ARTES

A network for Real-Time research and graduate Education in Sweden

Real-Time Graduate Student Conference 2000

Chalmers University of Technology

UPPSALA UNIVERSITY

Swedish Foundation for Strategic Research

Empty page.

Preface

This volume contains the papers presented at the 2nd ARTES Graduate Student Conference, held at Chalmers University of Technology in Göteborg, from March 16 to 17, 2000.

As of today, more than 70 graduate students have joined the ARTES network by registering as Real-Time Graduate Students, and thereby getting access to the benefits provided by ARTES, including free-of-charge participation at the ARTES Summer School, as well as at other ARTES conferences and meetings, not to mention the mobility support provided by ARTES. The 70+ real-time graduate students clearly indicate that a substantial amount of real-time research is conducted in Sweden today. Due to ARTES and other efforts, it is fair to say that Sweden is one of the world-leaders in real-time systems research. During 1999 more than 20 papers authored/co-authored by Swedish researchers were presented at the four leading real-time conferences: IEEE Real-Time Systems Symposium (RTSS), Euromicro Conference on Real-Time Systems (RTS), IEEE Real-Time Technology and Applications Symposium (RTAS) and The International Conference on Real-Time Computing Systems and Applications (RTCSA). In 2000 we expect this number to increase even further. The ARTES Graduate Student conference is one of the ARTES supported activities aiming at ensuring this.

The main idea with the ARTES Graduate Student Conference is to provide a forum for technical presentations and discussions among the Swedish graduate students active in the real-time area. For newly recruited graduate students it will provide an opportunity to experience ”a real conference situation” (maybe) for the first time. For everyone, the conference will be an excellent opportunity to, in a relatively short time, get an overview of the current state of the national research.

As an extra bonus, we have invited Prof. Jack Stankovic from University of Virginia, one of the founding fathers of modern real-time systems research. He will give an exciting tutorial on *Feedback Control Real-Time Scheduling*, as well as sharing his experiences and views on the work being presented. Our second invited speaker is Alexander Dean from CMU, who will talk about *Automating Hardware to Software Migration for Real-Time Embedded Systems.*

This conference has been organized with the support from the Department of Computer Engineering at Chalmers, for which we are grateful. I would in particular like to thank Ewa Wäingelin and Jan Jonsson for their excellent support. As always, ARTES deputy programme director Roland Grönroos has provided invaluable assistance in organizing the event.

The papers included in this volume show an impressing width and quality, and I’m certain that the conference will be an event with intense technical and other discussion.

Enjoy it!

Hans A. Hansson
ARTES Programme Director
http://www.docs.uu.se/artes/
http://www.mrtc.mdh.se/han/

UPPSALA UNIVERSITY

ARTES
A network for Real-Time research and graduate Education in Sweden

Swedish Foundation for Strategic Research

About ARTES 2000

The ARTES initiative.
Representants for Svenska NAtionella RealTidsföreningen (SNART) called on Bernt Ericson, at that time chairman in SSFs IT-group, in february 1995. Bernt asked Hans Hansson (at that time chairman in SNART) to coordinate the planning of a Swedish network for real-time research.

ARTES goals are

1. **to increase the number of PhDs and Licentiates** which play important roles in development of industrial real-time applications and products. Concretely, the long term goal is 50 graduate degrees per year in ARTES related areas (the level 1997 was about 20 degrees per year), and that at least 80% of the graduates start an industrial career.

   ARTES will reach 30 supported active PhD student this year, from 2004 will the number of exams increase and reach the goal at about 2005.

2. **to increase the efficiency of graduate education**. The goal is that no ARTES graduate student should have a study-time exceeding the nominal times of 2 years for a licentiate and 4 years for a PhD (compensating for departmental duties and periods of industrial work).

   It looks good so far, the PhD students keep a good speed.

3. **active industrial involvement in research and graduate education, as well as academic involvement in industry**. Concretely, each ARTES supported project should have at least one active cooperation activity, e.g., in the form of an industrial engineer participating in the project, or a graduate student performing parts of his/her research in industry.

   This goal has been reached.

4. **to maximise synergy between the real-time components** in strategic centers supported by SSF, as well as with other efforts in the area. Concretely, at least 50% of the ARTES supported projects should have a formalised or informal cooperation (e.g., in terms of joint papers) with related national programmes.

   All projects is carried out in research groups with support from several capital providers. ARTES has not evaluated how well this goal has been fullfilled.

5. **to increase national and international cooperation** in real-time systems research and education, and

   ARTES have increased the national cooperation within the area and stimulated international cooperation.

6. **to provide a broad base** for Swedish real-time systems research, and **to make Swedish real-time systems research world leading** in selected areas.

   ARTES have increased the cooperation in the area. Besides research are ARTES supporting SIGURT (SNART:s interest group for undergraduate education). This may lead to a higher level on grauate education due to introduction of more real-time isues in undergraduate levels. Swedish real time research is (almost) worldleader in several areas this is reflected by the contributions to international conferences (see Thomas Lundqvists, Man Lins och Andreas Ermedahls, reports in ARTES Mobility reports). The number of Swedish publications in leading Real time conferences has doubled the last year.

IEEE Real-Time Systems Symposium (RTSS'99):	3
Euromicro Conference on RTS (ECRTS'99):	4
IEEE RT Technology and Appl. Symp. (RTAS'99)	2
R-T Computing Systems and Appl. (RTCSA'99)	12

Roland Grönroos

Address	*Visiting address*	*Phone*	*Fax*	*Internet*
ARTES Box 325	*Polacksbacken, building 1, room 239*	*+46 18 471 6847*	*+46 18 550 225*	artes@docs.uu.se
SE-751 05 Uppsala,	*Lägerhyddvägen 2*			*http://www.artes.uu.se/*
Sweden	*Uppsala*			

Programme for
ARTES Real-Time Graduate Student Conference 2000

Thursday March 16

09.30-10.00	Registration and Coffee	
10.00-10.10	Introduction	Hans Hansson
10.10-11.00	Tutorial: Feedback Control Real-Time Scheduling	Jack Stankovic
11.00-11.10	Break	
11.10-11.30	Tutorial (cont'd)	
11.30-12.00	Using Full Simulation for Debugging Real-Time Properties of Operating Systems	Lars Albertsson, SICS
12.00-13.00	Lunch	
13.00-14.00	**Session on RTS Design I**	
	Modelling of Real-Time Embedded Systems in an Object-Oriented Design Environment with UML	Sorin Manolache, LiU
	Real-Time System Constraints: Where do They Come From and Where do They Go?	Cecilia Ekelin, Chalmers
	A Real-Time Animator for Hybrid Systems	T. Amnell, A. David, UU
14.00-14.10	Break	
14.10-15.10	**Session on RT networking**	
	Deadline Dependent Coding - A Framework for Wireless Real-Time Communication	Elisabeth Uhlemann, HH
	Wireless Networks for Manufacturing	Urban Bilstrup, HH
	Methods for integration of Heterogeneous Real-Time Services into High-Performance Networks	Carl Bergenhem, HH
15.10-15.40	Coffee	
15.40-16.40	**Session on RTS Design II**	
	Research Issues on System Architecture for Mechatronics Systems	De-jiu Chen, KTH
	Formal and Probabilistic Arguments for Component Reuse in Safety-Critical Real-Time Systems	Anders Wall, MdH
	Performance Tuning of Multithreaded Applications for Different Multiprocessor Platforms	Magnus Broberg, HK/R
16.40-16.50	Break	
16.50-17.50	**Session on execution time analysis**	
	Supporting Timing Analysis by Automatic Bounding of Loop Iterations	Mikael Sjödin, UU
	Bounding The Execution Time of Method Calls Combining Static Analysis and Dynamic Class Loading	Patrik Persson, LTH
	Obtaining execution time for small program fragments by testing	Markus Lindgren, MdH
18.00	Closing	
19.30	Conference Dinner	

Friday March 17

08.30-09.30	Automating Hardware to Software Migration for RT Embedded Systems	Alexander Dean. CMU
09.30-09.40	Break	
09.40-10.20	**Session on Multiprocessor Scheduling**	
	Fixed-Priority Preemptive Multiprocessor Scheduling: To Partition or not to Partition	Björn Andersson, Chalmers
	Dynamic Replication Decisions in Fault-Tolerant Multiprocessor-Based Real-Time Systems	Monika Andersson Wiklund, Chalmers
10.20-10.40	Coffee	
10.40-12.00	**Session on Fault Tolerance and Control**	
	Using Massive Time Redundancy to Achieve Node-level Transient Fault Tolerance	Joakim Aidemark, Chalmers
	AIDA II Automatic control in distributed applications	O. Redell, J. Elkhoury, KTH
	Feedback Scheduling of Control Tasks	Anton Cervin, LTH
	Schedulability Analysis for Systems with Data and Control Dependencies	Paul Pop, LiTH
12.00-12.10	Summary and closing	
12.10	Lunch	

Participants at ARTES graduate student day 2000

Name	E-mail	Affiliation
Joakim Aidemark	aidemark@ce.chalmers.se	Chalmers
Lars Albertsson	lalle@sics.se	SICS
Tobias Amnell	tobiasa@docs.uu.se	Uppsala University
Björn Andersson	ba@ce.chalmers.se	Chalmers
Monika Andersson Wiklund	monikaw@ce.chalmers.se	Chalmers
Bengt Asker	bengt.asker@mailbox.swipnet.se	ARTES
Johan Bengtsson	johanb@DoCS.UU.SE	Uppsala University
Carl Bergenhem	carl.bergenhem@ide.hh.se	Högskolan i Halmstad
Urban Bilstrup	Urban.Bilstrup@ide.hh.se	Högskolan i Halmstad
Magnus Broberg	magnusb@mailix.ide.hk-r.se	Högskolan Karlskrona/Ronneby
Anton Cervin	anton@control.lth.se	Lund Institute of Technology
De-Jui Chen	chen@damek.kth.se	KTH
Alexandre David	adavid@docs.uu.se	Uppsala University
Alexander Dean	adean@gop.ece.cmu.edu	Carnegie Mellon University
Julien d'Orso	juldor@docs.uu.se	Uppsala University
Cecilia Ekelin	cekelin@ce.chalmers.se	Chalmers
Jad Elkhoury	jad@md.kth.se	KTH
Jakob Engblom	jakob@docs.uu.se	IAR Systems AB & UU
Andreas Ermedahl	ebbe@csd.uu.se	Uppsala University
Roland Grönroos	Roland.Gronroos@docs.uu.se	ARTES
Sanny Gustavsson	sanny@ida.his.se	Högskolan i Skövde

Jörgen Hansson	jorha@ida.liu.se	Linköping University
Arshad Jhumka	arshad@ce.chalmers.se	Chalmers
Markus Lindgren	markus.lindgren@mdh.se	Mälardalens högskola
Birgitta Lindström	a96birli@ida.his.se	Högskolan i Skövde
Thomas Lundqvist	thomasl@ce.chalmers.se	Chalmers
Sorin Manolache	g-sorma@ida.liu.se	Linköping University
Simin Nadjm-Tehrani	snt@ida.liu.se	Linköping University
Robert Nilsson	robert.nilsson@ida.his.se	Högskolan i Skövde
Jonas Norberg		KTH
Patrik Persson	Patrik.Persson@cs.lth.se	Lund University
Paul Pop	paupo@ida.liu.se	Linköping University
Mikael Sjödin	mic@docs.uu.se	Uppsala University
John A. Stankovic	stankovic@cs.virginia.edu	University of Virginia
Samuel Stolberg-Rohr	samuels@md.kth.se	KTH
Diana Szentivanyi	diasz@ida.liu.se	Linköping University
Mattias Tullberg		KTH
Martin Törngren	martin@damek.kth.se	KTH
Elisabeth Uhlemann	elisabeth.uhlemann@cca.hh.se	Högskolan i Halmstad
Anders Wall	anders.wall@mdh.se	Mälardalens högskola
Mattias Weckstén		Högskolan i Halmstad

Empty page.

Feedback Control Real-Time Scheduling

Prof Jack Stankovic

BP America Professor and Chair
Department of Computer Science
University of Virginia

Despite the significant body of results in real-time scheduling, many real world problems are not easily supported. While algorithms such as Earliest Deadline First, Rate Monotonic, and the Spring scheduling algorithm can support sophisticated task set characteristics (such as deadlines, precedence constraints, shared resources, jitters, etc.), they are all open loop scheduling algorithms. Open loop refers to the fact that once schedules are created they are not adjusted based on continuous feedback. While open-loop scheduling algorithms can perform well in static or dynamic systems in which the workloads can be accurately modeled, they can perform poorly in unpredictable dynamic systems. In this project we are developing a theory and practice of feedback control real-time scheduling. Feedback control real-time scheduling defines error terms for schedules, monitors the error, and continuously adjust the schedules to maintain required and stable performance. We have developed a practical feedback control real-time scheduling algorithm, FC-EDF, which is a starting point in the long-term endeavor of creating of theory and practice of feedback control scheduling. We are applying out results to applications such as web servers, agile manufacturing, and defense systems. We are also proposing a novel way to specify and measure complex real-time systems based on control theory.

Updated Friday, 04-Feb-2000 16:55 by Roland Grönroos
e-mail: artes@docs.uu.se
Location: http://www.docs.uu.se/artes/events/gsconf00/jack_s2.shtml

Feedback Control Real-Time Scheduling

Professor Jack Stankovic
Department of Computer Science
University of Virginia

Outline of Presentation

- Motivation
- Our Goal
- Feedback Control Scheduling Architecture
- *The FC-EDF Algorithm*
- *Specifications and Metrics*
- Simulation Environment
- Performance Results
- Summary

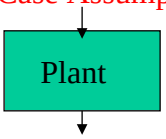
Motivation

- The emergence of soft real-time systems in unpredictable environments
  - agile manufacturing
  - command and control or other defense applications
  - web browsers (audio and video)
  - real-time database systems
- For Dynamic RTS in unpredictable environments
  - WCET too pessimistic, high variance in execution time, unbounded arrival rate, overload unavoidable

Applications Requirement

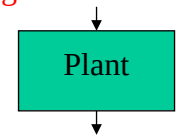
- Missing some deadlines inevitable and OK
  - Agile manufacturing
    - part or product is produced wrong

Worst Case Assumptions



100 products/day
No errors
$10 profit/product
Profit - $1,000

Average Case + FC Scheduling



1010 products/day
10 errors ($2 penalty)
$10 profit/product
Profit - $9,980

Motivation - What's Available

- RT Scheduling Paradigms
  - Static - predictable
    - all is known a priori (WCET, invocation times or worst case rates, resource needs, precedence, etc.)
    - Cyclic scheduling, RM, table driven, …
    - open loop
  - Dynamic - high degree of predictability
    - all is known except invocation times/rates (use WCET)
    - Spring algorithm - admission control and planning, or EDF
    - open loop

Claims

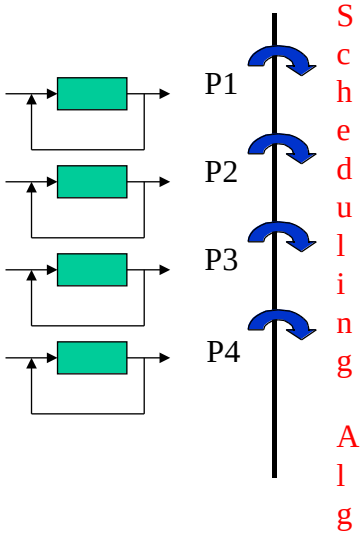
- Despite the significant body of results in real-time scheduling many real world problems and *NOT* easily supported!
- We need a new real-time scheduling paradigm based on *feedback* *!!!*

Goals

- *Theory and Practice of Feedback Control Real-Time Scheduling*
- PID control (but not restricted to this)
  - does not require precise analytical model of the system being controlled
  - not ad hoc either
- Explicit use of deadline based metrics
- New Specifications and Metrics for Soft Real-Time Systems

Feedback

- Front-End
  - feedback loops based on real world control
  - generate timing requirements/rates
  - generally fixed
  - handed to scheduling algorithm; recent work => tighter linkage
  - *Future: deadline range, more flexible periods*



Feedback

- Our Work - Applied to Scheduling itself
  - Multi-level feedback queue (ad hoc)
  - Network traffic flow - bandwidth or utilization
  - AED - RTDB (proportional control)
  - *Our work:*
    - Based on FC theory
    - Explicit use of deadline based metrics
    - PID control
  - *Future: Direct linkage to front-end (deadline range)*

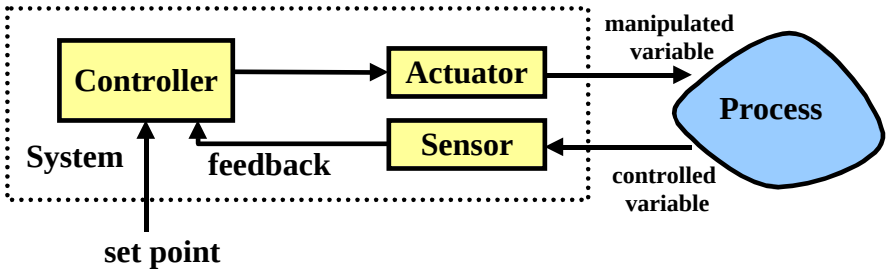
Practical Issues

- Not all deadlines are created equal (missing some is worse than missing others)
- Missing n in a row for one periodic task can cause instability
- Start simple: treat all deadlines as equal in importance

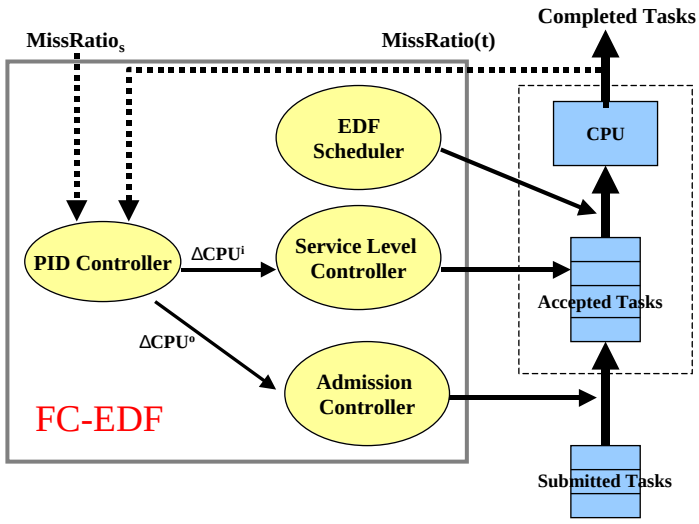
Approach

- Mapping feedback control concepts to scheduling
- Establishing control model for scheduling
  - map MR to utilization
- Applying control design/analysis to the scheduler to satisfy performance requirements
- System-wide evaluation - requirements and metrics from control theory

Feedback Control System



FC-EDF Scheduling Algorithm



Task Model

Each Task T

ET(1)	Val(1)
ET(2)	Val(2)
.	.
.	.
.	.
ET(N)	Val(N)

Service Levels

Nominal execution times used - not worst case

PID controller

- The PID controller periodically (sampling period) maps the error in term of the deadline miss ratio to the control signal - the required change in the requested CPU utilization

$$\Delta CPU(t) = C_P error(t) + C_I \sum_{IW} error(t) + C_D \frac{error(t) - error(t - DW)}{DW}$$

- PID control only requires an *approximate* system model to achieve satisfactory performance
- Deadline miss ratio is *directly* controlled

Service Level Controller

- Input - Delta CPU
- choose least important “degradable” task
- reduce level by 1, 2, …
- iterate
- if not all Delta CPU is accommodated, then pass the rest of the Delta CPU to AC

Admission Control

- New arrivals
  - Admit task i with highest service level possible such that the CPU(t) + ET(i) < 1.
  - Reject task if its lowest service level not acceptable
- When PID needs to reduce or increase CPU
  - Accommodate as much as possible of Delta CPU in service level controller
  - Adjust estimate of cpu utilization in system to modify amount admitted (in the future)

Two Questions

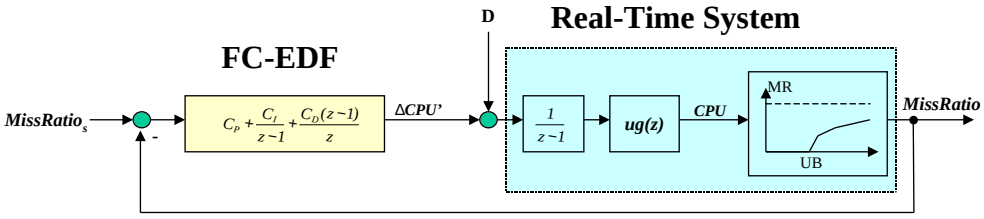
- Determine coefficients of PID control terms
- Establish link between MR and Delta CPU

PID Control Parameters

- How do we choose the parameters?
  - Use control theory
    - create model of system
    - solve model to get the *form* of the parameters

Example: CI > 0; CD < 1, 2CP - CI + 4CD < 4 and
2-2*CD*CD > CD*CP + CP - CI > 0

Scheduling Model



- *MissRatio(z)*: miss ratio
- $MissRatio_s$: miss ratio set point
- *CPU'(z)*: estimated CPU utilization
- *CPU(z)*: actual CPU utilization
- Δ*CPU'(z)*: the change in *CPU'(z)*
- *CPU(z) = ug(z)CPU'(z)*

Soft Real-Time Systems

- Larger and larger
- More complex
- Operate in non-deterministic environments
- Hypothesis: Current requirements specification and metrics are not sufficient
- Proposing: A (new) set of specifications and measurements based on control theory.

Today's Soft Real-Time Systems

- Specifications
  - meet as many deadlines as possible
  - meet 95% of the deadlines
- Metrics
  - missed deadline percentage
- Build Large System
  - manipulate algorithms, system parameters, etc. and run some "workload" to see what happens
  - iterate

A Proposed *(New)* Idea

- Exploit the notion of a Theory and Practice of Feedback Control Scheduling
- Specifications for a control system often involve requirements associated with the time response of the system
  - Overshoot
  - Settling time
  - Rise time

Specification - Example

- Miss deadline percentage (MDP) should be 2%
- Overshoot
  - MDP should not get greater than 10%
- Settling time - the time it takes the system transients to decay
  - settling time to return to the 2% MDP should be within 3 seconds

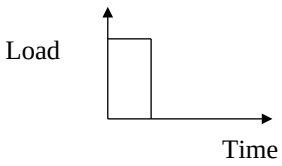
Measurements

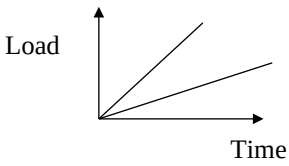
- MDP
- Overshoot
- Settling Time
- Rise Time
- Peak Time
- Steady State Error

- Sensitivity - relative change to MDP with respect to a relative change of a system parameter
  - execution time of tasks from the average
  - blocking time
  - external arrivals

Workloads

- Series of step functions as input load
  - burst arrivals of various sizes
- Series of ramp functions with different slopes
  - period of continuing increased load





Result

- A more complete picture of your system performance
  - average MDP (are you meeting the goal)
  - how far off from MDP will your system stray for given “stressing” workloads
  - does the system exhibit any steady state error
  - on overload - how soon does the system return to normal
  - what are the parameters that affect your performance the most, the least

Simulation Experiments

- Baseline algorithms
  - *EDF*: Basic EDF scheduling algorithm
  - *EDF+AC*: EDF with static admission control
  - *EDF+P*: EDF with proportional feedback control
- Periodic and independent tasks
- *execution time factor (etf):* The measure of the unpredictability of the workload. The smaller *etf* is, the more pessimistic the estimation is.

$$etf = avg\_exe\_time / est\_exe\_time$$

Experiments

- Experiment A
  - statistical description of average execution times is the same throughout the simulation
  - vary - how far the actual execution time is off from the estimated, called eft
    - etf = 1 => no error; etf < 1 => pessimistic; etf > 1 => optimistic
- Experiment B
  - statistical description of average execution times varies over the simulation

Experiment A

- Experiment A tests the robustness of the scheduler when the average execution time keeps the same but deviates from the estimation.
- Fixed *etf* for each run.
- Performance Metrics
  - *MRA:* Miss Ratio among admitted tasks
  - *UTIL:* CPU Utilization
  - *HRS:* Hit Ratio among submitted tasks
  - *VCR:* Value Completion Ratio

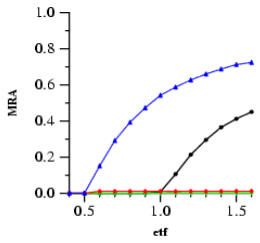


Figure 5 MRA vs. etf

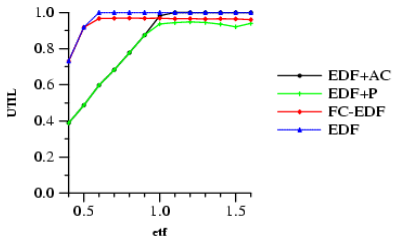


Figure 6 UTIL vs. etf

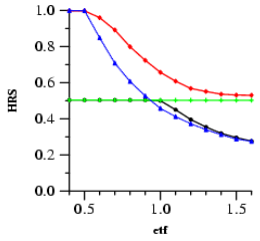


Figure 7 HRS vs. etf

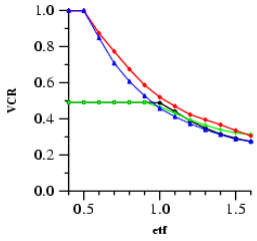


Figure 8 VCR vs. etf

Experiment B

- Experiment B tests the performance under conditions when the execution time changes dynamically.
- *etf* is changed every 300 sampling periods.

Interval(SP)	0-300	300-600	600-900	900-1200
etf	0.8	1.3	0.8	1.2

- Performance Metrics
  - *MRA:* Miss Ratio among admitted tasks
  - *UTIL:* CPU Utilization

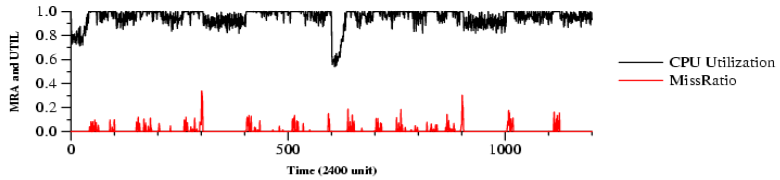


Figure 9 Sampling MissRatio and CPU Utilization of FC-EDF

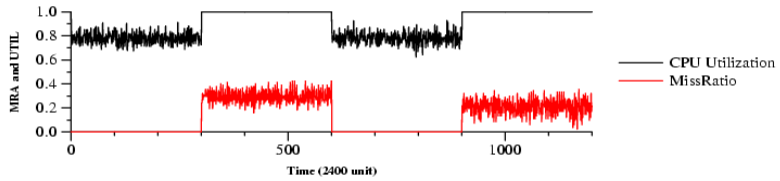


Figure 10 Sampling MissRatio and CPU Utilization of EDF+AC

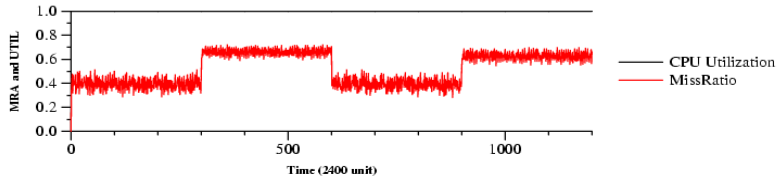


Figure 11 Sampling MissRatio and CPU Utilization of EDF

Additional Metrics

Load - step	162.5%
Overshoot	35%
Settling Time	720 ms
Steady-State MR	1%
Sensitivity (ET)	0

Summary

- Goal - a theory and practice of feedback control real-time scheduling
- Real-time systems in unpredictable environments
- FC-EDF
- *Future*
  - ♦ Link with front-end control
  - ♦ New performance specifications
  - ♦ Model non-linearities
  - ♦ More sophisticated deadline miss semantics
  - ♦ Control design methodology
  - ♦ Real-time web servers
  - ♦ BeeHive real-time database testbed

Related Work

- *Ad hoc* feedback scheduling
  - Mutilevel feedback queue [Blevins76]
  - Adaptive Earliest Deadline [Haritsa91]
- Adaptation mechanisms
  - Deadline/period adjustment [Abde97][Becc99][Shin99]
  - Multimedia QoS [Steinmetz95]
  - Imprecise computation [Liu91]
- Feedback resource management in non-rt area
  - Web resource management [Abde99]
  - Distributed multimedia QoS [Li98][Cen97]

Empty page.

Design and Evaluation of a Feedback Control EDF Scheduling Algorithm*

Chenyang Lu John A. Stankovic Gang Tao† Sang H. Son

Department of Computer Science, †Department of Electrical Engineering
University of Virginia, Charlottesville, VA22903
e-mail: {cl7v, stankovic, son}@cs.virginia.edu, †gt9s@virginia.edu

Abstract

Despite the significant body of results in real-time scheduling, many real world problems are not easily supported. While algorithms such as Earliest Deadline First, Rate Monotonic, and the Spring scheduling algorithm can support sophisticated task set characteristics (such as deadlines, precedence constraints, shared resources, jitter, etc.), they are all "open loop" scheduling algorithms. Open loop refers to the fact that once schedules are created they are not "adjusted" based on continuous feedback. While open-loop scheduling algorithms can perform well in static or dynamic systems in which the workloads can be accurately modeled, they can perform poorly in unpredictable dynamic systems. In this paper, we present a feedback control real-time scheduling algorithm and its evaluation. Performance results demonstrate the effectiveness of the algorithm when execution times vary from the worst case and when there are major shifts of total load in the system. A key part of this feedback solution is its explicit use of deadline based metrics.

1. Motivation and Introduction

Real-time scheduling algorithms fall into two categories: *static* and *dynamic* scheduling. In static scheduling, the scheduling algorithm has complete knowledge of the task set and its constraints, such as deadlines, computation times, precedence constraints, and future release times. The Rate Monotonic (RM) algorithm and its extensions [Liu73][Leho89] are static scheduling algorithms and represent one major paradigm for real-time scheduling. In dynamic scheduling, however, the scheduling algorithm does not have the complete knowledge of the task set or its timing constraints. For example, new task activations, not known to the algorithm when it is scheduling the current task set, may arrive at a future unknown time. Dynamic scheduling can be further divided into two categories: scheduling algorithms that work in *resource sufficient* environments and those that work in *resource insufficient* environments. Resource sufficient environments are systems where the system resources are sufficient to *a priori* guarantee that, even though tasks arrive dynamically, at any given time all the tasks are schedulable. Under certain conditions, Earliest Deadline First (EDF) [Liu73] is an optimal dynamic scheduling algorithm in resource sufficient environments. EDF is a second major paradigm for real-time scheduling [Stan98]. While real-time system designers try to design the system with sufficient resources, because of cost and highly unpredictable environments, it is sometimes impossible to guarantee that the system resources are sufficient. In this case, EDF's performance degrades rapidly in overload situations. The Spring scheduling algorithm [Rama84][Zhao87] can dynamically guarantee incoming tasks via on-line admission control and planning and thus is applicable in resource insufficient environments. Many other algorithms (e.g., RED algorithm [Butt95]) have also been developed to operate in this way. This planning-based set of algorithms represents the third major paradigm for real-time scheduling. However, despite the significant body of results in these three paradigms of real-time scheduling, many real world problems are not easily supported. While algorithms such as EDF, RM and the Spring scheduling algorithm can support sophisticated task set characteristics (such as deadlines, precedence constraints, shared resources, jitter, etc.), they are all "open loop" scheduling algorithms. Open loop refers to the fact that once schedules are created they are not "adjusted" based on continuous feedback. While open-loop scheduling algorithms can perform well in static or dynamic systems in which the workloads (i.e., task sets) can be accurately modeled, they can perform poorly in *unpredictable* dynamic systems, i.e., systems whose workloads cannot be accurately modeled. For example, the Spring scheduling algorithm assumes complete knowledge of the task set

* Supported in part by NSF grant CCR-9901706 and contract IJRP-9803-6 from the Ministry of Information and Communication of Korea.

except for their future release times. Systems with open-loop schedulers such as the Spring scheduling algorithm are usually designed based on *worst-case* workload parameters. When accurate system workload models are not available, such an approach can result in a highly underutilized system based on extremely pessimistic estimation of workload.

Unfortunately, many real-world complex problems such as agile manufacturing, robotics, adaptive fault tolerance, and C4I and other defense applications are not predictable. Because of this, it is impossible to meet every task deadline. The objective of the system is to meet as many deadlines as possible[1]. For example, autonomous robotic systems always suffer from sudden variations in computational load and overload situations due to highly variable execution times in robot control algorithms (e.g., sensor interpretation, motion planning, inverse kinematics and inverse dynamics algorithms) [Becc99]. For another example, in information and decision support systems, accurate knowledge about transaction resource and data requirements is usually not known *a priori*. The execution time and resource requirements of a transaction may be dependent on user input or dependent on sensor values. For these applications, a design based on the estimation of worst case execution times will result in extremely expensive and underutilized system. It is more cost effective to design for less than worst case, but sometimes miss deadlines.

Another important issue is that these scheduling paradigms all assume that timing requirements are known and fixed. The assumption is that control engineers design the system front-end control loops and generate resulting timing requirements for tasks. The scheduling algorithms then work with this fixed set of timing requirements. Real control systems, in general, are much more flexible and robust, e.g., instead of choosing a single deadline for a task which is passed on to the scheduling system, a deadline range might be acceptable to the physical system. If this range was passed to the scheduling system, the on-line scheduling might be more robust.

We believe that due to all these problems, solutions based on a new paradigm of scheduling, which we call feedback control real-time scheduling, is necessary for some important systems. A case for this was made in [Stan99]. In this current paper we fully develop a feedback control scheduling algorithm, discuss stability, and present the evaluation of the algorithm.

In the past, many forms of feedback control have appeared in real-time and non-real-time scheduling systems, but it has not been elevated to a central principle; rather most of the time it is used more as an afterthought and is usually *ad hoc*. Most of these techniques also did not address the key performance metric of real-time systems - the deadline miss ratios. More discussion of this appears later in the related work section.

[1] If such systems have some critical tasks, they are treated separately by static allocation of resources.

2. Approach

The mapping of control theory methodology and analysis to scheduling provides a systematic and scientific method for designing scheduling algorithms. Many aspects of this mapping are straightforward, but others require significant insight and future research. We now describe how such a mapping can be done and what research is necessary. In section 3 we present an instance of this mapping by creating an actual algorithm and a runtime scheduling structure.

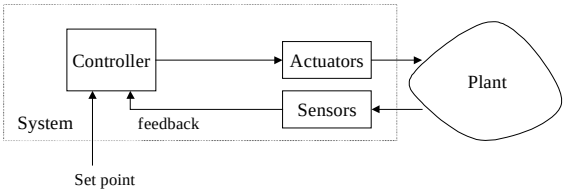


Figure 1 Architecture of Feedback Control Systems

2.1. Control theory and real-time scheduling

A typical feedback control system is composed of a controller, a plant to be controlled, actuators, and sensors (as illustrated in Figure 1). It defines a *controlled variable*, the quantity of the output that is measured and controlled. The *set point* represents the correct value of the controlled variable. The difference between the current value of the controlled variable and the set point is the *error*. The *manipulated variable* is the quantity that is varied by the controller so as to affect the value of the controlled variable. The system is composed of a feedback loop as follows. (1) The system periodically monitors and compares the controlled variable to the set point to determine the error. (2) The controller computes the required control with the control function of the system based on the error. (3) The actuators change the value of the manipulated variable to control the system. In the context of real-time scheduling problems, our approach is to regard a scheduling system as a feedback control system, and the scheduler as the controller. The scheduler utilizes feedback control techniques to achieve satisfactory system performance in spite of unpredictable system dynamics. We believe that the long term potential for a theory and practice of feedback control scheduling is significant; partly because we can also build upon the vast amount of knowledge and experience from control systems.

As a starting point, we will apply PID (Proportional-Integral-Derivative) control in schedulers. A basic form PID control formula is

$$Control\ (t) = C_P Error\ (t) + C_I \int Error\ (t)dt + C_D \frac{dError\ (t)}{dt} \qquad (1)$$

We choose PID control as the basic feedback control techniques in feedback control scheduling for the following reasons. (1) In term of control theory, the scheduling system is a *dynamic* system, i.e., a system whose output depends not only on the current input, but also on the previous system inputs. It is known that the current miss ratio of a scheduling system depends not only on the currently submitted task, but also on the previously submitted tasks that remain in the system for queuing, execution and blocking. It is known that PID control is a widely applicable control technique in dynamic systems. (2) Compared with other control techniques, an important feature of PID control is that it does not require a precise analytical model of the system being controlled. Instead, a PID controller designed based on an approximate model can achieve satisfactory performance. Due to the extremely complex behavior of current computer systems, it is impossible to precisely model the dynamics of real world scheduling systems. However, certain form of approximate modeling of the scheduling system can be built to help tune the PID control parameters more systematically (such an approximate model is presented in section 3.6 of this paper). (3) According to control theory, basic PID control can provide stable control in first and second order dynamic systems. In systems with higher order of dynamics, however, basic PID control can only provide approximate control, but an adaptive form of PID control can provide stable control for high order dynamic systems.

2.2. Feedback control real-time scheduling

To apply feedback control techniques in scheduling, we need to restructure schedulers based on the feedback control framework. We need to identify the controlled variable, the manipulated variable, the set point, the error, the control function, and the mechanisms of the actuators. We can then set up the feedback loops based on these selections. The choice of the controlled variable depends on the system goal. For example, the performance of real-time systems usually depends on how many tasks make (miss) their deadlines. We define the system deadline miss ratio as the percentage of tasks that miss their deadlines; this is a natural choice of the controlled variable. The manipulated variable must be able to affect the value of the controlled variable. In the real-time scheduling, it is a widely known fact that the deadline miss ratio highly depends on the system load, i.e., the requested CPU utilization of tasks in the system. Thus the requested CPU utilization can be used as the manipulated variable. Other possible choices of manipulated variables are the periods/deadlines of tasks in control applications when there exists the flexibility to adjust these task parameters [Seto96].

In summary, a feedback control scheduling system would start with a schedule based on the nominal assumptions of the incoming tasks (expected start time, expected execution time and deadline). The system would then monitor the actual performance of the schedule, compare it to the system requirements and detect differences. The system would call control functions to assess the impact of these differences and apply a correction to keep the system within an acceptable range of performance. Research is needed to answer the following open questions:

- What are the right choices of controlled variables, manipulated variables, set points, and effective control functions/mechanisms for feedback control scheduling?
- How to model a feedback control scheduling system?
- How to tune the control parameters to build a stable and high performance scheduler?
- How to integrate the flexible timing constraints derived from the front-end feedback control loops in control systems with an on-line feedback control scheduling algorithm?
- What is the impact of overhead in feedback control scheduling and how to minimize it?

3. Feedback Control EDF

We now present an algorithm called Feedback Control EDF (FC-EDF), which integrates PID control with an EDF scheduler. With FC-EDF, we demonstrate how to structure a real-time scheduler based on the feedback control framework. We will identify specific research issues related to feedback control scheduling using FC-EDF as a concrete example. The FC-EDF architecture (shown in Figure 2) can be generalized to be used as a framework of feedback control schedulers. For example, we can investigate feedback control RM by replacing the EDF scheduler in Figure 2 with a RM scheduler.

3.1. Overview of FC-EDF

To apply PID control to a scheduling system, we need to decide on the components of a scheduling system corresponding to those in a feedback control system (Figure 1). First, we need to choose the controlled variable and the set point of the system. The requirements of an ideal soft real-time scheduling algorithm should be to (1) provide (soft) performance guarantees to admitted tasks, i.e., maintain low miss ratio among admitted tasks; and (2) achieve high system throughput and utilization. To satisfy these requirements, FC-EDF chooses miss ratio among the admitted tasks, *MissRatio(t),* as the controlled variable and an (application dependent) small but non-zero value (e.g., $MissRatio_s$ = 1%) as the set point. Note that 0 is not chosen as the set point for the following reasons. A system with a set point of $MissRatio_s$ = 0 can ignore the second requirement of soft real-time scheduling, i.e., high utilization and throughput. When a system achieves a 0% deadline miss ratio but causes extremely low utilization (e.g., by unnecessarily rejecting too many tasks), a feedback control scheduler with $MissRatio_s$ = 0 will treat it as the correct state. In contrast, a feedback control scheduler with

a set point of $MissRatio_s \neq 0$ will always try to (lightly) overload the system to achieve high utilization. Note that in an unpredictable environment, it is impossible for a system to achieve 100% utilization and 0% miss ratio all the time and a tradeoff between miss ratio and utilization is unavoidable. There can be two approaches to deal with this tradeoff. The approach of admission control based on pessimistic estimation is the *pessimistic* approach, which always *avoids* deadline misses at the cost of low utilization and throughput. This approach has been widely used in hard real-time systems. On the other hand, the feedback control scheduling presented in this paper represents the *optimistic* approach, which maintains a low (but possibly non-zero) miss ratio and high utilization and throughput. When a high misses actually happens due to system load changes, the scheduler *corrects* the system state back to the satisfactory state, i.e., a state with low miss ratio and high utilization and throughput. This optimistic approach is especially preferable in soft real-time systems since it provides a soft performance guarantee in term of miss ratios while achieving high utilization and throughput at the same time (see performance evaluation in section 4).

Second, we choose the requested CPU utilization, i.e., the total CPU utilization requested by all the accepted tasks in the system, as the manipulated variable. The rational is that EDF can guarantee a miss ratio of 0% given the system is not overloaded, and in normal situations, the deadline miss ratios increases as the system load increases[2]. For simplicity of description, we will use requested utilization in place of requested CPU utilization in this paper. Third, we need to design the mechanisms (i.e., actuators) used by the scheduler to manipulate the requested utilization. An Admission Controller and a Service Level Controller are included in the FC-EDF scheduler as the mechanisms to manipulate the requested utilization. The Admission Controller can control the flow of workload into the system, and the Service Level Controller can adjust the workload inside the system.

The FC-EDF scheduler is composed of a PID controller, a Service Level controller, an Admission Controller and an EDF scheduler (Figure 2). The system performance *MissRatio(t)* is periodically fed back to the PID controller. Using the PID control formula (1), the PID controller computes the required control action $\Delta CPU(t)$, i.e., the total amount of CPU load that need to be added into (when $\Delta CPU(t)>0$) or reduced from (when $\Delta CPU(t)<0$) the system. Then the PID controller calls the Service-Level Controller and the Admission Controller to change the CPU load of the system by ΔCPU. This system control forms a feedback control loop in the scheduling system. The EDF scheduler schedules the accepted tasks according to the EDF policy.

[2] This is under the assumption that the domino effect is rare in real world applications.

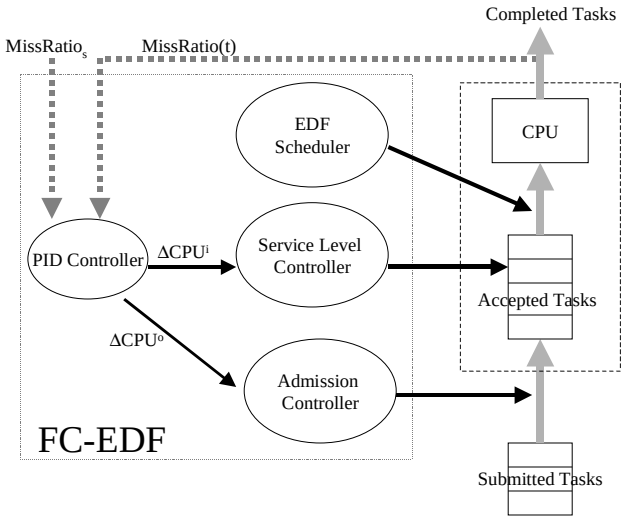


Figure 2 Architecture of FC-EDF

3.2. Task model

Our initial task model assumes that all tasks have soft deadlines and all the tasks are independent. For the convenience of description, this paper assumes a task model similar to the imprecise computation model [Liu91], but the scheduling algorithms presented do not depend on the imprecise computation model. Each task T_i submitted to the system is described with a tuple (I, ET, VAL, S, D). Each task T_i has one or more logical versions $I = (T_{i1}, T_{i2}, \ldots T_{ik})$. Note that when a task has multiple logical versions it does not necessarily mean that it has multiple implementations. An imprecise computation can have several different forms including milestone method, sieve function method or multiple version method [Liu91]. We call all these methods that can tradeoff computation value and time as multiple logical versions of a task for convenience of discussion. Each version has different execution time and different value. $ET = \{ET_{i1}, ET_{i2}, \ldots ET_{ik}\}$ (suppose $ET_{i1} \geq ET_{i2} \geq \ldots \geq ET_{ik}$) are the nominal execution times of different versions. Here, nominal execution time instead of worst case execution time is used in the system to achieve higher CPU utilization in the system. The execution time is described in the form of requested utilization. For example, $ET_{i1}=0.02$ means the 1st version of task T_i requires 2% of the CPU time. $VAL = \{VAL_{i1}, VAL_{i2}, \ldots VAL_{ik}\}$ represents the values of different implementation. In this research, different versions of a task are called service levels. We call a version with longer execution time and higher value a higher service level than another version with less execution time and lower value. Each task has a soft deadline D_i and a start time S_i.

Note that FC-EDF does not depend on the imprecise computation model. FC-EDF only needs a certain flexibility to adjust the CPU utilization. In our future work, we will extend the deadline D_i to a range of deadlines $(D_{i,min}, D_{i,max})$.

The deadline of a task T_i could be adjusted dynamically within the range. By adjusting the deadline of a task, the Service Level Controller can effectively change the requested CPU utilization in the system. This extension is based on the fact that digital control systems are usually robust, i.e., the task timing constraints are allowed to vary within a certain range without affecting critical control functions such as maintenance of system stability [Seto96]. Such extension is also applicable in multimedia systems, in which the QoS specifications of a multimedia application can be specified as intervals.

3.3. PID controller

The PID controller is the core of FC-EDF. It maps the miss ratio of accepted tasks (i.e., error) to the change in requested utilization (i.e., control signal) so as to drive the miss ratio back to the set point.

The PID controller periodically monitors the controlled variable *MissRatio(t)*, and computes the control $\Delta CPU(t)$ in terms of requested utilization with the following control formula, which is an approximation of formula (1).

$$\Delta CPU(t) = C_P error(t) + C_I \sum_{IW} error(t) + C_D \frac{error(t) - error(t - DW)}{DW} \quad (2)$$

where $error(t) = MissRatio_s - MissRatio(t)$. SP, C_P, C_I, C_D, IW and DW are tunable parameters of the PID controller. SP is the sampling period. The PID controller will be called every SP seconds. For convenience of presentation, we will take SP as the time unit in our following discussion. C_P, C_I and C_D are the coefficients of the PID controller. IW is the time window over which to sum the errors. Only errors in the last IW time units will be considered in the integral term. DW is the time window of derivative errors. Only the error change during the last DW time units will be considered in the derivative term (i.e., the derivative error is $(MissRatio(t\text{-}DW)\text{-}MissRatio(t))/DW$). The tuning of these parameters will be discussed in section 3.6. $\Delta CPU(t)$ is the output (i.e., control signal) of the PID controller. $\Delta CPU(t)>0$ means that the requested utilization should be increased, and $\Delta CPU(t)<0$ means that the requested utilization should be decreased.

The PID controller will call the Service Level Controller and the Admission controller to change the requested utilization of the system by ΔCPU. If the current requested utilization is $CPU(t)$, the Service Level controller and Admission Controller will change the requested utilization to $CPU(t)+\Delta CPU$. The PID controller always tries to change the requested utilization internally by calling Service Level Controller first so that the system could respond to the error faster. The admission controller is called only if the Service Level Controller cannot accommodate ΔCPU completely.

3.4. Service Level Controller

The Service Level Controller (SLC) changes the requested utilization in the system by adjusting the service levels of accepted tasks. For example, if it changes the service level of task T_i from T_{ik} to T_{ij}, it adjusts the requested utilization of the system by ET_{ij}-ET_{ik}, where ET_{ij} and ET_{ik} are the estimated CPU requirement of T_{ik} and T_{ij}, respectively. SLC returns the portion of ΔCPU not accommodated. This portion of control will be accommodated by the Admission Controller

3.5. Admission Controller

The Admission Controller (AC) controls the flow of workload into the system. When a new task T_i is submitted to the system, AC decides on whether it could be accepted into the system. Given current system-wide requested utilization $CPU(t)$ and the CPU requirement of the incoming task, the Admission Controller admits T_i with service level k if k is the highest level that satisfies $CPU(t)+ET_{ik}<1$; T_i is rejected if it cannot be admitted even with the lowest level.

The Admission Controller's parameter $CPU(t)$ may be adjusted when the PID controller cannot accommodate $\Delta CPU(t)$ completely with SLC. Suppose SLC changes the requested utilization by ΔCPU^i and $\Delta CPU^i<\Delta CPU$, AC will accommodate $\Delta CPU^o=\Delta CPU(t)-\Delta CPU^i$, the portion of ΔCPU not accommodated by SLC. It accommodates ΔCPU^o by adjusting the estimation of requested utilization $CPU(t)$ as following $est\_cpu\_util = 1 - \Delta CPU^o$. The Admission Controller will admit the workload of at most ΔCPU^o (when $\Delta CPU^o > 0$) or will not admit any new tasks until the system load has been reduced by amount of more than $|\Delta CPU^o|$ (when $\Delta CPU^o < 0$).

3.6. Modeling and Analysis of feedback control scheduling

3.6.1. Feedback control scheduling model

An important task of building a stable and high-performance PID-control-based scheduler is to tune the control parameters (i.e., C_P, C_I and C_D). One way is to tune the parameters by simulations. However, this approach would require large amount of experiments to gain enough confidence in the selected values of the parameters. A more scientific and systematic approach is to apply control theory analysis to select the PID control parameters. Such analysis requires an analytical model of the scheduling system. Before we present the model of a PID-control based scheduling system, we define the following notions in the discrete time domain:

1. SP is a constant sampling period, which is the time elapsed in interval [k, k+1], k being time instants.
2. *MissRatio(z)*: the miss ratio - the system output and the controlled variable.

3. $MissRatio_s$: the set point (i.e., the target) in term of miss ratio.
4. *CPU'(z)*: the estimated CPU utilization;
5. *CPU(z)*: the CPU utilization;
6. *ΔCPU'(z)*: the change in the estimated CPU utilization (*CPU'(z)*) . This is the system input;
7. *ug(k)*: the ratio of the actual total utilization to the estimation;
8. *mrg(k)*: the gain that maps the utilization (*CPU(z)*) to the miss ratio (*MissRatio(z)*).
9. *d(k)*: the disturbance (e.g., associated with the utilization bound).

Using the above notations, the estimated CPU utilization follows the following equation:

$$CPU'(z) = \Delta CPU'(z)/(z-1) \quad (3)$$

Since the precise execution time of each task is unknown and changes over time, the actual requested utilization is usually different from the estimated requested utilization. In term of control theory, *ug(k)* is a time-variant *gain* (called *utilization gain* in this paper) of the system. We can get the (actual) CPU utilization using *ug(k)*:

$$CPU(z) = ug(k)CPU'(z) \quad (4)$$

We can derive the deadline miss ratio based on the correlation between the deadline miss ratio and the system utilization. The relationship between the deadline miss ratio and the requested utilization can be modeled as

$$MissRatio(z) = mrg(k)CPU(z) - d(k) \quad (5)$$

where *mrg(k)* (*miss ratio gain*) is a time-variant *gain* that maps *CPU(z)* to *MissRatio(z)* in overload situations.

The transfer function of the PID controller is:

$$H(z) = C_P + C_I/(z-1) + C_D(z-1)/z \quad (6)$$

In summary, we present the block diagram of the FC-EDF scheduling system in Figure 3.

3.6.2. Stability Analysis

For a control system, there are two forms of stability: internal stability and BIBO stability. *Internal stability* is related to the system behavior due to initial conditions. If no input is applied, an internally stable system will settle to be close to an equilibrium set point within a definite amount of time. Although a real-time system could enter an internally unstable mode when the miss ratio rises monotonically over time (e.g., when deadlock occurs) even when there is no change in the workload, this paper will assume an independent task model and the system is assumed internally stable when it is made BIBO stable. The issue of internal stability will be investigated in our future research. *BIBO stability* means that the system output is always bounded for a bounded input. In the context of the scheduling system, this means that the miss ratio will be bounded for bounded changes in the workload. It is important to tune the PID controller such that the BIBO stability is satisfied to avoid uncontrollable performance degradation in a scheduling system. For convenience of description, we will use stability in place of BIBO stability in this paper. To demonstrate the importance of using a formal theory to describe feedback control real-time scheduling, we present the following stability analysis result (The mathematical proof is skipped due to the length limit on this paper).

Theorem 1: When *ug(k)mrg(k)* is close to 1, the FC-EDF scheduling system established by the block diagram in Figure 3 is stable if one of the following conditions is satisfied:

1. $C_I > 0$, $|C_D| < 1$, $2C_P - C_I + 4C_D < 4$, and $2 - 2C_D^2 > C_DC_P + C_P - C_I > 0$
2. $C_I = 0$, $|C_D| < 1$, and $0 < C_P + 2C_D < 2$

The assumption that *ug(k)mrg(k)* is close to 1 implies that (1) the estimated utilization is not too far from the actual utilization and (2) domino effect does not happen. Under this assumption (which we believe is true for most soft real-time systems), the stability condition gives the guidance on how to set the coefficients in the PID control such that the satisfactory scheduling performance can be maintained. This is one important reason for basing feedback control real-time scheduling on control theory.

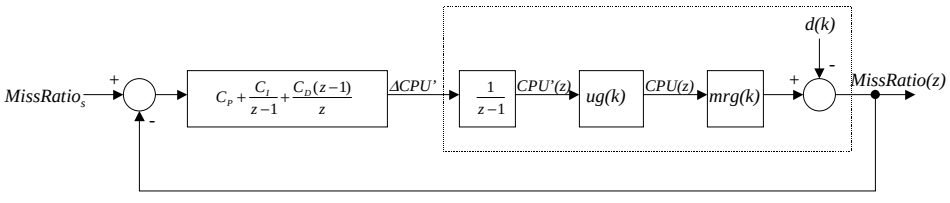


Figure 3. Block diagram of the FC-EDF scheduling system

4. Experiments and Results

4.1. Simulation Model

An uniprocessor simulator of a soft real-time system was used to study the performance of FC-EDF and baseline algorithms. The workload consists of independent periodic tasks. Each task instance has a deadline that equals its period. If a task instance is not completed by its deadline, it is immediately aborted. The simulator (Figure 4) has six components: *sources* that generate tasks; an *admission controller* that make admission/rejection decisions on submitted tasks; an *executor* that models the execution of the tasks; a *monitor* that periodically collects the performance statistics of the system; a *PID controller* that periodically computes control signals based on the performance errors; and a *service level controller* that adjusts the service levels of the tasks. A basic EDF scheduler is embedded in the executor. The admission controller, service level controller, and PID controller each can be turned on/off to emulate the different baseline scheduling algorithms.

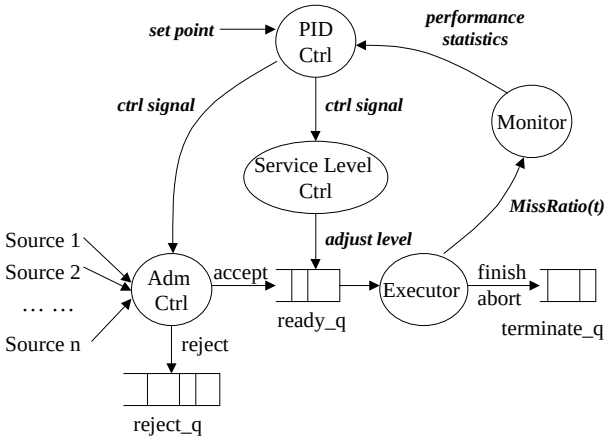


Figure 4. Feedback Control Scheduling Simulator

4.2. Workload Model

Each source is characterized with a period (P) (the deadline of each task instance equals its period), a set of worst case execution times $\{WCET_i\}$, a set of best case execution times $\{BCET_i\}$, a set of estimated execution times $\{EET_i\}$, a set of average execution times $\{AET_i\}$ ($WCET_i \geq AET_i \geq BCET_i$), and a set of values $\{VAL_i\}$, where $\#service\_level-1 \geq i \geq 0$. Each tuple ($P$, $WCET_i$, $BCET_i$, EET_i, AET_i, VAL_i) characterizes a service level and

$$EET_i = (WCET_i + BCET_i) * 0.5$$
$$AET_i = EET_i * etf$$

where *etf* (*execution time factor)* can be tuned to change the accuracy of the estimation. The larger the *etf* is, the more pessimistic the estimation is. For example, *etf* of 1.0 means that the estimated execution time is equal to the actual average execution time. The estimation is less than the average execution time when $etf < 1.0$; and greater than the average execution time when $etf > 1.0$. The actual execution time of each instance (which is unknown to the scheduler) is computed as a uniform random variable in interval $[AET_i, WCET_i]$ or $[BCET_i, AET_i]$ depending on a random *Bernoulli* trial with probability $(AET_i - BCET_i)/(WCET_i - BCET_i)$. In our experiments, the workload is composed of 40 periodic tasks and each task has two service levels. $WCET_0 = 2*WCET_1$, $WCET_i = 4*BCET_i$, $VAL_0 = 1.0$ and $VAL_1 = 0.5$. It follows that *etf* lies in the range of [0.4, 1.6]. A uniform distribution is used to generate $WCET_0$ and P of each task: $WCET_0 = uniform(5,10)$, $P = WCET_0 * uniform(10,15)$. P is adjusted such that 31 of the tasks are harmonious with least common multiplier of 2400 time units. The same workload is used for all the experiments. Each different experiment has varied *etf*. All the tasks arrive in the beginning of each experiment to overload the system. Each rejected task will attempt to get re-admitted all through the experiment.

