines design, simulation, prototyping, and deployment of embedded software in a single graphical programming tool-chain so that system development time is reduced. Via using the LabVIEW graphical development environment, it is easy to program systems with heterogenous executing devices quickly and reliably.

OPNET Developed by OPNET [55], OPNET Modeler is used for modeling and simulation of networks, devices, and protocols. With discrete event as the underlying model of computation, OPNET Modeler is designed to support hierarchical object-oriented modeling to facilitate both research and development of communication networks. From the highest to the lowest level, the hierarchical modeling architecture is network (e.g., LAN), node (e.g., devices such as routers), and process (e.g., protocols such as IP/TCP).

SCADE SCADE [34] represents Safety Critical Application Development Environment. It is the visual editor for Lustre, one of the popular synchronous languages. Lustre and SCADE are developed for safety-critical embedded software such as avionics software, where tasks are aligned to a master clock and its sub-clocks. Consistency and deadlock are checked via clock calculus [96]. The current version of SCADE is Version 6, which supports a unified design, modeling, and code generation environment.

Signal Processing Worksystem/Designer Signal Processing Worksystem (SPW) is developed at Cadence and later sold to CoWare. CoWare renamed SPW to Signal Processing Designer (SPD) [54]. The underlying model of computation of SPD is synchronous dataflow. SPD is an integrated environment used for system-level design, modeling, sim-

ulation, and implementation of complex digital signal processing systems such as wireless and multimedia products.

Simulink Simulink with Real-Time Workshop (RTW) [79] is developed at The MathWorks Incorporation for embedded control software and has been widely used in the automotive industry. The underlying models of computation that Simulink supports include synchronous reactive, continuous time. For models with the synchronous reactive semantics, Simulink achieves data determinism with snapshot of inputs and delayed commit of outputs. Note that Simulink introduces a unit delay only on slow to fast sampling rate changes.

Stateflow Stateflow [81] is also developed at The MathWorks Incorporation. It is a powerful model-based design and development tool. As its name implies, the underlying model of computation is finite state machine. Stateflow is good for designing complex control and supervisory logic systems.

1.3 Scheduling and Communication Mechanisms

For model-based software design, upon implementation, the functionality of each model block is accomplished by a run-time task. There are two options to implement a multi-rate model on a single processor system. In a single task implementation, all the functionality of the specification is implemented by a single run-time task (or executive), running at the base rate of the system. Such an implementation is easier to construct, but often characterized by poor resource utilization. A multi-task implementation typically

uses one task for each execution rate, and possibly more. All tasks are executed under the control of an operating system. Each implementation task τ_i is characterized by a set of parameters: priority π_i, period (periodically activated) or minimum inter-arrival time (sporadically activated) T_i, worst-case computation time C_i, worst-case response time R_i, and relative deadline D_i. The task execution time is usually finite and a task can be preempted according to its priority. Schedulability of tasks requires that $R_i \leq D_i$. Multi-task implementations allow for a much better schedulability. However, due to preemption, there may exist problems with inter-task communication, e.g., nondeterministic communication or data integrity problems. To address these problems, communication protocols have been proposed in literature. In the rest of this section, scheduling policies and communication mechanisms are discussed.

1.3.1 Scheduling Policies

Given a system specification consisting of a set of concurrent functional blocks, a software implementation, consisting of a set of software tasks, can be obtained by software synthesis tools. In a software synthesis process, there are two sub-problems: automatic code generation for each task and dynamic scheduling of the generated tasks. Software synthesis aims at minimizing real-time scheduling overhead. There are three classes of scheduling: static (schedule completely determined at compile time), dynamic (schedule determined at run time), quasi-static (most of the schedule computed at compile time, some scheduling decisions made at run time, but only when necessary). Among them, dynamic scheduling is usually used for real-time controls featured with preemption and suspension, static scheduling is good for data processing, and quasi-static scheduling is primarily for

data-dependent controls. In the following, real-time scheduling and quasi-static scheduling policies are summarized.

Real-Time Scheduling A real-time system implemented as software consists of a set of tasks. Tasks are enabled by repeating events in the environment and their execution must satisfy timing requirements. Real-time scheduling is difficult. There are various types of scheduling policies proposed for real-time tasks in literature [18], e.g., preemptive and non-preemptive policies. Under preemptive scheduling, the running task can be preempted by another active task with a higher priority, i.e., the enabled task with the highest priority is scheduled for execution at any time; while under non-preemptive scheduling, the running task cannot be interrupted after it is dispatched, i.e., once a task is chosen to be executed, it will run to completion even if some tasks with a higher priority become enabled.

In terms of the parameters on which scheduling decisions are based, real-time scheduling policies can be static or dynamic. For example, for priority-based scheduling, priorities can be either static or dynamic. Static priorities are assigned offline while dynamic priorities may change at run time. When there are multiple tasks enabled, the task with the highest priority is executed.

In terms of the schedule generation time, there are offline and online scheduling policies. An offline scheduling algorithm is executed on all the tasks in the system before actual task activation while an online scheduling algorithm is executed at run time upon new task arrival and running task termination. Offline scheduling is static and typical examples are Round-Robin (RR) scheduling and static cyclic scheduling. A Round-Robin schedule picks a task order and executes them forever in that order. A static cyclic executive [75]

picks a task sequence and executes that sequence forever. Online scheduling is dynamic. Typical online scheduling policies are priority-based. For online scheduling, the tasks can be either preemptive or non-preemptive. Compared with online scheduling, it is easy to construct and analyze an offline schedule. Without preemption and priority calculation, the overhead with offline scheduling is very low. But it may delay service to more urgent tasks. Liu-Layland [73] considers systems consisting of periodic tasks with fixed execution time. It is assumed that the worst-case response time of a task is smaller than the period of the task. It is proved that with preemptive static priority scheduling, the critical instant occurs when a task is enabled at the same time as all higher priority tasks. For Rate Monotonic (RM) [71] scheduling, a higher priority is assigned to a task with a shorter period. It is demonstrated that scheduling policies based on static priorities can schedule systems with a utilization less than 0.69. If scheduling based on static priority does not work, online scheduling based on dynamic priority can be used. The typical examples of dynamic scheduling policy is Earliest Deadline First (EDF) [73][31]. Its main idea is to assign the highest priority to the task with the closest deadline. It is shown that EDF can schedule any set of tasks with a utilization less than 1.

Quasi-Static Scheduling Petri nets model of computation provides a unified model for mixed control and data processing specifications. As presented in Section 1.2.1, most of the properties of Petri nets are decidable. Quasi-Static Scheduling (QSS) [29][28] of embedded software is based on Free-Choice Petri Nets (FCPN), where free choice means that the choice depends on the token value rather than the token arrival time. For FCPNs, schedulability can be checked before code generation. QSS finds one schedule for every conditional branch

at compile time while it chooses one of these schedules according to the actual value of the data at run time. QSS minimizes run-time overhead with respect to dynamic scheduling by automatically partitioning the system functions into a minimum number of concurrent tasks.

1.3.2 Communication Mechanisms

There are different ways to classify inter-task communication, for example, blocking versus non-blocking and synchronous versus asynchronous. Communication is non-blocking if the communication call may return before the communication operation completes, otherwise it is blocking. Specifically, for blocking read, processes cannot test for emptiness of input and must wait for an input to arrive before proceeding. For blocking write, processes must wait for a successful write before continuing. Both Milner's Calculus of Communicating Systems and Hoare's Communicating Sequential Processes use blocking write/read. Shared variables are examples of non-blocking write/read. FIFOs used in CFSMs and SDL (Specification and Description Language) are examples of non-blocking write and blocking read.

Communication is asynchronous if its execution proceeds at the same time with the execution of the programm, otherwise it is synchronous. A typical example of synchronous message passing is rendezvous and a representative example of asynchronous communication is asynchronous buffered communication. For rendezvous, no space is allocated for communication data. If one of the communicating blocks (either producer or consumer) reaches the point of communication, it stalls for synchronization until the other one is also ready to exchange data, i.e., the read and the write occur simultaneously.

Note that a non-blocking communication is not necessarily asynchronous. Both asynchronous and non-blocking communication routines need a separate wait or test call to make sure that the communication has completed and that resources can be safely reused [22]. Reading can be consumable or non-consumable, i.e., the result of each write can be read at most once or several times. For shared memory, multiple nondestructive reads are possible and writes delete previously stored data.

Communication can be either lossless or lossy. Events or tokens may be lost in lossy communication. For multi-rate concurrent systems, buffers are usually used to adapt different rates of the sender and the receiver(s). To achieve lossless communication with bounded memory, an appropriate buffer sizing is needed, otherwise it may need to block the sender. As a special example, FIFO communication buffers can be either bounded (e.g., CFSMs) or unbounded (e.g., SDL, Khan Process Networks, Petri Nets).

Any (real-time) communication of data between concurrent tasks that cannot be made atomic at the hardware level must be implemented by using some communication protocol to protect against access of the reader(s) while a write is in progress or against modification of the data by the writer task while a read is in progress. In the rest of this subsection, different implementation mechanisms in terms of blocking and non-blocking are discussed. A blocking mechanism is also called lock-based and usually requires semaphore (or spin lock) support from a real-time operating system. Non-blocking mechanisms do not need a support of locking and can be further sub-categorized as lock-free and wait-free as discussed below. Note that, though communication is emphasized in the discussion, these mechanisms are actually general models for resource sharing.

Blocking Mechanism For lock-based schemes, when a task wants to access the communication data while another task holds an exclusive lock, it blocks (usually on a semaphore), and releases the CPU voluntarily. When the lock is released, the task is restored in the ready state and can safely access the data. Lock-based mechanisms guarantee exclusive access to shared resources. Semaphores [112] are the most common method for locking a shared resource. Typical problems associated with resource sharing are priority inversion and deadlock. A radical solution consists in avoiding preemption during the execution of the critical sections, but this method affects all tasks (not only those using the given resource) and is only effective for very short critical sections. Different protocols have been developed to deal with this problem. The Priority Inheritance Protocol (PIP) [111] limits the priority inversion caused by resource constraints to a known upper bound in case critical sections are not nested. In the PIP, the blocking task inherits the highest priority of the tasks that are waiting for the resources it is currently holding, therefore avoiding preemption from tasks with a medium priority. The PIP allows to bound the priority inversion in most cases, but has a costly implementation and does not prevent chained blocking and even deadlock in case of nested critical sections. The Priority Ceiling Protocol (PCP) [111][26] avoids these two problems by extending the PIP. The PCP allows to bound the worst-case blocking time caused by priority inversion to a known value (one critical section, better than the PIP bound) in all cases, and avoids deadlock. The PCP introduces a priority ceiling for each semaphore equal to the highest priority of any task that can possibly lock it. A task is allowed to enter a critical section only when its priority is higher than the priority ceilings of all semaphores currently locked by other tasks. When a task blocks another one with a

higher priority, the blocking task gets the priority of the blocked task when it is in the critical section. The Immediate PCP (IPCP) and the Stack Resource Policy (SRP) [7], which is its generalization to dynamic priority schemes, further extend the PCP by supporting multi-unit resources and sharing of the application run-time stack. Compared with the PCP, a preemption level is statically introduced for each task. Each resource is assigned a current ceiling equal to the highest preemption level and the system ceiling is defined to be the maximum of the ceilings of all the locked resources at any given time. When a task uses a resource, its preemption level is immediately raised to the ceiling level of the resource even if it is not blocking any task, and the maximum between its preemption level and its priority becomes its execution priority. This means that any task that can be possibly blocked by it is not even allowed to start execution until the resource is unlocked. The worst-case bound on the blocking time is the same as that in the PCP, but the implementation of these protocols is considerably easier.

Non-blocking Mechanism The non-blocking mechanism can be either lock-free or wait-free. For a lock-free scheme, when a reader wants to access the communication data, it does so without blocking. At the end of the operation, it checks the consistency of the data. If it realizes there was a possible concurrent operation by the writer and the possibility of having read an inconsistent value, the reader task performs the read operation again. By leveraging the timing properties of tasks, the number of retries can be upper bounded.

A better non-blocking scheme is wait-free. Under the wait-free mechanism, readers and writers are protected against concurrent access to data by means of replication of the communication buffers and by leveraging information about the points in time when

they are going to access the resource or possibly other information that constrains the access to the resource (such as the priority or other scheduling-related information). Wait-free communication protocols have been mostly researched from the perspective of the programmers of concurrent real-time applications, interested in preserving the consistency of the data. The corresponding communication semantics is the so-called freshest value at execution time, meaning that each reader always obtains the latest data written by the writer task into the channel. Most of the protocols presented in the literature try to optimize the buffer size based on assumptions on tasks.

In the sequel, typical wait-free protocols with the execution-time freshest value semantics are reviewed. The first category of the methods provides buffer sizing and access procedures based on the number of readers that ensures mutual exclusion access. In [24], an asynchronous protocol is proposed to preserve data consistency for communication between a single writer and a single reader running on a shared-memory multiprocessor. With no assumption on task priorities and periods, three buffers are needed: one for the data being read, one for the data last written (current) by the writer, and another one for the writer to write a new data item into when the latest written buffer has not been read yet. To achieve data integrity, a hardware-supported Compare-And-Swap (CAS) instruction is used to atomically assign the reading position in the buffer array to reader tasks and to update the pointer to the last written value. This three-slot asynchronous protocol is extended to systems with single writer and multiple readers in [23]. The multi-reader asynchronous protocol needs $(\mathtt{N}+2)$ buffer slots, where $\mathtt{N}$ is the number of the readers in the system.

The other method provides buffer sizing and buffer indexing by using the Tempo-

ral Concurrency Control (TCC) that ensures writer and reader tasks never access the same data item at the same time. The size of the buffer can be computed by upper bounding the number of times the writer can produce new values while a given data item is considered valid by at least one reader. This concept is first introduced in [61] and [25], assuming as the data validity time the worst-case execution time of a reader. In [61], the Non-Blocking Write (NBW) protocol is presented for a single-writer multiple-reader system executing with a priority-based preemptive scheduling from a real-time operating system on a distributed real-time system consisting of a set of nodes connected by a broadcast communication channel. The writer has its own communication controller, therefore the writer task cannot be preempted or preempt other reader tasks and thus the non-blocking property of the writer is achieved. In [25], a timing-based wait-free mechanism called asynchronous circular buffering protocol is proposed for single-writer multiple-reader systems running on a shared-memory multiprocessor system with a single global clock. Priority-based preemptive scheduling on the same processor is assumed. It is further supposed that the worst-case response time is smaller or equal to the deadline and the deadline is smaller than or equal to the period. Under this protocol, data sharing is implemented through a sequential algorithm using a circular buffer and the size of the circular buffer is configured through the timing property of the set of tasks.

A combination of the previous two buffer sizing methods may be used to obtain a better buffer size through exploiting the temporal characteristics of real-time tasks. A transformation mechanism used to optimize buffer sizing based on [23] and [25] is proposed in [51] for implementations with the execution-time freshest value semantics. Based on

timing properties, the reader tasks are categorized as either fast or slow. The fast readers leverage the lifetime-based bound of the NBW [25], and the slow ones leverage the bound based on the maximum number of active reader instances as in [23]. Overall the space and temporal performance are both improved. The transformation is applicable to protocols for both single processor and multiprocessor systems.

1.4 Goal of Dissertation

Among those models of computation that are surveyed in this chapter, the synchronous reactive model of computation has strong formal properties, e.g., decidable termination and execution with bounded memory, which can be used to verify synchronous reactive systems. This model of computation is good for specifying periodic real-time tasks. There has been a steadily increasing interest in this model in both academia and industry. Synchronous reactive models have been traditionally used in hardware logic design and more recently for modeling control-dominated embedded applications. The synchronous reactive zero-time semantics is very popular because of the availability of the tools for simulation and formal verification of system properties. When implementing a high-level model into code, it is important to preserve its semantics, in order to retain the simulation and verification results. *However, this requires that the run-time behavior when the functions are implemented by a program with a finite execution time and possibly subject to preemption is provably equivalent to the synchronous reactive model with zero execution time and no preemption.* In general, defining such a provably correct implementation is nontrivial.

In the following, possible solutions based on the synchronous reactive commu-

nication mechanisms are discussed. There usually exists a time-space tradeoff between different communication mechanisms. Lock-based mechanisms are space-efficient because data buffers (usually in Random Access Memory (RAM)) do not need to be replicated, however, they typically require complex management procedures (usually in Read-Only Memory (ROM)). As discussed earlier, especially, they may introduce large blocking times. Lock-free mechanisms improve time efficiency, but they still suffer from long retry times in the worst case. Wait-free mechanisms are the most time-efficient, but may require a higher memory cost for the replicated buffer space.

In spite of the thread of research on wait-free communication with the freshest value semantics, this non-blocking communication has been rediscovered in the context of model-based software development and reformulated with the goal of finding communication mechanisms (typically with a minimum buffer size) for implementing a synchronous reactive semantics in the communication of a single writer with multiple reader tasks/blocks. The problem is somewhat different and yet both the optimization problems and the protocol solutions show many similarities that are not only worth a comparative study, but can possibly lead to a general theory and further improvement with respect to the state of the art. The communication mechanisms that provide data consistency with the freshest value semantics at task execution time cannot guarantee time determinism. The value that is read by a reader task depends upon the scheduling of the tasks, that is, on their execution times and the possible occurrence of preemption. For many control applications, this is not a concern, given the robustness of the control algorithm with respect to time delays. In other cases where the application may be sensitive to the ensuing jitter, the implementation

needs to satisfy more stringent requirements. The correct implementation of a synchronous reactive semantics for the above cases requires assigning a buffer index to the reader, which maintains the same value during the reader's execution. Reduction of memory cost due to communication is an open research issue. Hence this dissertation particularly focuses on the synchronous reactive model of computation and attempts at providing new results in this context.

The rest of the dissertation is structured as follows. Chapter 2 surveys the state of the art on synchronous reactive communication. Based on the limitations associated with existing solutions, it defines and motivates the problems that are going to be addressed. Chapter 3 introduces the platform-based design implementation technology for synthesizing synchronous reactive communication protocols and customizing a supporting real-time operating system automatically. Chapter 4 generalizes synchronous reactive communication to deal with arbitrary link delays and multiple activation instances per task. It provides tight bounds on communication buffers. Chapter 5 presents portable and efficient implementations for two synchronous reactive communication protocols. Furthermore, it provides automatics code generation support to complete the synchronous reactive model-based development flow. Based on the results from Chapter 5, Chapter 6 investigates memory optimization through automatically choosing proper communication protocols between a writer and its readers. Finally, Chapter 7 concludes the dissertation with summarizing contributions and pointing out directions for possible future work.

Chapter 2

State of the Art and Problem Statement

Starting from this chapter, the dissertation focuses on synchronous reactive communication between concurrent tasks that implement synchronous reactive model specifications. First of all, the synchronous reactive communication semantics is formally presented through defining the communicating writer instance for a reader. Then, current status of synchronous reactive communication implementation support is discussed for different execution platforms ranging from uni/multi-processor to distributed architectures. The second part of this chapter further analyzes current solutions and pinpoints existing limitations. Based on these, the dissertation defines the interesting problems (communication theory generalization, semantics preserving implementation, and memory optimization) that need to be addressed for model-based development of embedded software under the synchronous reactive semantics. Finally, it further motivates the problems that are solved in this study.

2.1 State of the Art of SR Communication

In this section, the synchronous reactive communication semantics is formally defined. Then, the issues that are associated with communication between concurrent tasks that implement an SR model specification are addressed. The SR communication semantics requires both data integrity and data determinism. To prevent nondeterministic communication and communication data from being corrupted, different communication schemes have been proposed. In the following, it summarizes the current status of semantics-preserving implementation of the synchronous reactive communication.

2.1.1 Synchronous Reactive Communication Semantics

In a synchronous reactive model, execution takes zero-time. For each writer or reader $\mathtt{b_i}$, $\mathtt{b_i(j)}$ and $\mathtt{a_i(j)} \in \mathbb{R}^+$ represent its $\mathtt{j^{th}}$ instance and its corresponding activation time. Under the synchronous reactive semantics, given that the execution time is zero, the activation time $\mathtt{a_i(j)}$ captures also the start time and the finish time of the same instance of $\mathtt{b_i}$. Given time $\mathtt{t} \geq \mathtt{0}$, define supremum $\zeta_\mathtt{i}(\mathtt{t})$ to be the number of times that $\mathtt{b_i}$ has occurred up to $\mathtt{t}$, i.e.,

$$\zeta_\mathtt{i}(\mathtt{t}) = \sup\{\mathtt{n} | \mathtt{a_i(n)} \leq \mathtt{t}\},$$

where the sup of an empty set is defined to be zero, so that if $\mathtt{b_i}$ has not occurred up to $\mathtt{t}$, then $\zeta_\mathtt{i}(\mathtt{t}) = \mathtt{0}$.

Let the input of $\mathtt{b_i}$ be $\mathtt{i_i}$ and output be $\mathtt{o_i}$. If $\mathtt{b_j}$ and $\mathtt{b_i}$ are in an input-output relationship ($\mathtt{b_j}$ takes the output of $\mathtt{b_i}$ as input) and $\mathtt{b_j}$ is of type feedthrough, there will be a communication link, denoted as $\mathtt{b_i} \rightarrow \mathtt{b_j}$, between them. Consider the case where the

link between $\mathtt{b_i}$ and $\mathtt{b_j}$ has no delay. The synchronous reactive semantics specifies that the input of the $\mathtt{n^{th}}$ occurrence of $\mathtt{b_j}$ is equal to the output of the last occurrence of $\mathtt{b_i}$ before $\mathtt{b_j(n)}$, that is,

$$\mathtt{i_j(n)} = \mathtt{o_i(m)}, \text{ where } \mathtt{m} = \zeta_\mathtt{i}(\mathtt{a_j(n)}).$$

If $\mathtt{b_i}$ has not occurred yet, then $\mathtt{b_i(0)}$ is used. If the link carries a unit delay, as indicated by $\mathtt{b_i} \overset{-1}{\rightarrow} \mathtt{b_j}$, the synchronous reactive communication semantics defines

$$\mathtt{i_j(n)} = \mathtt{o_i(m)}, \text{ where } \mathtt{m} = \mathtt{max}\{\mathtt{0}, \zeta_\mathtt{i}(\mathtt{a_j(n)}) - \mathtt{1}\}.$$

2.1.2 Synchronous Reactive Communication Implementation

Upon implementation, a synchronous reactive model is mapped to underlying executing platforms. However, notice that there is a fundamental difference between the simulation time of an SR model and the run time of the model implementation, i.e., zero execution time during simulation and finite execution time at run time. As presented in [122], due to preemption, this may lead to possible problems (nondeterministic communication or data integrity) with data transfers when buffers are indexed at run time. A multi-task implementation may raise issues with respect to the preservation of the behavior under the zero execution time assumption.

The top of Figure 2.1 illustrates the execution of a pair of blocks with the synchronous reactive zero-time semantics. The horizontal axis represents time. The vertical arrows capture the time instants when the blocks are activated and compute their outputs from the input values. Note that, as shown on the top of the figure, it is $\mathtt{i_j(n)} = \mathtt{o_i(m)}$ at simulation time, which is what the synchronous reactive communication semantics requires.

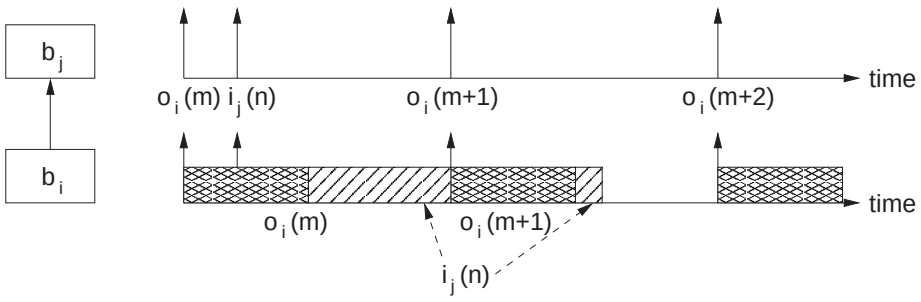

Figure 2.1: How Preemption May Change Values Read by Reader Tasks

The bottom of Figure 2.1 shows the possible problems with data transfers in a real-time multi-task implementation. Assume that a fast writer $\mathtt{b_i}$, implemented by a high priority task, communicates with a slow reader $\mathtt{b_j}$. The writer finishes its execution with producing output $\mathtt{o_i(m)}$ and the reader is executed right after. If the reader performs its read operation before the preemption by the next writer instance, then $\mathtt{i_j(n)} = \mathtt{o_i(m)}$. Otherwise, it is preempted and a new instance of the writer produces $\mathtt{o_i(m+1)}$. In case the read operation had not been performed before, the reader task reads $\mathtt{o_i(m+1)}$, in general different from the value $\mathtt{o_i(m)}$. This is known as nondeterministic communication. Even worse, in case the signal value is not read and written atomically, there is a finite probability that the writer task/block $\mathtt{b_i}$ preempts the reader task/block $\mathtt{b_j}$ while a read is in progress, resulting in an inconsistent value, namely a data integrity problem. The problems arise because each functional block is implemented at run time in the context of a task, that is, the code implementing its functionality, executing on a Central Processing Unit (CPU).

To solve these problems, different mechanisms have been proposed. In the following, some typical communication protocols under the synchronous reactive semantics are reviewed. As mentioned earlier, wait-free schemes are the preferred choice for the im-

plementation of semantics-preserving communication protocols due to their simplicity and efficiency.

A one-to-one communication mechanism that preserves the SR semantics under deadline monotonic static priority based preemptive scheduling and earliest deadline first scheduling on a single processor platform with the support of a real-time operating system has been presented in [109] and [116], respectively. They assume that the deadline is smaller than the sampling period. A two-place buffer and two buffer indices are required. Due to the single processor assumption, given that the code that updates the index variables is executed inside the kernel, at task activation time, there is no need for any hardware-level mechanism to ensure atomicity when swapping buffer pointers or comparing state variables. Figure 2.2 shows the high level code of the Double Buffer (DoB) mechanism. Clearly, the DoB mechanism takes advantages of the priority information of the writer and reader. The buffers are initialized with a given value and the writing index initially points to the first buffer. The reading index of a lower priority reader is also initialized because the assignment of the writing index depends on it. At activation time, the writing index of a lower priority

Data Structures	`boolean wrtIdx, rdIdx;`		`message buf[2];`
		Low to High Priority	High to Low Priority
	initialization	`wrtIdx = 0;` `buf[0] = buf[1] = ···`	`rdIdx = wrtIdx = 0;` `buf[0] = buf[1] = ···`
Writer	*activation*	`wrtIdx = !wrtIdx;`	`if (rdIdx == wrtIdx)` `  wrtIdx = !wrtIdx;`
	execution	`buf[wrtIdx] = ···`	
Reader	*activation*	`rdIdx = !wrtIdx;`	`rdIdx = wrtIdx;`
	execution	`··· = buf[rdIdx];`	

Figure 2.2: The Double Buffer Mechanism

writer is always toggled while the writing index of a higher priority writer is toggled only when the reading index and the writing index are equal. The reading indices of a higher and a lower priority reader are assigned as the negated value and the same value of the writing index, respectively. At execution time, the writer and reader directly writes into and reads from the slots that have been assigned at their activation times, respectively.

Communication in a Simulink SR model is defined to be point to point, i.e., between single writer and single reader running on a single processor. To achieve protected and deterministic communication under the SR semantics in Real-Time Workshop [77], Rate Transition Buffers (RTB) are inserted between blocks with different sampling rates in a Simulink SR model. RTBs work as a unit delay block for slow to fast sampling rate transitions and a zero-order hold block for fast to slow sampling rate transitions. The priority assignment used in Simulink is rate monotonic. RTBs run at the slow sampling rate but with a priority equal to that of the fast one. Models with the insertion of RTBs preserve the synchronous reactive semantics at real time. Note that the Rate Transition Buffering scheme is actually the double buffer mechanism implemented at the application level.

In the general case of multiple readers, wait-free mechanisms under the SR communication semantics can be constructed by leveraging two properties of the relationship between the writer and its readers. Similar to the wait-free mechanisms under the execution-time freshest value semantics in Section 1.3.2, the first method consists in computing an upper bound for the maximum number of buffers that can be used at any time by reader tasks. In [113], the writer communicates with a unit delay with `M` reader blocks with higher priorities and `N` reader blocks with lower priorities. The bound is defined for the case of

communication links with a unit delay, under the assumption that each task instance terminates before its next activation event. The proposed communication scheme is called the Dynamic Buffering Protocol (DBP). When unit delays are allowed on links, $(\mathtt{N}+2)$ buffers are demonstrably sufficient, where one buffer guarantees that the writer can safely update the latest data, one is for the higher priority reader tasks that need to access the previously written data, and `N` slots are for the lower priority reader tasks. In the context of single processor execution platform, according to the logic outlined in [113], the high level pseudo-code of the Dynamic Buffering (DyB) mechanism is defined in Figure 2.3. The data structures consist of an array `Buf[]` of buffers, an array `Read[]` containing one entry for each reader task with a lower priority, and an array `HPR[]` containing one entry for each reader task with a higher priority. The entry contains the index of the buffer item used by the corresponding reader or the keyword `FREE`. Furthermore, two other variables contain

<table>
<tr><td>Data Structures</td><td colspan="2">char cur, prev;
message Buf[N+2];</td></tr>
<tr><td rowspan="2">Writer</td><td>/* activation time */
prev = cur;
cur = FindFree();</td><td>/* execution time */
···
Buffer[cur] = ···</td></tr>
<tr><td colspan="2">FindFree() {
  if (prev ≠ j ∈ [1,N+2]) ∧ (∀i ∈ [1,N] Read[i] ≠ j)
  return j;
}</td></tr>
<tr><td>Lower Priority Reader</td><td>/* activation time */
if (delay[i])
  Read[i] = prev;
else
  Read[i] = cur;</td><td>/* execution time */
··· = Buffer[Read[i]];
···
/* termination time */
Read[i] = FREE;</td></tr>
<tr><td>Higher Priority Reader</td><td>/* activation time */
HPR[i] = prev;</td><td>/* execution time */
··· = Buf[HPR[i]];</td></tr>
</table>

Figure 2.3: The Dynamic Buffering Mechanism

the index of the latest written entry, `cur`, and the previous one, `prev`. The code implementing the operations of the writer and reader tasks is partly executed by the kernel at task activation time and partly by the tasks at execution time. With the assumption that the deadline of each task is equal to its period, all tasks complete before their deadlines, and hence only one task instance is active at any time. The authors of [113] demonstrate how the `M` high priority readers always use only one buffer (the one identified in the code by the `prev` index) and all the others require, in the worst case, a total of $(N + 1)$ positions.

The other method provides buffer sizing and access procedures by guaranteeing that writer and reader tasks never access the same data item at the same time. Similar to the TCC concept used for buffer sizing with the execution-time freshest value semantics in [25] and [61], the size of the buffer can be computed by upper bounding the number of times the writer can produce new values while a given data item is considered valid by at least one reader. This idea is also used in [10] for buffer sizing with no link delay while preserving the SR semantics. It assumes that the worst-case response time can be bigger than the sampling period. The offset, $o_{i,j}(m)$, is introduced between b_j and b_i as $o_{i,j}(m) = a_j(m) - a_i(n)$, where $n = \sup\{k | a_i(k) \leq a_j(m)\}$. Clearly, the largest value of $o_{i,j}$, denoted by $O_{i,j}$, is always smaller than the period of b_i, i.e., $O_{i,j} < T_i$. Define lifetime to be

$$l_j = O_{w,j} + R_j. \tag{2.1}$$

According to the TCC, the number of buffer items,

$$NB_{w,j} = \left\lceil \frac{l_j}{T_w} \right\rceil, \tag{2.2}$$

suffices for all the readers with lifetime $\leq l_j$. Implementation to single processor, multi-

processor, and distributed execution platforms are discussed, but no concrete tagging functions (i.e., indexing procedures) are given,

A buffer size upper bound is defined for SR semantics preserving communication in [10]. Similar to the buffer optimization in communication with the execution-time freshest value semantics in [51], the buffer bound can be improved by partitioning the reader tasks in two groups: fast and slow. The $\mathtt{NR}_\mathtt{w}$ readers of $\tau_\mathtt{w}$ are sorted by nondecreasing lifetime, so that $\mathtt{l_i} \leq \mathtt{l_{i+1}}$. The bound based on the data lifetime is used for the fast readers, and the bound based on the number of reader instances is used for the remaining slow readers. Let $\mathtt{j}$ be the partitioning index of the two groups. The reader tasks with index $\mathtt{i} \leq \mathtt{j}$ belong to the fast reader group while those with index larger than $\mathtt{j}$ fall into the slow reader group. Once $\mathtt{j}$ is chosen, the bound on the buffer size provided in [10] is

$$\mathtt{NB_w} = \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \sum_{\mathtt{i=j+1}}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{l_i}}{\mathtt{T_{r_i}}} \right\rceil, \tag{2.3}$$

where the first term represents a buffer shared among all consumers with $\mathtt{l_i} \leq \mathtt{l_j}$ and the second term is computed based on the number of activations of reader tasks inside the lifetime. In [10], the index $\mathtt{j}$ that defines the two groups is proposed to be computed as

$$\mathtt{j} = \max \left\{ \mathtt{i} \middle| \left\lceil \frac{\mathtt{l_i}}{\mathtt{T_w}} \right\rceil \leq \sum_{\mathtt{k=1}}^{\mathtt{i}} \left\lceil \frac{\mathtt{l_k}}{\mathtt{T_{r_i}}} \right\rceil \right\}. \tag{2.4}$$

It is difficult to implement synchronous models on distributed execution platforms. In [20], a layered approach is proposed to map synchronous models onto Time-Triggered Architectures (TTA), which is introduced in [62] as a strictly synchronous architecture. From high to low level, the layers are Simulink modeling and simulation, SCADE/Lustre programming and validation, and execution on TTAs. The TTA for distributed real-time

systems targets safety-critical applications such as automobile. A TTA-node consists of a communication controller and a host computer, between which the communication network is a strict data-sharing interface. To relax the strictly synchronous constraints from the TTA, Loosely Time-Triggered Architectures (LTTA) are introduced in [12], where clocks are periodic but not synchronized. A protocol consisting of a coherent system of logical clocks on LTTAs is proposed to guarantee synchrony. Implementation of synchronous model specifications on LTTAs is discussed in [115]. To accomplish the mapping from model specification to execution architecture, the synchronous model is first transformed to an intermediate model, the Finite FIFO Platform (FFP), which is similar to Kahn's Process Networks except that the queues are finite, and then the FFP is implemented on top of an LTTA.

2.2 Problem Statement

This dissertation focuses on synchronous reactive model-based design for embedded real-time systems implemented as software. There is no unified comprehensive article that addresses this problem thoroughly from model specification down to code generation. To lower the complexity of the problem, among the wide range of the target execution platforms, architectures with a single processor are considered. In the rest of this section, limitations of the state of the art of implementation of synchronous reactive communication on single CPU is discussed. Finally those problems that are going to be solved in this dissertation are presented and further motivated.

2.2.1 Generalization, Implementation, and Optimization

Along the line of research on the state-of-the-art synchronous reactive communication provided by [109][116][77][113][10], there are interesting research questions that are worth of more attention. First of all, it is desirable to generalize the state of the art to handle more general scenarios, which eventually subsumes the existing synchronous reactive communication protocols. It is well known that memory is expensive and scant for embedded real-time systems. It is beneficial to study the possibility of achieving a lower/tighter bound on communication buffers. Secondly, it is valuable to investigate how to preserve the SR semantics when implementing SR models down to code, how to achieve portable implementation for design reuse, how to support efficient implementation at the kernel level, and what supports are needed from real-time operating systems for semantics preservation. Design automation is playing a more and more important role for embedded system design. It is interesting to study how to automate the whole process of implementing SR models onto single processor execution platform with preserving the semantics of the synchronous reactive model of computation. Finally, the behavior and properties of model blocks may be quite different and a particular communication protocol may not provide a memory-efficient buffering implementation. It is important to study the possibility of improving the state of the art to save more memory on buffer consumption through utilizing different communication protocols between a writer and its readers. This dissertation attempts to answer all these questions.