4.3. Implementation of FC-EDF

The parameter settings of FC-EDF in all the experiments are listed in Table 1. The set point is the soft real-time performance target miss ratio that the scheduler will achieve. In practice, this value depends on the tolerance (to deadline misses) of the application. The *SP* is equal to the least common multiplier of the majority of the tasks such that similar number of task instances are submitted in each *SP*. *IW* controls the length of history that the controller need to consider. *DW* affects the agility of the controller to the sudden change in workload. C_P, C_I and C_D are coefficients of the PID controller. Note that the values of C_P, C_I and C_D are tuned so that they satisfy the stability condition established in Theorem 1 to guarantee the scheduling performance.

Set Point	0.01
SP	2400 (time unit)
IW	100 (SP)
DW	1 (SP)
C_P	0.5
C_I	0.05
C_D	0.1

Table 1 FC-EDF parameter setting

4.4. Baseline Algorithms

The following baseline algorithms are implemented and compared with FC-EDF in the experiments.

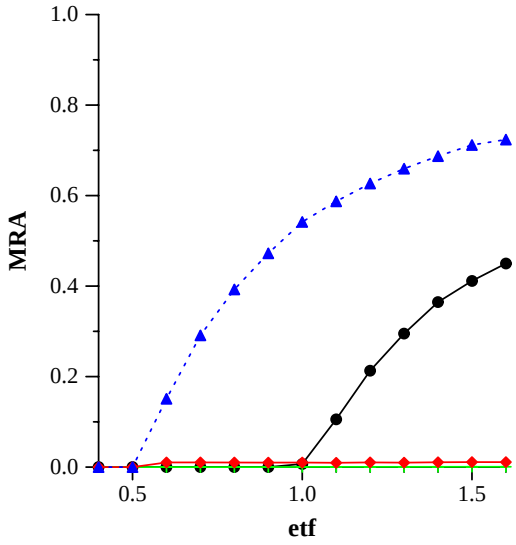


Figure 5 MRA vs. etf

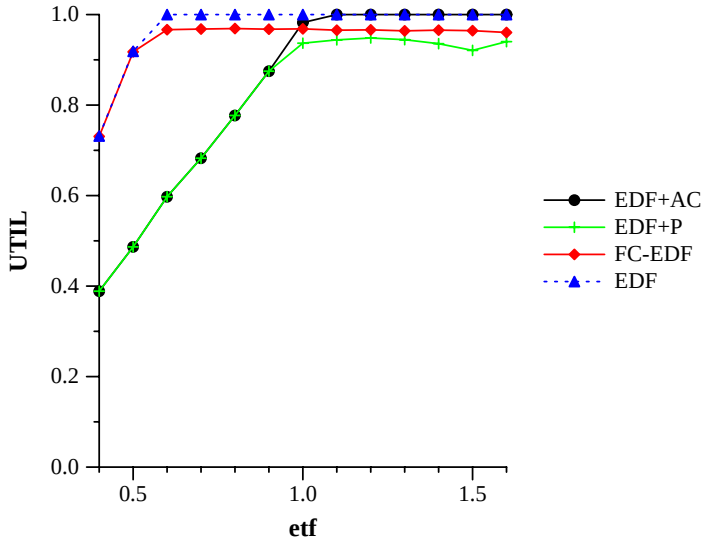


Figure 6 UTIL vs. etf

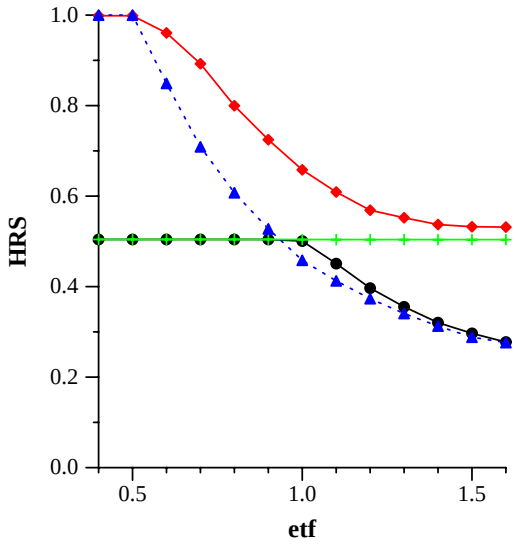


Figure 7 HRS vs. etf

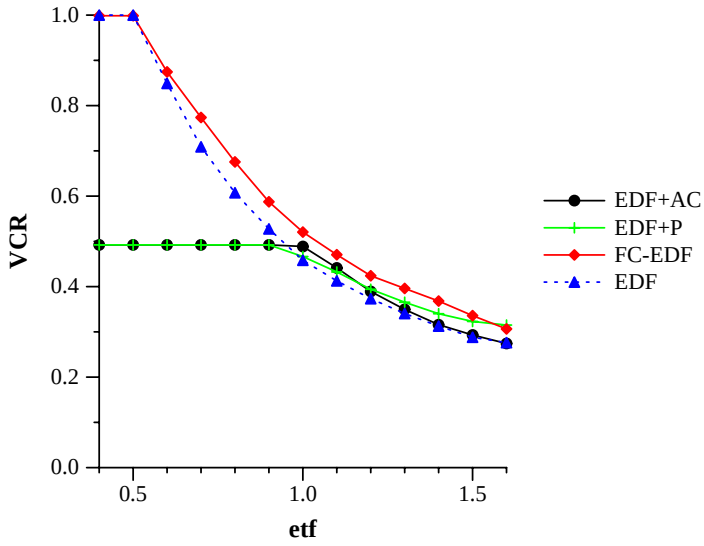


Figure 8 VCR vs. etf

- *EDF*: This is the basic EDF scheduling algorithm. It is implemented by turning off the admission controller, service level controller, and the PID controller.
- *EDF+AC* (EDF with static admission control): EDF+AC is implemented by turning off the service level controller and the PID controller. The admission decision is made once for each periodic task when it is submitted.
- *EDF+P* (EDF with proportional control): EDF+P is implemented and configured the same as FC-EDF except for $C_P = C_D = 0$. This algorithm is to test whether the simple proportional control can provide comparable performance as PID control.

4.5. Performance Metrics

- *Miss Ratio among admitted tasks* (*MRA = #Misses / #(Admitted Task Instances)*): Although a soft real-time task can tolerate a small percent of deadline misses, it usually requires a *soft guarantee* in term of miss ratio in order to maintain its normal functionality. Therefore, the average miss ratio among admitted tasks should be the most important performance metric.
- *CPU Utilization* (*UTIL*): Soft real-time systems should avoid under-utilizing the system resources to reduce costs, thus the CPU utilization is another metric under consideration.
- *Hit Ratio among submitted tasks* (*HRS = #Hits / #(Submitted Task Instances)*): Hit ratio among submitted tasks is a measure of the system throughput in term of the percentage of task instances completed in time among all the arrivals.
- *Value Completion Ratio* (*VCR =(Σ(Values of hit task instances) / Σ(Values of Submitted Task Instances)*): Value

completion ratio considers the quality of the results. A task that misses its deadline contributes 0 value. A task instance completed with a lower service level contributes a lower value.

4.6. Experiment A: Steady execution time

Experiment A evaluates the performance of the scheduling algorithms (EDF, EDF+AC, EDF+P and FC-EDF) when the average execution time of each periodic task is statistically steady, but different from the estimation. *etf* is constant through each run, but the actual execution time of each task instance is a random value that differs over time. Each algorithm made 13 set of runs with each set for a different *etf* values in the range [0.4, 1.6]. These experiments are interesting since, in real-world applications, it is impossible to make the estimation of execution time the same as the average actual execution time at all times. This problem is avoided in hard real-time systems by using worst-case estimation (*etf* << 1.0). However, soft real-time systems usually use *nominal* estimations instead of worst-case estimations due to cost considerations. It is thus important for a soft real-time system to maintain satisfactory performance even when the estimation is different from the average actual execution times. This is one place the value of feedback control is demonstrated. Experiment A evaluates the algorithms under situations when the execution time is statistically steady. The performance under conditions when the execution time changes dramatically is studied in Experiment B. This is another place where feedback proves very valuable. Experiment A results are illustrated in Figures 5-8. Each point in Figure 5-8 is the average value of 30 runs. The 90% confidence interval for each point of *MRA*, *UTIL*, *HRS*, and *VCR* is within ±0.0006, ±0.0058, ±0.0066 and ±0.0046, respectively. Each run lasts for 2880000 time units (1200 SP).

From Figure 5, we see that both FC-EDF and EDF+P maintains a very low *MRA* (<1%) throughout the *etf* interval. The *MRA* of FC-EDF is closer to the set point (1%) and thus slightly higher than EDF+P since the Integral control reduces the steady state error of the control. EDF's *MRA* increases sharply when *etf* > 0.5 and the system is overloaded. This shows pure EDF cannot provide acceptable performance in resource-insufficient systems. EDF+AC maintains low *MRA* when *etf* < 1, however, its performance degrades sharply when the average execution time is larger than its estimation (*etf* > 1). This shows that static admission control based on nominal estimations cannot provide (soft) performance guarantees under all situations. Figure 6 shows that EDF always achieves the highest CPU utilization since it never rejects any tasks. EDF+AC under-utilizes the system when the average execution time is lower than the estimation (*etf* < 1). This shows that static admission control cannot avoid wasting the system resources under all situations. Similarly, EDF+P also under-utilizes the utilization when *etf* < 1 similar to EDF+AC. This is because EDF+P does not consider the accumulated error, and the control signal (*ΔCPU*) computed with proportional control alone is too small to be accommodated when the task utilization can only be adjusted in discrete steps[3]. In comparison, with the PID control, FC-EDF maintains high CPU utilization in all the situations. We can also see that the CPU utilization of FC-EDF is consistently higher than EDF+P even when (*etf* > 1).

Figures 7 and 8 compare the throughput of the algorithms in term of *HRS* and *VCR*. EDF+AC and EDF+P has a low throughput when (*etf* < 1) as a result of admission control based on pessimistic estimation. EDF has the highest throughput when the system is lightly loaded and degrades fast as the system load increases. We should also note EDF achieves its throughput by greedily accepting all the tasks while at the cost of poor performance for *each* task (no performance guarantees for admitted tasks). FC-EDF's throughput is slightly lower than EDF when the system is lightly loaded (This is actually a result in the initial phase and the PID control drives the throughput to the same level as EDF after a short learning period). When the system is overloaded, FC-EDF consistently achieves the highest throughput in all situations (except for when *etf* = 1.6 when it is slightly lower than EDF+P).

In summary, Experiment A demonstrates that under situations when the estimation of execution time differs from the (statistically steady) actual execution time, FC-EDF is able to (1) provide a soft performance guarantee for admitted tasks; (2) achieving high system utilization; and (3) high throughput. None of the other algorithms under comparison can meet these goals simultaneously.

4.7. Experiment B: Dynamic execution time

In experiment B, the *etf* is dynamically changed every 720000 time units (300 SP) in order to study the response of the scheduling algorithms when the average system load changes dramatically. The *etf* settings in different intervals are listed in table 2.

Interval (SP)	0-300	300-600	600-900	900-1200
etf	0.8	1.3	0.8	1.2

Table 2 *etf* setting in Experiment B

[3] While a larger C_P can amplify the control signal, it will also cause severe oscillation and even instability in the system response.

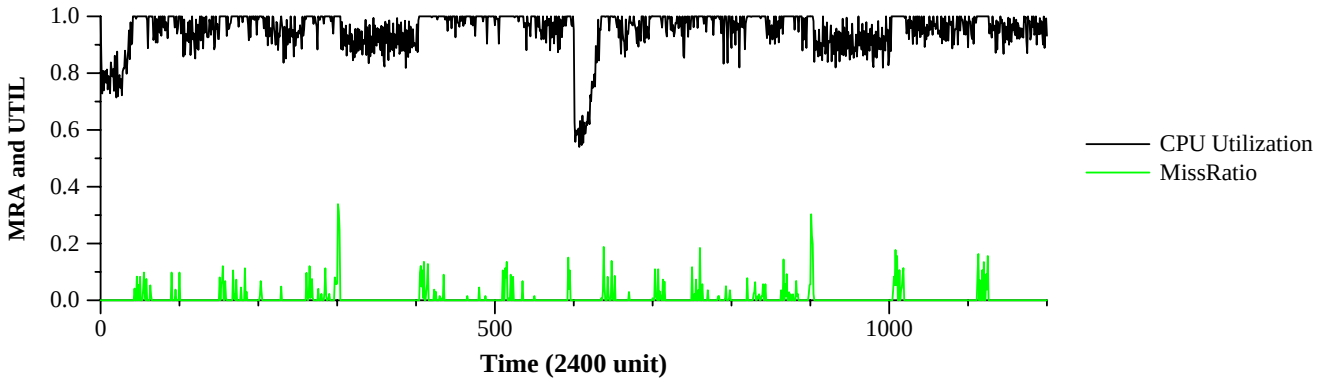


Figure 9 Sampling MissRatio and CPU Utilization of FC-EDF

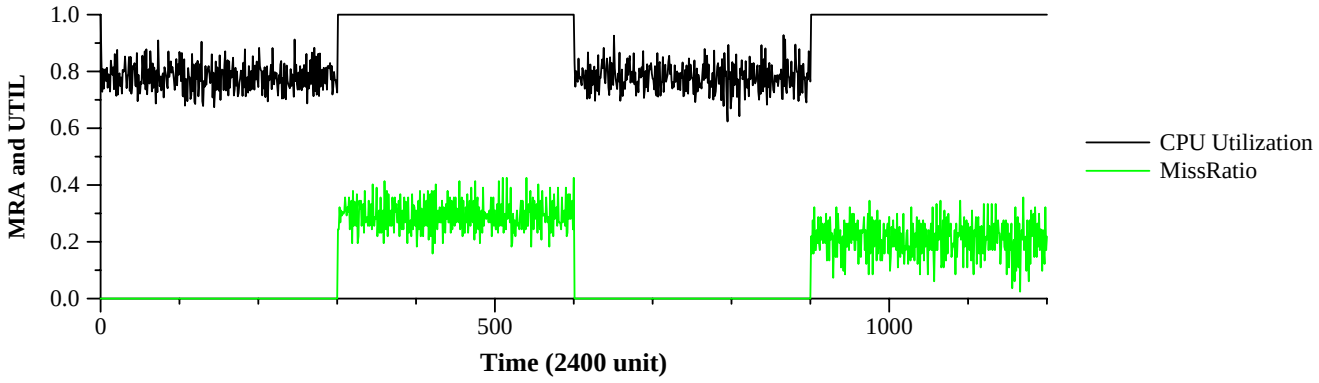


Figure 10 Sampling MissRatio and CPU Utilization of EDF+AC

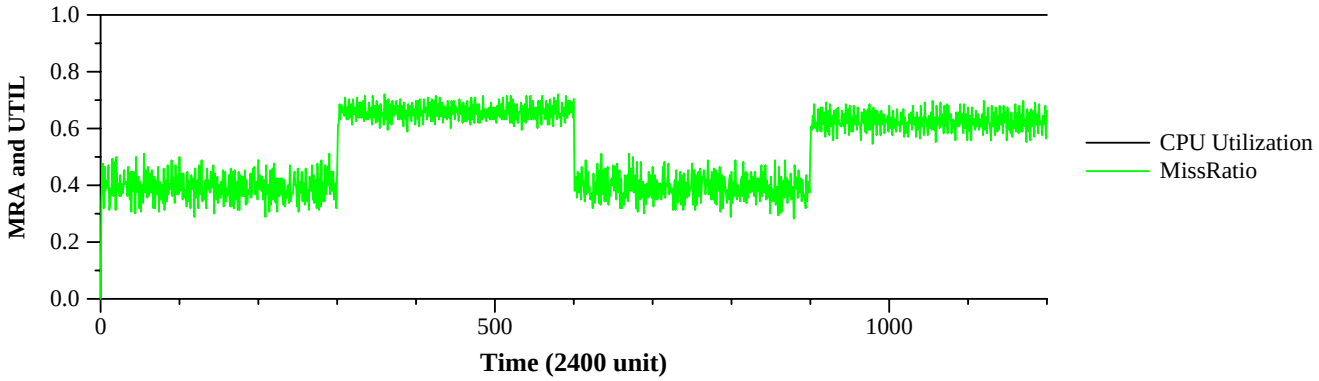


Figure 11 Sampling MissRatio and CPU Utilization of EDF

Figures 9-11 illustrates the sampled *MRA* and *UTIL* of FC-EDF, EDF+AC and EDF (EDF+P is dropped in this experiment since its performance is consistently worse than FC-EDF) in a typical run. From Figure 9, we can see that the system starts with approximately 80% of utilization (the top curve) and 0 miss ratio (the bottom curve) as a result of the pessimistic admission control. However, after a short learning period (43 SP), the PID control in FC-EDF causes the system to increase its utilization (by increasing service levels and admitting more tasks) to close to 100%. The system maintains a high utilization and low miss ratio until the end of 300 SP when the *etf* suddenly increases to 1.3, which implies that the system load suddenly increases by over 60%. This causes the big spike on miss ratio at time 300 SP in Figure 9. FC-EDF responds immediately by reducing the load (i.e., reducing service levels) to below 100% within 3 SP. In addition to the proportional control, the fast response is also due to the derivative control in FC-EDF which responses to the sudden increase in the miss ratio. The system comes back to low miss ratio and high utilization and stays this way until 600 SP when the *etf* is reduced to 0.8 (which corresponds to an over 60% loss in

system load). Again, the FC-EDF responds by increasing the system load. Within 35 SP, the utilization is driven back close to 100% and the system maintains low miss ratio and high utilization afterwards. At the end of 900 SP, the *etf* is increased to 1.2 and, similar to the time at 300 SP, the system is back to a satisfactory performance state. We can also see that all through the run, the *MRA* is 0 in most sampling periods but non-zero (around 10% except for the short transition periods between different *etf* intervals) in a small portion of sampling periods. This is the penalty for achieving high utilization (around 95% throughout the run except for the short transition periods) and throughput of the system. Note that due to the random nature of the system load, it is impossible to maintain both 0 miss ratio and 100% utilization all the time.

In contrast, from Figure 10, we see that EDF+AC achieves 0 miss ratio at the cost of low utilization (around 80%) in 0-300 SP and 600-900 SP (when *etf* < 1.0). However, the miss ratio is around 20-25% in 300-600 SP and 900-1200 SP (when *etf* > 1.0). From Figure 11, we can see the *MRA* is around 40% or 65% at different intervals for EDF, which performs even worse that EDF+AC.

	MRA	UTIL	HRS	VCR
FC-EDF	0.011	0.954	0.796	0.537
EDF+AC	0.127	0.889	0.440	0.431
EDF	0.518	1.000	0.482	0.482

Table 3 Overall performance in Experiment B

In summary, the overall performance of the algorithms is listed in Table 3 (Each data is the average value of 30 runs. The 90% confidence interval of each value of MRA, UTIL, HRS, and VCR is within ±0.0006, ±0.0023, ±0.0068, and ±0.0026, respectively). FC-EDF effectively adapts to the radical changes in the execution time and system load, and maintains satisfactory performance throughout the run time. In contrast, both EDF and EDF+AC are unable to handle the workload and render poor performance.

4.8. Discussions on overhead

Feedback control scheduling algorithms can introduce extra overhead compared with open-loop scheduling algorithms. The overhead is ignored in the simulation experiments for the following reasons. First, the feedback controller can be executed at a much lower rate than application tasks. In both experiments, FC-EDF maintains satisfactory performance while more than 100 task instances are executed between two subsequent invocations of the feedback controller. Second, the control algorithm is not complex. The bookkeeping of the miss ratio, the PID control function, and admission control adjustment all cost constant time. The service-level controller is a linear-time (in terms of the number of admitted tasks) algorithm. The overhead issue will be further investigated in our future research as we vary the period of the feedback controller.

5. Related Work

The idea of using feedback information to adjust the schedule has been used in general-purpose operating systems in the form of multi-level feedback queue scheduling [Blev76]. The system monitors each task to see if it consumes a time slice or does I/O and adjusts its priority accordingly. This type of control is a crude correction term that seems to work via *ad hoc* methods. No systematic study has been done. [Stee99] presented a feedback-based scheduling scheme that adjusts CPU allocation based on application-dependent *progress monitors*. This work is done in the context of general operating systems and the performance of real-time tasks is not addressed.

In the area of real-time databases, Haritsa et. al. [Hari91] proposed *Adaptive Earliest Deadline* (AED), a priority assignment policy based on EDF. In order to stabilize the performance of EDF under overload conditions, AED features a feedback control loop that monitors transactions' deadline miss ratio and adjusts transaction priority assignment accordingly. *Adaptive virtual deadline first* [Pang95] aims to achieve the fairness of an EDF based scheduler to transactions with different sizes. The linear correlation between deadline miss ratio and transaction size in the system is measured and fed back to the scheduler, which adjusts scheduling parameters dynamically. However, the feedback control in these algorithms is *ad hoc* and the issues of stability of the schedulers is not addressed.

[Seto96] and [Ryu98] both propose to integrate the design of a real-time controller with the scheduling of real-time control systems. [Seto96] introduced a system *performance index* to describe the control cost and tracking errors of a control system. Their approach was to assign frequencies to control tasks that optimize the system performance index under resource constraints in the system. [Ryu98] is based on a generic analytical feedback control system model, which expresses the system performance as functions of the end to end timing constraints at the system level. A heuristic algorithm is used to find the end-to-end timing constraints that can achieve the required system performance. They then use the *period calibration method* [Gerb95] to derive the timing constraints for each task in the system. Both [Seto96] and [Ryu98] aim at providing design tools that enable control engineers to take into consideration scheduling in the early design stage of control systems. The scheduling algorithms in both of these cases are still open loop scheduling algorithms.

[Becc99] and [Shin99] both utilized the flexible timing constraints as a mechanism for graceful performance degradation in control systems. Both of the works assumed the execution times are known and focused on how to reassign the periods for tasks to satisfy the utilization constraints. Instead, our work focuses on using feedback

control loops to maintain satisfactory deadline miss ratio when the task execution times change dynamically.

Jehuda and Israeli [Jehu98] proposed an *automated software meta-controller* to dynamically reconfigure real-time systems. Compared with feedback control real-time scheduling, the software meta-control is a higher level control that occurs only when the system mode changes.

In the area of multimedia communication, several papers [Meer97][Li98][Abde98] presented feedback control architectures and algorithms for QoS control. These works are targeted at supporting end-to-end QoS in distributed multimedia communication and they are not scheduling algorithms. Meeting task deadlines is not a focus of these works.

The idea and framework of feedback control scheduling has been presented in our earlier paper [Stan99]. As an evaluation and extension to our earlier work, this paper presents simulation experiments that verifies the advantage of feedback control scheduling in unpredictable dynamic environments. In this paper, we also present a discrete control model and stability analysis of the feedback control scheduling system.

6. Conclusion

In our work, we are systematically exploring the use of feedback control concepts in soft real-time scheduling systems. The goal is the development of a theory and practice of feedback control scheduling. This would be a new paradigm for real-time scheduling. We have also demonstrated feasibility of the basic approach by developing a specific algorithm called FC-EDF and scheduling architecture as presented here. Another contribution of our work is that the key performance metric of the real-time system – the deadline miss ratio – is *directly* controlled by the scheduler. The feedback control scheduler can achieve a soft performance guarantees in term of deadline miss ratios. An analytical model of the scheduling system is introduced and analyzed to facilitate tuning the scheduling algorithm systematically. The simulation experiments demonstrate that FC-EDF can maintain satisfactory deadline miss ratio and high system utilization when the workload changes dramatically.

Acknowledgment

The authors wish to thank Tarek Abdelzaher for finding an error in the earlier manuscript of this paper.

References

[Abde98] T. F. Abdelzaher and Kang G. Shin, "End-host Architecture for QoS-Adaptive Communication" *IEEE RTAS*, June 1998.

[Becc99] G. Beccari, et. al., "Rate Modulation of Soft Real-Time Tasks in Autonomous Robot Control Systems", *EuroMicro Conference on Real-Time Systems*, June 1999.

[Blev76] P. R. Blevins and C. V. Ramamoorthy, "Aspects of a dynamically adaptive operating systems", *IEEE Transactions on Computers*, Vol. 25, No. 7, pp. 713-725, July 1976.

[Butt95] G. Buttazzo and J. A. Stankovic, "Adding Robustness in Dynamic Preemptive Scheduling", *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems* (D. S. Fussell and M. Malek Ed.), Kluwer Academic Publishers, 1995.

[Gerb95] R. Gerber, S. Hong and M. Saksena, "Guaranteeing Real-Time Requirements with Resource-Based Calibration of Periodic Processes", *IEEE Transactions on Software Engineering*, Vol. 21, No. 7, July 1995.

[Hari91] J. R. Haritsa, M. Livny and M. J. Carey, "Earliest Deadline Scheduling for Real-Time Database Systems", *IEEE RTSS*, 1991.

[Jehu98] J. Jehuda and A. Israeli, "Automated Meta-Control for Adaptable Real-Time Software", *Real-Time Systems J.*, 14, 1998.

[Leho89] J. P. Lehoczky, L. Sha and Y. Ding, "The Rate Monotonic Scheduling Algorithm – Exact Characterization and Average Case Behavior", *IEEE RTSS*, 1989.

[Li98] B. Li, K. Nahrstedt, "A Control Theoretical Model for Quality of Service Adaptations", in *IEEE International Workshop on Quality of Service*, May 1998.

[Liu73] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *JACM*, Vol. 20, No. 1, pp. 46-61, 1973.

[Liu91] J. W. S. Liu, et. al., "Algorithms for Scheduling Imprecise Computations", *IEEE Computer*, Vol. 24, No. 5, May 1991.

[Meer97] J. B. de Meer, "On the Specification of EtE QoS Control", (A. Campbell and K. Nahrstedt Eds.) *Building QoS into Distributed Systems*, Chapman & Hall, 1997.

[Pang95] H. Pang and M. J. Carey, "Multiclass Query Scheduling in Real-Time Database System", *IEEE Trans. on Knowledge and Data Engineering*, vol. 7, no. 4, August 1995.

[Rama84] K. Ramamritham and J. A. Stankovic, "Dynamic task scheduling in distributed hard real-time systems", *IEEE Software*, Vol. 1, No. 3, July 1984.

[Ryu98] M. Ryu and S. Hong, "Toward Automatic Synthesis of Schedulable Real-Time Controllers", *Integrated Computer-Aided Engineering*, 5(3) 261-277, 1998.

[Seto96] D. Seto, et. al., "On Task Schedulability in Real-Time Control Systems", *IEEE RTSS*, December 1996.

[Shin99] K. G. Shin and C. L. Meissner, "Adaptation and Graceful Degradation of Control System Performance by Task Reallocation and Period Adjustment", *EuroMicro Conference on Real-Time Systems*, June 1999.

[Stan98] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, *Deadline Scheduling for Real-Time Systems – EDF and Related Algorithms*, Kluwer Academic Publishers, 1998.

[Stan99] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao, "The Case for Feedback Control Real-Time Scheduling", *EuroMicro Conference on Real-Time Systems*, June 1999.

[Stee99] D. C. Steere, et. al., "A Feedback-driven Proportion Allocator for Real-Rate Scheduling", *Operating Systems Design and Implementation*, Feb 1999.

[Zhao87] W. Zhao, K. Ramamritham and J. A. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints", *IEEE Transactions on Computers* 36(8), 1987.

Using Full System Simulation for Debugging Real-Time Properties of Operating Systems

Lars Albertsson
Computer and Network Architectures Laboratory
Swedish Institute of Computer Science
lalle@sics.se

February 28, 2000

Abstract

This paper proposes the use of instruction level, full system simulation for debugging real-time properties of commodity desktop operating systems. A design of a non-intrusive, predictable environment for debugging real-time operating systems is presented. The simulator based environment allows for interactive debugging of time critical sequences while correctly preserving execution time flow information. It is also shown how performance profiling data provided by an instruction level simulator reveal causes of real-time performance bottlenecks.

1 Introduction

Since long, there has been a rise in the popularity of applications with demand for short response times and throughput guarantees, such as media, entertainment and telecommunication systems. Unlike many real-time applications, these applications also have a large need for throughput performance. In order to satisfy such performance requirements, the applications often run on commodity hardware using operating systems designed for desktop or server use. However, these operating systems are not designed for the needs of real-time applications and fail to meet quality of service requirements. Hence, large research efforts have been put into design of operating systems providing quality of service without compromising performance, and also into appropriate modifications of existing operating systems. However, the complexity of the task and lack of powerful tools for real-time system analysis have prevented effective development of high performance real-time operating systems.

The debugger is one of the most important tools used in computer system programming. It is partially useful for operating system analysis, even though debugger functionality depends on the services provided by the operating system. However, the correctness of real-time operating systems depends on the time elapsed between points in execution. Debuggers generally ignore time flow and often pause program execution. Thus, the use of conventional debuggers

is insufficient for validation of real-time operating systems, as these debuggers lack a notion of temporal correctness.

This paper proposes the use of full system simulation for temporally correct debugging. Full system simulation effectively addresses two major problems in real-time analysis: lack of repeatability and time distortion from intrusion.

Section 2 provides some background regarding full system simulation and the benefits from using it for real-time analysis. In section 3, the design of a temporal debugging environment is described. Section 4 contains a short survey of related work in real-time analysis and simulation. Conclusions and future work are presented in section 5.

2 Background

2.1 Instruction Level Simulation

Many design and research areas benefit from simulation of computer systems. Thus, the level of detail provided by simulators range from models of microprocessor chip logic to coarse models of execution environment including operating system and libraries. A large group of simulators provide a model of a machine at the instruction set level, as it represents a well defined border between hardware and software.

Full system simulators differ from other simulation tools in that they model all the hardware in a system, and only the hardware. Tools that do not provide a complete hardware model need to also model software upon which the application under study is dependent, or disregard from aspects not modelled. Both alternatives compromise accuracy significantly.

In order for a simulator to be useful for real-time research, the model provided need not only be functionally correct, but also temporally correct. Thus, it is necessary to account for all items affecting execution time at the temporal resolution aimed for. For large applications, this includes delays from devices, memory management unit (MMU) and cache system.

Apart from providing a detailed hardware model, an instruction set simulator also provides detailed instrumentation of execution. It records statistics of hardware events, associated with the instruction triggering each event. Examples of events recorded by a simulator are: instruction execution count, cache memory misses and translation lookaside buffer (TLB) misses.

Traditionally, computer system simulators have been used for three purposes:

- **Hardware performance evaluation.** In a simulated environment, it is possible to build a software prototype of a design for a piece of hardware. Measurements of a simulation running applications on this virtual hardware give performance metrics of the design.
- **Software performance evaluation.** For many applications the performance is dependent on memory system behaviour. As the simulator collects statistics on memory system events, it reveals memory related bottlenecks in applications.

- **Debugging of operating systems.** Operating system debugging is inherently difficult, as services necessary for debugger operation are part of the program under study. Moreover, many aspects and problems in operating systems, such as race condition and deadlocks, are prone to intrusion. Therefore, they are hard to reproduce and may be impossible to catch in a debugging session using conventional tools.

It is our aim to apply simulation for debugging another class of programs with special characteristics, namely real-time operating systems and applications. As a simulated environment is completely artificial, many of the restrictions that apply to tools operating in a physical environment are removed. Hence, many of the major problems with building tools for real-time systems are eliminated.

2.2 Predictability

An artificial system has very few unpredictable factors. Thus, a simulated system starting execution in a known state will always execute along the same path. This is very useful, both for experiments and debugging, as it is possible to reproduce a state reached in execution. Thus, a user of a temporal debugger may detect that excessive time has passed at one point in execution, and restart simulation to examine the recently executed routines more carefully. This is similar to the methodology used for debugging the logical correctness of conventional programs. However, as time is part of the state the user wishes to verify, it is crucial that temporal behaviour is preserved between program runs.

2.3 Robustness

In physical systems, measurement of the system often also introduces a modification. This is referred to as the probe effect. As real-time system analysis tend to focus on short periods of time, even small amounts of temporal intrusion affect measurement quality. This limits both the accuracy and amount of measurement in such systems. Furthermore, the time period in focus is too short for a human user to draw conclusions about the correctness of the system. Thus, it is not possible to stop execution in order to analyse current state and step carefully forward, which is a commonly used way of debugging conventional systems.

In a simulated system, the time scale of the system under study is decoupled from the time scale of the system running the simulator. Time distortion due to probe effect is thereby eliminated. This enables the implementation of a temporal debugger, similar to a conventional debugger. The temporal debugger is also able to report time as perceived by the application. Hence, it may be used for validating temporal correctness and finding bugs violating desired real-time behaviour.

3 Design of Temporal Debugger Environment

3.1 Debugging Using a Simulator Backend

In order to provide a convenient user interface to the simulator, it is connected as a backend to a debugger. The simulator supports services normally provided by hardware and operating system. Examples of such services are reading memory, reading registers, setting breakpoints and single stepping. Apart from traditional debugging services, the debugger interface also allows the user to access simulator services. The service most interesting for real-time analysis is the ability to present current time with cycle count granularity. This semantically very small difference enables the user to step through a portion of code, checking for both functional and temporal errors. Figure 1 shows an example output from the debugger library included in the Simics simulator [MDG+98].

```
simscrit> where
#0  [win] 0x45b680 in posix_make_lock (filp = , fl = ,
          l = 0xfffff800001d6440) at locks.c:648
#1  [win] 0x459428 in sys_fcntl (fd = 0 '\000', cmd = 8 '\b',
          arg = 64 '@') at fcntl.c:140
#2  [win] 0x421b74 in sys32_fcntl (fd = 0 '\000', cmd = 8 '\b',
          arg = 64 '@') at sys_sparc32.c:600
#3  [win] 0x40fe74 in linux_sparc_syscall () + 0x34 in
          vmlinux-2.1.126-1.stabs

simscrit> list
643 /* Verify a "struct flock" and copy it to a "struct file_lock"
644  * as a POSIX style lock.
645  */
646 static int posix_make_lock(struct file *filp, struct file_lock *fl,
647                            struct flock *l)
648 {
649     off_t start;
650
651     memset(fl, 0, sizeof(*fl));
652
653     fl->fl_flags = FL_POSIX;

simscrit> print-time
   Number of cycles executed (CPU 0): 630034486
```

Figure 1: Example of the Simics simulator debugging interface. The simulation session shown is paused during boot of Ultrasparc Linux.

The technique used to debug for temporal errors is very similar to a familiar technique for debugging conventional applications. The user starts the simulated system and sets breakpoints at appropriate places. Whenever a breakpoint is triggered, the user verifies that the system is in an expected state. In this case, the state includes current time. The current time stamp is compared with the time stamp of the previous state. If more time than the system allows for has elapsed, a deadline violation is detected. The user makes a note of the point

in execution where the violation occurred, and the simulation is restarted from a known state. When the breakpoint preceding deadline violation is reached, the user proceeds by stepping through subroutines suspected of causing the problem. For each subroutine, the time spent is compared with time usually spent during successful execution. If the time differs significantly from the normal case, a possible cause of the deadline violation is detected. Again, the simulation is restarted in order to step into the violating routine. This procedure is repeated until the presumed problem cause have been thoroughly examined. At this point, the user hopefully has a clue of the cause of violation. Otherwise, he continues with examining other routines whose execution time differs from those of a successful execution.

3.2 Real-Time Performance Debugging with Instrumented Simulation

The methodology described above assumes that the user eventually will discover an erroneous piece of code. This piece of code is assumed to take an undesired execution path, or to set the program in an undesired state, affecting later execution. This assumption is reasonable for conventional applications. However, it is not always true for real-time systems, as time spent in a specific routine may be different between executions, even though program state and input is identical. This is due to the fact that application performance is dependent on memory system behaviour. Thus, execution time for a fraction of a program is also dependent on hardware state, such as cache memory contents.

In order to obtain clues about performance hazards caused by cache effects, the instrumentation provided by the simulator may be consulted. The simulator presents statistics of hardware events, associated with instruction address. In case the time elapsed differs in similar execution paths of a program, these statistics will point out instructions causing cache misses. Statistics for a fraction of the execution may be obtained by subtracting the statistics before and after execution. The values obtained are compared to values from a successful iteration, highlighting the specific hardware events causing delay. The debugger and simulator environment provide a summary of statistics corresponding to source code line, thereby exposing code portions affected by cache effects. Figure 2 shows an example of annotated source code listing.

4 Related Work

In many existing real-time operating systems and environments, only conventional debugging tools are available. These systems may only be used for validating and debugging functional, not temporal, correctness. However, some vendors provide some support for alternative debugging methods.

When developing programs for small, embedded systems, it is common to use an emulator as debugging backend. However, an emulator generally does not model components relevant to execution time, such as caches. Thus, a useful temporal model of execution cannot be provided.

```
(gdb-simics) prof-info
Active profilers, from 'left to right':
Column 1:  Instruction cache misses caused by program line
           ($SIM_SS_INSTR_MISS_WEIGHT = 0.000000)
Column 2:  Cache misses (writes) caused by program line
           ($SIM_SS_WRITE_MISS_WEIGHT = 0.000000)
Column 3:  Cache misses (reads) caused by program line
           ($SIM_SS_READ_MISS_WEIGHT = 0.000000)
Column 4:  TLB misses passed on to Unix emulation
           ($SIM_TLB_MISS_WEIGHT = 0.000000)
Column 5:  Number of (taken) branches *to* the code block
           ($SIM_TO_WEIGHT = 0.000000)
Column 6:  Number of (taken) branches *from* the code block
           ($SIM_FROM_WEIGHT = 0.000000)
Column 7:  Count of instruction execution (based on branch arcs)
           ($SIM_PC_WEIGHT = 0.000000)
Column 8:  Number of addresses from which instructions have
           been fetched ($SIM_INSTR_WEIGHT = 0.000000)

(gdb-simics) list *0x1f2ec
0x1f2ec is in bmexec (kwset.c:560).
555                                 {
556 0 0  66 0 8396    0 20289 3    d = d1[U(tp[-1])], tp += d;
557 0 0 106 0    0    0 16792 2    d = d1[U(tp[-1])], tp += d;
558 0 0   0 0    0 1450 25188 3    if (d == 0)
559                                   goto found;
560 0 0 115 0    0    0 20838 3    d = d1[U(tp[-1])], tp += d;
561 1 0 104 0    0    0 20838 3    d = d1[U(tp[-1])], tp += d;
562 0 0 126 0    0    0 13892 2    d = d1[U(tp[-1])], tp += d;
563 0 0   0 0    0 1109 20838 3    if (d == 0)
564                                   goto found;
```

Figure 2: Source code listing with performance profiling data. Example from Simics tutorial showing the GNU debugger as frontend to Simics in Unix emulation mode.

Some debugging tools work in combination with simulators providing cache modelling, resulting in good execution time prediction. Prior to recent advances in full system simulation, such simulators were not useful for running desktop operating system and large applications. The tools available have either been too incomplete too boot a commodity operating system, or too slow to run an application of realistic size [EST, And94].

Support for non-interactive debugging may be provided by inserting extra code to monitor system events. A trace is generated and sent to a monitoring system. This method is intrusive and has a performance impact. Hence, the amount of monitoring is limited. Furthermore, it is inflexible, as the monitoring process may not query for data not provided. In order to avoid the probe effect, some systems define the trace generation to be part of the production system [BJHL96, NGM98].

The intrusion issue may be avoided using dedicated hardware for bus monitoring. However, requiring extra hardware is generally inconvenient. Furthermore, both the hardware and software monitoring approaches put a great performance demand on the receiving system due to high volumes of generated data. If such a solution is acceptable, it is effective, as it captures almost all activities of interest [TFC90, GT91, Pla84].

The R2D2 debugger [Zen] is based on monitoring of software generated traces. It has been extended with a low priority task in the target system to answer queries from the debugger in case the system is idle. This provides some support for interactive debugging, although not very robust. Furthermore, it does not provide any information when the system is under stress, which usually is the information sought.

Mueller and Whalley [MW94] propose debugging of real-time applications using execution time prediction. The application is executed in a conventional debugger, supported by a cache simulator. Time elapsed is predicted by the simulator and reported during debugging. However, this prediction does not take operating system effects into account, and works best for small programs.

A few simulator research groups have managed to model a complete hardware system with sufficient detail and efficiency to run commodity operating systems with large workloads [DM84], [MDG+98]. The SimOS project [Her98] has made similar achievements, although the simulator presented is not strictly predictable and requires operating system modifications. Due to the accurate timing model provided, these simulators have proven to be effective tools for performance analysis [MM97], [RBDH97].

5 Conclusions and Future Work

It has been demonstrated that full system simulators are very useful tools for temporal debugging of real-time operating systems. As a simulated system operates in an artificial time scale, it may be paused to allow for interactive debugging without disturbing the temporal correctness of the system. Accurate analysis of temporal correctness is possible, as hardware devices affecting execution time are modelled in adequate detail. Furthermore, due to advances in

simulation technology, full system simulators are now capable of running large commodity operating systems.

Instruction level simulation tools have previously been used in small scale only. Therefore, user support is limited, and many tasks could be improved and automated. For example, rather than having the user compare time elapsed in subroutines, it is possible to build a tool for automatic correlation of time spent in subroutines to deadline violation. Such a tool would point out routines frequently triggering deadline misses.

Although the environment described in this paper is limited to debugging of operating systems only, it is possible and desirable to extend its use to user space applications. However, this is non-trivial, as a user space debugger expects to read virtual memory and registers, whereas the simulator provides access to physical devices only. In order to perform translation of debugger accesses, it is necessary to traverse virtual memory management data structures within the operating system.

References

[And94] William Anderson. An overview of Motorola's PowerPC simulator family. *Communications of the ACM*, 37(6):64–69, June 1994.

[BJHL96] Monica Brockmeyer, Farnam Jahanian, Constance Heitmeyer, and Bruce Labaw. An approach to monitoring and assertion-checking of real time specifications in Modechart. In *Proceedings of the Second IEEE Real-Time Technology and Applications Symposium*, Boston, USA, June 1996. IEEE Computer Society.

[DM84] J. K. Doyle and K. I. Mandelberg. A portable PDP-11 simulator. *Software Practice and Experience*, 14(11):1047–1059, November 1984.

[EST] Embedded support tools corporation. www.estc.com.

[GT91] F. Gielen and M. Timmerman. The design of DARTS: A dynamic debugger for multiprocessor real-time applications. In *Proceedings of 1991 IEEE Conference on Real-Time Computer Applications in Nuclear, Particle and Plasma Physics*, pages 153–161, Julich, Germany, June 1991.

[Her98] Stephen Alan Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, February 1998.

[MDG+98] Peter S. Magnusson, Fredrik Dahlgren, Håkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenström, and Bengt Werner. SimICS/sun4m: A Virtual Workstation. In *Proceedings of the 1998 USENIX Annual Technical Conference*, 1998.

[MM97] Johan Montelius and Peter Magnusson. Using SimICS to evaluate the Penny system. In Jan Małuszyński, editor, *Proceedings of the International Symposium on Logic Programming (ILPS-97)*, pages 133–148, Cambridge, October 13–16 1997. MIT Press.

[MW94] Frank Mueller and David B. Whalley. On debugging real-time applications. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[NGM98] Edgar Nett, Martin Gergeleit, and Michael Mock. An adaptive approach to object-oriented real-time computing. In Kristine Kelly, editor, *Proceedings of First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'98)*, pages 342–349, Kyoto, Japan, April 1998. IEEE Computer Society, IEEE Computer Society Press.

[Pla84] Bernhard Plattner. Real-time execution monitoring. *IEEE Transactions on Software Engineering*, SE-10(6):756–764, November 1984.

[RBDH97] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen Alan Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[TFC90] Jeffery J. P. Tsai, Kwang-Ya Fang, and Horng-Yuan Chen. A noninvasive architecture to monitor real-time distributed systems. *Computer*, 23(3):11–23, March 1990.

[Zen] R2D2 debugger, Zentropix. www.zentropix.com.

Empty page.

Modelling of Real-Time Embedded Systems in an Object-Oriented Design Environment with UML

Razvan Jigorea, Sorin Manolache, Petru Eles, Zebo Peng
Linköping University
{g-razji, g-sorma, petel, zebpe}@ida.liu.se

Abstract

This paper explores aspects concerning system-level specification, modelling and simulation of real-time embedded systems. By means of case studies, we investigate how object-oriented methodologies, and in particular UML, support the modelling of industrial scale real-time systems, and how different architectures can be explored by model simulation. We are mainly interested in the problem of system specification as it appears from the prospect of the whole design process. The discussion is illustrated by a large system model from the telecommunications area, the GSM base transceiver station.

1. Introduction

Current real-time embedded systems have to fulfill more and more complex requirements concerning functionality, timing, power consumption, reliability, cost, etc. Typical application areas for such systems are telecommunications, automotive industry, avionics, or industrial process control. Embedded systems are very often implemented on distributed architectures consisting of several programmable processors and application specific integrated circuits (ASICs) [1].

Due to the complexity of such systems, design environments have to be developed in order to assist the designer throughout the whole design process, starting from the system specification, going through the design phase, until the final implementation. In Figure 1 we show a very simplified view of such a design flow. One of the important tasks performed during the design phase is architecture exploration. Several architectures (which differ in number and kind of processors, interconnection structure, number of ASICs, memory structure, etc.) and alternatives for task partitioning are explored in order to find an efficient solution which also satisfies the imposed requirements.

An essential aspect of such an iterative design process is the ability to check the functionality of a certain design alternative and to estimate different parameters characteristic to it like, for example, timing. Although formal verification and static analysis are more and more

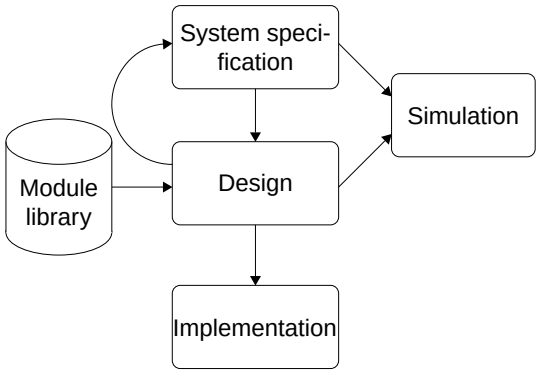


Figure 1. The design cycle of RT systems

used in this context, simulation is still the basic technique which allows to get a feedback concerning the degree to which a specification or design alternative fulfils certain requirements [2].

An important trend in current design methodologies for embedded systems is towards the reuse of pre-designed components as an alternative to the complete synthesis of the whole system. Such an approach is well known for the software design community but has been only recently considered by hardware designers as a practicable strategy in order to cope with increasing system complexity [3]. According to such a methodology, pre-designed programmable cores and ASICs (called intellectual property (IP) components), stored in a module library (Figure 1), are used as building blocks for new system designs.

In order to support such a complex design process, the system specification has to fulfill several requirements. Some of them are well known and have been much discussed in the literature [4, 5, 8]. Such are the support for concurrency, exception handling, timing, hierarchy, or nondeterminism. In this paper, however, we are mainly interested in the problem of system specification as it appears from the prospect of the whole design process. Thus, requirements like support for IP-based design, separation of control and dataflow, or facilities for system simulation and architecture exploration are of particular interest. Our main focus is on how object-oriented

methodologies, and in particular UML, support such requirements and how they can be used for the specification and design of complex embedded systems. The discussion is based on the experience gained from the modelling of a large application in the telecommunications area, namely the GSM Base Transceiver Station (BTS). This is a representative example of a large reactive embedded system and is therefore well suited to illustrate some design problems and possible solutions. Another example, an intelligent traffic lights controller, has been used for a set of experiments concerning the architecture exploration aspects.

In the following section we identify some of the main problems to be discussed. Section 3 describes the basic functionality of a GSM BTS, our main case study. Section 4 presents solutions to the problems identified in section 2, in the particular context of UML. In section 5 we focus on aspects related to architecture exploration. The last section presents our conclusions.

2. Modelling of Real-Time Embedded Systems

Embedded systems, and in general all real-time (RT) systems, have characteristics that make their design very complex. They usually exhibit a reactive behaviour, meaning that they respond in a well-defined manner to input stimuli. Moreover, the response is time-constrained, therefore it must occur within a more or less specific time window. Safe critical embedded systems must be also robust, showing correct behaviour even in unexpected circumstances.

Concerning the specification and design of RT embedded systems, several particular issues have been identified and discussed in the literature [4, 5, 8]. Such are state-transition oriented behaviour, inherent parallelism, timing aspects, exception handling, environment dependency, and non-functional characteristics (performance, safety, reliability, power consumption etc.).

As mentioned in the previous section, in this paper we are interested in particular aspects of system specification, which are related to the whole design process. Some of these problems have been identified as result of our work related to the modelling and design of RT systems in the telecommunications area. They are briefly introduced here and will be further discussed in the following sections:

- *Data and control flow separation.* The model should be organized as a set of functional and control layers. A functional layer performs operations on the incoming data, while a control layer coordinates clusters of units situated in a lower layer. The amount of control implemented in a certain layer should not exceed the minimum needed in order to coordinate the operation of the lower layers.
- *Generic functional and control units.* Many functional or control blocks share certain basic characteristics, differing just in some aspects. Identifying such generic units and specifying them as superclasses is essential in order to manage complexity during the modelling process.
- *Support for architecture exploration.* Both the specification methodology and the related design environment should provide the support for exploration of alternative implementation architectures.
- *Timeliness verification.* The model must be checked not only for functional correctness, but also for temporal correctness. We therefore need mechanisms for specification and checking of timing aspects.
- *Multiple abstraction levels.* The specification of a RT system often consists of blocks which are described at very different levels of abstraction. In the initial specification some blocks can be treated as black boxes and be specified at a high level of abstraction, while other blocks are specified in more detail. As result of subsequent design steps, some of the blocks are further refined while other are left unchanged. Despite such an unbalance in the abstraction level, a clear separation between specification of interfaces and that of the internal behaviour, allows the whole system to be kept coherent and executable.
- *Support for reuse.* As mentioned earlier, the reuse of predefined components (IP-based design) is of extreme importance in managing the complexity of the design process. Thus, specifications and models should be built with a high-degree of reusability as an essential goal. At the same time, the specification and design methodology has to support the reuse of existing components.

In the following section we introduce a typical telecommunication application of high complexity which in the subsequent sections will be used in order to illustrate the discussion. All the problems highlighted above will be analysed, along with proposed solutions, in the context of an OO design environment based on the UML.

3. The Base Transceiver Station (BTS)

The BTS is one of the devices in the canonical architecture of the Global System for Mobile Communication (GSM), as defined by the GSM standardisation group [7]. Figure 2 shows the place of the BTS in the context of the GSM architecture. The radio interface connects the BTS with the mobile stations (MS). A terrestrial link, the Abis interface, connects the BTS with the Base Station Controller (BSC). The BSC is further connected with the Network Switching Subsystem (NSS). The BTS comprises radio transmission and reception devices, and also all the signal processing specific to the radio interface. The radio interface is characterized by a high bit error rate. To counter this, complex modulation/demodulation algorithms are deployed in the BTS and computation intensive channel encoding/decoding methods are applied. Because of the large range of services GSM

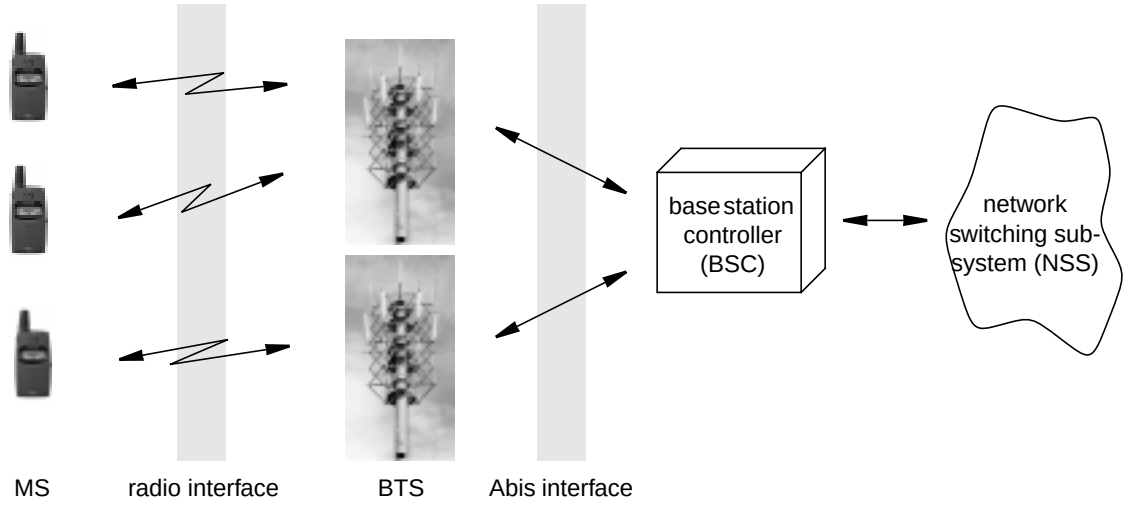


Figure 2. The BTS in the context of the GSM architecture

offers, the number of these channel encoding/decoding algorithms is also very large. The BTS also performs the so called characterization of the radio interface, i.e. it measures interference level and bit error rates, processes measurements of received power, and reports the results to the NSS. In order to adapt more users, a time division multiple access (TDMA) scheme is used. This allows a BTS to manage several communication channels multiplexed both in time and frequency. The BTS allocates/deallocates and changes the channel mode at the request of the NSS. Signalling protocols between the BTS and the mobile stations as well as between the BTS and the infrastructure are used in order to make such a channel management possible.

4. The BTS Model

In this section we first present the general structure of the BTS model. Next, solutions to the problems identified in section 2 are discussed. The underlying specification and design strategy is based on an OO methodology. The discussion here is in the particular context of the UML [5, 10]. General aspects concerning OO methodologies and the basics of the UML have been much discussed in the literature [8, 9, 10], and therefore are not mentioned in this paper.

4.1. General Description of the Model Architecture

In Figure 3 we show the structure of the BTS model. Our model has been organized on four layers. The bottom layer consists of purely functional processing units (called functional units — FU). The control tasks to be performed by the BTS are distributed among the other three layers. The responsibilities of the functional units are related to channel encoding/decoding and to data interleaving/deinterleaving.

The encoding/decoding algorithms are the following:

- CRC encoding/decoding;
- convolutional encoding/decoding in its various variants (punctured or not);
- tailing;
- fire codes (similar to CRC codes).

Subsequent interleaving/deinterleaving schemes are applied in order to build radio-bursts from data-blocks and to assemble data-blocks out of the incoming radio-bursts. Functional units in the model are specialized for a certain encoding/decoding or interleaving/deinterleaving task.

The first layer of control, above the functional layer, consists of the baseband controllers (BC). Their responsibility is to assure a correct sequence of operations (performed by the FUs) for each channel mode. BCs are specialized for the control of channels in a certain channel mode. By channel mode we understand a set of properties that characterize the channel. Such properties are:

- semantics of carried data (speech, data, signalling);
- gross data rate, which classifies channels in full and half rate channels;
- nett data rate;
- whether the channel is dedicated to a particular user at a moment in time or it is a broadcast (common) channel;
- whether the channel is a duplex one or not.

The next level of control consists of the transmitter-receivers (TRX). A TRX controls a set of BCs. It emits or receives continuously, but on a single frequency at a given moment. From the signalling point of view, each TRX corresponds to a signalling link. A TRX manages traffic and signalling for eight (due to the TDMA scheme) physical channels.

The TRXs use the services of the LAPD (Link Access Protocol for the “D” Channel) and LAPDm (Link Access Protocol for the “Dm” Channel) protocol interpreters [6]. These modules receive from the TRX the signalling bursts/blocks and are responsible for assembling the frames, maintenance of the link, and notification of the TRX regarding the received message. The two LAPD blocks

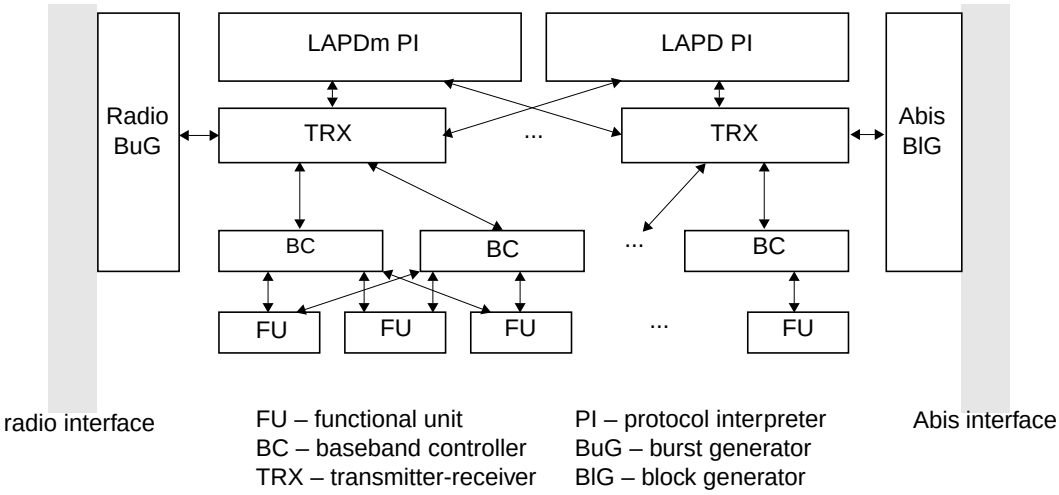


Figure 3. Structure of the GSM BTS model

represent the third and highest layer of control.

The Radio Burst Generator (BuG) is the module that models the radio subassembly of the BTS. It generates bursts which are sent to the TRX. The BuG has to be aware of the time division scheme, as demodulation is performed differently for different channels.

The Abis Block Generator (BlG) has a similar role to the Radio BuG. It generates blocks, simulating the time division scheme deployed on the terrestrial link. These blocks are then forwarded to the TRXs.

The BTS model has been specified and simulated with the UML using the iLogix Rhapsody environment [12].

4.2. Deployment of Control

Functional units can be, in principle, reused over many generations of a telecommunication application and often they can be also included in different designs. Such different designs and successive generations of the same design usually differ in their control aspects (e.g. the protocols) but not in the basic functions performed. The same philosophy applies to control units, as well. Protocols are organized on different layers and from one product to another only particular layers are modified (upgraded) while other layers are reused. Therefore, in order to support reusability of functional and control units, a minimal amount of control has to be deployed to any processing unit in the model. By doing this, the potential of reusing the modelling units increases significantly. Figure 4 shows the UML statechart for one of the functional units which performs a part of the channel encoding. The unit, a CRC encoder, is unaware of the channel whose processing chain it belongs to. Thus, it could be used as a building block for both speech encoding and for fire codes, as well as for any other design in which CRC encoding is performed. The unit has a very simple interface: after a processing delay (procD) has elapsed from the moment it receives a processing command (evCRCEncCmd) from a controlling entity, it will notify the controlling entity about completion of the processing and offer the transformed data (evCmdCompleted(result)).

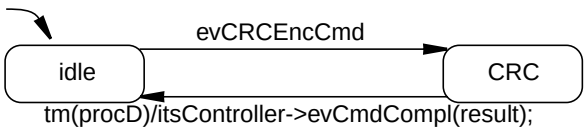


Figure 4. Statechart for CRC processing unit

The same principle of minimum amount of control deployment was adopted for the controlling units. A baseband controller, for example, manages functional units according to the mode of the channel it controls. At the same time, it is completely unaware of the existence of the other channels and it does not know anything about the time-slots structure of the access scheme (a statechart of a baseband controller is shown in Figure 8).

The TRX, on the other hand, is in charge of the eight channels it manages. It is the only unit which has to be aware of the timing scheme in order to identify a channel correctly and to activate the right baseband controller (the statechart of the TRX is depicted in Figure 9).

4.3. Separation of Control and Data Flows

Another aspect which improves the reusability of the various elements of the model is the separation of data and control flows. Object-oriented modelling particularly suits this requirement. There are objects which model entities on the data path, and objects which model controlling units. Due to the inherent loose coupling between objects, the separation of control and data flow is easy to achieve. This aspect is well highlighted in Figure 3. Functional units are operating on the data path, while no control functions are embedded within them. On the other hand, baseband controllers as well as TRXs do not perform any processing on the data flow. They just coordinate lower level entities (other controllers or functional units).

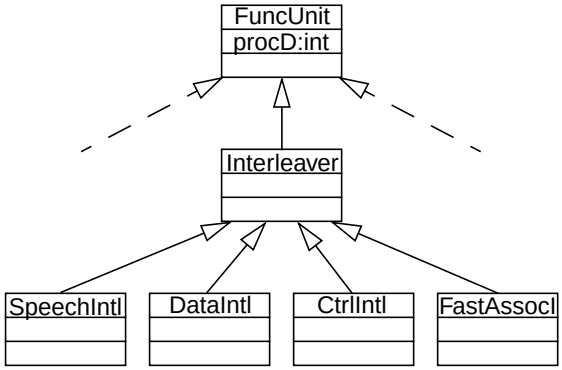


Figure 5. Generic functional units

4.4. Uniform Interfaces and Generic Units

Various types of low level control or processing units are treated uniformly by the upper layers. This highly simplifies the modelling process. Such an uniformity is achieved by means of the inheritance relationship.

Figure 5 illustrates a class hierarchy for functional units. Particular interleavers are derived from a generic interleaver which again is a specialization of a generic functional unit. For the upper, control layers, every functional unit appears as this generic unit. A similar strategy also applies to control units.

Figure 6 shows the class diagram of a generic downlink baseband controller. The generic controller aggregates generic functional units of the classes Interleaver (see also Figure 5), ConvEnc, and ParityEnc. A specific BC aggregates specialized FUs, depending on the particular channel mode it controls. A data downlink controller, for example, is an instantiation of the generic class DownLkCtrl, and aggregates an instantiation of the class DataConvEnc and of the class DataIntl (see also Figure 5). The corresponding class diagram is shown in Figure 7.

4.5. Timing Aspects

Specification of timing aspects is mandatory in order to verify timeliness and to perform architecture exploration.

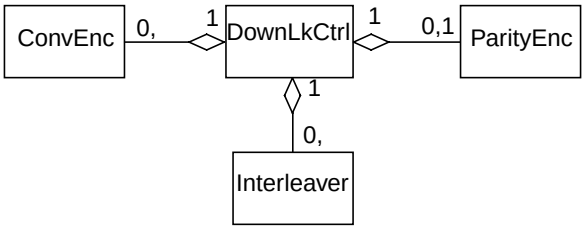


Figure 6. Structure of a downlink controller

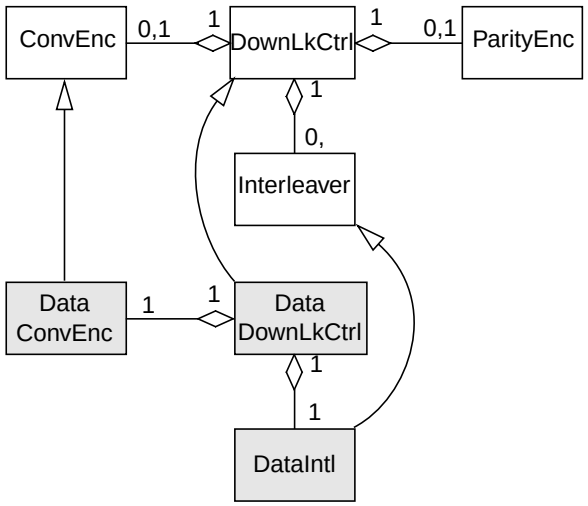


Figure 7. Specialization of a downlink controller

Modelling of such timing aspects comprises specification of deadlines and execution delays.

Execution delays are modelled as time-triggered state transitions (see Figure 4). The parameter which characterizes the execution delay on a certain functional unit is specified as an attribute of that unit. During architecture exploration, this delay depends on the particular processor to which the unit is assigned for execution. This delay has to be estimated [11] and the resulting value is assigned to the unit. The delay attribute is specified in the superclass of all functional units (procD in Figure 5).

The controlling units are in charge of coordinating the functionality of lower lever units. They are not characterized by a processing delay in the sense the functional units are. However, the control units have to

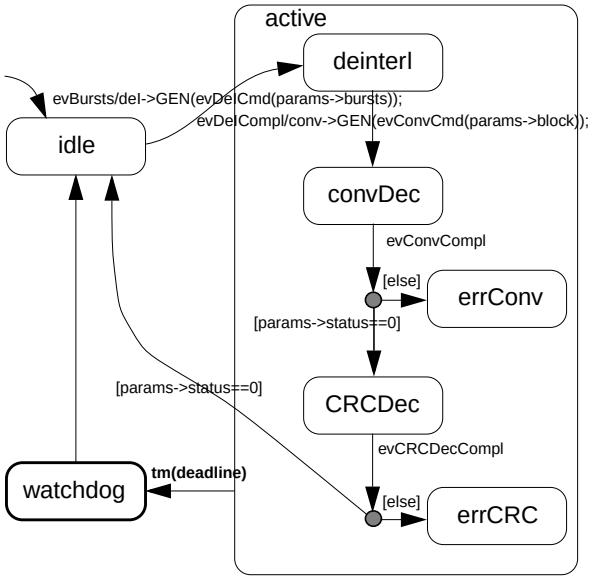


Figure 8. Statechart of a baseband controller

monitor whether certain predefined time intervals have passed and to take certain decision if a deadline has been reached. Deadline is modelled by introducing an additional state, which is entered after a certain time interval has elapsed. In Figure 8 we illustrate this mechanism for a baseband controller. If the execution delay of the processing chain exceeds the predefined value of *deadline*, the watchdog state is entered. The TRX interprets this as an exceptional situation.

4.6. Concurrency Issues

Usually, telecommunication devices, and the BTS makes no exception, perform a set of well specified operations on multiple data flows. Identification of this parallelism is important in the analysis phase because it significantly influences the object structure of the model. In the UML, parallelism can be expressed in two ways. First, all the objects are considered to be parallel entities. Synchronization is easily achieved by means of message exchanges. Second, an entity which has been modelled as a single object can exhibit itself a concurrent behaviour. In this case the statechart corresponding to the object is specified as a set of orthogonal (concurrent) components. Such a case is shown in Figure 9, where the statechart of a TRX is depicted. The TRX has to handle both uplink and downlink traffic, as well as signalling information addressed to it. Those activities are independent to each other and can be performed concurrently.

4.7. Multiple Abstraction Levels

Our main focus was set on the specification and design of the digital components of the BTS. However, in order to perform simulation and architecture exploration, the whole functionality of the BTS had to be modelled. Thus, the radio subassembly was simply modelled as a radio burst generator, at a very high abstraction level. The strong encapsulation, typical to the object-oriented approach, allows for an uniform treatment of entities specified at different abstraction levels. They permit us to concentrate on the refinement of that part of the model we are mainly interested in.

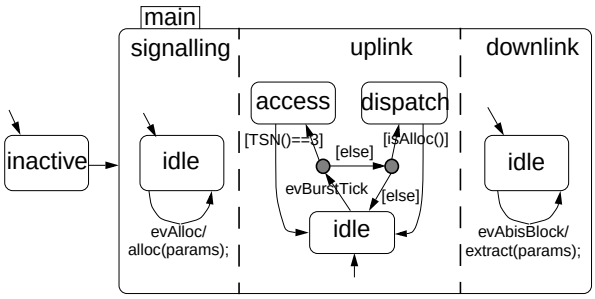


Figure 9. Statechart of a TRX

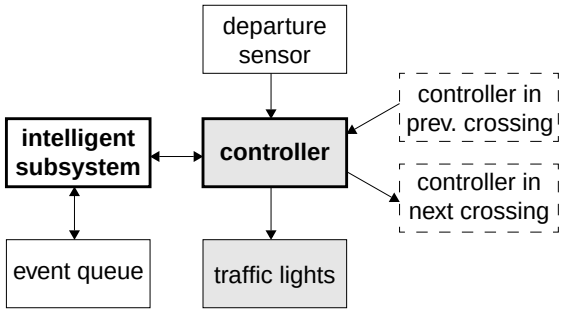


Figure 10. Intelligent traffic lights controller

5. Architecture Exploration

In order to complete our goal in exploring the way UML suits the real-time systems design cycle, we imagined a simpler case study, which allows to easily demonstrate how architecture exploration can be performed. The example we have chosen is an intelligent, adaptive traffic lights controller (see Figure 10).

The controller was designed for a typical two road crossing (one main road, crossed by a secondary road), including pedestrian sideways. It is connected with the controllers in the previous and next crossing on the main road. Beyond the crossing, on the main road, there is a departure sensor, used for detecting if a car left the current crossing, heading to the next crossing on the main road. This is needed in order to perform statistical analysis on the time needed for a car to get from one crossing to another. According to the statistical data, the intelligent subsystem will adjust the synchronization and traffic lights timing in order to have an optimal “green wave” on the main road,

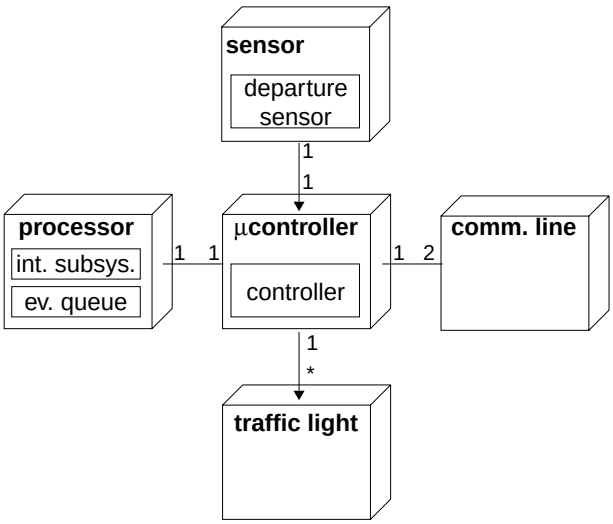


Figure 11. Deployment diagram

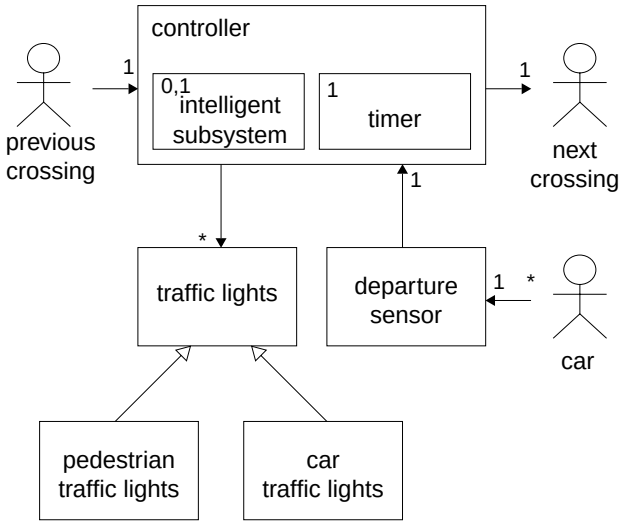


Figure 12. Object model diagram

which adapts to the current traffic condition.

In Figure 10, the shaded blocks represent the core of the traffic lights controller. We considered that the controller is physically mapped on a microcontroller, and the intelligent subsystem (which performs statistical analysis and adjusts the timing accordingly) is mapped on a processor. This mapping is illustrated by the UML deployment diagram in Figure 11.

A high-level object model diagram of the intelligent traffic lights controller is presented in Figure 12. The model was conceived in a way which allows code generation and simulation with or without the intelligent subsystem.

We assumed that the average waiting time for a car is the performance parameter of interest. Thus, we explored how different processors and communication lines affect this performance parameter. The processing efficiency (PE) is a parameter of the processor which runs the intelligent subsystem, and the communication efficiency (CE) reflects the performance of the communication lines connecting our controller to the two controllers in the adjacent crossings.

We define PE and CE as follows:

$$PE = \frac{v_{min}}{v}$$

$$CE = \frac{\delta_{min}}{\delta}$$

where v is the execution time for the intelligent subsystem on the currently considered processor, δ is the communication delay for an instance of data transfer on the currently considered infrastructure for the communication lines, v_{min} is the value of v corresponding to the fastest processor in the module library, and δ_{min} is the value of δ corresponding to the fastest communication link in the module library.

The experimental results are presented in Table 1. We run the same scenario (i.e., input vectors), first for the traffic lights controller without the intelligent subsystem, and second for the traffic lights controller with intelligent subsystem, departure sensor and communication lines. For the case without an intelligent subsystem, the controller works by granting access to the tracks according to a round robin strategy. For both settings we explored several values for PE and CE. As it can be seen from Table 1, the intelligent subsystem produces a significant increase in performance. The quality of the system is degraded for lower performances of the processor and/or communication line.

Table 1. Simulation results

Intelligent subsystem	Average waiting time for a car (seconds)			
	PE=0.9 CE=0.9	PE=0.9 CE=0.6	PE=0.6 CE=0.9	PE=0.6 CE=0.6
No	11.56	11.56	11.56	11.56
Yes	1.66	5.82	9.09	11.31

6. Conclusions

We discussed several issues concerning the modelling of complex embedded real-time applications using an OO methodology with the UML.

We have concentrated on particular issues which are characteristic to the modelling of large telecommunication applications. Using the particular example of a GSM BTS we showed how the UML based methodology allows the proper layering of a model, the separation of control and data flows, as well as the definition and combination of generic units. The main objective is to facilitate complexity management during the modelling phase. At the same time, reusability (IP-based design) and architecture exploration are supported.

Using a smaller example, which allows for a more detailed discussion, we also presented an example of architecture exploration and gave some experimental results.

References

[1] R. Ernst, *"Codesign of Embedded Systems: Status and Trends"*, IEEE Design & Test of Computers, April-June, 1998, pp. 45-54.

[2] J. Axelsson, *"Holistic Object-Oriented Modelling of Distributed Automotive Real-Time Control Applications"*, Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 1999, pp. 85-92.

[3] M. Keating, P. Bricaud, *"Reuse Methodology Manual for System-on-a-Chip Designs"*, Kluwer Academic Publishers, 1998.
[4] A. Sarkar, R. Waxman, J. Cohoon, *"Specification-Modelling Methodologies for Reactive-System Design"*, in "High-Level System Modelling: Specification Languages", eds. J. M. Bergé et al., Kluwer Academic Publishers, 1995, pp. 1-34.
[5] B. Douglass, "*Doing Hard Time*", Addison-Wesley, 1999.
[6] M. Mouly, M. Pautet, *"The GSM System for Mobile Communication"*, Palaiseau, 1992.
[7] *"GSM 03.02"*, European Telecommunications Standards Institute (ETSI),
http://www.etsi.org.
[8] B. Selic, G. Gullekson, P. Ward, *"Real-Time Object-Oriented Modeling"*, John Wiley & Sons, 1994.
[9] G. Booch, "*Object Oriented Design With Applications"*, The Benjamin/Cummings Publishing, 1991.
[10] G. Booch, J. Rumbaugh, I. Jacobson, *"The Unified Modeling Language User Guide"*, Addison-Wesley, 1999.
[11] J. Gong, D. Gajski, S. Bakshi, *"Software Estimation Using A Generic Processor Model"*, Proceedings of the European Design and Test Conference, 1995, pp. 498-502.
[12] *"Rhapsody Reference Guide, Release 2.1"*, iLogix, 1999.

Real-Time System Constraints: Where do They Come From and Where do They Go?

Cecilia Ekelin and Jan Jonsson

Department of Computer Engineering
Chalmers University of Technology
S–412 96 Göteborg, Sweden
{*cekelin,janjo*}*@ce.chalmers.se*

Abstract

Within the real-time community a great deal of research has been, and is being, conducted on scheduling. There exist a diversity of different scheduling algorithms as well as constraints that should be satisfied by a feasible schedule. What is not commonly discussed though is how the constraints relate to the requirements on the actual real-time system. In fact, many constraints are mere artifacts that cannot be traced to the system design. This often results in an over-constrained and infeasible scheduling problem. To avoid this we are conducting a survey on industrial real-time system requirements. The aim is to give a better understanding of how to derive adequate constraints. Furthermore, we believe that by considering constraints imposed by industrial requirements such as product cost or hardware limitations, the time complexity of schedule generation can be greatly reduced. In this paper we present some early results from our survey. This includes a description of real-time system requirements and how they relate to commonly used real-time constraints.

Keywords: *real-time constraints, real-time scheduling, system design, constraint programming, complexity reduction*

1 Introduction

Scheduling in distributed real-time systems involves assigning n tasks to m processors such that all constraints on the tasks are satisfied. A common instance of the problem is to find the optimal schedule according to some objective. In real-time literature, several scheduling algorithms that deal with specific real-time constraints (periods, release times etc.) have been proposed. However, in most cases there is no clear connection between the theoretical scheduling problem and the practical design of the real-time system being scheduled. This makes it difficult to verify the correctness of the constraints with respect to the requirements in the system design. Are the scheduling constraints due to limitations in hardware, cost or system behavior? Or are they more or less intelligent guesses? Previous research [1, 2] have argued that the latter often is the case. Adding constraints manually is an iterative process because the *ad hoc* assignment often leads to an over-constrained and infeasible system. Apparently, the reason for fabricating constraints is the lack of expressive power in current tools and the lack of scheduling tools that handle complex constraints. The facility to consider factors such as cost and hardware performance is also lacking in most proposed scheduling algorithms. However, for industrial real-time system developers, these factors are extremely important in order to provide cost-effective solutions. In particular since the trend in real-time system development is moving towards open systems [3], that is, the system designer is restricted to use the hardware components available on the market. Designing real-time systems with these prerequisites also affects the definition of an optimal schedule. The optimality of a schedule might not only concern the behavior of the tasks but also the cost of the resulting hardware configuration.

In an attempt to aid industry in this design process, we have begun work that attempts to identify requirements like cost and hardware limitations, as well as behavioral aspects of industrial real-time systems. This paper reports some results from this work and also demonstrates how these requirements can be translated into constraints. Apart from increasing the expressive power we also believe that, by introducing additional constraints, the scheduling search space will be reduced without making the system over-constrained. In particular, we are interested in investigating how this hypothesis can aid in reducing the search complexity for optimal schedule generation.

The rest of this paper is organized as follows. Section 2 describes some common real-time system requirements. Section 3 lists some real-time constraints typically found in literature. Section 4 presents our allocation and scheduling

approach. In Section 5 related work is discussed and Section 6 outlines our future work.

2 Identification of requirements

When trying to identify the requirements on a real-time system from an industrial perspective it is necessary to work in close relation to industry. We are therefore performing our investigation in cooperation with Mecel AB in Göteborg, Sweden. Mecel AB develops tools and distributed real-time systems for the automotive industry. However, as an initial attempt we have limited the survey to requirements and constraint constructs found in literature. So far, we have found that the requirements can be divided into three groups which are further described below.

2.1 Behavioral requirements

A model of a real-time system includes tasks, that determine the behavior of the system, and resources, that are used by tasks. The behavior of the system imposes restrictions on its implementation which affects the scheduling constraints. Typical behavioral requirements could be task execution order or task allocation.

2.2 Temporal requirements

Most development tools for real-time systems focus on modeling the system behavior but lack the expressive power to handle temporal requirements [4]. This is a serious disadvantage since real-time systems have a temporal behavior as well as an operational. The temporal behavior depends on the environment that the system interacts with. That is why the requirements rarely are stated for each task individually but rather on chains of tasks constituting a specific function. For instance, a function like cruise control might have a required update rate which is set independent of the number of tasks that are part of the function. Most scheduling constraints stem from temporal requirements.

2.3 Cost requirements

Apart from mere system requirements, it is in the interest of industry that development of real-time systems is made cost-effective. That is why development using off-the-shelf hardware components has become an interesting alternative. Instead of making hardware for the software, the software has to be made for the hardware. This puts additional strain on the system design since the flexibility is decreased. Even though less flexibility is likely to reduce scheduling complexity, additional hardware constraints have been introduced.

3 Derivation of constraints

In the introduction it was stated that most real-time system constraints are artifacts. The reason for this is the inability of current tools to translate system requirements into system constraints in an adequate way. In the translation the requirements are usually divided into a number of simple constraints. Unfortunately, the *ad hoc* methods used to derive the constraints often leads to an over-constrained and infeasible system. In our approach we try to keep the constraints as close as possible to the original requirements. Below we discuss how a number of constraints, that frequently appear in real-time literature, relates to typical system requirements.

3.1 Task model

We use the following variables to denote task properties for a task i.

- S_i actual start time
- E_i worst-case execution time
- R_i release time, i.e., the earliest allowed start time
- D_i deadline, i.e., the latest allowed finish time
- P_i period, i.e., the rate that the task is executed with
- B_i blocking time, i.e., the time the task is spent preempted by other tasks
- N_i execution node, i.e., the processor the task is scheduled to execute on

Also, c is used to denote an arbitrary constant where applicable.

3.2 Preemption

Apart from the derivation of constraints, it has to be decided whether or not tasks are allowed to preempt each other. By allowing preemption it might be possible to find schedules for designs that are otherwise infeasible. Let n denote the number of tasks. Then the time a task i is blocked due to preemption is defined as:

$$B_i = \sum_{j=1}^{n} B_j :$$

$$\{i \neq j, S_i \leq S_j, S_j \leq S_i + B_i + E_i, N_i = N_j\}$$

Context switching or task invocation time is usually assumed to be small enough to be neglected or included in the task's worst-case execution time. However, there might be some real-world problems where this approximation is not desirable. For example, in preemptive scheduling where it is not known beforehand how many times a task will be preempted. Most likely, the context switch occurring at a preemption will be significant since the state of the preempted task must be stored away or retrieved. The context switch time c can easily be incorporated into our previous equation as:

$$B_i = \sum_{j=1}^{n} (B_j + c) :$$

$$\{i \neq j, S_i \leq S_j, S_j \leq S_i + B_i + E_i, N_i = N_j\}$$

3.3 Absolute timing constraints

Absolute timing constraints are direct restrictions on the temporal behavior of a task.

Execution times express the worst-case execution times of the tasks. They depend on the hardware the task is scheduled to execute on. This is of importance in a distributed system where the execution time for a task may differ between the nodes. Let $E_{i,N}$ be the execution time of task i on node N. Then this constraint is defined as:

$$E_i = E_{i,N} : \{N = N_i\}$$

Deadlines correspond to the responsiveness required by the system. Usually, there only exists an explicit deadline for a chain of tasks (end-to-end deadline) and not for each individual task. A common approach is to split the end-to-end deadline into shorter deadlines which are assigned to each task [5]. This operation may put an unnecessary strain on the task set. To avoid this, we assign the end-to-end deadline as the deadline for all tasks in the chain. Let D be the end-to-end deadline. Let T_D be all tasks restricted by D. Then the deadline constraint is defined as:

$$\forall i \in T_D | D_i = D$$

To ensure that no deadlines are missed, the following constraint is needed:

$$S_i + B_i + E_i \leq D_i$$

This assumes that the precedence relations have been modeled as described later in this paper.

Release times are mostly connected to periodic processes where they restrict the start time of task invocations. This is explained further below. Many scheduling algorithms impose release times on the tasks as a way to obtain mutual exclusion between tasks that access the same resource. The drawback of this technique is the same as for the deadline assignment mentioned previously, namely the risk of over-constraining the task set. To ensure that a task is not started before its release time, the following constraint is needed:

$$S_i \geq R_i$$

Periods express how often a task should be executed. This is linked to how accurate the system needs to be for interaction with its environment. Periods may restrict the deadlines since the deadline of a task often is less than its period ($D_i \leq P_i$). Even if that is not true ($D_i > P_i$), the k:th invocation of a task must be completed before invocation $k + 1$. A way to handle scheduling of periodic tasks is to find the least common multiple of all periods and then schedule each invocation as an individual task. The different scenarios impose the following constraints:

$$R_{i,k} = (k-1) \cdot P_i : \{k > 0, D_i \leq P_i\}$$

$$R_{i,k} = (k-1) \cdot D_i : \{k > 0, D_i > P_i\}$$

Separation constraints [1] express an interval of values that the period of a task should belong to. In contrast to deadlines, periods might be limited by a maximum and/or minimum period value which ensures that the required functionality is obtained. Let l and u be the minimum and maximum value respectively. Then this constraint is defined as:

$$l \leq P_i \leq u$$

3.4 Relative timing constraints

Relative timing constraints are also known as local constraints. That is, they express how two tasks relate to each other. Some common constructs are explained below. Other constructs are possible [6, 7] and can be expressed in a similar manner.

Precedence constraints deal with the order the tasks should be executed in. Such constraints are often derived from the system design because the behavior of a system is modeled as sequences of operations. If task i should precede task j the constraint is defined as:

$$S_i + B_i + E_i \leq S_j$$

Distance constraints are a stronger variant of precedence constraints. In addition to constraining their order of execution, they also express the minimum distance in time between two tasks i and j. A reason for this could be limitations in the processing speed of the environment that the tasks interact with. It could also be derived from delays in the communication hardware between two communicating tasks. If the distance is c, the constraint is defined as:

$$S_i + B_i + E_i + c \leq S_j$$

Freshness constraints [1] are the opposite of distance constraints. They express the maximum distance in time between two (consecutive) tasks i and j. This is used to express that a task uses some result produced by another task. If the tasks are too far apart in time, the result have become inaccurate when it is about to be used. This constraint is typically found in database applications. If the freshness is c, the constraint is defined as:

$$S_i + B_i + E_i \leq S_j \leq S_i + B_i + E_i + c$$

Correlation constraints [1] are related to freshness constraints in that they describe the maximum difference between the finish times of two (concurrent) tasks i and j. This appears when a third task uses the results from the two tasks and a too large a time-skew between the parameters makes the task's computation faulty. If the correlation is *c*, the constraint is defined as:

$$|(S_i + B_i + E_i) - (S_j + B_j + E_j)| \leq c$$

Harmonicity constraints [1] relate the periods of two communicating tasks. It is desired that the period of the consumer task i is exactly divisible by the period of the producer task j. If this is the case it is much easier for the consumer to keep track of the received messages since they always will arrive with the same interval. This constraint is defined as:

$$P_i = c \cdot P_j : \{c > 0\}$$

3.5 Resources

To be able to perform their operations, the tasks may require to use specific hardware components. The performance or functionality of the available components is limited by how expensive the hardware is allowed to be.

Locality constraints concern the allocation of tasks to different nodes. A task might require a specific unit for its execution that is not present on all nodes. We express this by simply removing impossible nodes from the task's node set. This is a *hard-coded* constraint. Let *N* be an impossible node. This is then defined as:

$$N_i \neq N$$

It could also be the case that communicating tasks (i and j) should be located on the same node. A reason for this could be that the operating system requires that all tasks within a process are located on the same node. Another reason could be to lower the cost for the communication network. This is known as a *clustering* constraint. The constraint is defined as:

$$N_i = N_j$$

Locality constrains are often implementation recommendations made by the designer, and less frequently given in the specification.

Communication between tasks requires a communication media. The time to send a message depends on performance offered by the communication channel. Messages sent between tasks must themselves be scheduled on the communication network which affects the scheduling of the tasks. Message scheduling is non-preemptive. There exist predefined constraint constructs for this kind of scheduling, e.g., the *serialized* constraint [8].

Devices are resources other than processors used by the tasks during execution. It might be the case that only a limited number of tasks can access a device at the same time, e.g., shared memory or bus communication. This means ensuring that the amount of the resource used, by concurrent tasks, never at any point in time is exceeding a given limit. This can be expressed using the *cumulative* constraint [8].

3.6 ORing

ORing constraints [6] are not that common but can be used to express alternative operations. For instance, to explicitly state that a task i must not preempt another task j (an EXCLUDES relation) means than task i must execute before or after task j. Any types of constraints can be ORed. For *c* constraints this is simply defined as:

$$constraint_1 \vee constraint_2 \vee ... \vee constraint_c$$

4 Allocation and scheduling

Most published real-time scheduling algorithms are "hard-coded" with respect to the constraints, task properties and resources they can handle. Even-though a specific method in most cases outperforms a generic, the former soon becomes quite complex and it becomes difficult to include new features. That is why we have chosen the constraint programming approach.

The tool we have selected for our experiments is SICStus Prolog [8] and its associated constraint solver for finite domains [9]. The identified real-time system requirements can be expressed using primitive (low-level) constraints which are combined to express more complex (high-level) constraints. The advantage of this generality is that it is easy to introduce new high-level constraints as long as they can be composed by a combination of low-level constraints. The translation of the constraints into code is pretty straightforward. An interesting feature is the possibility to guide the search for feasible schedules by varying the search parameters. By using "correct" parameters the search time can be significantly reduced. We hope that it will be possible to automatically detect which heuristic that is best suited for a specific problem instance.

5 Related work

A great motivation for this paper was the analysis of the origins of timing constraints presented by Ramamritham in [2]. The gap between the system design process and the implementation has been discussed in [10]. We continue this work by proposing guidelines for the derivation of constraints from system requirements. In [1], a technique to automatically generate periods and deadlines for each task from end-to-end deadlines, is proposed. A similar technique to generate intervals instead of single numbers for the start times has been presented in [7]. Both techniques are based on constraint solving using Fourier-Motzkin variable elimination. The ideas and techniques for constraint solving

are already contained in the concept of constraint programming systems.

The use of constraint programming for scheduling has been investigated in the areas of operations research and artificial intelligence. However, the scheduling aspects in these areas are somewhat different from the ones concerning real-time systems. For instance, the optimization objective is different as well as the constraints to be considered. So far, we have only found one paper that addresses the use of constraint programming for scheduling of real-time systems [11]. However, this work only addresses the problem of finding a feasible schedule, while we are more interested in finding the optimal and most cost-effective schedule.

6 Future work

As previously mentioned, we are currently working together with the industry on a survey on real-time system requirements. A remaining issue of importance is the definition of an optimal schedule. To this end, we plan to implement a scheduling framework in SICStus Prolog which will incorporate the described real-time constraints, including a new notion of optimality. At a later stage we will investigate how the constraints and search heuristics can be used to reduce the complexity of the scheduling process. Previous work have shown that this is a viable approach [12, 13].

References

[1] R. Gerber, S. Hong, and M. Saksena, "Guaranteeing Real-Time Requirements with Resource-Based Calibration of Periodic Processes," *IEEE Trans. on Software Engineering*, vol. 21, no. 7, pp. 579–592, July 1995.

[2] K. Ramamritham, "Where do Time Constraints Come From and Where do They Go?," *International Journal of Database Management*, vol. 7, no. 2, pp. 4–10, 1996.

[3] J. A. Stankovic, "Real-Time and Embedded Systems," *http://www-ccs.cs.umass.edu/sdcr/*.

[4] G. Bucci, M. Campanai, and P. Nesi, "Tools for Specifying Real-Time Systems," *Real-Time Systems*, vol. 8, no. 2/3, pp. 117–172, Mar. 1995.

[5] M. Di Natale and J. A. Stankovic, "Dynamic End-to-End Guarantees in Distributed Real-Time Systems," *Proc. of the IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, Dec. 7–9, 1994, pp. 216–227.

[6] T. M. Chung and H. G. Dietz, "Language Constructs and Transformation for Hard Real-Time Systems," *Proc. of ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, Canada, June 1995, pp. 4–10.

[7] R. Gerber, W. Pugh, and M. Saksena, "Parametric Dispatching of Hard Real-Time Tasks," *IEEE Trans. on Computers*, vol. 44, no. 3, pp. 471–479, Mar. 1995.

[8] Intelligent Systems Laboratory, *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, 1995, http://www.sics.se/sicstus/.

[9] M. Carlsson, G. Ottosson, and B. Carlson, "An Open-Ended Finite Domain Constraint Solver," *Proc. of the Int'l Symposium on Programming Languages: Implementations, Logics, and Programs*, H. Glaser et al., Eds., Southampton, UK, Sept. 3–5, 1997, vol. 1292 of *Lecture Notes in Computer Science*, pp. 191–206, Springer Verlag.

[10] M. Saksena, "Real-Time System Design: A Temporal Perspective," *Proc. of IEEE Canadian Conference on Electrical and Computer Engineering*, Waterloo, May 1998.

[11] M. Schild and J. Würtz, "Off-Line Scheduling of a Real-Time System," *Proc. of CP97 Workshop on Industrial Constraint-Directed Scheduling*, Schloss Hagenberg, Austria, Oct. 1997.

[12] J. Jonsson, "Effective Complexity Reduction for Optimal Scheduling of Distributed Real-Time Applications," *Proc. of the IEEE Int'l Conf. on Distributed Computing Systems*, Austin, Texas, May 31 –June 5, 1999, pp. 360–369.

[13] I. Ahmad and Y.-K. Kwok, "Optimal and Near-Optimal Allocation of Precedence-Constrained Tasks to Parallel Processors: Defying the High Complexity Using Effective Search Techniques," *Proc. of the Int'l Conf. on Parallel Processing*, Minneapolis, Minnesota, Aug. 10–14, 1998, pp. 424–431.

Empty page.

A Real-Time Animator for Hybrid Systems (Extended Abstract)

Tobias Amnell, Alexandre David
Wang Yi
Department of Computer Systems, Uppsala University
{adavid, tobiasa, yi} @docs.uu.se

Abstract

In this paper, we present a real time animator for dynamical systems that can be modeled as hybrid automata i.e. standard finite automata extended with differential equations. We describe its semantic foundation and its implementation in Java and C using CVODE, a software package for solving ordinary differential equations. We show how the animator is interfaced with the UPPAAL tool to demonstrate the real time behavior of dynamical systems under the control of discrete components described as timed automata.

1 Introduction

UPPAAL is a software tool for modeling, simulation and verification of real time systems that can be described as timed automata. In recent years, it has been applied in a number of case studies [4, 5, 6, 7, 8], which demonstrates the potential application areas of the tool. It suites best the class of systems that contain only discrete components with real time clocks. But it can not handle hybrid systems, which has been a serious restriction on many industrial applications. This work is to extend the UPPAAL tool with features for modeling and simulation of hybrid systems.

A hybrid system is a dynamical system that may contain both discrete and continuous components whose behavior follows physical laws [1], e.g. process control and automotive systems. In this paper, we shall adopt hybrid automata as a basic model for such systems. A hybrid automaton is a finite automaton extended with differential equations assigned to control nodes, describing the physical laws. Timed automata [2] can be seen as special class of hybrid automata with the equation $\dot{x} = 1$ for all clocks x. We shall present an operational semantics for hybrid automata with dense time and its discrete version for a given time granularity. The discrete semantics of a hybrid system shall be considered as an approximation of the continuous behavior of the system, corresponding to sampling in control theory.

We have developed a real time animator for hybrid systems based on the discrete semantics. It can be used to simulate the dynamical behavior of a hybrid system in a real time manner. The animator implements the discrete semantics for a given automaton and sampling period, using the differential equation solver CVODE. Currently the engine of the animator has been implemented in Java and C using CVODE. We are aiming at a graphical user interface for editing and showing moving graphical objects and plotting curves. The graphical objects act on the screen according to physical laws described as differential equations and synchronize with controllers described as timed automata in UPPAAL.

The rest of the paper is organized as follows: In the next section we describe the notion of hybrid automata, the syntax and operational semantics. Section 3 is devoted to implementation details of the animator. Section 4 concludes the paper.

2 Hybrid Systems

A hybrid automaton is a finite automata extended with differential equations describing the dynamical behavior of the physical components.

2.1 Syntax

Let X be a set of real-valued variables X ranged over by x, y, z etc including a time variable t.

We use $\dot{x}$ to denote the derivative (rate) of x with respects to the time variable t. Note that in general $\dot{x}$ may be a function over X; but $\dot{t} = 1$. We use $\dot{X}$ to stand for the set of differential equations in the form $\dot{x} = f(X)$ where f is a function over X.

Assume a set of predicates over the values of X; for example, $2^x + 1 \leq 10$ is such a predicate. We use $\mathcal{G}$ ranged over by g, h etc to denote the set of boolean combinations of the predicates, called *guards*.

To manipulate variables, we use concurrent assignments in the form: $x_1 := f_1(X) \ldots x_n := f_n(X)$ which takes the current values of the variables X as parameters for f_i and updates all x_i's with $f_i(X)$'s simultaneously. We use Γ to stand for the set of concurrent assignments.

We shall study networks of hybrid automata in which component automata synchronize with each other via complementary actions. Let $\mathcal{A}$ be a set of action names. We use $\mathcal{A}ct = \{\ a? \mid \alpha \in \mathcal{A}\ \} \cup \{\ a! \mid \alpha \in \mathcal{A}\ \} \cup \{\ \tau\ \}$ to denote the set of actions that processes can perform to synchronize with each other, where τ is a distinct symbol representing internal actions.

A hybrid automaton over X, $\dot{X}$, $\mathcal{G}$, $\mathcal{A}ct$ and Γ is a tuple $\langle L, E, I, T, L_0, X_0 \rangle$ where

- L is a finite set of names standing for control nodes.
- E is the equation assignment function: $E : L \to 2^{\dot{X}}$.
- I is the invariant assignment function: $I : L \to \mathcal{G}$ which for each node l, assigns an invariant condition $I(l)$.
- T is the transition relation: $T \subseteq L \times (\mathcal{G} \times \mathcal{A}ct \times \Gamma) \times L$. We denote $(l, g, \alpha, \gamma, l')$ by $l \xrightarrow{g,\alpha,\gamma} l'$.
- $l_0 \in L$ is the initial node.
- X_0 is the initial variable assignment.

To study networks of automata, we introduce a CCS-like parallel composition operator. Assume that $A_1, ..., A_n$ are automata. We use $\overline{A}$ to denote their parallel composition. The intuitive meaning of $\overline{A}$ is similar to the CCS parallel composition of $A_1, ..., A_n$ with *all* actions being restricted, that is, $\overline{A} = (A_1|...|A_n)\backslash\mathcal{A}ct$. Thus only synchronization between the components A_i is possible. We call $\overline{A}$ a *network of automata*. We simply view $\overline{A}$ as a vector and use A_i to denote its ith component.

Example In figure 1 we give a simple example hybrid automaton which describes a bouncing ball and a touch sensitive floor. The left automaton defines three variables, x, the horizontal distance from the starting point, the height y and the speed upwards u. Initially the x-speed is 1, the ball is at 20 m height and the gravitational constant is 9.8. The variables will change according to their equations until the transition becomes enabled when $y <= 0$. The middle automaton is a model of a sensor that will issue a signal when the ball hits the floor. The right automaton is on the UPPAAL side. It synchronizes with the sensor signal and resets a clock

z. If the intervals between signals are longer than 5 time units it will return to the initial location, but the first interval that is shorter will lead to the location *low_bounces*.

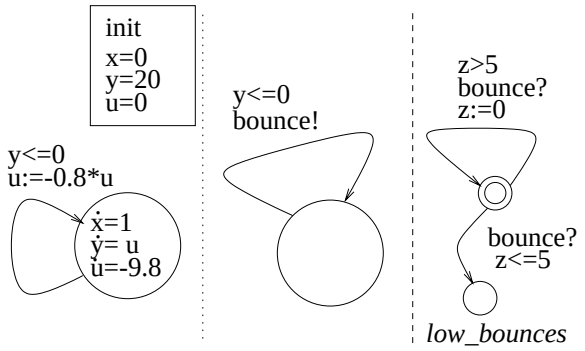


Figure 1: Bouncing ball with touch sensitive floor and control program.

2.2 Semantics

To develop a formal semantics for hybrid automata we shall use variable assignments. A *variable assignment* is a mapping which maps variables X to the reals. For a variable assignment σ and a delay d (a positive real), $\sigma + d$ denotes the variable assignment such that

$$(\sigma + d)(x) = \sigma(x) + \int_{\Delta} \dot{x} dt$$

For a concurrent assignment γ, we use $\gamma[\sigma]$ to denote the variable assignment σ' with $\sigma'(x) = Vale\sigma$ whenever $(x := e) \in \gamma$ and $\sigma'(x') = \sigma(x')$ otherwise, where $Vale\sigma$ denotes the value of e in σ. Given a guard $g \in \mathcal{G}$ and a variable assignment σ, $g(\sigma)$ is a boolean value describing whether g is satisfied by σ or not.

A *node vector* $\bar{l}$ of a network $\overline{A}$ is a vector of nodes where l_i is a location of A_i. We write $\bar{l}[l_i'/l_i]$ to denote the vector where the ith element l_i of $\bar{l}$ is replaced by l_i'.

A *state* of a network $\overline{A}$ is a configuration $(\bar{l}, \sigma)$ where $\bar{l}$ is a node vector of $\overline{A}$ and σ is a variable assignment.

The *semantics of a network* of automata $\overline{A}$ is given in terms of a labelled transition system with the set of states being the configurations. The transition relation is defined by the following three rules:

- $(\bar{l}, \sigma) \rightsquigarrow (\bar{l}[l_i'/l_i], \gamma_i[\sigma])$ if $l_i \xrightarrow{g_i\ \gamma_i} l_i'$ and $g_i(\sigma)$ for some l_i, g_i, γ_i.
- $(\bar{l}, \sigma) \rightsquigarrow (\bar{l}[l_i'/l_i, l_j'/l_j], (\gamma_j \cup \gamma_i)[\sigma])$ if $l_i \xrightarrow{g_i\ a!\ \gamma_i} l_i'$, $l_j \xrightarrow{g_j\ a?\ \gamma_j} l_j'$, $g_i(\sigma)$, $g_j(\sigma)$, and $i \neq j$, for some l_i, l_j, g_i, g_j, α, γ_i, γ_j.
- $(\bar{l}, \sigma) \rightsquigarrow (\bar{l}, \sigma + \Delta)$ if $I(\bar{l})(\sigma)$ and $I(\bar{l})(\sigma + \Delta)$.

where $I(\bar{l}) = \bigwedge_i I(l_i)$.

2.3 Tick semantics

The operational semantics above defines how an automaton will behave at every real-valued time point with arbitrarily fine precision. In fact, it describes all the possible runnings of a hybrid automata, that are the sequences of alternating transitions between delays and actions in the form:

$s_0 \overset{\Delta_0}{\rightsquigarrow} (\bar{l}_0, \sigma_0 + \Delta_0) \overset{\tau}{\rightsquigarrow} (\bar{l}_1, \sigma_1) \overset{\Delta_1}{\rightsquigarrow} (\bar{l}_1, \sigma_1 + \Delta_1) \overset{\tau}{\rightsquigarrow} (\bar{l}_2, \sigma_2) \ldots (\bar{l}_i, \sigma_i) \overset{\Delta_i}{\rightsquigarrow} (\bar{l}_i, \sigma_i + \Delta_i) \overset{\tau}{\rightsquigarrow} (\bar{l}_{i+1}, \sigma_{i+1})$

In practice, the "sampling" technique is often used to analyze a system. Instead of examining the system at *every* time point, only certain *typical* time points are chosen to capture or approximate the full system behavior. Based on this idea, we shall adopt a time–step semantics called δ–semantics relativized by the granularity δ, which describes how a hybrid system shall behave in every δ time units. In practical applications, the time granularity δ is chosen according to the nature of the differential equations involved. In a manner similar to sampling of measured signals the sampling interval should be short for rapidly changing functions. To achieve finer precision, we can choose a smaller granularity.

We identify the δ–transitions as follows:

- $(\bar{l}, \sigma) \overset{\delta}{\mapsto} (\bar{l}, \sigma + \delta)$ if $(\bar{l}, \sigma) \overset{\delta}{\rightsquigarrow} (\bar{l}, \sigma + \delta)$ and

- $(\bar{l}, \sigma) \stackrel{\alpha}{\mapsto} (\bar{l}', \sigma')$ if $(\bar{l}, \sigma) \stackrel{\alpha}{\rightsquigarrow} (\bar{l}', \sigma')$

The "sampled" runnings of a hybrid automaton will be in the form:
$(\bar{l}_0, \sigma_0) \stackrel{\delta}{\mapsto} (\bar{l}_0, \sigma_0 + \delta) \stackrel{\tau}{\mapsto} (\bar{l}_1, \sigma_1) \stackrel{\delta}{\mapsto} (\bar{l}_1, \sigma_1 + \delta) \stackrel{\tau}{\mapsto} (\bar{l}_2, \sigma_2) \ldots (\bar{l}_i, \sigma_i) \stackrel{\delta}{\mapsto} (\bar{l}_i, \sigma_i + \delta) \stackrel{\tau}{\mapsto} (\bar{l}_{i+1}, \sigma_{i+1})$

In the following section, we shall present a real time animator based on the δ-semantics. For a given hybrid automaton, the animator works as an interpretor computing the δ-transitions step by step but in real time, using CVODE, a differential equations solver.

3 Implementation

Our goal is to extend the UPPAAL tool to deal with hybrid systems. The UPPAAL GUI is written in Java and the differential equation solver that we have adopted, CVODE, is written in C. This gives the natural architecture of the animator: the animator itself with the objects is written in Java and the engine of the animator in C, connected through the Java native interface (JNI). The two main layers of the implementation are the animation system and the CVODE layers.

3.1 The Animation System Layer

The system to be modeled is defined as a collection of *objects*. Each object is described by a hybrid automaton with its corresponding variables. Every state of the hybrid automaton has a set of equations and transitions. The equations, conditions (guards) and assignments are given as logical/arithmetic expressions with ordinary mathematical functions such as sine, cosine ..., and also user defined functions.

The evaluation of the object equations, conditions and assignments is written in C. Each animator object, i.e. a hybrid process, is associated with one UPPAAL process that is an abstraction of the hybrid part and a bridge to UPPAAL. The abstraction is modeled as a stub process that performs the same synchronizations as the hybrid counterpart. This choice of implementation is motivated by the desire to model-check the rest of the UPPAAL processes as a closed system. Figure 2 shows the association of animator objects with UPPAAL automata.

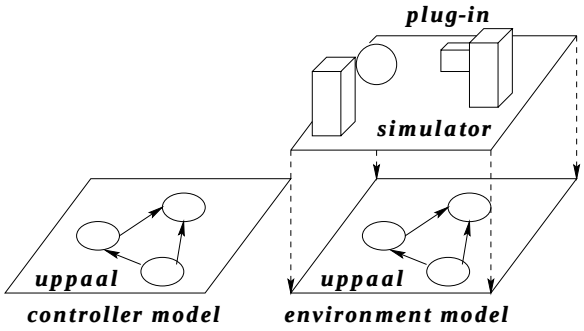


Figure 2: Association between animator objects and UPPAAL automata.

3.2 The CVODE Layer

At the heart of the animator we have used the CVODE [9] solver for ordinary differential equations (ODE's). This is a freely available ODE solver written in C, but based on two older solvers in Fortran.

The mathematical formulation of an initial value ODE problem is

$$\dot{x} = f(t, x),\ x(t_0) = x_0,\ x \in \boldsymbol{R}^N. \quad (1)$$

Note that the derivative is only first order. Problems containing higher order differential equations can be transformed to a system of first order. When using CVODE one gets a numerical solution to (1) as discrete values x_n at time points t_n.

CVODE provides several different methods for solving ODE's, suitable for different types of problem. But since we aim at general usage of the animator engine we cannot assume any certain properties of the system to solve. Therefore we only use the full dense solver and assume that the system is well behaved (non-stiff in numerical analysis terminology). This will give neither the most memory efficient nor the best solution, but the most general.

We use one CVODE solver for the whole system. This is set up and started with new initial values at the beginning of each delay transition. The calculations are performed stepwise, one "tick" (δ–transition) at a time. After each tick all the conditions of the current state are checked, if any is evaluated to true one must be taken, continuing in the same state is not possible. If an assignment on the transition changes a variable the solver must be reinitialized before the calculations can continue.

It is worth pointing out that the tick length δ is independent of the internal step size used by the ODE solver, the solver will automatically choose an appropriate step size according to the function calculated and acceptable local error of the computation. From the solvers point of view the tick intervals can be seen as observation or sampling points.

After each tick the system variables are returned to the Java side of the animator where they are used either to update a graph or as an input to move graphical objects.

Example, continued In figure 3 a plot of the system described in figure 1 is shown. The plot shows the height and the distance of the bouncing ball. Not shown in the figure is that the touch sensitive floor will create a signal every time the ball hits the floor, and that the system will continue running until the time between bounces is less than 1 second.

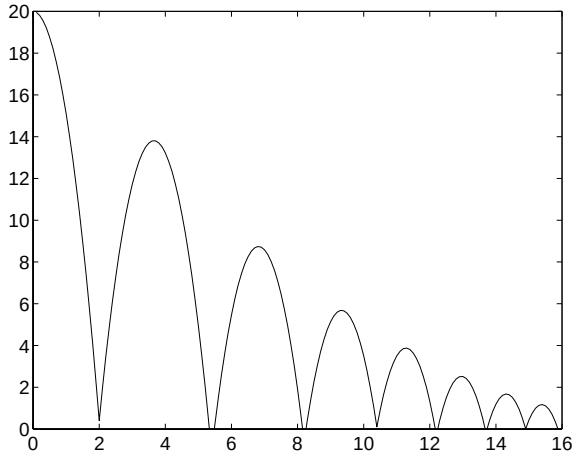


Figure 3: Bouncing ball on touch sensitive floor that continues until bounces are shorter than 1 second

4 Conclusion

We have presented a real time animator for hybrid automata. For a given hybrid automaton modeling a dynamical system and a given time granularity representing sampling frequency, the animator demonstrates a possible running of the system in real time, which is a sequence of sampled transitions. The animator has been implemented in Java and C using CVODE, a software package for solving differential equations. As future work, we aim at a graphical user interface for editing and showing moving graphical objects and plotting curves. The graphical objects act on the screen according to the differential equations and synchronize with controllers described as timed automata in UPPAAL.

References

[1] Thomas A. Henzinger. *The Theory of Hybrid Automata.* Proceedings of the 11th Annual IEEE Symposium on Logic on Computer Science (LICS 96), pp. 278-292.

[2] R. Alur and D.L. Dill. *A Theory of Timed Automata.* Theoretical Computer Science, 125:183-235,1994.

[3] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL*: a Tool-Suite for Automatic Verification of Real-time Systems.* In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, Hybrid Systems III, Lecture Notes in Computer Science 1066, pages 232-243. Springer-Verlag 1996.

[4] Kåre J. Kristoffersen, Kim G. Larsen, Paul Pettersson and Carsten Weise. *Experimental Batch Plant - VHS Case Study 1 Using Timed Automata and UPPAAL.* Deliverable of EPRIT-LTR Project 26270 VHS (Verification of Hybird Systems).

[5] Magnus Lindahl, Paul Pettersson and Wang Yi *Formal Design and Analysis of a Gear Controller.* In Proceedings of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. Gulbenkian Foundation, Lisbon, Portugal, 31 March - 2 April, 1998. LNCS 1384, pages 281-297, Bernhard Steffen (Ed.).

[6] Henrik Lönn and Paul Pettersson. *Formal Verification of a TDMA Protocol Start-Up Mechanism.* In Proceedings of 1997 IEEE Pacific Rim International Symposium on Fault-Tolerant Systems, pages 235-242. Taipei, Taiwan, 15-16 December, 1997.

[7] Klaus Havelund, Arne Skou, Kim G. Larsen and Kristian Lund. *Formal Modelling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL.* In Proceedings of the 18th IEEE Real-Time Systems Symposium, pages 2-13. San Francisco, California, USA, 3-5 December 1997.

[8] P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. *The bounded retransmission protocol must be on time!* In Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. Enschede, The Netherlands, April 1997. LNCS 1217, pages 416-431.

[9] S. Cohen and A. Hindmarsh. *CVODE, a Stiff/Nonstiff ODE Solver in C.* Computers in Physics, 10(2):138-43, March-April 1996.