2.2.2 Limitations of Current Solutions

As discussed in Section 2.1.2, the double buffer mechanism [109][116] supports the synchronous reactive communication semantics. The Rate Transition Buffer blocks [77] used in Simulink from The MathWorks is the best known industrial support for preserving the synchronous reactive communication semantics via the double buffer mechanism. As presented in Section 2.1.2, in Simulink, communication is defined to be one to one, which means that a pair of buffers are needed for each pair of communicating blocks under the double buffer communication scheme. This implies that two readers can never share the same buffer and hence there would be no opportunity for memory optimization. Since the RTB is implemented at the application level, in order to preserve both data integrity and data determinism, Simulink requires that the communicating blocks must be periodic and their sampling periods must be harmonic, which means that one is an integer multiple of the other. Furthermore, to maintain the sampling period harmonicity, the writer and its reader tasks must be activated with the same scheduling phase.

As shown in Section 2.1.2, the dynamic buffering mechanism [113] is defined between a writer and its multiple readers and is a relatively general communication protocol with the synchronous reactive semantics. But still it does not answer how to size the communication buffers for scenarios where there are arbitrary communication link delays or there are multiple instances per task in the system.

As presented in Section 2.1.2, the optimization work based on a hybrid communication protocol given in [10] is very interesting, but it has two problems that need to be improved. Firstly, as shown later in Section 4.3, their proposed approach easily suffers from

sub-optimality due to the proposed definition of the partitioning index, i.e., Equation 2.4. A better solution can be easily found by considering all the possible partitioning index values. In addition to the above sub-optimality, another main problem with the bound expressed by Equations 2.3 and 2.4, however, is the definition of the number of buffers that are required by the slow reader tasks, which corresponds to the second term on the right hand side of Equation 2.3. Instead of considering the maximum number of instances that can be activated for each reader task inside the lifetime and then adding them, it is better to upper bound the maximum number of slow reader instances that can be active at any time, reflected by the worst-case response time. A simple example from [113] illustrates this problem. The illustrative example consists of one writer and two reader tasks, with priorities lower than that of the writer. The period of the writer is 2 units while the periods of the two readers are 3 and 5 units. Assuming that the worst-case response times of the readers are equal to their respective periods, then the bound based on the DBP in [113] is 3 buffers (two lower priority readers and no communication link delay), whereas the bound calculated according to [10] is 4.

From the above discussion, it is clear that there are serious limitations with the state-of-the-art synchronous reactive communication protocols that target single processor execution architectures. This dissertation, therefore, aims at removing all these limitations.

2.2.3 Motivation

As the industry structure is undergoing a revolutionary change, system-level design becomes growingly the key to success. As embedded system design complexity continuously increases, more functionality has been shifted to software, which indicates that system

designs imply major emphasis on software. Model-based design for embedded software is playing a more critical role. In particular, synchronous reactive models are increasingly used for development of embedded control algorithms.

The synchronous reactive model of computation is very expressive and it can be used to specify a wide range of applications. Due to the assumptions made in [109][116][113][10] (e.g., no link delay or maximum unit delay, single task instance, etc.), the state-of-the-art SR communication protocols work only for a small subset of the application domain. To address a broader and practically useful design space, these synchronous implementations need to be generalized.

Similarly, as presented in Section 2.2.2, the application-level implementation of the Rate Transition Block in Simulink can only achieve the synchronous reactive semantics for applications with strict restrictions on sampling periods. For applications with nonharmonic periods, deterministic communication cannot be supported by the RTB. This is clearly unacceptable for applications that are sensitive to nondeterministic communication.

Though it is clear that synchronous reactive communication protocols should be implemented at both the kernel and the application levels, no existing literature addresses the support for a kernel-level implementation, let alone the implementation tradeoffs for different synchronous reactive communication protocols.

Indeed, a thorough investigation of all the questions raised in Section 2.2.1 and a comprehensive study on how to remove those limitations discussed in Section 2.2.2 are indispensable. All these justify the necessity of this dissertation.

Part II

Theory Generalization, Implementation, and Optimization

Chapter 3

Implementation Technology

In a semantics-preserving implementation of synchronous reactive communication, readers need to be ensured to access the value produced by the correct instance of the writer task. In particular, the buffer slot that contains the item produced by the writer has to be defined at the writer's activation time. Similarly, the buffer slot read by a reader is defined at the reader's activation time. Later, at execution time, the writer/reader will directly use the buffer positions defined earlier. Both writer and reader tasks, however, are not guaranteed to start execution at their release times because of scheduling delays. Therefore, in general, the selection of the buffer entry that will be written into or read from must be delegated to the operating system. In this chapter, the platform-based design methodology is presented for automatic configuration of real-time operating systems and automatic synthesis of synchronous reactive communication protocols. To achieve portability, application programming interfaces with an emphasis on OSEK are discussed. Finally, the complete model-based design flow, integrated with the platform-based design methodology, is given.

3.1 Platform-Based Design Methodology

Embedded systems exist for ubiquitous computing and they are usually implemented as SOCs. Challenges and benefits of SOC design are discussed in [21]. Reuse is the key to SOC design. Each successive generation of IC technology enables the creation of products that are significantly faster, consume much less power, and offer more capabilities at a greatly reduced system cost. Platform-Based Design (PBD) methodology is presented in [21] for meeting the challenges of comprehensive SOC design due to design productivity. Design flexibility and productivity can be traded off for each other through different levels of platforms. Platform-based design leads to productivity gains through emerging design technologies and methodologies.

The idea of PBD is originated from personal computer design community and has been generalized and formalized in [36][108][107] to alleviate the increasing time-to-market pressure on designs of embedded real-time systems. A platform is, in general, an abstraction that covers a number of possible refinements into a lower level. A platform, i.e., a layer of abstraction of the system, hides the complexity of the underlying computation, communication, sensing, and control. A platform library for building networked embedded real-time systems consists of a set of pre-designed primitive components to support design reuse, which improves the design efficiency and design cost. Two adjacent layers and the mapping tools between them form a platform stack. Therefore, in a PBD methodology, the design and implementation are structured into abstraction layers and mapping tools provide the conceptual glue that binds together adjacent abstraction layers. A PBD methodology is neither a top-down nor a bottom-up approach. For every platform, there are two views:

one is used to map the upper layer of abstraction into the platform (top-down mapping) and the other is used to define the class of lower level of abstraction implied by the platform (bottom-up exporting). Instead, it is indeed a meet-in-the-middle approach, which aims at providing efficient and effective distribution of resources at a low cost.

3.2 PBD for Real-Time Operating Systems

From the above discussion, it is clear that protocols preserving the SR semantics need kernel-level support to assign reading and writing buffer indices. A Real-Time Operating System (RTOS) enables communication among software tasks, hardware, and other system resources. It coordinates software tasks by scheduling those that are ready to execute. Supports from an RTOS are important for synchronous reactive model-based embedded software design. To achieve an efficient automatic generation of the task model and the corresponding configuration of the RTOS procedures and data structures, a platform-based design methodology is introduced as follows.

Figure 3.1 illustrates the stack view of the platform-based design methodology for real-time operating systems. From top to bottom, it shows the application domain, the task and communication resource model platform, the mapping tools from the application to the solution space, the virtual real-time operating system component platform, and the underlying execution architecture. In the rest of this section, detailed information is provided for each of them.

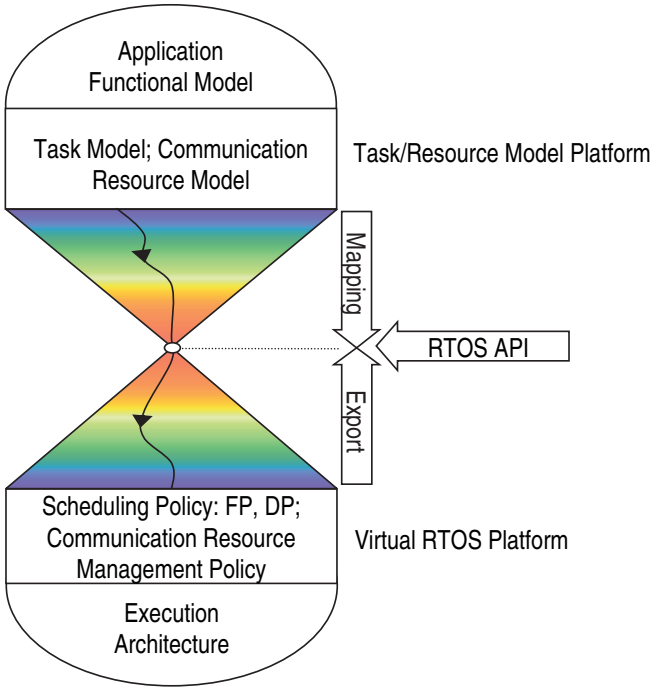

Figure 3.1: Platform-Based Design for RTOS

3.2.1 Application Model and Execution Architecture

The very top of Figure 3.1 describes the system behavior in terms of functional and aspectual (i.e., non-functional, performance) requirements, which can be formally specified using a formal language or a specification tool. For general embedded real-time systems, the applications can range from safety-critical hard real-time systems to multimedia. For example, safety-critical hard real-time applications can be functionally modeled as a network of blocks that execute according to the synchronous reactive semantics [119][14].

To meet different computation demands, the underlying execution platform at the bottom of Figure 3.1 can be uniprocessor, multiprocessor, or even distributed architectures.

3.2.2 Platform Models and Middle Meeting Point

In this subsection, the platforms and the middle meeting point in the picture of the stack view in Figure 3.1 are discussed in details.

Task/Resource Model Platform The task/resource model platform abstracts the characteristics of the set of application tasks and their interaction. Tasks are execution entities to carry out application functionalities. They are usually categorized as either real-time or non-real-time. Real-time tasks are further categorized as hard-real-time and soft-real-time. Tasks are characterized by a set of parameters. Task interaction is usually through shared resources such as communication buffers. As discussed in Section 1.3.2, task communication model can be either blocking or non-blocking.

Virtual RTOS Platform The virtual RTOS platform defines the scheduling policies and inter-task communication protocols that may be used. In the RTOS platform library, the scheduling policies may be static priority based, e.g., Rate Monotonic [73][71] and Deadline Monotonic [72][6], or dynamic priority based, e.g., Earliest Deadline First [73][31] and Least Laxity First [31][89]. Inter-task communication protocols may support blocking (lock-based), or non-blocking (lock-free, wait-free) inter-task data transfer and they may provide an implementation for different communication semantics. As discussed in Section 1.3.2, examples of lock-based protocols are the Priority Inheritance Protocol [111], the Priority Ceiling Protocol [111][26], the Stack Resource Policy [7], etc. Examples of lock-free protocols are the concurrent reading and writing protocol [63], the Non-Blocking Writing protocol, etc. Examples for wait-free protocols are the buffering tagging protocol [10], the Dynamic

Buffering Protocol [113], the multi-reader asynchronous protocol [23], etc.

The Middle Meeting Point Software standards have become increasingly important in industry. To support portability of real-time application software, RTOS Application Programming Interface (API) industrial standards have been developed, e.g., OSEK/VDX [101], POSIX [104], and μITRON [90].

OSEK/VDX standard is mainly for the automotive industry. OSEK stands for open systems and the corresponding interfaces for automotive electronics while VDX represents Vehicle Distributed eXecutive. OSEK/VDX is an industrial standard for an open-ended architecture for distributed control units in vehicles. μITRON represents Micro Industrial The Real-time Operating system Nucleus. μITRON real-time kernel specification is a Japanese industrial standard and has been used by more than half of the embedded developers in Japan. POSIX is the acronym for Portable Operating System Interface. It is a set of international standards consisting of base definitions, system interfaces, shell and utilities, and rationale. Defining a programming interface, POSIX standard specifies source code level API and thus supports source code portability. POSIX standard is UNIX-like and used for general-purpose applications.

In addition to the above industrial RTOS standards, there are also RTOS research frameworks developed by academia. For example, Soft and Hard Real-time Kernel (SHaRK), is a research kernel designed for testing and implementing new scheduling algorithms. ShaRK implements the standard POSIX 1003.13 PSE52 interface.

An RTOS API standard is introduced at the meeting point of the platform stack in Figure 3.1 to achieve portability and vendor-independence of software. Through this

meeting point, the application tasks may use the kernel services by calling the RTOS APIs. With the PBD methodology, a modular RTOS can be automatically configured from the virtual RTOS platform library according to the application tasks and the underlying execution architecture. An RTOS designed in this way only includes the required features and thus it achieves a small footprint.

As discussed above, there are a large number of available options for the RTOS synthesis problem. This dissertation focuses on a real-time operating system that supports a single processor execution platform with a preemptive, priority-based scheduling policy and conforms to the OSEK/VDX.

3.3 OSEK/VDX

In this section, the OSEK/VDX API standards used to implement the SR semantics-preserving protocols are introduced. The OSEK/VDX standards originated from France and Germany and have been widely used in the automotive industry. OSEK/VDX includes operating system (OS), communication (Com), network management (NM), and debugging (ORTI). In the following, only OSEK OS is discussed through summarizing the OSEK OS software architecture, kernel services, and the OSEK implementation language that is used during system generation.

3.3.1 Software Architecture

The OSEK OS architecture is designed to support OS scalability and application software portability. Three processing levels are defined in the OSEK OS. From high to

low priority, they are interrupt level, logical scheduler level, and task level. Tasks are categorized as either basic or extended based on whether they can enter a wait state by calling the `WaitEvent` kernel service. A basic task is not allowed to wait on an event. To support design reuse and to ease software upgrade, four conformance classes are defined according to the number of active activations per task, the task type, and the number of tasks per priority level.

To support application portability, minimum requirements are defined for all four conformance classes as shown in Table 3.1. For example, for BCC1, the minimum requirement specifies single active task activation, eight active tasks, distinct task priority assignment, eight priority levels, one alarm, one application mode, and no event. Any application that meets the minimum requirements is portable to any OSEK-compliant operating system that supports the same conformance classes.

	Basic		Extended	
	BCC1	BCC2	ECC1	ECC2
Multiple Active Task Instances	No	Yes	BT: No ET: No	BT: Yes ET: No
# of Tasks not in Suspend State	8		16 (Any Comb. of BT/ET)	
> 1 Task/Priority	No	Yes	No	Yes
# of Events/Task	-		8	
# of Priority Levels	8		16	
Resources	RES_SCHEDULER	8 (including RES_SCHEDULER)		
Internal Resources	2			
Alarm	1			
Application Mode	1			

Table 3.1: Minimum Requirements for OSEK CC

3.3.2 Kernel Services

In OSEK OS, the kernel services are structured into different functionality groups, including interrupt management, task management, resource management, event management, alarm management, and error treatment. Table 3.2 summarizes kernel services defined in the OSEK OS standard. In the following, the kernel services are briefly discussed in functionality groups.

Task Model	Basic task; Extended task
Synchronization	Event; Semaphore
Semaphore Sync Protocol	Priority Ceiling Protocol (Highest Locker Protocol)
Inter-task Comm Mechanism	Global variable; Message; Message filtering/notification
Task Management	Activate/terminate/chain/state reference
Scheduling	Non-/full/mixed preemptive; Round Robin (same level)
Multiple Activation	BCC2 tasks and basic tasks in ECC2
Memory Management	No virtual memory (MMU); No dynamic allocation
Stack Sharing	Yes for BCC and No for ECC
Interrupt Handling	ISR Categories 1 and 2; Nesting allowed
Time Management	Counter/alarm (relative/absolute; single/cyclic)
Error Management	Hook routine; Error code; Fatal/application error

Table 3.2: Summary of the OSEK OS Standard

Interrupt Management An Interrupt Service Routine (ISR) has a statically assigned priority level higher than that of tasks. There are two categories of ISRs specified in the OSEK OS standard. An ISR of Category 1 is not allowed to use any kernel services and it cannot be preempted. Termination of an ISR of Category 1 does not force any rescheduling. On the other hand, kernel services are allowed in an ISR of Category 2 and rescheduling will be performed at the end of its execution if there are no other pending ISRs.

Task Management A basic task has three states: running, ready, and suspended. In addition to these, an extended task has a fourth state: waiting. A task can be activated by either `ActivateTask` or `ChainTask` and it can only be terminated by itself by calling `TerminateTask`. The number of tasks in the system remains constant and multiple concurrent instances of a task are not allowed. If an activation call is made for a task that is already active, the request will be queued until the current instance terminates. Therefore, there is no need for dynamic task creation or deletion.

Resource Management Typical resources like mutex and semaphore are used for synchronization or coordination of tasks. Resource management controls access to shared resources such as memory, program sequences, etc. Access to multiple resources is strictly under the Stack Resource Protocol, i.e., the Last-In-First-Out (LIFO) principle. To prevent priority inversion and deadlock, the Priority Ceiling Protocol is used in the OSEK OS.

Event Management In addition to semaphore, synchronization can also be achieved by using events. The kernel primitive `WaitEvent` is only accessible to extended tasks. An event is owned by an extended task and it can be set via kernel primitive `SetEvent` by either a basic task, an extended task, or even an ISR of Category 2. Events are non-consumable and therefore they need to be cleared via kernel primitive `ClearEvent` by their owners after being used. Note that both event and semaphore are a blocking synchronization mechanism.

Alarm Management Alarms are managed in a layered manner. OSEK supports relative and absolute alarms and an alarm can be either single or cyclic. On the OSEK OS kernel side, counters are measured in ticks and at least one counter is generated from a

hardware or software timer. On the application side, primitives managing alarms such as `SetRelAlarm/SetAbsAlarm` are provided. An alarm can be associated with only one counter, but a counter can be used as a reference for more than one alarm. An alarm can be used to activate a task, set an event, or call an alarm callback routine.

Hook Management The hook mechanism is primarily used for error handling, tracing, and debugging purposes. This mechanism allows application-specific functionalities to be processed internally by the kernel. As part of the OS, a hook routine has a higher priority than all tasks and it cannot be preempted by an ISR of Category 2. In particular, `ErrorHook` is called upon the occurrence of application errors. `PreTaskHook` is called before executing a new task but after moving the task to the running state while `PostTaskHook` is called after executing the current task but before leaving the running state. `StartupHook` is called at the end of the OS initialization and before the scheduler starts running while `ShutdownHook` is called during the OS shutdown.

OS Execution Management There are three system services in this category. Called from tasks, ISRs of Category 2, or hooks, `GetActiveApplicationMode` returns the current application mode. `StartOS` can only be called from the application's *main* function with a specific execution mode. `ShutdownOS` is called to abort the overall system execution by a task, ISR of Category 2, `ErrorHook`, `StartupHook`, or the OS itself due to an occurred fatal error or an undefined internal state.

3.3.3 OSEK Implementation Language

The OSEK Implementation Language (OIL) [100] has been designed to support modular configuration for system generation of an application. In this subsection, the detailed contents of the OIL configuration files are presented. OIL is a mechanism used to configure an OSEK application inside a particular CPU. The OIL description of an OSEK application consists of a set of OIL objects that are characterized by a set of attributes and references. Attributes and references can be either standard or optional (application-specific). Refer to Table 3.3 for all OSEK OIL objects and their properties.

An OIL configuration is composed of two parts: implementation definition and application definition. The former defines all standard and application-specific attributes

Object	Mandatory	Standard Attribute	Standard Reference
CPU	yes	-	-
OS	yes (= 1)	STATUS; USERESSCHEDULE; USEGETSERVICEID; Hooks; USEPARAMETERACCESS	-
APPMODE	yes (≥ 1)	-	-
TASK	yes (≥ 1)	PRIORITY; SCHEDULE; ACTIVATION; AUTOSTART	MESSAGE; EVENT; RESOURCE
COUNTER	no	MAXALLOWEDVALUE; TICKSPERBASE; MINCYCLE	-
RESOURCE	no	RESOURCEPROPERTY	-
EVENT	no	MASK	-
ISR	no	CATEGORY	MESSAGE; RESOURCE
MESSAGE	no	NOTIFICATION; etc.	-
NWMESSAGE	no	SIZEINBITS; etc.	IPDU
COM	no (= 1)	COMTIMEBASE; etc.	-
IPDU	no	SIZEINBITS; etc.	-
NM	no (= 1)	-	-

Table 3.3: OIL Objects and Their Properties

and their properties for a particular OSEK implementation while the latter defines the set of objects and their corresponding attribute values for an OSEK application. All attributes used in an application definition must be defined in the corresponding implementation definition.

3.4 The Complete Design Flow

After presenting all the preliminary materials on model-based design and OSEK, in this section, the complete design flow that integrates model-based design, platform-based design, and the OSEK application development process is presented. As shown in Figure 3.2, applications are specified at the system level as synchronous reactive models. First of all, a model-based design tool takes as input the synchronous reactive model specification and automatically generates application tasks and an OIL configuration file. Notice that code that implements the synchronous reactive communication protocols is embedded in the implementation of tasks. Then, the OIL files are fed to the System Generator (SG), which automatically configures a kernel by choosing the required modules and customizing the data structure attributes based on the configuration file. Finally, the application source code that is automatically generated from the system specification, the selected module files from the OSEK OS kernel library, and the additional application files produced by the SG are compiled and linked together to produce an executable file for the application.

This dissertation mainly focuses on the top part of Figure 3.2, i.e., automatic generation of portable application tasks and an OSEK OIL configuration file that preserve the synchronous reactive communication semantics.

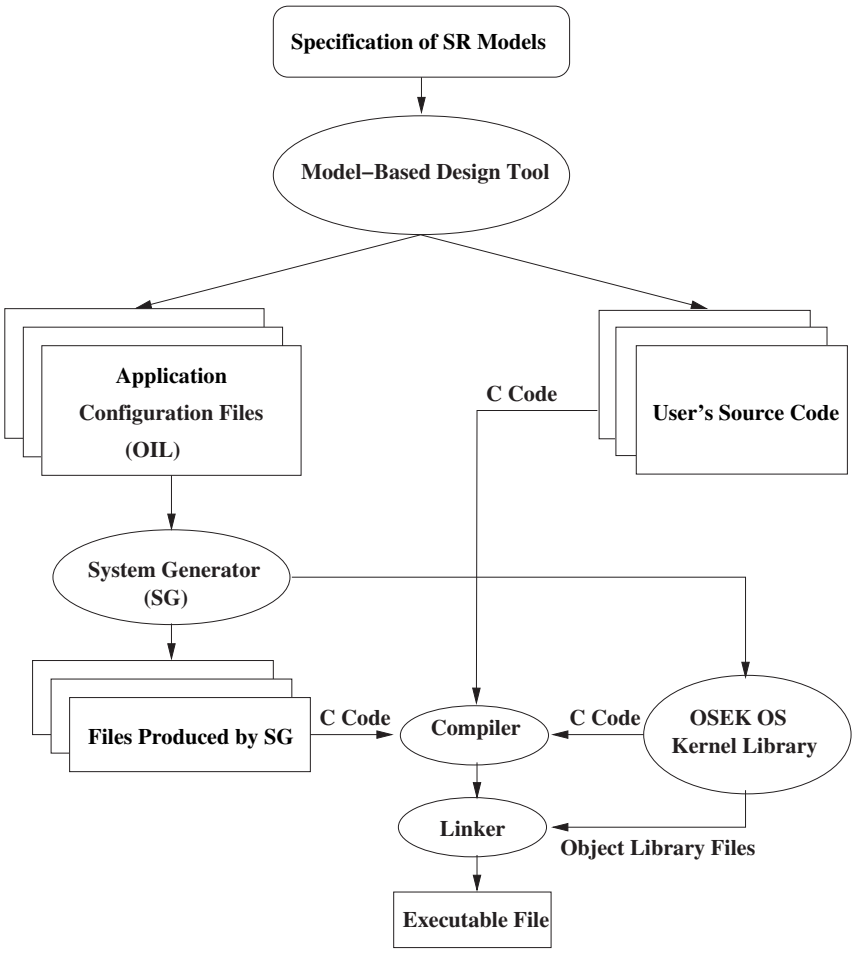

Figure 3.2: The Complete Design Flow

Chapter 4

Theory Generalization and Bound Improvement

In this chapter, the synchronous reactive communication semantics is generalized to consider arbitrary link delays. Extensions on the synchronous reactive communication theory and the code that implements the read and write operations in the general case of multiple-unit delays and deadlines (worst-case response times) larger than periods are provided. Without giving imperative code implementation, [113] only provides the safety criterion for the selection of a free buffer. Many implementations meeting the criterion are possible, with tradeoffs between space and time complexity. In this dissertation, one possible solution with constant execution time is provided in the context of the general case. In particular, two constant time SR communication protocols based on the number of reader instances [113][10] and on temporal concurrency control [10] are presented. Finally, improvements on buffer bound are defined with respect to the above two protocols.

4.1 Generalized One-to-Many Communication

In this section, the synchronous reactive communication semantics is first generalized to SR models where there may exist arbitrary communication link delays. Then, this link delay generalization naturally leads to a broader one to many communication scenario.

4.1.1 SR Semantics under Arbitrary Link Delay

In the synchronous reactive (Simulink) model [122][94], each block computes two functions: the output function and the state update function. Under the semantics of the SR model of computation, the execution of the block functions takes zero-time, that is, the result is computed instantaneously when the block is activated. The activation of a block can be constrained to be periodic (i.e., released at multiples of a given period) or event-triggered, possibly with a minimum inter-arrival time. Two blocks $\mathtt{b_i}$ and $\mathtt{b_j}$ may be activated at the same time, that is, $\mathtt{a_i(k)} = \mathtt{a_j(m)}$. Blocks communicate with each other through ports. Due to the possibility of existing ports of either input or output type, a block can be a reader, a writer, or both. A system of the SR model is a network of blocks without zero-delay cycles. The corresponding fixed point condition together with the definition of a partial order of the block execution ensures that the semantics is always well-defined.

In general, the communication link can possibly carry a $\mathtt{k}$-unit delay, as indicated by $\mathtt{b_i} \overset{-\mathtt{k}}{\rightarrow} \mathtt{b_j}$. Links with multiple-unit delays can be specified as a design parameter. However, a legitimate value depends upon the deadline and the period of the writer task [94]. For a $\mathtt{k}$-unit link delay, the SR communication semantics defines

$$\mathtt{i_j(n)} = \mathtt{o_i(m)}, \text{ where } \mathtt{m} = \max\{\mathtt{0}, \zeta_\mathtt{i}(\mathtt{a_j(n)}) - \mathtt{k}\}.$$

Each functional block b_i is implemented in the context of a task executing on a CPU, meaning that the code implementing the output and the state update functions is executed with an upper bounded execution time in the context of a thread. Recall that tasks are denoted as τ_i. The block to task mapping consists of a relation between a block and a task and of a static scheduling (execution order) of the block code inside the task. $\mathcal{M}(b_i, j, n)$ is used to indicate that b_i is executed as the n^{th} block in the context of τ_j.

The task model is captured by a set of parameters. For each instance with index m of τ_i, define as release time, $r_i(m)$, the instant when the task is ready for execution (assume $r_i(m) = a_i(m)$), the start time, $s_i(m)$, the instant when it obtains the control of the CPU, and the finish time, $f_i(m)$, the instant when it completes its execution. Introduced in Section 1.3 but without providing a definition, the worst-case response time is the time separation between the finish time and the activation time. Recall that in the real-time domain, τ_i is also characterized by a relative deadline D_i, which can be smaller, equal to, or greater than T_i. When the relative deadline is larger than the period or minimum inter-arrival time, there may exist multiple instances of the same task active (but not executing) at the same time.

The model used in this dissertation assumes that the code implementing a block only reads the input and writes the output once per activation. Reads and writes can happen at any time during the execution of the block and are not atomic. After being mapped to a task, a block is executed with the task's priority.

While communication is defined among blocks, the buffer bounds are computed based on the number of tasks. If multiple reader blocks communicating with the same writer

are mapped into the same task, they need to be counted as one because they may share the same communication buffer. In the following, the response time of a task is assumed as the representative of the worst-case response time of all the blocks that are mapped into it. This assumption, although pessimistic, is safe.

4.1.2 One-to-Many Communication

The existing SR communication protocols are extended with respect to multiple-unit link delays and multiple instances of a task. A prerequisite of any semantics-preserving implementation is that the reader block must be implemented with a priority lower than that of the writer or its scheduling must be constrained by a precedence order for a link $\mathtt{b_i} \rightarrow \mathtt{b_j}$ without delay [10]. To guarantee that the latency is not higher than a unit delay in the low to high priority communication [113], it must be $\mathtt{D_w} \leq \mathtt{T_w}$ for the writer task. However, when the deadline (worst-case response time) of a writer is greater than its period, at least λ units of delay are required in any link from the writer to a higher priority reader, where λ is defined as the smallest integer such that

$$\mathtt{R_w} \leq \mathtt{D_w} \leq \lambda \mathtt{T_w}.$$

Figure 4.1 defines a generalization of the scenarios in [113], where only the case of unit delay links is considered. As shown in Figure 4.1, writer task $\tau_\mathtt{w}$ communicates with $\mathtt{N}$ lower priority readers. Of those, $\mathtt{N_0}$ read data with no delay, $\mathtt{N_1}$ with unit delay, and so on, until $\mathtt{N_p}$ with $\mathtt{p}$-unit delay. Similarly, $\tau_\mathtt{w}$ communicates with $\mathtt{M}$ higher priority readers. Of those, $\mathtt{M_0}$ are connected with λ-unit delay links, $\mathtt{M_1}$ with $(\lambda + \mathtt{1})$-unit delay links, and so on, until $\mathtt{M_q}$ with $(\lambda + \mathtt{q})$-unit delay links. Consider all readers, the maximum link delay κ

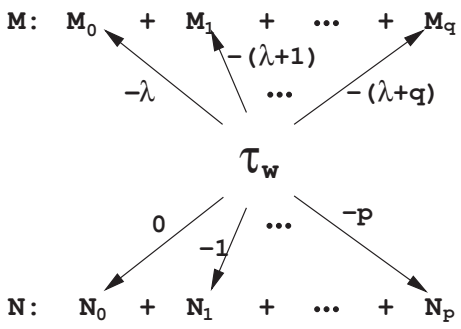

Figure 4.1: A General Scenario: Tasks with Link Delays and Priority Levels

is $\max(\mathtt{p}, \lambda + \mathtt{q})$.

For the following discussion, additional definitions are required. Each writer task is labeled as $\tau_{\mathtt{w}}$, and each reader task as $\tau_{\mathtt{r_i}}$. Let $\mathtt{NR_w}$ and $\mathtt{NB_w}$ be the number of readers and the number of buffers for the writer task $\tau_{\mathtt{w}}$, respectively.

The example shown in Table 4.1 is used as a running example to demonstrate how the proposed method improves on the existing approaches. The example consists of one writer and seven reader tasks. For the purposes of simplicity and comparison, it assumes that the writer has the highest priority and no delays are defined on the links. The first four rows refer to the task names, reader task indices, task periods, and worst-case computation times. The meaning of remaining rows will be explained in the rest of this section.

4.2 Generalized SR Communication Protocols

A buffer-based communication scheme usually consists of two parts: buffer sizing and buffer indexing. In the following, for each generalized protocol, the communication buffer sizing mechanism is first presented and then the corresponding buffer indexing pro-

1	Task	τ_{w}	τ_{r1}	τ_{r2}	τ_{r3}	τ_{r4}	τ_{r5}	τ_{r6}	τ_{r7}
2	i	-	1	2	3	4	5	6	7
3	T_i	20	8	10	12	22	40	80	240
4	C_i	2	1	2	2	4	4	5	10
5	$R_i = C_i + \sum_{k \in hp(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$ (Eq. 4.1)	2	3	5	7	16	35	77	235
6	$l_i = O_{wi} + R_i$ (Eq. 2.1)	-	23	25	27	36	55	97	255
7	$\left\lceil \frac{l_i}{T_w} \right\rceil$	-	2	2	2	2	3	**5**	**13**
8	$\left\lceil \frac{l_i}{T_{r_i}} \right\rceil$	-	3	3	3	2	2	2	2
9	$\sum_{k=1}^{i} \left\lceil \frac{l_k}{T_{r_k}} \right\rceil$	-	3	6	9	11	13	15	**17**
10	$\sum_{k=i}^{NR_w} \left\lceil \frac{l_k}{T_{r_k}} \right\rceil$	-	17	14	11	8	6	4	2
11	$\left\lceil \frac{l_i}{T_w} \right\rceil + \sum_{k=i+1}^{NR_w} \left\lceil \frac{l_k}{T_{r_k}} \right\rceil$ (Eq. 2.3)	17	16	13	10	8	**7**	**7**	**13**
12	$\left\lceil \frac{R_i}{T_{r_i}} \right\rceil$	-	1	1	1	1	1	1	1
13	$\sum_{k=1}^{i} \left\lceil \frac{R_k}{T_{r_k}} \right\rceil$	-	1	2	3	4	5	**6**	7
14	$\sum_{k=i}^{NR_w} \left\lceil \frac{R_k}{T_{r_k}} \right\rceil$	-	7	6	5	4	3	2	1
15	$\left\lceil \frac{l_i}{T_w} \right\rceil + \sum_{k=i+1}^{NR_w} \left\lceil \frac{R_k}{T_{r_k}} \right\rceil$ (Eq. 4.6)	7	8	7	6	**5**	**5**	**6**	13
16	$\sum_{k=1}^{i} \left\lceil \frac{l_k}{T_{r_k}} \right\rceil + 1$	-	4	7	10	12	14	16	18
17	$\sum_{k=i}^{NR_w} \left\lceil \frac{l_k}{T_{r_k}} \right\rceil + 1$	-	18	15	12	9	7	5	3
18	$\left\lceil \frac{l_i}{T_w} \right\rceil + \left(\sum_{k=i+1}^{NR_w} \left\lceil \frac{l_k}{T_{r_k}} \right\rceil + 1 \right)$ (Eq. 4.8)	18	17	14	11	9	**8**	**8**	13
19	$\sum_{k=1}^{i} \left\lceil \frac{R_k}{T_{r_k}} \right\rceil + 1$	-	2	3	4	5	6	7	8
20	$\sum_{k=i}^{NR_w} \left\lceil \frac{R_k}{T_{r_k}} \right\rceil + 1$	-	8	7	6	5	4	3	2
21	$\left\lceil \frac{l_i}{T_w} \right\rceil + \left(\sum_{k=i+1}^{NR_w} \left\lceil \frac{R_k}{T_{r_k}} \right\rceil + 1 \right)$ (Eq. 4.9)	8	9	8	7	**6**	**6**	7	13

Table 4.1: An Example of Single-Writer Multiple-Reader Configuration ($R \leq T$)

cedure is provided. Previous work assumes that the worst-case response time is smaller or equal to the period. In an implementation using the OSEK automotive operating system standard [101], this is indeed the case for BCC1 and ECC1 class implementations. However, the case of the worst-case response time larger than the period has practical relevance, as confirmed by the other OSEK conformance classes (i.e., BCC2 and ECC2). As shown in the rest of this section, a generalization to the case $R > T$ is indeed possible.

4.2.1 Generalized Dynamic Buffering Protocol

The first mechanism used to size a communication buffer is based on the active number of lower priority reader instances of a writer. The writer and its readers share a buffer array for data communication. As discussed in the following, the writer writes data into the buffer in a spatially non-sequential order, therefore, this communication protocol is also known to be based on spatially-out-of-order writes.

Buffer Sizing of GDBP The generalization to multiple-unit link delays and the case `R` $>$ `T` requires a different set of data structures. In case the delay on a link can be up to `k` units, the `cur` and `prev` index variables in [113] are no more sufficient. Similar to the unit delay case presented in [113], on the writer side, it needs to keep only one copy of the current and the previous `k` buffer indices, i.e., an array of (`k` + `1`) elements is required to store the indices referring the current and the last `k` elements written by the writer. A circular array, `pos[k+1]`, as shown in Figure 4.2, fulfills this purpose. Integer variable `cur` is used to index the entry in `pos[]` that stores the current buffer index and `pos[(cur+j)%(k+1)]` (`j` $\leq$ `k`) marks the index of the element of the shared buffer array containing the item with a `j`-unit

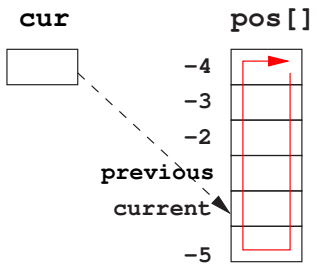

Figure 4.2: Data Structures for Writer Side

delay. When a new instance of the writer task is activated, the old buffer index with a `k`-unit delay becomes the new buffer index with a $(\texttt{k}+1)$-unit delay, which is not needed and therefore can be used for storing the new current buffer index. Similarly, the old current buffer index becomes the new buffer index with a unit delay.

The data structures on the reader side also need to be changed with respect to the version in [113] as follows. Each reader task now needs multiple entries in `Read[]`, one for each instance that may be active at the same time. As shown in Figure 4.3, these may be folded into a single array with each reader task using a contiguous subset of `Read[]` at constant offset `Roff[]` as a circular buffer of indices (one for each instance) with size `Ir[]`. The array `Ri[]` is used to index the currently released instance of a reader. Note that the functions of a specific `Ri[]` and its corresponding contiguous subset in `Read[]` are similar to those of `cur` and `pos[]`, respectively. Because there may be multiple active instances of a task existing in the system, a separate array `Rii[]`, similar to `Ri[]`, is needed to recover

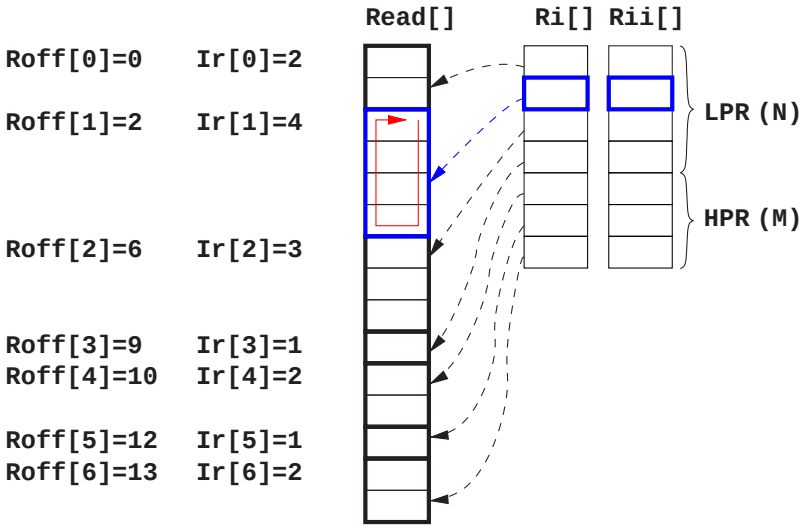

Figure 4.3: Data Structures for Reader Side (`D` > `T`)

the buffer index of the reader at execution time.

Figure 4.4 shows the data structures that are needed for the generalized dynamic buffering mechanism. When the worst-case response times can be larger than the periods, the total number of instances of lower priority readers that can be active at any time is computed as

$$\mathtt{ILPR_w} = \sum_{\mathtt{j} \in \mathtt{lp(w)}} \left\lceil \frac{\mathtt{R}_{\tau_{\mathtt{r_j}}}}{\mathtt{T_j}} \right\rceil,$$

where `lp(w)` represents the set of readers with a lower priority than that of the writer and $\mathtt{r_j}$ is the index of the reader task. Assuming all tasks have unique priorities, the worst-case response time can be computed according to the schedulability theory [5][17]:

$$\mathtt{R}_{\tau_{\mathtt{r_i}}} = \mathtt{C}_{\tau_{\mathtt{r_i}}} + \sum_{\mathtt{j} \in \mathtt{hp(i)}} \left\lceil \frac{\mathtt{R}_{\tau_{\mathtt{r_i}}}}{\mathtt{T_j}} \right\rceil \mathtt{C_j}, \tag{4.1}$$

where $\mathtt{C}_{\tau_{\mathtt{r_i}}}$ is the worst-case computation time of the task containing the reader block and the summation is extended over `hp(i)`, i.e., all the tasks $\tau_{\mathtt{j}}$ with a priority higher than that

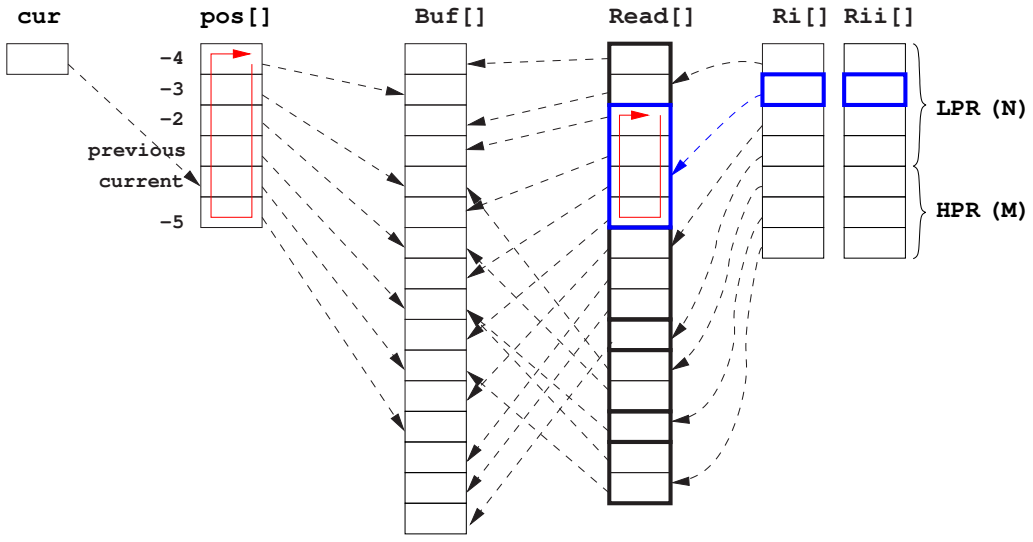

Figure 4.4: Communication Scheme Based on Spatially-out-of-Order Writes

of τ_{r_i}. Note that a better bound could be obtained by considering the actual position of the reader block inside the task [93].

Some of the entries in arrays `pos[]` and `Read[]` may certainly share buffer indices, but in the worst case, all of them may index unique buffer entries. Therefore, the size of the buffer can be computed as the following:

$$NB_w = ILPR_w + k + 1, \tag{4.2}$$

among which $ILPR_w$ slots are reserved for lower priority reader instances, `k` slots store the writer outputs with a delay from unity to `k` units, and one entry is for the writer to write into a new data item. Note that all the higher priority readers share the same copy of the communication data with a certain communication link delay. Clearly, this buffer sizing mechanism is based on the number of instances of low priority readers.

Before moving on, the bound based on the maximum number of active reader instances defined in Equation 4.2 is applied to the example shown in Table 4.1. The generalized method results in a size bound of 8 items, which is identical to the computed buffer bound obtained when using the sizing formula in [113]. This is expected because Formula 4.2 is a generalization of [113].

Buffer Indexing of GDBP The generalization to multiple-unit link delays and the case $R > T$ also requires a different set of procedures for the reader and writer tasks to access the communication data structures. There are different ways to implement the `FindFree()` procedure that is used to find a free buffer slot by the writer at its activation time. As shown on the left hand side of Figure 4.5, an array implementation of the list contains two fields: the use count (`use`) and the next free slot index (`NextFree`). The array `use` is used

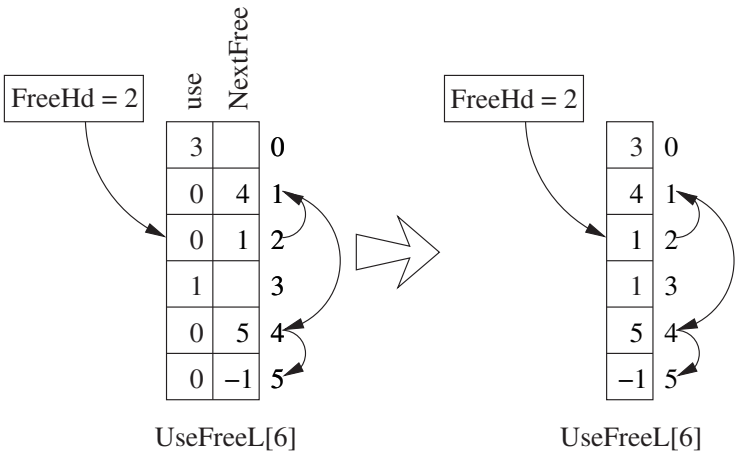

Figure 4.5: A Use Free List Data Structure

to keep track of how many references are currently using the corresponding entry of the communication buffer. Such references include the active reader task instances using the buffer or the possibility that the buffer contains one of the last written `k` instances and the current one. A zero value of `use[i]` means that the `i`th entry of `Buf[]` is free and can be overwritten by the writer. The use of the array `use` to track the buffers currently in use may lead to a very simple linear time implementation of the routine `FindFree()` that is used to find the first available buffer slot. However, `FindFree()` must be executed at the activation time of the writer by the kernel or at the highest priority level. A linear time implementation executing in kernel mode at task activation time is highly undesirable and it is worthwhile to trade off some memory space for a faster implementation if possible. A constant time implementation of the same routine is possible via implementing a free list that keeps track of the indices of the free elements, as shown on the left hand side of Figure 4.5. The start of the free list is indicated by `FreeHd` and the free list is terminated by a value of -1. It is not difficult to find that the values of the `use` fields along the free list

are all zeros. Therefore, to further save memory, the two columns can be compacted into one, as shown on the right hand side of Figure 4.5, containing the value of the next free slot index or the use count. The free entry can be obtained by getting the `FreeHd` value, and list updates can be performed in constant time.

The data structures and the indexing functions for the reader and writer tasks are defined in Figures 4.6 and 4.7. In the code implementation, the writer has the responsibility of updating the index to the element of `pos[]` containing the index of the buffer element

Data Structures

```
char pos[k+1];                  char UseFreeL[NBUF];  /* init with 0, # of users */
char Read[NINSTS];              char cur, FreeHd;     /* pos[cur] is current */
message Buf[NBUF];              char Ri[NR], Rii[NR]; /* init with 0 */
```

Buffer Management Routines

```
char FindFree() {
 char buf_id;
 buf_id = FreeHd;
 FreeHd = UseFreeL[FreeHd]
 return buf_id
}
```

```
UseDec(char i) {
  if (--UseFreeL[i]) {
   UseFreeL[i] = FreeHd;
   FreeHd = i;
  }
}
```

Figure 4.6: Data Structures and Supporting Routines for the Generalized CTDBP

	Writer: max k-unit delays	**Low/High Priority Reader:** delay[i] $\leq$ k

activation	`cur = (cur-1) % (k+1);`
	`UseDec(pos[cur]);`
	`pos[cur] = FindFree();`
	`UseFreeL[pos[cur]] = 1;`
		`Ri[i] = (Ri[i]+1) % Ir[i];`
		`i_id = Roff[i] + Ri[i];`
		`Read[i_id] = pos[(cur+delay[i])%(k+1)];`
		`UseFreeL[Read[i_id]]++;`

execution	`···`
	`Buf[pos[cur]] = ···`
	`···`
		`Rii[i] = (Rii[i]+1) % Ir[i];`
		`i_id = Roff[i] + Rii[i];`
		`··· = Buf[Read[i_id]];`

termination		`UseDec(Read[i_id]);`

Figure 4.7: Writer/Reader Code for the Generalized CTDBP

that will store the next written value. Similarly, the readers need to update the `Ri[]` and `Rii[]` at application task activation time and execution time, respectively. In addition, the writer and the readers must increment the value of `UseFreeL[i]` whenever they are going to write into or read from the buffer position of index `i`, and decrement the `UseFreeL[i]` via calling `UseDec(i)` when they finish using buffer position `i`. If `UseFreeL[i]` goes back to zero, the corresponding buffer slot needs to be freed and added back to the free list. Note that `FindFree()` does not have to check for the availability of at least one free buffer slot, because this is guaranteed by the mechanism that is used to size the communication buffer array.

Note that, though the array `pos[]` is presented with the notion of link delays, however, similar to `Read[]`, it also embeds the notion of multiple writer instances if the worst-case response time of the writer is bigger than its period. In this case, in addition to those previous slots `(-1,-2,···)`, more slots `(+1,+2,···)` are needed to accommodate multiple concurrent writer instances.

4.2.2 Generalized Temporal Concurrency Control Protocol

The second mechanism used for synchronous reactive communication allows a writer to write data into a circular communication buffer. Hence, such a buffer sizing mechanism, as described in the following, is also known as being based on spatially-in-order writes.

Buffer Sizing of GTCCP First, notice that the generalization to multiple-unit link delays and the case `R` $>$ `T` requires the same set of data structures as shown in Figures 4.2

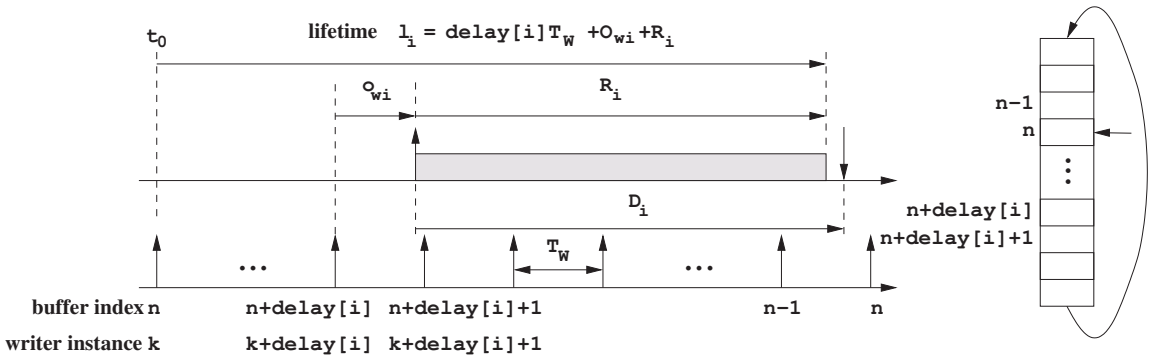

Figure 4.8: Communication Scheme Based on Spatially-in-Order Writes

and 4.3. Assume that some writer instance k happens at time $a_w(k)$ and the writer updates a buffer position of index n as shown in Figure 4.8. The item in position n is used by the readers that are activated during the time interval between $a_w(k + \texttt{delay}[i])$ and $a_w(k + \texttt{delay}[i] + 1)$ and use a communication link with a delay[i]-unit delay.

The buffer slot indexed by n must remain valid until any reader activated in these intervals has finished its execution. Future instances of the writer use buffer slots with indices n+1, n+2, and so on, until, eventually, the buffer index wraps around the circular buffer and goes back to position n-1. The condition for a correct buffer sizing is that all the reader instances that used the previous buffer at position n have finished using the data when some future writer instance goes back to position n and overwrites it.

Note that this size bound includes the buffer item reserved for the writer to update its current value. Recall that O_{wi} is the maximum offset between any activation of the writer and its reader τ_{r_i}. If the writer is periodic or sporadic with a minimum inter-arrival time, then it is $O_{wi} \leq T_w$. Let delay[] be the delay on the communication link. The definition of lifetime for a given reader given by Equation 2.1 can be easily extended. Figure 4.8

illustrates the basic idea about the communication mechanism based on a circular array. Define the maximum lifetime of the data produced by writer $\mathtt{w}$ for reader $\mathtt{r_i}$, denoted by $\mathtt{l_i}$, as follows:

$$\mathtt{l_i} = \mathtt{delay[i]} \times \mathtt{T_w} + \mathtt{O_{wi}} + \mathtt{R_i}, \tag{4.3}$$

where the worst-case response time can be computed for each reader using Equation 4.1.

A trivial bound consists in adding the number of buffers that are required by each reader $\tau_{\mathtt{r_j}}$. However, this estimate is clearly too pessimistic and can be improved. Let $\mathtt{NR_w}$ be the number of readers of writer $\mathtt{w}$ and the buffer size can be computed as follows:

$$\mathtt{NB_w} = \max_{1 \le \mathtt{i} \le \mathtt{NR_w}} \left\lceil \frac{\mathtt{l_i}}{\mathtt{T_w}} \right\rceil. \tag{4.4}$$

Under this communication mechanism, the writer just keeps writing data into the next slot in the circular buffer at its own sampling rate. Clearly this buffer sizing method mainly relies on the temporal properties of a writer and its readers. The writer writes into the circular buffer sequentially and thus it is also known as buffering sizing based on in-order writes.

Before moving to discuss the buffer indexing procedure, the bound based on the maximum number of writes defined in Equation 4.4 is applied to the example shown in Table 4.1. The generalized method based on TCC results in a size bound of 13 items, which is due to the last reader as shown in row 7 in Table 4.1.

Buffer Indexing of GTCCP Notice that the code implementation shown in Figure 4.7 represents a general implementation of a semantics-preserving single-writer to multiple-reader communication. Its idea can still be used for the indexing procedure of the Gener-

Data Structures

```
char pos[k+1];
message Buf[NBUF];
char Read[NINSTS];
```

```
char cur, FreeB;        /* pos[cur] is current */
char Ri[NR], Rii[NR];   /* init with 0 */
```

Buffer Management Routine

```
char FindFree() {
 return (pos[cur]+1) % NBUF
}
```

Figure 4.9: Data Structures and Supporting Routines for the Generalized TCCP

alized TCCP. However, the code implementation for GTCCP is much simpler than that of the GDBP since the GTCCP does not need to maintain a shared use free list data structure. As shown in Figure 4.9, the implementation of the `FindFree()` procedure is very simple: increment and modulo by the buffer size. Figure 4.10 shows the writer and reader code. Notice that due to the absence of the use free list data structure, the code for both writer and reader at activation time is simpler and there is no reader termination code that needs special treatment. The good thing about this method is that it does not require much bookkeeping to achieve constant execution time for finding a safe buffer slot for the writer.

	Writer: max k-unit delays	**Reader:** delay[i] $\leq$ k

activation	`cur = (cur-1) % (k+1);`
	`pos[cur] = FindFree();`
		`Ri[i] = (Ri[i]+1) % Ir[i];`
		`i_id = Roff[i] + Ri[i];`
		`Read[i_id] = pos[(cur+delay[i])%(k+1)];`

execution	`···`
	`Buf[pos[cur]] = ···`
	`···`
		`Rii[i] = (Rii[i]+1) % Ir[i];`
		`i_id = Roff[i] + Rii[i];`
		`··· = Buf[Read[i_id]];`

Figure 4.10: Writer/Reader Code for the Generalized TCCP

4.3 Buffer Bound Based on Hybrid Scheme

From Sections 4.2.1 and 4.2.2, it is clear that the GTCCP is faster than the GDBP although constant search algorithms exist for both. The buffer sizing mechanisms show that the GDBP is good for slow readers and the GTCCP is good for fast readers. Considering that a writer may have readers with dramatically different temporal characteristics, it may be memory efficient to use a hybrid communication scheme between a writer and its readers. This idea has been used in [10], however the proposed buffer bound suffers from limitations as discussed in Section 2.2.2. In the following, further improvement and generalization to incorporate multi-unit link delays are provided.

4.3.1 Buffer Bound Improvement

In the context of [10], i.e., no communication link delay, the buffer sizing is given by Equations 2.1, 2.3, and 2.4. Apply them to the example shown in Table 4.1 and the details are illustrated by the rows from 5 to 11. The partitioning index given by Equation 2.4 is 7, which means that all the readers are categorized as fast. The corresponding buffer size is 13, as shown in Row 11. Obviously, this buffer bound corresponds to a pure TCCP as computed in Section 4.2.2.

However, when going through the entries in Row 11, it is easy to find that the best buffer bound is actually 7, which corresponds to a partitioning index value of 5 or 6. This sub-optimality associated with the definition of the partitioning index given by Equation 2.4 can be eliminated by considering all the possible values of $\mathtt{j}$ as follows:

$$\mathtt{j} \in 0..\mathtt{NR_w} | \mathtt{min} \left\{ \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \sum_{\mathtt{i}=\mathtt{j}+1}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{l_{r_i}}}{\mathtt{T_{r_i}}} \right\rceil \right\}. \tag{4.5}$$

In reality, as explained in Section 4.2.1, each instance of a slow reader requires no more than one buffer at any time, which is captured by its worst-case response time. Therefore, the bound can be improved as follows:

$$\mathtt{NB_w} = \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \sum_{\mathtt{i=j+1}}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{R_{r_i}}}{\mathtt{T_{r_i}}} \right\rceil . \tag{4.6}$$

Note that the first term on the right hand side of Equation 4.6 represents the shared buffers for fast readers that use the TCCP while the second term stands for the dedicated buffers for slow readers that use the DBP. This improved bound is always demonstrably at least as good as the bounds in [113] and [10], if not better, given that it is always

$$\mathtt{R_i} < \mathtt{l_i} = \mathtt{O_{wi}} + \mathtt{R_i}$$

and, therefore, clearly

$$\left\lceil \frac{\mathtt{R_{r_i}}}{\mathtt{T_{r_i}}} \right\rceil \leq \left\lceil \frac{\mathtt{l_{r_i}}}{\mathtt{T_{r_i}}} \right\rceil .$$

For the three-task example [113] shown in Section 2.2.2, the solution with 3 buffers is found using Equation 4.6.

The partitioning index corresponding to Equation 4.6 is defined as

$$\mathtt{j} \in \mathtt{0..NR_w} | \mathtt{min} \left\{ \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \sum_{\mathtt{i=j+1}}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{R_{r_i}}}{\mathtt{T_{r_i}}} \right\rceil \right\}. \tag{4.7}$$

When apply buffer sizing Formula 4.6 and the idea of Equation 2.4 to define the partitioning index to the example shown in Table 4.1, the computed partitioning index is 6 and the corresponding bound is 6 as shown in Row 15. However, when Equations 4.6 and 4.7 are used, the optimal buffer bound is found to be 5, which corresponds to a value of 4 or 5 for the partitioning index.

Notice that Equations 2.3 and 4.6 can only used for the purpose of buffer sizing. In practice, when considering buffer indexing procedures, the slot reserved for the current writer instance in the first terms on the right hand side cannot be shared by the slow and fast readers since they have different data structure supports. Therefore, they should be adjusted as following:

$$\mathtt{NB_w} = \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \left(\sum_{\mathtt{i=j+1}}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{l_i}}{\mathtt{T_{r_i}}} \right\rceil + 1 \right) \tag{4.8}$$

and

$$\mathtt{NB_w} = \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \left(\sum_{\mathtt{i=j+1}}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{R_{r_i}}}{\mathtt{T_{r_i}}} \right\rceil + 1 \right). \tag{4.9}$$

Their corresponding partitioning indices are computed as

$$\mathtt{j} \in \mathtt{0..NR_w} | \mathtt{min} \left\{ \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \left(\sum_{\mathtt{i=j+1}}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{l_i}}{\mathtt{T_{r_i}}} \right\rceil + 1 \right) \right\} \tag{4.10}$$

and

$$\mathtt{j} \in \mathtt{0..NR_w} | \mathtt{min} \left\{ \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \left(\sum_{\mathtt{i=j+1}}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{R_{r_i}}}{\mathtt{T_{r_i}}} \right\rceil + 1 \right) \right\}, \tag{4.11}$$

respectively. The details of buffer sizing using Equations 4.8 and 4.9 are shown in rows from 16 to 21 in Table 4.1. The bound based on Equations 4.8 and 4.10 is found to be 8 as shown in Row 18, with the partitioning index to be either 5 or 6. The bound based on Equations 4.9 and 4.11 is found to be 6 as shown in Row 21, with the partitioning index to be either 4 or 5.

4.3.2 Generalization Based on Hybrid Scheme

In this subsection, the buffer sizing in Section 4.3.1 is generalized for synchronous reactive models that have multiple-unit link delays. For this purpose, the lifetime definition

given by Equation 2.1 is replaced by Equation 4.3. With this generalization, the buffer sizing formulae 2.3, 2.4, 4.5, 4.6, and 4.7 are good for sizing synchronous reactive models with multiple task instances and multiple-unit link delays. However, when incorporating buffer indexing procedures, Equations 4.8, 4.10, 4.9, and 4.11 need to be adjusted to deal with possible multiple-unit link delays as followings:

$$\mathtt{NB_w} = \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \left(\sum_{\mathtt{i=j+1}}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{l_i}}{\mathtt{T_{r_i}}} \right\rceil + 1 + \max_{\mathtt{k=j+1}}^{\mathtt{NR_w}} \mathtt{delay[k]} \right), \tag{4.12}$$

$$\mathtt{j} \in \mathtt{0..NR_w} | \mathtt{min} \left\{ \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \left(\sum_{\mathtt{i=j+1}}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{l_i}}{\mathtt{T_{r_i}}} \right\rceil + 1 + \max_{\mathtt{k=j+1}}^{\mathtt{NR_w}} \mathtt{delay[k]} \right) \right\}, \tag{4.13}$$

$$\mathtt{NB_w} = \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \left(\sum_{\mathtt{i=j+1}}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{R_{r_i}}}{\mathtt{T_{r_i}}} \right\rceil + 1 + \max_{\mathtt{k=j+1}}^{\mathtt{NR_w}} \mathtt{delay[k]} \right), \tag{4.14}$$

and

$$\mathtt{j} \in \mathtt{0..NR_w} | \mathtt{min} \left\{ \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \left(\sum_{\mathtt{i=j+1}}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{R_{r_i}}}{\mathtt{T_{r_i}}} \right\rceil + 1 + \max_{\mathtt{k=j+1}}^{\mathtt{NR_w}} \mathtt{delay[k]} \right) \right\}. \tag{4.15}$$

As discussed in Section 4.3.1, the second terms on the right hand side of Equations 4.12 and 4.14 accounts for the buffer consumption by the slow readers and the corresponding communication protocol is the GDBP. However, there is no guarantee that all slow readers have a lower priority than the writer. Therefore the bound given by Equations 4.12 and 4.14 can be further improved as follows:

$$\mathtt{NB_w} = \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \left(\sum_{\mathtt{i} \in \mathtt{lp(w)} \bigwedge \mathtt{i=j+1}}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{l_i}}{\mathtt{T_{r_i}}} \right\rceil + 1 + \max_{\mathtt{k=j+1}}^{\mathtt{NR_w}} \mathtt{delay[k]} \right) \tag{4.16}$$

and

$$\mathtt{NB_w} = \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \left(\sum_{\mathtt{i} \in \mathtt{lp(w)} \bigwedge \mathtt{i=j+1}}^{\mathtt{NR_w}} \left\lceil \frac{\mathtt{R_{r_i}}}{\mathtt{T_{r_i}}} \right\rceil + 1 + \max_{\mathtt{k=j+1}}^{\mathtt{NR_w}} \mathtt{delay[k]} \right). \tag{4.17}$$

Their corresponding definitions for the partitioning index are

$$\mathtt{j} \in 0..\mathtt{NR_w} | \mathtt{min} \left\{ \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \left(\sum_{\mathtt{i} \in \mathtt{lp(w)} \bigwedge \mathtt{i}=\mathtt{j}+1} \left\lceil \frac{\mathtt{l_i}}{\mathtt{T_{r_i}}} \right\rceil + 1 + \max_{\mathtt{k}=\mathtt{j}+1}^{\mathtt{NR_w}} \mathtt{delay}[\mathtt{k}] \right) \right\} \tag{4.18}$$

and

$$\mathtt{j} \in 0..\mathtt{NR_w} | \mathtt{min} \left\{ \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_w}} \right\rceil + \left(\sum_{\mathtt{i} \in \mathtt{lp(w)} \bigwedge \mathtt{i}=\mathtt{j}+1} \left\lceil \frac{\mathtt{R_{r_i}}}{\mathtt{T_{r_i}}} \right\rceil + 1 + \max_{\mathtt{k}=\mathtt{j}+1}^{\mathtt{NR_w}} \mathtt{delay}[\mathtt{k}] \right) \right\}. \tag{4.19}$$

Consider the buffer sizing given by Equations 4.17 and 4.19. When the partitioning index is `-1`, i.e., all the readers fall into the slow reader category, the bound is exactly the same as Equation 4.2 given in Section 4.2.1. On the other hand, when the partitioning index is found to be $\mathtt{NR_w}$, i.e., all the readers fall into the fast reader category, the bound is exactly the same as Equation 4.4 given in Section 4.2.2.

4.3.3 Buffer Requirement Evaluation

In this subsection, the generalized buffer sizing approaches discussed above are applied to two examples.

An Example with $\mathtt{R} > \mathtt{T}$ The first example is shown in Table 4.2. Similar to the example shown in Table 4.1, there is one writer and seven readers in this example. The first four rows in Table 4.2 are task names, reader task indices, worst-case computation times, and sampling periods. Assume that priorities are assigned based on the rate monotonic policy. Then, as shown in Row 5, the first three readers have higher priorities and the last four readers have lower priorities than the writer. As shown in Row 6, the communication link delay design parameters are chosen to be unit and two units of the sampling rate of the writer for the first four and last three readers, respectively.

1	Task	τ_w	τ_{r1}	τ_{r2}	τ_{r3}	τ_{r4}	τ_{r5}	τ_{r6}	τ_{r7}
2	i	-	1	2	3	4	5	6	7
3	C_i	2	1	2	2	4	4	9	10
4	T_i	20	8	10	12	22	40	80	240
5	Relative priority with respect to writer	-	H	H	H	L	L	L	L
6	$delay[i]$	-	1	1	1	1	2	2	2
7	$R_i = C_i + \sum_{k \in hp(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$	7	1	3	5	16	35	107	879
8	$l_i = delay[i] \times T_w + O_{wi} + R_i$	-	41	43	45	56	95	167	939
9	$\left\lceil \frac{l_i}{T_w} \right\rceil$	-	3	3	3	3	**5**	**9**	47
10	$\left\lceil \frac{l_i}{T_{r_i}} \right\rceil$	-	6	5	4	3	3	3	4
11	$\sum_{k=1}^{i} \left\lceil \frac{l_k}{T_{r_k}} \right\rceil$	-	6	11	15	18	21	**24**	28
12	$\sum_{k=i}^{NR_w} \left\lceil \frac{l_k}{T_{r_k}} \right\rceil$	-	28	22	17	13	10	7	4
13	$\left\lceil \frac{l_i}{T_w} \right\rceil + \sum_{k=i+1}^{NR_w} \left\lceil \frac{l_k}{T_{r_k}} \right\rceil$	28	25	20	16	13	**12**	**13**	47
14	$\left\lceil \frac{R_i}{T_{r_i}} \right\rceil$	-	1	1	1	1	1	2	4
15	$\sum_{k=1}^{i} \left\lceil \frac{R_k}{T_{r_k}} \right\rceil$	-	1	2	3	4	**5**	7	11
16	$\sum_{k=i}^{NR_w} \left\lceil \frac{R_k}{T_{r_k}} \right\rceil$	-	11	10	9	8	7	6	4
17	$\left\lceil \frac{l_i}{T_w} \right\rceil + \sum_{k=i+1}^{NR_w} \left\lceil \frac{R_k}{T_{r_k}} \right\rceil$	11	13	12	11	**10**	**11**	13	47
18	$\sum_{k \in lp(w) \bigwedge k=1}^{i} \left\lceil \frac{l_k}{T_{r_k}} \right\rceil + 1 + \max_{k=1}^{i} delay[k]$	-	2	2	2	5	9	**12**	16
19	$\sum_{k \in lp(w) \bigwedge k=i}^{NR_w} \left\lceil \frac{l_k}{T_{r_k}} \right\rceil + 1 + \max_{k=i}^{NR_w} delay[k]$	-	16	16	16	16	13	10	7
20	$\left\lceil \frac{l_i}{T_w} \right\rceil + \left(\sum_{k \in lp(w) \bigwedge k=i+1}^{NR_w} \left\lceil \frac{l_k}{T_{r_k}} \right\rceil + 1 + \max_{k=i+1}^{NR_w} delay[k] \right)$	16	19	19	19	16	**15**	**16**	47
21	$\sum_{k=1}^{i} \left\lceil \frac{R_k}{T_{r_k}} \right\rceil + 1 + \max_{k=1}^{i} delay[k]$	-	2	2	2	3	**5**	7	11
22	$\sum_{k \in lp(w) \bigwedge k=i}^{NR_w} \left\lceil \frac{R_k}{T_{r_k}} \right\rceil + 1 + \max_{k=i}^{NR_w} delay[k]$	-	11	11	11	11	10	9	7
23	$\left\lceil \frac{l_i}{T_w} \right\rceil + \left(\sum_{k \in lp(w) \bigwedge k=i+1}^{NR_w} \left\lceil \frac{R_k}{T_{r_k}} \right\rceil + 1 + \max_{k=i+1}^{NR_w} delay[k] \right)$	**11**	14	14	14	13	**14**	16	47

Table 4.2: An Example of Single-Writer Multiple-Reader Configuration ($R \leq T$ or $R > T$)

Row 7 shows that worst-case response times of reader tasks 6 and 7 are bigger than their respective periods, which implies that there are multiple instances of these two tasks. None of the previous work can deal with this general scenario with multiple instances of a task and multiple-unit link delays. Rows from 10 to 13 show the details of buffer sizing

based on the Equations 2.3 and 2.4, with a direct generalization of the definition of data lifetime (i.e., Equation 4.3). As shown in Row 13, Equations 2.3, 2.4, and 4.3 give a buffer size of 13, with the partitioning index to be 6. When eliminating the sub-optimality with Equation 2.4, Equations 2.3, 4.5, and 4.3 give a buffer size of 12, with the partitioning index to be 5. Rows from 14 to 17 show the buffer sizing details when reader task instances are computed based on worst-case response time, i.e., Equation 4.6. As shown in Row 17, if following the idea of Equation 2.4, the partitioning index is computed as 5 and the corresponding bound is 11. However, Equations 4.6, 4.7, and 4.3 give a better bound of 10, with the partitioning index to be 4. In the context of Rows from 10 to 13, Rows from 18 to 20 show the result when taking priorities and separate copies of current and delay slots into account for slow readers. When using the idea of Equation 2.4, the partitioning index is 6 and the corresponding bound is 16. From Equations 4.16 and 4.18, the partitioning index is found to be 5 and the corresponding buffer size is 15. Finally, corresponding to the context of Rows from 14 to 17, Rows from 21 to 23 show the result when taking priorities and separate copies of current and delay slots into account for slow readers. Based on the idea of Equation 2.4, the computed partitioning index is 5 and the corresponding bound is 14. But Equations 4.17 and 4.19 give a better bound of 11, with the partitioning index to be -1, which means that the optimal bound is obtained when categorizing all readers as slow. Notice also that the last entry of Row 23 gives a bound of 47 when all readers are considered as fast ones, which is identical to the result of the GTCCP as shown by the last entry in Row 9.

An Automotive Case Study The buffer sizing approaches are compared on a real case study consisting of an automotive application. The application description, provided by a car electronics supplier, consists of a robotized gear shift system. The application is a complex network of functions, in which approximately two hundred AUTOSAR runnables (functions called in response to events) execute at different rates and communicate by exchanging data signals. The runnables are mapped into 16 tasks, as shown in Table 4.3. Some of the tasks are sporadic and are represented in the table with their worst-case inter-arrival rates.

The first three columns of Table 4.3 are task indices, periods (or worst-case inter-arrival times), and priorities. The periods and priorities are directly taken from the automotive application. There are 7 different worst-case rates in the example. Column 4 shows

Task	Period (msec)	Priority	C_i (μsec)	NOP	NLPR	NHPR	Utilization (%)
0	1000	10	1500	4	0	0	0.15
1	1000	9	5002	4	3	0	0.50
2	10	13	148	4	0	0	1.48
3	5	16	208	4	0	1	4.16
4	10	12	100	3	0	2	1.00
5	1000	1	131100	3	2	0	13.11
6	1000	5	150000	3	2	1	15.00
7	10	15	330	4	1	12	3.30
8	10	11	10	6	1	1	1.00
9	1000	4	100000	3	14	2	10.00
10	1000	2	120000	3	13	2	12.00
11	4	14	39	2	4	18	0.98
12	12	7	820	2	10	6	6.83
13	50	8	1000	0	0	0	2.00
14	100	6	9850	1	11	6	9.85
15	1000	3	110000	0	29	4	11.00

Table 4.3: An Example Derived from an Automotive Industrial Application

the worst-case execution time of tasks, among which only 7 are available as measures taken on the real application. The others have been assigned in an arbitrary way, to achieve the utilization values used in the experiments (the real application has a quite high utilization).