Deadline Dependent Coding - A Framework for Wireless Real-Time Communication

Elisabeth Uhlemann and Per-Arne Wiberg
Halmstad University, SE-301 18 Halmstad,
http://www.hh.se, {elisabeth.uhlemann, per-arne.wiberg}@ide.hh.se

***Abstract* -- A framework for real-time traffic over a wireless channel is proposed in this paper. The deadline dependent coding (DDC) scheme makes use of modern tools from the information theory. A combination of hybrid ARQ and soft decision decoding is used to maximise the probability of delivering information before a deadline. The strategy of DDC is to combine different coding and decoding methods with ARQ in order to fulfil the application requirements. These requirements are formulated as two QoS parameters: deadline (t_{DL}) and probability for delivery before this deadline (P_d). The application can negotiate these parameters with the DDC protocol, thus creating a flexible scheme. It is still possible to make probabilistic guarantees on the communication mechanism.**

***Index Terms* -- real-time, communication**

I. INTRODUCTION

There is a tremendous development in the wireless communication field. New technologies and products reach the market at increasing rate. Some parts of this evolution look very promising for industrial applications.
Cabling always caused the industry problems in terms of costs and feasibility in general. Some applications even demand a wireless access in order to function.
There have been some implementations of wireless communication systems for industrial use. The problem of guaranteeing real-time delivery has been solved in quite an ad hoc manner. The consequence is that it is hard or impossible to make any judgement of the security aspects.
In this paper we will describe a framework of working with such unsecure transfer mechanisms as radio channels.
A real-time communication channel is characterised by the fact that it is equally important to deliver in time, as it is to deliver the correct data. In literature the real-time field often mentions two classes of real-time systems; hard and soft real-time systems. A hard real-time system is a system where you cannot tolerate any late delivery of a result. In a soft system on the other hand you can tolerate a late delivery but with a degraded value to the system.
The uncertainty concerning security in wireless transmission has prevented it from being used in hard real-time systems.
If we have control over the communication channel security, wireless communication can replace cables in a vast number of applications. A class of industrial applications is measurement and control on rotating parts. Here it is obvious why wireless communication is necessary.
Another application is communication to and from different kind of vehicles in factory automation situations. In control systems in general, cabling is something we would like to replace.
In all these cases it is not clear if it is a hard – or a soft real-time system. In general we would like to turn away from the notion of hard and soft real-time systems as if there only exist these two classes. In our framework we introduce a probabilistic view of the communication mechanism. This means it is no longer meaningful to talk about hard or soft real-time systems. We rather talk about deadline for delivery and the probability to succeed in delivering before this deadline. We therefore introduce two parameters: deadline (t_{DL}) and probability for delivery (P_d). These two parameters can be used for setting bounds on the communication system, or stating requirements.
The values of these parameters can be derived by means of classical safety analysis methods. In the process of making requirement analysis of the system t_{DL} and P_d is one result. If this shall be a powerful tool in designing real-time systems including wireless communication links, there must be mechanism in the protocol for guaranteeing the parameters.
If the two parameters are viewed as quality of service parameters (QoS) of a communication mechanism it means that a protocol layer can negotiate the parameters with an underlying layer. If we give the protocol these properties the communication system can guarantee the delivery based on the parameters or reject the transmission request. If the request is rejected the application have a possibility through an exemption.
What we strive for is to maximise the probability that the communication system will be able to accept the transmission request. Aspects that must be considered is not only static and dynamic channel properties like throughput and error rate, but also multi-user interference.
In communication systems one can use several means for increasing the probability of success. Different diversities are such means. Time diversity is one and frequency diversity is another. Both these tools are used in our approach to guarantee t_{DL} and P_d.
The main idea is that the t_{DL} and P_d parameters are mapped onto a retransmission protocol. The retransmission protocol

plays the role of maximising the success rate and still being able to reject requests that cannot be handled.
The protocol performs a series of transmissions with different amount of redundant information, thus the closer to the deadline the more information the receiver will have in the decoding process. We denote this series of transmissions a *transmission suite*.
In a multi-user radio system it is important to keep the interference, from all acting nodes, down. This is one of the reasons for trying to get the information across with as little redundant information as possible, thus the lower amount of redundant bits in the beginning of the transmission suite. Closer to deadline it is more important to get the information across than not interfering with others; so more redundant information is needed.
In section II the background and the main ideas behind the DDC scheme are described. A previous work by the authors [1] evaluating a DDC scheme based on hard decision decoding is discussed.
In this work a decoder based on a soft decision mechanism will be evaluated. Different decoding techniques and algorithms using soft decision decoding are examined in section III. The application of these algorithms to the DDC scheme is discussed.

II. Background

A node communicating in real-time must not only remit the correct information, but also deliver it within a certain limit of time, the deadline. As a missed deadline can have disastrous consequences, a real-time system must leave some sort of performance guarantee, a quality of service. When a hard real-time system is designed it is often assumed that the deadline must be met with unity probability. This is a situation that cannot be achieved with any physical system. As the consequences for missing a deadline is disastrous in a hard real-time system we want to quantify the probability for this event.
The application designer states the probability as a requirement. The communication mechanism then transforms this request into actions in the protocol, using models of the communication channel and different communication methods.
We base our system on the Code Division Multiple Access (CDMA) method, the Deadline Dependent Coding (DDC) scheme, and linear block codes as explained below.

A. Media Access Method

Code Division Multiple Access (CDMA) uses orthogonal codes in order to separate the different users and thereby providing multiple access.
The advantage of CDMA in our case is that it rejects multipath reception to some extent. Multipath reception is a typical phenomenon in radio communication and is the result of the message bouncing of obstacles and having to travel different paths of different length. Thus, multiple copies of the same message are received within a certain time window. The radio channel also experiences fading, usually with a Rayleigh distribution. By spreading the messages to be sent, by means of orthogonal codes, they use a wider frequency spectrum, thus reducing the fading damages on a particular frequency band. CDMA also gives instant access to the media, an important issue in a real-time system, where time is a valuable resource. Less administration is required when a new user is added as opposed to TDMA for example. Naturally here is a limit to the number of simultaneous users in a CDMA system, but it still degenerates gracefully.

B. Deadline Dependent Coding

In [1] the Deadline Dependent Coding (DDC) scheme was used. This coding scheme strives to meet the demands on deadline (t_{DL}) and probability of a correct delivery before the deadline (P_d), and at the same time keeping the activity factor of the network as low as possible. This is realised by using hybrid ARQ [2], which means that both an error correcting code and a retransmission scheme is used as opposed to just an error detecting code and retransmissions. The message to be transmitted is coded differently depending on the deadline and the requested delivery probability. In the beginning of the time window a high rate coded message, i.e. few redundant bits, is transmitted, see Fig. 1. If no acknowledge signal is received, the procedure will be repeated until a certain retransmission deadline is reached. When this occurs the receiver performs a bit-wise majority voting procedure in order to restore the correct message. If the receiver still fails to decode the result from the majority voting procedure a low rate coded message, i.e. a large number of redundant bits, will be sent as a last attempt to get a correct delivery before deadline.

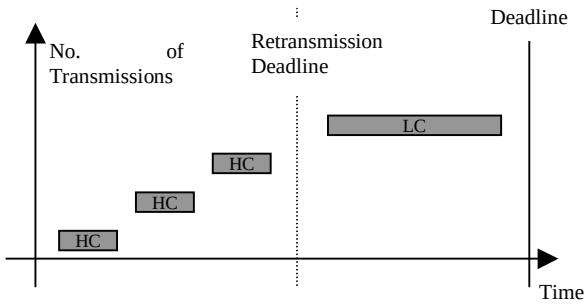


Fig. 1. The Deadline Dependent Coding (DDC) Transmission Suite.

C. Linear Block Codes

The error correcting codes used in the DDC scheme are Reed-Solomon codes. These codes are especially good at correcting burst errors, which is a common error type in a wireless system due to the fading.
Block codes introduce controlled amounts of redundancy into a transmitted data stream, providing the receiver with the ability to detect and correct errors caused by noise on

the communication channel. The data stream is divided into blocks and redundancy is added to each block independently, thus producing a larger block.
Each code can be described by a generator matrix, which is multiplied with the data bits to produce a code word, and a parity check matrix, which when multiplied with the received code word is zero if no errors are present.
The data source generates blocks of k messages symbols taking values from the Galois Field, GF(q) [2]. Each message block is then encoded, generating a code word of n code word symbols, each symbol taking values from GF(q). Consequently, the total amount of redundancy, r, introduced is $r = n - k$. The code word symbols are then sent to the modulator. We have used a BPSK modulation technique and as Reed-Solomon codes are nonbinary, the q–ary code word symbols must be translated into binary channel symbols before transmission. The receiver recovers the channel symbols from the demodulator and passes them along to the translator for conversion back to q-ary code word symbols. The received code word symbols are sent in n-symbol blocks to the error control decoder. As the noisy channel corrupts the modulated carrier, the symbol block arriving at the error control decoder contains errors.

D. Hard Decision Decoding

To decode the Reed-Solomon codes a bounded distance decoder was used in [1].
A bounded distance decoder will select the code word closest in *hamming* distance [2] to the received code word if and only if that distance is less than the bounded distance. If there is no code word within the bounded distance a decoder failure is declared. The hamming distance between two code words is the number of positions in which they differ. A bounded distance decoder will make an error whenever the received code word is within the bounded distance of another code word than that which was sent, see *Fig. 2*.

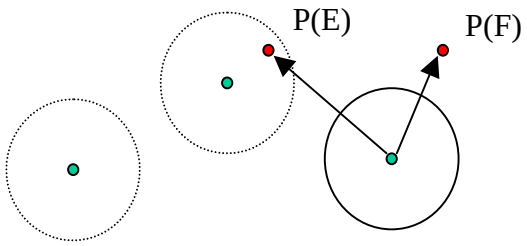


Fig. 2. Bounded Distance Decoding. In the centre of each sphere is a code word. P(F) is the probability of decoder failure and P(E) the probability of decoder error.

The hamming distance implies hard decision decoding, i.e. the decoder first determine whether it is a binary one or a zero in a particular position of the received code word by means of quantization and thereafter calculates the code word distance based on the binary representation of the code word. Decisions based directly on the unquantized demodulator output, so-called soft decisions decoding, requires a more complex decoder that can handle analog inputs, but offers a significant performance improvement over hard decision decoding.

III. Soft Decision Decoding of Linear Block Codes

Soft decision decoding is now introduced in the DDC scheme in order to enhance the performance with respect to probability of delivering before the deadline.
We have developed a toolbox with different decoding methods which can used in the DDC framework. These methods are based on a trellis representation of block code. The trellis concept is explained in section III.A. Two different decoding algorithms, the Viterbi and the BCJR algorithm, are explained in section III.B and III.C respectively. Both these decoding algorithms are used in conjunction with the trellis. The Viterbi algorithm has a lower complexity and is thus faster. The BCJR algorithm on the other hand can be used in a turbo decoding scheme, explained in section III.D, resulting in a faster and more powerful decoding strategy. Turbo decoding, which is an iterative decoding strategy, used in conjunction with ARQ gives new possibilities for adapting the transmission suite and the coding to a particular t_{DL} and P_d request.
Which strategy to use in which situation to make the most of the remaining time is what is to be examined and simulated in the near future.

A. Trellis Representation of Linear Block Codes

For many years the algebraic decoding of linear block codes was considered the only possible. Convolutional codes on the other hand could be decoded using soft decision decoding with the Viterbi algorithm on a trellis. The fact that block codes also can be interpreted as trellises was first described in [3] and later in [4].
Consider a binary (n,k) block code C over $GF(q)$ with parity check matrix, H. A message of k information bits is shifted into the encoder and is encoded into a code word of n code bits [5]. The code words in C are all the n-tuples $\mathbf{x}$ with elements from $GF(q)$, such that $\mathbf{Hx}=\mathbf{0}$. One n-bit code word is shifted out on the channel each interval t. This procedure can be viewed as a finite state machine, FSM, and C can be represented by an n-section trellis diagram of length t. A trellis is a directed graph consisting of n+1 states and branches, see Fig. 3.
A branch in the i-th section of the trellis connects a state $S_{i-1} \in \sum_{i-1}(C)$ to a state $S_i \in \sum_i(C)$ and is labeled with a code bit u_i that represents the encoder output at the bit interval from time (i-1) to time i. A branch represents a state transition. The branches diverging from the same state have different labels. Each path from the initial state S_1 to the final state S_t is a code word in C.

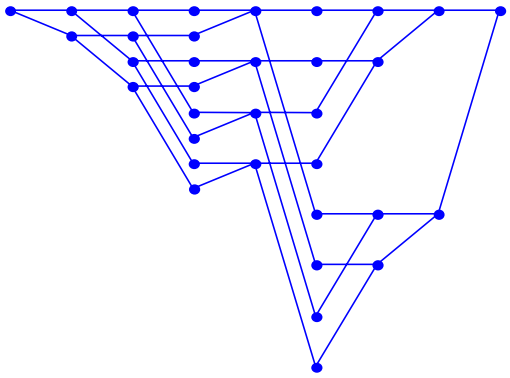


Fig. 3. Trellis for RM(8,4). Horizontal branches represent the label binary zero.

The collection of nodes at time *i* is obtained from the nodes at time (*i*-1) by:

$$S_i(l) = S_{i-1}(j) + \alpha_m \mathbf{h}_i; \quad m = 0,1,...,q-1, \quad \forall j \in \sum_i(C) \tag{1}$$

The transitions between the nodes at time *i*-1 and *i* are labeled by the particular value of $u_i = \alpha_m$.

The trellis representation described can be used for soft decision decoding of the block codes in the DDC scheme described in section II.C.

B. *Decoding with Viterbi*

A trellis can be used to perform soft decision decoding based, no longer on the hamming distance between code words, but the Euclidean distance. We simply search trough the trellis looking for the sequence that minimizes the Euclidean distance. However, we do not have to compute the distance for all q^k possible sequences. We can use the Viterbi algorithm [6] to eliminate certain sequences that merge in the trellis with other sequences that are much closer to the received sequence. The Viterbi algorithm works as follows:

Step 1) For block codes we know that we start at time *t*=0 in the all zero state, so we initialize its metric to zero. We then calculate the branch metric for the branches emerging from the all zero state. Either Euclidean metric or hamming metric may be used. The paths or survivors and their metric are stored for each new state.

Step 2) Increase *t* by one. Compute the metric for all the paths entering a state by computing the new branch metric and adding that to the corresponding state metric of the surviving state from the previous time step. For each step with a merger, store the path with the largest metric, the survivor, and its metric and discard all other paths.

Step 3) If *t* is smaller than the length of the trellis repeat step 2, otherwise choose the all zero state as "winner" and choose the corresponding sequence of survivors as output sequence. It should be noted that it is only for block codes that we know that we are in the all zero state in the end, otherwise it would not be a code word.

C. *Decoding with BCJR*

The Viterbi algorithm is optimal in the sense that it minimizes the word error probability by selecting the maximum likelihood sequence (called maximum likelihood sequence detection, MLSD). It does not, however, minimize the symbol or bit error probability, by choosing in each case the symbol or bit with the maximum aposteriori probability (MAP). The complexity of a MAP decoder is significantly higher and is, in general, not an alternative to MLSD since in most applications a MLSD decoder and a MAP decoder will have virtually identical performance. The main asset of the MAP decoder is that it produces aposteriori information bit probabilities, which can be used, in an iterative or concatenated decoding scheme.

An algorithm that does perform MAP decoding is the BCJR-algorithm [3]. The BCJR algorithm calculates the aposteriori probabilities for each symbol using the trellis as described below.

We define

$$\alpha_t(m) = \Pr\{S_t = m; Y_1^t\} \tag{2}$$

as the joint probability of the partial received sequence $(Y_1,...,Y_{t-1})$ and the state $S_t = m$

$$\beta_t(m) = \Pr\{Y_{t+1}^{\tau}, S_t = m\} \tag{3}$$

as the conditional probability of the remaining sequence $(Y_{t+1},...,Y_\tau)$ excluding *t* given the state at time *t* is $S_t = m$; and

$$\gamma_t(m',m) = \Pr\{S_t = m; Y_t \mid S_{t-1} = m'\} \tag{4}$$

as the joint conditional probability of Y_t and that the state at time *t* is $S_t = m$, given that the state at time *t-1* is $S_{t-1} = m'$.

Step 1) $\alpha_0(m)$ and $\beta_t(m)$ are initialized according to:

$$\alpha_0(0) = 1, \text{ and } \alpha_0(m) = 0, \quad \text{for } m \neq 0 \tag{5}$$

and

$$\beta_\tau(0) = 1, \text{ and } \beta_\tau(m) = 0, \quad \text{for } m \neq 0 \tag{6}$$

Step 2) As soon as Y_t is received, the decoder computes:

$$\alpha_t(m) = \sum_{m'=0}^{M-1} \Pr\{S_{t-1} = m'; S_t = m; Y_1^t\} = \sum_{m'} \alpha_{t-1}(m') \cdot \gamma_t(m',m) \tag{7}$$

and

$$\gamma_t(m',m) = \sum_X p_t(m \mid m') \cdot q_t(X \mid m',m) \cdot R(Y_t, X) \tag{8}$$

where $p_t(m \mid m')$ are the transition probabilities of the trellis, $q_t(X \mid m', m)$ the output, where X belongs to some finite discrete alphabet, and $R(Y_t, X)$ are the derived block transition probabilities.

Step 3) Once the complete sequence Y_1^t has been received, the decoder recursively computes:

$$\beta_t(m) = \sum_{m'=0}^{M-1} \Pr\{S_{t+1} = m'; Y_{t+1}^{\tau} \mid S_t = m\} = \sum_{m'} \beta_{t+1}(m') \cdot \gamma_{t+1}(m, m') \quad (9)$$

and the APP can then be calculated as:

$$\sigma_t(m', m) = \Pr\{S_{t-1} = m'; Y_1^{t-1}\} \cdot \Pr\{S_t = m; Y_t \mid S_{t-1} = m'\} \cdot \Pr\{Y_{t+1}^{\tau}, S_t = m\} = \alpha_{t-1}(m') \cdot \gamma_t(m', m) \cdot \beta_t(m) \quad (10)$$

It should be noted that the BCJR has to go through the trellis twice, once for calculating $\alpha_{t-1}(m')$ and $\gamma_t(m', m)$ while the sequence is being received and once recursively to calculate $\beta_t(m)$ and the APP.

D. Turbo Decoding

If some sort of concatenated coding scheme is used, iterative or turbo decoding can be used. The idea with iterative decoding is to divide the aposteriori probabilities, obtained from the BCJR algorithm, into three parts, a priori information, intrinsic information and most importantly extrinsic information. The extrinsic information is the indirect information you gain about a particular data bit from the other bits in the concatenated scheme. This extrinsic information is the fed back into the decoder as a priori information and the information is decoding once again, but this time with enhanced a priori knowledge, see Fig. 4.

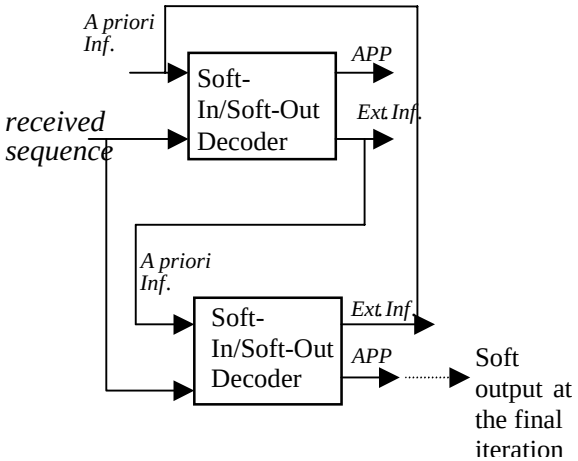


Fig. 4. Iterative (Turbo) Decoding

The advantage of turbo decoding is that one approaches the optimality of the MAP decoder with each iteration, but a much lower decoder complexity is required. The BCJR algorithm used in an iterative decoding scheme is now an option to the low complexity Viterbi algorithm.

In the DDC scheme the turbo decoding scheme may be used in conjunction with ARQ. The receiver may, for example, choose to do additional iterations on the already received results instead of asking for a new transmission, if the channel is sensitive to multi-user interference or alternately, ask for more information while iterating the already received data.

IV. Conclusions

We have put forward a framework for transmitting real-time data over a radio channel based on the Quality of Service parameters deadline (t_{DL}) and probability for correct delivery before deadline (P_d). Initial studies show a high probability of delivering the data before deadline. In this ongoing work we use a set of tools for handling information, such as hybrid ARQ, trellis representation of linear block codes, the Viterbi algorithm, the BCJR algorithm and Turbo decoding. These methods are used in conjunction with the deadline dependent coding (DDC) scheme in order to maximise the probability of delivering a message before a given deadline.

The DDC-protocol strategies for using the tools in different situations are not yet studied in detail. As an example, when high probability, P_d is required and the deadline, t_{DL} is small, two simultaneous actions can be taken: require more information from the transmitter and simultaneously performing some turbo decoding iterations for maximising the probability of correctly decoding the data block before deadline.

The required deadline and probability of delivery can be negotiated by the application, thus forming a flexible transmission mechanism. We believe that DDC might be a way of making it possible to use wireless radio channels in time- and safety critical applications.

V. References

[1] H. Bengtsson, E. Uhlemann and P.-A. Wiberg, "Protocol for wireless real-time systems", *Proc. of the 11th Euromicro Conference on Real Time Systems, York*, England, UK, June 9-11, 1999.

[2] S. Lin and D. J. Costello, Jr., "*Error Control Coding: Fundamentals and Applications*", Prentice-Hall, Inc. Englewood Cliffs, New Jersey 07632, 1983.

[3] L. R. Bahl, J. Cocke, F.Jelinek, and J. Raviv, Optimal decoding of linear codes for minimizing symbol error rate, *IEEE Transaction on Information Theory*, vol. IT-20, pp. 284-287, Mar. 1974.

[4] J. K. Wolf, "Efficient maximum-likelihood decoding of linear block codes using a trellis", *IEEE Transaction of Information Theory*, vol. 24, pp. 76-80, 1978.

[5] S. Lin, T. Kasami, T. Fujiwara and M. Fossorier, *Trellises and Trellis-Based Decoding Algorithms for Linear Block Codes*, Kluwer Academic Publishers, 1998.

[6] G. D. Forney, Jr.,"The Viterbi Algorithm", *Proceedings IEEE*, Vol. 61, pp. 268-278, 1973.

Empty page.

Wireless Networks for Manufacturing

Urban Bilstrup
Centre for Computer Systems Architecture, Halmstad University
urban.bilstrup@cca.hh.se

Per-Arne Wiberg
Centre for Computer Systems Architecture, Halmstad University
per-arne.wiberg@cca.hh.se

Abstract

Wireless technology's like Bluetooth and W-LAN has reached a tremendous attention the last years. The manufacturing industry could probably make large benefits from using these technologies. In this paper we propose new definitions of QoS parameters to be able to handle real-time traffic in wireless networks. Initial assumptions regarding media access in a multihop network is presented. Finally simulations showing that it is possible to map the new QoS parameters on a routing algorithm.

1. Introduction

The last years, tremendous steps have been taken in the field of local wireless communication. Concepts like Bluetooth, homeRF and W-LAN have been developed. These technologies will be found in all kind of different electronic devices, like mobile phones laptops, palmtops, and watches. We have found that the industry can benefit from using these technical advances in an industrial manufacturing environment. Likely applications are; mobile test-instruments, temporarily sensors, sensors mounted at moving machine parts and autonomous robots. The traffic in these types of systems is often delay sensitive and/or error sensitive. Unfortunately are the radio media known to be stochastic in its nature, causing non-deterministic delay by re-transmissions or bit errors in the messages.

A voice channel from a bas-station to a mobile phone, this is a real-time communication link with delay constraints on the delivery of a packet, but a voice channel can tolerate a bit error rate (BER) of 0.1 %. If a sampled data value from a sensor should be sent over a communication link, such a high BER can not be tolerated. Often when real-time communication is discussed, the message delivers before deadline is the quality parameter and that is true, as long as the media not causing any message errors or any re-transmission is preformed. When the message is totally destructed by interference (forward error correction have not been able to extract the information), then the only way of recovering the information is re-transmission of the packet or accepting that the information is lost.

If we want time and safety deterministic communication over the radio media, we have to define the quality of the media. The quality of a radio channel is often defined by the bit error rate (BER). From this we can define the probability of error free delivery (P) before a deadline (D). These two parameters (P, D) can be used for setting bounds on the communication system, or stating requirements of a link or a path through a network.

The layers in a communication protocol stack are given the requirements from the layer above. And the resources from the layer below, each layer has the possibility to optimize the resources given from below with the requirements from the layer above. Each layer strives to maximize the performance of the communication system with respect to D and P.

2. Link management

Meeting quality of service (QoS) guaranties over one radio link in a real-time communication system is fundamental, but because of the insecure radio media adaptation to the physical environment is necessary. Aspects considered of a radio media is not static like bit error rate of wired link, the error rate is variable depending on the fading radio signal, multi-user interference, interference caused from certain types of equipment, like microwave ovens and arc welders. Different types of diversity can be introduced to combat the interference, examples of diversity methods are; time diversity, space diversity, code diversity or simply increase the signal power. But all these methods have their drawback; our aim is to dynamically find the most efficient method or combination of methods to maximize the performance over a link or a path over a number of links through a network. At the same time minimizing the multi-user interference (signal power) to not reduce the system performance due to the requirements of the QoS parameters P and D for the link, one method with code and time diversity is presented in [2].

3. Media access

If a large number of units should be able to communicate with each other, media access methods must be used. Deterministic media access methods like TDMA and polling schemes does not scale with increased number of units, the bandwidth must be reused on a very local basis. Today this is often done with bas-stations, dividing an operation area into cells, each unit in the cell is accessed by the base station according to a TDMA schedule or different units are given different frequencies (FDMA). This gives a static infrastructure, but when the number of units continue to increase the size of the cell will get smaller and smaller and the number of base-stations increases and the wired backbone will get more and more complex. There are already today industries that are considering putting a Bluetooth [1] units in their products in order to down load software wireless in the factory. Sooner or later there will be large amounts of such units per 100 square meters that wants to connect to access points.

One possible way out of this is the use of ad-hoc networks forming clusters depending on the density of units, the units in a cluster adapting their signal power to physical size of a cluster minimizing the multi-users interference. This is possible if each cluster is able to perform the media access function of the base-station and repeating a message through the cluster to neighbor clusters until an access point (base station) to the wired network is reached by the message. The Bluetooth [1] standard provide the possibility creating these clusters. The standard provides both synchronous messages and asynchronous messages inside a cluster, very close to the function of a field bus [15], synchronous message are time triggered and the asynchronous messages are event triggered. Also inter cluster communication is possible but there is however no support for inter cluster synchronous messages. The exchange between two clusters will be done through a gateway unit, belonging to two clusters. This unit will experience a scheduling problem to which clusters it should belong to at the moment. The standard has no support for routing a message through a network of clusters. Is it possible to match the QoS parameters P and D to each cluster and solving media access scheduling at the gateway units between two cluster, then it should be possible to route a message through a network of clusters.

Each cluster schedule the traffic inside the cluster, a synchronous channel (time-triggered) passing through the cluster is scheduled statically as virtual circuit. Event-triggered messages are scheduled dynamically as asynchronous traffic. A message must be routed either to an access point to the wired backbone or to a destination in the local area.

4. Network layer simulations

In this section an on-going work on mapping the QoS parameters D and P on a routing protocol for a wireless real-time multihop network is described. Multihop networks [3,4,5,6,7] means that a packet is able to travel longer distance then the nodes radio range (single hop), by using multiple hops over other nodes.

In this work no clustering methode is used, forcing us to use a non-deterministic media access method, combining direct sequence spread spectrum (DSSS) with carrier sense multiple access (CSMA). This is not a time deterministic media access, because of the collisions that can happen at the receiver, but as mentioned in previous chapter the cluster method will be implemented and then asynchronous messages are possible through the network. The interesting part so far is the behavior of the routing algorithms and the possibility to match the parameters P and D to an insecure media. No access points to a wired network are yet present which will probably be the bottleneck points of the local multihop network.

The advantage of a wireless multihop networks, is that it does not use an infrastructure like the base station networks. The multihop network does not have any critical nodes that the network survival is dependent upon. A transmitting node only has to reach its nearest neighbors, instead of covering the whole area; this causes less interference to the surrounding traffic. Multihop networks gives a local and distributed view of the network, and probably the most intense traffic are local in an industrial environment.

In a wireless network, a node has radio connection with multiple nodes. The environment is mobile and the radio link between nodes is subject to frequent changes, therefore the flexibility of the routing is very important.

The higher layers above the network layer is able to negotiate with the network layer protocol about the QoS parameters.

The higher layer asks the network layer protocol about the possibility to send a packet over the network with a deadline (D). The network protocol answers with a P that tells the about the possibility for a packet to reach the destination nod before D expires. Then the higher layers is able to reconsider D and try again until an agreement about the QoS parameters D and P is reached, and then the transmission starts.

Weights (W) are assigned to each radio connection (links) between neighbor nodes. The value of W reflects the QoS of a link between two nodes. In this work, the link weight W is the time between a node generating or receiving a packet until it is received correct by next destination node. Other properties can be mapped to W

such as: bit error rate (BER), probability of error free delivery or number of hops.

Information is stored in every node about W for each outgoing link. The update mechanism [8] below is used to update W; this gives a moderate adaptation of W value on each link. The learning rate η moderates the update of W, to combat frequent link weight fluctuations (ping pong effects between two links)

$$Wnew = W + \eta(Wupdate - W)$$

The routing algorithm tries to find a path through the network; it calculates the sum of the weights (Wt) for different paths from source to destination. The path with the best Wt is chosen, this path gives the highest probability for a packet to reach the destination node, before deadline expires.

The routing algorithm in use at the moment is the highly dynamic Destination-Sequenced Distance-Vector routing algorithm (DSDV) [9]. DSDV is more distributed and gives less routing control overhead than the more centralized link-state method [10], which is another well-known routing algorithm.

The calculation of P given Wt and D, is one of the most critical parts of this protocol. At a source node a routing table gives Wt and the next node address along the path to the destination, D is given by the application. P is then retrieved from a look-up table with the index D and Wt. The look-up table is developed using statistics from simulations, modeling the distribution of P in respect to D and Wt.

4.1 Simulations

A discrete event, network simulator (NESU)[12] has been developed and used for the performance evaluation. NESU is a flexible platform, designed for evaluating and comparing network routing algorithms. All the nodes have their individual parameters, such as speed, trajectory, and packet intensity etc. The nodes can move in a rectangular region (in this case 60 x 60 m) according to a given motion model. All nodes have a fixed radio transmitting power, which in our case gives a transmitting radius about 20 m, depending on the interference in the area. The radio channel model [2,13] determines the radio connection between two nodes, the BER is calculated for every packet sent over the link. If the BER ratio is over a specified threshold, the packet is considered destroyed.

A 25-node mobile multihop network has been simulated, using the following simulation parameters. The radio transmitter/receiver has been assumed the PRISM chipset [14]. The PRISM chipset gives a bandwidth of 1Mbit/s and a transmitting power of 63 mw, it is based on DSSS multiplexing. The nodes move with a speed ranging from 0 to 3 m/s. The time between two data packets are generated at each node, is exponential distributed. If there is no possible path between source and destination the packet is deleted and counted as destroyed, a data packet is 3000 bits long and takes 3ms to send one hop.

In this, ongoing work initial simulations have shown following results: When the node motion is Brownian, there is a possibility that the nodes are grouped into different clusters without any connections. Due to this effect, only 85% of correct arrival are reached even if the deadline approaches infinity, this can be considered by power control inside a cluster where the nodes control their transmitting power according to the density of nodes.

We have also investigated a more likely motion model for the nodes, where the nodes are moving in a more predictable way. The intense traffic with small deadlines (20ms, average 1,5 hop) is local, typically this can be a moving smart sensor sampling a value on a large machine sending the value to a smart actuator. An autonomous robot traveling around the factory delivering things at different machines, has a less intense but more ”global” traffic with longer deadline (50ms, average 2,5 hop). Some nodes are not mobile at all they may be control computers sampling data from sensors and sending data to actuators. In a more planed environment like this, about 95% of the packets arrive correct before their deadline is expired. Multi-user interference in this case causes packet loss. Collisions occur when two nodes tries to send to the same node at the same time or when a node is busy transmitting and another nod tries to reach the busy node. These collisions are very time consuming and the deadline expires sometimes, because of re-transmission.

Ppredicted	Preal
0.3	0.34
0.4	0.39
0.5	0.50
0.6	0.62
0.7	0.73
0.8	0.82
0.9	0.89
1.0	0.97

Table 1: Real probability Vs predicted probability for a message path through the network before deadline is expired.

Although the degree of correct arrival have not been the best depending on the insecure radio media and collisions, the prediction of a packets correct arrival is possible (Table1). This is the key to map the parameters P and D to a multihop network. We can say that a message has probability P of reaching its destination before the deadline is expired.

5. Conclusions

Industries have a great interest of using the wireless technical advances in an industrial manufacturing environment. But the problem is that today, are the definitions for real-time communication traffic not suitable for the radio media.

The ability to say that a message has a probability of correct delivery to destination before deadline gives the opportunity to discuss real-time communication over the wireless media.

The multihop network simulations show that the one most important factor for delivering a packet in time is the control of collisions in receiving nodes, which possible can be done with clusters and another media access protocol then the combining of DSSS and CSMA.

Another important guarantee is that every node in the network has a path to the other nodes. This problem can be solved with clusters that adapt their physical size to the density of nodes. The fact that multihop network is scaleable with respect to bandwidth when new nodes are added is very much in favor of these network.

6. Future work

We will focus our future work on refining the protocol and dealing with the receiving collision problem, this will be done by introducing the method of clustering nodes and use a time deterministic media access scheme inside each cluster.

The scheduling protocol for the gateway nodes between two clusters must be dealt with. And the parameters P and D must be mapped on this schedule.

One other thing that should be looked into are the use of power control to control the physical cluster size with respect to the density of nodes.

7. References

[1] J. Haartsen, “Bluetooth – The universal radio interface for ad hoc, wireless connectivity“, Ericsson Review, No. 3, 1998, pp. 110-117.

[2] H. Bengtsson E. Uhlemann and P. Wiberg “Protocol for Wireless Real-Time Systems”, *Proc. ECRTS’99,* York , UK, June. 9-11 1999.

[3] J. Jubin and J. D Craighill, ”The DARPA packet radio network protocol”, *Proc IEEE*, Jan. 1997.

[4] T. Chen, J. T. Tsai, and M. Gerla, "QoS Routing Performance in Multihop, Multimedia, Wireless Networks", in *Proc. IEEE ICUP97 part2*, 1997, pp. 557-451.

[5] C. Cheng and R. Riley, “A loop-free extended bellman-ford routing protocol without bouncing effect”, *In Proc. Of ACM SIGCOMM Conf.*, 1989.

[6] D. B Johnson, ”Routing in Ad Hoc Networks of Mobile Hosts”, *Proceedings of the IEEE Workshop on Mobile Computing and Applications,* Dec. 1994.

[7] V.D. Park and M.S. Corson, “A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks”, *in Proc. IEEE INFOCOM’97*, Kobe, Japan, Apr. 7-11 1997.

[8] S. Kumar, ”Confidence based Dual Reinforcement Q-Routing: an On-line Adaptive Network Routing Algorithm*”, Report AI98-267* The University of Texas at Austin, Artificial Intelligence Laboratory, May. 1998.

[9] C. E. Perkins and P. Bhagwat, ”Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers*”, Proc. of ACM SIGCOMM ,* Aug. 1994, pp. 234-244.

[10] A.S. Tanenbaum, *Computer Networks*, Third Edition, Prentice Hall PTR, Upper Saddle River, New Jersey, USA, 1996, ISBN 0-13-349945-6.

[11] R.C. Dixon, *Spread spectrum systems*, third edition, John Wiley & Sons inc., USA, 1994, ISBN 0-471-59342-7

[12] S. Ackmer U. Bilstrup L. Svalmark, “Routing Protocol for Wireless Real-Time Multihop Network*”, technical report CCA-9906*, Halmstad University, Jan. 1999.

[13] S.Y. Seidel, “914 MHz Path Loss Prediction Models for Indoor Wireless Communication in Multifloored Buildings”, *IEEE Transactions on Antennas and Propagation*, vol 40, NO. 2, Feb. 1992.

[14] Advanced Micro Device, AM79C930, Advanced Micro Device Inc. 1995.

[15] J.R. Pimentel, *Communication Networks for Manufactoring*, first edition, Prentice-Hall inc., USA, 1990, ISBN 0-13-154402-0

Empty page.

Position Statement for:

Methods for integration of Heterogeneous Real-Time Services into High-Performance Networks

Carl Bergenhem[1]

CC-lab, School of information science, computer- and electrical engineering, Halmstad University, Box 823, S-301 18 Halmstad, Sweden email: carl.bergenhem@ide.hh.se, Phone: +46 (0)35 167483, Fax: +46 (0)35 120348

1. Introduction:

The work described in this statement is to be performed within the ARTES project with the same name as above. The project has been active since January 2000. The described areas of focus only account for a part of the ARTES project. This work package, together with the survey of research and industrial applications, accounts for the work to be done during the first year.

Current work is based on previous work reported in [1 - 5]. The network, referred to as control channel based fibre ribbon ring (CC-FPR), is based on fibre ribbon links connected together in a ring topology, see fig. 1. Each link is unidirectional and consists of 10 fibres. A medium access method referred to here as one cycle method (OCM) was proposed in [1,2]. The medium access method together with the network provides support for, among other services, slot reservation and spatial reuse of slots. The medium access method uses a separate fibre or channel for sending network arbitration information between the nodes in the network. This implies that there is no arbitration overhead sent over the data channel. The medium access method supports both fair sharing and reservation of bandwidth.

The future work proposed in this report is aimed at an increased understanding of how different classes of real-time traffic (hard and soft traffic) affect each other in a CC-FPR network. Simulation of the network loading will be used as a method to gather results. We believe that this study also will result in a more general understanding, or at least directions for further study, in the area of integrated support for hard and soft real-time traffic.

Previous work [3, 4] has shown that it is possible to extend the functionality of the network with functions for parallel processing. This support may increase performance in parallel processing applications and will be further studied and analysed with focus on real-time support. Again, simulation will account for a large part of the results

[1]A large portion of this document is copied from the ARTES project application authored by Magnus Jonsson and Bertil Svensson

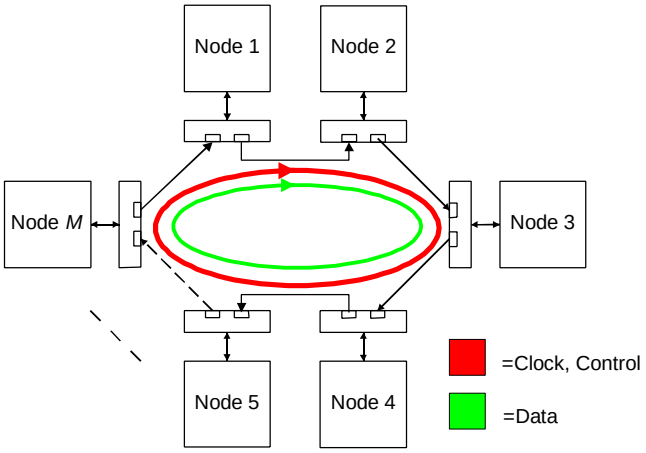


Fig 1. The control channel based fibre ribbon pipeline ring network.

In [5] a new medium access method referred to as the two cycle method (TCM) was proposed in order to get better support for heterogeneous real-time traffic. Its properties will be analysed and compared to the original method OCM by making simulations. A comparative study will be performed based on simulations of both OCM and TCM. Conclusions will be drawn as to the two methods suitability for different applications.

Lastly a case study of current and emerging real-time communication methods and network technologies will be performed in parallel with this first-year work-package. Possible target applications, which future work may be directed towards, will be identified.

2. Main areas of focus (work package)

2.1. Extra functionality

In the studied network a separate fibre is used for distributing network arbitration information. This control fibre may also be used for “special services”; sending extra information such as barrier synchronisation, global reduction, acknowledge of packets and flow control. Services such as global reduction and barrier synchronisation, are useful in parallel processing and may increase performance. Some research has already been published [3, 4]. The extended functionality of the control channel will be further investigated and the results of it studied, especially in terms of real-time support. Implications of its use, latency of packet acknowledgement etc. will also be studied.

2.2. A new medium access method

A proposition of a new medium access method, two-cycle method (TCM), was reported in [5]. The method has not yet been presented or published at an international conference. TCM differs from the original method, OCM, in that the nodes in the network may vote for access to slots. A control package is sent twice around the ring as opposed to once in OCM. This gives some drawbacks such as latency but also, as further research may prove, advantages. For example since it is always the message with highest priority, e.g., earliest deadline that gets to reserve a slot, one can regard the TCM method as to provide global optimisation of message timing constraints. How to best make use of this will be investigated. The integration with already supportedd real-time services in OCM will also be investigated. Simulations of TCM will be carried out, see below. Average- and worst case analysis of the method will confirm the simulation results.

2.3. A study and simulation of network load

A study of the implications of combining hard real-time traffic with soft quality of service (QoS) constrained real-time traffic in a ring network, such as previously studied, will be conducted. The motivation for this study is to see how useful the network capacity, remaining after a network already has a certain load of hard real-time (HRT) traffic, is to soft real-time (SRT) traffic. In other words, the capacity overhead required in a network to successfully perform certain tasks will be investigated? This knowledge is useful when designing networks and making capacity planning and performance predictions.

The questions posed in the part of the research will be answered partly by doing simulations. Statistical models of hard real-time, QoS constrained soft real-time will be defined together with a definition of the network, see fig. 2. These parameters may be decided by studying similar work and by using common sense.

The simulation will be executed with the notion that the HRT load will affect the possible load of SRT traffic and that it may not be possible to schedule traffic to 100% of maximum network capacity.

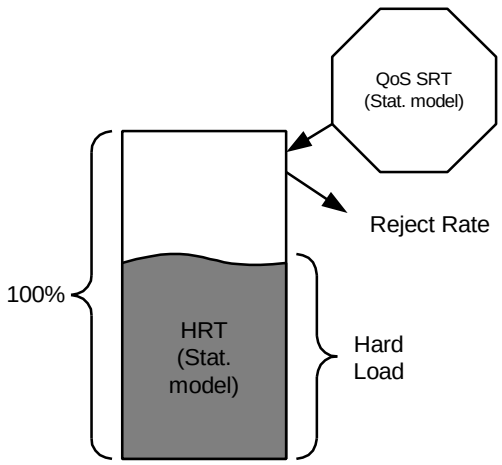


Fig 2. The model used for simulation

Questions that simulation will try to answer:

- How does a certain load level of hard traffic effect incoming soft traffic, e.g. reject rate, jitter etc.?
- How much QoS SRT traffic the network can be loaded with at a certain HRT load level.
- Can an optimum balance of traffic be found?

Simulations will be carried out with varying parameters on a basic network configuration. The performance of TCM and OCM will be evaluated and compared. Conclusions may be drawn as to the suitability of the two methods in different situations. A metric for best describing network load will be chosen or developed.

2.4. Case study and future work

A survey of state of the art communication networks will be conducted in parallel to the rest of the work. This will also give an opportunity to study emerging industrial applications with high communications demands and will thus give guidelines for future work. In the survey it will thus be possible to identify applications that may draw benefit from the technology conceived within the scope of this work. Further research efforts may be directed towards studying particular applications. Examples of systems that have previously been studied are radar signal processing systems [6].

3. Summary

CC-FPR, the network that this research is based on has in previous work demonstrated promising properties of determinism etc. In this work proposal, we present ideas for further investigation and improvements towards integration of heterogeneous real-time services. We believe that the simulations and study will give indications of how to best integrate soft and hard real-time communication in high performance networks.

4. References

[1] Jonsson, M., "Control-channel based fiber-ribbon pipeline ring network," *Proc. Massively Parallel Processing using Optical Interconnections (MPPOI'98)*, Las Vegas, NV, USA, June 15-17, 1998, pp. 158-165.

[2] M. Jonsson, "Two fiber-ribbon ring networks for parallel and distributed computing systems," *Optical Engineering*, vol. 37, no. 12, pp. 3196-3204, Dec. 1998.

[3] M. Jonsson, C. Bergenhem, and J. Olsson, "Fiber-ribbon ring network with services for parallel processing and distributed real-time systems," *Proc. ISCA 12th International Conference on Parallel and Distributed Computing Systems (PDCS-99)*, Fort Lauderdale, FL, USA, Aug. 18-20, 1999, pp. 94-101.

[4] M. Jonsson, C. Bergenhem, and J. Olsson, "Fiber-ribbon ring network with services for parallel processing and distributed real-time systems," *Proc. SNART'99 Real-Time Systems Conference*, Linköping, Sweden, Aug. 24-25, 1999, pp. 46-54.

[5] C. Bergenhem and J. Olsson, "Protocol suite and demonstrator for a high performance real-time network," *Master thesis, Centre for Computer Architecture (CCA)*, Halmstad University, Sweden, Jan. 1999.

[6] M. Taveniku, A. Åhlander, M. Jonsson, and B. Svensson, "The VEGA moderately parallel MIMD, moderately parallel SIMD, architecture for high performance array signal processing," *Proc. 12th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP'98)*, Orlando, FL, USA, Mar. 30 - Apr. 3, 1998, pp. 226-232.

Empty page.

Research Issues on System Architecture for Mechatronics Systems

De-jiu Chen
Royal Institute of Technology, Mechatronics Laboratory, Department of Machine Design,
100 44 Stockholm, Sweden
chen@damek.kth.se

Abstract: The ARTES project no: A4-9805 aims to progress the state of the art on software architectures for complex control systems. A specific goal of the project is to provide a system architecture suitable for scalable, maintainable and reconfigurable mechatronics systems controlled by an embedded distributed computer system. This are requirements found by the legged locomotion systems that are being studied and used as a case study within the project. In this report, we briefly introduce the system characteristics and give introductory knowledge on the key research issues around system architecture.

Background

A mechatronics system embodies technologies from several engineering disciplines in the domains of mechanics, automatic control, computer software and hardware. In such a system, the major portion of system functionality is realized by adopting automatic control through computer software and hardware, as well as a variety of sensors and actuators. As this approach promises enhanced functionality, performance and flexibility, more and more traditional mechanical solutions in modern machinery are being improved or replaced by the mechatronics solutions. For example, *ABS* (Antilock Brake Systems) for the performance of the breaking system of a car or the *x-by-wire* flight controls in a modern passenger airplane.

There is inherent complexity in this multidisciplinary approach. First, the complexity arises from the multidimensional interdependency across the involved domain technologies (i.e., mechanics, electronics, automatic control, computer software and hardware). To complete a specific task or to satisfy a desired non-functional quality attributes (e.g., versatility, modifiability, maintainability, etc.), this interdependency needs to be correctly established and handled in the system. Second, the software makes the system structure and behavior less comprehensible and controllable. This is caused by the lack of physical constraints and the large "state-space" of software (e.g., the fine-grained codes and their complex interrelations). In contrast to the physical constraints in other engineering disciplines, there are only storage and (temporal) performance constraints on software. This gives tremendous design freedom of software; however, also makes the design more error-prone. As a mechatronics system has physical impact on its environment, software failure is often intolerable.

To facilitate the development and maintenance of the system, a variety of measures managing the complexity are needed. At this point, managing the software complexity becomes extremely important for the system functionality and dependability. In the following sections, we will list some key concepts on how to handle the complexity with the focus on software. The aim is to illustrate some important research issues around the system architecture.

An architecture-based design approach

To handle the complexity, the design must progress in a top-down and evolutionary manner. In a systematic top-down approach, the system is hierarchically decomposed into lower level constituent units and their relationships. With the abstractions, we can concentrate on the system temporal and spatial structures, which in turn determine the essential properties of the system. For software, it means that programming-in-the-large and the programming-in-the-small issues are separated. Meanwhile, the design should also progress evolutionarily in the sense that we map requirements on the system decompositions and remedy any inconsistency as the design goes on.

The above mentioned design approach is an architecture-based design approach. Although no consensus on the definition of system architecture has been reached today, we believe architecture is a high-level (i.e., abstract) representation of the logical, temporal and spatial structures of the system. It shows how the system properties in terms of functional and non-

functional quality attributes are determined by its constituent units and their relations. Working concretely with architecture for the system design implies the following issues.

System architectural description

One of the most important research issues around system architecture is the description of the system abstractions, i.e. system architectural description. It is the basis for all design activities, e.g., comprehension, communication quality prediction, trade-off, decision-making, manipulation, modification, reuses, etc.

The most fundamental requirement on the description is *completeness* in the sense that the described information should be sufficient for the purposes. For mechatronics software, the description should at least include the functional and behavioral/temporal aspects of the system. To facilitate the comprehension, the description should be based on the principle of *separation-of-concerns*. It means that the description should be given in multiple hierarchical layers and orthogonal views (e.g., the structural, functional, behavior, and reliability views, etc.).

The research of architectural description for mechatronics systems should focus on elaborating the needed information for the design, as well as integrating and improving legacy modeling techniques.

Software components

In practice, no design proceeds in a purely top-down manner. During the design, the designers have to use knowledge relating to the system implementation. Otherwise, a semantic gap between an architecture and its implementation may arise. For instance, they may consider similar control algorithms and software codes for quality prediction, and available off-the-shelf components (e.g., sensors, microprocessors, communication bus system, real-time operating system, drive units and actuators, etc.) for implementation. Thus, accessing bottom-up information is unavoidable in a top-down design approach and has impacts on the system architecture.

Fundamentally, a software component is a constituent unit of a software system. As the system exists in different hierarchical layers, there are also different classes of software components. For the real-time software, we may consider the software hierarchy runs from the automatic control solutions at the top level to the software implementation at the lower level (e.g., class and object in an Object-Oriented language). The constituent parts in these layers have multiple interrelations established by the adopted automatic control logic (e.g., required control data flow, desired timing and synchronization), the chosen software implementation (e.g., time consumption and shared timing resource), and the shared hardware resources (e.g., I/O, memory, processor). Thus, to support the design and to achieve flexibility and reuse, the interface of a software component should provide sufficient information so that these multiple interrelations can be established properly.

Some answers on how to classify the software components in the system and how to define their logical, structural and temporal contexts, as well as the required implementation resources and services, should be provided by the research.

Architectural principles and styles

During the design, the designers also use the knowledge from the engineering background (i.e., evaluation criteria, and mature solutions on partitioning/structuring, allocating/distributing, scheduling, etc.). Architectural principles and styles formulate the essential features of existing mature solutions of software, which can be compared with the principles and mature solutions in other well-established engineering domains. Under the circumstances, it is easier to reuse the existing solutions, to elicit hidden requirements in time, and to provide early quality predictions for the domain systems. Thus, research efforts are needed to identify and categorize useful architectural principles and styles.

Conclusions

To illustrate the key research issues around system architecture, we have briefly introduced the system characteristics and the architecture based system design approach. The fundamental problems needs to be tackled by the research are: what constitute the system software and how they interact with each other; how the system architectural description should be provided so that comprehension and analyses are facilitated; how to make software engineering more mature and constrained. A summarization of the project status can be found in the project homepage (http://www.damek.kth.se/~chen.html).

Formal and Probabilistic Arguments for Component Reuse in Safety-Critical Real-Time Systems

H. Thane and A. Wall

Mälardalen Real-Time research Center (MRTC)

Mälardalen University, Västerås, Sweden

{awl,hte}@mdh.se

Abstract: In this paper we are going to introduce a novel framework for formal and probabilistic arguments of component reuse in safety-critical real-time systems. Using both quantitative and qualitative descriptions of component attributes and assumptions about the environment, we can relate input-output domains, temporal characteristics, fault hypotheses, reliability levels, and task models. Using these quantitative and qualitative attributes we can deem if a component can be reused or not. We can also deem how much and which subsets, of say input-output domains, that need additional functional and safety verification. This framework will give formal and probabilistic arguments for reuse of components in safety-critical real-time systems.

1 Introduction

The introduction of computers into safety-critical systems lays a heavy burden on the software designers. The public and the legislators demand reliable and safe computer systems, equal to or better than the mechanical or electromechanical parts they replace. The designers must have a thorough understanding of the system and more accurate software design and verification techniques than have usually been deemed necessary for software development. However, since computer related problems, relating to safety and reliability, have just recently been of any concern for engineers, there exists no holistic engineering knowledge of how to construct safe and reliable computer based systems. There exist only tidbits of knowledge and no silver bullets that can handle everything. Some people do nonetheless, with an almost religious glee, decree that their method, principle or programming language handles or kills all werewolves (bugs) [8].

In order to be able to design software that is as safe and reliable as the mechanical or electromechanical parts it replaces, and to reduce the time to market (TTM) focus of current research and practice has turned to the reuse of proven software components. The components can be arbitrary small or large in terms of functionality. The basic idea is that the system designer should be able to procure (or reuse) software components in the same manner as hardware components can be acquired. Hardware component like nuts, bolts, CPUs, memory, A/D converters can be procured based on functionality, running conditions (environment) and their reliability. Hardware components are typically divided into two classes, commercial and military grade components, where military grade electronics can withstand greater temperature spans, handle shock and humidity better than commercial components. However, for software it is not so easy to make the same classification since the metrics of software are not based on physical attributes, but rather on its design, which is unique for each new design. In fact, software has no physical restrictions what so ever, like mass, size, number of components. Software has neither structure nor function related attributes like strength, density and form. The sole physical entity that can be modeled and measured by software engineers is *time*. With but a few exceptions all safety critical systems are also real-time systems.

In this paper we are going to introduce a framework for formal and probabilistic arguments of component reuse in safety-critical real-time systems, based on the restrictions that time, and the component contracts impose on the system behavior.

1.1 What are software components?

There exists yet no commonly agreed upon definition of what constitutes a software component. Most people do however agree that the concept of components is the idea of reusable software entities. In this paper, for the sake of enabling analysis of components in safety-critical real-time systems we do not only define a component in terms of software. A component must also be well documented in terms of design documentation and preferably also in terms of test documents, and in evidence of the reliability achieved.

We have chosen a hierarchical/recursive definition of components in real-time systems (RTS). The smallest components are the actual assembly instructions provided by the processor, some way up in the hierarchy a component is a task or process. A task is a unit that usually performs a single calculation, for instance calculating a new process value based on measurements collected by another task. On the next level, several tasks can work together in a transaction where the transaction itself is a component. A transaction is a set of functionally related and cooperating tasks, e.g., sample-calculate-actuate loops in control systems. The relationship between the cooperating tasks with respect to precedence (execution order), interactions (data-flow), and a period time typically define each transaction.

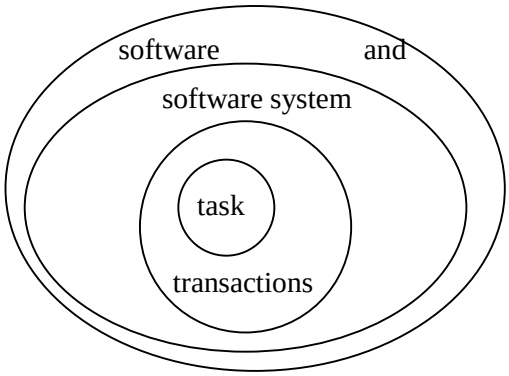


Figure 1. Hierarchical definition of components for real-time systems.

All tasks and transactions together make up a software system that also might be a component from the complete systems point of view. As an example, the software controlling the air bag in a car could be considered as a component from the air bags point of view.

As this definition eventually forces us to define the universe as a software component, we have to stop the recursion somewhere. At least on the level of abstraction in this paper, the operating system is not considered a component. We like to think of the operating system as the infrastructure for the computer system. It provides the necessary services for the software but it also restricts the possibilities of implementing the requirements put upon tasks and transactions.

2 Component contracts

A contract is a specification for a component in terms of what functionality a component provides. Furthermore, the contract states the conditions under which the specified functionality works. In other words, a contract of a software component is a statement saying that: provided that certain assumptions are fulfilled, the specified functionality will be provided and error free. Such contracts can be described with arbitrary formality. The more formal the contracts are, the richer possibilities of verifying that the component will work correctly in a system, or that the software system will work correctly with the new component as a part of it. In this paper we will settle for a functional description in some natural language although we could use some formal temporal language such as timed automata [1]. However, the main focus in this paper is on the specific demands put on component contracts in safety-critical real-time systems.

As the temporal behavior is of great importance for the correctness of real-time systems, the temporal assumptions have to be specified in the contracts. The temporal attributes make up the task model that is used when formally verifying the temporal behavior of the system, i.e., if the system is schedulable or not. However, there exist two different task models for tasks in a real-time system, one that exists on the design level and one that is actually provided by the given infrastructure. Typically, the design task model consists of end-to-end deadlines for transaction, jitter constraints, etc which is realized in the implementation using the temporal attributes at hand in the infrastructure. As a consequence, we will divide the contract into two different main parts, the design task model and the infrastructure task model.

2.1 The design task model

As discussed in the introduction, our view on components in real-time systems is that they are either single tasks or a transaction consisting of tasks and transactions. Thus, the contract must specify whether or not the component is a transaction or not. If the component is indeed a transaction it has a precedence relation between the cooperation tasks and there is a flow of data between the tasks. Note that a single task is a specialization of a transaction as it is a transaction without any precedence relations or data-flow. There might be several different ways to represent precedence relations. We will, however settle with a simple graphical notation called a precedence graph. A precedence graph is simply a directed graph visualizing all tasks in the transaction interconnected by an arrow indicating the direction of the precedence relation between those tasks. In Figure 2 a simple precedence graph is shown where task *A* precedes, task *B* and task *C*. We can also see that task *B* and *C,* precede task *D*.

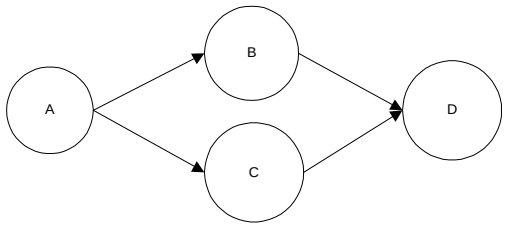


Figure 2 A simple precedence graph

Concerning the data-flow in a component we are interested in exposing three different attributes in the contract namely: where, how and what. The attribute “Where”, declares the receiver of data, i.e. another task within the component. The data can flow throughout the component using different strategies. The two different possibilities are synchronous and asynchronous. If data is transported in an asynchronous manner, the contract must specify how the data is treated on the receiver side. This is important since data can be consumed either faster or slower than it is produced in a multi-rate transaction. If data is produced in a faster pace than it is consumed, the contract must specify whether or not new data should overwrite old data or if data should be buffered to provide history. Finally, concerning communication, the size of data is important, as the size will restrict the speed at which the component can execute.