Columns 5, 6, and 7 represent the numbers of output ports (writers), lower-priority readers, and higher-priority readers respectively for the task. Also, in the information available from the real application, the communication topology is only defined as communication flows among the components. Based on these values, assumptions are made about the estimated communication among runnables and finally among tasks, thereby completing the definition of the communication topology. As shown in Columns 5, 6, and 7, there are 46 writers and 145 readers in the derived example. In the experiments, a set of worst-case computation times have been assigned to tasks to obtain different utilizations. All entries in Column 8 refer to the values for the case with a utilization `U = 92.4%`.

This case study is a typical example of system configuration in which the simulation-based approach in [116] cannot be applied because of the sporadic tasks in the system, for which the exact activation times are unknown.

The buffer sizes computed from the sizing methods presented in the previous sections are given in Table 4.4. The first two rows represent the test cases and their corresponding utilizations. Altogether, 13 system configurations are tried, with a utilization range from 62.7% to 92.4%. Row 3 shows the results from Equations 2.3, 2.4, and 4.3, i.e., derived from Baleani's original formula in [10] with the consideration of link delays. Row 4 shows the results based on Equation 4.6, the idea of Equation 2.4, and Equation 4.3, i.e., when using the worst-case response time to compute the number of reader instances instead

1	Case	1	2	3	4	5	6	7	8	9	10	11	12	13
2	Utilization %	62.7	67.7	75.7	80.6	85.6	86.2	88.2	89.6	89.7	90.7	91.2	92.2	92.4
3	Baleani's [10]	128	128	128	134	134	134	134	134	134	134	134	134	134
4	(l -> R)	128	128	128	134	134	134	134	134	134	134	134	134	134
5	Partition index	127	127	127	130	130	130	130	130	130	130	130	130	130
6	Fix both	127	127	127	130	130	130	130	130	130	130	130	130	130
7	With priority	123	123	123	124	124	124	124	124	124	124	124	124	124
8	CTDBP	162	162	162	162	162	162	162	162	162	162	162	162	162
9	TCCP	991	1145	1289	1370	1575	1595	1612	1612	1614	1656	1656	1656	1656

Table 4.4: Experimental Results of the Automotive Example

of the lifetime. Row 3 and Row 4 show that the corresponding bounds are the same, which is because the system configuration parameters do not trigger the sub-optimality in Equation 2.3. Row 5 shows the results when using Equations 2.3, 4.5, and 4.3, i.e., the improved partitioning index instead of Baleani's original one. The results in Row 5 are smaller than the corresponding ones in Row 3. This clearly demonstrates the sub-optimality due to the definition of the partitioning index in Equation 2.4. Note that the results in Rows from 3 to 5 have taken the buffer indexing procedure into account, i.e., the slow readers keep a separate set of current and previous buffer slots. Row 6 shows the results when using Equations 4.14 and 4.15, i.e., considering the improvements in Row 4 and Row 5. The results in Row 6 are the same as those in Row 5. This further illustrates the sub-optimality associated with Equation 2.4. Row 7 shows the results when using Equations 4.17 and 4.19, i.e., taking into account the priorities when sizing the slow readers using the idea of the DBP. The results shown in Row 7 are smaller than those in Row 6, which justifies that the slow readers may have either a higher or lower priority than its writer. Rows 8 and 9 show the sizing results obtained by using the CTDBP and the TCCP, respectively.

The buffering sizing method proposed in this dissertation allows saving at least 10 buffer positions at higher utilizations when compared with [10], which is approximately a 7.5% improvement. The DBP policy by itself can overestimate the number of required buffers by 25%, which is the amount of space gained by separating the buffer handling policies for slow and fast tasks. Also, the TCCP policy clearly performs very poorly when the application is a mix of high rate and low rate tasks as shown in this case study.

The table results correspond to the bounds expressed by the formulae, but *do not take into account the time overheads that are necessary for the implementation of the communication procedures.* These overheads consist of additional time spent executing code at the kernel level, with a possibly high impact on schedulability at very high utilizations. The next step is an assessment of the time required for the execution of the buffer access policies and a new set of experiments that show how these overheads are likely to affect the results. It is possible to define an optimization problem in which the memory required for the implementation of communication is minimized within the real-time constraints of the application.

Chapter 5

Portable Implementation

For embedded software development, efficiency (in terms of time and space) and portability of application software have become more and more important. In this chapter, two synchronous reactive communication protocols with constant search time, i.e., the Constant Time Dynamic Buffering Protocol and the Temporal Concurrency Control Protocol, are presented first and then their corresponding portable efficient implementation details are given under the OSEK API. A detailed OIL configuration file is presented for customization of a supporting real-time operating system. In addition, the tradeoffs between different protocols in terms of time, space, and implementation complexity are analyzed. Finally, this chapter is concluded by providing automatic code generation support for two synchronous reactive communication protocols: the Double Buffer Protocol and the Constant Time Dynamic Buffering Protocol.

5.1 Efficient SR Communication Protocols

The proposed code implementation is very efficient, as required by its execution in the kernel mode. In this section, two synchronous reactive communication protocols are presented. They are special cases of the communication protocols presented in Section 4.2. The special version (maximum unit link delay and single instance for all tasks) of the protocols is implemented with the consideration of software portability in Section 5.2. However, the same principle can be applied to implement the generalized version of the SR communication protocols. Notice that, similar to generalized communication protocols, the protocols presented in the following are also for a single writer and multiple readers. Note that none of `pos[]`, `Ni[]`, and `Nii[]` is needed due to the assumption that worst-case response times are smaller than periods.

5.1.1 Dynamic Buffering Protocol

When restricting the maximum communication link delay to be unity and assume that there exists only single instance per task, the Generalized Dynamic Buffering Protocol presented in Section 4.2.1 degenerates to the Dynamic Buffering Protocol, whose data structures and buffer indexing procedures are shown in Figures 5.1 and 5.2. Comparison of the generalized and the special versions shows that the constant time implementations of the search procedure `FindFree()` are exactly the same, but the buffer indexing of the DBP is simpler than that of the GDBP.

Apply the assumptions of maximum unit link delay and single task instance, buffer

Data Structures	
`char Read[NINSTS];`	`char UseFreeL[NBUF];`
`message Buf[NBUF];`	`char cur, prev, FreeHd;`
Buffer Management Routines	

`char FindFree() {`
`  char buf_id;`
`  buf_id = FreeHd;`
`  FreeHd = UseFreeL[FreeHd]`
`  return buf_id`
`}`
	`UseDec(char i) {`
	`  if (--UseFreeL[i]) {`
	`  UseFreeL[i] = FreeHd;`
	`  FreeHd = i;`
	`  }`
	`}`

Figure 5.1: Data Structures and Supporting Routines for CTDBP ($\mathtt{D} \leq \mathtt{T}$)

	Writer: maximum unit delays	**Low/High Priority Reader**

activation	`UseDec(prev);`
	`prev = cur;`
	`cur = FindFree();`
	`UseFreeL[cur] = 1;`
		`if (delay[i])`
		`  Read[i] = prev;`
		`else`
		`  Read[i] = cur;`
		`if (isHPR[i]==0)`
		`  UseFreeL[Read[i]]++;`

execution	`Buf[cur] = ···`	`··· = Buf[Read[i]];`
termination		`if (isHPR[i]==0)` `  UseDec(Read[i]);`

Figure 5.2: Writer/Reader Code for CTDBP ($\mathtt{D} \leq \mathtt{T}$)

sizing Equation 4.2 can be simplified as

$$\mathtt{NB} = \mathtt{N} + 2 \tag{5.1}$$

for the DBP, where N is the number of lower priority readers of the writer.

5.1.2 Temporal Concurrency Control Protocol

Similarly, when applying the assumption of maximum unit communication link delay and only single instance per task, the Temporal Concurrency Control Protocol can

be derived from the GTCCP presented in Section 4.2.2. Figures 5.3 and 5.4 illustrate the data structures and buffer indexing procedures of the TCCP. Similarly, comparison shows the buffer indexing of the TCCP is simpler than that of the GTCCP.

For the TCCP, communication buffer can be sized via Equation 4.4 with taking maximum unit link delay into account.

Data Structures	**Buffer Management Routine**

`char cur, FreeB;`
`message Buf[NBUF];`
`char Read[NINSTS];`
	`char FindFree() {`
	`  return (cur+1) % NBUF`
	`}`

Figure 5.3: Data Structures and Supporting Routines for TCCP ($\mathtt{D} \leq \mathtt{T}$)

	Writer: maximum unit delay	**Reader**

activation	`prev = cur;`
	`cur = FindFree();`
		`if (delay[i])`
		`  Read[i] = prev;`
		`else`
		`  Read[i] = cur;`

execution	`Buf[cur] = ···`	`··· = Buf[Read[i]];`

Figure 5.4: Writer/Reader Code for TCCP ($\mathtt{D} \leq \mathtt{T}$)

5.2 Portable Implementation under OSEK API

In a priority-based multi-task implementation with maximum unit communication link delay, the link delay must be equal to one for readers with a priority higher than the writer, while for readers with a lower priority, it can be either zero or one. As discussed in Section 5.1, a pair of variables (`cur`, `prev`) refers to the indices of the latest written item and the previous one.

Note that the protocols presented in Section 5.1 are for single writer and its readers. When considering synchronous reactive models with multiple writers, more data structures need to be introduced. For convenience, Table 5.1 summaries the notations. Variables `cur`, `prev`, `N`, and `WrtInit` are defined for each output port (each writer) in the system. All readers and writers share array `Buf[SysNB]` for communication. The total buffer size required by the system, `SysNB`, is simply the sum of buffer sizes of all writers:

$$\texttt{SysNB} = \sum_{1 \le \texttt{o} \le \texttt{SysNOP}} \texttt{NB}_{\texttt{w}_\texttt{o}}, \tag{5.2}$$

where $\texttt{NB}_{\texttt{w}_\texttt{o}}$ is computed as specified by the DBP (Equation 5.1) or the TCCP (Equation 4.4) protocols, respectively.

`NR`	number of readers	`NT`	number of tasks
`delay`	link delay	`IsHPR`	relative priority
`WrtInit`	initial output value	`pri`	task priority
`Buf[]`	shared comm. buffer	`SysNB`	total buffer size
`SysNIP`	number of input ports	`SysNOP`	number of output ports
`cur`	buffer slot with latest data	`prev`	buffer slot with previous data
`N`	number of lower priority readers	`Read[i]`	buffer slot used by reader `i`

Table 5.1: Notations Used to Describe a System

Based on the descriptions of the SR semantics-preserving protocols in Section 5.1 and the OSEK basics in Section 3.3, portable implementations in BCC1 are presented in the following. Note, only standard features of OSEK are used and no modification to the kernel is required. The implementation that was previously referred as kernel-level will often be performed by the highest priority task or OS hook routines that cannot be interrupted and can therefore guarantee atomicity.

In BCC1 and BCC2, events are not available and the alarm mechanism is the

only way to activate periodic tasks. Since the minimum requirements allow one alarm only, it can be used to periodically activate a dispatcher task that, in turn, activates the application tasks at their respective rates. The dispatcher is periodically activated by an alarm, statically configured as cyclic, every `GCDR` time units, which denotes the Greatest Common Divider of the sampling Rates of application tasks.

5.2.1 Dispatcher, Application, and Initialization Tasks

In this subsection, model implementation tasks are discussed with detailed data structures. For portability purpose, a dispatcher task is used to activate all other periodic application tasks that implement the system functionality. The initialization of all data structures is accomplished via a task called init. The definitions for dispatcher, application tasks, and init for each implementation are presented in the following.

Dispatcher The data structures for the task `dispatcher` are shown and declared in Figures 5.5 and 5.6. The array `TickL[]` has a dimension of `LCMR`, the Least Common Multiple of the sampling Rates of application tasks. Each `TickL[i]` entry has two fields: `DispHd` and `size`. `DispHd` points to the first task on the dispatch table `DTab[]` and `size` indicates the number of tasks that need to be activated at this specific `tick` value. The array `DTab[]` contains the tasks that need to be activated from `tick = 0` to `tick = LCMR-1`. The entries of `DTab[]` are used to index the tasks in the task descriptor array presented later in this subsection.

The top right column in Figure 5.6 is the initialization of the data structures used by the dispatcher task. The bottom part of the figure shows the dispatcher task

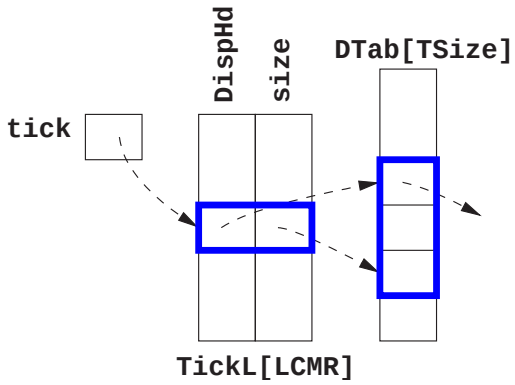

Figure 5.5: Task Dispatcher Data Structures

implementation. The meaning of the variables labeled in front of the implementation will be clear in Chapter 6. During its execution, the counter `tick` is incremented modulo `LCMR`. Then, the value of the field `DispHd` of `TickL[tick]` is checked. If it is "-1", no task needs to be activated. Otherwise, the tasks in `DTab[]`, as specified by the `(DispHd, size)` are processed. For each of them, the dispatcher processes its input and output ports, performing the read and write procedures specified by the earlier protocols. Specifically, it calls `FindFree()` as defined in Figure 5.1 or 5.3 to find a safe buffer slot for each output port. For each reader, according to Figure 5.2 or 5.4, it defines the buffer slot that the task will be using during its execution. Then, dispatcher activates the task by calling OSEK API `ActivateTask` and, at the end, calls `TerminateTask` to terminate.

Instead of constructing a static dispatch table as shown in Figures 5.5 and 5.6, an alternative implementation of the dispatcher is provided in Figure 5.7. A scheduler , e.g., a rate monotonic scheduler (Line 2 in Figure 5.7), can be used to determine those tasks that need to be activated. To achieve this, for each application task, define a corresponding counter. The counters are initialized appropriately as shown in Figure 5.7 so that all

Declaration

```
struct TickEntry {
  char DispHd;
  char size;
} TickL[LCMR];

char tick;
char DTab[TSize];
```

Compute TSize

```
char TSize = 0;
for (i=0; i<NT; i++) {
  TSize += LCMR/TaskL[i].rate;
}
```

Init without Phase Shift

```
tick = -1; i1 = 0;
for (j = 0; j < LCMR; j++) {
  TickL[j].DispHd = -1;
  TickL[j].size = 0;
  for (i = 0; i < NT; i++) {
    if (j % TaskL[i].rate == 0) {
      i2 = TickL[j].size + i1;
      DTab[i2] = i;
      TickL[j].size++;
  } }
  if (TickL[j].size != 0) {
    TickL[j].DispHd = i1;
    i1 += TickL[j].size;
} }
```

Implementation of Task dispatcher

	1	`Task(dispatcher) {`
$C_{k,d,1}$	2	`  tick = (tick+1) % LCMR;`
$C_{k,d,2}$	3	`  if (TickL[tick].DispHd != -1) {`
$C_{k,\tau,1}$	4	`    for (k = 0; k < TickL[tick].size; k++) {`
	5	`      idx = DTab[k+TickL[tick].DispHd]; /* task id */`
	6	`      for (i = 0; i < TaskL[idx].NOP; i++) { /* writers */`
	7	`        idx2 = TaskL[idx].OPHd + i;`
$C_{k,w}$	8	`        ··· /* kernel-level writer code */`
	9	`      }`
	10	`    }`
$C_{k,\tau,2}$	11	`    for (k = 0; k < TickL[tick].size; k++) {`
	12	`      idx = DTab[k+TickL[tick].DispHd]; /* task id */`
	13	`      for (i = 0; i < TaskL[idx].NIP; i++) { /* readers */`
	14	`        idx2 = TaskL[idx].IPHd + i;`
$C_{k,r}$	15	`        ··· /* kernel-level reader code */`
	16	`      }`
$C_{k,\tau,3}$	17	`      ActivateTask(idx);`
	18	`    }`
	19	`  }`
$C_{k,d,3}$	20	`  TerminateTask();`
	21	`}`

Figure 5.6: Declaration, Initialization, and Implementation of Task Dispatcher

Data Structure

```
char period[NT], counter[NT];
```

Initialization

```
/* systemStartup = 1; */
for (i=0; i<NT; i++) {
  counter[i] = period[i] - 1;
}
```

Auxiliary Function

```
RMS(void) {
  for (i=0; i<NT; i++) {
    counter[i]++;
    if (counter[i] == period[i]) {
      counter[i] = 0;
    }
  }
}
```

```
1  Task(dispatcher) {
2    RMS();
3    /* kernel-level index assignment */
4    for (i=0; i<NT; i++) {
5      if (counter[i] == 0) {
6        ··· /* code for all writers of task i */
7      }
8    }
9    for (i=0; i<NT; i++) {
10     if (counter[i] == 0) {
11       ··· /* code for all readers of task i */
12       ActivateTask(i);
13     }
14   }
15   TerminateTask();
16 }
```

Figure 5.7: Alternative Implementation of Task Dispatcher

application tasks are scheduled at system startup. The counters are incremented at the system base rate and reset to 0 when reaching the periods of the corresponding tasks. A zero counter value means that the corresponding task needs to be scheduled for execution.

Application Tasks In the following, implementations of application tasks under the two different communication protocols are defined. The declaration of the data structures that are common to both is summarized in Figure 5.8.

```
#DEFINE LCMR X      #DEFINE SysNIP X    #DEFINE TSize X     #DEFINE NT X
#DEFINE GCDR X      #DEFINE SysNOP X    #DEFINE SysNB X
message Buf[SysNB];                     char Read[SysNIP];
```

Figure 5.8: Common Data Structure Declaration

Implementation of CTDBP Figure 5.9 shows the data structures of the system implementation under the CTDBP. The task descriptor array `TaskL` has an entry for each task specifying its rate, priority, execution flag, and references to its input and output port information. `rate` and `pri` capture sampling rate and static priority information, respectively. A execution flag `done` is used to determine when a context switch is executed upon the termination of a lower priority reader.

The input port descriptor contains the owner task identifier, the communication source port, the link delay, and the relative priority of the reader with respect to its writer. Similarly, the output port descriptor specifies the properties of each output port, including the owner task identifier, the reference to the list of the free entries, `FreeHd`, the `cur` and `prev` variables, and the `BufHd` and `NB` variables. `BufHd` and `NB` specify a contiguous segment in the buffer array `Buf[]` for each output port. The `owner` field in the port descriptors is used

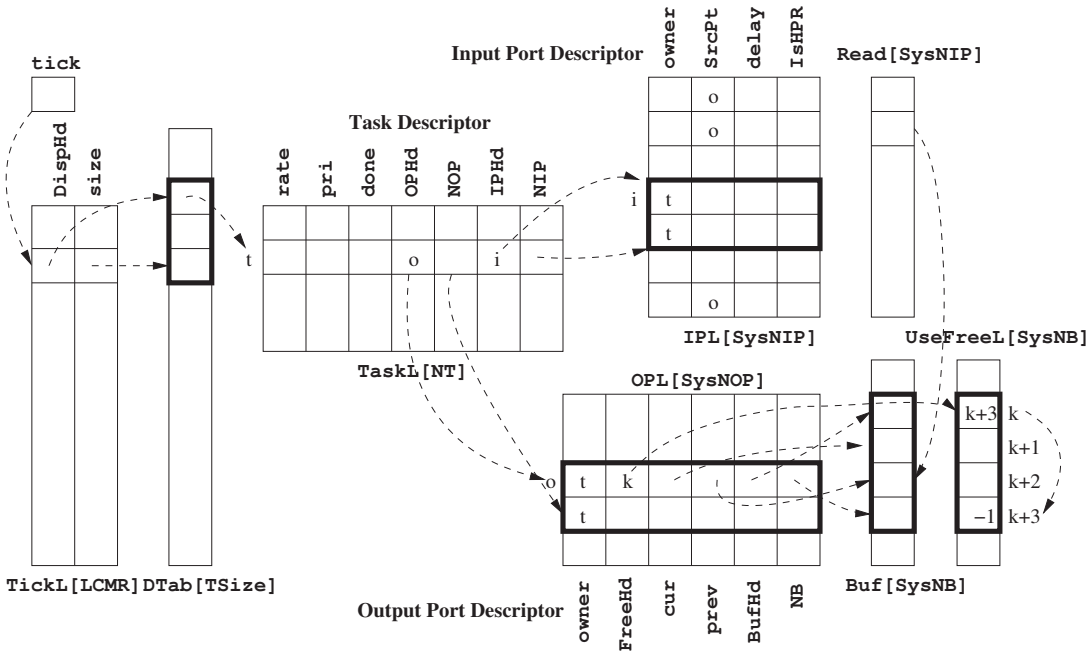

Figure 5.9: Data Structures of System Implementation with CTDBP

to identify the ownership. All the ports that refer to the same task are stored in contiguous locations in the respective port descriptor, as specified by (`OPHd, NOP`) and (`IPHd, NIP`) in the task descriptor. The corresponding declarations are shown in Figures 5.10, 5.8, and 5.6.

In addition, Figure 5.10 shows the data structure initialization details. As shown in Figure 5.11, the communication protocol executes at two levels: on task activation at the

Data Structure

```
struct TaskEntry {
  char rate;
  char pri;
  char done;
  char OPHd;
  char NOP;
  char IPHd;
  char NIP;
} TaskL[NT] = {
  {X,X,0,X,X,X,X}, ···};
```

```
struct OPEntry {
  char owner;
  char FreeHd;
  char cur;
  char prev;
  char BufHd;
  char NB;
} OPL[SysNOP] = {
  {X,0,0,0,0,X}, ···};
char UseFreeL[SysNB];
```

```
struct IPEntry {
  char owner;
  char SrcPt;
  char delay;
  char IsHPR;
} IPL[SysNIP] = {
  {X,X,X,X},
  ···
};
```

Initialization

```
OPL[0].BufHd = 0; /* var and buffer init */
OPL[0].cur = OPL[0].prev = OPL[0].BufHd;
for (i = 1; i < SysNOP; i++) {
  OPL[i].BufHd = OPL[i-1].BufHd + OPL[i-1].NB;
  OPL[i].cur = OPL[i].prev = OPL[i].BufHd;
}
··· /* init buffers buffers accordingly */
for (i = 0; i < SysNOP; i++) { /* init of free list*/
  UseFreeL[OPL[i].BufHd] = 2; /* not free */
  OPL[i].FreeHd = OPL[i].BufHd + 1;
  for (j = 2; j < (OPL[i].NB-1); j++) {
    k = j + OPL[i].BufHd;
    UseFreeL[k] = k + 1;
  }
  UseFreeL[OPL[i].NB-1+OPL[i].BufHd] = -1;
}
```

Figure 5.10: Data Structure Declaration and Initialization for CTDBP

	/* *activation time* */	/* *execution time* */

Ψ	/* *each writer i* */
	UseDec(i, OPL[i].prev);
	OPL[i].prev = OPL[i].cur;
	OPL[i].cur = FindFree(i);
	UseFreeL[OPL[i].cur] = 1;
		···
		/* *each writer k* */
		Buf[OPL[k].cur] = ···
		···
		/* *each reader k* */
		··· = Buf[Read[k]];
		···
		/* *termination time: atomic for LPR* */
		if (IPL[k].IsHPR == 0) {
		t1 = Read[k];
		t2 = IPL[k].SrcPt;
		UseDec(t2,t1);
		}

Γ	/* *each reader i* */
	j = IPL[i].SrcPt;
	if (IPL[i].delay)
	Read[i] = OPL[j].prev;
	else
	Read[i] = OPL[j].cur;
	if (IPL[i].IsHPR == 0)
	UseFreeL[Read[i]]++;	

Auxiliary Functions		

void UseDec(char i, char j){
UseFreeL[j]--;
if(UseFreeL[j] == 0) {
UseFreeL[j] = OPL[i].FreeHd;
OPL[i].FreeHd = j;
} }
		char FindFree(char i) {
		t = OPL[i].FreeHd;
		OPL[i].FreeHd = UseFreeL[t];
		return t;
		} /* O(1) */

Figure 5.11: Implementation of CTDBP

kernel level and during execution by the application task code. At activation time, if the task is a writer, for all its output ports, the `use` count of the buffer item referred by `prev` needs to be decremented. If the new count drops to zero, this buffer slot is freed by updating the free list and the corresponding `FreeHd`. Also, the `cur` index and its corresponding `use` count are updated. On the other hand, if the task is a reader task, for all its input ports, the reading indices are assigned according to the specified communication link delay parameter. For lower priority readers, their corresponding use counts need to be incremented.

The dispatcher task has the same structure as shown in Figure 5.6 and the kernel-level code for the CTDBP from Figure 5.11 is executed.

```
TASK(AppTask_i) {
  TaskL[i].done = false;
  ...
  Buf[OPL[k].cur] = ... /* each writer */
  ...
  ... = Buf[Read[k]];   /* each reader */
  ...
  TaskL[i].done = true;
  /* atomic hook code by PostTaskHook*/
  TerminateTask();
}
```

```
void PostTaskHook(void) {
  char id, j, k, nip;
  GetTaskID(id);
  if (TaskL[id].done) {
    nip = TaskL[id].NIP;
    for (j=0; j<nip; j++) {
      k = j + TaskL[id].IPHd;
      ... /* Critical Section in Fig 5.11 */
    }
  }
}
```

Figure 5.12: OSEK Implementation of Application Task with CTDBP

Figure 5.12 shows the application-level code required by the CTDBP. The hook mechanism provided by OSEK is used to let lower priority readers atomically update the buffer free list at termination time. Specifically, the `PostTaskHook` is used to execute the critical section upon the termination of these tasks. Note that the `PostTaskHook` routine executes at each context switch and for all the tasks in the system. The status flag `done` indicates for which tasks the `PostTaskHook` needs to be executed and also ensures that the operations in the `PostTaskHook` are only executed at task termination time. The `done` flag of each task is set to false at the beginning of the task, and changed to true with the last task statement. The `PostTaskHook`, shown in Figure 5.12, first obtains the identifier of the active task by calling the OSEK API `GetTaskID`. Then, it checks whether its `done` flag is set to true. If so, the updates required by the communication protocol are performed.

A second option to achieve the atomicity of the termination code of lower priority readers is to let the dispatcher execute the termination code as shown in Figure 5.13. At system startup, the termination code segment in the dispatcher is not needed. This is accomplished by checking the flag, `systemStartup`. At system startup, `systemStartup` is

Dispatcher in Figure 5.6	Dispatcher in Figure 5.7

`if (!systemStartup) {`
`  for (k=0; k<TickL[tick].size; k++){`
`  idx = DTab[k+TickL[tick].DispHd];`
`  for (i=0; i<TaskL[idx].NIP; i++){`
`  idx2 = TaskL[idx].IPHd + i;`
`  ···/* termination code in Fig. 5.11 */`
`  }`
`  }`
`  systemStartup = 0;`
`}`
	`if (!systemStartup) {`
	`  for (i=0; i<NT; i++) {`
	`  if (counter[i] == 0) {`
	`  for (j=0; j<TaskL[i].NIP; j++) {`
	`  idx2 = TaskL[i].IPHd + j;`
	`  ···/* termination code in Fig. 5.11 */`
	`  }`
	`  } }`
	`  systemStartup = 0;`
	`}`

Figure 5.13: Achieve Atomicity of Termination Code via Dispatcher

initialized as true as indicated in Figure 5.7 and later on during execution the flag becomes false. The corresponding code segments in Figure 5.13 need to be placed between Lines 3 and 4 in Figure 5.6 and between Lines 2 and 3 in Figure 5.7.

When a task terminates, the task decrements the `use` count of the buffer slot for all the input ports that receive data from a higher priority writer. As discussed earlier, when the `use` count of a buffer slot becomes zero, the slot is returned to the free list and the corresponding writer's `FreeHd` may need to be updated using operations that are not atomic. Since `FreeHd` and `UseFreeL[]` are shared by each writer with its lower priority readers, atomicity of the critical section at termination time must be guaranteed by any correct implementation. The constant time `FindFree()` shown in Figure 5.11 takes the writer's index as input and returns the current `FreeHd` after assigning the index of the second entry on the free list as the new `FreeHd`. Under the DBP buffer sizing, it is guaranteed that `FreeHd` always refers to a valid entry. The memory requirement of the implementation in Figure 5.9 is summarized in Table 5.2.

variable	char	message
count	7×NT+6×SysNOP+5×SysNIP+2×LCMR+TSize+SysNB+1	$SysNB_D$

Table 5.2: Memory Requirement for System Implementation with CTDBP

Implementation of TCCP Figure 5.14 shows the data structures used for the OSEK implementation of the TCCP. Compared with Figure 5.9, the data structures for the TCCP implementation are simpler, since no bookkeeping is required by the `FindFree()`. As discussed in Section 4.2.2, the mechanism based on spatially-in-order writes is used for the buffer sizing in the TCCP. The size of the shared buffer for the system and the buffer size of each writer are computed using Equations 5.2 and 4.4, respectively. Similar to the CTDBP, there are three descriptors to characterize each task, with its input and output ports. The data structure declaration and the initialization code is shown in Figure 5.15. Each writer is assigned a continuous segment of `Buf[]`, identified by the pair `(BufHd,NB)`,

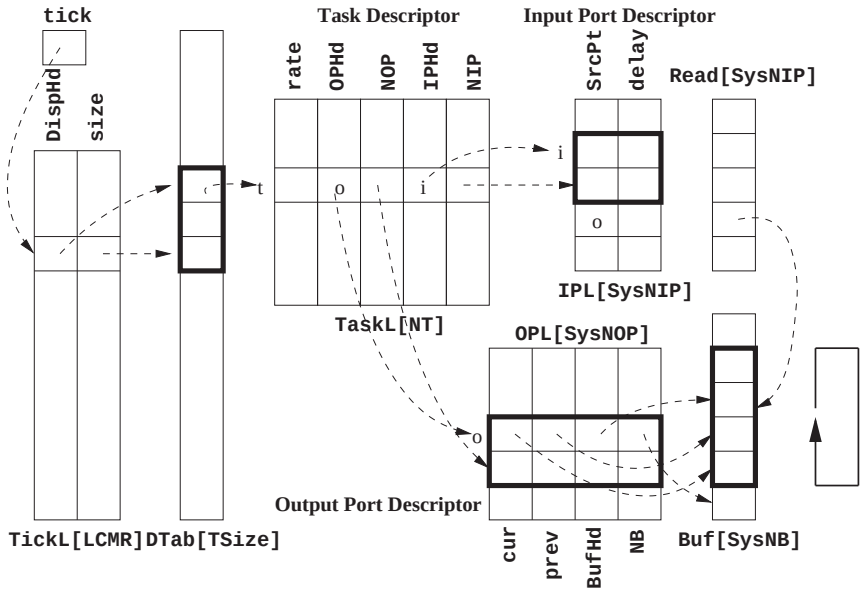

Figure 5.14: Data Structures of System Implementation with TCCP

Data Structure

```
struct TaskEntry {
  char rate;
  char OPHd;
  char NOP;
  char IPHd;
  char NIP;
} TaskL[NT] = {{X,X,X,X,X}, ···};
```

```
struct OPEntry {
  char cur;
  char prev;
  char BufHd;
  char NB;
} OPL[SysNOP] = {
  {0, 0, 0, X}, ···};
```

```
struct IPEntry {
  char SrcPt;
  char delay;
} IPL[SysNIP] = {
  {X, X},
  ···
};
```

Initialization

```
OPL[0].cur = OPL[0].prev = OPL[0].BufHd = 0;
for (i = 1; i < SysNOP; i++) {
  OPL[i].BufHd = OPL[i-1].BufHd + OPL[i-1].NB;
  OPL[i].cur = OPL[i].prev = OPL[i].BufHd;
}
··· /* init buffers accordingly */
```

Figure 5.15: Data Structure Declaration and Initialization for TCCP

as illustrated in Figure 5.14. As shown in the buffer indexing procedure in Figure 5.16, all the task ports need to be processed. The declarations are shown in Figures 5.15, 5.8, and 5.6. Similar to the implementation of the CTDBP discussed earlier in this section, for the TCCP, the dispatcher task executes the kernel-level code and the init task executes the initialization code. As shown in Figure 5.17, the definition of application tasks shares

	/* *activation time* */	/* *execution time* */
Δ	/* *each writer i* */ OPL[i].prev = OPL[i].cur; OPL[i].cur = FindFree(i);	··· /* *each writer k* */ Buf[OPL[k].cur] = ···

Φ	/* *each reader i* */
	i2 = IPL[i].SrcPt;
	if (IPL[i].delay)
	  Read[i] = OPL[i2].prev;
	else
	  Read[i] = OPL[i2].cur;
		···
		/* *each reader k* */
		··· = Buf[Read[k]];
		char FindFree(char idx) {
		  return (OPL[idx].cur+1) % OPL[idx].NB;
		} /* O(1) */

Figure 5.16: Implementation of TCCP

```
TASK(AppTask_i) {
  ...
  /* each writer w */
  Buf[OPL[w].cur] = ...
  ...
  /* each reader r */
  ... = Buf[Read[r]];
  ...
  TerminateTask();
}
```

Figure 5.17: OSEK Implementation of Application Task with TCCP

the same structure with its counterpart in Figure 5.12, but is simpler because there is no termination code for the input ports and hence no hook routine is needed. Unlike the CTDBP, no bookkeeping operation is required for readers at termination time. Since the writer writes data into a circular buffer, the `FindFree()` simply increments `cur` modulo the buffer size, and returns the remainder as the new `cur`. Similar to the CTDBP, because there may be multiple writers in the system, the `FindFree()` takes as argument the index of the writer and returns its queue reference `FreeHd`. The memory requirement of the implementation in Figure 5.14 is summarized in Table 5.3.

variable	**char**	**message**
count	5×NT+4×SysNOP+3×SysNIP+2×LCMR+TSize+1	SysNB_T

Table 5.3: Memory Requirement for System Implementation with TCCP

Initialization Task The data structures of the communication protocols need to be initialized to obtain a correct behavior. In addition, the initialization of the data structures of the task dispatcher in Figure 5.6 is performed by the OSEK task `init`, as shown in Figure 5.18. The init task executes at system startup. The data structures storing static information, such as `isHPR`, etc, are also initialized at system startup. Besides data structure initialization, task init calls OSEK API `SetRelAlarm` to set up the cyclic alarm,

```
TASK(init) {
  ··· /* init implementation specific auxiliary data structure */
  ··· /* init data structure required by protocol */
  ··· /* init dispatcher as in Figure 5.6 */
  SetRelAlarm(dispAlarm, 0, GCDR);  /* set up the relative periodic alarm */
  TerminateTask();
}
```

Figure 5.18: General Structure of Task Init

`dispAlarm`, associated with the task dispatcher. Finally, it calls `TerminateTask` to finish. Note that for both the CTDBP and the TCCP, the init tasks have the same structure of Figure 5.18, but with different data structures to initialize.

Complexity Comparison and Discussion In the following, the two implementations discussed above are compared. According to Tables 5.2 and 5.3, the CTDBP requires more auxiliary data structures than the TCCP, including the counter/free-list structure. The buffer sizes for the CTDBP and the TCCP are not comparable since they are based on completely different buffer sizing mechanisms. Although the CTDBP may lead to a smaller memory requirement [94], the CTDBP implementation is more complex because of the code required for finding a free buffer slot and for accounting for the buffer usage, and the necessity to update the shared use free list. Note that the use of `PostTaskHook` to obtain atomic termination of lower priority readers brings a spatial overhead. Furthermore, since the hook mechanism is mainly designed for debugging and error management, its use also introduces a time overhead at each context switch. Detailed information will be given in Sections 6.1 and 6.2.