The temporal attributes for a real-time component specifies the required temporal behavior for the component. As discussed, these attributes must not necessarily have there correspondence in the infrastructure, but should rather be realized using the constructions provided by the infrastructure. As we focus on reuse in this paper, we want our components to be as general as possible in terms of the temporal constraints. The generality is obtained by specifying the temporal constraints as time intervals. Later on in Section 3, we will elaborate further on how these intervals are obtained. Transaction components (where a single task is a special case) have period times, jitter constraints on period times, end-to-end deadlines, etc. However, it is not trivial to represent the period time for a component consisting of several tasks all running with different period times (multi-rate). We propose that period times are only valid for single-task components. The period time of a transaction is inherited from the system in which the component will be reused. The tolerances on the period times of a component are given in terms of decreased/increased inter arrival times, however where the precedence, and mutual exclusion relations, are preserved between the sub-components. This is a way of parameterize the component to fit in a new environment.

2.2 Infrastructure

Derived from the recursive definition of components and component relations, we can with respect to real-time systems define the components forming the basis for running the real-time tasks as the infrastructure, i.e., the real-time operating system and its attributes. We are now going to give a qualitative classification of the infrastructure for real-time operating systems based on their execution strategy, synchronization mechanisms, and communication mechanisms.

Execution strategy

The execution strategy of a real-time system defines how the tasks of the system are run. A usual classification is event and time triggered systems. An event-triggered system is a system where the tasks activation is determined by an event that can be of external or internal origin, e.g., an interrupt, or a signal from another task. There are usually no restrictions what so ever on when events are allowed to arrive, since the time base is dense (continuos). For time-triggered systems events are only allowed to arrive into the system, and activate tasks, with a certain minimum and maximum inter-arrival time; the time-base is sparse. The only event allowed is the periodic activation of the real-time kernel, generated by the real-time clock.

Another characteristic of an execution strategy is the support of single tasking or multitasking. That is, can multiple tasks share the same computing resource (CPU), or not? If the infrastructure does support multitasking does it also support task interleavings, i.e., does it allow an executing task to be preempted during its execution by another task, and then resumed after the completion of the preempting task?

A very common characteristic of an execution strategy, especially for multi-tasking RTS, is the infrastructure task model. The task model defines execution attributes for the tasks [2][7][11][12]. For example:

- **Execution time**, C_j. Where $C_j \subseteq [BCET_j, WCET_j]$, i.e., the execution time for a task, *j*, varies depending on input in an interval delimited by the task's best case execution time ($BCET_j$) and its worst case execution time ($WCET_j$).
- **Periodicity**, T_j. Where $T_j \subseteq [T_j^{min}, T_j^{max}]$, i.e., the inter-arrival time between the activations of a task, *j*, is delimited by an interval ranging from the minimum inter-arrival time, to the maximum inter-arrival time. For strictly periodic RTS, $T_j^{min} = T_j^{max}$.
- **Offset**, O_j. Where O_j defines an offset relative the period start at which the task *j* should be activated. Offsets also go by the name release times.
- **Priority**, P_j. Where P_j defines the priority of task *j.* When several tasks are activated at the same time, or when an activated task has higher priory than the currently running task, the priority determines which task should have access to the computing resource.
- **Deadline**, D_j. Where D_j defines the deadline for the task, *j*, that is, when it has to be finished. The deadline can be defined relative task activation, its period time, absolute time, etc.

For different infrastructures the task model vary with different flavors of the above-exemplified attributes.

Synchronization

Depending on the infrastructure we can either make necessary synchronizations between tasks off-line if it is time triggered and supports offsets, or we can synchronize tasks on-line using primitives like semaphores. In the off-line case we guarantee precedence and mutual exclusion relations by separating tasks time-wise, using offsets.

Communication

Communication between tasks in RTS can be achieved in a multitude of ways. We can make use of shared memory which is guarded by semaphores, or time synchronization, or we can via the operating system infra structure send messages, or signals between tasks. Depending on the relation between the communicating tasks, in respect to periodicity, the communication can vary between totally synchronous communication to totally asynchronous communication. That is, if task *i* sends data to task *j*, and both tasks have equal periodicity, $T_j = T_i$, we can make use of just one shared memory buffer. However, if $T_j > T_i$ or $T_j < T_i$ the issue gets more complicated. Either we make use of overwriting semantics (state-based communication), using just a few buffers, or we record all data that has been sent by the higher frequency task so that it can be consumed by the lower frequency task when it is activated. There are several approaches to solving this problem [3][4][5].

2.3 Failure semantics

Components can fail in different ways. The manner in which they fail can be categorized into failure modes. Failure modes are defined through the effects, as perceived by the component user:

1. ***Sequential failure behavior***

   This failure mode includes:

   - *Control failures*, e.g., selecting the wrong branch in an if-then-else statement.
   - *Value failures*, e.g., assigning an incorrect value to a correct (intended) variable.

- *Addressing failures*, e.g., assigning a correct (intended) value to an incorrect variable.
- *Termination failures*, e.g., a loop statement failing to complete because the termination condition is never satisfied.
- *Input failures, e.g.*, receiving an (undetected) erroneous value from a sensor.

Multitasking and real-time failure behavior

2. ***Ordering failures***, e.g., violations upon precedence relations or mutual exclusion relations.
3. ***Synchronization failures***, i.e., ordering failures but also deadlocks.
4. ***Interleaving failures***, e.g., side effects caused by non-reentrant code, and shared data, in preemptively scheduled systems.
5. ***Timing failures.*** This failure mode, yields a correct result (value), although the procurement of the result is time-wise incorrect. For example, deadline violations, early start of task, incorrect period time, too much jitter, too many interrupts (too short inter-arrival time), etc.
6. ***Byzantine and arbitrary failures***. This failure mode is characterized by a non-assumption, meaning that there is no restriction what so ever with respect to which effects the component user may perceive. Therefore, has the failure mode been called malicious or fail-uncontrolled. This failure mode includes two-faced behavior: a component can output "X is true" to one component user, and "X is false" to another component user.

The above listed failure modes build up a hierarchy where byzantine failures are based on the weakest assumption (a non-assumption) on the behavior of the components, and sequential failures are based on the strongest assumptions. Hence byzantine failure is the most severe and sequential failure the least severe failure mode. The byzantine failure mode covers all failures classified as timing failures, which in turn covers interleaving failures, and so on.

More formally: Sequential failures $\subset$ Ordering failures $\subset$ Synchronization failures $\subset$ Interleaving failures $\subset$ Timing failures $\subset$ Byzantine failures. That is, $1 \subset 2 \subset 3 \subset 4 \subset 5 \subset 6$.

The component user can also characterize the failure modes according to the viewpoints domain and perception. A distinction can be made between primary failures, secondary failures and command failures [6]:

- *Primary failures*

  A primary failure is caused by an error in the software of the component so that its output does not meet the specification. This class includes byzantine failure modes, and sequential failure modes.

- *Secondary failures*

  A secondary failure occurs when the input to a component does not comply with the specification. This can happen when the component is used in an environment not designed for, or when the output of a preceding task (component) does not comply with the specifications of a succeeding task's (component) input. This class includes interleaving failures, and sequential input failure modes.

- *Command failures*

  Command failures occur when a component delivers the correct result but at the wrong time or in the wrong order. This class covers timing failures, synchronization failures, and ordering failures.

The above classification of failure modes is not restricted to individual instances of failures, but can be used to classify the failure behavior of components, which is called a component's failure semantics [9].

- *Failure semantics*

  A component exhibits a given failure semantic if the probability of failure modes, which are not covered by the failure semantic, is sufficiently low.

If a given component is defined to have ordering failure semantics, then all individual failures of the component should be ordering or sequential failures. The possibility of more severe failures, like timing failures, should be sufficiently low. The failure semantic is a probabilistic specification of the failure modes a component may exhibit, which has to be chosen in relation to the application requirements. In other words, the failure semantics defines the most severe failure mode a component user may have to consider. Fault-tolerant systems are designed with the assumption that any component that fails will do so according to a given failure semantic. If nonetheless, the failure of a component violates the failure semantic, then nothing is guaranteed and the whole system might fail. This consideration leads to the important concept of assumption coverage [10], which is closely related to the definition of failure semantics.

- *Assumption coverage*

  Assumption coverage is defined as the probability that the possible failure modes defined by the failure semantic of a component proves to be true during practical conditions on the fact that the component has failed.

For components defined as having byzantine failures semantics the assumption coverage is always 1.0, i.e., the confidence in the failure semantic is total. This is safe since byzantine failures belong to the most severe failure mode. In all other cases the assumptions of the failure semantics may be violated because the component may show byzantine behavior. Consequently, will assumption coverage (confidence) be less than total (1.0).

The assumption coverage is a critical parameter when designing and reusing reliable components. If the assumptions are relaxed too much in order to achieve good assumption coverage, the design becomes overly complicated since severe failure semantics (like byzantine) have to be considered. On the other hand, if too many assumptions are made, the system design is easier, but the assumption coverage might be unacceptably low. In practice there is thus a need to compromise between system complexity and assumption coverage.

3 Component based analysis

In this section we are going to introduce a framework for formal and probabilistic arguments of component reuse in safety-critical real-time systems. The basic idea is to provide evidence, based on the components contracts and the experience accumulated, that a component can be reused immediately, or if only parts can be reused - or not at all. That is, we want to relate the environment for which the component was originally designed and verified for, with the new environment where it is going to be reused. Depending on the match with respect to input-output domains and temporal domains we want to deem how much of the reliability from the earlier use of the component can be inherited in the new environment. Faced with a non-match we are usually required to re-verify the entire component. However, we would like to make use of the parts that match and only re-verify the non-matching parts.

Specifically, we are now going to introduce two analyses, one with respect to changes in the input-output domain, and one with respect to changes in the temporal domain.

3.1 Input-output domain analysis

We are now going to establish a framework for comparative analysis of components input-output domains, i.e. their respective expected inputs and outputs, as defined by the components contracts (interfaces). But, we are also going to relate the input-output domain to its verified and experienced reliability, according to certain failure semantics. We can represent the input-output domain for a component, c, as a tupel of inputs and outputs; $\Pi(c) \subseteq I(c) \times O(c)$. Where the input domain, $I(c)$, is defined as a tupel of all interfaces and their respective sets of input; $I(c) \subseteq I(c)_1 \cup \ldots \cup I(c)_n$. Further, the output domain, $O(c)$, is defined as a tupel of all interfaces and their respective sets of output; $O(c) \subseteq O(c)_1 \cup \ldots \cup O(c)_n$.

The basic idea is that, given that we have a reliability profile, $R(c)$, of a component's input-output domain, $\Pi(c)$ and that the attributes for the real-time components are fixed, we can deem how well the component would fare in a new environment. That is, we would be able to make comparative analysis of the experienced input-output domain and the domain of the new environment.

For example, assume that we have designed and verified a component, c, that has an input domain $I(c)$ corresponding to a range of integer inputs, $I(c)$ = [40,70], to a certain level of reliability.

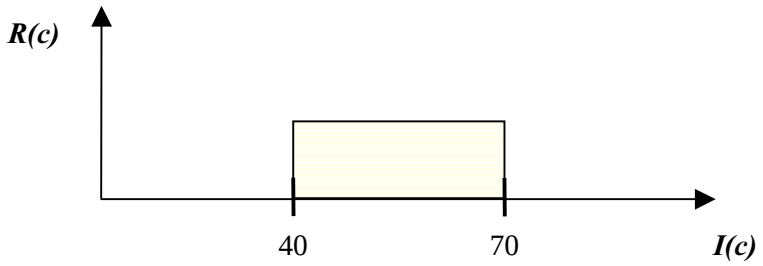


Figure 3 The reliability for input domain [40,70].

Consider now that we reuse this component in another system, where all things are the same except for the input domain, $I_2(c)$ = [50,80], and that we by use and verification have achieved another level of reliability.

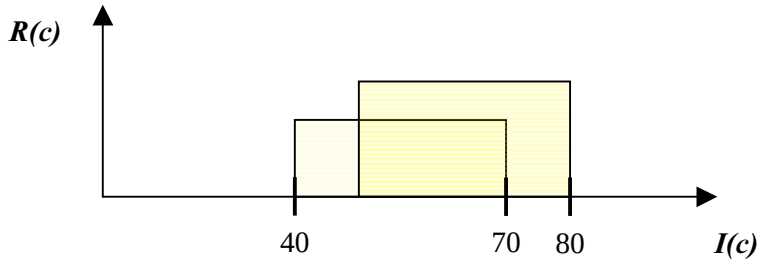


Figure 4 The reliability for joint input domain.

We can now introduce a new relation called the experienced input-output-reliability domain, $E(c) \subseteq (\Pi_1(c) \times R_1(c)) \cup \ldots \cup (\Pi_n(c) \times R_n(c))$, which represent the union of all input-output domains and the achieved reliability for each of these domains. This union is illustrated in Figure 4. Using this $E(c)$ we can deem if we can reuse the component immediately in a new environment $\Pi_{new}(c)$ and inherit the experienced reliability, or we can put another condition on the reuse, a reliability requirement.

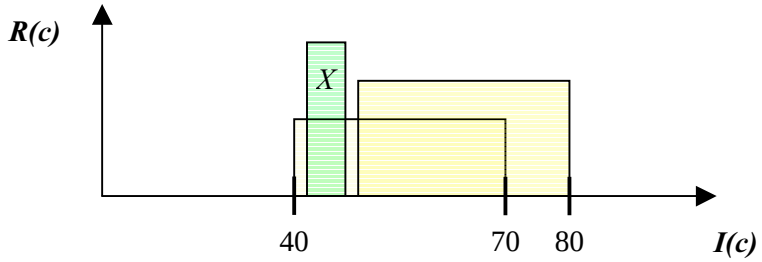


Figure 5 Reliability cannot be guaranteed in new environment.

In Figure 5, the new environment and the area named *X*, illustrates the reliability requirement. If it can be shown that $\Pi_{new}(c) \times R_{new}(c) \subseteq E(c)$ then the component can be reused without re-verification. However if the reliability requirement cannot be satisfied as illustrated in Figure 5, reuse cannot be done without additional verification.

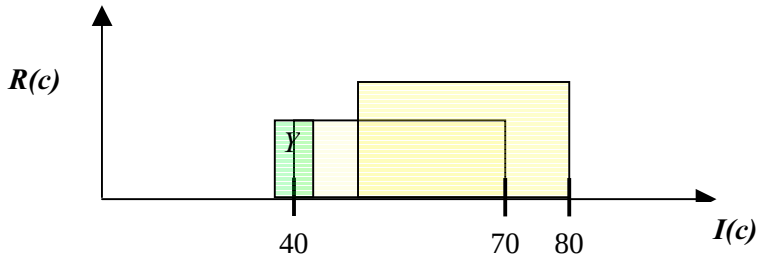


Figure 6 Verification only necessary for limited area.

If the $\Pi_{new}(c) \times R_{new}(c) \subset E(c)$, then we cannot immediately reuse the component without re-verification, however we need not re-verify the component entirely. It is sufficient to verify the part of the input domain that is non-overlapping with the experienced one, i.e., $(\Pi_{new}(c) \times R_{new}(c)) \setminus E(c)$.

By making this classification we can provide arguments for immediate reuse, arguments for re-verification of only cut sets and non-reuse. For example, assume that the $\Pi_{new}(c)$ (the area named *Y* in the Figure 6) has an $I_{new}(c) = [35,45]$ then we can calculate the cut set for $I_{new}(c)$ and $I(c)$ to be $I_{cut}(c) = [35,40]$

3.2 Temporal analysis

Whenever a new component is to be used in a real-time system, the new temporal behavior of the system must be verified, i.e. that all tasks are schedulable. For real-time systems schedulability means that no task in the system will ever violate its timing requirements, for instance deadlines, start times, jitter, etc. In order schedule a system the components have to be decomposed into their smallest possible entities, i.e., their tasks. Thus, the level of abstraction provided by the component concept is violated.

In this section we will discuss the impact of changing some of the temporal attributes for a component when reusing it. The following situations are dealt with:

- The same infrastructure and the same input-output domain
- The same input-output domain but a different infrastructure

A change in the temporal domain is always initiated by the system in which the component will run in. For instance, if the new system executes on different hardware, the execution times might vary (a faster or slower CPU). The changes can also originate from the requirements of the new system, for example the component has to run with a higher frequency than originally designed for.

3.2.1 The same infrastructure and input-output domain

In this case, the infrastructure is exactly the same in the new environment as the one previously used. Consequently, we have an exact match between the sets of services provided (denoted as S below). The services required by the design task model for the component have been satisfied earlier which implicates that they still are satisfied in the new environment. Thus, $S(c) \subseteq S_{old}(infrastructure)$ and $S_{new}(infrastructure) = S_{old}(infrastructure)$ holds, where $S(c)$ is the set of services required by component c.

Moreover, the new input-output domain for the new instance of component c, $\Pi_{new}(c)$, is completely within the verified range of input-output for the component $\Pi(c)$. Formally the following must hold: $\Pi_{new}(c) \subseteq \Pi(c)$.

The only alteration is in one or several of the temporal attributes for the component. Such a change requires the system to undertake a schedulability analysis [2][7][12] where the component is decomposed into its smallest constituents, the tasks. The component can be considered to fit into the new system if the system is schedulable and the relations between the tasks in the component, as well as the tasks in the new system, are not violated.

3.2.2 The same input-output domain but different infrastructure

There exist two different types of infra-structural changes:

- One where the infrastructures are different, but where the infrastructure parts pertaining to the component are the same. More precisely, if $S(c) \subseteq S_{new}(infrastructure) \cap S_{old}(infrastructure)$, then the necessary services are provided by the new infrastructure. If this is the case, and a correct mapping of the services required by the component to the new infrastructure is performed, we can reuse the component with same confidence in its reliability as in the original environment.
- One where the infrastructures are different and the infrastructure parts pertaining to the component are non-compliant. Formally if the $S(c) \setminus S_{new}(infrastructure) \neq \emptyset$.

In the latter case where the new infrastructure does not enable the same implementation of the design task model, a new mapping from the design must be performed. This mapping is a matter of implementing the new infrastructure model using the services provided in the new infrastructure. If this mapping is not possible then we cannot reuse the component at all. If the mapping is possible, we can still argue if this is reuse at all, since major parts of the component must be modified. Here we make a clear distinction between modify and parameterize, where modifications are changes in the source code and parameterizations leave the source code unchanged, but its behavior can be changed dynamically.

As an example, consider a component that has been proven to have certain reliability in an infrastructure that provides synchronization between tasks using time-wise separation. If now the component is reused in an infrastructure that handles synchronization using semaphores the synchronization strategy has to be changed. Consequently, we cannot assume anything about the reliability regarding synchronization failures. We have to assume weaker failure semantics since

the preconditions for which the reliability estimates regarding synchronization failures are no longer valid. That is, we cannot guarantee anything regarding synchronization failures in reusing the component.

The graph below illustrates the different reliabilities for the fault hypotheses (assumptions of failure semantics as introduced in section 2.3), 1 through 5 for component *c*.

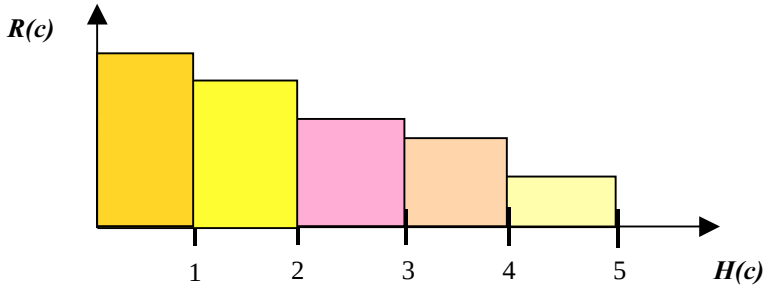


Figure 7 The reliability/fault hypothesis before reuse in a new infrastructure.

If the assumptions made for the reliability measure of fault hypothesis 3 is changed, the reliability is inherited from fault hypothesis 2 (see Figure 8), and we cannot say anything about the reliability regarding stronger failure semantics.

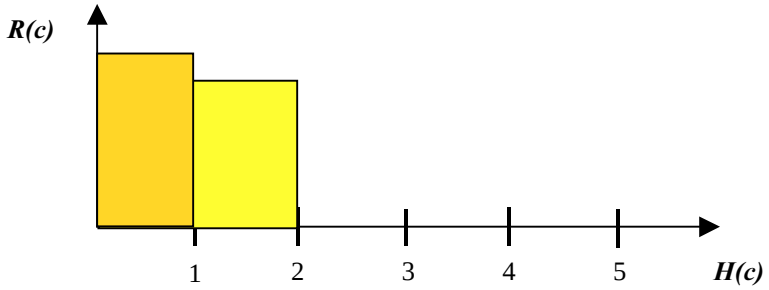


Figure 8 The new reliability/fault hypothesis after reuse in a new infrastructre.

Just as in the previous case where the infrastructure was not changed, the schedulability analysis must be performed all over again, ensuring that no temporal constraints are violated in the component as well in the rest of the system.

3.2.3 A measurement of reusability for components

For every new environment in which a component is successfully reused, the broader is the usability of that component. That is, it has been empirical proven that the component can be used in different environments, and is thus highly reusable. The graph in Figure 9 exemplifies this for changes in the period times of a component. The greater the reliability (denoted as *R),* the more confident we can be that the component will work correctly for that given period time. The greater the number of period times covered by the diagram, the more reusable the component is. However, one can argue that a component that has been reused in a lot of different environments but where every reuse resulted in a low reliability is really reusable? Consequently, the reusability is a combination of the number of environments where the component has been used, and the success of every such reuse.

Such a graph can be generated for every type of attribute in the component contract. For instance, the different types of real-time operating systems for which the component has been reused. In this case, the distribution is also a measurement of the portability for the component.

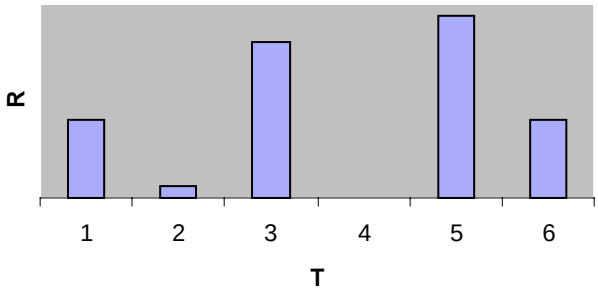


Figure 9 Distribution of period times for which the component has been reused

4 Conclusions

In this paper we have presented a novel framework for arguments of reuse of components in safety-critical real-time systems. In this framework, we formally describe component contracts in term of its temporal constraints given in the design phase (the design task model) and the temporal attributes available in the implementation (the infrastructure). Furthermore, the input-output domain for a component is specified in its contract. By relating the input-output domain, fault hypotheses, probabilistic reliability levels and the temporal behavior of the component, we can deem if a component can be reused or not. We can also deem how much and which subsets, of say input-output domains, that need additional functional and safety verification based on reliability requirements in the environment in which the reuse is intended.

5 References

[1] Alur R. and Dill D. A theory of timed automata, Theoretical Computer Science vol. 126 pp. 183-235, 1994

[2] Audsley N. C., Burns A., Davis R. I., Tindell K. W. *Fixed Priority Pre-emptive Scheduling: A Historical Perspective.* Real-Time Systems, The Int. Journal of Time-Critical Computing Systems, Vol. 8(2/3), March/May, 1995.

[3] Eriksson C., Mäki-Turja J., Post K., Gustafsson M., Gustafsson J., Sandström K., and Brorsson E. *An Overview of RTT: A design Framework for Real-Time Systems.* Journal of Parallel and Distributed Computing, vol. 36, pp. 66-80, Oct. 1996.

[4] H. Kopetz and J. Reisinger The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronization Problem. In Proceedings of he 14th Real-Time Systems Symposium, pp. 131-137, 1993.

[5] J. Chen and A. Burns Asynchronous Data Sharing in Muliprocessor Real-Time Systems Using Process Consensus. 10th Euromicro Workshop on Real-Time Systems, June 1998,

[6] Leveson N. G. *Safeware - System, Safety and Computers*. pp. 414. Addison Wesley 1995. ISBN 0-201-11972-2.

[7] Lui C. L. and Layland J. W.. *Scheduling Algorithms for multiprogramming in a hard real-time environment*. Journal of the ACM 20(1), 1973.

[8] Parnas D.L., van Schouwen J., and Kwan S.P. *Evaluation of Safety-Critical Software*. Communication of the ACM, 6(33):636-648, June 1990.

[9] Poledna S. *Replica Determinism in Distributed Real-Time Systems: A Brief Survey*. Real-Times systems Journal, Kluwer A.P., (6):289-316, 1994.

[10] Powell D. *Failure Mode Assumptions and Assumption Coverage.* In Proc. 22nd International Symposium on Fault-Tolerant Computing. IEEE Computer Society Press, pp.386-395, July 1992.

[11] Puschner P. and Koza C. Calculating the maximum execution time of real-time programs. Journal of Real-time systems, Kluwer A.P., 1(2):159-176, September, 1989.

[12] Xu J. and Parnas D. *Scheduling processes with release times, deadlines, precedence, and exclusion, relations*. IEEE Trans. on Software Eng. vol. 16, pp. 360-369, 1990.

Empty page.

Performance Tuning of Multithreaded Applications for Different Multiprocessor Platforms

Magnus Broberg

Department of Computer Science
University of Karlskrona/Ronneby
Soft Center, S-372 25 Ronneby, Sweden
Magnus.Broberg@ipd.hk-r.se

Abstract

Performance tuning of a parallel application is often hard. The use of standards, such as POSIX threads, makes it possible to move a multithreaded application from one platform to another. Doing performance tuning for many platforms are even tougher since the implementation of the standards may vary on different operating systems. The developer need tools for analysing how the application will behave on different operating systems in order to do adequate performance tuning.

In this paper we present a technique based on cross-simulation that will solve the issues above. The technique uses a monitored execution of a multithreaded application on a single processor workstation running the Solaris operating system. Then the technique, which has been implemented in a tool, simulates a multiprocessor with an arbitrary number of processors running either Solaris or Linux. The tool then displays the behaviour of the application on the selected target configuration.

Validation, using a subset of the SPLASH-2 benchmark suite, shows that the tool predicts speed-ups correctly. The average error in the predicted speed-up when simulating Linux is 5.8%. All this can be done using the ordinary Solaris workstation on the developer's desk, without even having a multiprocessor.

1. Introduction

Writing parallel applications for multiprocessors is often hard. In many cases the reason for using a multiprocessor is to achieve more computing power for the application. Doing performance tuning of a multithreaded application for a multiprocessor is an important but tedious task and few tools are available for the developer. Most tools require the application to be executed on the target multiprocessor, e.g., [5, 7, 9, 18].

When introducing standards, such as POSIX threads [3], the aim is to make applications portable. This makes it possible to move one application from one operating system, e.g. Solaris, to another operating system, e.g. Linux. However, it is not obvious that an application tuned for one operating system will run efficiently on another operating system. Different characteristics for the operating systems may lead to different tuning or trade-offs when tuning the application for both operating systems.

The developer then has to tune the application for several operating systems, using different tools for the different operating systems. The developer must also have access to all combinations of multiprocessors and operating systems in order to do the performance tuning. Not only a tedious task to do, in many cases it is impossible (and expensive) to have all the machines needed.

In this paper we present a tool called VPPB (Visualization of Parallel Program Behaviour). The tool is capable of executing a multithreaded application (using POSIX Threads) on a single processor workstation with Solaris and shows how the execution would be if the application were executed on an SMP (Symmetric MultiProcessor) with an arbitrary number of processors, running

either Solaris or Linux. The predictions are with good accuracy. The tool has perviously been used for prediction on Solaris for Solaris-threads [1, 2, 15]. The tool has now been extended to cover the major parts of POSIX threads as well. The tool can now also mimic the Linux 2.2 operating system. Based on an execution of an application on a single processor Solaris workstation, the tool is able to predict the behaviour of the application on a multiprocessor running Linux. Thus, the tool is able to perform cross-simulation. We use the term cross-simulation when a program is monitored on one operating system and then simulated for another (target) operating system. The performance tuning of the application can be done on the developer's ordinary workstation without the need for a multiprocessor.

The rest of this paper is as follows. In Section 2 we give an overview of the VPPB system and in Section 3 Solaris, Linux, and POSIX threads are briefly discussed. Section 4 shows the validation of the predictions and Section 5 shows that the same application will execute differently on different operating systems. Section 6 discusses the result and points at some future work, the conclusions are found in Section 7.

2. Overview of VPPB

The VPPB consists of three major parts, the *Recorder*, the *Simulator*, and the *Visualizer*. The workflow when using the VPPB system is shown in Figure 1. The developer writes the multithreaded program (a) in Figure 1, compiles it, and an executable binary file is obtained. After that, the program is executed on a uni-processor. When starting the monitored execution (b), the *Recorder* is automatically placed *between* the program and the standard thread library. Every time the program uses the routines in the thread library, the call passes through the *Recorder* (c) which records information about the call, i.e., the identity of the calling thread, the name of the called routine, the time the call was made, and other parameters. The *Recorder* then calls the original routine in the thread library. When the execution of the program finishes all the collected information is stored in a file, the *recorded information* (d). The recording is done without recompilation or relinking of the application, making our approach very flexible.

The *Simulator* simulates a multiprocessor execution. The main input for the simulator is the *recorded information* (d) in Figure 1. The simulator also takes the configuration (e) as input, such as the target operating system, number of processors, etc. The output from the simulator is information describing the predicted execution (f).

Using the *Visualizer* the predicted parallel execution of the program can be inspected (g). The Visualizer uses the simulated execution (f) as input. The main view of the (predicted) execution is a Gant diagram. When visualizing a simulation, it is possible for the developer to use the mouse to click on a certain interesting event, get the source code displayed, and the line making the call that generated the event highlighted. With these facilities the developer may detect problems in the program and can modify the source code (a). Then the developer can re-run the execution to inspect the performance change. The VPPB system is designed to work for C or C++ programs that uses the built-in thread package [7] and POSIX threads [3] on the Solaris 2.X operating system. The Simulator is able to simulate both Solaris and Linux.

3. POSIX Threads and Some Key Differences Between Solaris and Linux

Both Solaris and Linux 2.2 implement the POSIX thread interface [3]. However, there are differences between the implementations. Solaris uses a two-layered approach illustrated in Figure 2 [6, 12]. In the user level there are user level threads. These threads are executed in a non preemptive way. This means that the thread must, in some way, voluntarily give up the execution in favor of another thread. This could be done explicitly or implicitly by calling a synchronization primitive that blocks the thread or releases a previously blocked thread with higher priority.

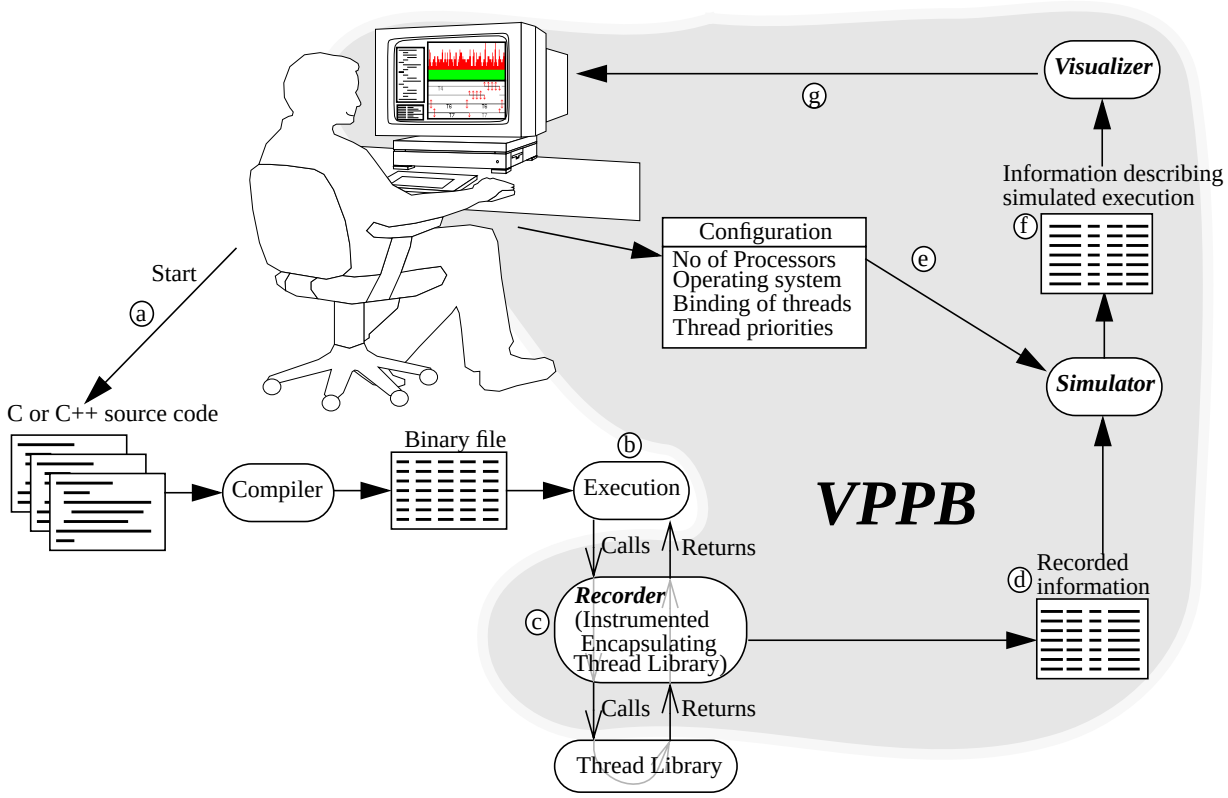


Figure 1: A schematic flowchart of the VPPB system.

The kernel level threads (also known as LightWeight Process, LWP [15]) are scheduled by the kernel in a pre-emptive fashion. The scheduling is priority based, with a round robin policy on each priority level. The priority is changed over time, the longer time an LWP executes the lower priority and longer time slices it gets. Each CPU has it's own run queue. In order to migrate an LWP to another CPU the LWP must have been in the queue for some pre-defined time without being selected by the current CPU. The user level threads are executed by the LWPs. This means that the kernel only schedules LWPs, the user level threads are not seen by the kernel. From the user level threads' point of view the LWPs could be seen as virtual CPUs. A user level thread might be bound to a specific LWP, i.e., the user level thread will (indirectly) be scheduled in a pre-emptive manner. Non-bound user level threads will execute on the LWPs that are not bound to any thread. There could be many LWPs to serve this kind of user level threads.

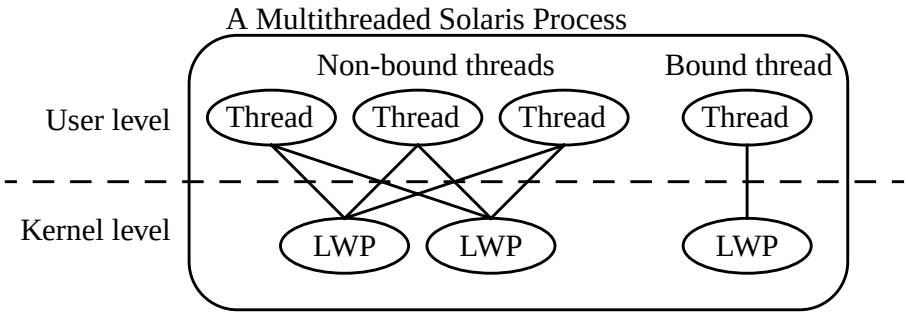


Figure 2: The thread model in Solaris. A bound thread may only execute on the LWP that it is bound to, whereas a non-bound thread may execute on any LWP that is not bound to a user level thread.

Linux [8] uses an single-layered approach, similar to merging the Thread and LWP concepts in Solaris together into one single unit. The creation of threads is made by cloning the parent thread, using the system call clone. This is quite similar to fork, but the cloned threads share the same address space, open files, etc. However, each thread gets its own process id and is scheduled as a process. The scheduling is pre-emptive and time-sliced, as for any other process in Linux. The time-slices are fixed (210 milliseconds). Linux calculates a goodness-value in order to choose between which thread/process to execute next. This goodness-value represents two things, first it represents how long the thread has executed in its current time slice. A thread that has been executed a small part of its time slice will be given a higher goodness value than threads that have executed a longer part of their time slice. The second thing is that threads are favourized to run on the same processor as previously. This is done by increasing the goodness value for the thread when the processor it previously executed on looks for a new thread to execute. Newly awakened threads can force another thread from a processor when the newly awakened thread has a higher goodness value than the other thread.

There are also differences between synchronizations as well. In Linux all locks has ordered queues. When a thread releases the lock, the first thread in the queue is given the lock and put in the running queue. In Solaris, there is no ordered queue (for guaranteed). When a thread releases a lock, it may lock it again, before any other thread (both those waiting in the queue or other) has been able to grab the lock.

4. Validation of the Predicted Execution Times

4.1. The SPLASH-2 benchmark suite

The validation of the predictions was made using a subset of the SPLASH-2 benchmark suite [17]. The applications that we used from the SPLASH-2 suite are listed with the data set in Table 1. All applications that we used are from the scientific and engineering domain.

Table 1: The parallel applications together with the data set sizes we used.

Application	Description	Data set size/Input data
Ocean (contiguous)	Simulate eddy currents in an ocean basin	258-by-258 grid
Water-Spatial	Molecular dynamics simulation, O(N) algorithm	512 molecules, 30 time steps
FFT	1-D $\sqrt{n}$ Six-step Fast Fourier Transform	4M points
Radix	Integer radix sort	16M keys, radix 1024
LU (contiguous)	Blocked LU-decomposition of a dense matrix	768x768 matrix, 16x16 blocks
Raytrace	Producing a raytraced picture	balls4
Barnes	Simulates interaction between a number of bodies in three dimensions	2048 bodies
Cholesky	Factors a sparse matrix into the product of a lower triangular matrix and its transpose	tk29.0
Radiosity	Producing a raytraced picture	Default, batch mode, en 0.1

All executions were made on a Sun Ultra Enterprise 4000 with 8 processors and 512 MByte memory. The operating systems in this validation was Solaris 2.6 and Linux 2.2 (RedHat 6.1). The compiler used for both Solaris and Linux was the egcs-1.1.2 (a.k.a. gcc 2.91.66). Since the SPLASH-2 applications are designed to create one thread per physical processor, one log file

were made for each processor setup when using the Recorder. Thus, 9 applications running each on 4 different CPU setups generated a total of 36 log files. The benchmarks were modified in order to remove spinning locks and task stealing as described in [2].

4.2. Validation results

Table 2 shows the measured and predicted speed-up for the 9 applications from the SPLASH-2 benchmark suit on the Solaris platform. The real speed-up is the middle value of 5 executions of the application. The error is defined as |((Real speed-up) - (Predicted speed-up))/(Real speed-up)|, where $|-x| = |x| = x$, for all $x > 0$.

Due to the recordings, the monitored uni-processor execution takes somewhat longer than an ordinary uni-processor execution of the application. However, our measurements showed that the execution time overhead for doing the recordings was very small. The maximum overhead, which was obtained for Radiosity, was 16.4% of the total execution time, but still more than half of the 36 log files caused less than 0.5% overhead. Another concern was the size of the log files. 75% of the log files were less than 2Mbyte in size. The largest log file, which was obtained for Radiosity, was 20.4 MByte. This file could be handled without any problems. Consequently, neither the execution time overhead, nor the size of the log files caused problems for these applications.

Table 3 shows the measured and predicted speed-up for the 9 applications from the SPLASH-2 benchmark suite on the Linux 2.2 platform. The columns are the same as in Table 2.

The mean error for Solaris was 1.6% while when simulating the Linux platform the mean error was less than 5.8%. Thus, the error on the Linux platform is then on average 3.6 times larger than on the Solaris platform.

4.3. Study of the application with the largest error, Ocean

There are three applications in Table 3 that has large errors in the predictions. These are Ocean, FFT, and Radiosity. Ocean has the largest error and we will focus on that application for this study.

The first thing to notice is the total CPU time needed to execute the application (without the Recorder) behaves quite differently on Solaris and Linux. Since the monitoring is done on Solaris all the overhead, etc., on Solaris is incorporated in the monitoring. Thus, if Solaris behaves differently than Linux, the estimation can be no good. The CPU time needed to execute Ocean is shown as the first row in Table 4. The values are normalized to the case for 1 processor for easy comparison reasons. As can be seen, Solaris increases the needed CPU time with up to twelve percent when executing the 8 threaded Ocean. On the other hand, Linux needs three percent less CPU time to execute Ocean with 8 threads than with one thread. The difference between Solaris and Linux is then 15.5% (1.12 / 0.97).

By increasing the data set for Ocean to 514 and 1026, theses differences decrease as shown in the two last rows in Table 4, to 5.0% and 1.1%, respectively for the case with 8 threads. It is then most likely that the predictions also will be more accurate for Linux as the needed CPU time does not differ. In Table 5 the predictions for Ocean on Linux is shown. As assumed the error in the predictions drops as the data set increases. The measured CPU time can then act as an indicator of the degree of reliability of the prediction.

The measured CPU time may not only act as an indicator, it could also be used to compensate the prediction. In the case of Ocean with data set 258 there is an difference of 15.5% between Solaris and Linux. Thus, the monitored execution on 8 processors was 15.5% longer on Solaris than on Linux and this made the predicted execution for Linux 15.5% longer as well, assuming an even distribution of the overhead over the whole execution. If the predicted execution was 15.5% longer, the predicted speed-up will only be 86.6% (1 / 1.155) compared to the speed-up without the differences in CPU time. Thus, by adding 15.5% more speed-up we are able to compensate for

Table 2: Measured and predicted speed-ups for the benchmark applications on Solaris 2.6.

Application		2 processors	4 processors	8 processors
Ocean	Real Speed-up	1.98	3.67	4.39
	Pred. Speed-up	1.90	3.35	4.35
	Error	4.0%	8.7%	0.9%
Water-spatial	Real Speed-up	1.99	3.93	7.41
	Pred. Speed-up	1.97	3.83	7.24
	Error	1.0%	2.5%	2.3%
FFT	Real Speed-up	1.58	2.22	2.76
	Pred. Speed-up	1.59	2.23	2.80
	Error	0.6%	0.5%	1.4%
Radix	Real Speed-up	1.99	3.96	7.66
	Pred. Speed-up	1.99	3.96	7.79
	Error	0.0%	0.0%	1.7%
LU	Real Speed-up	1.78	3.02	4.45
	Pred. Speed-up	1.78	3.00	4.50
	Error	0.0%	0.7%	1.1%
Raytrace	Real Speed-up	1.88	2.97	4.86
	Pred. Speed-up	1.88	2.94	4.83
	Error	0.0%	1.0%	0.6%
Radiosity	Real Speed-up	1.85	3.60	5.78
	Pred. Speed-up	1.81	3.42	5.89
	Error	2.2%	5.0%	1.9%
Barnes	Real Speed-up	1.94	3.56	6.35
	Pred. Speed-up	1.93	3.57	5.97
	Error	0.5%	0.3%	6.0%
Cholesky	Real Speed-up	1.60	2.30	2.94
	Pred. Speed-up	1.60	2.30	2.90
	Error	0.0%	0.0%	1.4%

the differences in CPU time. The predicted speed-up is 4.35 and with a 15.5% increase it will be 5.02. The latter is much closer the real measured value of 5.77.

The result of calculating in the same way for the other data sets and number of processors is shown in Table 5 within square brackets. The average error in Table 4 has decreased from 7.8% to 5.6% due to this compensation. In three cases the predictions become worse than without compensation, however, those errors are still within the range of errors for the Solaris prediction in Table 2 as well as the errors reported in [2].

Table 3: Measured and predicted speed-ups for the benchmark applications on LINUX 2.2.

Application		2 processors	4 processors	8 processors
Ocean	Real Speed-up	1.99	3.75	5.77
	Pred. Speed-up	1.90	3.35	4.35
	Error	4.5%	10.7%	24.6%
Water-spatial	Real Speed-up	1.99	3.95	7.75
	Pred. Speed-up	1.97	3.83	7.24
	Error	1.0%	3.0%	6.6%
FFT	Real Speed-up	1.71	2.56	3.40
	Pred. Speed-up	1.59	2.23	2.80
	Error	7.0%	12.9%	17.6%
Radix	Real Speed-up	1.98	3.93	7.17
	Pred. Speed-up	1.99	3.96	7.79
	Error	0.5%	0.8%	8.6%
LU	Real Speed-up	1.81	3.11	4.80
	Pred. Speed-up	1.78	3.00	4.50
	Error	1.7%	3.5%	6.2%
Raytrace	Real Speed-up	1.82	2.85	4.41
	Pred. Speed-up	1.88	2.94	4.83
	Error	3.3%	3.2%	9.5%
Radiosity	Real Speed-up	1.85	3.75	5.05
	Pred. Speed-up	1.81	3.42	5.89
	Error	2.2%	8.8%	16.6%
Barnes	Real Speed-up	1.94	3.51	6.03
	Pred. Speed-up	1.93	3.57	5.98
	Error	0.5%	1.7%	0.8%
Cholesky	Real Speed-up	1.60	2.30	2.93
	Pred. Speed-up	1.60	2.30	2.90
	Error	0.0%	0.0%	1.0%

5. Scheduling Differences

When dealing with performance debugging, the speed-up metric does not give all the information needed. If the speed-up is not as the desired speed-up there are some kind of performance bottlenecks in the application. The VPPB system has, as mentioned in Section 2, the ability to show the execution flow as a Gant diagram. This diagram can help the developer understand where the bottlenecks are and how to remove them.

Table 4: Normalized CPU time required to execute the OCEAN benchmark with different data sets on Solaris and Linux.

Data set	Operating System	1 Thread	2 Threads	4 Threads	8 Threads
258	Solaris	1.00	0.99	1.03	1.12
	Linux	1.00	0.96	0.97	0.97
514	Solaris	1.00	1.02	1.04	1.05
	Linux	1.00	1.00	1.01	1.00
1026	Solaris	1.00	1.00	1.00	1.01
	Linux	1.00	1.01	1.00	1.00

Table 5: Ocean on Linux with different data sets. A compensated values are given in square brackets.

Data set		2 Processor	4 Processor	8 Processor
258	Real Speed-up	1.99	3.75	5.77
	Pred. Speed-up	1.90 [1.96]	3.35 [3.56]	4.35 [5.02]
	Error	4.5% [1.5%]	10.7% [5.1%]	24.6% [13.0%]
514	Real Speed-up	1.84	3.67	5.27
	Pred. Speed-up	1.94 [1.98]	3.71 [3.82]	4.51 [4.74]
	Error	5.4% [7.6%]	1.1% [4.1%]	14.4% [10.1%]
1026	Real Speed-up	1.96	3.83	6.93
	Pred. Speed-up	1.90 [1.88]	3.74 [3.74]	6.66 [6.73]
	Error	3.1% [4.1%]	2.3% [2.3%]	3.9% [2.9%]

In Section 4 there is very little difference between the predicted speed-up for the Solaris platform and the Linux platform. Although the speed-up is similar, the execution flow may not be the same. To illustrate that issue Figure 3 shows the same execution segment of the Barnes benchmark. Each horizontal line in the figure represent an executing thread over time. Different symbols indicate different events, e.g., an arrow facing downwards represents a locking operation. Different colours (appear as different gray shades in black and white printing) are used for mutexes, semaphores, etc. Though the execution flow is quite different in Figure 3 the source code is the same. This shows that the execution flow is different when using different operating system, and thus the performance bottlenecks may also differ between the operating systems.

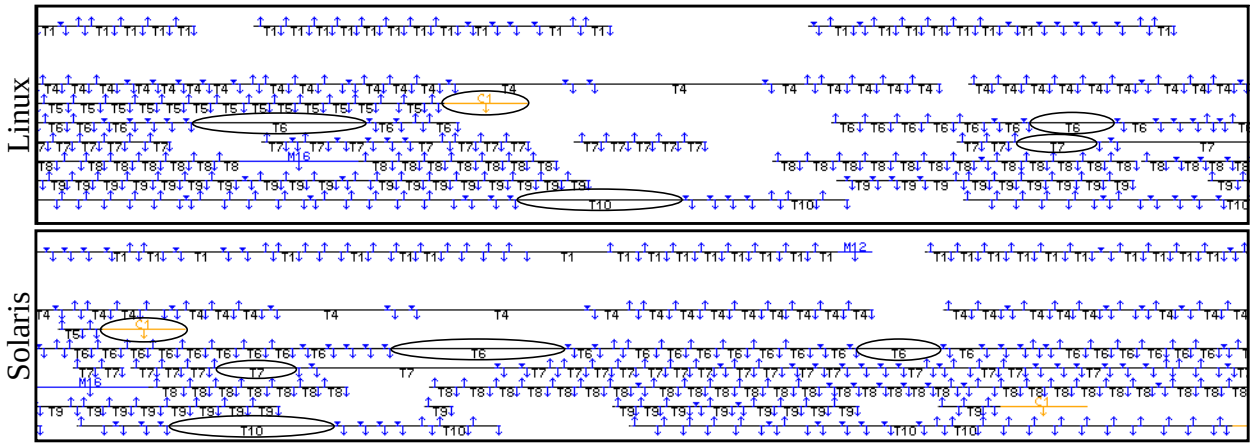


Figure 3: Execution flows for Barnes. Solaris is the upper graph, Linux is the lower. The five ovals in the Solaris graph indicates the same events as the five ovals in the Linux graph.

6. Discussion and Related Work

6.1. Comments on the validation

Validation of the cross-simulation was done on a Sun Enterprise 4000 with 8 CPUs by executing 9 benchmarks from the SPLASH-2 benchmark suite. Only three applications had larger than ten percent error in the predicted speed-up. A further study of the application with the largest error (Ocean) shows that the error is reduced when the data set grows.

As the data set increases the overhead for executing Ocean on Solaris decreases. Since this overhead is included in the monitored data used for the cross-simulation the simulation results for small data sets may not be very accurate. By compensating the predictions with the measured overheads, the error of the predictions was reduced with up to nearly a factor of two.

The reason for the increased usage of CPU time on Solaris, when increasing the number of threads in the Ocean application has not been found. A further study of the reason for this behaviour on Solaris is considered to be future work. Understanding why the error for the benchmarks FFT and Radiosity is almost twice as large as the largest error on Solaris could also be interesting future work.

Currently the VPPB system implements all the POSIX thread primitives that are common between Solaris and Linux except the thread cancelling primitives. The semaphore primitives are supported, although semaphores are not a part of the POSIX threads, but originate from POSIX.1b.

6.2. Other tools

There are a number of tools, shown in Table 6, which make it possible to visualize the (predicted) behaviour of a parallel application using any number of processors. However, these tools are either developed for message passing systems or for non-standard programming environments.

Only a few tools, PARAVER and SIEVE, can do some kind of cross-simulation. The PARAVER tool uses the DIMEMAS simulator [4]. DIMEMAS is a simulator for distributed memory multiprocessors, and the ability to re-configure in order to mimic different operating systems is limited. In [10] there is no validation of the predictions at all. In SIEVE all simulation is performed by scrips working like macros on a spread-sheet, where the spread-sheet is the trace file. By supplying different scripts different operating system can be simulated. The SIEVE system does not address the issue of data monitoring. In [13] there is no validation of the predictions at all.

Table 6: Comparison of some visualization tools similar to VPPB.

Name	Platform	Language	Cross-simulation	Comment	Ref-erence
CHIP^{3}S	Mathematica	CHIP^{3}S	Yes	Pseudo language	[10]
PARAVER	Any PVM3 platform	Any with PVM3	Yes	Message passing	[11]
PERFSIM	TMC CM-5	CM-Fortran	No	Limited visualization	[16]
PIE	VAX 11/780, VAX 11/784, MicroVAX	MP with Pascal	No	Pseudo language	[14]
SIEVE	BBN GP1000	pC++	Yes	Hard to use the scripts	[13]

7. Conclusion

POSIX threads are used to make multithreaded applications portable. However, the implementation of POSIX threads differs for different operating systems. Thus, an application will not always have the same performance and behaviour on different operating systems. Tuning multithreaded applications for a multiprocessor is hard. Tuning applications for good performance on several operating systems is even harder. Most tools use a real execution of the multithreaded application on a given operating system in order to give the developer support in the performance tuning, e.g., [5, 7, 9, 18]. It is often impractical and expensive to have several multiprocessors in order to run different operating systems.

In this paper we have presented a tool, called VPPB, that based on an execution of a multithreaded application on an ordinary single processor Solaris workstation can predict the behaviour of application on a multiprocessor with arbitrary number of processors, running Solaris or Linux 2.2. The predictions are accurate, with an mean error of 5.8% for the predicted speed-ups for Linux, and even better for Solaris. The validation was made by using 9 of the benchmarks from the SPLASH-2 benchmark suite on a Sun Enterprise 4000 with eight processors.

During a detailed study of the Ocean benchmark we have shown that it is possible to find an indicator that shows how reliable the predictions are. The indicator is based on the CPU time required to execute an application with a different number of threads. This indicator works on a single processor workstation with Solaris.

The Ocean benchmark also shown that the indicator can be successfully used to compensate the predictions in order to get more accurate predictions. With Ocean the error in the predictions was reduced with up to almost a factor of two.

Thus, we have shown that it is possible to use the described tool to predict the behaviour of an multithreaded application on a multiprocessor with either Solaris or Linux, with the means of a single processor workstation running Solaris.

8. Acknowledgments

We would like to thank Henrik Persson for rewriting the simulator for the Solaris threads in order to achieve a much faster simulator. This simulator was used as the base when adding ability for POSIX threads as well as support for the Linux platform.

References

[1] M. Broberg, L. Lundberg, and H. Grahn, “VPPB - A Visualization and Performance Prediction Tool for Multithreaded Solaris Programs”, in *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, USA, pp. 770-776, 1998.

[2] M. Broberg, L. Lundberg, and H. Grahn, “Visualization and Performance Prediction of Multithreaded Solaris Programs by Tracing Kernel Threads”, in *Proceedings of the 13th International Parallel Processing Symposium*, San Juan, Puerto Rico, pp. 407-413, 1999.

[3] D. Butenhof, “Programming with POSIX Threads”, Addison-Wesley, 1997, ISBN 0-20-163392-2.

[4] S. Girona, T. Cortes, J. Labarta, V. Pillet, A. Peres, and E. Lopez, “Deriverable OPS4A of the project Basic research APPARC, Effect of short term scheduling on message passing multiprogrammed systems”, http://www.wi.leidenuniv.nl/CS/HPC/apparc-deliverables/OpS4a.html, 1994.

[5] M. Heath and J. Etheridge, “Visualizing the Performance of Parallel Programs,” *IEEE Software 8(9)*, pp. 29-39, 1991.

[6] S. Khanna, M. Sebrée, and J. Zolnowsky, “Realtime Scheduling in SunOS 5.0,” in *Proceedings of the Winter ‘92 USENIX*, June 1992.

[7] S. Kleiman, D. Shah, and B. Smaalders, “Programming with threads,” Prentice Hall, 1996, ISBN 0-13-172389-8.

[8] S. Maxwell, “Linux Core Kernel Commentary,” Coriolis Open Press, 1999, ISBN 1-57610-469-9.

[9] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, “The Paradyn Parallel Performance Measuring Tools,” *IEEE Computer 28, vol 28, no. 11*, pp. 37-46, 1995.

[10] E. Papaefstathiou, D. J. Kerbyson, G. R. Nudd, and T. J. Atherton, “An Overview of the CHIP^{3}S Performance Prediction Toolset for Parallel Systems,” in *Proceedings of 8th ISCA International Conference on Parallel and Distributed Computing Systems*, Florida, USA, pp. 527-533, 1995.

[11] V. Pillet, J. Laboarta, T. Cortes, and S. Girona, “PARAVER: A Tool to visualize and Analyse Parallel Code,” University of Politencia, Catalonia, CEPBA/UPC Report No. RR-95/03, February 1995.

[12] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks, “SunOS 5.0 Multithreaded Architecture,” Sun Soft, Sun Microsystems Inc., September 1991.

[13] S. R. Sarukkai and D. Gannon, “SIEVE: A Performance Debugging Environment for Parallel Programs,” *Journal of Parallel and Distributed Computing*, Vol. 18, pp. 147-168, 1993.

[14] Z. Segall and L. Rudolph, “PIE: A Programming and Instrumentation Environment for Parallel Processing,” *IEEE Software*, 2(6):22-37, November 1985.

[15] SunSoft, “Solaris Multithreaded Programming Guide,” Prentice Hall, 1995, ISBN 0-13-160896-7.

[16] S. Toledo, “PERFSIM: A Tool for Automatic Performance Analysis of Data-Parallel Fortran Programs,” in *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, Virginia, IEEE Computer Society Press, February 1995.

[17] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24-36, June 22-24, 1995.

[18] Q. Zhao and J. Stasko, “Visualizing the Execution of Thread-based Parallel Programs,” *Graphics, Visualization, and Usability Centre, Georgia Institute of Technology, Technical Report GIT-GVU-95-01*, 1995.

Empty page.

Supporting Timing Analysis by Automatic Bounding of Loop Iterations[†]

Christopher Healy
Computer Science Department, Florida State University, FL, USA

Mikael Sjödin
Department of Computer Systems, Uppsala University, Sweden

Viresh Rustagi
Silicon Spice, Mountain View, CA, USA

David Whalley and Robert van Engelen
Computer Science Department, Florida State University, FL, USA

September 23, 1999

Abstract.

Static timing analyzers, which are used to analyze real-time systems, need to know the minimum and maximum number of iterations associated with each loop in a real-time program so accurate timing predictions can be obtained. This paper describes three complementary methods to support timing analysis by bounding the number of loop iterations. First, an algorithm is presented that determines the minimum and maximum number of iterations of loops with multiple exits. Even when the number of iterations cannot be exactly determined, it is desirable to know the lower and upper iteration bounds. Second, when the number of iterations is dependent on unknown values of variables, the user is asked to provide bounds for these variables. These bounds are used to determine the minimum and maximum number of iterations. Specifying the values of variables is less error prone than specifying the number of loop iterations directly. Finally, a method is given to tightly predict the execution time of inner loops whose number of iterations is dependent on counter variables of outer level loops. This is accomplished by formulating the total number of iterations of a loop in terms of summations and solving the resulting equation. These three methods have been successfully integrated in an existing timing analyzer that predicts the performance for optimized code on a machine that exploits caching and pipelining. The result is tighter timing analysis predictions and less work for the user.

Keywords: Hard Real-Time Systems, Worst-Case Execution Time, Static Analysis

1. Introduction

To be able to predict the Best-Case Execution Times (BCETs) and Worst-Case Execution Times (WCETs) of a program, one must know

[†] Part of this work has be previously published in the *Proceedings of IEEE Real-Time Technology and Applications Symposium*, June 1998, under the title "Bounding Loop Iterations for Timing Analysis".

the number of iterations that can be performed by the loops in the program. Under certain conditions, such as a loop with a single exit, many compilers statically determine the exact number of loop iterations (Benitez and Davidson, 1988). Applications for determining this number include loop unrolling (Hennessy and Patterson, 1996), software pipelining (Lam, 1988), and exploiting parallelism across loop iterations (Stone, 1990). When the number of iterations cannot be exactly determined, it is desirable in a real-time system to know lower and upper bounds on the number of iterations. These bounds can be used by a timing analysis tool to more accurately predict BCETs and WCETs.

Many existing timing analyzers require that a user specify the number of iterations of each loop in the program. This specification may be requested interactively (Park and Shaw, 1991; Li et al., 1995). Thus, each time the timing analyzer is invoked for a program, the bounds for every loop in the program must be specified, which is error prone and tedious for the user. Alternatively, one could specify this information as assertions in the source code to prevent repeated specifications of the same information (Burns et al., 1996; Puschner and Koza, 1989; Kligerman and Stoyenko, 1986).

However, there are still several disadvantages. First, the user is still required to write the assertions. Second, there is no guarantee that the user will specify the correct minimum and maximum number of iterations. This problem may easily occur when a user changes the loop, but forgets to update the corresponding assertion. Also, code generation strategies, such as whether to place instructions for the loop exit condition code at the beginning or end of the loop, may cause the number of loop iterations of the transformed loop to vary by one iteration from the loop in the source code. Finally, compiler optimizations, such as loop unrolling or software pipelining, may affect the number of times a loop iterates. Inhibiting different code generation strategies or compiler optimizations to more easily estimate loop bounds would sacrifice performance, which is quite undesirable.

It would be more appropriate to have the compiler automatically and efficiently determine the bounds for each loop in a program when possible. This paper describes three methods that support timing analysis by bounding the number of loop iterations. First, an algorithm is presented that determines bounds on the number of iterations for loops with multiple exits. Second, support is provided for loops whose number of iterations is dependent on loop-invariant variables. Finally, a method is given to accurately predict the number of iterations for inner loops, whose number of iterations varies depending upon the values of counter variables of enclosing outer loops. All three of these methods are efficiently implemented and result in less work for a user. The

last method also results in tighter timing analysis predictions. These methods were implemented by modifying the *vpo* compiler (Benitez and Davidson, 1988) to analyze loops and the information about number of loop iterations is passed to a timing analyzer (Arnold et al., 1994; Healy et al., 1995; White et al., 1997; Healy and Whalley, 1999a) to predict performance. Note that these methods applied in *vpo* could be used in other compilers or on assembly or machine code files as well.

2. Related Work

Previous work to bound the number of loop iterations has used abstract interpretation (Ermedahl and Gustafsson, 1997) and symbolic execution (Lundqvist and Stenström, 1998; Liu and Gomez, 1998) to automatically derive the number of loop iterations. These approaches are quite powerful, but effectively requires simulating all paths of a loop for every loop iteration. Thus, they require significant analysis overhead, which would be undesirable when analyzing long running programs.

Our work on bounding iterations for nested loops was inspired by the work of Sakellariou (Sakellariou, 1997; Sakellariou, 1996). He calculated the total number of iterations for loops that are dependent on counter variables of outer loops in order to obtain better load balance by assigning approximately the same number of loop iterations to each processor. The approach used was to formulate summations representing the number of loop iterations by hand and to interface to a mathematical package off-line to solve the equations. In this paper, we describe a method to automatically calculate the average number of times that a loop will iterate and how to use this information to obtain tighter timing predictions.

3. Bounding Iterations for Loops with Multiple Exits

In this section we present a method to determine a bounded number of iterations for natural loops with multiple exits. The method includes the following steps. (1) First, the conditional branches within the loop that can affect the number of loop iterations are identified. (2) Next, we calculate when each of the identified branches can change its result based on the number of loop iterations performed. (3) Afterwards, the range of loop iterations when each of these branches can be reached is determined. (4) Finally, the minimum and maximum number of iterations for the loop is calculated. These steps are described in the following subsections.

3.1. Identifying the Branches That Can Affect the Number of Loop Iterations

In this subsection some terms are defined to facilitate the presentation of the methods in this paper. A more complete description of these terms can be found elsewhere (Aho et al., 1986). We define the number of loop iterations as the number of times the header is executed once the loop is entered (Arnold et al., 1994). A *basic block* is a sequence of instructions with a single entry point at the beginning and a single exit point at the end. A *natural loop* is a loop with a single entry point. The *header* of a natural loop is the single basic block where the loop is entered. Transitions from within the loop to the header are called *back edges*. Block A *dominates* block B if every path from the initial node of the control flow graph to B has to first go through A. For instance, the header of a natural loop dominates all blocks in the loop. Similarly, block B *postdominates* block A if all control paths from block A to the exit node of the graph contains block B. A block always dominates and postdominates itself.

An *iteration branch* in a loop is a conditional transfer of control, where the choice between the two outgoing transitions can directly or indirectly affect the number of loop iterations. The iteration branches in the loop that can directly affect this number are branches that have (1) a transition to a basic block outside the loop or (2) a transition to the header block of the loop or to a block that is postdominated by the header. Iteration branches that can indirectly affect the number of loop iterations are those branches whose two successors are postdominated by different iteration branches. Figure 1 shows an algorithm to calculate the set of iteration branches I for a loop. The worst-case complexity of the algorithm is $O(B^2I^2)$, where B is the number of basic blocks in the loop and I is the number of iteration branches. However, we believe that the average complexity would be closer to $O(B)$ since iteration branches that indirectly affect the number of loop iterations are not common, particularly in numerical applications.

Figure 2(a) contains the code for a toy example C function that will be used to illustrate the algorithm for calculating loop iteration bounds for loops with multiple exits. Figure 2(b) depicts the RTLs, representing SPARC assembly instructions, that the *vpo* compiler has generated for this function. Note that the relational operator in the conditional branches are sometimes reversed from the relational expressions in the source code. (No delay slots have been filled in order to simplify the example.) Figure 2(c) explains the RTL notation used. The loop consists of basic blocks 2, 3, 5, 6, 7, and 8. The header of the loop is block 7. The algorithm shown in Figure 1 identifies block 5

```
//Find the iteration branches that can directly affect the number of iterations
I = {}
FOR each block B in the loop L DO
    IF (B has two successors S1 and S2) THEN
       IF (S1 ∉ L) OR (S2 ∉ L) OR
          (S1 ∈ PostDom(Header(L))) OR (S2 ∈ PostDom(Header(L))) THEN
            I = I ∪ B
       END IF
    END IF
END FOR

//Find the iteration branches that can indirectly affect the number of iterations
DO
    FOR each block in B in the loop L DO
       IF (B has two successors S1 and S2) AND (B ∉ I) THEN
          IF (there exists J, K ∈ I AND J ≠ K AND
              S1 ∈ PostDom(J) AND S2 ∈ PostDom(K)) THEN
                I = I ∪ B
          END IF
       END IF
    END FOR
WHILE (any change to I)
```

Figure 1. Finding the Set of Iterations Branches for a Loop

as containing an iteration branch since it has a transition to block 6, which is postdominated by the loop header. Blocks 3, 5, and 7 are identified as having iteration branches since they have a transition to block 4, which is not in the loop. Block 2 is added to the set of blocks containing iteration branches since it can indirectly affect the number of iterations by transferring control to either block 3 or block 5, which both have been identified as containing iteration branches.

Once the blocks containing iteration branches for the loop have been identified, a precedence is established that represents the order that these blocks can be executed on any given iteration of the loop. This precedence relationship can be represented as a Directed Acyclic Graph (DAG). The nodes in the DAG represent the blocks containing the iteration branches and two additional nodes, `continue` and `break`. Figure 3 shows the DAG depicting the precedence relationship between the blocks containing exit conditions from Figure 2. The construction of the DAG can conceptually be accomplished by starting with the graph representing the loop, replacing all back edges with

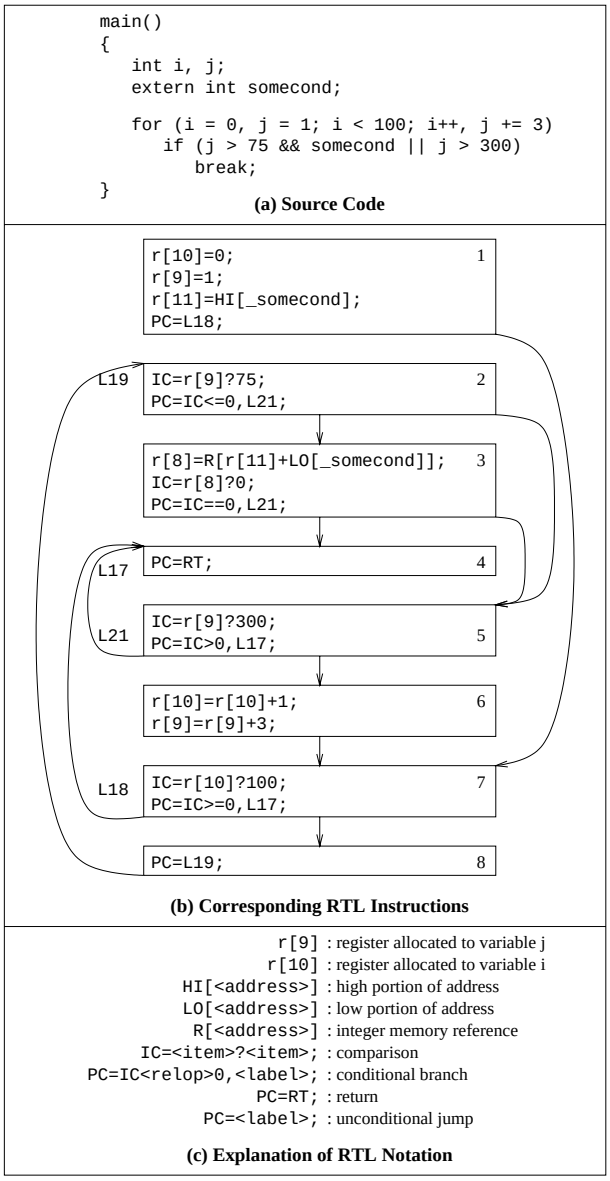


Figure 2. Example Loop with Multiple Exits

transitions to `continue`, replacing each transition out of the loop with a transition to `break`, and collapsing all nodes that do not represent iteration branches. The actual implementation of the DAG construction started with only nodes representing `continue`, `break`, and blocks containing iteration branches and used domination and postdomination information to establish the edges between the nodes. This algorithm

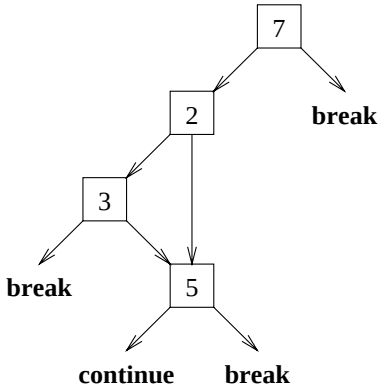


Figure 3. Precedence Relationship between Iteration Branches from Figure 2

is essentially a sort and requires $O(I \log I)$ complexity, where I is the number of iteration branches in the loop.

3.2. Determining When Each Iteration Branch Changes Direction

In this subsection a technique is presented that calculates when each iteration branch can change its result based on the number of loop iterations performed. This technique is similar to those used by other compilers that can calculate the number of iterations of a loop with a single exit (Benitez and Davidson, 1988). For each iteration branch we derive the information shown in Table I. When all of the requirements listed in Table I are satisfied, the iteration branch is classified as *known*. Otherwise, the iteration branch is classified as *unknown*. Note that detection of *unknown* iteration branches in a loop does not mean that the number of iterations of a loop cannot be bounded. Using the derived values, we apply Equation 1 to straightforwardly calculate on which iteration, N_i, that a *known* iteration branch i will change direction. Table II shows the values derived for the example in Figure 2. The iteration branch in block 3 is classified as *unknown* since the *variable* `somecond` is not a basic induction variable. The complexity of this algorithm is $O(I)$, where I is the number of iteration branches, since each iteration branch need only be examined once.

$$N_i = \left\lfloor \frac{limit_i - (initial_i + before_i) + adjust_i}{before_i + after_i} \right\rfloor + 2 \qquad (1)$$

In addition, we have to select a value for *adjust* and checks have to be made in case the iteration branch will always or never be satisfied. Table III shows under what conditions we can use Equation 1 to determine when an iteration branch changes direction. The column "Test

Table I. Information Calculated for Each Iteration Branch

Term	Explanation	Requirement
variable	The control variable on which the branch depends, i.e., the variable being compared in the block containing the iteration branch.	The control variable must be a basic induction variable, that is a variable `v` whose only assignments within the loop are of the form `v := v` $\pm$ `c` where `c` is a constant. In addition, we require that the variable change by a constant integer amount on every loop iteration. We ensure this by checking that each basic block containing an assignment to a basic induction variable dominates all of the blocks containing the tails of the back edge transitions.
limit	The value being compared to the *variable* in the block containing the branch.	The limit must be an integer constant. We will describe how this requirement can be relaxed in Section 4.
relop	The relational operator used to compare the *variable* and the *limit*. I.e., the iteration condition is: "*variable relop limit*".	Our initial description requires that the relational operator be an inequality operator (i.e. `<`, `<=`, `>=`, and `>`). We will describe how to relax this requirement in Section 3.5 to more accurately handle the equality operators (i.e. `==` and `!=`).
initial	The value of the *variable* when the loop is entered.[1]	The initial value must be an integer constant. We will describe how this requirement can be relaxed in Section 4.
before	The amount by which the *variable* is changed before reaching the iteration branch in each iteration.	The amount by which the control variable is incremented or decremented must be an integer constant and these changes must occur on each complete iteration of the loop.[2]
after	The amount by which the *variable* is changed after reaching the iteration branch in each iteration.	The amount by which the control variable is incremented or decremented must be an integer constant and these changes must occur on each complete iteration of the loop.
adjust	An adjustment value of -1, 0, or 1, which compensates for the difference between relational operators (e.g. `<` and `<=`).	

[1] This value is found by searching backwards in the control flow for assignments to *variable*. The search starts with the *preheader*, which is the block outside the loop preceding the loop header.

[2] In other words, the basic blocks containing these changes must dominate every predecessor block of the header that is in the loop.

Table II. Derived Information for Each Iteration Branch in Figure 2

branch	variable	register	limit	relop	initial	before	after	adjust	class	N
block 2	j	r[9]	75	<=	1	0	3	0	known	26
block 3	somecond	r[8]	0	==	N/A	0	0	N/A	unknown	N/A
block 5	j	r[9]	300	>	1	0	3	0	known	101
block 7	i	r[10]	100	>=	0	0	1	-1	known	101

Table III. How to Determine When a Branch with an Inequality Test Changes Direction

Operator	Condition	Test Result	*adjust*
<=	$first \leq limit$ & $incr > 0$	is false on the Nth iteration	0
<=	$first \leq limit$ & $incr \leq 0$	always true	
<=	$first > limit$ & $incr \geq 0$	always false	
<=	$first > limit$ & $incr < 0$	is true on the Nth iteration	1
<	$first < limit$ & $incr > 0$	is false on the Nth iteration	-1
<	$first < limit$ & $incr \leq 0$	always true	
<	$first \geq limit$ & $incr \geq 0$	always false	
<	$first \geq limit$ & $incr < 0$	is true on the Nth iteration	0
>	$first \leq limit$ & $incr > 0$	is true on the Nth iteration	0
>	$first \leq limit$ & $incr \leq 0$	always false	
>	$first > limit$ & $incr \geq 0$	always true	
>	$first > limit$ & $incr < 0$	is false on the Nth iteration	1
>=	$first < limit$ & $incr > 0$	is true on the Nth iteration	-1
>=	$first < limit$ & $incr \leq 0$	always false	
>=	$first \geq limit$ & $incr \geq 0$	always true	
>=	$first \geq limit$ & $incr < 0$	is false on the Nth iteration	0

Where $first = initial + before$, $incr = before + after$,
N is defined in Equation 1, and *adjust* is used in Equation 1.

Result" shows under what conditions we can conclude that the iteration condition will always/never be true and when we should use Equation 1 to determine at which iteration the branch changes direction, as well as, what value to use for *adjust* in Equation 1. Note that when *before+after* $= 0$ we need not use Equation 1 and thus we do not cause a divide by zero exception.

Figure 4 shows two loops where Equation 1 cannot be applied. Our implementation detects that the loop in Figure 4(a) exits after a single iteration. Recall that the number of iterations is the number of times that the loop header block (i.e. testing $i > 100$ in the example) is exe-

cuted once the loop is entered (see Section 3.1). The loop in Figure 4(b) is classified as *unbounded* since the loop may never exit depending on how overflow of negative integer values is handled.

```
for(i = 0; i > 100; i++)
   A;
```
(a) Loop That Exits Immediately

```
for(i = 0; i < 100; i--)
   A;
```
(b) Loop That May Never Exit

Figure 4. Two Loops Which are Handled Without Equation 1

3.3. Determining the Iterations When Each Iteration Branch Can Be Reached

The next step is to determine the iterations on which it is possible to execute each node of the DAG. Calculating these ranges requires $O(I)$ complexity, where I is the number of iteration branches. We record this information as a range of iterations and attach a range to each node and edge. The DAG is processed top-down.

The head of the DAG is assigned the range $[1..\infty]$. All other nodes are assigned a range that is the union of the ranges of all incoming edges. The outgoing edges of a node i are assigned ranges using one of the following two rules:

1. If iteration branch i is *known*, then $relop_i$ and the direction of the increment (i.e. the sign of $before_i$+$after_i$) is used to determine which edge is taken on the first $N_i - 1$ iterations. That edge is assigned the range that is the intersection of $[1..N_i - 1]$ and the range of node i. The other outgoing edge is assigned the range that is the intersection of $[N_i..\infty]$ and the range of node i. If a range assigned to an outgoing edge is empty, then this edge corresponds to an infeasible transition and is deleted from the DAG.

2. If iteration branch i is *unknown*, then both outgoing edges are assigned the same range as node i.

Figure 5 shows the DAG of iteration branches in Figure 3 on page 7 with the range of possible iterations for each node and edge also depicted. Nodes with *known* iteration branches are marked with a **K** and *unknown* iteration branches are marked with a **U**. Iteration branch 7 will take the transition to branch 2 on the first 100 iterations. Note this iteration range of [1..100] corresponds to the variable `i`'s value range of [0..99]. At this point, all values of variables have been abstracted as

ranges of loop iterations. Node 5's transition to a *break* is deleted since the range associated with that transition is empty (i.e. the transition is not possible).

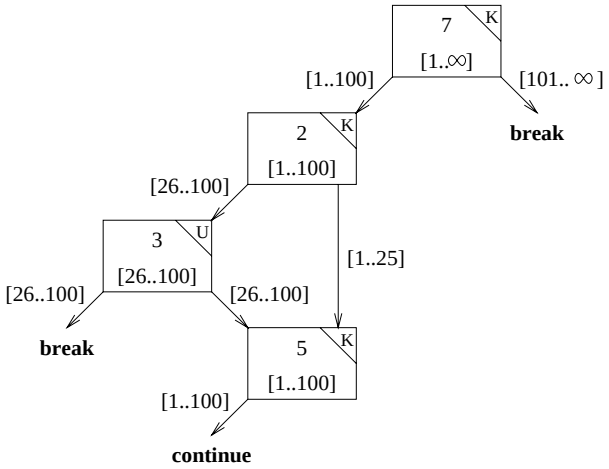


Figure 5. DAG of Branches with Ranges of Iterations

3.4. Determining the Minimum and Maximum Loop Iterations

The ranges of iterations associated with each node and edge of the DAG can be used to calculate the minimum and maximum number of iterations for the loop. To determine the minimum and maximum iteration value for each iteration branch, the DAG is processed in bottom-up order. The algorithm requires $O(I)$ complexity, where I is the number of iteration branches. The minimum and maximum iteration values for the root node of the DAG will be the minimum and maximum iteration values for the entire loop. Figure 6 defines the notation used in this subsection. Note that the range has been calculated using the technique presented in Section 3.3.

The following rules are used to assign minimum and maximum iteration values to edges.

1. If an edge is pointing to a *break*, then both the *edge_exit_min* and *edge_exit_max* are assigned the value of *edge_range_min*. (If there is a transition to a *break*, then the loop can only make that transition once.) This is the only point where a *bounded* value can be introduced since these are the only points where the loop can exit.

2. If an edge is pointing to a *continue*, then the *edge_exit_min* and *edge_exit_max* values for that edge are marked as *unbounded*, which

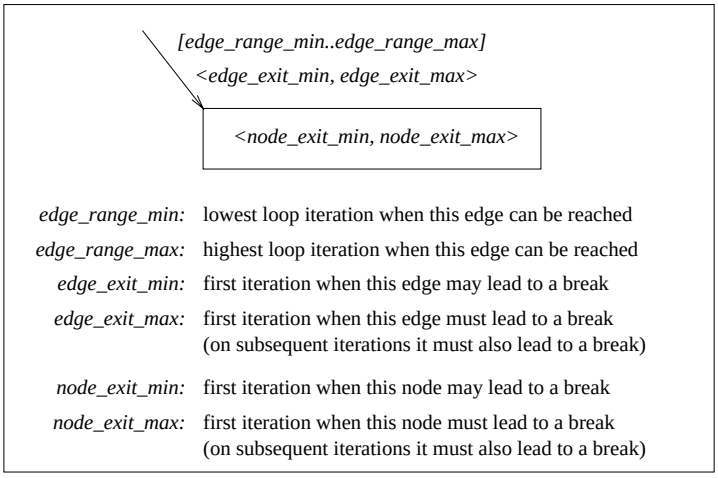


Figure 6. Notation Used in Rules for Assigning Iteration Values

we will represent with '_'. (These transitions do not supply any information about when the loop exits.)

3. If the iteration branch associated with a node is classified as *known*, then the *node_exit_max* for the node is set to the smallest of the *bounded edge_exit_max* values on the outgoing edges or is denoted as *unbounded* if both outgoing edges have *unbounded edge_exit_max* values. (The loop has to exit when it will encounter a *break*.)

4. If the iteration branch associated with a node is classified as *unknown*, then the *node_exit_max* for the node is set to the largest of the *edge_exit_max* values on the outgoing edges of the node or is denoted as *unbounded* if either outgoing edge has an *unbounded edge_exit_max* value. (Use the largest value when it is not guaranteed that the node will actually reach the exit associated with a lower value.)

5. The *node_exit_min* for a node is set to the smallest of the *bounded edge_exit_min* values on the outgoing edges of the node or is denoted as *unbounded* if both outgoing edges have *unbounded edge_exit_min* values. (The smallest value represents the first possibility to exit the loop.)

6. An edge not leading to a *break* or *continue* is an edge leading to a node representing an iteration branch. The *edge_exit_min* and *edge_exit_max* values assigned to the edge depend upon one of three possible relations between the range of the edge and the iteration values of the node. These relations and the corresponding

edge assignments are depicted in Table IV. For example, the edge assignment when *node_exit_min* satisfies case 1 and *node_exit_max* satisfies case 2 would be ⟨*edge_range_min*, *node_exit_max*⟩. Case 1 depicts that the *edge_exit* is set to *edge_range_min* since this is the first iteration the edge can be traversed when the edge may lead to a *break*. Case 2 shows that the *edge_exit* is set to the *node_exit* when it is within the range of iterations that the edge is executed. Case 3 illustrates that the *edge_exit* is set to *unbounded* when there is no iteration on which the edge will be traversed after the edge can lead to a *break*.

Table IV. Rules for Assigning Iteration Values to an Incoming Edge

Case	Condition	Test	Edge_Exit
1	● ——	node_exit < edge_range_min	edge_range_min
2	—●—	edge_range_min ≤ node_exit & node_exit ≤ edge_range_max	node_exit
3	—— ●	edge_range_max < node_exit	_

Legend: —— = [edge_range_min .. edge_range_max]
● = node_exit (i.e. node_exit_min or node_exit_max)

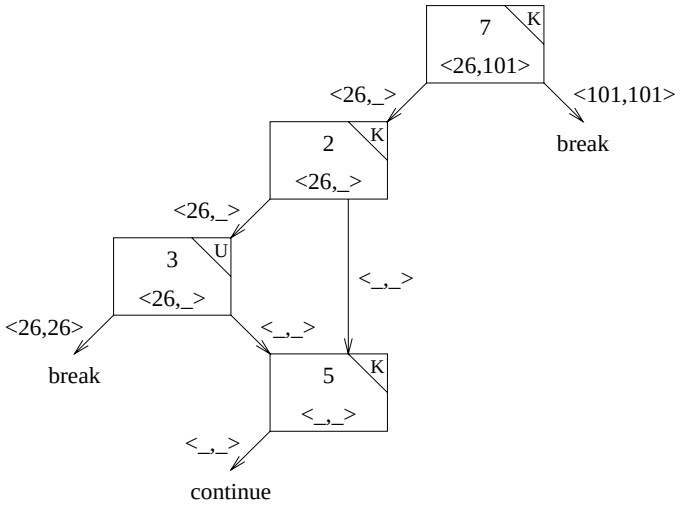


Figure 7. DAG of Iteration Branches with Minimum and Maximum Iteration Values

Figure 7 shows the same DAG as in Figure 5 on page 11, but with minimum and maximum iteration values assigned to edges and nodes. Node 5 and its incoming edges are assigned *unbounded* values since

```
for (i = 0; i != 100; i++)
   A;
```

(a) Bounded Loop

```
for (i = 0; ; i++) {
   if (i < 100 && somecond)
      continue;
   if (i == 50)
      break;
   }
```

(b) Potentially Unbounded Loop

```
for (i = 0; i != 100; i += 3)
   A;
```

(c) Unbounded Loop

Figure 8. Examples of Loops with Iteration Branches Using Equality Operators

there is no transition to a *break* for the range of loop iterations in which they are executed. Node 3 is assigned a minimum iteration value of 26 since that is the first possible iteration at which the node can take a transition to a *break*. Node 3's maximum iteration value is *unbounded* since node 3's iteration branch is classified as *unknown* and there is no guarantee that the transition to the *break* from node 3 will ever be taken. The minimum and maximum iterations for the entire loop is 26 and 101, respectively, since these are the iteration values in node 7, which is the root exit condition.

3.5. Supporting Iteration Branches Using Equality Operators

As stated in Table I on page 8, an iteration branch using an equality operator (i.e. `==` or `!=`) was earlier described as always being treated as an *unknown* branch. This may result in looser, but safe iteration bounds for loops containing equality operators. One reason for not addressing iteration branches that use the equality operators is that they may cause loop iteration ranges to become noncontiguous and would complicate the algorithms for bounding the number of iterations. However, in many cases iteration branches with equality operators can be handled using only contiguous ranges of iterations. For instance, Figure 8(a) contains a loop with an equality operator that our implementation was able to successfully bound. Our implementation classifies iteration branches with equality operators as *known* when the following three additional requirements to those specified in Table I are satisfied:

1. First, every path ending in a back edge in the loop must include the iteration branch with the equality operator. Figure 8(b) shows

Table V. How to Determine When an Equality Test Changes Direction

Operator	Condition	Test Result	*adjust*
==	$first < limit \ \& \ incr > 0$	is true on the Nth iteration	-1
==	$first > limit \ \& \ incr < 0$	is true on the Nth iteration	1
==	$first = limit \ \& \ incr = 0$	always true	
==	$first = limit \ \& \ incr \neq 0$	is false on the 2nd iteration	
==	otherwise	always false	
!=	$first < limit \ \& \ incr > 0$	is false on the Nth iteration	-1
!=	$first > limit \ \& \ incr < 0$	is false on the Nth iteration	1
!=	$first = limit \ \& \ incr = 0$	always false	
!=	$first = limit \ \& \ incr \neq 0$	is true on the 2nd iteration	
!=	otherwise	always true	

Where $first = initial + before$, $incr = before + after$,
N is defined in Equation 1, and *adjust* is used in Equation 1.

an example of a loop that may not execute the test for equality on the iteration in which the loop could exit.

2. Next, one of the outgoing transitions of the iteration branch with an equality operator must be to a *break*.

3. Finally, the following expression, which is part of Equation 1, must result in an integral value.

$$\frac{limit_i - (initial_i + before_i)}{before_i + after_i}$$

In other words, the *variable* must equal the *limit* of the iteration branch on some iteration. Figure 8(c) depicts a situation where the *variable* `i` will be assigned values (0, 3, ... 99, 102, ...) that will skip over the *limit* (100).

When the above requirements are fulfilled we have to check the initial value and increments to the variable, similar to Table III on page 9, and also choose a value for *adjust*. Table V shows when we can use Equation 1 on page 7 to determine on which iteration an equality test will change direction. Note that when $before+after = 0$ we need not use Equation 1 and thus we do not cause a divide by zero exception.

4. Supporting a Non-constant Loop-Invariant Number of Iterations

Sometimes a bounded number of iterations for a loop cannot be determined since the loop exit conditions involve the values of variables. Traditionally, timing analyzers have resolved this problem by requiring a user to specify the maximum number of iterations for a loop interactively (Park and Shaw, 1991; Li et al., 1995) or as an assertion in the source code. (Burns et al., 1996; Puschner and Koza, 1989) Unfortunately, there is no guarantee that the user will specify the correct number of iterations. Compilers may employ different code generation strategies or compiler optimizations that can affect the number of loop iterations. Thus, even an astute user may specify the number of loop iterations incorrectly.

Frequently the variables on which the number of loop iterations depend are loop invariant. In this case, a loop-invariant expression is calculated to represent the number of loop iterations. Essentially, we will still use Equation 1 on page 7, but relax the requirement that the *limit* and *initial* values have to be constants. Figure 9 shows an example function and it's corresponding SPARC RTLs. (Some compiler optimizations, such as loop strength reduction, have not yet been performed to simplify the example.) In this example, the control variable for the loop is `r[13]` and the limit is `r[12]`, which is loop invariant. The block preceding the loop is examined to determine the expression associated with the limit, which is expanded in the following steps:

```
1.  r[12]                   # from instruction 12
2.  r[9]+r[10]              # from instruction 5
3.  r[9]+R[r[10]+LO[_n]]    # from instruction 4
4.  r[9]+R[HI[_n]+LO[_n]]   # from instruction 3
5.  m+n
```

The register `r[9]` has been allocated to the argument `m`, whose value was also passed to the function in the same register. The compiler remembers the register and the blocks where each live range of a local variable or argument is allocated to a register. Thus, the compiler was able to associate the register `r[9]` with the argument `m` and that the memory reference is to the global variable `n`. We use Equation 1 to generate a symbolic expression (containing the local variable `m` and global variable `n`) to represent the number of iterations, as shown in Figure 10.

When the compiler can determine that the number of iterations is non-constant and loop invariant, the loop-invariant expression is passed

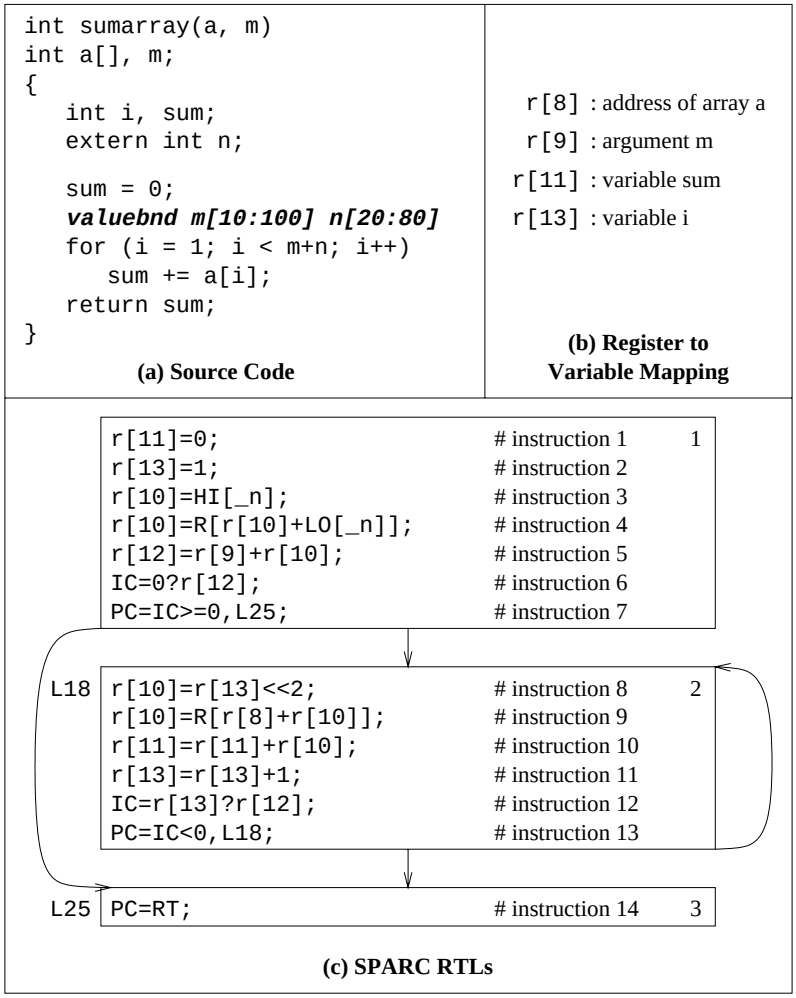


Figure 9. Loop with a Non-constant Loop-Invariant Number of Iterations

to the timing analyzer. The user is prompted by the timing analyzer for the minimum and maximum values for each variable in this expression. To simplify identification of these variables, the timing analyzer also informs the user of the function and line number associated with the loop. After receiving the minimum and maximum values for these variables, the timing analyzer automatically calculates the minimum and maximum number of loop iterations.[3]

[3] Note that the timing analyzer will not permit the number of iterations to be fewer than 1. In the above example, a user may indicate that the minimum values of `m` and `n` are both 0. Simply substituting these values in the expression would result in the number of loop iterations being -1. But if the loop is entered, then it has to

$$
\begin{aligned}
N &= \left\lfloor \frac{limit - (initial + before) + adjust}{before + after} \right\rfloor + 2 \\
&= \left\lfloor \frac{\mathtt{m} + \mathtt{n} - (1 + 1) + -1}{1 + 0} \right\rfloor + 2 \\
&= \mathtt{m} + \mathtt{n} - 1
\end{aligned}
$$

Figure 10. Finding a Symbolic Bound for Example in Figure 9

The authors also modified the compiler to allow the user to specify assertions about the minimum and maximum values of variables associated with loops. The boldface line in Figure 9(a) contains assertions for the minimum and maximum values of the variables `m` and `n`. The compiler uses the loop-invariant expression and replaces the variables with the minimum and maximum specified values. The minimum number of iterations of 29 and the maximum number of iterations of 179 is automatically passed to the timing analyzer and no user intervention is required. Of course, the analysis will only be as accurate as the assertions themselves.

When a loop-invariant expression cannot be calculated, the timing analyzer will prompt the user for the minimum and maximum number of iterations instead of values of variables. However, we have found that a constant or loop-invariant number of iterations can be typically calculated for most loops in the numerical benchmarks and applications we have examined.

5. Bounding Iterations for Non-Rectangular Loop Nests

The previous sections described approaches to determine the minimum and maximum number of iterations for a loop, given that the number of iterations depends only upon either constant or loop-invariant values. Unfortunately, many nested loops do not fulfill this requirement.

In this section we will describe a novel method to determine the number of iterations for a nested loop whose iteration bound depends upon the loop index of an outer loop. Such a loop nest is called *non-rectangular*. A typical example of a non-rectangular loop nest is the loop nest of the bubble sort program in Figure 11.

execute at least one iteration since the number of iterations is defined as the number of times the loop header block is executed.

```
for(i = 0; i < 99; i++)
   for(j = i+1; j < 100; j++)
      if(a[i] > a[j]) swap(a,i,j);
```

Figure 11. A Typical Non-Rectangular Loop Nest

Non-rectangular loop nests have long presented a problem for timing analyzers since the resulting timing predictions are typically quite loose (Healy et al., 1995; Hur et al., 1995; Li et al., 1995). In fact, these overly pessimistic predictions may indicate that a program does not meet its timing constraints, when it actually does.

This section describes a general and efficient method for obtaining tight timing predictions for non-rectangular loop nests usually encountered in programs. This is accomplished by formulating the number of loop iterations in terms of summations, where each summation represents the number of iterations to be executed by a loop. Such an equation can be efficiently solved given that certain restrictions are met.

5.1. Formulating the Number of Iterations

In this subsection we describe how a loop nest may be formulated in terms of summations. The framework we present was based on work by Sakellariou (Sakellariou, 1997; Sakellariou, 1996). The number of iterations of a single loop, where the loop variable is incremented by one (so called *unit stride*), can be represented by a summation when the lower bound (a) is less than or equal to the upper bound (b), as shown in Equation 2.

$$N = \sum_{i=a}^{b} 1 = \begin{cases} b - a + 1 & \text{if } a \leq b \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Figure 12 shows how two different loop nests can be formulated in terms of summations. The total number of iterations to be executed by the innermost loop in each loop nest are calculated by solving the corresponding equation. The Bernoulli formula shown in Equation 3, where $p \geq 1$ and $n \geq 1$ and B_k is the Bernoulli number of order k, can be used to evaluate terms in a summation.

$$\sum_{i=1}^{n} i^p = \frac{1}{p+1} \sum_{k=0}^{p} \binom{p+1}{k} B_k (n+1)^{p-k+1} \tag{3}$$

```
for (i=1; i<99; i++)
    for (j=i+1; j<100; j++)
        A;
```

$$
\begin{aligned}
N &= \sum_{i=1}^{98} \sum_{j=i+1}^{99} 1 \\
&= \sum_{i=1}^{98} \left(\sum_{j=1}^{99} 1 - \sum_{j=1}^{i} 1 \right) \\
&= \sum_{i=1}^{98} (99 - i) \\
&= \sum_{i=1}^{98} 99 - \sum_{i=1}^{99} i \\
&= 4851
\end{aligned}
$$

(a) Loop Nest from Sort Program

```
for (j=1; j<=100; j++)
    for (i=j; i<=100; i++)
        for (k=1; k<j; k++)
            A;
```

$$
\begin{aligned}
N &= \sum_{j=1}^{100} \sum_{i=j}^{100} \sum_{k=1}^{j-1} 1 \\
&= \sum_{j=1}^{100} \sum_{i=j}^{100} (j-1) \\
&= \sum_{j=1}^{100} \left(\sum_{i=1}^{100} (j-1) - \sum_{i=1}^{j-1} (j-1) \right) \\
&= \sum_{j=1}^{100} \left(\sum_{i=1}^{100} j - \sum_{i=1}^{100} 1 \sum_{i=1}^{j-1} j + \sum_{i=1}^{j-1} 1 \right) \\
&= \sum_{j=1}^{100} \left(102j - j^2 - 101 \right) \\
&= 102 \sum_{j=1}^{100} j - \sum_{j=1}^{100} j^2 - 101 \sum_{j=1}^{100} 1 \\
&= 166650
\end{aligned}
$$

(b) Loop Nest from LU Decomposition Program

Figure 12. Deriving the Total Number of Iterations for Two Loop Nests

The constraint on the bounds in Equation 2 results from the fact that the value of the sum must equal 0 if the lower bound a is greater than the upper bound b. The explicit constraint is necessary to accurately count the number of iterations of so-called *zero-trip* loops. Zero-trip loops do not execute the loop body when the lower bound exceeds the upper bound, given that the stride is positive.

We can represent summations with non-unit strides, where the stride s is specified along with the lower bound a and upper bound b. Equation 4 shows how a non-unit stride can be used in a conventional summation, where E is an expression and $E[i \leftarrow si + a]$ denotes the substitution of all free occurrences of i by $si+a$. This is effectively a change in variables and does not change the value of the summation. The change allows summations with strides to be represented by normalized summations

(summations with stride 1).

$$I = \sum_{i=a}^{b,s} E = \sum_{i=0}^{\lfloor (b-a)/s \rfloor} E[i \leftarrow si + a] \tag{4}$$

Summations with non-unit strides are more difficult to evaluate since one has to deal with summations of floors. Equation 5 shows how a floor can be converted to an expression involving a modulo operation (%). A modulo operation can often be simplified using Equation 6 (Sakellariou, 1996).

$$\left\lfloor \frac{n}{m} \right\rfloor = \frac{n - n\%m}{m} \text{ , if } m > 0 \ \& \ n > 0 \tag{5}$$

$$\sum_{i=0}^{n} (i\%d)^p = \begin{cases} \sum_{i=0}^{n} i^p & \text{if } n < d \\ \sum_{j=0}^{\lfloor n/d \rfloor - 1} \sum_{i=0}^{d-1} + i^p \sum_{i=0}^{n\%d} i^p & \text{if } n \geq d \end{cases} \tag{6}$$

However, summations involving modulo operations are more difficult to simplify when two or more loops have non-unit strides and the bounds are symbolic. Fortunately, this situation rarely occurs. Equations 2–6 can be used to correctly determine that the total iterations for the loop nest in Figure 13 is 1717. Unfortunately, sometimes an expression in a summation may contain a product of two or more terms containing modulo operations. In this case, an approximation of the iteration count is used, which is shown in Equation 7.

$$\sum_{i=a}^{b,s} E \approx \sum_{i=a}^{\lfloor b/s \rfloor} E/s \tag{7}$$

```
for (i=0; i<100; i++)
    for (j=i; j<100; j+=3)
        A;
```

Figure 13. A Loop Nest Containing a Non-unit Stride

As suggested by Sakellariou (Sakellariou, 1996; Sakellariou, 1997), a computer algebra system can be exploited off line to solve the equations

of summations. However, computer algebra systems, such as *Maple* (Char et al., 1988), give inaccurate results when the bounds restriction on the summation is violated in Equation 2. In general, every loop iteration count problem that is cast as a summation should evaluate to zero if the lower bound is greater than the upper bound. However, it is not always possible to evaluate the test when the bounds are symbolic. For example, consider the loop nest in Figure 14. The inner loop is a zero-trip loop for values of i greater than 2. We define a *partially zero-trip* loop to be a loop that is zero-trip depending on values of index variables of outer loop(s). By applying Equation 2, the iteration count of the partially zero-trip loop can be defined as shown in Figure 14. Clearly, the result is $N = 3$. However, a naive evaluation without the bounds test results in $N = -7$. This means that when a computer algebra system is to be used off line, the summations should be guarded with bounds tests. Unfortunately, computer algebra systems cannot effectively deal with the simplification of nested summations with additional tests on the bounds of inner summations. The reason is that the test may be symbolic, as shown in Figure 14. The solution is to isolate possible conditions on the iteration variable from the test and to simplify summations as shown in Equation 8 for any expression e. Note that c may not necessarily lie within the range $[a..b]$ and relations besides $<$ may be used.

$$\sum_{i=a}^{b} \begin{cases} E & \text{if } i < c \\ 0 & \text{otherwise} \end{cases} = \begin{cases} \sum_{i=a}^{\min(b,c)} E & \text{if } a < c \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

```
for (i=1; i<8; i++)
    for (j=i; j<3; j++)
        A;
```

$$N = \sum_{i=1}^{7} \begin{cases} 3 - i & \text{if } i < 3 \\ 0 & \text{otherwise} \end{cases}$$

Figure 14. A Partially Zero-Trip Loop

5.2. Implementation

The implementation for evaluating the summations described in the previous section was accomplished by using the algebraic simplifier portion of the Ctadel system (van Engelen et al., 1996; van Engelen et al., 1997). The authors' timing analyzer (Healy et al., 1999) and Ctadel were compiled separately, but Ctadel is directly integrated into the timing analyzer by linking the object files. This avoids unnecessary overhead that would result from passing expressions between the timing analyzer and Ctadel by operating systems calls. The summations are formulated in the timing analyzer and Ctadel is invoked as a C function with the summation parameters as arguments.[4]

Another complication when dealing with zero-trip loops in the timing analyzer is due to the way the timing analyzer counts iterations. As mentioned in Section 3.1, the number of loop iterations is the number of times the loop header is executed, as opposed to the number of times the loop body is encountered. Thus, when a loop is entered, it is guaranteed to iterate at least once. The zero-trip case in Equation 8 can be modified to indicate a single iteration, as shown in Equation 9.

$$\sum_{i=a}^{b} \begin{cases} E & \text{if } i < c \\ 1 & \text{otherwise} \end{cases} = \begin{cases} \sum_{i=a}^{\min(b,c-1)} E & \text{if } a < c \\ 0 & \text{otherwise} \end{cases} + \begin{cases} \sum_{i=\max(a,c)}^{b} 1 & \text{if } c \leq b \\ 0 & \text{otherwise} \end{cases} \quad (9)$$

Figure 15 shows how the loop nest in Figure 14 can be formulated as a summation and solved to produce an accurate number of iterations. Note that the test in Figure 15 has iteration variable i isolated to the left of the relation. An isolation algorithm is used by Ctadel to analyze the test and isolate the variable.

It is known that the detection of zero-trip loops in the general case is NP-complete, because it amounts to solving a integer linear programming problem. Similarly, adjusting the bounds of loops to avoid partially zero-trip loops is NP-complete. This normalization process can be performed with the Fourier-Motzkin (FM) elimination method (Wolfe, 1996). However, one can argue that real-world algorithms rarely exhibit (partially) zero-trip loops, because algorithms with partially zero-trip loops are deemed to be inefficient.

The timing analyzer verifies that there are no zero-trip loops for an inner loop by expanding its initial value and limit. Likewise, the timing

[4] The authors have created a Web page demonstrating the functionality of the Ctadel. It can calculate the number of loop iterations for a loop nest specified by the user. The URL is `http://www.cs.fsu.edu/~engelen/iternum.cgi`.

$$
\begin{aligned}
N &= \sum_{i=1}^{7} \begin{cases} 3-i & \text{if } i < 3 \\ 1 & \text{otherwise} \end{cases} \\
&= \sum_{i=1}^{2} (3-i) + \sum_{i=3}^{7} 1 \\
&= 3 + 5 \\
&= 8
\end{aligned}
$$

Figure 15. Deriving the Number of Iterations for the Loop Nest in Figure 14

analyzer is able to verify that there are no partially zero-trip loops in the loop nest. However, if the verification is inconclusive, the loop nest may or may not contain (partial) zero-trip loops. For instance, consider the loop nest in Figure 16. The expansion of the innermost loop initial value and limit is depicted in Figure 17. The timing analyzer is able to guarantee that the inner loop is not zero-trip since the initial value is never greater than the limit.

```
for (i=0; i<10; i++)
    for (j=i; j<11; j++)
        for (k=i-3; k<j+8; k++)
            A;
```

Figure 16. Innermost Loop Detected Zero-Trip Free by the Timing Analyzer

Initial Value	Limit
$i-3$	$j+8$
$[0..9]-3$	$[i..10]+8$
$[-3..6]$	$[[0..9]..10]+8$
	$[0..10]+8$
	$[8..18]$

Figure 17. Expanding Initial and Limit Values of Innermost Loop in Figure 16

Now consider the loop in Equation 18 and the corresponding expansion of the initial value and limit in Figure 19. The test is inconclusive. However, the loop nest is not zero-trip due to the $j < i$ condition in the middle loop. Since the range analysis can be used to safely verify if a loop is partially zero-trip, it is possible to use the results in deciding which summation solver to use. For example, the loop in Figure 16 can be safely cast into a summation without a bounds tests, while the summations for the loop in Figure 18 requires a bounds test (see Figure 15 for an example bounds test). The disadvantage of having a bounds test is that a loop with a stride poses problems for solving the summation because the summation bounds test may contain modulo operations on the iteration variable, which prohibits the application of Equation 9.

```
for (i=1; i<10; i++)
    for (j=0; j<i; j++)
        for (k=j; k<i; k++)
            A;
```

Figure 18. Innermost Loop Nest Detected Zero-Trip Free by CTADEL

Initial Value	Limit
j	$i-1$
$[0..i]$	$[1..9]-1$
$[0..[1..9]]$	$[0..8]$
$[0..9]$	

Figure 19. Expanding Initial and Limit Values of Innermost Loop in Equation 18

The timing analyzer decides among three possible solution methods to evaluate the summation representing a loop nest:

- CTADEL evaluates the summation while testing the bounds of the index variables.
- CTADEL evaluates the summation without testing for bounds.
- The timing analyzer derives conservative lower and upper bounds on the sum, based on constant bounds given in outer level loops.

The algorithm for selecting the appropriate method is described in Figure 20. The exact solutions are computed using safe assumptions in the possible presence of partially zero-trip loops, using either method (1) or (2). This algorithm will resort to method (3) only in the presence of multiple loops with non-unit strides.

```
The timing analyzer verifies that the loop nest is not (partially) zero-trip.
IF the check is successful THEN
   The loop nest is formulated into summation without bounds tests and
   presented to CTADEL.
   IF CTADEL is able to solve the summation THEN
      RETURN the integer count.
   ELSE
      CTADEL could not solve the summation in the presence of two or more
      loops with non-unit strides.
      RETURN conservative lower and upper bounds on the sum.
   END IF
ELSE
   The check is inconclusive and the loop nest is cast into a summation with
   bounds tests.
   The rewritten summation is presented to CTADEL.
   IF CTADEL is able to solve the summation THEN
      RETURN the integer count.
   ELSE
      CTADEL could not solve the summation in the presence of two or more
      loops with non-unit strides.
      RETURN conservative lower and upper bounds on the sum.
   END IF
END IF
```

Figure 20. Algorithm for Selecting a Solution Method for Summations

The following approach is used in the timing analyzer to obtain tight predictions of non-rectangular loop nests whose total iterations in a loop nest are known. The timing analyzer calculates WCET and BCET predictions based on the maximum and minimum number of iterations, respectively, for the loop whose number of iterations varies. These predictions are made in case a user requests the WCET or BCET predictions for the loop. In addition to these absolute predictions, the timing analyzer also calculates *average* WCET and BCET predictions for each loop. To calculate the average number of iterations for a loop, we divide the total iterations by the total number of times the loop is entered. For instance, in the previous subsection we found that the total number of iterations for the innermost loop from the *sort* program in

Figure 12 on page 20 was 4851. We also calculate the number of times the current loop is entered by calculating the total number of iterations for the loop that encloses the current loop. In this example, the innermost loop is entered 98 times. Thus, the average number of iterations for the loop is 49.5 (4851/98). The average number of iterations is used to calculate the average WCET and BCET predictions. When a non-integer is calculated, we round up for the WCET prediction and truncate for the BCET prediction since our loop analysis algorithm is designed to work on an integral number of iterations.

5.3. Results

Table VI shows programs that were evaluated using the approach of calculating an average number of iterations for loops. These programs benefit from using this approach since they each contain one or more non-rectangular loop nests. Note that the *Sort* program has been used in the past as one of the test programs to evaluate our timing analyzer (Arnold et al., 1994; Healy et al., 1995; White et al., 1997). The size of a program is measured as number of assembly instructions in the compiled and optimized program.

Table VI. Test Programs Containing Non-Rectangular Loop Nests

Name	Description or Emphasis	Size
Hes	Reduces a 100x100 matrix to Hessenberg Form	221
Integ	Evaluates a Double Integral over a Trapezoidal Region	45
Interp	Polynomial Interpolation of 500 Points	178
LU	LU Decomposition of a 100x100 Matrix	278
Sort	Bubble sort of 500 Integers	130
Sym	Tests If a 500x500 Matrix Is Symmetric	50

Table VII shows the best and worst-case cycles required for executing with instruction caching and pipelining for the MicroSPARC I (Texas Instruments, 1993). The *previous ratio* and *current ratio* columns show that when the timing analyzer used the average inner loop predictions, the predicted execution times were significantly tighter. *Interp* showed a significant improvement in best case since the best case number of iterations for the inner loop of a non-rectangular loop nest was 1, which was significantly lower than the average number of iterations. If the timing analyzer did not use an average number of inner loop iterations in worst case, then the number of loop iterations for the triangular loops in *Interp*, *Sort*, and *Sym* would have been approxi-

mately double. The WCET of these programs are nearly exact using the average number of iterations. The *Integ* program had a higher best-case *previous ratio* and a lower worst-case *previous ratio* since there were other loops in this program that contributed more significantly to the total execution time. The *Sort* and *Sym* programs did not have a significant underestimation (i.e. *previous ratio*) in best case. In the best case for *Sort* the values were initially sorted and the sort function exited once the array has been detected to be in ascending order. Likewise, the *Sym* program terminates when it finds the first pair of values that are not equal. *Hes* and *LU* are unlike the other programs in that they contain some triply nested loops. In some loop nests the loop variables of the innermost and middle loops depend on the outermost index variable. In other loop nests the innermost loop variable depends on the loop variable of the middle loop, which in turn depends on the loop variable of the outer loop. CTADEL correctly determines the exact number of loop iterations in all of these cases and the results are more accurate WCET predictions compared to its *previous ratio*s. However, the improvement in BCET for *LU* was less substantial.[5]

Table VIII shows the response time of the timing analyzer for each of the test programs. To obtain these measurements, the timing analyzer was invoked for each test program ten times on a Sun HPC 3000 processor. The figures in the table represent the averages of the ten trials. Note that the times reported here include the analysis of both best and worst case predictions, which occurred in the same invocation of the analyzer. We found that the number of conditional constructs (e.g. *if* statements) rather than the number of loops and functions, tends to have the biggest impact on the analysis time since it affects the number of paths that must be analyzed.

6. Coding Conventions to Make Loop Bounds Predictable

We have found that a programmer can write code where the timing analyzer can accurately determine the number of iterations when the following conventions are used. We do realize that these conventions

[5] The timing predictions for the *Hes* and *LU* programs are still fairly loose. This is primarily due to the fact that several loops were preceded by guards resulting from `if` statements and loop code generation strategies. Each of these guards tests the value of a loop control variable. The authors have recently done work in detecting this type of constraint (Healy and Whalley, 1999b) when dealing with rectangular loop nests. We anticipate to extend this analysis to non-rectangular loop nests for the final version of this paper. This ability to detect constraints on loop control variables should substantially tighten both the WCET and BCET predictions for the *Hes* and *LU* programs.

Table VII. Timing Analysis Results

Best-Case Results					
Name	Observed Cycles	Previous Estimated Cycles	Previous Ratio	Current Estimated Cycles	Current Ratio
Hes	306,341	13,614	0.044	256,516	0.837
Integ	19,160,842	12,785,618	0.667	19,135,118	0.999
Interp	6,485,878	143,064	0.022	6,479,865	0.999
LU	13,792,698	278,683	0.020	637,383	0.046
Sort	19,966	19,950	0.999	19,950	0.999
Sym	160	160	1.000	160	1.000

Worst-Case Results					
Name	Observed Cycles	Previous Estimated Cycles	Previous Ratio	Current Estimated Cycles	Current Ratio
Hes	55,747,317	130,932,770	2.281	57,389,258	1.029
Integ	22,538,082	30,023,163	1.332	22,553,163	1.001
Interp	25,469,403	50,702,358	1.991	25,479,405	1.000
LU	22,436,763	141,900,455	6.324	26,410,255	1.177
Sort	7,672,281	15,251,603	1.988	7,672,292	1.000
Sym	2,747,654	5,481,220	1.995	2,747,698	1.000

cannot always be used for some programs, such as non-numerical applications. However, we believe these conventions can be followed for most numerical applications.