5.2.2 OIL Configuration File

In this subsection, the OIL configuration file for the implementation of the communication protocols is defined. Figure 5.19 shows the basic structure of an OIL configuration file. Inside the container `CPU` declaration, objects are statically specified. The application

```
OIL_VERSION = "2.5";
/* Implementation Definition */
IMPLEMENTATION myOSEKOS {
  ...
}; /* End of myOSEKOS */
/* Application Definition */
CPU myCPU { /* container */
  /* OS Object */
  OS myOS {
    STATUS = STANDARD;
    STARTUPHOOK = FALSE;
    ERRORHOOK = FALSE;
    SHUTDOWNHOOK = FALSE;
    PRETASKHOOK = FALSE;
    POSTTASKHOOK = TRUE;
    USEGETSERVICEID = FALSE;
    USERESSCHEDULER = FALSE;
  };
  /* Task Object */
  TASK AppTask_j {
    PRIORITY = X_j;
    SCHEDULE = FULL;
    ACTIVATION = 1;
    AUTOSTART = FALSE;
  };
  ...
  TASK dispatcher {
    PRIORITY = X_d;
    SCHEDULE = NON;
    ACTIVATION = 1;
    AUTOSTART = FALSE;
  };
  TASK init {
    PRIORITY = X_i;
    SCHEDULE = NON;
    ACTIVATION = 1;
    AUTOSTART = TRUE {
      APPMODE = AppMode0;
    };
  };
  /* Alarm Object */
  ALARM dispAlarm {
    COUNTER = SysTimer;
    ACTION = ACTIVATETASK{
      TASK = dispatcher;
    };
    AUTOSTART = TRUE {
      ALARMTIME = 0;
      CYCLETIME = GCDR;
      APPMODE = AppMode0;
    };
  };
  /* Counter Object */
  COUNTER SysTimer {
    MINCYCLE = x;
    MAXALLOWEDVALUE = x;
    TICKSPERBASE = x;
  };
  /* Application Mode Object */
  APPMODE AppMode0 {
    VALUE = AUTO;
  };
}; /* End of myCPU */
```

Figure 5.19: OIL Configuration File

tasks are defined by the generic declaration of `AppTask_j`. The `SCHEDULE` attribute is set as `FULL`, indicating a fully preemptive scheduling policy. Under the assumption that the deadlines of application tasks are not greater than their respective periods, the `ACTIVATION` attribute is set to one (as required in BCC1). Application tasks are periodic and are activated by the task dispatcher, therefore the attribute `AUTOSTART` is set to `FALSE`. For the task init, the configuration is similarly specified, with the `AUTOSTART` attribute turned to on and a single application mode assigned to the `APPMODE` attribute. The task dispatcher activates application tasks and performs part of the communication protocol operations on behalf of the kernel. Therefore, its priority should be higher than those of all application tasks, and its `SCHEDULE` attribute is set to `NON`, indicating a non-preemptive scheduling. The task dispatcher is activated by an alarm, `dispAlarm`, so its `AUTOSTART` attribute is set to `FALSE` and an alarm object is specified accordingly. The alarm is associated with a counter, which is another object defined in the OIL file. The alarm is configured to activate the task dispatcher through setting its attribute `ACTION` as `ACTIVATETASK`. Finally, the alarm's `AUTOSTART` attribute is set to `TRUE` and the period of `dispAlarm` is set to `GCDR`.

When the constant time `FindFree()` of the DBP is used, the atomicity of the termination code that updates the shared use free list may be guaranteed by the `PostTaskHook` mechanism, which is turned on by setting the corresponding attribute `POSTTASKHOOK` as `TRUE` in the OS object as shown in Figure 5.19.

5.3 Automatic Code Generation Support

The Code generation environment used in this dissertation is bases on the Real-Time Workshop. The SR communication functionality (the dispatcher and the communication buffer) is coded in C via the System Function. To obtain the required format for the generated code, system-level Target Language Files need to be modified accordingly. The Embedded Coder produces C code for implementations of system with the SR communication protocols. The SR communication protocols supported with automatic code generation in this dissertation include the Double Buffer (DoB) Protocol as shown in Section 2.1 and the Constant Time Dynamic Buffering (CTDyB) Protocol as discussed in Sections 5.1 and 5.2. Only cases that allow single instance per task are considered. Illustrative examples of systems under the DoB Protocol and the CTDyB Protocol are used to show the generated code. The generated application code together with the source code of the ePICos18 is compiled via the MCC18 compiler and the object files are linked through the MPLINK linker. Finally, the MPLAB SIM is used to emulate the implementation on the PIC18F452 microcontroller execution platform.

5.3.1 Code Generation Environment

The Real-Time Workshop (RTW) [79] extends the capabilities of the Simulink and MATLAB by providing automatic code generation, packaging, and compilation directly from Simulink models. The RTW code generation environment is used from prototyping to deployment and forms the foundation of the RTW product. Two important features of the RTW are code generation for user-created blocks via S-Functions and code customization

flexibility via the Target Language Compiler.

Simulink System Functions The capability of the Simulink environment can be extended by System Functions (S-Function). An S-Function [80] is a description of a Simulink block and can be coded in C, MATLAB, etc. Similar to the built-in Simulink blocks, S-Function blocks use a set of defined APIs to interact with the Simulink engine. S-Functions are compiled by the mex utility and the compilation results are in Mex-files.

After writing an S-Function for an algorithm, the corresponding block can be put in a customer-defined library for future reuse. The user interface of the block can be customized by the masking technique. To customize the code generated for an S-Function block, a corresponding Target Language Compiler file needs to be prepared.

Target Language Compiler As an indispensable part of the RTW, Target Language Compiler (TLC) [78] enables customization of generated code. The RTW build process converts a graphical Simulink model into an intermediate form of the Simulink block diagram, which includes all the model-specific information required for code generation. Then the TLC transforms the intermediate description into target-specific code.

The TLC includes block-level and system-level files. The block-level (block-target) files correspond to the Simulink blocks and the system-level (system-target) files capture model-wide information that specifies header and parameter information for code generation. The TLC files explicitly control the way how RTW generates code and are open source to customers. Therefore, it is very flexible to customize the generated code. One important feature of TLC is that it allows to generate efficient code for customized blocks that are

built via S-Functions. By default, the TLC generates non-inlined code for S-Functions, which incurs overhead due to the existence of a large data structure for each instance of an S-Function block in the model. When a TLC target file does not exist for an S-Function block, its C code is invoked via a function call. Thus, extra run-time overhead is involved whenever functions within an S-Function block are called. The overheads can be eliminated by inlining the S-Function through creating a TLC file for the S-Function. Inlining an S-function directly integrates the S-function block's functionality into the generated code so that it improves the generated code efficiency and reduces memory usage, which is especially important for Embedded Real-Time (ERT) targets.

RTW Embedded Coder The Real-Time Workshop Embedded Coder (RTWEC) [76] is an add-on product of the Simulink code generator. It provides a framework for development of embedded software, aiming at optimization for execution speed and memory usage. The C code generated by the RTWEC conforms to ANSI C and ISO C standards. The RTWEC can generate single-rate or multi-rate code using periodic sampling rates specified in Simulink models. For a multi-tasking implementation of a multi-rate model, it generates separate functions for the base rate and sub-rate tasks via the rate grouping technique. In the following, the RTW environment is extended to support code generation for the SR communication protocols.

5.3.2 SR Communication Protocol Code Generation

Before delving into code generation, one question that needs to be answered is how to handle data transfer at the application task level. For the protocols shown in Fig-

ures 2.2 (the DoB) and 5.2 (the CTDyB), indexing procedures at the kernel and application levels are presented with declared data structures. Note that the DoB Protocol is actually a special case of the DyB Protocol. For a writer and its reader, under the DyB, for communication from high to low priority, when no link delay is defined, the buffer size is 2; for communication from low to high priority, when unit link delay is defined, the buffer size is also 2. This is exactly what the DoB requires. Therefore the implementation presented in Section 5.2.1 is also applicable to the DoB Protocol. As discussed in Section 5.2.1, the kernel-level functionality can be implemented as a task dispatcher. Similar to the rate transition buffer block in Simulink [79], the application-level indexing can be implemented as a communication buffer, which reads in and writes out data at the sampling rates of the writer and reader, respectively.

The next issue to consider is how to manage communication protocol data structures. The principle is to keep data that does not cross S-Function boundaries local and maintain data that has to be shared between the dispatcher and the communication buffer in global memory. The dispatcher writes data into the shared memory while communication buffers read data from the shared memory when needed. For example, for the DoB in Figure 2.2, `wrtIdx` and `rdIdx` are accessed by both the dispatcher and the communication buffer (on behalf of the writer and the reader) and thus they are maintained in shared memory. Similarly, for the CTDyB in Figure 5.2, `cur` and `Read[]` are maintained in shared memory. However, `prev`, `FreeHd`, and `UseFreeL[]` are kept local by the dispatcher. For both mechanisms, `Buf[]` is maintained locally by the communication buffer.

Programming Language C is chosen to program the dispatcher and communication

buffer S-Functions. With a customized interface, the dispatcher and communication buffer blocks are placed in a custom library. Although C-Mex files are sufficient for simulation, to improve efficiency of the generated code, TLC files are prepared to inline their corresponding S-Functions.

Task Dispatcher Figure 5.20 shows the custom SR communication implementation library. The second block and fourth block in the top row are the task dispatchers for the DoB and the CTDyB, respectively. They both have no input port and a variant number of output ports for writing and reading indices. The number of output ports depends on the number of communication buffer blocks used in a model and the number of readers the communication buffer blocks may have. In the library shown in Figure 5.20, with the

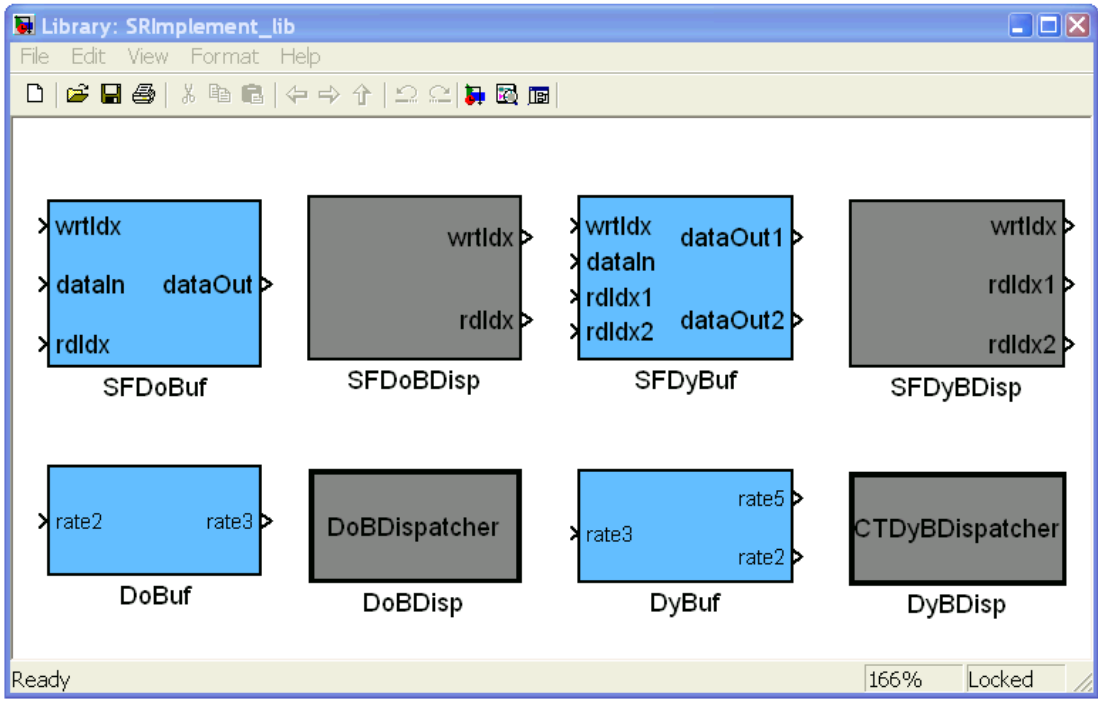

Figure 5.20: SR Communication Implementation Library Blocks

default configuration, the DoB dispatcher assigns indices for a writer and its reader, while the DyB dispatcher assigns indices for a writer and its two readers.

The sampling of the task dispatcher is block-based at the base rate. With the information on the priority transition type and the communication link delays, the dispatcher writes out assigned writing and reading buffer indices at the specified sampling rates of the writers and readers, respectively.

The dispatcher block is constructed as an "atomic" block, which means that the code corresponding to the dispatcher is always generated as a separate function. This is required so that rate grouping would not merge the dispatcher function with the possible application task whose sampling rate is equal to the base rate, which happens when the base rate coincides with the fastest sampling rate specified in the model.

To generate efficient code, a corresponding TLC file, which specifies how the functionality is turned into implementation code, is prepared for the task dispatchers.

Communication Buffer In Figure 5.20, the first block and third block in the top row are the communication buffers for the DoB and the CTDyB, respectively. As discussed earlier, the communication buffer block handles shared data between a writer and its reader(s) at the application task level. The communication buffer block for the DoB has three input ports (writing index, data input, and reading index) and one output port. However, the counterpart for the CTDyB has a variant number of input and output ports, depending on the number of readers it may have. Similar to the DoB case, each reader contributes to one input port (for reading index) and one output port (data output). In the library shown in Figure 5.20, the communication buffer for the CTDyB transfers data between a writer and

its two readers.

Because the sampling rates of the writer and readers are different, the sampling of the communication buffer block needs to be port-based. During execution, when data from the writer comes, the buffer block reads both the data value and the assigned writing index value and stores the data into the buffer slot that is identified by the writing index. Similarly, when a reader is expecting communication data, the buffer block reads in the assigned reading index value and then outputs the data identified by the reading index to the output port with which the reader connects.

To generate efficient code, a TLC file for the communication buffer block needs to be coded in a way compliant to rate grouping so that the rate grouping technique can be used when generating application task functions.

Interaction between Dispatcher and Communication Buffer The last issue to be addressed is how to support writing and reading index sharing between the dispatcher and communication buffer S-Functions. This can be accomplished by using the built-in Simulink block: data store memory. The interaction between the dispatcher block and communication buffer blocks can consequently be supported by using the corresponding data store write and data store read blocks. The dispatcher writes assigned writing and reading indices into the data store memory via the data store write block while on behalf of the writer and the readers the communication buffers obtain the assigned indices through the data store read block.

To make model specification constructing easier, the communication buffer can be masked with its corresponding data store reads into a wrapper block. Similarly, a wrapper

block can be generated by grouping the dispatcher block and its corresponding data store writes. The dispatcher and communication buffer wrapper blocks are shown in the second row in Figure 5.20.

With the integration of the dispatcher, communication buffers, and the data store memory/read/write blocks, the SR communication semantics can always be guaranteed, regardless whether the sampling rates of the writer and the readers are in a harmonic relationship or whether their scheduling phases are the same. This maintains the rigorous semantics of synchronous reactive model-based design for a broader range of application domain.

Model-Wide TLC File Customization The SR communication protocols are implemented using the above discussed C-Mex S-Functions and their corresponding TLC files enable generation of efficient code. However, model-wide TLC files still need to be customized so that code can be generated in the format that is required by the two-level implementation for preserving the SR semantics. In particular, TLC files that control sampling rate scheduling, base rate function generation, and multi-rate scheduler exporting need to be modified to obtain the required code format. Furthermore, to generate OSEK tasks, the *main* function of the application, and the OIL configuration file, an OSEK-specific TLC library file is prepared, which is loaded by the RTWEC at the beginning of code generation.

Code Generation Results In the following, a couple of examplary Simulink models are used to show the generated code of communication protocols with the SR semantics.

Example of the DoB The example shown in Figure 5.21 is a Simulink model using the DoB mechanism. The writer consists of an adder that adds its result at the previous sampling step and a constant of 2. The reader simply corresponds to an output port. The sampling rates of the writer and the reader are specified as 2 and 3 seconds, respectively, which are clearly not harmonic. Hence an implementation using the built-in Simulink rate transition buffer block cannot guarantee deterministic communication. Actually Simulink does not even allow for the configuration of code generation to have deterministic communication for nonharmonic sampling periods. Given the specified sampling rates, the task dispatcher is sampled at a base rate of 1 second. The communication buffer is assumed to take an initial value of 3.

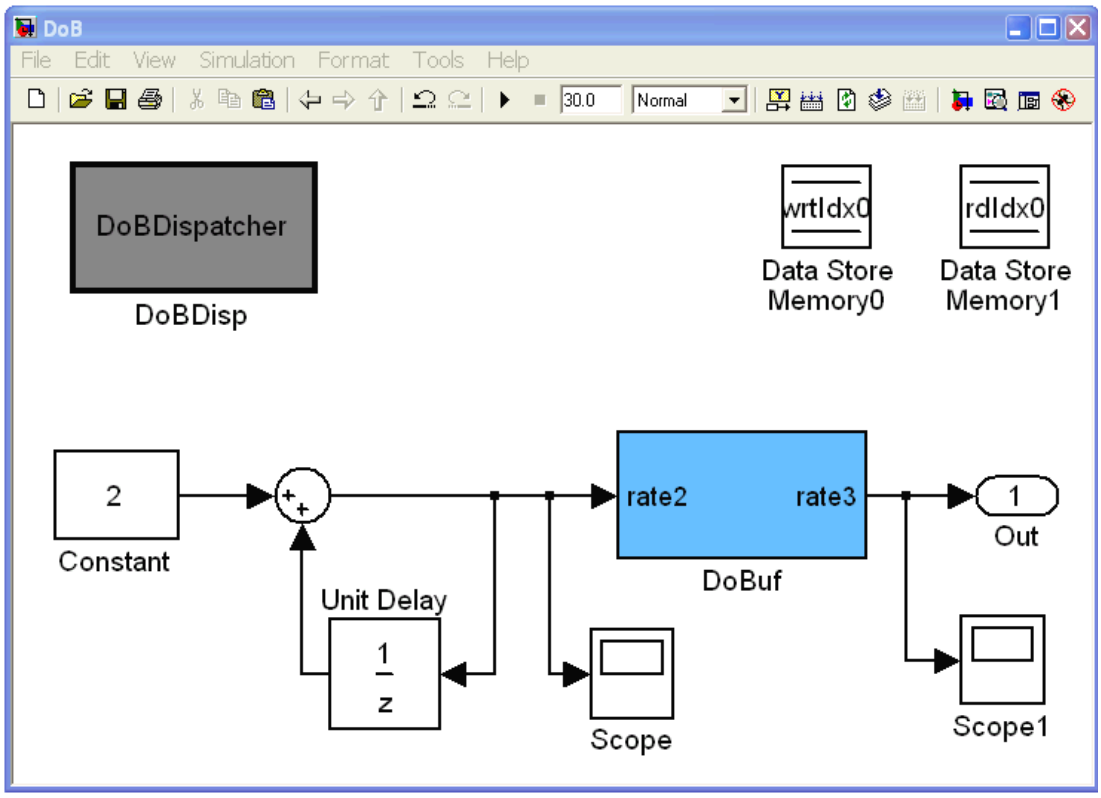

Figure 5.21: Example of the Double Buffer Mechanism

```
/* Using RTW multitasking execution of model */
TASK(dispatcher)
{
  {                                     /* Sample time: [1.0s, 0.0s] */
    rate_monotonic_scheduler();
  }

  /* SubSystem: '<Root>/DoBDisp' */
  DoB_DoBDisp();
  TerminateTask();
}
```

Figure 5.22: Code of the DoB Task Dispatcher

The SR semantics-preserving implementation of the DoB example consists of 4 tasks: dispatcher, init, and two application tasks: subRate_1 and subRate_2. Figure 5.22 shows the generated code of the dispatcher task. Note that the implementation of the dispatcher task in Figure 5.7 is used in the example. At the beginning, a rate monotonic scheduler executes to update the scheduling counter flags of application tasks. Then, the DoB dispatcher function, i.e., `DoB_DoBDisp`, is called to assign the writing index for the higher priority writer according to the DoB mechanism when the writer task subRate_1 needs to be scheduled. Similarly, the reading index is assigned for the reader task subRate_2. Next the dispatcher function calls OSEK API `ActivateTask` to activate the writer and reader tasks as needed. Finally, the dispatcher task calls `TerminateTask` to finish.

Figure 5.23 shows the generated code of the init task. Model specific initialization is performed by the `DoB_initialize` function. Before task init calls `TerminateTask`, it sets up a relative alarm for activating task dispatcher periodically at the base rate. Note that the period of the alarm is 1000 ticks of the system timer that it is attached to (a system timer with a period of 1 msec is assumed in the generated code), and therefore it is exactly 1 second.

```
TASK(init)
{
  /* Initialize model */
  DoB_initialize(1);

  /* Base rate will run every           : 1.0 seconds
     Original rate specified in model   : 1.0 seconds */
  SetRelAlarm(dispatcherAlarm, 1, 1000);
  TerminateTask();
}
```

Figure 5.23: Code of the DoB Task init

As a representative of application tasks, Figure 5.24 shows the generated code of task subRate_2. It first calls the generated step function for sub-rate 2. In function `DoB_step2`, the reading index `rdIdx0`, assigned by the task dispatcher, is read in first. Then, the communication buffer passes the value of the buffer slot identified by `rdIdx0` to the reader. Task subRate_2 finishes by calling OSEK API `TerminateTask`.

```
TASK(subRate_2)
{
  DoB_step2();
  TerminateTask();
}
```

Figure 5.24: Code of Application Task SubRate_2

Example of the CTDyB Figure 5.25 shows a Simulink model that uses the DyB protocol with a constant search procedure to find a free slot. In this example, the communication buffer transfers data between a writer and its two readers. The writer is similar to the one in Figure 5.21 except that the added constant is 1. The sampling rate of the writer is assumed to be 3 seconds. It assumes that the fast reader has a sampling rate of 2 seconds while the slow reader's sampling rate is 5 seconds. Notice that the specified rates

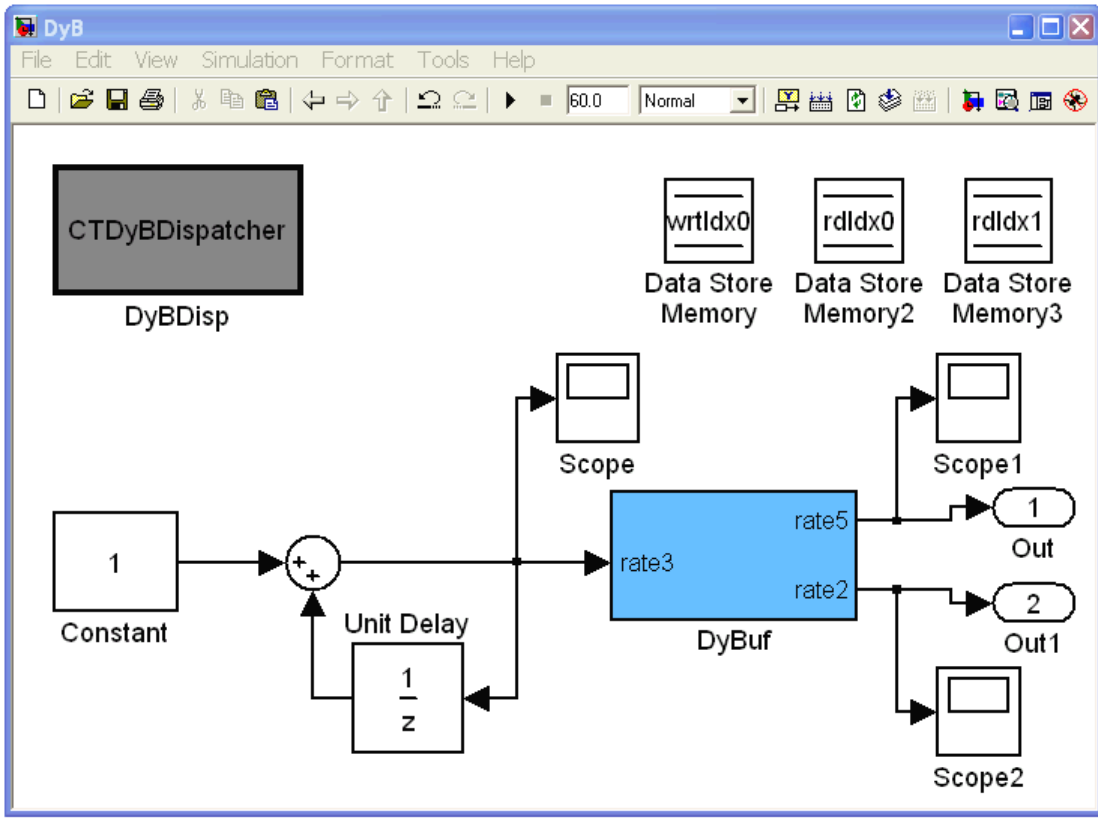

Figure 5.25: Example of the Dynamic Buffering Mechanism

are not in a harmonic relationship. Based on the specified sampling rates, the dispatcher is sampled at a base rate of 1 second. With the assumption of the rate monotonic priority assignment, three buffer slots are needed according the DyB mechanism. The communication buffer is assumed to take an initial value of 8.

The SR semantics-preserving implementation of the DyB example consists of five tasks: dispatcher, init, and three application tasks: subRate_1 (fast reader), subRate_2 (writer), and subRate_3 (slow reader). Due to the structural similarity between the generated code for the CTDyB and the DoB examples, in the following, rather than presenting the detailed generated code for the CTDyB case, only major differences are highlighted.

When the writer task needs to be scheduled, according to the CTDyB, the use count of the buffer slot pointed by `prev` is decremented and the slot is freed if its use count drops to zero. Then, the `prev` is updated and `cur` is assigned with the first free slot on the free list and variable `FreeHd` is updated accordingly. On the reader side, the dispatcher first assigns the reading index for the fast reader as the value of `prev` when it needs to be scheduled. Similarly, at the activation time of the slow reader, its reading index is assigned as the value of `cur` since the writer task has a higher priority. Finally, the dispatcher increments the use count of the slot that assigned to the slow reader. The atomic termination code for the lower priority reader task (subRate_3) is executed at the beginning of the DyB task dispatcher. As discussed in Figures 5.7 and 5.13, at system startup, this segment of code is not needed. The first nested if block checks whether the slow reader task needs to be scheduled. Upon its activation, the dispatcher decrements the use count of the buffer slot read by the slow reader's previous activation and frees the buffer slot when its use count value becomes zero.

5.3.3 Code Validation Environment and Results

The RTW Embedded Coder provides the Software-In-the-Loop (SIL) validation technique for subsystems. The generated code of the SR communication protocol implementations needs to be executed together with an RTOS, but the RTW code validation environment does not have such a capability. In the rest of this section, the code validation environment (the execution microcontroller, the real-time operating system, and the tools) is first discussed and then the corresponding validation results of the above two examples are presented.

Execution Microcontroller: PIC18F452 During execution, only one task can have access to the processing unit, i.e., exclusive access to the processor, the RAM memory, and the hardware stack. Particularly, the PIC18F452 [86] processor is chosen as the underlying execution platform in this dissertation. The PIC18F452 processor is type of RISC and it is an enhanced flash microcontroller with a high performance up to 10MIPS (Millions of Instructions Per Second) operations. The features of the PIC18F452 are shown in Table 5.4. The PIC18F452 processor has a C compiler optimized Instruction Set Architecture (ISA) with 16-bit wide instructions and it supports priority levels for interrupts. Its data path has a width of 8 bits. It has linearly addressable memory: 32 KBytes program memory and 1.5 KBytes data memory, as shown in Figure 5.26. The data memory is implemented as static RAM and it contains Special Function Registers (SFRs) and General Purpose Registers (GPRs). Separate buses are used for data and program memory and therefore concurrent data and instruction accesses are supported. The RESET vector starts at 0000h and the interrupt vectors start at 0008h and 0018h for high priority and low priority interrupts, respectively.

The PIC18 processors manage the hardware stack that is dedicated for function calls via the PUSH/POP instructions. The return address stack is a piece of RAM with 31 words with a width of 21 bits. It supports any combination of up to 31 program

Device	On-Chip Program Memory		On-Chip RAM (Bytes)	Data EEPROM (Bytes)	Interrupt Source	Instruction Set
	FLASH (Bytes)	# of Single Word Instr				
PIC18F452	32K	16K	1.5K	256	18	75

Table 5.4: Features of the PIC18F452 Microcontroller

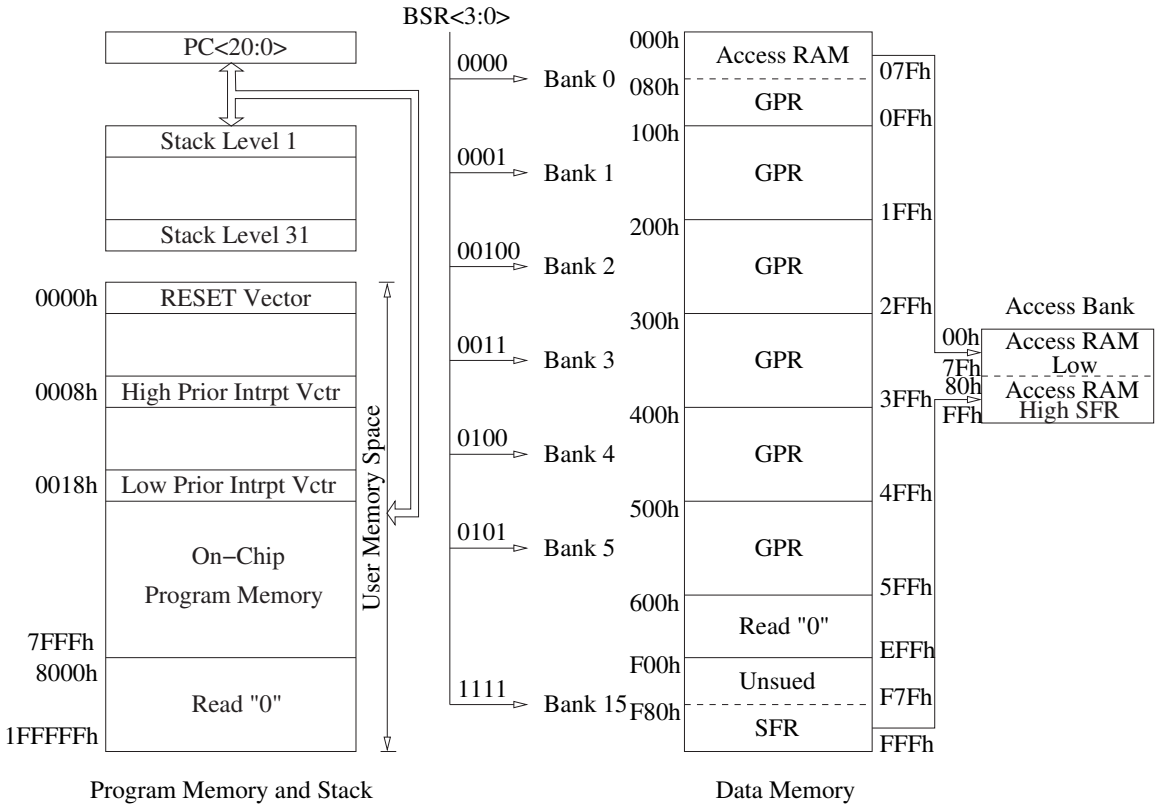

Figure 5.26: Memory Layout of the PIC18F452 Microcontroller

calls and interrupts. Upon execution of CALL/RCALL or acknowledgement of an interrupt, the stack pointer is first incremented and the value of the Program Counter (PC) is pushed onto the stack slot pointed by the stack pointer, while upon execution of RETURN/RETLW/RETFIE, the PC value is popped off the stack and the stack pointer is decremented accordingly.

Based on the clock input, four non-overlapping quadrature clocks (Q1, Q2, Q3, and Q4) are generated via a clock divider as shown in Figure 5.27. An Instruction Cycle (IC) is defined as the four Q cycles (from Q1 to Q4). The program counter is incremented during Q1 and the instruction is fetched from the program memory and latched into the

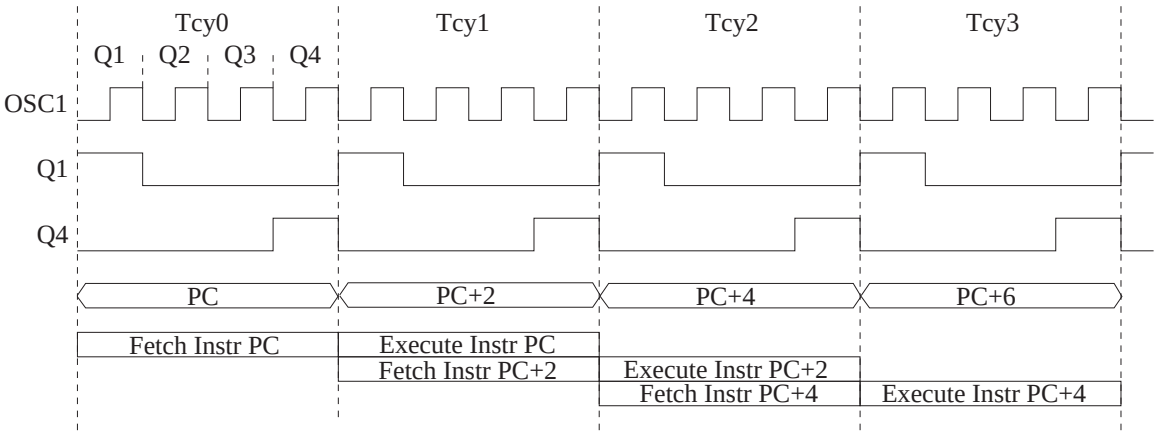

Figure 5.27: Instruction Pipeline of the PIC18F452 Microcontroller

instruction register during Q4. The instruction decoding and execution take place during the next Q1 to Q4. Effectively, each instruction takes one IC due to the pipelining.

The ePICos18 The ePICos18 [120], standing for the enhanced PICos18, is an OSEK-compliant real-time operating system. Based on the PICos18, the ePICos18 provides a constant time kernel scheduler.

The PICos18 The PICos18 [105], developed by Pragmatec SARL Company [106] in France, is a multi-task, preemptive, and real-time kernel that fully conforms to the OSEK standard for the PIC18 family of microcontrollers from the Microchip Technology Inc [82]. Being real-time, the kernel guarantees a deterministic latency for task switching from the current one to another that is more urgent. The kernel is open source and distributed under the terms of the GNU General Public License [41].

In the original implementation of the PICos18 (Version 2.10), the four Task Control Tables (TCT) are not indexed by task identifier. Instead, the task identifiers are stored in

the lower four bits in the table storing task state and identifier. Therefore, all system services related to task (except `GetTaskID`), resource, and event managements need to search through the TCTs to find the right entry. Hence, the execution time depends on the position of the task in the TCTs in RAM. What is even worse is that all of them need to be synchronized, which implies that all four TCTs may need to be updated after sorting (Bubble Sort is used in Version 2.10) is applied upon adjustment based on task priority. To lower RAM consumption, the original implementation does not save static information, for instance, the address and size of the software stack of a task. Unfortunately, it does not save the location of a task in the task descriptor specified in the input file either. During context switches, the kernel has to look for the task linearly through the task descriptor located in ROM. Therefore the search time depends on the place where the task is found in the input task descriptor file. These greatly affect the determinism of a real-time operating system.

Support of a Constant Time Kernel The real-time kernel of the PICos18 is improved in terms of determinism and execution time, which leads to the ePICos18 [120]. The constant time kernel scheduler in the ePICos18 is based on the two-level bitmap technique, which trades off a moderate increase in memory consumption for better kernel service performance. With the support of a constant time kernel, the execution times for the enhanced version of system services for task management, event management, and resource management are independent of the number of application tasks and hence good determinism is achieved.

As the performance analysis results in [120] illustrate, the ePICos18 does not

sacrifice performance: it has a kernel service execution time that is not longer than that of the original PICos18. The ePICos18 is used to support for simulation and validation of application implementation throughout this dissertation.

Development/Validation Environment The Windows-based Microchip compiler toolsuite, MPLAB Integrated Development Environment (IDE), is used as the application development and validation environment. The MPLAB IDE consists of the MPASM Assembler [83], the MCC18 compiler [84], and the MPLINK linker [83]. The MPASM and the MCC18 transform assembly files and C files into relocatable object files, respectively. The MPLAB Object Librarian manages pre-compiled code to be used with the MPLINK object linker. The MPLINK object linker combines all the object files to generate a unique HEX file that can be loaded onto a microprocessor. The compiled application may be analyzed/validated through simulation/emulation [85]. The MPLAB SIM simulator supports code development in a PC-hosted environment and microcontroller simulation at the instruction level. The verification technique is based on the SIL, a common technique used for verification after generating code but before downloading binary to target hardware for execution. Furthermore, the MPLAB ICE 2000 supports enhanced features such as trace, trigger, and data monitoring. The MPLAB In-Circuit Debugger (ICD) is a powerful run-time development tool with the in-circuit debugging capability.

The 40 MHz-10 MIPS PIC18F452 [86] microcontroller from the MicroChip Technology Inc, as discussed above, is used as the target execution hardware platform. The ePICos18 is running on the PIC18F452 microcontroller to provide run-time support for application tasks.

Validation Results To verify that the implemented applications execute as expected, the MPLAB SIM simulator is used to simulate the generated code. During simulation, the SCL (Stimulus Control Language) Workbook feature of the register trace is used to log the input and output(s) of the communication buffer into text files. Then, a program such as matlab or excel is used to plot the simulation results.

MPLAB SIM Results Figure 5.28 shows the MPLAB SIM results of the generated code from the DoB Simulink model with the Double Buffer Protocol as shown in Figure 5.21. The graph shows the output of the writer and the input of the reader versus time (in seconds). The solid curve displays the writer's output. The value of the output increments by 2 at its sampling rate of 2 seconds. The dashed curve in the graph shows that the reader reads the current output value of the writer, which is expected because the reader has a lower priority than the writer.

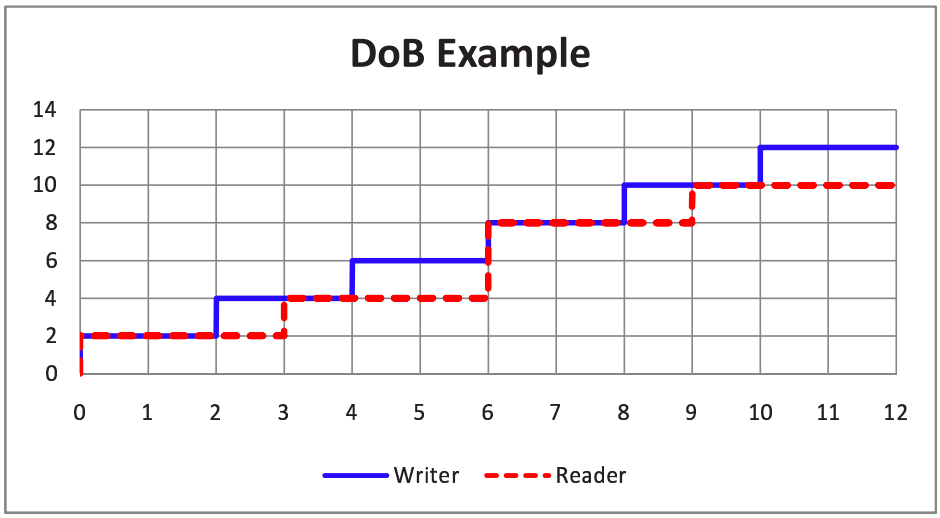

Figure 5.28: MPLAB SIM Result of the DoB Example

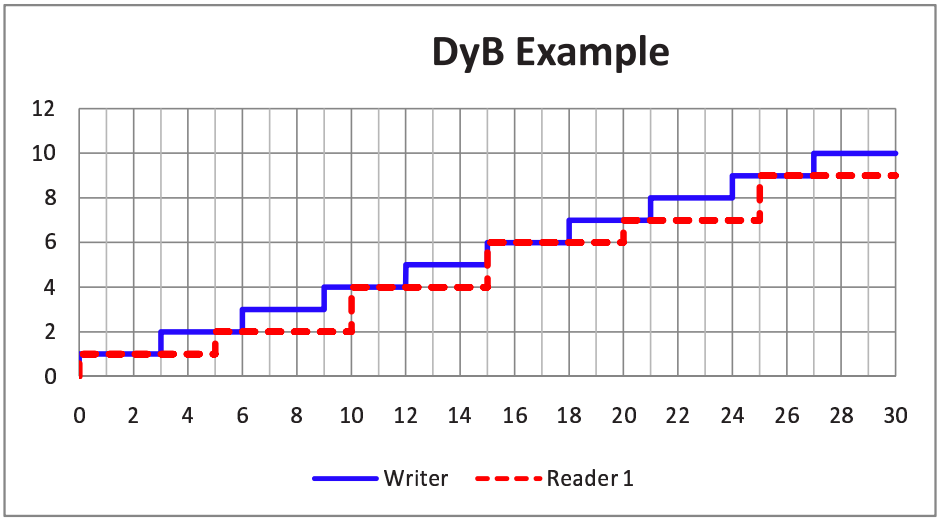

Figure 5.29: MPLAB SIM Result of the DyB Example (1)

Similarly, Figures 5.29 and 5.30 show the MPLAB SIM results of the generated code from the DyB Simulink model with the Constant Time Dynamic Buffering Protocol as shown in Figure 5.25. The solid curve in Figures 5.29 and 5.30 is the writer's output. The

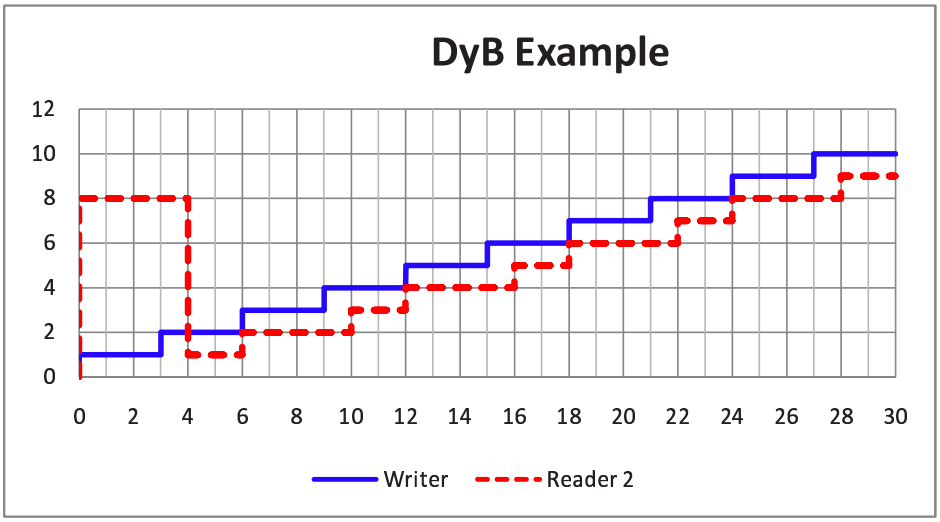

Figure 5.30: MPLAB SIM Result of the DyB Example (2)

value of the output increments by one at a sampling rate of 3 seconds. The dashed curve in Figures 5.29 shows the input of reader 1, the slow reader in the DyB example. This case is similar to what is shown in Figure 5.28 since the communication is from a higher priority writer to a lower priority reader.

Different from what is shown in 5.29, the dashed curve in Figures 5.30 shows the input of reader 2, the fast reader in the DyB example. Because the reader has a higher priority than the writer and the communication link has a unit delay, the first two instances of the fast reader read 8, which is the initial value of the buffer, before the first instance of the writer finishes its execution. During the following sampling intervals, reader 2 always reads the output value of the writer with a unit delay. The simulation results show that the implemented models execute as expected at run time and therefore the synchronous reactive communication semantics is preserved.

RTW SIM Results It is natural to simulate a Simulink model at design time to verify the correctness of the behavior. Comparing the simulation results from the RTW SIM and the MPLAB SIM can also validate the generated code.

Figure 5.31 shows the RTW SIM results of the DoB Simulink model as shown in Figure 5.21. The results from Scope and Scope1 are the writer output and the reader input, respectively. Figure 5.32 shows the RTW SIM result of the DyB Simulink model as shown in Figure 5.25. The results from Scope, Scope1, and Scope2 are the writer output, the slow reader input, and the fast reader input, respectively.

Scrutinization and comparison of the MPLAB SIM results and the RTW SIM results show that they are indeed match each other behaviorally. However, there is actually

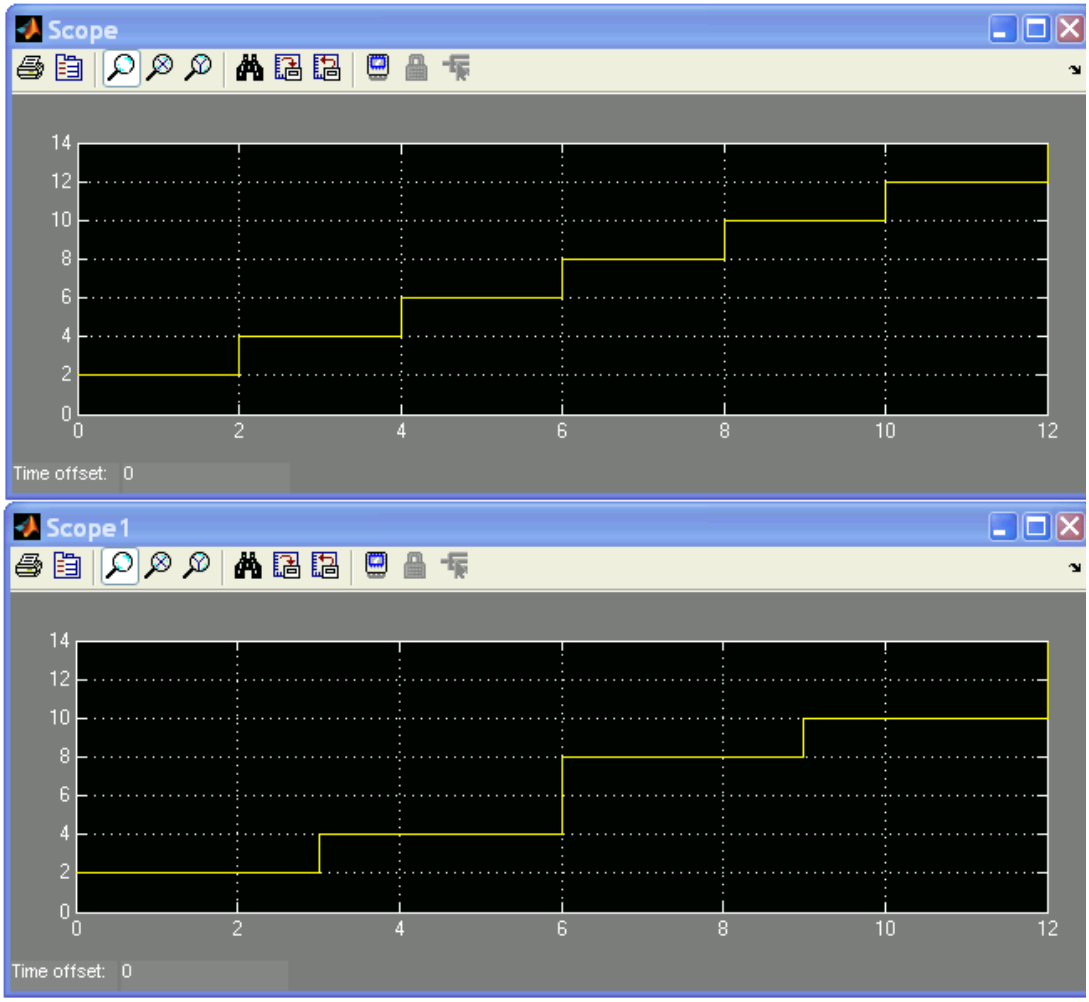

Figure 5.31: RTW SIM Result of the DoB Example

a fundamental difference between these two sets of simulation results. The execution takes zero time in the RTW SIM, but this is not true in the MPLAB SIM. Because the execution times of the illustrating example models are in microseconds/milliseconds and the unit used in the figures is second, the execution of the MPLAB SIM seems to take no time. As a matter of fact, writing and reading occur shortly after their respective sampling time instants.

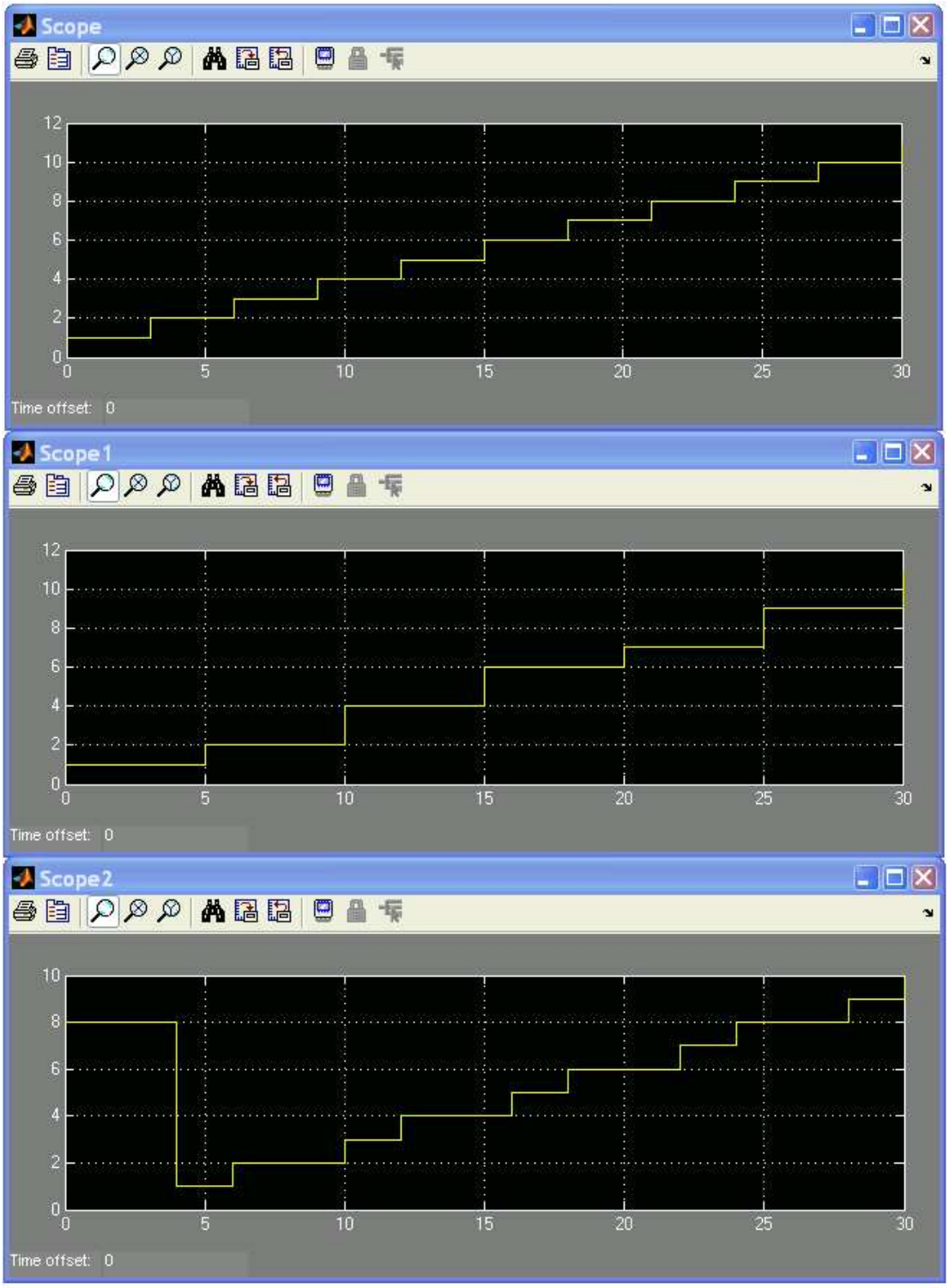

Figure 5.32: RTW SIM Result of the DyB Example

Chapter 6

Buffer Sizing Optimization with Timing Constraints

As discussed in Section 4.3, a better buffer bound may be obtained through using a hybrid communication scheme between a writer and it readers. The basic idea behind it is that the Temporal Concurrency Control Protocol is good for fast readers while the Dynamic Buffer Protocol is good for slow readers. The improved bound on buffer size is given by Equations 4.17 and 4.19 in Section 4.3. But in practice, the optimization process is more complex because it further involves scheduling overheads due to context switches. Furthermore, the CTDBP may take a longer time to find a safe buffer slot than the TCCP and this may lead to scheduling constraint violation. These issues have not been addressed before and this chapter is dedicated for this purpose. In this chapter, it first formulates the buffer sizing optimization problem under timing constraints. With the measured execution time for context switches and communication protocols on the PIC18F452 microcontroller

and the ePICos18, the formulation is reformulated as a mixed integer linear programming problem, which is applied to test cases that are either generated randomly or derived from an automotive industrial application. The optimization problems are solved by glpsol/cplex solvers and results are compared.

Unlike Section 4.3 which addresses a general scenario (i.e., possibly multiple active instances per task), only cases that allow single task instance are considered in this chapter. However, it is clear that with more computational efforts, the same optimization technique applies to the general scenarios in Section 4.3.

6.1 Buffer Sizing Optimization

The two protocols presented in Section 5.1 have different characteristics. The CT-DBP needs a relatively long time to find a safe buffer slot for the writer to write into, while the TCCP is fast but can be quite expensive in terms of memory requirement when there are readers with long data lifetimes. A mixed scheme [94] may be used to reduce memory consumption. However, when taking into account the temporal cost for the implementation of the protocols, the choice of the best communication scheme depends upon the temporal constraints of the system. There exists the opportunity to minimize the buffer bounds by automatically choosing inter-task communication mechanisms under timing constraints.

Consider a design problem consisting of a set of communicating tasks with hard real-time constraints ($\mathtt{R} \leq \mathtt{D} \leq \mathtt{T}$) and scheduled with fixed priorities. Each task implements a set of communicating reader and/or writer blocks. The problem is to minimize the memory space required for the communication buffers by partitioning the reader blocks

into two groups, slow reader blocks, for which the CTDBP should be used, and fast reader blocks, communicating through the TCCP.

Assume that the atomicity of the termination code of lower priority readers is achieved by the hook mechanism in case of the CTDBP. As discussed in [122][121][94], the auxiliary data structures for CTDBP and TCCP are similar and the memory increase due to `PostTaskHook` is small. Specifically, e.g., the footprint overhead due to the `PostTaskHook` in the final image of system implementation under the CTDBP is 93 Bytes. So they will not be considered in the formulation of the optimization problem.

6.1.1 Parameters, Variables, Cost Function, and Timing Constraints

The following description of the variables and constraints of the optimization problem formulation applies to each subsystem consisting of a writer $\mathtt{i}$ and its readers $\mathtt{j}$.

Parameters Parameters $\mathtt{p}_{\mathtt{i},\mathtt{j}}$ and $\mathtt{L}_{\mathtt{i},\mathtt{j}}$ are introduced to capture the priority ordering of tasks and the communication topology of the system, respectively, as follows:

$$\mathtt{p}_{\mathtt{i},\mathtt{j}} = \begin{cases} 1 & \text{if } \pi_{\mathtt{i}} > \pi_{\mathtt{j}} \\ 0 & \text{otherwise} \end{cases}$$

and

$$\mathtt{L}_{\mathtt{i},\mathtt{j}} = \begin{cases} 1 & \text{if } \mathtt{w}_{\mathtt{i}} \longrightarrow \mathtt{r}_{\mathtt{j}} \\ 0 & \text{otherwise.} \end{cases}$$

Optimization variables A boolean variable $\mathtt{x_{i,j}}$ is introduced for each pair of writer and reader to define the communication mechanism to be used, i.e.,

$$\mathtt{x_{i,j}} = \begin{cases} 0 & \text{if } \mathtt{w_i} \xrightarrow{\text{TCCP}} \mathtt{r_j} \\ 1 & \text{if } \mathtt{w_i} \xrightarrow{\text{CTDBP}} \mathtt{r_j}. \end{cases}$$

For each writer $\mathtt{w_i}$, the number of slow and fast readers can be easily computed as:

$$\mathtt{NRS_i} = \sum_{\mathtt{j=1}}^{\mathtt{NR}} \mathtt{x_{i,j}} \mathtt{L_{i,j}} \quad \text{and} \quad \mathtt{NRF_i} = \mathtt{NR_i} - \mathtt{NRS_i}.$$

Of those, the number of slow readers with a lower priority than that of the writer is:

$$\mathtt{NS_i} = \sum_{\mathtt{j=1}}^{\mathtt{NR}} \mathtt{x_{i,j}} \mathtt{p}_{\mathtt{w}^{(i)},\mathtt{r}^{(j)}} \mathtt{L_{i,j}},$$

where $\mathtt{w}^{(i)}$ and $\mathtt{r}^{(i)}$ stand for the identifier of the task implementing the writer and reader block $\mathtt{i}$, respectively.

Correspondingly, based on Equations 4.2 and 5.1, the buffer size that is required for the implementation of the communication for the slow readers is $\mathtt{NBS_i}$, computed as:

$$\mathtt{NBS_i} = \begin{cases} \mathtt{NS_i} + 1 + \max\limits_{1 \le \mathtt{j} \le \mathtt{NR}} \mathtt{x_{i,j}} \mathtt{delay[j]} \mathtt{L_{i,j}} & \text{if } \mathtt{NRS_i} > 0 \\ 0 & \text{if } \mathtt{NRS_i} = 0, \end{cases}$$

and based on Equation 4.4, the buffer size for the implementation of the communication for the fast readers is:

$$\mathtt{NBF_i} = \max_{1 \le \mathtt{j} \le \mathtt{NR}} \left(1 - \mathtt{x_{i,j}}\right) \left\lceil \frac{\mathtt{l_j}}{\mathtt{T}_{\mathtt{w}^{(i)}}} \right\rceil \mathtt{L_{i,j}}.$$

Cost function The cost function of the formulated optimization problem is defined as the total buffer size for the system:

$$\mathtt{NB} = \sum_{\mathtt{i=1}}^{\mathtt{NW}} \mathtt{NB_i} = \sum_{\mathtt{i=1}}^{\mathtt{NW}} (\mathtt{NBF_i} + \mathtt{NBS_i}).$$

Timing constraints The deadline constraints required by schedulability can be simply expressed as:

$$\forall \tau_i, R_i \leq D_i.$$

In the evaluation of R_i, Equation 4.1 needs to be modified with accounting for the scheduling overhead (context switch overhead) as following:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + CSO. \tag{6.1}$$

Note that the set of higher priority tasks $hp(i)$ in Equation 6.1 also includes the dispatcher. Furthermore, R_i depends on C_k, the worst-case execution time of the protocol at the kernel level, which in turn depends on the type of communication mechanism. In the following, how the communication implementation affects R_i is evaluated. Two more intermediate boolean variables are introduced:

$$\lambda_i = \min(1, NRS_i) \quad \text{and} \quad \rho_i = \min(1, NRF_i).$$

$\lambda_i = 0$ and $\rho_i = 0$ mean that writer i communicates with all its readers using the TCCP and the CTDBP, respectively. When $\lambda_i = \rho_i = 1$, writer i uses a hybrid protocol with all its readers. From the definition, $\lambda_i + \rho_i > 0$.

The update of the communication data structures (buffer indices) is performed by the kernel at the highest priority level for each task at the time of its activation. Hence, it can be modeled in the response time formula of each task as a set of additional interference terms, one for each task, carrying the corresponding overhead. Then, the interference portion in Equation 6.1 becomes

$$\sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j = \sum_{j \in (hp'(i) = hp(i) \setminus \{d\})} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \left\lceil \frac{R_i}{T_d} \right\rceil C_d,$$

where $\mathtt{d}$ identifies the dispatcher task. Assume that the dispatcher task implementation in Figure 5.6 is chosen. To ease characterization, the dispatcher is partitioned into three segments as labeled in Figure 5.6. The first segment accounts for the minimum execution time ($\mathtt{C_{k,d}}$) when no task needs to be activated. It further includes three parts: the update of `tick` ($\mathtt{C_{k,d,1}}$) at Line 2, the `if` statement ($\mathtt{C_{k,d,2}}$) at Line 3, and the call to `TerminateTask` ($\mathtt{C_{k,d,3}}$) at Line 20. The second segment corresponds to the minimum execution time ($\mathtt{C_{k,\tau}}$) associated with the task that needs to be activated. It is also composed of three portions: the setup and iteration termination check of the `for` loops to process writers/readers ($\mathtt{C_{k,\tau,1}}$)/($\mathtt{C_{k,\tau,2}}$) at Lines (4-7)/(11-14) and the call to `ActivateTask` ($\mathtt{C_{k,\tau,3}}$) at Line 17. $\mathtt{C_{k,\tau}}$ is the execution time of a task with a single writer and reader that needs to be activated but excludes the cost for index assignments. The third segment is the index assignment cost ($\mathtt{C_{k,w}}$/$\mathtt{C_{k,r}}$) for a single writer/reader at Line 8/15, characterizing the temporal cost of the communication protocol.

Based on the above characterization of the dispatcher task, further expansion of the last term in the above equation gives:

$$\sum_{\mathtt{q}=1}^{\mathtt{NR}} \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_{r(q)}}} \right\rceil \mathtt{C_{k,r}} + \sum_{\mathtt{j}=1}^{\mathtt{NW}} \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_{w(j)}}} \right\rceil \mathtt{C_{k,w}} + \sum_{\mathtt{j}=1}^{\mathtt{NT}} \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_j}} \right\rceil \mathtt{C_{k,\tau}} + \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_d}} \right\rceil \mathtt{C_{k,d}},$$

The execution times for reading and writing are:

$$\mathtt{C_{k,r}} = \Gamma \mathtt{x_{\mathcal{S}(q),q}} + \Phi(1 - \mathtt{x_{\mathcal{S}(q),q}}) \text{ and } \mathtt{C_{k,w}} = \Psi\lambda_\mathtt{j} + \Delta\rho_\mathtt{j},$$

respectively, where $\mathcal{S}(\mathtt{q})$ returns the identifier of the writer that communicates with reader $\mathtt{q}$, Ψ/Γ represent the execution time for assigning writer/reader indices for the CTDBP as shown in Figure 5.11, and Δ/Φ stand for the execution time for assigning writer/reader indices for the TCCP as shown in Figure 5.16.

The cost of context switches, $\mathtt{CSO}$ in Equation 6.1 can be computed as the sum of context switch overheads due to the higher priority application tasks, the system clock interrupt handler, and the task dispatcher.

$$\mathtt{CSO} = \sum_{\mathtt{j} \in \mathtt{hp'(i)}} \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_j}} \right\rceil \left(\mathtt{CS_{i,j}} + \mathtt{CS_{j,ter}}\right) + \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_{clk}}} \right\rceil \left(\mathtt{CS_{i,clk}} + \mathtt{CS_{clk,ter}}\right) + \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_d}} \right\rceil \left(\mathtt{CS_{clk,d}} + \mathtt{CS_{d,ter}}\right),$$

where $\mathtt{CS_{i,j}}$ stands for the cost due to context switch from $\mathtt{i}$ to $\mathtt{j}$. $\mathtt{clk}$ and $\mathtt{ter}$ represent the clock handler and task termination, respectively.

Assume in case of the CTDBP the atomicity of the termination code for lower priority readers is obtained via the hook mechanism. To model the overhead of $\mathtt{PostTaskHook}$, two more additional boolean variables are introduced for the application under study:

$$\lambda = \min(\mathtt{1}, \sum_{\mathtt{i=1}}^{\mathtt{NW}} \lambda_{\mathtt{i}}) \quad \text{and} \quad \rho = \min(\mathtt{1}, \sum_{\mathtt{i=1}}^{\mathtt{NW}} \rho_{\mathtt{i}}).$$

When $\lambda = \rho = \mathtt{1}$, both the TCCP and the CTDBP are used in the system, implying that a $\mathtt{PostTaskHook}$ routine is used. Therefore, $\mathtt{C_{k,\tau}}$ is computed as:

$$\mathtt{C_{k,\tau}} = \lambda \mathtt{C^D_{k,\tau}} + \rho(\mathtt{1} - \lambda)\mathtt{C^T_{k,\tau}}.$$

Similarly the cost of each type of context switch is:

$$\mathtt{CS} = \lambda \mathtt{CS^D} + \rho(\mathtt{1} - \lambda)\mathtt{CS^T}.$$

6.1.2 Complete Formulation

In summary, the complete formulation of the optimization problem is as follows:

$\mathtt{minimize}$

$$\mathtt{NB} = \sum_{\mathtt{i=1}}^{\mathtt{NW}} (\mathtt{NBF_i} + \mathtt{NBS_i})$$

such that

for all tasks $\forall \mathtt{i} \in \mathcal{T}$

$$\mathtt{R_i} \leq \mathtt{D_i}$$

$$\begin{aligned}\mathtt{R_i} = \mathtt{C_i} &+ \sum_{\mathtt{j} \in \mathtt{hp'(i)}} \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_j}} \right\rceil \left(\mathtt{C_j} + \mathtt{CS_{i,j}} + \mathtt{CS_{j,ter}}\right) \\ &+ \sum_{\mathtt{q}=1}^{\mathtt{NR}} \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_{r^{(q)}}}} \right\rceil \left(\Gamma \mathtt{x_{\mathcal{S}(q),q}} + \Phi(1 - \mathtt{x_{\mathcal{S}(q),q}})\right) \\ &+ \sum_{\mathtt{j}=1}^{\mathtt{NW}} \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_{w^{(j)}}}} \right\rceil \left(\Psi \lambda_\mathtt{j} + \Delta \rho_\mathtt{j}\right) + \sum_{\mathtt{j}=1}^{\mathtt{NT}} \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_j}} \right\rceil \mathtt{C_{k,\tau}} \\ &+ \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_d}} \right\rceil (\mathtt{C_{k,d}} + \mathtt{CS_{clk,d}} + \mathtt{CS_{d,ter}}) + \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_{clk}}} \right\rceil (\mathtt{CS_{i,clk}} + \mathtt{CS_{clk,ter}})\end{aligned} \tag{6.2}$$

for all reader tasks $\forall \mathtt{i} \in \mathcal{R}$

$$\mathtt{l_i} = \mathtt{delay[i]T_{w^{(\mathcal{S}(i))}}} + \mathtt{O_{\mathcal{S}(i),i}} + \mathtt{R_{r^{(i)}}}$$

for all writer tasks $\forall \mathtt{i} \in \mathcal{W}$

$$\mathtt{NBF_i} = \max_{1 \leq \mathtt{j} \leq \mathtt{NR}} \left(1 - \mathtt{x_{i,j}}\right) \left\lceil \frac{\mathtt{l_j}}{\mathtt{T_{w^{(i)}}}} \right\rceil \mathtt{L_{i,j}} \tag{6.3}$$

$$\mathtt{NBS_i} = \lambda_\mathtt{i} \left(\sum_{\mathtt{j}=1}^{\mathtt{NR}} \left(\mathtt{x_{i,j} p_{w^{(i)},r^{(j)}} L_{i,j}} \right) + 1 + \kappa_\mathtt{i} \right) \tag{6.4}$$

$$\kappa_\mathtt{i} = \mathtt{max}_{1 \leq \mathtt{j} \leq \mathtt{NR}} \left(\mathtt{delay[j] x_{i,j} L_{i,j}} \right) \tag{6.5}$$

$$\lambda_\mathtt{i} = \min \left(1, \sum_{\mathtt{j}=1}^{\mathtt{NR}} \mathtt{x_{i,j} L_{i,j}} \right) \tag{6.6}$$

$$\rho_\mathtt{i} = \min \left(1, \mathtt{NR_i} - \sum_{\mathtt{j}=1}^{\mathtt{NR}} \mathtt{x_{i,j} L_{i,j}} \right) \tag{6.7}$$

$$\lambda = \min\left(1, \sum_{i=1}^{NW} \lambda_i\right), \qquad \rho = \min\left(1, \sum_{i=1}^{NW} \rho_i\right) \tag{6.8}$$

$$C_{k,\tau} = \lambda C^{D}_{k,\tau} + \rho(1-\lambda)C^{T}_{k,\tau} \tag{6.9}$$

$$CS = \lambda CS^{D} + \rho(1-\lambda)CS^{T} \tag{6.10}$$

The context switch overheads CS depend on the underlying real-time operating system. In the next section, an evaluation of the time costs and overheads that need to be associated to the protocol implementation alternatives for an OSEK-compliant operating system is presented. These data are used to complete the formulation of the problem, to derive its MILP reformulation, and to evaluate the opportunity for optimization in the experimental section.

6.2 ePICos18-Based Evaluation

In this section, the performance of the implementations presented in Section 5.2 is evaluated using the ePICos18 executing on the 40 MHz-10 MIPS PIC18F452 [86] microprocessor under the Microchip MPLAB Integrated Development Environment. Compiled applications are analyzed through the MPLAB SIM simulator at the instruction level and the MPLAB ICE 2000. Due to pipelining, an instruction takes one Instruction Cycle (IC), which corresponds to 0.1 μsec. For performance evaluation, the cycle of the software system timer used to manage alarms and counters in the ePICos18 is assumed to be 5 msec.

6.2.1 Application Tasks and Dispatcher

In this subsection, performance analysis is conducted for application implementation tasks and the dispatcher task that is used to activate application tasks during execution.

Application Tasks Table 6.1 summarizes the performance overhead comparison of application tasks when using the TCCP and the CTDBP. As shown in Figure 5.17, no bookkeeping code is required for application tasks under the TCCP. An application task with the CTDBP executes also fast, but the kernel performs the required functionality in a post task hook routine on behalf of the application task. For the application tasks under the CTDBP shown in Figure 5.12, measurements show that the costs to reset/set the flag `done` at the beginning and at the end of the task executions are 23 ICs and 25 ICs, respectively. Analysis on the hook routine in Figure 5.12 shows that its cost is 43 ICs if no application task is executed before it is called, and 77 ICs if some application task was executed but did not terminate when the call was made; $(115+233\times \mathtt{N}^{\tau}+96\times \mathtt{M}^{\tau})$ ICs in the worst case if some application task just terminated and triggers the call, where $\mathtt{N}^{\tau}$ and $\mathtt{M}^{\tau}$ are the number of lower and higher priority readers of task τ, respectively. The dispatcher is not an application task and the overhead imposed on it by `PostTaskHook` is always 43 ICs. Table 6.1 illustrates the time advantage of the TCCP with respect to the implementation of application tasks.

	Overhead in Application Task	`PostTaskHook`
TCCP	0	0
CTDBP	48	$115+233\times \mathtt{N}^{\tau}+96\times \mathtt{M}^{\tau}$

Table 6.1: Application Task Overhead Comparison (in ICs)

i	-	$C_{k,i,1}$	$C_{k,i,2}$	$C_{k,i,3}$	$C_{k,i} = \sum_{j=1}^{3} C_{k,i,j}$	$C_{k,w}$	$C_{k,r}$
d	-	14	17	269	300	-	-
τ	TCCP	127	127	253	507	$\Delta = 181$	$\Phi = 133$
	CTDBP	127	127	296	550	$\Psi = 365$	$\Gamma = 193$

Table 6.2: Dispatcher Performance Evaluation (in ICs)

Task Dispatcher Table 6.2 shows the breakdown measurement of the dispatchers with the TCCP and the CTDBP. Because both versions of the dispatcher share exactly the same structure, their $C_{k,d}$ values are 300 ICs. Note that there is no bookkeeping code in the implementation of application tasks when using the TCCP. $C_{k,\tau,3}$ of the TCCP is the execution time of `ActivateTask`, which is equal to 253 ICs [120]. The difference of 43 ICs between the CTDBP and the TCCP is due to the call to `PostTaskHook` in the scheduler when calling `ActivateTask` inside the CTDBP dispatcher. This overhead is about 17%. The last two columns of Table 6.2 show the measured performance for kernel level assignment of writing and reading indices. The $C_{k,w}$ and $C_{k,r}$ characterize the temporal cost of the communication protocol. The results demonstrate that the TCCP is 102% and 45% faster than the CTDBP for writer and reader processing at the kernel level, respectively. Clearly, when taking into account the memory requirements for different protocols, an appropriate communication protocol is highly dependant upon the temporal properties of the system.