1. When possible, make loop exit conditions only dependent on loop counter variables.
2. Use local integer variables for loop counter variables.
3. Try not to increment or decrement loop counter variables in conditionally executed code.
4. When possible, use integer constants for the initial value, limit, and increments of loop counter variables. Otherwise, try to use loop invariant values.
5. Try to avoid non-unit strides in non-rectangular loop nests.

Table VIII. Analysis Response Times in Seconds

Name	Analysis Time
Hes	0.73
Integ	0.14
Interp	0.36
LU	1.10
Sort	0.25
Sym	0.22
Average	0.47

6. Try to avoid conditionally executed loops in non-rectangular loop nests.

7. Conclusions

In this paper we have presented three different methods for bounding the number of iterations of a loop. First, a method was described that determines the minimum and maximum number of iterations of loops with multiple exits and also detects infeasible paths. For instance, loops of the form in Figure 21(a) that can exit prematurely when some condition becomes true are quite common and the bounded number of iterations of such loops can be detected by the general algorithm presented in the paper.

Second, a method to derive a symbolic expression representing the number of iterations is presented. The symbolic expression is used to bound the number of iterations of loops which have a non-constant number of iterations. Figure 21(b) shows an example of this common type of loop. The user can specify the minimum and maximum values of the variables in the symbolic expression by placing assertions in the source code or by interactively responding to prompts from the timing analyzer. These assertions are more reliable than specifying the minimum and maximum number of loop iterations directly since the user does not have to be aware of the code generation strategies or optimizations performed by the compiler. Also, if value range analysis of variables is deployed the bounds of the variables can be automatically provided by the compiler.

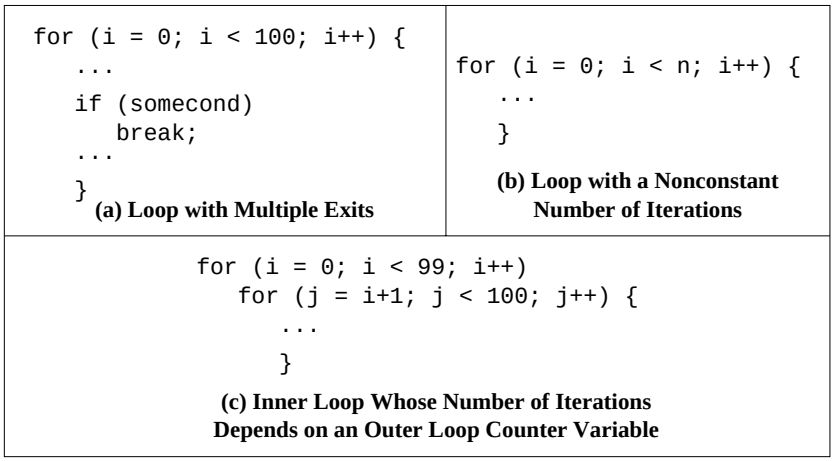


Figure 21. Common Forms of Loops

Finally, timing analysis support is given to tightly predict the execution time of a non-rectangular loop nest, i.e. a loop nest where the number of iterations of an inner loop is dependent on counter variables of outer level loops. These loop nests, such as the one shown in Figure 21(c), appear frequently in programs and can result in significant overestimations in worst-case predictions (as well as underestimations in best-case predictions). Our approach more tightly predicts the number of iterations when the initial value or limit of the control variable in an inner loop depends on a control variable of an enclosing outer loop.

```
IF A loop variable has a non-constant loop-invariant initial value, limit, or stride
   that is not dependent on an outer loop variable AND
   There are no other loop variables to bound the number of loop iterations THEN
      Use information provided by the user (assertions or responses to queries) as
      described in Section 4 to obtain bounds on these variables.
END IF
Calculate the minimum and maximum iterations as described in Section 3.
IF The value of the loop variable is dependent on an outer loop variable THEN
      Calculate an average number of iterations for the loop,
      using the techniques described in Section 5.
END IF
```

Figure 22. Algorithm for Selecting a Solution Method for Bounding Loop Iterations

Figure 22 shows the algorithm used to decide which of the techniques presented in this paper use for a particular loop.

These methods have been successfully integrated in an existing compiler and an associated timing analyzer that predicts the performance for optimized code on a machine that exploits caching and pipelining. The result is tighter and more reliable timing analysis predictions and less work for the user.

Acknowledgements

The authors thank Jack Davidson for allowing *vpo* to be used for this research. Manuel Benitez implemented the original algorithm in *vpo* to calculate the number of iterations of a loop with a single exit condition. Frank Mueller provided several helpful suggestions on an earlier draft of this paper. Rizos Sakellariou provided several suggestions on a later draft. The conversations with Rizos Sakellariou about his work on calculating the total number of loop iterations for a non-rectangular loop nest proved to be invaluable. The authors also appreciate the suggestions from the anonymous reviewers, which measurably improved the quality of the paper. This work was supported in part by NSF grant EIA-9806525.

References

Aho, A. V., R. Sethi, and J. D. Ullman: 1986, *Compilers Principles, Techniques, and Tools*. Addison-Wesley.

Arnold, R., F. Mueller, D. Whalley, and M. Harmon: 1994, 'Bounding Worst-Case Instruction Cache Performance'. In: *Proceedings of the Fifteenth IEEE Real-Time Systems Symposium*. pp. 172–181.

Benitez, M. E. and J. W. Davidson: 1988, 'A Portable Global Optimizer and Linker'. In: *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*. pp. 329–338.

Burns, A., R. Chapman, and A. Wellings: 1996, 'Combining Static Worst-Case Timing Analysis and Program Proof'. *Real-Time Systems Journal* **11**, 145–171.

Char, B., K. Geddes, G. Gonnet, M. Monagan, and S. Watt: 1988, 'MAPLE Reference Manual'.

Ermedahl, A. and J. Gustafsson: 1997, 'Deriving Annotations for Tight Calculation of Execution Time'. In: *Proceedings of European Conference on Parallel Processing*. pp. 1298–1307.

Healy, C., R. Arnold, F. Mueller, D. Whalley, and M. Harmon: 1999, 'Bounding Pipeline and Instruction Cache Performance'. *IEEE Transactions on Computers* **48**(1), 53–70.

Healy, C. A. and D. B. Whalley: 1999a, 'Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints'. In: *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. pp. 79–88.

Healy, C. A. and D. B. Whalley: 1999b, 'Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints'. In: *Proceedings of the IEEE Real-Time Technology and Applications Symposium.*

Healy, C. A., D. B. Whalley, and M. G. Harmon: 1995, 'Integrating the Timing Analysis of Pipelining and Instruction Caching'. In: *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium.* pp. 288–297.

Hennessy, J. and D. Patterson: 1996, *Computer Architecture: A Quantitative Approach, Second Edition.* Morgan Kaufmann.

Hur, Y., Y. Bae, S. Lim, S. Kim, B. Rhee, S. Min, C. Park, M. Lee, H. Shin, and C. Kim: 1995, 'Worst Case Timing Analysis of RISC Processors: R3000/R3010 Case Study'. In: *Proceedings of the IEEE Real-Time Systems Symposium.*

Kligerman, E. and A. Stoyenko: 1986, 'Real-Time Euclid: A Language for Reliable Real-Time Systems'. *IEEE Transactions on Software Engineering* **12**(9), 941–949.

Lam, M.: 1988, 'Software Pipelining: An Effective Scheduling Technique for VLIW Machines'. In: *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation.* pp. 318–328.

Li, Y. S., S. Malik, and A. Wolfe: 1995, 'Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software'. In: *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium.* pp. 298–307.

Liu, Y. and G. Gomez: 1998, 'Automatic Accurate Time-Bound Analysis for High-Level Languages'. In: *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems.* pp. 31–40.

Lundqvist, T. and P. Stenström: 1998, 'Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques'. In: *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems.* pp. 1–15.

Park, C. Y. and A. C. Shaw: 1991, 'Experiments with a Program Timing Tool Based on a Source-Level Timing Schema'. *Computer* **24**(5), 48–57.

Puschner, P. and C. Koza: 1989, 'Calculating the Maximum Execution Time of Real-Time Programs'. *Real-Time Systems* **1**(2), 159–176.

Sakellariou, R.: 1996, 'On the Quest for Perfect Load Balance in Loop-Based Parallel Computations'. Ph.D. thesis, Department of Computer Science, University of Manchester.

Sakellariou, R.: 1997, 'Symbolic Evaluation of Sums for Parallelising Compilers'. In: *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics.* pp. 685–690.

Stone, H. S.: 1990, *High-Performance Computer Architecture, Second Edition.* Addison Wesley.

Texas Instruments, I.: 1993, 'Product Preview of the TMS390S10 Integrated SPARC Processor'.

van Engelen, R., L. Wolters, and G. Cats: 1996, 'Ctadel: A Generator of Multi-Platform High Performance Codes for PDE-based Scientific Applications'. In: *Proceedings of the 10th ACM International Conference on Supercomputing.* pp. 86–93.

van Engelen, R., L. Wolters, and G. Cats: 1997, 'Tomorrow's Weather Forecast: Automatic Code Generation for Atmospheric Modeling'. *IEEE Journal of Computational Science and Engineering* **4**(3), 22–31.

White, R. T., F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon: 1997, 'Timing Analysis for Data Caches and Set-Associative Caches'. In: *Proceedings of the IEEE Real-Time Technology and Applications Symposium.* pp. 192–202.

Wolfe, M. J.: 1996, *High Performance Compilers for Parallel Computers.* Addison-Wesley.

Bounding The Execution Time of Method Calls
Combining Static Analysis and Dynamic Class Loading

Patrik Persson
Department of Computer Science, Lund University
Patrik.Persson@cs.lth.se

Extended Abstract, February 2000

Background

Object-oriented languages are becoming more and more used in real-time software development, but simpler languages such as C or assembly code are more common. One reason is that the timing properties of object-oriented programs are relatively complex to predict. Nevertheless, as increasingly complex embedded systems become ubiquitous in our society, modern languages will be needed to develop their software. Object-orientation is indeed already popular for analysis and design of embedded real-time systems [1]; now we need the language support for it.

More specifically, we address worst-case execution time (WCET) analysis for object-oriented languages. Such analyses are necessary for safe scheduling in real-time systems. We base our approach on the Java language, which has received much attention in the community of embedded real-time systems. Java is compiled to *bytecode*, a stack-based code model for a Java virtual machine (JVM). The JVM is almost always a piece of software which can be implemented on more or less any hardware platform.

Java's bytecode approach was originally taken to support code mobility over the Internet (applets), but has some interesting applications for embedded systems. Java is designed to support dynamic class loading, that is, to load code while the system is running. Such dynamic loading is desirable in some embedded systems, such as industrial robots and telephone exchanges, which often cannot be taken off-line without significant economic penalties.

We want to use Java for embedded real-time systems. While we want to perform static analysis to ensure predictable scheduling, we also want to retain the dynamic class loading. As we will show, these demands at first appear contradictory, but can be combined in a manner consistent with software engineering practice.

Timing Issues in Object-Oriented Languages

Although object-oriented languages (such as Java) share much of their semantics with procedure-oriented languages such as C or Pascal, a few key properties of these languages are of special importance in WCET analysis:

Garbage collection. A traditional garbage collector imposes unpredictable delays in the garbage-collected process, which is clearly intolerable in a hard real-time system. A number of real-time garbage collection techniques have been developed to remedy this problem. These techniques require information about the memory usage of the real-time process in question, something we have treated in our previous work [3].

Virtual method calls. Unlike ordinary function or procedure calls, the code to execute upon a virtual method call is not statically known: it is determined at run-time. Static WCET prediction of such a call requires special techniques.

We will now elaborate on WCET prediction of virtual method calls.

WCET Analysis of Method Calls

An automatic WCET analysis of method calls is actually possible in some cases. If information about all possibly called implementations of a particular methods are available for analysis, one safe WCET estimation is the longest WCET of these implementations.

This approach assumes that *any* of the existing implementations may be called from a given call site. Such an approximation may be improved by using existing type analysis techniques (e.g., [4]) developed for optimizing compilers. A similar approach is taken in [2].

However, a global analysis is not always possible. In particular, Java supports dynamic class loading, which means that new implementations of a given virtual method may be introduced at run-time. To handle such an environment, another approach is required.

Timing Constraints as Method Signature Information

We will now present an approach to handling timing requirements on virtual methods in the context of dynamic class loading. We base this approach on two observations:

- *The WCET of a method call should be known to the programmer implementing that call, even if the implementation is not yet available.* If it is not, the programmer has no way of fulfilling these requirements.

- *When a new class is loaded, it must not break the timing assumptions of the existing system.* A new implementation of a virtual method should not cause any code *calling* that method to miss its deadline.

These requirements are analogous to those for the type system in any statically-typed programming language. For example, assume that the function f accepts a single integer argument. This information is used both for semantic analysis of calls to f (to ensure that a call passes an integer argument) and of f itself (to ensure that it accepts an integer argument).

Timing constraints, such as bounds on WCETs of virtual methods, should be treated in a manner similar to such type information. We thus argue that *timing constraints on virtual methods should be expressed in the method's signature, along with the types of the parameters and the return value.*

We express this information as annotations in the form of "tagged comments". Such comments can be parsed and understood by a WCET analysis tool, yet ignored by a traditional compiler.

Example: An Embedded Controller

To give concrete form to the discussion above, we outline a small example of an embedded controller. The framework allows a method *display* in the operator interface to be called (to display, e.g., the control parameters) upon each execution of the controller. The controller should be possible to use with a variety of operator interfaces, but we want to bound the WCET of *display* to maintain predictability of the controller.

Figure 1 is an outline of how such a design can be expressed in our signature-based approach.

Conclusion

We have presented an approach to managing bounds on WCETs of virtual method calls in the context of dynamic class loading, something that must be handled if object-oriented languages are to be used properly for hard real-time systems. The approach gains from using a specialized class loading mechanism which can check and handle these timing requirements. Such mechanisms are well catered for in the current Java platform (in the form of specialized class loaders).

Our specification-oriented approach is consistent with established software engineering principles, considering the similarity to static typing in programming languages.

```
class PIDController extends RealTimeThread {
   private double uc, y, u, v;
   private GUI myGUI = null;
   ...
   public synchronized void setGUI(GUI g) { myGUI = g; }
   public synchronized GUI getGUI()       { return myGUI; }
   public void run() {
      long t = currentTime();
      for(;;) {
         y  = IO.getY();
         uc = IO.getUc();
         calc_output();
         IO.setU(u);
         update_states();
         GUI g = getGUI();
         if (g != none) g.display(uc, y, u);         // (A)
         t += 100;
         waitUntil(t);
      }
   }
}

class GUI {
   ...
   abstract public void display(double uc,
                                double y,
                                double u)
      /*$ time-bound 25ms */;                        // (B)
}

class SomeGUI extends GUI {
   public void display(double uc,
                       double y,
                       double u) {
      // Real-time stuff goes here.                    (C)
   }
}
```

Figure 1: Expressing WCET bounds for the operator interface in an embedded controller. The call at (A) has a bounded WCET, since the top-level declaration of the called method at (B) has a WCET bound associated with it. The implementation at (C) must adhere to this bound.

References

[1] B. P. Douglass. *Doing Hard Time: Developing Real-Time Systems With UML, Objects, Frameworks, and Patterns.* Addison-Wesley, 1999.

[2] C. Eriksson, J. Mäki-Turja, K. Post, M. Gustafsson, J. Gustafsson, K. Sandström, and E. Brorsson. An Overview of RealTimeTalk: A Design Framework for Real-Time Systems. *Journal of Parallel and Distributed Computing*, Vol. 36, 1996.

[3] P. Persson. Live Memory Analysis for Garbage Collection in Embedded Systems. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, Atlanta, GA, May 1999. ACM SIGPLAN Notices 34(7), pages 45-54.

[4] V. Sundaresan, C. Razafimahefa, R. Vallée-Rai, and L. Hendren. Practical Virtual Method Call Resolution for Java. Sable Technical Report No. 1998-7, McGill University, Canada, 1998.

Empty page.

Obtaining execution time for small program fragments by testing

Markus Lindgren
MRTC, Mälardalens Högskola
markus.lindgren@mdh.se
http://www.idt.mdh.se/personal/mle

Abstract

A novel idea on how execution time of smaller parts of programs can be measured will be presented (work in progress). The advantage of the approach is that it is applicable to many types of processors, without requiring any changes to the method.

The measured time will later be used in high-level analysis to compute the ``WCET'' of the entire program.

Empty page.

Automating Hardware to Software Migration for Real-Time Embedded Systems

Alexander Dean
Carnegie Mellon University

Hardware to software migration helps embedded system designers meet design goals by moving real-time functionality from dedicated hardware to software running on a general purpose CPU. Cutting hardware improves cost, size, weight, component availability, reliability and power consumption. Using software increases flexibility and can cut time to market. The central problem is sharing the CPU efficiently among the multiple threads while ensuring that all real-time operations execute on time.

Currently context-switching and busy-waiting are used to switch threads and schedule operations; these methods slow down the developer and use the processor inefficiently. Tight timing constraints force programming in assembly code, greatly complicating system development. Context switches and busy-wait nops increase program run-time. Despite these drawbacks, design pressures often force developers to manually migrate functions, as evidenced by software implementations of modems, communication controllers, and even video controllers.

This talk describes Software Thread Integration, a compiler-based technique for automatically interleaving multiple real-time software threads into one at the assembly language level, resulting in low-cost or free concurrency. The fine-grain concurrency of integrated threads is ideal for implementing real-time functions which replace dedicated hardware. The compiler follows timing directives to automatically integrate assembly language threads while maintaining semantic and timing correctness. This frees the user to program in a high-level language yet keep the timing accuracy of assembly language. This talk will describe the integration process and results for various systems, including a high-temperature industrial prototype which was built and tested.

Biographical Information:

Alex Dean is finishing his Ph.D. on Software Thread Integration at Carnegie Mellon University's ECE Department. As an engineer in an industrial R&D lab he developed architectures and networks for embedded systems which control automobiles, jet aircraft engines, elevators and HVAC systems.

Updated Thursday, 10-Feb-2000 17:00 by Roland Grönroos
e-mail: artes@docs.uu.se
Location: http://www.docs.uu.se/artes/events/gsconf00/dean.shtml

Empty page.

Fixed-Priority Preemptive Multiprocessor Scheduling: To Partition or not to Partition

Björn Andersson and Jan Jonsson

Department of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
{*ba,janjo*}*@ce.chalmers.se*

Abstract

Traditional multiprocessor real-time scheduling partitions a task set and applies uniprocessor scheduling on each processor. For architectures where the penalty of migration is low, such as uniform-memory access shared-memory multiprocessors, the non-partitioned method becomes a viable alternative. By allowing a task to resume on another processor than the task was preempted on, some task sets can be scheduled where the partitioned method fails.

We address fixed-priority scheduling of periodically arriving tasks on m equally powerful processors having a non-partitioned ready queue. We propose a new priority-assignment scheme for the non-partitioned method. Using an extensive simulation study, we show that the priority-assignment scheme has equivalent performance to the best existing partitioning algorithms, and outperforms existing fixed-priority assignment schemes for the non-partitioned method. We also propose a dispatcher for the non-partitioned method which reduces the number of preemptions to levels below the best partitioning schemes.

1 Introduction

Shared-memory multiprocessor systems have recently made the transition from being resources dedicated for computing-intensive calculations to common-place general-purpose computing facilities. The main reason for this is the increasing commercial availability of such systems. The significant advances in design methods for parallel architectures have resulted in very competitive cost–performance ratios for off-the-shelf multiprocessor systems. Another important factor that has increased the availability of these systems is that they have become relatively easy to program.

Based on current trends [1] one can foresee an increasing demand for computing power in modern real-time applications such as multimedia and virtual-reality servers. Shared-memory multiprocessors constitute a viable remedy for meeting this demand, and their availability thus paves the way for cost-effective, high-performance real-time systems. Naturally, this new application domain introduces a new intriguing research problem of how to take advantage of the available processing power in a multiprocessor system while at the same time account for the real-time constraints of the application.

This paper contributes to solving this problem by addressing the issue of how to utilize processors to execute a set of application tasks in a dynamic operating environment with frequent mode changes. By mode changes we mean that the characteristics of the entire task set changes at certain points in time, for example, because new tasks enter or leave the system (dynamically arriving events), or because the existing tasks need to be re-scheduled with new real-time constraints (QoS negotiation). In particular, this paper elaborates on the problem to decide whether a *partitioned* (all instances of a task should be executed on the same processor) or *non-partitioned* (execution of a task is allowed to be preempted and resume on another processor) method should be used to schedule tasks to processors at run-time. The relevance of this problem is motivated by the fact that many multiprocessor versions of modern operating systems (for example, Solaris or Windows NT) today offer run-time support for both the partitioned and the non-partitioned method.

We study the addressed problem in the context of *preemptive, fixed-priority scheduling*. The reasons for this are the following. First, the fixed-priority scheduling policy is considered to be the most mature (in terms of theoretical framework) of all priority-based scheduling disciplines. As a consequence thereof, most (if not all) existing task partitioning schemes for multiprocessor real-time systems are based on a fixed-priority scheme. Second, most modern operating systems provide support for fixed-priority scheduling as part of their standard configuration, which makes it possible to implement real-time scheduling policies on

these systems.

It has long been claimed by real-time researchers [2, 3, 4, 5, 6] that tasks should be partitioned. The intuition behind this recommendation has been that a partitioning method (i) allows for the use of well-established uniprocessor scheduling techniques and (ii) prevents task sets to be unschedulable with a low utilization. In this paper, we show that the decision of whether to partition or not cannot solely be based on such intuition. In fact, we demonstrate in this paper that the partitioned method is *not the best approach to use in a dynamic operating environment*. To this end, we make two main research contributions.

C1. We propose a new fixed-priority scheme for the non-partitioned method which circumvents many of the problems identified with traditional fixed-priority schemes. We evaluate this new priority-assignment scheme together with the rate-monotonic scheme for the non-partitioned method and compare their performance with that of a set of existing bin-packing-based partitioning algorithms. The evaluation indicates that the new scheme clearly outperforms the non-partitioned rate-monotonic scheme and provides performance at least as good as the best bin-packing-based partitioning algorithms.

C2. We propose a dispatcher for the non-partitioned method which reduces the number of preemptions. The dispatcher combined with our proposed priority-assignment scheme causes the number of preemptions to be less than the number of preemptions generated by the best partitioning schemes.

We evaluate the performance of the scheduling algorithms for multiprocessor real-time systems in the context of *resource-limited* systems where the number of processors is fixed, and use as our performance metrics the success ratio and least system utilization when scheduling a population of randomly-generated task sets. Since existing partitioning algorithms have been proposed to be applied in a slightly different context — minimizing the number of used processors in a system with an unlimited amount of processors — their actual performance on a real multiprocessor system has not been clear.

The rest of this paper is organized as follows. In Section 2, we define our task model and review related work in fixed-priority multiprocessor scheduling. We then present the new priority-assignment scheme in Section 3, and evaluate its performance in Section 4. In Section 5, we propose and evaluate the new context-switch-aware dispatcher. We conclude the paper with a discussion in Section 6 and summarize our contributions in Section 7.

2 Background

We consider a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n independent, periodically-arriving real-time tasks. The characteristics of each task $\tau_i \in \tau$ is described by the pair (T_i, C_i). A task arrives periodically with a period of T_i and with a constant execution time of C_i. Each task has a prescribed deadline, which is the time of the next arrival of the task.

The *utilization* u_i of a task τ_i is $u_i = C_i/T_i$, that is, the ratio of the task's execution time to its period. The utilization U of a task set is the sum of the utilizations of the tasks belonging to that task set, that is, $U = \sum_i C_i/T_i$. Since we consider scheduling on a multiprocessor system, the utilization is not always indicative on the load of the system. This is because the original definition of utilization is a property of the task set only, and does not consider the number of processors. To also reflect the amount of processing capability available, we introduce the concept of *system utilization*, U_s, for a task set on m processors, which is the average utilization of each processor, that is, $U_s = U/m$.

We consider a multiprocessor system with m equally powerful processors. The system uses a run-time mechanism where each task is assigned a unique and fixed priority. There is a ready queue which stores tasks that currently do not execute but are permitted to execute. As soon as a processor becomes idle or a task arrives, a dispatcher selects the next task (the task with the highest priority) in the ready queue and schedules it on a suitable processor.

Recall that the focus of this paper is on preemptive scheduling on a multiprocessor architecture. Multiprocessor real-time scheduling differs from uniprocessor real-time scheduling in that we need to determine not only when a task should execute, but also on which processor to execute. In preemptive uniprocessor scheduling, a task can be preempted by another task, and resume its execution later at a different time on the same processor. In preemptive multiprocessor real-time scheduling, a task that is preempted by another task still needs to resume its execution later at a different time, but the task may resume its execution on a different processor. Based on how the system wants to resume a task's execution, two fundamentally different methods can be used to implement preemptive multiprocessor scheduling, namely, the partitioned method and the non-partitioned method[1].

With the partitioned method, the tasks in the task set are divided in such a way that a task can only execute on one processor. In this case, each processor has its own ready queue and tasks are not allowed to migrate between processors. With the non-partitioned method, all tasks reside in a global ready queue and can be dispatched to any processor. After being preempted, the task can resume its execution on any processor. The principle for the partitioned method and

[1] Some authors refer to the non-partitioned method as "dynamic binding" or "global scheduling".

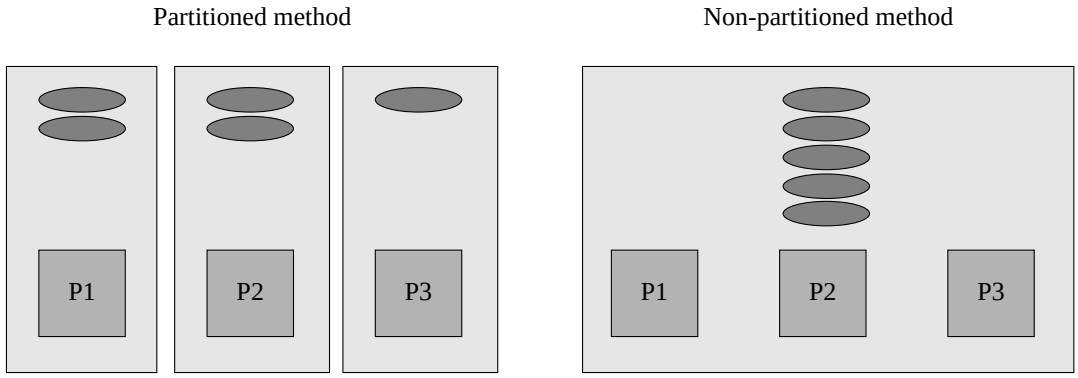


Figure 1: With the partitioned method a task can only execute on one processor. With the non-partitioned method a task can execute on any processor.

the non-partitioned method is illustrated in Figure 1.

For fixed-priority scheduling of periodically-arriving tasks on a multiprocessor system, both the partitioned method and the non-partitioned method have been addressed in previous research. Important properties of the real-time multiprocessor scheduling problem were presented in a seminal paper by Leung and Whitehead [7]. For example, they showed that the problem of deciding if a task set is schedulable (that is, all tasks will meet their deadlines at run-time) is NP-hard for both the partitioned method and the non-partitioned method. They also observed that the two methods are not comparable in effectiveness in the sense that there are task sets which are schedulable with an optimal priority assignment with the non-partitioned method, but cannot be scheduled with an optimal partitioning algorithm and conversely.

When comparing the partitioned method and the non-partitioned method, other researchers have listed the following disadvantages for the non-partitioned method. First, a task set may not be schedulable on multiple processors even though it has a system utilization that approaches zero [2, 3, 4, 5]. Second, the plethora of well-known techniques for uniprocessor scheduling algorithms (such as shared resource protocols), cannot be used [8]. Third, the run-time overhead is greater because the assignment of a processor to a task needs to be done each time the task arrives or resumes its execution after being preempted [9].

Among the two methods, the partitioned method has received the most attention in the research literature. The main reason for this is probably that the partitioned method can easily be used to guarantee run-time performance (in terms of schedulability). By using a uniprocessor schedulability test as the admission condition when adding a new task to a processor, all tasks will meet their deadlines at run-time. Now, recall that the partitioned method requires that the task set has been divided into several partitions, each having its own dedicated processor. Since an optimal solution to the problem of partitioning the tasks is computationally intractable, many heuristics for partitioning have been proposed, a majority of which are versions of the bin-packing algorithm[2] [2, 3, 4, 5, 8, 10, 11, 6, 9]. All of these bin-packing-based partitioning algorithms provide performance guarantees, they all exhibit fairly good average-case performance, and they can all be applied in polynomial time (using sufficient schedulability tests).

The non-partitioned method has received considerably less attention, mainly because of the disadvantages listed above, but also because of the following two reasons. First, no effective optimal priority-assignment scheme has been found for the non-partitioned method. It is a well-known fact that the rate-monotonic priority assignment scheme proposed by Liu and Layland [12] is an optimal priority assignment for the uniprocessor case. Unfortunately, it has been shown that the rate-monotonic priority-assignment scheme is no longer optimal for the non-partitioned method [3, 7]. Second, no effective schedulability tests exist for the non-partitioned method. Recall that, for the partitioned method, existing uniprocessor schedulability tests can be used. The only known exact (necessary and sufficient) schedulability test for the non-partitioned method has an exponential time-complexity [13]. Liu has proposed a sufficient schedulability test for the rate-monotonic priority-assignment scheme [14], which, unfortunately, becomes very pessimistic when the number of tasks increases. Recently, another sufficient schedulability test for the rate-monotonic priority-assignment scheme was proposed inde-

[2]The bin-packing algorithm works as follows: (1) sort the tasks according to some criterion; (2) select the first task and an arbitrary processor; (3) attempt to assign the selected task to the selected processor by applying a schedulability test for the processor; (4) if the schedulability test fails, select the next available processor; if it succeeds, select the next task; (5) goto step 3.

pendently by three research groups [15, 16, 17]. Unfortunately, this schedulability test has the disadvantage of becoming pessimistic for task sets running on a large number of processors [15, 16]. These schedulability tests all have a polynomial (for [17], a pseudo-polynomial) time-complexity. These research groups also found that the condition for a critical instant in uniprocessor scheduling cannot be applied in multiprocessor scheduling. This is an important observation since it indicates that completely new approaches must be used to devise a schedulability test for the non-partitioned method.

Since there currently exist no efficient[3] schedulability tests for non-partitioned method, it is tempting to believe that the non-partitioned method is inappropriate for real-time systems. However, our study of the non-partitioned method for real-time systems is motivated by the following two reasons. First, many real-time systems do not rely on a schedulability test, for example, those which employ feedback control scheduling [18, 19, 20] or QoS negotiation techniques [21]. Second, even if a real-time system does rely on a schedulability test, we believe that in developing better priority-assignment schemes, we may pave the way for effective schedulability tests in the future.

To focus on the core problem, we make the following assumptions in the remainder of this paper:

A1. Tasks require no other resources than the processors. This implies that tasks do not contend for an interconnect, such as a bus, or critical sections.

A2. Tasks are synchronous in the sense that their initial arrivals occur at the same instant of time and then the tasks arrive periodically.

A3. A task can always be preempted. In practice, though, one has to be aware of the fact that operating systems typically use tick-driven scheduling where the ready queue can only be inspected at a maximum rate.

A4. The cost of preemption is zero. We will use this assumption even if a task is resumed on another processor than the task was originally preempted on. In a real system, though, the cost of context switch is typically in the order of $100\ \mu s$ [22].

3 Priority assignment

One of our major contributions in this paper is a new priority-assignment scheme for the non-partitioned method. In this section, we begin by recapitulate some known results concerning priority-assignment schemes for multiprocessor scheduling. We then proceed to motivate our new priority-assignment scheme and discuss how it can be optimized for the non-partitioned method.

While the partitioned method relies on well-known optimal uniprocessor priority-assignment schemes, such as the rate-monotonic scheme, it is not clear as to what priority-assignment scheme should be used for the non-partitioned method. To that end, Leung [13], and later Sáez *et al.* [9], evaluated dynamic-priority schemes and showed that the least-laxity-first strategy performs the best for the non-partitioned case. In the case of fixed-priority scheduling, adopting the idea used by Audsley [23] and Baruah [24] (testing for lowest priority viability) is unsuitable because existing schedulability tests for non-partitioned fixed-priority scheduling are too pessimistic. The only known results report that the rate-monotonic priority assignment scheme does not work well for the non-partitioned method. This is because, for rate-monotonic scheduling on a uniprocessor, a sufficient (but not necessary) condition for schedulability of a task set is that the utilization of the task set is less than or equal to $\ln 2$. For non-partitioned multiprocessor scheduling using the rate-monotonic priority assignment, no such property exists. Originally presented by Dhall [2, 3] (and later repeated in, for example, [4, 5, 7]), the basic argument is as follows. Assume that the task set $(T_1 = 1, C_1 = 2\epsilon), (T_2 = 1, C_2 = 2\epsilon), \ldots, (T_m = 1, C_m = 2\epsilon), (T_{m+1} = 1 + \epsilon, C_{m+1} = 1)$ should be scheduled using the rate-monotonic priority assignment scheme on m processors. The situation for $m = 3$ is shown in Figure 2. In this case, τ_{m+1} will have the lowest priority and will only be scheduled after all other tasks have executed in parallel. The task set is unschedulable and as $\epsilon \to 0$, the utilization becomes $U = 1$ no matter how many processors are used. Another way of formulating this is that the system utilization $U_s = U/m$ will decrease towards zero as m increases. We will refer to this observation as *Dhall's effect*.

Now, we observe that if τ_{m+1} could somehow be assigned a higher priority, the given task set would be schedulable. To see this, simply assign task priorities according to the difference between period and execution time of each task. Then, τ_{m+1} would be assigned the highest priority, and the task set would still be schedulable even if the execution time of any single task would increase slightly.

The fundamental problem demonstrated with the example above can be summarized as follows. In fixed-priority scheduling using the rate-monotonic scheme, many tasks with short period, but with relatively (with respect to period) short execution time, can block the available computing resources so that tasks with longer period, but with relatively long execution time, will miss their deadlines. This indicates that a more appropriate strategy to assign priorities for the non-partitioned method would be to reflect both time criticality and resource demands of each task.

[3] By "efficient" we mean that the schedulability test can be done in polynomial time (as a function of tasks, not task invocations) and that the schedulability test always deems task sets as schedulable if the task set has a utilization which is less than a fixed ratio of the number of processors.

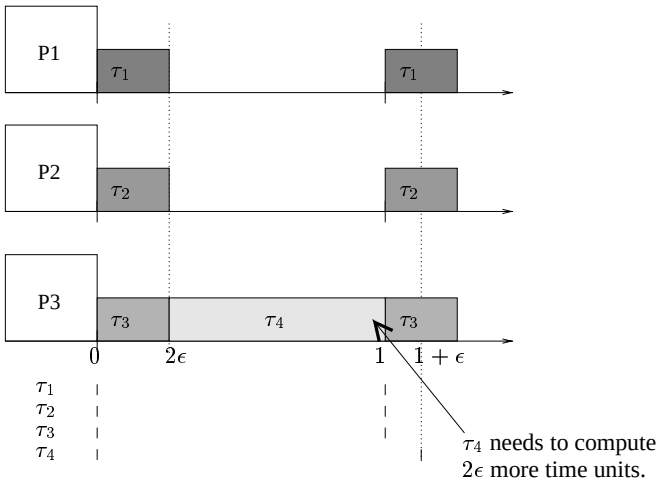


Figure 2: If the rate-monotonic priority assignment for the non-partitioned method is used, there are unschedulable task sets with a low utilization.

An example of such a strategy would be one that prioritizes tasks according to the difference between the period of a task and its execution time[4]. Based on this, we now propose a new priority assignment scheme, called *TkC*, where the priority of task τ_i is assigned according to the weighted difference between its period and its execution time, that is, $T_i - k \cdot C_i$, where k is a global *slack factor*. Note that, by using this model, we can represent two traditionally used priority assignment schemes, namely the rate monotonic (when $k = 0$) and the slack monotonic (when $k = 1$) schemes. Below, we will reason about what k should be selected to contribute to the best performance for the non-partitioned method.

3.1 Reasoning about k

It is clear from the discussion above that, in order to escape from Dhall's effect, we should select $k > 0$[5]. The remaining question is then what values of k yield the best performance. Even for all $0 < k \leq 1$, something similar to Dhall's effect can occur. Assume that the task set $(T_1 = 1, C_1 = \frac{1}{L}), (T_2 = 1, C_2 = \frac{1}{L}), \ldots, (T_m = 1, C_m = \frac{1}{L}), (T_{m+1} = L^2 + \frac{1}{L}, C_{m+1} = L^2 - \frac{1}{2} \cdot L)$ should be scheduled using the TkC priority-assignment scheme with $0 < k \leq 1$ on m processors. The situation for $m = 3$ is shown in Figure 3. Now, as $L \to \infty$, the task set is unschedulable with a utilization $U = 1$. On the other hand, selecting too large a k is a bad idea since task priorities will then be selected such that the tasks with the longest execution time obtains the highest priority. Assume that the task set $(T_1 = 1, C_1 = \epsilon), (T_2 = 1, C_2 = \epsilon), \ldots, (T_m = 1, C_m = \epsilon), (T_{m+1} = \epsilon, C_{m+1} = \epsilon^2)$ should be scheduled using the TkC priority-assignment scheme with $k \to \infty$ on m processors. The situation for $m = 3$ is illustrated in Figure 4. Now, as $\epsilon \to 0$, this task set is unschedulable with a utilization $U = 0$ no matter how many processors are used. Of course, the system utilization $U_s = U/m$ will also decrease toward 0 as m increases. Consequently, this effect is even worse than Dhall's effect.

In conclusion, we observe that k should be greater than 1, but not too large. Lauzac *et al.* [16] evaluated the performance of the non-partitioned fixed-priority scheduling. Unfortunately, they only considered $k = 0$ (that is, the rate-monotonic scheme), which clearly is not the best k. In Section 4.2, we simulate scheduling with different values of k to determine which k is best. In Section 4.3, we show that the non-partitioned method using the TkC priority-assignment scheme outperforms all existing fixed-priority schemes for the non-partitioned method and performs at least as good as the best practical online partitioning schemes.

4 Performance evaluation

In this section, we will conduct a performance evaluation of the non-partitioned and partitioned method. Our evaluation methodology is based on simulation experiments using randomly-generated task sets. The reasons for this are that (i) simulation using synthetic task sets more easily reveal the average-case performance and robustness of a scheduling algorithm than can be achieved by scheduling a single application benchmark, and (ii) with the use of simulation we can compare the best k from simulation with the derived interval of k from our reasoning in Section 3.1.

[4]This is what is typically referred to as the *slack* of the task.

[5]Selecting $k < 0$ can also cause Dhall's effect.

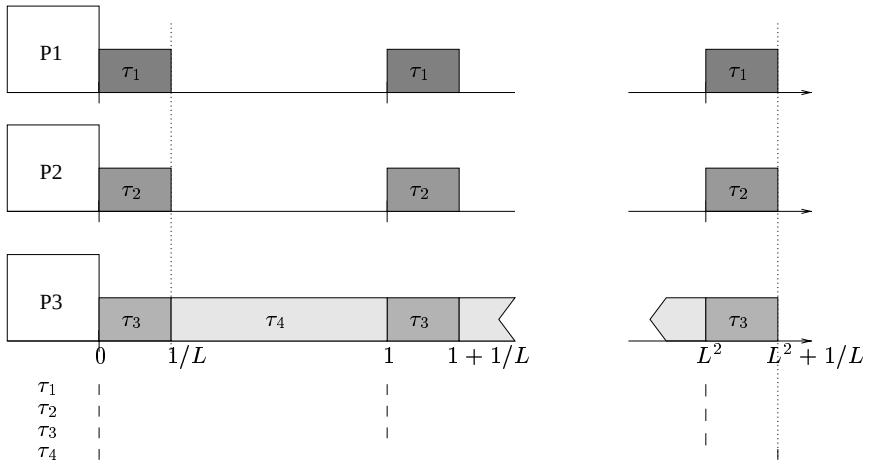


Figure 3: If the TkC priority assignment ($0 < k \leq 1$) for the non-partitioned method is used, there are unschedulable task sets with a low utilization.

4.1 Experimental setup

Below, we describe our performance evaluation. Unless otherwise stated, we conduct the performance evaluation using the following experimental setup.

Task sets are randomly generated and their scheduling is simulated with the respective method on four processors. The number of tasks, n, follows a uniform distribution with an expected value of $E[n] = 8$ and a standard deviation of $0.5\,E[n]$. The period, T_i, of a task τ_i is taken from a set $\{100, 200, 300, \ldots, 1600\}$, each number having an equal probability of being selected. The utilization, u_i, of a task τ_i follows a normal distribution with an expected value of $E[u_i] = 0.5$ and a standard deviation of $stddev[u_i] = 0.4$. If $u_i < 0$ or $u_i > 1$ then a new u_i is generated. The execution time, C_i, of a task τ_i is computed from the generated utilization of the task, and rounded down to the next lower integer. If the execution time becomes zero, then the task is generated again.

The priorities of tasks are assigned with the respective priority-assignment scheme, and, if applicable, tasks are partitioned and assigned to processors. All tasks arrive at time 0 and scheduling is simulated during $\mathrm{lcm}(T_1, \ldots, T_n)$. The reason for selecting small values of m and $E[n]$ is that $\mathrm{lcm}(T_1, \ldots, T_n)$ grows rapidly as n is increased, causing simulations to take too long time.

Two performance metrics are used for evaluating the simulation experiments, namely *success ratio* and *least system utilization*. The success ratio is the fraction of all generated task sets that are schedulable with respect to an algorithm[6] . The success ratio is a recognized performance metric in the real-time community and measures the probability that an arbitrary task set is schedulable with respect to a certain algorithm. The least system utilization is defined as the minimum of the system utilization of all the task sets that we simulated and found to be unschedulable. The least system utilization is primarily used to see if an algorithm is likely to suffer from Dhall's effect and similar effects, that is, if a task set will be unschedulable even for a low utilization. The plots showing the least system utilization will be less smooth than those from the simulations of success ratio. This is because the least system utilization reflects a minimum of numbers rather than an average.

Two non-partitioned priority-assignment schemes are evaluated, namely *RM* which is the rate-monotonic priority-assignment scheme, and *Tk1.1C* which is the TkC priority-assignment scheme with $k = 1.1$ (in Section 4.2, we will show that this is indeed the best k). Four bin-packing-based partitioning schemes are studied, namely RRM-BF [10], RM-FFDU [11], RMGT [8], and R-BOUND-MP [6][7]. The reason for selecting these algorithms is that we have found that they are the partitioning algorithms which provide the best performance in our experimental environment. Since partitioning schemes use a schedulability test as a part of the partitioning, the success ratio is here also a guarantee ratio. Note that this property does not apply for scheduling algorithms using the non-partitioned method. We have also evaluated a hybrid partitioned/non-partitioned algorithm, which we will call *RM-FFDU+Tk1.1C*. The reason for considering a hybrid solution is that we may be able to increase

[6]Since the number of task sets for each point differs between plots (5 000 000 in Figure 5 and 2 000 000 in Figures 6 and 7, we obtain different estimates of the error. With 95% confidence, we obtain errors that are less than 0.0006 for Figure 5 and 0.0014 for Figures 6 and 7.

[7]To ensure correct operation of our simulator, we repeated previous experiments [10, pages 235–236] and [8, pages 1440–1441] with identical results, and [11, pages 36–37] with similar results.

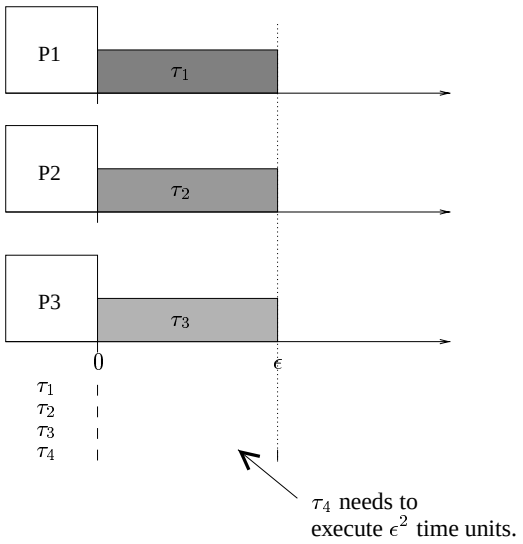


Figure 4: If the TkC priority assignment ($k \to \infty$) for the non-partitioned method is used, there are unschedulable task sets with a low utilization.

processor utilization with the use of non-partitioned tasks, without jeopardizing the guarantees given to partitioned tasks. The RM-FFDU+Tk1.1C scheme operates in the following manner. First, as many tasks as possible are partitioned with RM-FFDU on the given number of processors. Then, the remaining tasks (if any) are assigned priorities according to the Tk1.1C priority-assignment scheme. In order not to jeopardize the partitioned tasks, the priorities of the non-partitioned tasks are set to be lower than the priority of any partitioned task.

Below, we make two major experiments. First, we will vary k to find the best k and compare the results with our reasoning in Section 3.1. Second, we will compare the performance of the Tk1.1C method with that of the partitioning algorithms.

4.2 Finding the best k

Figure 5 shows the success ratio and the least system utilization as a function of the slack factor k. From the plot we make two important observations. First, the least system utilization and the success ratio are correlated. This implies that selecting an appropriate k to maximize the least system utilization will provide a good (though not necessary optimal) k for the success ratio. Second, choosing $k = 1.1$ provides the best success ratio. This corroborates that our reasoning in Section 3.1 was accurate.

4.3 Performance comparison

In Figures 6 and 7, we show the performance as a function of the number of processors. The success ratio associated with the Tk1.1C priority-assignment scheme is higher than that of any other scheduling algorithm. This clearly indicates that our new parametrized priority-assignment scheme works as predicted. In particular, we notice that the traditional RM scheme is outperformed by more than 10 percentage units. The least system utilization associated with Tk1.1C is equivalent to that of the best existing partitioning algorithms. Note how the least system utilization of RM decreases considerably as the number of processors increases, simply because of Dhall's effect. Also observe that Tk1.1C does not appear to suffer from Dhall's effect.

In a complementary study [25], we have also investigated the robustness of the scheduling algorithms by varying the other parameters ($E[n]$, $E[u_i]$ and $stddev[u_i]$) that determine the task set. Here, we observed that the relative ranking of the algorithms does not change; the Tk1.1C priority-assignment scheme still offers the highest success ratio. Furthermore, Tk1.1C continues to provide a least utilization which is equivalent to that of the best existing partitioning algorithms. This further supports our hypothesis that Dhall's effect does not occur for Tk1.1C.

From the plots in Figures 6 and 7, we can also make the following two observations. First, the hybrid partitioning/non-partitioning scheme consistently outperforms the corresponding partitioning schemes. This indicates that such a hybrid scheme is a viable alternative to use in multiprocessor systems that mixes real-time tasks of different criticality. Second, the success ratio of RM is not as bad as suggested by previous studies [2, 3, 4, 5, 7]. The reason for this is of course that Dhall's effect, although it exists, does not occur frequently. This observation corroborates a recent study [16].

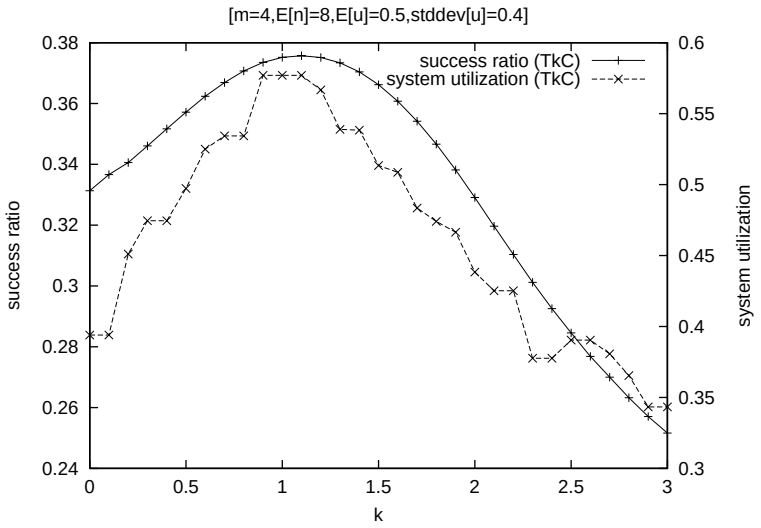


Figure 5: Success ratio and least system utilization as a function of the slack factor k.

For all partitioning algorithms, sufficient schedulability tests were originally proposed to be used. Now, if we instead use a schedulability test based on response-time analysis for the partitioning schemes, the success ratio can be expected to increase, albeit at the expense of a significant increase in computational complexity. Recall that response-time analysis has pseudo-polynomial time complexity, while sufficient schedulability tests typically have polynomial time-complexity. We have simulated the partitioning algorithms using response-time analysis [25] and made the following observation. The FFDU algorithm now occasionally provides a slightly higher success ratio than Tk1.1C, while other partitioning algorithms did not improve their performance that much, and still had a lower success ratio than the Tk1.1C. Note that this means that, even if the best partitioning algorithm (R-BOUND-MP) were to use response-time analysis, it would still perform worse than Tk1.1C. This should not come as a surprise since, for R-BOUND-MP, the performance bottleneck is the partitioning and not the schedulability test [6].

5 Context switches

It is tempting to believe that the non-partitioned method causes a larger amount of (and more costly) context switches than the partitioned method. In this section, we will propose a dispatcher for the non-partitioned method that reduces the number of context switches by analyzing the current state of the scheduler. We will also compare the number of context switches generated by the best non-partitioned method using our dispatcher with that of the best partitioning scheme.

5.1 Dispatcher

The non-partitioned method using fixed-priority scheduling does only require that, at each instant, the m tasks with the highest priorities are executed; it does not require a task to run on a specific processor. Hence, as long as the cost for a context switch is zero, the task-to-processor assignment at each instant does not affect schedulability. However, on real computers the time required for a context switch is non-negligible. If the task-to-processor assignment is selected arbitrarily, it could happen (in theory) that all m highest-priority tasks execute on another processor than they did the last time they were dispatched, even if these m tasks were the ones that executed last. Hence, to reduce the risk of unnecessary context switches, we need a dispatcher that not only selects the m highest-priority tasks, but also selects a task-to-processor assignment such that the number of context switches is minimized.

We now propose a heuristic for the task-to-processor assignment that will reduce the number of context switches for the non-partitioned method. The heuristic is intended to be used as a subroutine in the dispatcher in an operating system. The basic idea of the task-to-processor assignment algorithm is to determine which of the tasks that must execute now (that is, have the highest priority) have recently executed, and then try to execute those tasks on the same processor as their previous execution. The algorithm for this is described in Algorithm 1.

5.2 Comparison of the number of context switches

Two terms contribute to the time for context switches: (i) operating system overhead, including register save and restore and time to acquire a lock for the ready queue, and (ii)

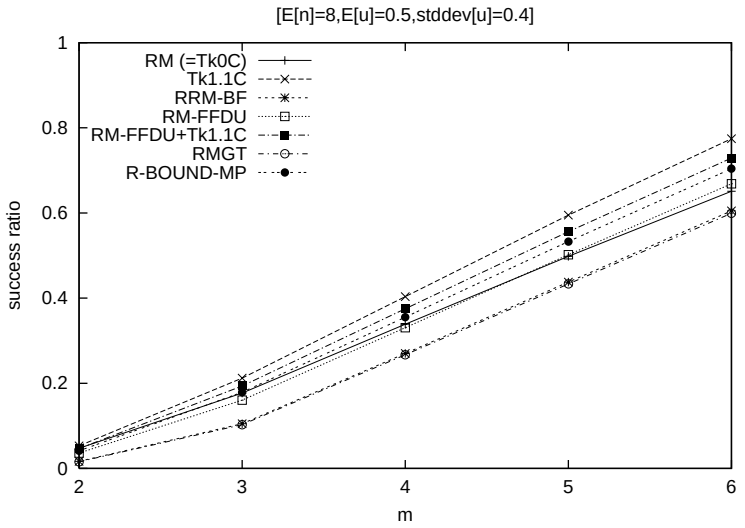


Figure 6: Success ratio as a function of the number of processors.

Algorithm 1 Task-to-processor assignment algorithm for the non-partitioned method.

Input: Let τ_{before} be the set of tasks that just executed on the m processors. On each processor p_i, a (possibly non-existing) task $\tau_{ij,before}$ executed. Let $\tau_{highest}$ be the set of tasks that are in the ready queue and has the highest priority.

Output: On each processor p_i, a (possibly non-existing) task $\tau_{ij,after}$ should execute.

$E = \{p_i : (\tau_{ij,before} \neq \text{non} - \text{existing}) \wedge (\tau_{ij,before} \in \tau_{highest})\}$
for each $p_i \in E$
    remove $\tau_{ij,before}$ from $\tau_{highest}$
    $\tau_{ij,after} \leftarrow \tau_{ij,before}$
for each $p_i \notin E$
  if $\tau_{highest} \neq \emptyset$
    select an arbitrary τ_j from $\tau_{highest}$
    remove τ_j from $\tau_{highest}$
    $\tau_{ij,after} \leftarrow \tau_j$
  else
    $\tau_{ij,after} \leftarrow \text{non} - \text{existing}$

an increase in execuction time due to cache reloading. We assume that the penalty for cache reloading when a task is preempted, and resumes on another processor, is the same as if the task would resume on the same processor after being preempted. If the preempting task (that is, the new task) has a large working set, this should be a reasonable assumption. We also assume that the operating system overhead to resume a preempted task on the same processor is the same as when the task resumes on another processor. Under these assumptions, the cost of a context switch is the same for the partitioned method and the non-partitioned method. Hence, we can count the number of preemptions during an interval of $\text{lcm}(T_1, T_2, \ldots, T_n)$ and use that as a measure of the impact of context switches on the performance of the partitioned method and the non-partitioned method. Note, however, that this does not measure the impact of context switches on schedulability, but gives an indication on the amount of overhead introduced.

To reveal which of the two methods (partitioned or non-partitioned) that suffers the highest penalty due to context switches, we have simulated the scheduling of randomly-generated task sets and counted the number of preemptions. We simulate scheduling using Tk1.1C and R-BOUND-MP because they are the best (as demonstrated in Section 4.3) schemes for the non-partitioned method and the partitioned method, respectively. We use the same experimental setup as described in Section 4.1. We varied the number of tasks and counted the number of preemptions only for those task sets for which both Tk1.1C and R-BOUND-MP were schedulable. Since different task sets have different $\text{lcm}(T_1, \ldots, T_n)$, the impact of task sets with large $\text{lcm}(T_1, \ldots, T_n)$ will be too large. To make each task set equally important, we select to use *preemption density* as a measure of the context-switch overhead. Preemption density of a task set is defined as the number of preemptions during a $\text{lcm}(T_1, \ldots, T_n)$ divided by the $\text{lcm}(T_1, \ldots, T_n)$ itself. We then take the average of the preemption density over a set of 100 000 simulated task sets[8].

The results from the simulations are shown in Figure 8. We observe that, on average, the preemption density for the best non-partitioned method (Tk1.1C) is lower than the preemption density for the best partitioned method (R-BOUND-MP). The reason for this is that, for the partitioned

[8]Hence we obtain an error of 0.0004 with 95% confidence.

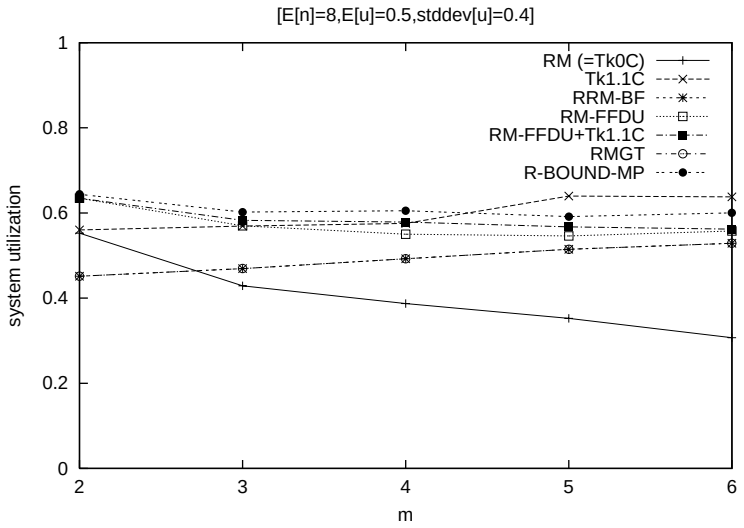


Figure 7: Least system utilization as a function of the number of processors.

method, an arriving higher-priority task must preempt a lower-priority task. With the non-partitioned method, a higher priority task can sometimes execute on another idle processor, thereby avoiding a preemption. Figure 9 illustrates a situation where the non-partitioned method with a context-switch-aware dispatcher causes the number of preemptions to be less than the number of context switches for a partitioned method.