6.2.2 Context Switch Latency of ePICos18

Analysis in [120] shows that the costs to save and restore the context is (308+12×`NRA`) and (354+11×`NRA`) ICs, respectively, where `NRA` is the number of return addresses on the hardware stack. If the kernel service `Schedule` is called from an ISR, its execution time is 11 ICs.

Otherwise, in user mode, its execution time is (786+12×NRA+PstTskHk+11×NRA′+PreTskHk) ICs, where NRA and NRA′ are the number of return addresses of the current running task and the restored next running task, respectively, PstTskHk and PreTskHk are the costs due to PostTaskHook and PreTaskHook, respectively. In non-user mode, its execution is shorter because no application task is running. When all application tasks have unique priorities, there is no hook involved, and there are two return addresses to be saved/restored, the latency of the kernel scheduler is 809 ICs, which is 80.9 μsec and 1.6% of a system tick.

The execution of PostTaskHook is needed after the termination of the application tasks. However, according to the OSEK standard, it is called upon each context switch during system execution. As analyzed in Section 6.2.1, the temporal overhead due to calling the routine when the functionality in the hook is not needed is 77 ICs in the worst case, which is 7.7 μsec. This implies that the kernel scheduler latency increases to 88.6 μsec. The relative overhead increase is 9.5% and the context switch occupies 1.8% of a system tick.

The costs of context switch for applications under the TCCP and the CTDBP are measured and the results are summarized in Tables 6.3 and 6.4, where $\mathtt{NS}^{\mathtt{j}}$ and $\mathtt{MS}^{\mathtt{j}}$ represent the numbers of lower priority and higher priority slow readers of task j, respectively.

	$\mathtt{CS}_{\mathtt{clk,ter}}$	$\mathtt{CS}_{\mathtt{clk,d}}$	$\mathtt{CS}'_{\mathtt{clk,d}}$
mode	user	user	kernel
to	$\tau_{\mathtt{j}}$	dispatcher	
TCCP	786+12×NRA+11×NRA′	575+12×NRA	267
CTDBP	863+12×NRA+11×NRA′	652+12×NRA	310

Table 6.3: Cost of Context Switches from Clock Interrupt (in ICs)

	$\mathtt{CS}_{\mathtt{i,j}}$	$\mathtt{CS}_{\mathtt{i,clk}}$	$\mathtt{CS}_{\mathtt{j,ter}}$	$\mathtt{CS}_{\mathtt{d,ter}}$
from	$\tau_{\mathtt{i}}$		$\tau_{\mathtt{j}}$	dispatcher
to	$\tau_{\mathtt{j}}$	clk	$\tau_{\mathtt{i}}$	
TCCP	$786+12\times\mathtt{NRA}+11\times\mathtt{NRA}'$	489	$478+11\times\mathtt{NRA}'$	
CTDBP	$863+12\times\mathtt{NRA}+11\times\mathtt{NRA}'$	489	$593+233\times\mathtt{NS}^j+11\times\mathtt{NRA}'+96\times\mathtt{MS}^j$	$521+11\times\mathtt{NRA}'$

Table 6.4: Cost of Context Switches from a Task (in ICs, assume $\pi_{\mathtt{j}} > \pi_{\mathtt{i}}$)

6.3 MILP Reformulation

The formulation of the optimization problem in Section 6.1 contains functions (ceiling, minimum, and maximum) and product terms of variables, and is not suitable for automatic processing by a linear or convex optimization solver. However, based on the performance evaluation results of kernel services and context switches with the ePICos18 from Section 6.2, it is possible to reformulate the nonlinear terms in such a way that the problem becomes of MILP (Mixed Integer Linear Programming) type. In the following, bold lettering is used to denote constants (typically for upper and lower bounds) that are used to reformulate the problem constraints to avoid nonlinear terms. This improves readability and allows to easily recognize products of variables, as opposed to products of a variable and one or more constant factors.

6.3.1 Ceiling and Minimum/Maximum Functions

As a first step of reformulation into an MILP format, consider the ceiling, minimum, and maximum functions in this subsection.

Ceiling function To reformulate the ceiling function in Equations 6.2 and 6.3, auxiliary integer variables $\alpha_{\mathtt{i,j}}$ and $\beta_{\mathtt{j,i}}$ are introduced to represent the worst-case number of preemp-

tions from a higher priority task and the number of writer task activations in the lifetime of a datum, respectively. These variables are constrained as follows:

$$\alpha_{i,j} = \left\lceil \frac{R_i}{T_j} \right\rceil, \;\; 0 \leq \alpha_{i,j} - \frac{R_i}{T_j} < 1, \;\; i \in \mathcal{T}, \;\; j \in (\mathcal{T} \cup \{d, clk\}),$$

$$\beta_{j,i} = \left\lceil \frac{l_j}{T_i} \right\rceil, \;\; 0 \leq \beta_{j,i} - \frac{l_j}{T_i} < 1, \;\; i \in \mathcal{T}, \;\; j \in \mathcal{R}.$$

Note that constraints with strict inequalities of type $f(x) < c$ can be approximated by $f(x) \leq c - c'$, where $c' << c$.

The worst-case response time in Equation 6.2 can now be expressed as:

$$\begin{aligned} R_i =& C_i + \sum_{j \in hp'(i)} \alpha_{i,j} \left(C_j + CS_{i,j} + CS_{j,ter}\right) + \\ & \sum_{q=1}^{NR} \alpha_{i,r^{(q)}} \left(\mathbf{\Gamma} x_{\mathcal{S}(q),q} + \mathbf{\Phi}(1 - x_{\mathcal{S}(q),q})\right) + \sum_{j=1}^{NW} \left(\alpha_{i,w^{(j)}} \left(\mathbf{\Psi}\lambda_j + \mathbf{\Delta}\rho_j\right)\right) + \\ & \sum_{j=1}^{NT} \alpha_{i,j} C_{k,\tau} + \alpha_{i,d} \left(C_{k,d} + CS_{clk,d} + CS_{d,ter}\right) + \alpha_{i,clk} \left(CS_{i,clk} + CS_{clk,ter}\right). \end{aligned} \tag{6.11}$$

Similarly, the ceiling can be removed from the formulation of the buffer size in Equation 6.3 for the TCCP:

$$(1 - x_{i,j}) \left\lceil \frac{l_j}{\mathbf{T_{w^{(i)}}}} \right\rceil \mathbf{L_{i,j}} = (1 - x_{i,j}) \beta_{j,w^{(i)}} \mathbf{L_{i,j}}. \tag{6.12}$$

Minimum function The definitions of λ_i (Eq. 6.6), ρ_i (Eq. 6.7), λ, and ρ (Eq. 6.8) involve the evaluation of a minimum function.

The minimum in Eq. 6.6 has the following meaning. Each λ_i is a binary variable that assumes value 1 if at least one of the $x_{i,j}$ is equal to one and zero otherwise. The following set of linear constraints provides an equivalent assignment:

$$0 \leq \lambda_i \leq 1, \; x_{i,j} \leq \lambda_i \leq NRS_i \text{ for } 1 \leq i \leq NW, \; 1 \leq j \leq NR.$$

Similarly, the determination of the values of ρ_i, λ, and ρ can be reformulated in terms of linear constraints.

Maximum function Maximum function appears in Equations 6.3 and 6.5. The following linear constraints require that the left hand side of the previous equations is always larger than any of the elements of the set that needs to be maximized.

$$\mathtt{NBF_i} \geq \left(1 - \mathtt{x_{i,j}}\right) \beta_{\mathtt{j},\mathtt{w}^{(i)}} \mathtt{L_{i,j}} \text{ for } 1 \leq \mathtt{i} \leq \mathtt{NW},\ 1 \leq \mathtt{j} \leq \mathtt{NR},$$

$$\kappa_i \geq \mathtt{delay[j]} \mathtt{x_{i,j}} \mathtt{L_{i,j}} \text{ for } 1 \leq \mathtt{i} \leq \mathtt{NW},\ 1 \leq \mathtt{j} \leq \mathtt{NR}.$$

Since both $\mathtt{NBF_i}$ (explicitly) and κ_i (implicitly through Eq. 6.4) appear in the cost function, its minimization results in the correct evaluation of the maximum.

6.3.2 Variable Products

The last obstacle for a linear formulation is quadratic terms appearing in the problem formulation: the product of an integer variable and a binary variable ($\alpha_{\mathtt{i},\mathtt{r}^{(q)}} \mathtt{x}_{\mathcal{S}(\mathtt{q}),\mathtt{q}}$, $\alpha_{\mathtt{i},\mathtt{w}^{(j)}} \lambda_\mathtt{j}$, $\alpha_{\mathtt{i},\mathtt{w}^{(j)}} \rho_\mathtt{j}$ (Eq. 6.11), $\beta_{\mathtt{j},\mathtt{w}^{(i)}} \mathtt{x_{i,j}}$ (Eq. 6.12), and $\lambda_\mathtt{i} \kappa_\mathtt{i}$ (Eq. 6.4)) and the product of binary variables ($\lambda_\mathtt{i} \mathtt{x_{i,j}}$ (Eq. 6.4)). Since the integer variables are bounded, the products can be linearized as discussed in [2][40][99]. $\alpha_{\mathtt{i,j}}$ are both lower and upper bounded:

$$\boldsymbol{\alpha^{l}_{i,j}} = \left\lceil \frac{\mathbf{R^{l}_{i}}}{\mathbf{T_j}} \right\rceil \leq \alpha_{\mathtt{i,j}} \leq \left\lceil \frac{\mathbf{R^{u}_{i}}}{\mathbf{T_j}} \right\rceil = \boldsymbol{\alpha^{u}_{i,j}}.$$

$\mathbf{R^{l}_{i}}$ and $\mathbf{R^{u}_{i}}$ are the lower and upper bound of the worst-case response time of $\tau_\mathtt{i}$, which can be obtained by considering the cases where all readers are of type fast and slow, respectively. Given these bounds, $\alpha_{\mathtt{i},\mathtt{r}^{(q)}} \mathtt{x}_{\mathcal{S}(\mathtt{q}),\mathtt{q}}$ can be reformulated as:

$$\gamma_{\mathtt{i,q}} = \alpha_{\mathtt{i},\mathtt{r}^{(q)}} \mathtt{x}_{\mathcal{S}(\mathtt{q}),\mathtt{q}},\ \boldsymbol{\alpha^{l}_{i,r^{(q)}}} \mathtt{x}_{\mathcal{S}(\mathtt{q}),\mathtt{q}} \leq \gamma_{\mathtt{i,q}} \leq \boldsymbol{\alpha^{u}_{i,r^{(q)}}} \mathtt{x}_{\mathcal{S}(\mathtt{q}),\mathtt{q}},$$

$$\alpha_{i,r^{(q)}} - \boldsymbol{\alpha^{u}_{i,r^{(q)}}} \left(1 - x_{\mathcal{S}(q),q}\right) \leq \gamma_{i,q} \leq \alpha_{i,r^{(q)}} - \boldsymbol{\alpha^{l}_{i,r^{(q)}}} \left(1 - x_{\mathcal{S}(q),q}\right).$$

Similarly, $\alpha_{i,w^{(j)}}\lambda_j$, $\alpha_{i,w^{(j)}}\rho_j$, $\beta_{j,w^{(i)}}x_{i,j}$, $\kappa_i\lambda_i$, and $\lambda_i x_{i,j}$ can be reformulated in constraints that are amenable to processing by an MILP solver.

Other remaining products are $\alpha_{i,j}C_{k,\tau}$, $\alpha_{i,j}CS_{i,j}$, $\alpha_{i,d}CS_{clk,d}$, $\alpha_{i,d}CS_{d,ter}$, $\alpha_{i,clk}CS_{i,clk}$, and $\alpha_{i,clk}CS_{clk,ter}$ (Eq. 6.11). From Equations 6.9 and 6.10, clearly each of them actually includes two quadratic terms ($\lambda\alpha_{i,j}, \rho\alpha_{i,j}$) and a cubic term ($\rho\lambda\alpha_{i,j}$), where j can be an application task, the dispatcher, or the clock interrupt handler. $\lambda\alpha_{i,j}$ is reformulated as:

$$\mu_{i,j} = \lambda\alpha_{i,j}, \quad \lambda\boldsymbol{\alpha^{l}_{i,j}} \leq \mu_{i,j} \leq \lambda\boldsymbol{\alpha^{u}_{i,j}},$$

$$\alpha_{i,j} - \boldsymbol{\alpha^{u}_{i,j}}(1-\lambda) \leq \mu_{i,j} \leq \alpha_{i,j} - \boldsymbol{\alpha^{l}_{i,j}}(1-\lambda).$$

Note that $\rho\alpha_{i,j}$ can be linearly reformulated in a similar way. The cubic term $\rho\lambda\alpha_{i,j} = \rho\mu_{i,j}$ is reformulated as:

$$\psi_{i,j} = \rho\mu_{i,j},\ 0 \leq \psi_{i,j} \leq \rho\boldsymbol{\alpha^{u}_{i,j}},\ \mu_{i,j} - (1-\rho)\boldsymbol{\alpha^{u}_{i,j}} \leq \psi_{i,j} \leq \mu_{i,j}.$$

The only remaining product term (in Eq. 6.11) is

$$\alpha_{i,j}CS_{j,ter} = \alpha_{i,j}(\lambda CS^{D}_{j,ter} + \rho(1-\lambda)\boldsymbol{CS^{T}_{j,ter}}). \tag{6.13}$$

From Table 6.4, it is known that $\boldsymbol{CS^{T}_{j,ter}}$ is constant. However,

$$CS^{D}_{j,ter} = 593 + 233 \times NS^{j} + 11 \times NRA' + 96 \times MS^{j}$$

$$= \boldsymbol{CS'^{D}_{j,ter}} + 137 \times NS^{j} + 96 \times NRS^{j},$$

where $NS^{j} = \sum_{m=1,NR}^{j=r^{(m)}} x_{\mathcal{S}(m),m} p_{w^{(\mathcal{S}(m))},r^{(m)}}$ and $MS^{j} = NRS^{j} - NS^{j}$. NRS^{j} is the number of readers of τ_j that use the CTDBP, i.e., $NRS^{j} = \sum_{m=1,NR}^{j=r^{(m)}} x_{\mathcal{S}(m),m}$. Among the product terms in

Equation 6.13, only $\lambda\alpha_{i,j}x_{\mathcal{S}(m),m} = \mu_{i,j}x_{\mathcal{S}(m),m}$ is left for reformulation, which is performed as follows:

$$\varsigma_{i,j,m} = \mu_{i,j}x_{\mathcal{S}(m),m}, \quad 0 \leq \varsigma_{i,j,m} \leq \boldsymbol{\alpha^{u}_{i,j}}x_{\mathcal{S}(m),m}$$

$$\mu_{i,j} - \boldsymbol{\alpha^{u}_{i,j}}(1 - x_{\mathcal{S}(m),m}) \leq \varsigma_{i,j,m} \leq \mu_{i,j}.$$

6.3.3 Complete MILP Formulation

In this subsection, the complete MILP reformulation is summarized. The notations used in the formulation are self-explanatory.

`minimize`

$$NB = \sum_{i=1}^{NW} (NBF_i + NBS_i)$$

`such that`

for all tasks $\forall i \in \mathcal{T}$

$\lambda\alpha^{l}_{i,d} \leq \nu_i \leq \lambda\alpha^{u}_{i,d}$ (for $\nu_i = \lambda\alpha_{i,d}$)

$\alpha_{i,d} - \alpha^{u}_{i,d}(1 - \lambda) \leq \nu_i \leq \alpha_{i,d} - \alpha^{l}_{i,d}(1 - \lambda)$

$\rho\alpha^{l}_{i,d} \leq \phi_i \leq \rho\alpha^{u}_{i,d}$ (for $\phi_i = \rho\alpha_{i,d}$)

$\alpha_{i,d} - \alpha^{u}_{i,d}(1 - \rho) \leq \phi_i \leq \alpha_{i,d} - \alpha^{l}_{i,d}(1 - \rho)$

$0 \leq \omega_i \leq \rho\alpha^{u}_{i,d}$ (for $\omega_i = \rho\lambda\alpha_{i,d} = \rho\nu_i$)

$\nu_i - \alpha^{u}_{i,d}(1 - \rho) \leq \omega_i \leq \nu_i$

$\lambda\alpha^{l}_{i,clk} \leq \xi_i \leq \lambda\alpha^{u}_{i,clk}$ (for $\xi_i = \lambda\alpha_{i,clk}$)

$\alpha_{i,clk} - \alpha^{u}_{i,clk}(1 - \lambda) \leq \xi_i \leq \alpha_{i,clk} - \alpha^{l}_{i,clk}(1 - \lambda)$

$\rho\alpha^{l}_{i,clk} \leq \chi_i \leq \rho\alpha^{u}_{i,clk}$ (for $\chi_i = \rho\alpha_{i,clk}$)

$$\alpha_{\mathtt{i,clk}} - \alpha^{\mathtt{u}}_{\mathtt{i,clk}}\,(1-\rho) \leq \chi_{\mathtt{i}} \leq \alpha_{\mathtt{i,clk}} - \alpha^{\mathtt{l}}_{\mathtt{i,clk}}\,(1-\rho)$$

$$0 \leq \varphi_{\mathtt{i}} \leq \rho\alpha^{\mathtt{u}}_{\mathtt{i,clk}} \;\; (\text{for } \varphi_{\mathtt{i}} = \rho\lambda\alpha_{\mathtt{i,clk}} = \rho\xi_{\mathtt{i}})$$

$$\xi_{\mathtt{i}} - \alpha^{\mathtt{u}}_{\mathtt{i,clk}}\,(1-\rho) \leq \varphi_{\mathtt{i}} \leq \xi_{\mathtt{i}}$$

$$\mathtt{R_i} \leq \mathtt{D_i}$$

$$\begin{aligned}\mathtt{R_i} = \mathtt{C_i} &+ \sum_{\mathtt{j}\in \mathtt{hp'(i)}} \Big(\alpha_{\mathtt{i,j}}\mathtt{C_j} + \mu_{\mathtt{i,j}}(\mathtt{CS}^{\mathtt{D}}_{\mathtt{i,j}} + \mathtt{CS'}^{\mathtt{D}}_{\mathtt{j,ter}})\Big) + \\ &\sum_{\mathtt{j}\in \mathtt{hp'(i)}} \sum_{\mathtt{m}=1,\mathtt{NR}}^{\mathtt{j}=\mathtt{r}^{(\mathtt{m})}} \varsigma_{\mathtt{i,j,m}} \Big(137\mathtt{p}_{\mathtt{w}^{(\mathcal{S}(\mathtt{m}))},\mathtt{j}} + 96\Big) + \sum_{\mathtt{j}\in \mathtt{hp'(i)}} (\sigma_{\mathtt{i,j}} - \psi_{\mathtt{i,j}})(\mathtt{CS}^{\mathtt{T}}_{\mathtt{i,j}} + \mathtt{CS}^{\mathtt{T}}_{\mathtt{j,ter}}) \\ &+ \sum_{\mathtt{q}=1}^{\mathtt{NR}} \big((\Gamma - \Phi)\gamma_{\mathtt{i,q}} + \Phi\alpha_{\mathtt{i},\mathcal{Q}}\big) + \sum_{\mathtt{j}=1}^{\mathtt{NW}} \big(\Psi\delta_{\mathtt{i,j}} + \Delta\epsilon_{\mathtt{i,j}}\big) \\ &+ \sum_{\mathtt{j}=1}^{\mathtt{NT}} \big(\mu_{\mathtt{i,j}}\mathtt{C}^{\mathtt{D}}_{\mathtt{k},\tau} + (\sigma_{\mathtt{i,j}} - \psi_{\mathtt{i,j}})\mathtt{C}^{\mathtt{T}}_{\mathtt{k},\tau}\big) + \alpha_{\mathtt{i,d}}\mathtt{C_{k,d}} + \nu_{\mathtt{i}}(\mathtt{CS}^{\mathtt{D}}_{\mathtt{clk,d}} + \mathtt{CS}^{\mathtt{D}}_{\mathtt{d,ter}}) \\ &+ (\phi_{\mathtt{i}} - \omega_{\mathtt{i}})(\mathtt{CS}^{\mathtt{T}}_{\mathtt{clk,d}} + \mathtt{CS}^{\mathtt{T}}_{\mathtt{d,ter}}) + \alpha_{\mathtt{i,clk}}\mathtt{CS_{i,clk}} + \xi_{\mathtt{i}}\mathtt{CS}^{\mathtt{D}}_{\mathtt{clk,ter}} + (\chi_{\mathtt{i}} - \varphi_{\mathtt{i}})\mathtt{CS}^{\mathtt{T}}_{\mathtt{clk,ter}}\end{aligned}$$

for all writer tasks $\forall i \in \mathcal{W}$

$$\mathtt{NBS_i} = \lambda_{\mathtt{i}} \left(\sum_{\mathtt{j}=1}^{\mathtt{NR}} \Big(\mathtt{x_{i,j}p}_{\mathtt{w}^{(\mathtt{i})},\mathtt{r}^{(\mathtt{j})}}\mathtt{L_{i,j}}\Big) + 1 + \kappa_{\mathtt{i}}\right)$$

$$\mathtt{NBS_i} = \sum_{\mathtt{j}=1}^{\mathtt{NR}} \Big(\varpi_{\mathtt{i,j}}\mathtt{p}_{\mathtt{w}^{(\mathtt{i})},\mathtt{r}^{(\mathtt{j})}}\mathtt{L_{i,j}}\Big) + \lambda_{\mathtt{i}} + \theta_{\mathtt{i}},$$

$$0 \leq \theta_{\mathtt{i}} \leq \kappa^{\mathtt{u}}_{\mathtt{i}}\lambda_{\mathtt{i}} \;\; (\text{for } \theta_{\mathtt{i}} = \lambda_{\mathtt{i}}\kappa_{\mathtt{i}})$$

$$\kappa_{\mathtt{i}} - \kappa^{\mathtt{u}}_{\mathtt{i}}(1-\lambda_{\mathtt{i}}) \leq \theta_{\mathtt{i}} \leq \kappa_{\mathtt{i}}$$

$$0 \leq \lambda_{\mathtt{i}} \leq 1, \qquad 0 \leq \rho_{\mathtt{i}} \leq 1$$

$$\lambda_{\mathtt{i}} \leq \lambda \leq \sum_{\mathtt{j}=1}^{\mathtt{NW}} \lambda_{\mathtt{j}}, \qquad \rho_{\mathtt{i}} \leq \rho \leq \sum_{\mathtt{j}=1}^{\mathtt{NW}} \rho_{\mathtt{j}}$$

for all reader tasks $\forall \mathtt{i} \in \mathcal{R}$

$$\mathtt{l_i} = \mathtt{delay[i]T_{w^{(\mathcal{S}(i))}}} + \mathtt{O_{\mathcal{S}(i),i}} + \mathtt{R_{r^{(i)}}}$$

for all tasks $\forall \mathtt{i}, \mathtt{j} \in \mathcal{T}$

$$0 \leq \alpha_{\mathtt{i,j}} - \frac{\mathtt{R_i}}{\mathtt{T_j}} < 1 \ \text{(for } \alpha_{\mathtt{i,j}} = \left\lceil \frac{\mathtt{R_i}}{\mathtt{T_j}} \right\rceil)$$

$$\lambda \alpha^{\mathtt{l}}_{\mathtt{i,j}} \leq \mu_{\mathtt{i,j}} \leq \lambda \alpha^{\mathtt{u}}_{\mathtt{i,j}} \ \text{(for } \mu_{\mathtt{i,j}} = \lambda \alpha_{\mathtt{i,j}})$$

$$\alpha_{\mathtt{i,j}} - \alpha^{\mathtt{u}}_{\mathtt{i,j}} (1 - \lambda) \leq \mu_{\mathtt{i,j}} \leq \alpha_{\mathtt{i,j}} - \alpha^{\mathtt{l}}_{\mathtt{i,j}} (1 - \lambda)$$

$$\rho \alpha^{\mathtt{l}}_{\mathtt{i,j}} \leq \sigma_{\mathtt{i,j}} \leq \rho \alpha^{\mathtt{u}}_{\mathtt{i,j}} \ \text{(for } \sigma_{\mathtt{i,j}} = \rho \alpha_{\mathtt{i,j}})$$

$$\alpha_{\mathtt{i,j}} - \alpha^{\mathtt{u}}_{\mathtt{i,j}} (1 - \rho) \leq \sigma_{\mathtt{i,j}} \leq \alpha_{\mathtt{i,j}} - \alpha^{\mathtt{l}}_{\mathtt{i,j}} (1 - \rho)$$

$$0 \leq \psi_{\mathtt{i,j}} \leq \rho \alpha^{\mathtt{u}}_{\mathtt{i,j}} \ \text{(for } \psi_{\mathtt{i,j}} = \rho \lambda \alpha_{\mathtt{i,j}} = \rho \mu_{\mathtt{i,j}})$$

$$\mu_{\mathtt{i,j}} - \alpha^{\mathtt{u}}_{\mathtt{i,j}} (1 - \rho) \leq \psi_{\mathtt{i,j}} \leq \mu_{\mathtt{i,j}}$$

for all tasks $\forall \mathtt{i}, \mathtt{j} \in \mathcal{T}$ and *for all reader tasks* $\forall \mathtt{m} \in \mathcal{R}$

$$0 \leq \varsigma_{\mathtt{i,j,m}} \leq \alpha^{\mathtt{u}}_{\mathtt{i,j}} \mathtt{x}_{\mathcal{S}(\mathtt{m}),\mathtt{m}} \ \text{(for } \varsigma_{\mathtt{i,j,m}} = \lambda \alpha_{\mathtt{i,j}} \mathtt{x}_{\mathcal{S}(\mathtt{m}),\mathtt{m}} = \mu_{\mathtt{i,j}} \mathtt{x}_{\mathcal{S}(\mathtt{m}),\mathtt{m}})$$

$$\mu_{\mathtt{i,j}} - \alpha^{\mathtt{u}}_{\mathtt{i,j}} \left(1 - \mathtt{x}_{\mathcal{S}(\mathtt{m}),\mathtt{m}}\right) \leq \varsigma_{\mathtt{i,j,m}} \leq \mu_{\mathtt{i,j}}$$

for all tasks $\forall \mathtt{i} \in \mathcal{T}$ and *for all reader tasks* $\forall \mathtt{q} \in \mathcal{R}$

$$\alpha^{\mathtt{l}}_{\mathtt{i},\mathtt{r}^{(\mathtt{q})}} \mathtt{x}_{\mathcal{S}(\mathtt{q}),\mathtt{q}} \leq \gamma_{\mathtt{i,q}} \leq \alpha^{\mathtt{u}}_{\mathtt{i},\mathtt{r}^{(\mathtt{q})}} \mathtt{x}_{\mathcal{S}(\mathtt{q}),\mathtt{q}} \ \text{(for } \gamma_{\mathtt{i,q}} = \alpha_{\mathtt{i},\mathtt{r}^{(\mathtt{q})}} \mathtt{x}_{\mathcal{S}(\mathtt{q}),\mathtt{q}})$$

$$\alpha_{\mathtt{i},\mathtt{r}^{(\mathtt{q})}} - \alpha^{\mathtt{u}}_{\mathtt{i},\mathtt{r}^{(\mathtt{q})}} \left(1 - \mathtt{x}_{\mathcal{S}(\mathtt{q}),\mathtt{q}}\right) \leq \gamma_{\mathtt{i,q}} \leq \alpha_{\mathtt{i},\mathtt{r}^{(\mathtt{q})}} - \alpha^{\mathtt{l}}_{\mathtt{i},\mathtt{r}^{(\mathtt{q})}} \left(1 - \mathtt{x}_{\mathcal{S}(\mathtt{q}),\mathtt{q}}\right)$$

for all tasks $\forall \mathtt{i} \in \mathcal{T}$ and *for all writer tasks* $\forall \mathtt{j} \in \mathcal{W}$

$$\alpha^{\mathtt{l}}_{\mathtt{i},\mathtt{w}^{\mathtt{j}}}\lambda_{\mathtt{j}} \leq \delta_{\mathtt{i},\mathtt{j}} \leq \alpha^{\mathtt{u}}_{\mathtt{i},\mathtt{w}^{\mathtt{j}}}\lambda_{\mathtt{j}} \;\; (\text{for } \delta_{\mathtt{i},\mathtt{j}} = \alpha_{\mathtt{i},\mathtt{w}^{\mathtt{j}}}\lambda_{\mathtt{j}})$$

$$\alpha_{\mathtt{i},\mathtt{w}^{\mathtt{j}}} - \alpha^{\mathtt{u}}_{\mathtt{i},\mathtt{w}^{\mathtt{j}}}\left(\mathtt{1}-\lambda_{\mathtt{j}}\right) \leq \delta_{\mathtt{i},\mathtt{j}} \leq \alpha_{\mathtt{i},\mathtt{w}^{\mathtt{j}}} - \alpha^{\mathtt{l}}_{\mathtt{i},\mathtt{w}^{\mathtt{j}}}\left(\mathtt{1}-\lambda_{\mathtt{j}}\right)$$

$$\alpha^{\mathtt{l}}_{\mathtt{i},\mathtt{w}^{\mathtt{j}}}\rho_{\mathtt{j}} \leq \epsilon_{\mathtt{i},\mathtt{j}} \leq \alpha^{\mathtt{u}}_{\mathtt{i},\mathtt{w}^{\mathtt{j}}}\rho_{\mathtt{j}} \;\; (\text{for } \epsilon_{\mathtt{i},\mathtt{j}} = \alpha_{\mathtt{i},\mathtt{w}^{\mathtt{j}}}\rho_{\mathtt{j}})$$

$$\alpha_{\mathtt{i},\mathtt{w}^{\mathtt{j}}} - \alpha^{\mathtt{u}}_{\mathtt{i},\mathtt{w}^{\mathtt{j}}}\left(\mathtt{1}-\rho_{\mathtt{j}}\right) \leq \epsilon_{\mathtt{i},\mathtt{j}} \leq \alpha_{\mathtt{i},\mathtt{w}^{\mathtt{j}}} - \alpha^{\mathtt{l}}_{\mathtt{i},\mathtt{w}^{\mathtt{j}}}\left(\mathtt{1}-\rho_{\mathtt{j}}\right)$$

for all writer tasks $\forall \mathtt{i} \in \mathcal{W}$ and *for all reader tasks* $\forall \mathtt{j} \in \mathcal{R}$

$$\mathtt{NBF}_{\mathtt{i}} \geq (\beta_{\mathtt{j},\mathtt{w}^{(\mathtt{i})}} - \eta_{\mathtt{i},\mathtt{j}})\mathtt{L}_{\mathtt{i},\mathtt{j}}$$

$$\mathtt{0} \leq \beta_{\mathtt{j},\mathtt{w}^{(\mathtt{i})}} - \frac{\mathtt{l}_{\mathtt{j}}}{\mathtt{T}_{\mathtt{w}^{(\mathtt{i})}}} < \mathtt{1} \;\; \left(\text{for } \beta_{\mathtt{j},\mathtt{w}^{(\mathtt{i})}} = \left\lceil \frac{\mathtt{l}_{\mathtt{j}}}{\mathtt{T}_{\mathtt{w}^{(\mathtt{i})}}} \right\rceil\right)$$

$$\beta^{\mathtt{l}}_{\mathtt{j},\mathtt{w}^{(\mathtt{i})}}\mathtt{x}_{\mathtt{i},\mathtt{j}} \leq \eta_{\mathtt{i},\mathtt{j}} \leq \beta^{\mathtt{u}}_{\mathtt{j},\mathtt{w}^{(\mathtt{i})}}\mathtt{x}_{\mathtt{i},\mathtt{j}} \;\; (\text{for } \eta_{\mathtt{i},\mathtt{j}} = \beta_{\mathtt{j},\mathtt{w}^{(\mathtt{i})}}\mathtt{x}_{\mathtt{i},\mathtt{j}})$$

$$\beta_{\mathtt{j},\mathtt{w}^{(\mathtt{i})}} - \beta^{\mathtt{u}}_{\mathtt{j},\mathtt{w}^{(\mathtt{i})}}(\mathtt{1}-\mathtt{x}_{\mathtt{i},\mathtt{j}}) \leq \eta_{\mathtt{i},\mathtt{j}} \leq \beta_{\mathtt{j},\mathtt{w}^{(\mathtt{i})}} - \beta^{\mathtt{l}}_{\mathtt{j},\mathtt{w}^{(\mathtt{i})}}(\mathtt{1}-\mathtt{x}_{\mathtt{i},\mathtt{j}})$$

$$\kappa_{\mathtt{i}} \geq \mathtt{delay}[\mathtt{j}]\mathtt{x}_{\mathtt{i},\mathtt{j}}\mathtt{L}_{\mathtt{i},\mathtt{j}}$$

$$\mathtt{0} \leq \varpi_{\mathtt{i},\mathtt{j}} \leq \mathtt{x}_{\mathtt{i},\mathtt{j}} \;\; (\text{for } \varpi_{\mathtt{i},\mathtt{j}} = \lambda_{\mathtt{i}}\mathtt{x}_{\mathtt{i},\mathtt{j}})$$

$$\lambda_{\mathtt{i}} + \mathtt{x}_{\mathtt{i},\mathtt{j}} - \mathtt{1} \leq \varpi_{\mathtt{i},\mathtt{j}} \leq \lambda_{\mathtt{i}}$$

$$\mathtt{x}_{\mathtt{i},\mathtt{j}} \leq \lambda_{\mathtt{i}} \leq \sum_{\mathtt{m}=\mathtt{1}}^{\mathtt{NR}} \mathtt{x}_{\mathtt{i},\mathtt{m}}\mathtt{L}_{\mathtt{i},\mathtt{m}}$$

$$\mathtt{1} - \mathtt{x}_{\mathtt{i},\mathtt{j}} \leq \rho_{\mathtt{i}} \leq \sum_{\mathtt{m}=\mathtt{1}}^{\mathtt{NR}} (\mathtt{1}-\mathtt{x}_{\mathtt{i},\mathtt{m}})\mathtt{L}_{\mathtt{i},\mathtt{m}}$$

$$\mathtt{0} \leq \lambda \leq \mathtt{1} \qquad \textit{and} \qquad \mathtt{0} \leq \rho \leq \mathtt{1}$$

Because the above reformulation does not introduce relaxation or estimation, there exists a one-to-one correspondence of the optimal solutions between the MILP problem and the original optimization problem.

6.4 Experiments

In this section, two case studies based on the formulated MILP are preformed. For the first case study, hypothetical system configurations are randomly generated by TGFF while the systems in the second case study are derived from an automotive industrial application. The formulated MILP problems are solved by using the stand-alone LP/MIP glpsol [38] solver or the ILOP CPLEX software package [53].

6.4.1 Case Study I

In this subsection, the results of the application of the discussed optimization method are presented for randomly generated task graphs.

Task Graph Generation The test cases are generated by using the TGFF [33] tool. Altogether, 809 system configurations are generated with an average number of 12 tasks per system (a minimum of 8 and a maximum of 16). Each task implements a maximum of 4 writer and 8 reader blocks and each communication link can have up to 2-unit delays. Other task set attributes are randomly generated. The execution times of the tasks are uniformly distributed in $[3 \times 10^4, 9 \times 10^4]$ ICs (average of 6×10^4 ICs). Task periods are uniformly distributed in $[5.5 \times 10^5, 14.5 \times 10^5]$ ICs (average of 10^6 ICs). Task priorities are statically assigned based on the rate monotonic policy. It is assumed that task deadlines are equal to their respective periods. The system overheads and the execution times of the communication protocols have been defined according to the ePICos18-based evaluation in Section 6.2.

Results and Discussion Among the randomly generated system configurations, 158 are unschedulable under either the CTDBP or the TCCP. Among the remaining 651 schedulable systems, 601 are schedulable under both the CTDBP and the TCCP and 50 are only schedulable under the TCCP.

Among the 601 schedulable systems, 105 have a smaller buffer size under the CTDBP than under the TCCP. Figure 6.1 illustrates the relative buffer size improvement at the end of the optimization process with respect to implementations consisting of the CTDBP or the TCCP alone. The horizontal axis (X) represents the percentage of improvement obtained after optimization and the corresponding vertical axis (Y) value denotes the fraction

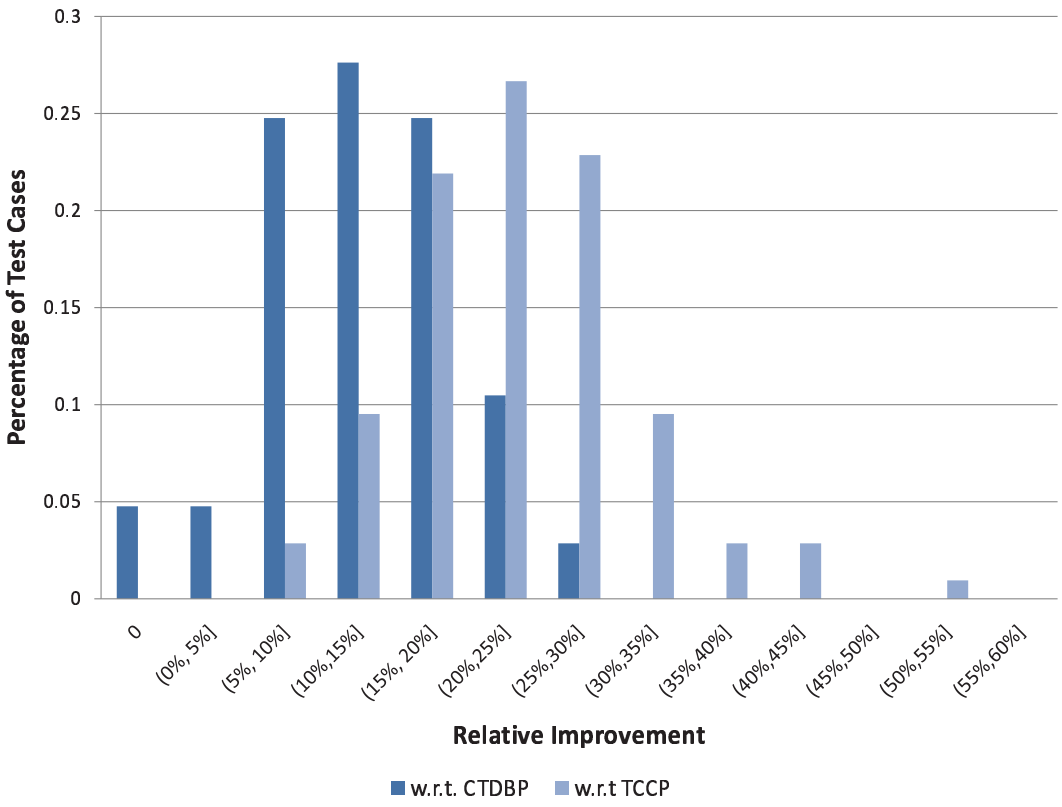

Figure 6.1: Systems with a Smaller Buffer Size under CTDBP

of systems for which the improvement was obtained. For example, 0 on the X axis means no improvement and the Y value of 4.8% means that for 4.8% of the systems the CTDBP is the optimal choice. The best improvement with respect to the CTDBP is between 25% and 30%. Improvements with respect to the TCCP are higher, up to more than 50%, albeit in only 1% of the cases. An average improvement of 14% and 24.2% on the buffer size are achieved with respect to the CTDBP-based and the TCCP-based buffer sizes, respectively.

On the other hand, 463 systems have a larger buffer size under the CTDBP than under the TCCP and Figure 6.2 shows their relative improvements. For 66.1% of these system configurations, the buffer sizes under the TCCP are optimal. An average improvement

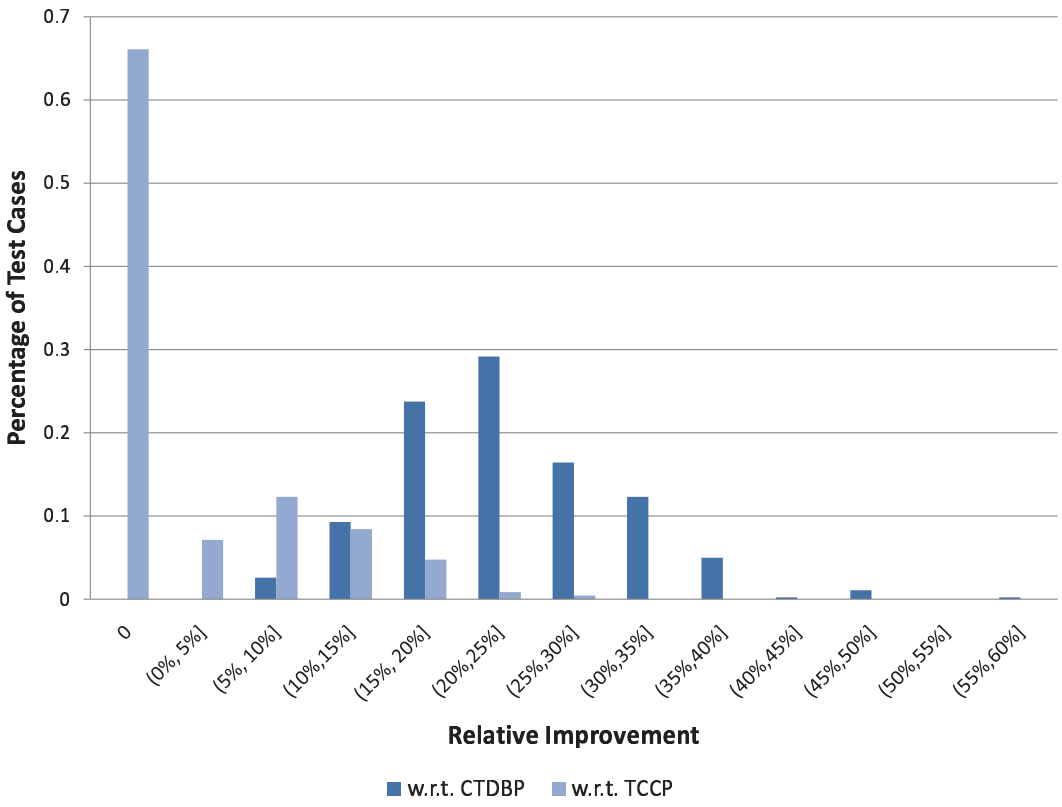

Figure 6.2: Systems with a Smaller Buffer Size under TCCP

of 23.2% and 4.7% on buffer sizes are achieved with respect to the CTDBP-based and the TCCP-based buffer sizes, respectively.

Other 33 systems have equal buffer sizes under the CTDBP and the TCCP. Figure 6.3 shows the optimization results. The buffer size is improved upon optimization for 93.9% of them. Experiments also demonstrate that an average relative buffer saving of 16.4% is achieved. Additionally, Figure 6.3 shows the optimization results for those 50 systems that are schedulable only under the TCCP. Experiments show that for 74% of them the buffer sizes under the TCCP are optimal. On average, an improvement of 6.3% on buffer size can be achieved.

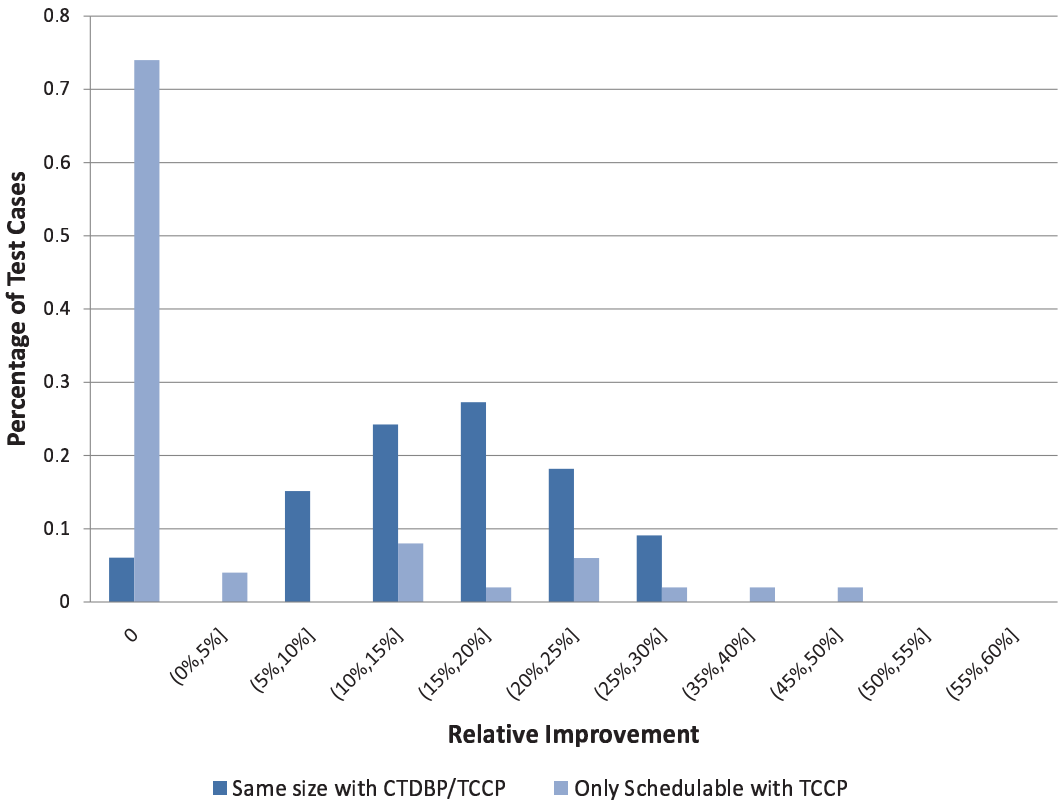

Figure 6.3: Systems with an Equal Buffer Size or Those only TCCP-Schedulable

A comparison of the results in Figures 6.1, 6.2, and 6.3 shows that optimization results in limited improvements when only the TCCP gives a feasible solution. In such cases, even if there exist slow readers (leading to long lifetimes), it is impossible to use the CTDBP, which is inherently good for slow readers since the TCCP has better time performance than the CTDBP and the system is schedulable only under the TCCP.

6.4.2 Case Study II

Consider the automotive example shown in Table 4.3 in Section 4.3.3. The number of buffers required by the different methods when the implementation overheads for the communication procedures are included in the analysis are shown in Table 6.5.

The first two rows in Table 6.5 summarize the test cases and their utilizations. The utilization is increased beyond that of Case 13. Of those 150 system configurations that are tried beyond 92.4% (some with extremely small increases of the computation times), none is schedulable under the policy with the lower overheads (i.e., the TCCP). It indicates that the system is actually very close to its utilization bounds because of overheads as well as the fixed priority scheduling policy.

1	Case	1	2	3	4	5	6	7	8	9	10	11	12	13
2	Utilization %	62.7	67.7	75.7	80.6	85.6	86.2	88.2	89.6	89.7	90.7	91.2	92.2	92.4
3	CTDBP	162	162	162	162	162	162	162	162	Non	Non	Non	Non	Non
4	TCCP	1089	1260	1410	1516	1750	1775	1799	1799	1799	1828	1828	1828	1828
5	MILP optimal	123	123	124	124	124	124	124	124	126	126	384	1828	1828
6	CTDBP	162	162	162	162	162	162	162	162	162	162	162	162	162
7	TCCP	1027	1182	1324	1410	1625	1648	1679	1679	1681	1712	1712	1712	1712
8	MILP optimal	123	123	124	124	124	124	124	124	126	126	126	126	126

Table 6.5: Experimental Results of Case Study II

Rows from 3 to 5 show the results with all scheduling overheads. Rows 3 and 4 give the bounds based on the CTDBP and the TCCP alone, respectively. Row 3 shows that the higher utilization configurations (starting from Case 9, at 89.7% utilization) are unschedulable under the CTDBP, the policy with the larger overheads, or, in general under any policy that does not leverage the tradeoffs between the two methods like the proposed optimization approach in this dissertation. For those that are schedulable, the obtained buffer bound is 162.

All 13 cases are schedulable under the TCCP, albeit with large buffer requirements, from 1089 to 1828. The results of the MILP optimization are shown in Row 5. In most cases the memory required is very close to the bound derived without considering the implementation overheads in Table 4.4 in Section 4.3.3. With the increase in utilization, the requirements also increase. Eventually, very sharply, and very close to the possible system schedulability bound, the optimal bound becomes identical to the bound obtained by using the TCCP alone. This shows how the method can actually leverage the tradeoffs between time and memory and cover the implementation space obtainable with each of the two methods alone.

Rows from 6 to 8 gives the buffer sizing results without considering the scheduling overhead due to context switches. Row 6 shows all test cases are schedulable under the CTDBP. This is because the consumed processor time due to context switches may be used to schedule functionalities of systems with bigger utilizations. In addition, the buffer bounds under the TCCP shown in Row 7 are smaller than the corresponding ones in Row 4, which further justifies that context switch overheads make data lifetime longer and thus

lead to bigger buffer sizes. Row 8 shows the optimal buffer sizes without considering context switch overheads. Due to the removal of the context switch overhead, the optimal bounds for Cases 11, 12, and 13 are much smaller than their corresponding values in Row 5. This is because more readers with long data lifetimes can be categorized as slow without violating the timing constraints.

Part III

Summary

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This dissertation explored the opportunity for the synthesis of the real-time operating system layer in the implementation of embedded software models according to the platform-based design methodology. As the stack view of platform-based design of real-time operating systems shows, the methodology consists of the task/resource model platform, the virtual RTOS platform, and the standard RTOS APIs. The dissertation focused on the synthesis problem of inter-task communication protocols. The problem was studied from a well defined set of functional models (synchronous reactive with zero-execution time semantics) and the scope was restricted to a simple task/resource model platform defined on a single processor execution architecture. With respect to the issues arising in the implementation of semantics-preserving inter-task communication, a survey on the state-of-the-art protocols was conducted. Two buffer sizing mechanisms based on sequentially in-order and out-of-order writes were investigated. The conditions under which the platform and the

protocols provide a semantics-preserving implementation were defined.

Next, the dissertation generalized synchronous reactive communication to be capable of handling a general model, allowing for multiple active task instances at the same time and communication links with arbitrary delays. The dissertation demonstrated how it is possible to leverage task timing information to obtain a tight buffer bound in a synchronous reactive semantics-preserving implementation of communication channels between a writer and multiple reader tasks (sporadic or periodic) executing at different rates with unknown activation phases in the generalized context. The bound subsumes and in general improves on existing bounds.

Because synchronous reactive communication protocols define buffer indices for writers and readers at task activation time, generally they require a kernel-level implementation. Efficient and portable OSEK implementations were presented for the Constant Time Dynamic Buffering Protocol (CTDBP) and the Temporal Concurrency Control Protocol (TCCP) by providing detailed data structures and imperative code for the search procedure used to find a safe slot for a writer. To meet the minimum requirements of the BCC1 defined in OSEK for portability, only one alarm is used to periodically activate a task dispatcher that in turn activates all other application tasks at their proper activation time. Two different implementations for the task dispatcher were presented. One of them is based on a static dispatch table pre-built at compile time while the other one uses a scheduler to dynamically decide whether a task needs to be activated. There are two options to obtain the atomicity of the termination code for lower priority readers to flag their completion in case of the CTDBP. The first one uses the hook mechanism (`PostTaskHook`) while the

second one lets the task dispatcher perform the atomic termination code for lower priority readers. Comparison of the implementations showed that the TCCP implementation uses fewer auxiliary data structures and the CTDBP has a higher implementation complexity due to the management of the use free list to support a constant time search algorithm. Memory requirements were compared quantitatively for different implementations. Furthermore, temporal characteristics of different implementations were measured under the PIC18F452 microcontroller and the OSEK-compliant ePICos18. Without the PostTaskHook, the overhead of context switch is about 80.9μsec. Turning on the PostTaskHook routine increases the context switch cost by 77 instruction cycles, which is 7.7μsec if the processor frequency is 40MHz. This implies a 9.5% relative overhead increase. A performance analysis showed that the TCCP is faster for assigning writing/reading indices than the CTDBP.

Automatic code generation was supported for two synchronous reactive communication protocols: the double buffer protocol and the constant time dynamic buffering protocol. The generated code, with the ePICos18, was validated by emulation on the PIC18F452 microcontroller through the MPLAB IDE simulator. Code validation results confirmed that by splitting the protocol into parts executing at task activation time and task execution time, respectively, the generated implementation code can always guarantee both time and value determinism, regardless of whether the periods of the writer and readers are harmonic (required by the Rate Transition Block in Simulink) or not. The emulation results were also compared to the Real-Time Workshop simulation results and consistent behaviors were obtained.

Memory is scant and expensive for embedded real-time systems and an imple-

mentation with a minimum buffer size is often desirable. The TCCP is good to implement communication between a writer and its fast readers while the CTDBP is good for communication implementation between a writer and its slow readers. But the CTDBP may require longer access times and possibly lead to the violation of deadline constraints in real-time applications. This dissertation demonstrated the feasibility of an MILP-based optimization approach that provides the minimum memory implementation of a set of communication channels within the deadline constraints of the tasks. The optimization process selectively chooses either the TCCP or the CTDBP under timing constraints for each pair of writer and reader. It was demonstrated that the optimal buffer size may be achieved under a hybrid communication protocol and the optimal partition of the readers is given by the values of the decision variables.

7.2 Future Work

The efficiency of generated code of the applications using the double buffer mechanism where the sampling rates of a writer and its reader are harmonic can be improved by avoiding storing the data into the shared buffers. For these applications, the shared data can be transferred directly from the writer to its reader. In addition, automatic code generation using the proposed model blocks needs to be applied to bigger synchronous reactive model specifications.

The MILP-based optimization in this dissertation assumes that the static priorities of tasks are known. It is better to let the optimization process assign priorities to optimize buffer consumption as well as to achieve better schedulability. In this dissertation, the

implementation of the dispatcher and a method of obtaining terminating atomicity for lower priority readers are considered in the MILP formulation. It is interesting to investigate other options.

This dissertation focused on applications in the hard real-time domain. It is of interest to investigate how the techniques presented here may be applied to the application domain of the soft real-time type. It is also interesting and practically useful to provide a similar thorough study for implementation of synchronous reactive models on execution platforms that are multiprocessor or distributed architectures.

Bibliography

[1] *GME 5 User's Manual*. Institute for Software Integrated Systems, Vanderbilt University.

[2] Warren P. Adams and Richard J. Forrester. A Simple Recipe for Concise Mixed 0-1 Linearizations. *Operations Research Letters*, 33:55–61, 2005.

[3] Arvind and Vinod Kathail. A Multiple Processor Data Flow Machine That Supports Generalized Procedures. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 291–302, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[4] Modelica Association. Modelica: Modeling of Complex Physical Systems. Available at http://www.modelica.org/.

[5] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[6] N. Audsley, A. Burns, and A. Wellings. Deadline Monotonic Scheduling Theory and Application. *Control Engineering Practice*, 1:71–78, 1993.

[7] T.P. Baker. Stack-Based Scheduling of Real-Time Processes. *Real-Time Systems*, 3, 1991.

[8] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-Software Co-design Of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[9] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *Computer*, 36(4):45–52, 2003.

[10] M. Baleani, A. Ferrari, L. Mangeruca, and A. Sangiovanni Vincentelli. Efficient Embedded Software Design with Synchronous Models. In *Proceedings of the 5th ACM International Conference on Embedded Software*, 2005.

[11] Albert Benveniste and Gerard Berry. The Synchronous Approach to Reactive and Real-Time Systems. In *Proceedings of the IEEE*, volume 79, pages 1270–1282, September 1991.

[12] Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé Marchand, Jean-Pierre Talpin, and Stavros Tripakis. A Protocol for Loosely Time-Triggered Architectures. In *Proceedings of the Second International Conference on Embedded Software*, pages 252–265, London, UK, 2002. Springer-Verlag.

[13] Gérard Berry. The Foundations of Esterel. In Gordon D. Plotkin, Colin Stirling, and

Mads Tofte, editors, *Proof, Language, and Interaction*, pages 425–454, Cambridge, MA, USA, 2000. The MIT Press.

[14] Gérard Berry and Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[15] Joseph Tobin Buck. *Scheduling Dynamic Dataflow Graphs With Bounded Memory Using The Token Flow Model*. PhD thesis, 1993.

[16] Joseph Tobin Buck. Static Scheduling and Code Generation from Dynamic Dataflow Graphs with Integer-Valued Control Streams. In *Proceedings of 28th Asilomar Conference on Signals, Systems, and Computers*, 1994.

[17] A. Burns and A. J. Wellings. Engineering a Hard Real-Time System: From Theory to Practice. *Software - Practice and Experience*, 25(7):705–726, July 1995.

[18] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Verlag, second edition, 2005. ISBN: 0-387-23137-4.

[19] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A Declarative Language for Real-Time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM.

[20] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and

Peter Niebert. From Simulink to SCADE/Lustre to TTA: A Layered Approach for Distributed Embedded Applications. *SIGPLAN Notices*, 38(7):153–162, 2003.

[21] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew J. McNelly, and Lee Todd. *Surviving the SOC Revolution: A Guide to Platform-Based Design.* Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[22] Frédérique Chaussumier, Frederic Desprez, and Loïc Prylli. Asynchronous Communications in MPL - The BIP/Myrinet Approach. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 485–492, London, UK, 1999. Springer-Verlag.

[23] J. Chen and A. Burns. A Fully Asynchronous Reader/Write Mechanism for Multiprocessor Real-Time Systems. Technical Report YCS 288, University of York, May 1997.

[24] J. Chen and A. Burns. A Three-Slot Asynchronous Reader/Writer Mechanism for Multiprocessor Real-Time Systems. Technical Report YCS 286, University of York, January 1997.

[25] J. Chen and A. Burns. Loop-Free Asynchronous Data Sharing in Multiprocessor Real-Time Systems Based on Timing Properties. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, pages 236–246, Washington, DC, USA, 1999. IEEE Computer Society.

[26] M. Chen and K. Lin. Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems. *Journal of Real-Time Systems*, 2, 1990.

[27] National Instruments Corporation. Labview. Available at http://www.ni.com/labview.

[28] J. Cortadella, A. Kondratyev, L. Lavagno, and Y. Watanabe. Quasi-Static Scheduling for Concurrent Architectures. *Proceedings of the Third International Conference on Application of Concurrency to System Design*, pages 29–40, June 2003.

[29] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Claudio Passerone, and Yosinori Watanabe. Quasi-Static Scheduling of Independent Tasks for Reactive Systems. In *Proceedings of the International Conference of Appliaction and Theory of Petri Nets*, 2002.

[30] J. B. Dennis. First Version of a Data Flow Procedure Language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[31] M. L. Dertouzos. Control Robotics: The Procedural Control of Physical Processes. *Information Processing*, 1974.

[32] J. L. Diaz-Herrera. A Survey of System-Level Design Notations for Embedded Systems. Department of Computer Science, Southern Polytechnic State University, Marietta, GA 30060.

[33] R. Dick, D. Rhodes, and W. Wolf. TGFF: Task Graphs for Free. In *Proceeding of the International Workshop Hardware/Software Codesign*, pages 97–101, 1998.

[34] Franois Xavier Dormoy. SCADE 6 – A Model Based Solution for Safety Criti-

cal Software Development. Available at http://www.esterel-technologies.com/technology/WhitePapers/, 2008.

[35] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming Heterogeneity - The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[36] A. Ferrari and A. Sangiovanni-Vincentelli. System Design: Traditional Concepts and New Paradigms. In *Proceedings of the 1999 IEEE International Conference on Computer Design*, page 2, Washington, DC, USA, 1999. IEEE Computer Society.

[37] Chamberlain Fong. Discrete-Time Dataflow Models for Visual Simulation in Ptolemy II. Technical Report UCB/ERL M01/9, Electrical Engineering and Computer Sciences Department, University of California, Berkeley, 2001.

[38] Free Software Foundation, Inc. *GNU Linear Programming Kit Reference Manual*. Available at http://www.gnu.org/software/glpk/.

[39] Daniel D. Gajski. System-Level Design Methodology. Center for Embedded Computer Systems, University of California, Irvine, available at http://www.cecs.uci.edu/~gajski/, 2004.

[40] F. Glover and R. E. Woolsey. Further Reduction of Zero-One Polynomial Programming Problems to Zero-One Linear Programming Problems. *Operations Reseach*, 21:156–161, 1974.

[41] GNU. The General Public License. Available at http://www.gnu.org/copyleft/gpl.html.

[42] Gert Goossens, Johan Van Praet, Dirk Lanneer, Werner Geurts, Augusli Kifli, Clifford Liem, Pierre, and G. Paulin. Embedded Software in Real-Time Signal Processing Systems: Design Technologies. In *Proceedings of the IEEE*, volume 85, pages 436–454, 1997.

[43] Mentor Graphics. Electronic System Level Design. Available at http://www.mentor.com/products/esl/.

[44] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming Real-Time Applications with SIGNAL. In *Proceedings of the IEEE*, volume 79, pages 1321–1336, September 1991.

[45] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The Synchronous Dataflow Programming Language Lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[46] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[47] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A Time-Triggered Language for Embedded Programming. Technical report, Berkeley, CA, USA, 2001.

[48] Thomas A. Henzinger and Christoph M. Kirsch. The Embedded Machine: Predictable, Portable Real-Time Code. *ACM Transaction on Programming Languages and Systems*, 29(6):33, 2007.

[49] Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic. Schedule-Carrying Code. In *Proceedings of the International Conference on Embedded Software*, 2003.

[50] Charles Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[51] Hai Huang, Padmanabhan Pillai, and Kang G. Shin. Improving Wait-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 303–316, Berkeley, CA, USA, 2002. USENIX Association.

[52] Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu, Stephen Neuendorffer, Yuhong Xiong, Yang Zhao, and Haiyang Zheng. Overview of the Ptolemy Project. Technical Report UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003.

[53] ILOG. ILOG CPLEX. Available at http://www.ilog.com/products/cplex/.

[54] CoWare Incorporation. Coware Signal Processing Designer – Implementing Algorithms for Platform-Driven ESL Design. Available at http://www.coware.com.

[55] OPNET Technologies Incorporation. OPNET Modeler. Available at http://www.opnet.com.

[56] Mansour Jaragh and Kassem Saleh. Synthesis of Communications Protocol Converters Using the Timed Petri Net Model. *Journal of Systems and Software*, 47(1):53–69, 1999.

[57] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *Information Processing*, pages 471–475, 1974.

[58] Gilles Kahn and David B. MacQueen. Coroutines and Networks of Parallel Processes. In *Information Processing*, pages 993–998, 1977.

[59] Richard M. Karp and Rayamond E. Miller. Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[60] Kurt Keutzer, Sharad Malik, Richard Newton, Jan Rabaey, and Alberto Sangiovanni-Vincentelli. System Level Design: Orthogonolization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[61] H. Kopetz and J. Reisinger. The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronization Problem. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, December 1993.

[62] Hermann Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[63] Leslie Lamport. Concurrent Reading and Writing. *Communications of the ACM*, 20(11):806–811, 1977.

[64] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal–A Data Flow-Oriented Language for Signal Processing. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34(2):362–374, Apr 1986.

[65] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, May 2001.

[66] Edward A. Lee. Modeling Concurrent Real-Time Processes Using Discrete Events. *Annals of Software Engineering*, 7(1-4):25–45, 1999.

[67] Edward A. Lee. What's Ahead for Embedded Software? *Computer*, 33(9):18–26, 2000.

[68] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. In *Proceeding of the IEEE*, volume 75, pages 1235–1245, 1987.

[69] Edward A. Lee and Thomas M. Parks. Dataflow Process Networks. In *Proceedings of the IEEE*, volume 83, 1995.

[70] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A Unified Framework for Comparing Models of Computation. *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[71] John P. Lehoczky, Lui Sha, and Ye Ding. The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the* 10^{th} *IEEE Real-Time Systems Symposium*, pages 166–171, Santa Monica, CA USA, December 1989.

[72] J.Y.-T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2, 1982.

[73] Chung Laung Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[74] Jie Liu. Continuous Time and Mixed-Signal Simulation in Ptolemy II. Technical Report UCB/ERL M98/74, EECS Department, University of California, Berkeley, 1998.

[75] C. Douglass Locke. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. *Real-Time Systems*, 4(1):37–53, 1992.

[76] The MathWorks Inc. *The MathWorks Real-Time Workshop Embedded Coder: User's Guide*. Available at http://www.mathworks.com.

[77] The MathWorks Inc. *The MathWorks Real-Time Workshop for Use with Simulink User's Guide: Modeling, Simulation, Implementation*. Available at http://www.mathworks.com.

[78] The MathWorks Inc. *The MathWorks Real-Time Workshop: Target Language Compiler*. Available at http://www.mathworks.com.

[79] The MathWorks Inc. *The MathWorks Real-Time Workshop: User's Guide*. Available at http://www.mathworks.com.

[80] The MathWorks Inc. *The MathWorks Simulink: Writing S-Functions*. Available at http://www.mathworks.com.

[81] The MathWorks Inc. *Stateflow: User's Guide*. Available at http://www.mathworks.com.

[82] Microchip Technology Inc. Available at http://www.microchip.com.

[83] Microchip Technology Inc. *MPASM Assembler, MPLINK Object Linker, MPLIB Object Librarian User's Guide*. Available at http://www.microchip.com.

[84] Microchip Technology Inc. *MPLAB C18 C Compiler User's Guide*. Available at http://www.microchip.com.

[85] Microchip Technology Inc. *MPLAB IDE, Simulator, Editor User's Guide*. Available at http://www.microchip.com.

[86] Microchip Technology Inc. *PIC18FXX2 Data Sheet*. Available at http://www.microchip.com.

[87] Robin Milner. *A Calculus of Communication Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[88] Jayadev Misra. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, 18(1):39–65, 1986.

[89] A.K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, Massachussetts Institute of Technology, 1983.

[90] μITRON. μITRON Standard, Version 4.0. Available at www.sakamura-lab.org/TRON/ITRON/DOC/iim00/microITRON4.pdf.

[91] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[92] Praveen K. Murthy, Shuvra S. Bhattacharyya, and Edward A. Lee. Joint Minimization of Code and Data for Synchronous Dataflow Programs. *Formal Methods in System Design*, 11(1):41–70, 1997.

[93] Marco Di Natale. Optimizing the Multitask Implementation of Multirate Simulink Models. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 335–346, Washington, DC, USA, 2006. IEEE Computer Society.

[94] Marco Di Natale, Guoqiang Wang, and Alberto Sangiovanni Vincentelli. Optimizing the Implementation of Communication in Synchronous Reactive Models. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 169–179, 2008.

[95] Zainalabedin Navabi. *VHDL: Analysis and Modeling of Digital Systems*. McGraw-Hill, Inc., New York, NY, USA, 1997.

[96] Mirabelle Nebut. An Overview of the Signal Clock Calculus. In *Synchronous Languages, Applications, and Programming*, volume 88, Porto, Portugal, July 2003. Electronic Notes in Theoretical Computer Science.

[97] EE249 Lecture Notes. Design of Embedded Systems: Models, Validation, and Synthesis. University of California, Berkeley, available at http://www.eecs.berkeley.edu/Courses/.

[98] EE290N Lecture Notes. Advanced Topics in System Theory: Concurrent Models of Computation for Embedded Software. University of California, Berkeley, available at http://www.eecs.berkeley.edu/Courses/.

[99] Muhittin Oral and Ossama Kettani. A Linearization Procedure for Quadratic and Cubic Mixed-Integer Problems. *Operations Reseach*, 40(S1):109–116, 1992.

[100] OSEK. OSEK Implementation Language (OIL), Version 2.5. Available at http://www.osek-vdx.org.

[101] OSEK. OSEK OS, Version 2.2.3. Available at http://www.osek-vdx.org.

[102] Martin Otter and Hilding Elmqvist. Modelica – Language, Libraries, Tools, Workshop and EU-project RealSi. German Aerospace Center, Oberpfaenhofen, Germany and Dynasim AB, Lund, Sweden, 2001.

[103] Pierre G. Paulin, Clifford Liem, Marco Cornero, Franois Naabal, and Gert Goossens. Embedded Software in Real-Time Signal Processing Systems: Application and Architecture Trends. In *Proceedings of the IEEE*, pages 419–435, 1997.

[104] POSIX. POSIX Standard. Available at http://www.posix.com.

[105] Pragmatec. *PICos18: Real-Time Kernel for PIC18*. Available at http://www.picos18.com.

[106] Pragmatec SARL Inc. Available at http://www.pragmatec.net.

[107] Alberto Sangiovanni-Vincentelli. Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[108] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-Based Design and Software Design Methodology for Embedded Systems. *IEEE Design and Test of Computers*, 18(6):23–33, 2001.

[109] N. Scaife and P. Caspi. Integrating Model-Based Design and Preemptive Scheduling in Mixed Time- and Event-Triggered Systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 119–126, Washington, DC, USA, 2004. IEEE Computer Society.

[110] Erwin Schoitsch. Embedded Systems - Introduction. In *ERCIM News: Sperical - Embedded Systems*, number 52, January 2003.

[111] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE transaction on computers*, 39(9):1175–1185, September 1990.

[112] Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin. *Operating System Concepts*. John Wiley & Sons, six edition, 2003. ISBN: 978-0-471-25060-9.

[113] Christos Sofronis, Stavros Tripakis, and Paul Caspi. A Memory-Optimal Buffering Protocol for Preservation of Synchronous Semantics under Preemptive Scheduling. *Proceedings of the 6th ACM International Conference on Embedded Software*, October 2006.

[114] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language (4th edition)*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[115] Stavros Tripakis, Claudio Pinello, Albert Benveniste, Alberto Sangiovanni-Vincent, Paul Caspi, and Marco Di Natale. Implementing Synchronous Models on Loosely Time Triggered Architectures. *IEEE Transactions on Computers*, 57(10):1300–1314, 2008.

[116] Stavros Tripakis, Christos Sofronis, Norman Scaife, and Paul Caspi. Semantics-Preserving and Memory-Efficient Implementation of Inter-Task Communication on Static-Priority or EDF Schedulers. *Proceedings of the 5th ACM International Conference on Embedded Software*, 2005.

[117] Frank Vahid and Tony Givargis. *Embedded System Design: A Unified Hardware/Software Introduction.* John Wiley & Sons, 2002.

[118] P.H.A. van der Putten, J.P.M. Voeten, M.C.W. Geilen, and M.P.J. Stevens. System Level Design Methodology. *Proceedings of IEEE Computer Society Workshop on System Level Design*, pages 11–16, April 1998.

[119] Alberto Valderruten Vidal, Manuel Vilares Ferro, and Jorge Grana Gil. Instrumentation of Synchronous Reactive Models for Performance Engineering. In *European Software Engineering Conference*, pages 76–89, 1995.

[120] Guoqiang Wang. The Enhanced PICos18: an O(1) OSEK-compliant Real-Time Operating System. Master's thesis, The Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, December 2007.

[121] Guoqiang Wang, Marco Di Natale, and Alberto L. Sangiovanni-Vincentelli. Spatial and Temporal Cost Analysis on OSEK Implementations of Synchronous Reactive

Semantics Preserving Communication Protocols. Technical Report UCB/EECS-2008-149, EECS Department, University of California, Berkeley, December 2008.

[122] Guoqiang Wang, Marco Di Natale, and Alberto Sangiovanni Vincentelli. An OSEK/VDX Implementation of Synchronous Reactive Semantics Preserving Communication Protocols. In *Proceesings of the Workshop on Operating Systemss Platforms for Embedded Real-Time Applications*, pages 38–47, Pisa, Italy, July 2007.