The reasoning and evaluations in this section indicate that the cost of context switches for the non-partitioned method can be made significantly less than for the partitioned method. This means that it is should be possible to demonstrate even more significant performance gain in terms of schedulability for the non-partitioned method relative to the partitioned method on a real multiprocessor system.

6 Discussion

The simulation results reported in Section 4.3 indicate that the non-partitioned method in fact performs better than its reputation. Besides the advantages demonstrated through our experimental studies, there are also other benefits in using the non-partitioned method. Below, we discuss some of those benefits.

The non-partitioned method is the best way of maximizing the resource utilization when a task's actual execution time is much lower than its stated worst-case execution time [16]. This situation can occur when the execution time of a task depends highly on user input or sensor values. Since the partitioned method is guided by the worst-case excution time during the partitioning decisions, there is a risk that the actual resource usage will be lower than anticipated, and thus wasted if no dynamic exploitation of the spare capacity is made. Our suggested hybrid approach offers one solution to exploiting system resources effectively, while at the same time providing guarantees for those tasks that require so. The hybrid solution proposed in this paper applies the partitioned method to the task set until all processors have been filled. The remaining tasks are then scheduled using the non-partitioned approach. An alternativ approach would be to partition only the tasks that have strict (that is, hard) real-time constraints, and then let the tasks with less strict constraints be scheduled by the non-partitioned method. Since it is likely that shared-memory multiprocessors will be used to schedule mostly tasks of the latter type, we expect even better performance (in terms of success ratio) for the hybrid solution.

The non-partitioned method can perform mode changes faster than the partitioned method since tasks are not assigned to dedicated processors. For the partitioned method it may be necessary to repartition the task set during the mode change, something which significantly decreases the number of instants in time that a mode change can take place at. In a complementary study, we have observed pairs of task sets between which a mode change can only be allowed to take place at a few instants in time, otherwise deadlines will be missed. The lack of capability to perform fast mode changes limits the degree of flexibility in the system, which is a serious drawback when working with dynamic application environments.

As shown in this paper, the non-partitioned method can be extended to account for processor affinity with the aid of our proposed context-switch-aware dispatcher. The net result of this extension is that the non-partitioned method incurs fewer context switches at run-time than the partitioned method on the average. However, it is important to real-

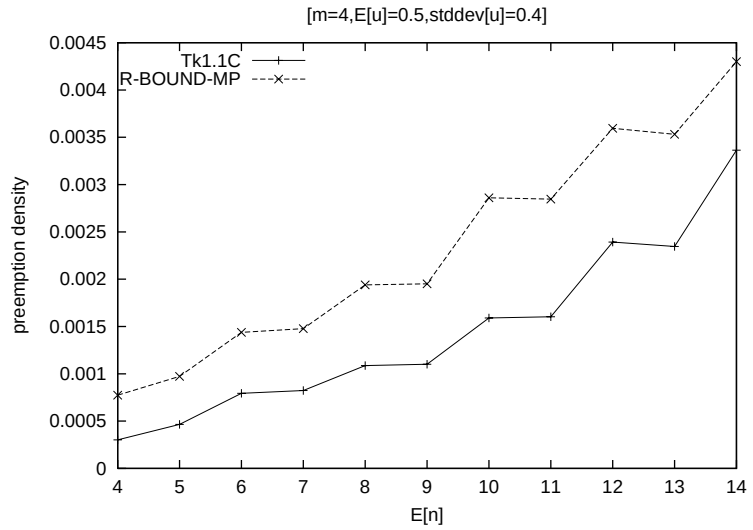


Figure 8: Preemption density as a function of the number of tasks in the system.

ize that this new dispatcher requires more synchronization between processors. The overall effect of this extra synchronization remains to be seen and is consequently a subject for future research. Also, it will be necessary to assess the real costs for context switches and cache reloading and its impact on schedulability. However, this warrants evaluation of real applications on real multiprocessor architectures, which introduces several new system parameters to consider.

Finally, it should be mentioned that the results obtained for the TkC priority-assignment scheme constitute a strong conjecture regarding the existence of a fixed-priority assignment scheme for the non-partitioned method that does not suffer from Dhall's effect. The results also indicate that there is a need to configure TkC for each application and system size. However, in order to use TkC in a system with exclusively strict real-time constraints, it is also important to find an effective schedulability test. In our future research we will therefore focus on (i) proving that TkC does in fact not suffer from Dhall's effect, (ii) constructing a (most likely, sufficient) schedulability test for TkC, and (iii) devising a method to derive the best slack factor k for any given task set and multiprocessor system.

7 Conclusions

In this paper, we have addressed the problem of scheduling tasks on a multiprocessor system with changing workloads. To that end, we have made two major contributions. First, we proposed a new fixed-priority assignment scheme that gives the non-partitioned method equal or better performance than the best partitioning schemes. Since the partitioned method can guarantee that deadlines will be met at run-time for the tasks that are partitioned, we have also evaluated a hybrid solution that combined the resource-effective characteristics of the non-partitioned method with the guarantee-based characteristics of the partitioned method. The performance of the hybrid solution was found to be at least as good as any partitioned method. Second, we proposed a context-switch-aware dispatcher for the non-partitioned method that contributes to significantly reducing the number of context switches taken at run-time. In fact, we show that the number of context switches taken is less than that of the partitioned method for similar task sets.

References

[1] K. Diefendorff and P. K. Dubey. How multimedia workloads will change processor design. *IEEE Computer*, 30(9):43–45, September 1997.

[2] S. Dhall. *Scheduling Periodic-Time-Critical Jobs on Single Processor and Multiprocessor Computing Systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champain, 1977.

[3] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, January/February 1978.

[4] S. Davari and S.K. Dhall. On a real-time task allocation problem. In *19th Annual Hawaii International Conference on System Sciences*, pages 8–10, Honolulu, Hawaii, 1985.

[5] S. Davari and S.K. Dhall. An on-line algorithm for real-time task allocation. In *Proc. of the IEEE Real-Time Systems Symposium*, volume 7, pages 194–200, New Orleans, LA, December 1986.

[6] S. Lauzac, R. Melhem, and D. Mossé. An efficient RMS admission control and its application to multiprocessor scheduling. In *Proc. of the IEEE Int'l Parallel Processing Symposium*, pages 511–518, Orlando, Florida, March 1998.

[7] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, December 1982.

[8] A. Burchard, J. Liebeherr, Y. Oh, and S.H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 44(12):1429–1442, December 1995.

[9] S. Sáez, J. Vila, and A. Crespo. Using exact feasibility tests for allocating real-time tasks in multiprocessor systems. In *10th Euromicro Workshop on Real Time Systems*, pages 53–60, Berlin, Germany, June 17–19, 1998.

[10] Y. Oh and S. H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Systems*, 9(3):207–239, November 1995.

[11] Y. Oh and S. H. Son. Fixed-priority scheduling of periodic tasks on multiprocessor systems. Technical Report 95-16, Department of Computer Science, University of Virginia, March 1995. Available at ftp://ftp.cs.virginia.edu/pub/techreports/CS-95-16.ps.Z.

[12] C. L. Liu and J. 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.

[13] J. Y.-T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(2):209–219, 1989.

[14] C. L. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. In *JPL Space Programs Summary 37-60*, volume II, pages 28–31. 1969.

[15] B. Andersson. Adaption of time-sensitive tasks on shared memory multiprocessors: A framework suggestion. Master's thesis, Department of Computer Engineering, Chalmers University of Technology, January 1999. Available at http://www.ce.chalmers.se/staff/ba/master_thesis/ps/thesis.ps and http://www.docs.uu.se/snart/prizes.shtml#1999.

[16] S. Lauzac, R. Melhem, and D. Mossé. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *10th Euromicro Workshop on Real Time Systems*, pages 188–195, Berlin, Germany, June 17–19, 1998.

[17] L. Lundberg. Multiprocessor scheduling of age contraint processes. In *5th International Conference on Real-Time Computing Systems and Applications*, Hiroshima, Japan, October 27–29, 1998.

[18] J. A. Stankovic, C. Lu, and S. H. Son. The case for feedback control real-time scheduling. Technical Report 98-35, Dept. of Computer Science, University of Virginia, November 1998. Available at ftp://ftp.cs.virginia.edu/pub/techreports/CS-98-35.ps.Z.

[19] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao. The case for feedback control real-time scheduling. In *Proc. of the EuroMicro Conference on Real-Time Systems*, volume 11, pages 11–20, York, England, June 9–11, 1999.

[20] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proc. of the IEEE Real-Time Systems Symposium*, Phoenix, Arizona, December 1–3, 1995.

[21] S. Brandt and G. Nutt. A dynamic quality of service middleware agent for mediating application resource usage. In *Proc. of the IEEE Real-Time Systems Symposium*, volume 19, pages 307–317, Madrid, Spain, 1998.

[22] J. C. Mogul and A. Borg. Effect of context switches on cache performance. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, Santa Clara, CA USA, April 8–11, 1991.

[23] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS 164, Dept. of Computer Science, University of York, York, England Y01 5DD, December 1991.

[24] S. K. Baruah. Static-priority scheduling of recurring real-time tasks. Technical report, Department of Computer Science, The University of North Carolina at Chapel Hill, 1999. Available at http://www.cs.unc.edu/~baruah/Papers/staticdag.ps.

[25] B. Andersson and J. Jonsson. Fixed-priority preemptive scheduling: To partition or not to partition. Technical Report No. 00-1, Dept. of Computer Engineering, Chalmers University of Technology, S–412 96 Göteborg, Sweden, January 2000. Available at http://www.ce.chalmers.se/staff/ba/TR/TR-00-1.ps.

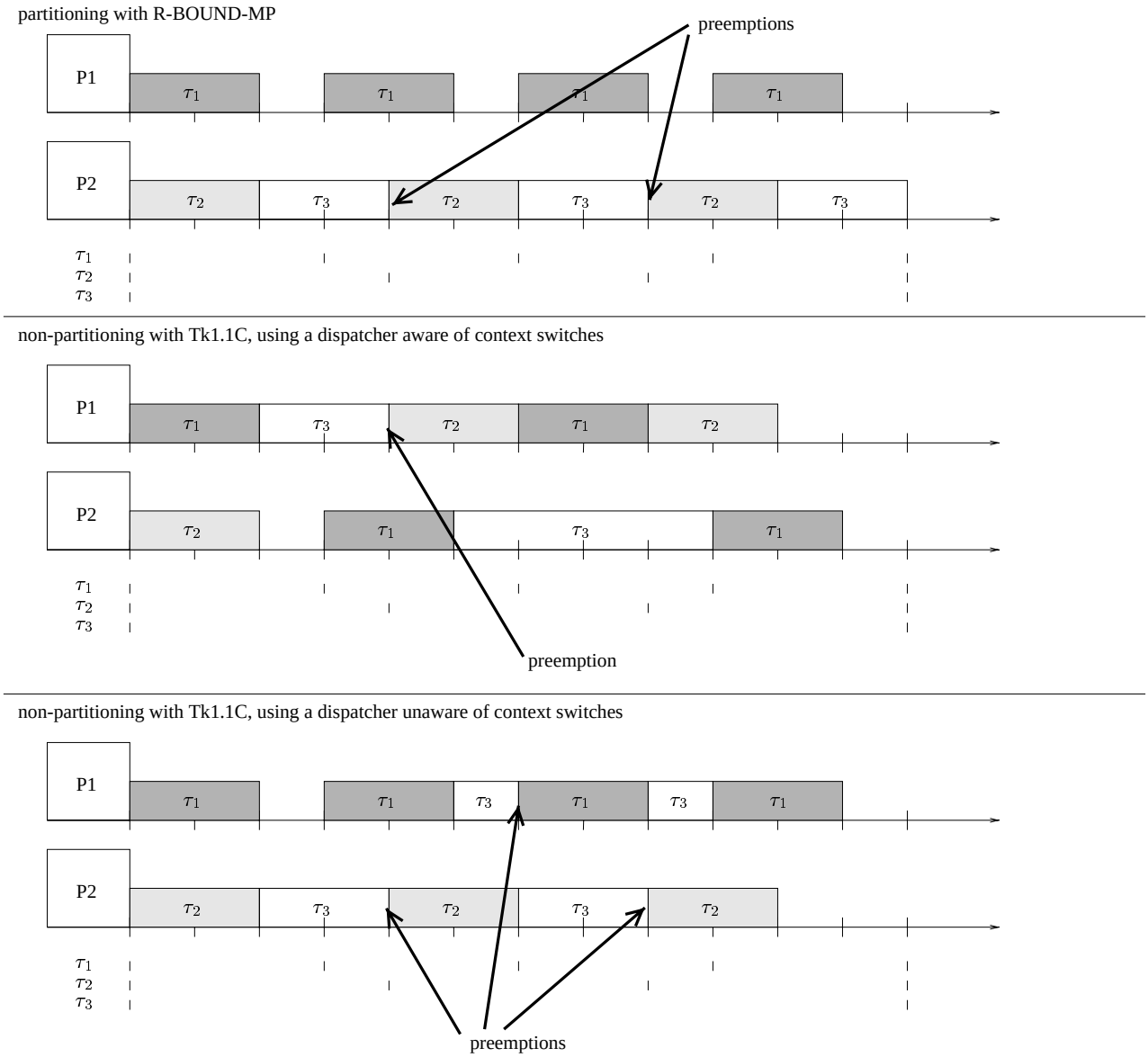


Figure 9: Preemptions for non-partitioned method and partitioned method.

Empty page.

Dynamic Replication Decisions in Fault-Tolerant Multiprocessor-Based Real-Time Systems

Monika Andersson Wiklund and Jan Jonsson

Department of Computer Engineering
Chalmers University of Technology
S–412 96 Göteborg, Sweden
monikaw,janjo@ce.chalmers.se

Abstract

Fault-tolerance is an important property for many critical real-time applications such as telecommunication servers or space-borne signal-processing computers, both for static and dynamically arriving tasks. In this paper, we describe two method for supporting fault-tolerance for dynamically arriving tasks by replicating the tasks and scheduling the copies of each critical task. The methods are varieties of the Primary/Backup method for providing fault-tolerance and the idea is to dynamically decide whether the replication should be temporal or spatial given the execution time and deadline of the task and the load in the system. This offers a flexibility that improves the schedulability of the dynamically arriving tasks.

1 Introduction

Multiprocessor systems available today are used for new high-performance real-time applications, such as telecommunication servers or space-borne signal-processing computers. These systems work in hostile environments where it is not only important that data is processed correctly before a deadline, but also necessary to maximize the successful completion of tasks even in the presence of faults. One way of doing this is to replicate the tasks so that a copy of the task can be completed even if the original task fails. Depending on how the failure of a task is detected, there have to be a different number of copies of each original task.

The most commonly-used method to provide fault-tolerance is the Primary/Backup method where there are two copies of each task, one primary and one backup copy. In most cases, it is assumed that there is a function which detects any failures of a copy when the execution is completed. Since the function tells us when a copy fails and which copy it is that fails, it is possible to deallocate the resources used by the backup copy if the primary copy executes successfully.

However, in many systems it is not always possible to implement such a function, which means that the fault detection must be made in some other way. One method for detecting faults is then to execute multiple copies and compare the results generated by the copies; unfortunately, it is impossible to know which copy failed unless an odd number of copies are used. In some systems it is also sometimes possible to detect a fault before the task is completed because of special hardware mechanisms, for example signature checking. In this case it is possible to remove the copy and start a new execution immediately when the fault is detected.

In all of the above methods, replication is used to achieve fault-tolerance. The difference between the methods is only the number of replicas and how the faults are detected in each method. In both cases the replication can be temporal, spatial, or a combination of both. Which replication method is the best depends on the fault model, the purpose of the replication, and the properties of the system. If both transient, intermittent and permanent faults must be handled, the replication have to be spatial, or both temporal and spatial; but, if only transient faults are handled it is sufficient to use temporal replication. In this work we only consider transient or intermittent faults and thus we use both temporal and spatial replication.

To accept as many dynamically-arriving tasks as possible, it is necessary to execute as few copies as possible. This can be achieved by deallocating the backup copy if the original copy completes its execution successfully. The deallocation releases resources that will not be needed for the successful task and makes it possible to use these resources for any dynamically arriving task that enters the system. However, this deallocation and reuse of resources is only possible when the replication is either temporal or both temporal and spatial, which implicates that it is desirable to use temporal replication whenever it is possible.

In order to achieve fault tolerance for dynamically arriving tasks and to be able to guarantee as many new tasks as possible in the schedule we need to decide which is the best possible method for replicating each new task. One way to do this is to examine the deadline and execution time of the task and, if the deadline is long enough, use temporal replication, otherwise, spatial replication is used. However, the

load in the system and the fault model must also be considered while making such decisions.

In this paper, we propose two methods for dynamic replication of dynamically arriving tasks on a multiprocessor system. In the first method a function is used for fault detection and all faults are detected at the end of the execution, while in the second method multiple copies are used for fault detection and faults can be detected at any time during the execution. Both methods assume that we have a known baseload containing periodic tasks with real-time guarantees, and the tasks that are handled by the method are aperiodic (e.g., telecommunication packets for GSM) and there is no information available for these tasks until they arrive in the system.

To verify and test the methods we use extensive simulation studies. The results from a preliminary study are promising, and verifies the functionality of the proposed methods.

The remainder of this paper is organized as follows: In Section 2 we discuss related work in the area of fault-tolerant real-time systems and replication. Section 3 describes the fault models while Section 4 present the proposed replication methods. The implementation and simulation of the methods are discussed in Section 5. Finally, in Section 6 we summarize our results and discuss future work.

2 Related work

Hou and Shin [1] present a technique where a replication decision is made dynamically off-line by determining whether or not the task module should be replicated in a system with limited resources. Since the resources are limited, only those modules which have deadlines too tight for time redundancy should be replicated in space. The backup copy and the original copy of the task may not run on the same processor and the modules are allocated with an off-line Branch&Bound algorithm. The main objective of this method is to replicate only those modules that have to be replicated, not to find the best possible replication method.

González et al [2] offer a method that uses alternative lists for choosing which method of fault-tolerance should be used. Each arriving task has a list that contains up to three methods, TMR, Primary/Backup, and Primary/Exception. The methods are considered in the order they appear in the list and the scheduler tries to schedule the tasks with the different methods until a feasible schedule is found or the list is empty. With this approach the choice of fault-tolerance is made off-line since the lists must be prepared in advance. This means that the system load and load distribution have no influence over the replication decision.

Mossé, Melhem, and Ghosh [3, 4] present the Primary/Backup principle of replication for independent, dynamically arriving tasks. Backup overloading and resource deallocation is used to achieve high schedulability and the primary copy of a task is always scheduled as early as possible while the backup is scheduled as late as possible but with as much overloading[1] as possible. There are no dynamic decisions in this method since all backups are scheduled after the primary and on another processor than the primary.

The Distance Myopic algorithm [5, 6] also uses the Primary/Backup method. By computing the relative distance in position between the primary and backup copies in the dispatch queue, a flexible level of backup overloading gives a trade-off between number of faults that can be handled, and system performance. All dynamic tasks arrive centrally and are distributed to a processor via the dispatch queues and each processor then executes the tasks in its dispatch queue. When a task is executed successfully, the processor invokes a resource reclaiming algorithm which allows unused resources to be rescheduled and the backup copy to be deallocated (unless it has to execute as well).

Fohler [7] presents a method to achieve adaptive fault-tolerance for distributed systems by first providing a basic reliability level and then divide the schedule into disjoint execution intervals and define the spare capacities for each interval. An on-line scheduler is invoked at the end of each interval and takes care of aperiodic tasks that arrived during the interval by using the spare capacities. After this the spare capacities are updated and the scheduling decision is executed in the next interval using EDF. It is assumed that only one task at a time can be affected by faults and the number of replicas can be increased during runtime if there are enough spare capacities.

Ahn, Kim, and Hong [8] use a variety of the Primary/Backup method where the primary copy of a task may overlap the backup copy of another task. To prevent the domino effect, i.e., that the activation of a backup task causes a primary task to miss its deadline the scheduling is made by forming checkpoint cycles. There must be at least two processors in the cycle that can finish the task currently executing and the newly activated task within their deadlines if a domino effect ahould be avoided. Each task is scheduled on the processor that gives the longest possible checkpoint chain until a cycle is formed, then the processor that satisfies the condition above and who gives the longest checkpoint cycle is chosen.

Srinivasan and Shoja [9] present a method in which periodic tasks with two versions, primary and alternate, are scheduled dynamically to maximize the number of scheduled primaries. The tasks are first scheduled statically and during the execution of the tasks the dynamic scheduler is run to enhance the number of primaries that can be executed. If a primary executes without failure, the alternate for this task is redundant and a, so far, unscheduled primary might be scheduled instead. To find a schedulable primary as quickly as possible, all nodes needs to contain information about the unscheduled primaries in the other nodes. Each node thus has two lists, the schedule to follow

[1] Overloading means that several backup copies are scheduled in the same time slot. This can only be done if the primary copies of all the backups are scheduled on different processors.

and a shortest job list that contains all the unscheduled primaries across the system. When a message from the shortest job list is executed, a message is sent to all nodes and each shortest jobs list is updated.

These methods differ from that of ours in the following ways. Several of the methods use information about tasks that have not yet entered the system, which is not always possible. Also, none of the methods decides dynamically how the replication of the dynamically arriving tasks should be done, which means that some task might be ruled unschedulable even though they might be schedulable if another replication method is used.

3 Fault Models

As in most techniques for fault-tolerant systems, the work in this paper assumes that the *single-fault assumption* is valid, that is, only one faulty result may be produced per replica set.

The faults may appear either in the hardware or in the software and they can be classified as being permanent, intermittent or transient.

Permanent faults in a hardware circuit are caused by, for example, electromigration[2], gate-oxide breakdown, and radiation, and can be detected in self-tests or by repeated execution. When a node can be suspected to contain a permanent fault it has to be stopped and tested, and, if it is faulty, exchanged or repaired. A malicious class of permanent faults that can not usually be detected or prevented by redundancy are design errors.

Transient faults are mostly caused by environmental effects such as radiation or cross-talk (electromagnetic interference) and these faults will not cause any remaining damage on the node. The faults can be detected either by hardware mechanisms, such as checksums or parity coding, or they can be detected by software mechanisms such as voting or integrity checks on the results from the computation.

Intermittent faults might be caused by electromagnetic interference that is caused by, for example radiation, a cellular phone, or a radar antenna which are periodically sending electromagnetic signals. Another possible cause for intermittent faults are undetected permanent faults that only become active from time to time. This means that intermittent faults really are special cases of either permanent or transient faults and can be detected in the same way.

In our methods, permanent faults can only be handled if the replication method operates in a way such that no replicas of the same task are scheduled on the same processing node, or that a cold spare takes over for the stopped node when a permanent fault is detected.

4 Our Dynamic Replication Methods

The replication methods used in our approach are based on the Primary/Backup method, that is, each dynamic task is replicated and have one primary copy which always executes and one or two backup copies that only execute if the primary copy was unable to finish its execution or if a fault is detected. In this work it is assumed that only transient or intermittent faults can occur; however, it is possible to adjust the method to allow for permanent failures as well.

[2]Movement of metal due to momentum exchange with electrons causing preferred direction of diffusion. [10]

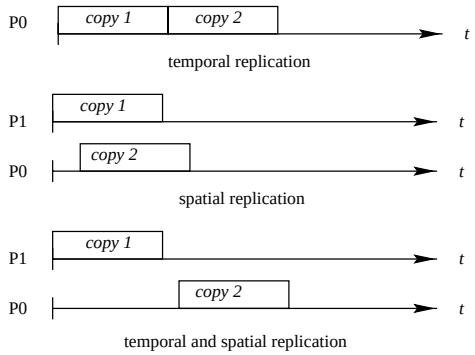


Figure 1: Replication varieties for two copies.

Our first method for dynamic replication assumes that there exists a function which detects whether the task is successful or not when the execution is completed. Thus only one copy of each task is needed, the original and a backup. Our second method assumes that no such function exists, and we therefore use two copies of each task for voting; however, it is possible to find some faults before the execution is completed if the faults are detected by hardware mechanisms.

The way in which the scheduling is done means that sometimes the replication can be both temporal and spatial at the same time. That is, if the replication is temporal, but a backup copy cannot be scheduled on the same processor as the primary copy, then the replication will be both temporal and spatial. It is important to note that a backup copy always must execute after the primary copy if the replication is temporal, and that it may be partially or completely overlapped with the primary copy if the replication is spatial.

Also, by using a distributed scheduling algorithm [11] for scheduling, the replication strategy will depend on the load in the system. Unless we have to handle permanent failures, it is preferred that the primary and the backup copy should be scheduled on the same processor; however, if this is not possible due to high system load, we can allow the copies to be scheduled on different processors.

4.1 Method 1: Two copies of each task

The dynamic replication strategy using two copies is very simple. When a dynamic task arrives in the system, the deadline and execution time of the task is examined. If the difference between the deadline (D_i) and the execution time (C_i) multiplied with a certain replication breakpoint, s, is larger than the task execution time of a potential backup task, then the task is replicated in time; otherwise it is replicated in space. Depending on the load in the system, the replication can also be a combination of both (see Figure 1).

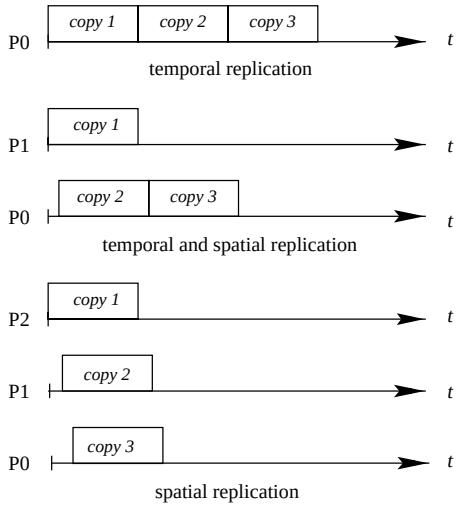


Figure 2: Replication varieties for three copies.

The reason for having the replication breakpoint parameter is that there might be extra cost involved in the execution, for example, a communication cost may be involved if the primary and backup copies are situated on different processors.

$$D_i - C_i \times s \geq C_i \rightarrow \textit{replicate in time}$$

When the task has been replicated, the two copies of the task are scheduled using a distributed scheduling technique such as the Random scheduling algorithm or the Flexible algorithm [11]. First, the primary copy is scheduled on the processor it arrived to, or if it is not possible to schedule it on this processor, it is sent to one of the other processors. Which other processor is chosen to send the copy to depends on which scheduling technique is used; a processor can either be chosen randomly or on the basis of the load in the system.

If the replication is temporal, the method attempts to schedule a backup copy on the same processor as the primary copy. If this is not possible, a processor is chosen accordingly to the scheduling technique and the method attempts to schedule the copy on this processor. For spatial replication, on the other hand, the backup copy may not be scheduled on the same processor as the primary, that is, another processor must be chosen before the scheduling attempt. Both the primary and the backup copies must be scheduled for the task to be accepted; if either copy cannot be scheduled the task is rejected.

The method can be altered to handle permanent failures by scheduling all copies on different processors whenever possible.

4.2 Method 2: Three copies of each task

In this method we need at least two, but sometimes three, copies of each task since we assume that there is no function for detecting faults, that is, there is no integrity check for the result of the computation. We assume that at most one copy

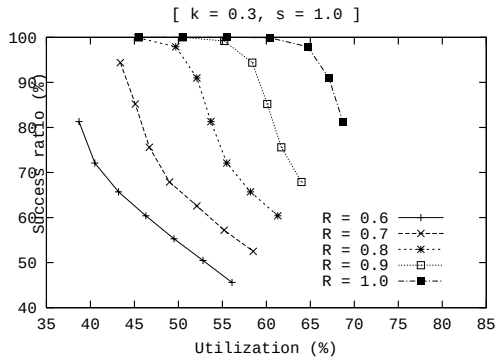


Figure 3: The influence of deadline ratio, R.

of each task can be faulty, and thus three copies of each task is enough for fault detection. At first, two copies are executed and their results are compared; if the results match, the task is successful and the third copy is unnecessary. If the results do not match, the third copy must be executed and compared with the first results so that the correct result can be identified.

Since the third copy of each task only is needed when a fault has been detected, it is desirable that this copy is scheduled after the other copies so that it can be deallocated in case it will not be needed. There are three possible combinations for replicating the task: (i) all copies are temporal replicas, (ii) two copies are spatial replicas and one is temporal, or (iii) all copies are spatial replicas (see Figure 2). Which combination is used depends on the deadline of the task in the following way. If the deadline is long enough to allow all copies to execute after each other within the deadline, only temporal replication is used. If the deadline is too short for two copies to execute after each other, only spatial replication is used; otherwise, one spatial and one temporal replication is used.

To avoid that certain types of faults affect more than one copy in the same way, it is possible to place all copies on different processors and make sure that they do not start at exactly the same time. This means that we will mix the temporal and spatial properties of the replicated copies so that all copies will differ in both time and space.

5 Implementation and Simulation

We have implemented our methods in GAST [12] (Generic Allocation and Scheduling Tool), a software package developed at Chalmers University of Technology. The implementation of the proposed methods has been performed in steps. Initially, a simple version of the first method without resource deallocation was implemented and later resource deallocation was added. The next step was to implement a simple version of the second method; this implementation has just been finished.

When the simple method of the first version was implemented, we ran a small simulation study to verify that it behaved as expected. Later, we will add the results of newer simulation studies on the full implementation.

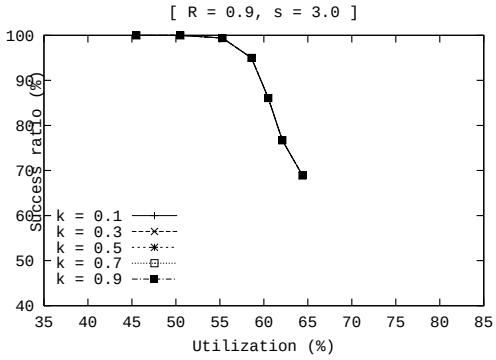


Figure 4: The influence fault intensity, k.

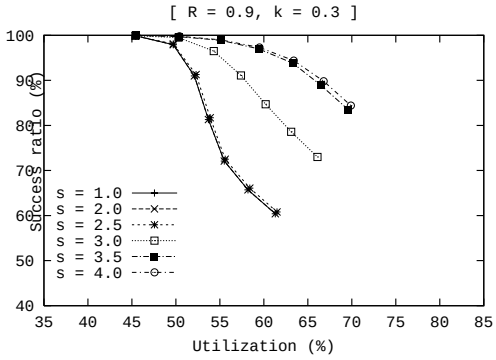


Figure 5: The influence of replication breakpoint, s.

For the small simulation study, we used a system with four nodes, a baseload of periodic tasks with periods of 100 time units and dynamically arriving tasks with a minimal inter-arrival time of 75 time units. Each task had an execution time that was randomly chosen from a normal distribution with a mean execution time of 15 time units and a variance of 1. The influence of three different parameters was examined by running 128 simulations for each setting. The parameters were; the deadline/period ratio (R), the fault intensity (k), and the replication breakpoint (s). The mean execution time was increased in steps of 5 from a start value of 5 to a final value of 45 for each combination of the parameters. We use a non-preemptive, time-driven execution model and we use success ratio as a measure of performance in our simulations. We define the success ratio as the fraction of tasks that can be scheduled with our method out of all dynamically arriving tasks.

The effect of the deadline/period ratio is shown in Figure 3. It is clear from the plots that if the deadline is shortened (that is, the ratio R is decreased) the success ratio is also decreased.

In Figure 4, the effect of the fault intensity is shown. As can be seen there is no effect on the success ratio when the fault intensity is increased. This is due the fact that we do not use resource deallocation in this simulation study. Because of this, backup copies that are not needed cannot be removed from the schedule and, thus, it does not matter to the success ratio how large the fault intensity is. If we use resource deallocation, however, there should be a difference in success ratio if the the fault intensity k is varied.

The effect of the replication breakpoints is shown in Figure 5. Recall that, when the replication breakpoint is low, only spatial replication is used and when it is high only temporal replication is used. The figure shows that the success ratio decreases when the replication breakpoint is increased. The reason for this is that, for shorter deadlines, it is harder to find room for temporal copies than for spatial copies because the amount of free time between the baseload tasks is too small.

6 Summary and Future Work

We have presented two methods for dynamic replication of dynamically arriving tasks in a real-time multiprocessor system. The methods are varieties of the Primary/Backup method but have been extended to use the most suitable replication mode for each arriving task based on task characteristics and system load. The two methods are very similar except for how the fault detection is made.

The implementation of the first method is finished and verified by a small simulation study, and a simple implementation of the second method have just been finished. We will perform a more extensive simulation study to find which parameters are most influential on the success ratio for the methods. We will also investigate how much it helps to use resource deallocation. Finally, we will make a full implementation of the second method and perform further simulation studies to see if there are any differences between the full implementation and our simpler version.

References

[1] C.-J. Hou and K. G. Shin, "Replication and Allocation of Task Modules in Distributed Real-Time Systems," *Proc. of the IEEE 24th International Symposium on Fault- Tolerant Computing*, Austin, TX, June 15–17, 1994, pp. 26–35.

[2] O. Gonzlez, H. Shrikumar, J. A. Stankovic, and K. Ramamritham, "Adaptive Fault Tolerance and Graceful Degradation Under Dynamic Hard Real-time Scheduling," *Proc. of the 18th IEEE Real-Time Systems Symposium*, San Francisco, CA, Dec. 2–5, 1997, pp. 79–89.

[3] D. Moss, R. Melhem, and S. Ghosh, "Analysis of a Fault-Tolerant Multiprocessor Scheduling Algorithm," *Proc. of the IEEE 24th International Symposium on Fault- Tolerant Computing*, Austin, TX, June 15–17, 1994, pp. 16–25.

[4] S. Ghosh, R. Melhem, and D. Moss, "Fault-Tolerant scheduling on a Hard Real-Time Multiprocessor System," *Proc. of 8th International Parallel Processing Symposium*, Cancun, Mexico, Apr. 26–29, 1994, pp. 775–782.

[5] G. Manimaran and S. R. Murthy, "A Fault Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and Its Analysis," *IEEE Trans. on Parallel and Distributed Systems*, vol. 9, no. 11, pp. 1137–1152, Nov. 1998.

[6] G. Manimaran and S. R. Murthy, "A new Study for Fault-Tolerant Real-Time Dynamic Scheduling Algorithms," *Journal of Systems Architecture*, vol. 45, no. 1, pp. 1–13, Oct. 1998.

[7] G. Fohler, "Adaptive Fault-Tolerance with Statically Scheduled Real-Time Systems," *Proc. of the Ninth Euromicro Workshop on Real Time Systems*, Toledo, Spain, June 11–13, 1997, pp. 161–167.

[8] K. Ahn, J. Kim, and S. Hong, "Fault-Tolerant Real-Time Scheduling using Passive Replicas," *Proc. on the Pacific Rim International Symposium on Fault-Tolerant Systems*, Taipei, Taiwan, Dec. 15–16, 1997, pp. 98–103.

[9] A. Srinivasan and G. C. Shoja, "A Fault-Tolerant Dynamic Scheduler for Distributed Hard-Real-Time Systems," *Proc. on the Pacific Rim Conference on Communications Computers and Signal Processing*, Victoria, BC, Canada, May 19–21, 1993, pp. 268–271.

[10] C. Hawkins, "Field Returns: Reliability Failure Mechanisms and Defect Electronic Test Properties," *Tutorial B, IEEE European Test Workshop*, Sitges, Spain, May 26, 1998, pp. XX–XX.

[11] K. Ramamritham and J. A. Stankovic, "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Trans. on Computers*, vol. 38, no. 8, pp. 1110–1123, Aug. 1989.

[12] J. Jonsson, "GAST: A Flexible and Extensible Tool for Evaluating Multiprocessor Assignment and Scheduling Techniques," *Proc. on the Int'l Conf. on Parallel Processing*, Minneapolis, Minnseota, Aug. 10–14, 1998, pp. 441–450.

Using Massive Time Redundancy to Achieve Node-level Transient Fault Tolerance

J. Aidemark, J. Karlsson
Laboratory for Dependable Computing
Chalmers University of Technology
S-412 96 Göteborg, Sweden
{aidemark, johan}@ce.chalmers.se

1. Introduction

Distributed real-time systems are increasingly being used to control critical functions in automotive and aerospace applications, such as fly-by-wire, break-by-wire, and steer-by-wire systems. These systems must be fault-tolerant to be safe and reliable.

For a cost-effective implementation of a fault-tolerant distributed real-time system, it is necessary to design the computer nodes in the system in such way that they exhibit well-behaved failure semantics. This is achieved by including error detection and fault-tolerance mechanisms at the node-level. Nodes that exhibit only[1] fail-stop/(crash) failures or omission failures [1] can be considered to have well-behaved failure semantics.

Fail-stop semantics implies that the node produces either correct results (at the right time) or produces no results at all, i.e. it stops producing results when affected by a fault. An omission failure occurs if the node fails to produce a particular result, but continues to produce correct (and timely) results after the omission failure occurred. These failures can be handled effectively by fairly simple distributed redundancy management protocols, see e.g. the MARS system [2].

Previous research has shown that transient faults are common in digital systems [3]. Transient faults can be caused by particle radiation, e.g. in aircraft at high altitude by high-energy neutrons, or in spacecraft by heavy-ions. Power fluctuations and electromagnetic interference are other causes of transient faults in computer systems.

In this paper we describe an implementation of a small real-time kernel that achieves transient fault tolerance at the node-level. The objective is to tolerate transient faults at the node-level whenever possible. For permanent faults and transient faults that cannot be handled at the node-level, the node should fulfil fail-stop or omission failure semantics. These properties are achieved by combining hardware and software error detection mechanisms (EDMs) with massive time redundancy. The kernel uses fixed-priority pre-emptive scheduling [4].

We consider real-time systems with hard deadlines where fault-tolerance is achieved by executing each critical task on two nodes with fail-stop and omission failure semantics. When such a system is used in an environment where transients are common, such as a car or a satellite, the time it takes to recover from a transient fault has a significant impact on the overall system reliability. The longer it takes to recover from a transient fault, the higher the probability that the system crashes because another transient fault affects the remaining non-faulty node.

Many existing systems, e.g. MARS [2], do not provide node-level transient fault tolerance, which means that the recovery of all transient faults are handled via the distributed redundancy management protocol. In this case, the recovery time is in the order of seconds or minutes. The advantage of handling transient faults at the node-level is that the average recovery time for transients can be reduced by several orders of magnitude, which improves the overall reliability.

Node level transient fault tolerance also improves the system resiliency to correlated node failures, which occur when a single disturbance causes transient faults in several nodes simultaneously.

[1] That is, the probability for other types of failures is assumed to be negligible.

2. Related work

Error detection by comparing the result from two replicated executions of critical tasks was used in the MARS system [2].

Recovery from transients faults can be achieved by using a checkpointing scheme [5]. Checkpointing is done by saving the state of the processor to stable storage in regular intervals or when certain data is updated. When a fault occur, the system is restored to the last checkpoint and the execution can proceed.

Fault-tolerant scheduling of real-time tasks in a uniprocessor environment is studied in [6]. This work is based on the assumption that transient faults are always detected, and that recovery can be achieved by a retry/ re-execution of the affected task. The aim is to allocate adequate time for the recovery action (i.e. the retry/ re-execution). In [6] time is reserved prior to execution using a technique called overloading. Overloading reserve less time than needed to re-execute all tasks based on the condition that at most one fault can occur during a specific time interval. An approach for dynamically allocating time for re-execution upon a recovery request is presented in [7].

Techniques such as robust data structures, assertions, plausibility tests and checking execution time of system routines were used in [2] to detect transient faults during execution of operating system code.

3. Basic ideas

In massive time redundancy, a task is executed at least three times to produce three or more copies of a result. A majority vote is performed on the copies to mask any faults (cf. massive redundancy in hardware, e.g. a TMR system).

In our real-time kernel, each critical task is always executed twice during normal operation. We use the term *task replica*, or just *replica*, to denote a particular instance of the task execution. The results of the two replicas are compared to detect errors. If the two results do not match, a third replica of the task is executed. The results of the three replicas are then checked by a majority vote. If none of the results match, no result is delivered, which leads to a omission failure. If two results match, they are accepted as a valid result of the task.

Errors can also be detected by hardware and software EDMs such as illegal op-code detection or variable constraint checks. In this case, the affected replica is immediately terminated, and a new replica is started.

We assume that there is enough slack (unused cpu time) available to allow at least one task to execute three replicas, without causing any other task to miss a deadline. However, if two or more task are affected by near-coincident transient faults, it may not be enough slack available to allow all of them to execute three replicas, without causing other tasks to miss their deadlines.

After an error is detected, the kernel checks the deadline of the task to determine if it is possible to execute an additional task replica before the deadline. If not, no result is delivered and an omission failure occurs. If time is available, a new replica is started. The task result is delivered only when two matching results have been produced before the deadline.

We assume a simple task model where the input is read in the beginning of the task and the result is delivered at the end of the task. We also assume that the input to a task is not updated until the task result is delivered.

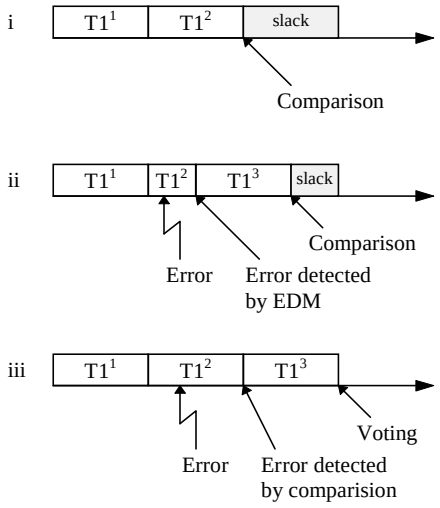


Figure 1: Critical task with replicated executions and fault handling.

Three time diagrams of the execution of task replicas is shown in Figure 1. The time for the execution of the kernel code is not included in the diagrams. The figure shows three different scenarios: (i) Fault free operation, where two replicas $T1^1$ and $T1^2$ of task T1 is executed and their results are compared to detect errors. The

results match and can be delivered directly, not using any of the available slack. (ii) The error is detected by an EDM during execution of the second replica $T1^2$. The affected replica $T1^2$ is terminated and a new replica $T1^3$ is immediately started. $T1^3$ will use time reclaimed from the removed replica $T1^2$ as well as time from the available slack After two replicas have been executed, a comparison is made to confirm that the results match before the task result is delivered. (iii) The error is detected by the comparison. Thus, the results from the two replicas $T1^1$ and $T1^2$ are dissimilar. It is not possible to decide which of the two replicas that produced the correct result so a third replica of the task is started, called $T1^3$, using the available slack. The results of the three executions are check by a majority vote. As previously described, no result is delivered if none of the results match. If two results match, they are accepted as a valid task result.

4. Implementation and Validation

We are implementing a small real-time kernel that achieves transient fault tolerance by using the ideas described above. The kernel will be implemented for the Thor microprocessor developed by Saab Ericsson Space AB.

Fault injection will be performed to estimate the probability of violating the fail-stop or omission failure semantics. We will use two fault injection tools, FIMBUL [8] and MEFISTO-C [9]. FIMBUL inject faults in the physical unit of Thor using the scan-chains. Faults can be injected in the registers and in the cache. MEFISTO-C inject faults in the VHDL model of the Thor processor. Here, faults can be injected in any state element (i.e. a flip-flop, latch or register) of the processor.

5. Summary and Discussion

This paper describes an implementation of a real-time kernel that achieves transient fault tolerance by using massive time redundancy. The real-time kernel is being implemented for the Thor microprocessor, and will be validated by using fault injection to estimate the probability of violating the fail-stop or omission failure semantics. Although the estimated coverage results will be specific for the chosen microprocessor, the experiments will give valuable insight into the usefulness of the proposed technique. In the current version, the kernel only handles transient faults that occur during execution of application tasks.

Future work include the development of techniques that ensure fail-stop semantics when transient faults occur during the execution of the kernel code. We will also investigate different policies for handling near-coincident transient faults that affect more than one task. Another interesting issue is to study error containment between tasks and between tasks and the kernel.

6. References

[1] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", *Communications of ACM*, Vol.34, No.2, 1991, pp. 56-78.

[2] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, "Tolerating transient faults in MARS" *20th Int. Symp. on Fault-Tolerant Computing* (FTCS-20), IEEE, Newcastle Upon Tyne, U.K., June 1990. pp. 466-473.

[3] R.K. Iyer, D.J. Rossetti and M.C. Hsueh, "Measurement and Modeling of Computer Reliability as Affected by System Activity", *ACM Trans. on Computer Systems*, Aug 1986, pp. 214-237.

[4] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings, "Fixed Priority Pre-emptive Scheduling: An Historical Perspective", *Real-Time Systems*, vol. 8, no. 2/3, 1995, pp. 129-154.

[5] D. K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice Hall, USA, 1996, pp. 160-192.

[6] S. Ghosh, "Guaranteeing Fault Tolerance through Scheduling in Real-Time Systems", *Ph.D. Thesis, Dept. Computer Science. University of Pittsburgh*, USA, 1996.

[7] P. Mejia-Alvarez, and D. Mossé, "A responsiveness approach for scheduling fault recovery in real-time systems" *Real-Time Technology and Applications Symposium*, IEEE, Vancouver, Canada, June 1999. pp. 4-13.

[8] P. Folkesson, S. Svensson, and J. Karlsson, "A comparison of simulation based and scan chain implemented fault injection", *28th Int. Symp. on Fault-Tolerant Computing* (FTCS-28), IEEE, Munich, Germany, June, 1998. pp. 284-293.

[9] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: the MEFISTO tool", *24th Int. Symp. on Fault-Tolerant Computing* (FTCS-24), IEEE, Austin, TX, USA, June 1994. pp. 66-75.

Empty page.

AIDA II
Automatic control in distributed applications

Ola Redell, Jad El-khoury, Martin Törngren
KTH, Department of Machine design,
Mechatronics lab, 100 44 Stockholm
{ola | jad | martin}@md.kth.se

1 Introduction

Machine industry faces a shift where more and more functionality is implemented in software on distributed computer systems. Many difficult problems then face designers including system modelling, analysis and management of multi disciplinary design teams. There is a need to transfer theoretical results to industrial practice, necessitating the integration of existing work from separate disciplines. Industrial practice, on the other hand, needs to be shifted towards model building, analysis and prediction in order to avoid clashes in system integration.

Several useful modelling approaches and computer aided tools exist for the design of control, software and mechanical systems. These support tools however fall short when the applications are implemented in embedded and distributed real-time computer systems. A number of additional design issues then face designers including

- Structuring, partitioning and allocation, i.e. how to establish structures and the mappings between them (e.g. functions/threads/processors).
- Execution and communication policies. The definition of policies and mechanisms for triggering, scheduling, synchronisation and communication.
- Performance requirements. The definition of timing requirements and related trade-offs.
- Error detection and handling policies. In particular additional failure modes of distributed real-time systems must be identified and handled.

The provision of a modelling framework that enables interdisciplinary work is a prerequisite in order to support these design issues.

2 Approach and status

The objective in the AIDA research project is to develop a modelling framework and methods for analysis of real-time behaviour in order to support the development of distributed, heterogeneous control systems.

The AIDA project has been running since 1996. In the first phase of the project (1996-1998) the work was focused on the design of a modelling framework to support the design and specification of embedded distributed real-time control systems. This work has been carried out in co-operation with a industrial reference group including Scania, SAAB Military Aircraft, Atlas Copco Controls and ABB Robotics. The modelling framework has been used in a case study to describe possible implementations of a control system for a four legged vehicle [5].

The continuation of AIDA, the AIDA II project, is carried out in cooperation with SAAB Combitech Software. The project includes continued theoretical work on the modelling framework, the development of a prototype tool-set and an accompanying design method, more case studies and strengthened industrial cooperation.

2.1 Modelling real-time requirements for control systems

The AIDA modelling framework provides a number of models (and views) related to domains; functions, software, hardware and mechanical interfaces, and provides structural and timing behaviour models for each domain.

The basic idea of the modelling framework is that the models should include all information needed to completely specify the system's implemented behaviour and requirements. This includes execution times, functional decomposition, partitioning, allocation, scheduling policies, communication media etc. With such a modelling framework as a base for a CAE (Computer Aided Engineering) tool-set it should be possible to develop functionality that can assist system designers to test, analyse and compare different implementations regarding real-time measures important for control implementations (such as response times and system induced jitter).

2.2 Timing analysis of control implementations

Automatic control systems are by design highly dependant of (and sensitive to changes in) the final timing in the target system. Generally, control activities are periodic and put hard requirements on jitter in both sampling and actuation. With these facts in mind, it is amazing how little support there is for off-line pre-implementation timing analysis of control implementations. Today's implementation of control systems is obviously based on a combination of expert knowledge, conservative assumptions and trial and error.

Within the area of fixed priority, single processor scheduling, some important results have been developed during the last decade [8]. However, for distributed systems with various scheduling policies on many different processors and communication links (not always fixed priority) few similar results exist. The AIDA tool-set will include a tool that helps in analysis of timing properties of control systems implementations, specifically implementations on distributed heterogeneous hardware. Hence further research has to be done in this area and when enough results have been achieved, the tool is to be developed.

Current work is partly focused on co-simulation of control applications with communication resources and processor scheduling policies.

3 Results

The most recent publication gives an overview of the modelling framework and compares the AIDA framework with representative models with a basis in object-oriented analysis and design (UML real-time extensions), structured analysis and design (DARTS/DA), and real-time scheduling research, [3]. The modelling framework is described in full in [4]. The modelling framework has been used in a case study to describe possible implementations of a control system for a four legged vehicle [5]. A survey of scheduling and allocation methods for distributed real-times systems is presented in [6]. The licentiate thesis, [7], contains papers [2,4-6], a description of the tool-set and an overview of related work.

4 References

[1] Törngren Martin and Wikander Jan. A decentralization methodology for real-time control applications. J. of Control Engineering Practice, Vol. 4 No. 2, Feb. 1996, Pergamon.

[2] Törngren Martin and Redell Ola. A Mechatronics Test-Bed for Embedded Distributed Control Systems. Proc. of Algorithms and Architectures for Real-Time Control, AARTC'97, Vilamoura, April 1997.

[3] Törngren Martin and Redell Ola. A Modelling Framework to support the design and analysis of distributed real-time control systems. Invited Paper. To appear in the Journal of Microprocessors and Microsystems, Elsevier, special issue based on selected papers from the Mechatronics 98 proceedings

[4] Redell Ola and Törngren Martin. Preliminary design of models for the AIDA tool-set, Technical report, Dept. of Machine design, KTH, 9805.

[5] Redell Ola. Modelling and Implementation Analysis of the Distributed Real-Time Control System for a Four Legged Vehicle. Technical report Dept. of Machine design, KTH, 9805.

[6] Redell Ola. Global Scheduling in Distributed Real-Time Computer Systems, An Automatic Control Perspective. Technical report Dept. of Machine design, KTH, 9805.

[7] Redell Ola. Modelling of Distributed Real-Time Control Systems - An Approach for Design and Early Analysis, Licentiate Thesis, 9805.

[8] Burns, A. Preemptive Priority-Based Scheduling: An Appropriate Engineering Approach, Ch. 10 in Advances in Real-Time Systems, ISBN 0-13-083348-7, Prentice-Hall, 1995

Empty page.

Feedback Scheduling of Control Tasks

Anton Cervin **Johan Eker**

Department of Automatic Control
Lund Institute of Technology
Box 118, SE-221 00 Lund
Sweden
{anton,johane}@control.lth.se

Abstract

The paper presents a feedback scheduling mechanism in the context of co-design of the scheduler and the control tasks. We are particularly interested in controllers where the execution time may change abruptly between different modes, such as in hybrid controllers. The proposed solution attempts to keep the CPU utilization at a high level, avoid overload, and distribute the computing resources evenly among the tasks. The feedback scheduler is implemented as a periodic or sporadic task that assigns sampling periods to the controllers based on execution-time measurements. The controllers may also communicate feedforward mode-change information to the scheduler. As an example, we consider hybrid control of a set of double-tank processes. The system is evaluated, from both scheduling and control performance perspectives, by co-simulation of controllers, scheduler, and tanks.

1. Introduction

There is currently a trend towards more flexible real-time control systems. By combining scheduling theory and control theory, it is possible to achieve higher resource utilization and better control performance. To achieve the best results, co-design of the scheduler and the controllers is necessary. This research area is only beginning to emerge, and there is still a lot of theoretical and practical work to be done, both in the control community and in the real-time community.

Control tasks are generally viewed by the scheduling community as *hard* real-time tasks with *fixed* sampling periods and *known* WCETs. Upon closer inspection, neither of these assumptions need necessarily be true. For instance, many control algorithms are quite robust against variations in sampling period and response time. Controllers can be designed to switch between different modes with different execution times and perhaps also different sampling intervals. It is also possible to consider control systems that are able to do a trade-off between the available computation time and the control loop performance.

As an example throughout this paper, we study the problem of scheduling a set of hybrid-control tasks. Such tasks are good examples of tasks that do not really meet the assumptions commonly made in the scheduling theory. A hybrid controller switches between different modes, which may have very different execution-time characteristics. Utilizing only worst-case execution-time (WCET) estimates in the scheduling design can result in very low resource utilization, slow sampling, and low control performance. On the other hand, if instead, for instance, average-case execution-time estimates are used in the scheduling design, the CPU may experience transient overloads during run-time. This, again, can result in low control performance, and even temporary shut-down of the controllers.

In this work, we present a feedback scheduler for control tasks that attempts to keep the CPU utilization at a high level, avoid overload, and distribute the computing resources evenly among the tasks. While we want to keep the number of missed deadlines as low as possible, control performance is our primary objective. Thus, control tasks, in our view, fall in a category somewhere between hard and soft real-time tasks. The known-WCET assumption is relaxed by the use of feedback from execution-time measurements. We also introduce feedforward to further improve the regulation of the utilization.

The structure of the feedback scheduler is shown in Figure 1. A set of control tasks generate jobs that are fed to the run-time dispatcher. The scheduler gets feedback information about the actual execution time, c_i, of the jobs (it is assumed that this information can be provided by real-time operating system). It also gets feedforward information from control tasks that are about to switch mode. This way, the scheduler can proact rather than react to sudden changes in the workload. The scheduler tries to keep

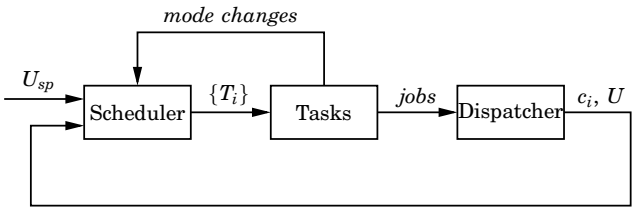


Figure 1 The feedback scheduling structure.

utilization, U, as close as possible to the utilization setpoint, U_{sp}. This is done by manipulating the sampling periods, $\{T_i\}$. The choice of utilization setpoint depends on the scheduling policy of the dispatcher, and on the sensitivity of the controllers to missed deadlines. Notice that the well-known, guaranteed utilization bounds of 100% for earliest-deadline-first (EDF) scheduling and 69% for fixed-priority scheduling are not valid in this context, since the assumptions about known, fixed WCETs and fixed periods are violated.

The calculated task periods should reflect the relative importance of the different control tasks. One possibility is to assign nominal sampling periods to the controllers off-line. The feedback scheduler can then do linear rescaling of the task periods to achieve the desired utilization. The controllers are informed of the new sampling periods and may adjust their parameters if necessary. Other possibilities could include on-line optimization of a control performance criterion over the task periods, subject to the utilization constraint. The feedback scheduler is in the end also implemented as a periodic or sporadic task that consumes computing resources. There is a fundamental trade-off between the time that should be spent doing scheduling, and the time left over for control computations.

1.1 Related Work

The related work falls into three categories. The first one is the field of integrated control system and real-time system design. In [Seto *et al.*, 1996], sampling period selection for a set of control tasks is considered. The performance of a task is given as function of its sampling frequency, and an optimization problem is solved to find a set of optimal task periods. Co-design of real-time control systems is also considered in [Ryu *et al.*, 1997], where the performance parameters are expressed as functions of the sampling periods and the input-output latencies. [Shin and Meissner, 1999] deals with on-line rescaling and relocation of control tasks in a multi-processor system. A simulator for co-design is introduced in [Eker and Cervin, 1999]. It facilitates co-simulation of control task execution, scheduling, and continuous plant dynamics.

The second area of related work is on quality-of-service (QoS) aware real-time software, where a system's resource allocation is adjusted on-line in order to maximize the performance in some respect. In [Li and Nahrstedt, 1998] a general framework is proposed for controlling the application requests for system resources using the amount of allocated resources for feedback. It is shown that a PID controller can be used to bound the resource usage in a stable and fair way. A resource allocation scheme called Q-RAM is presented in [Rajkumar *et al.*, 1997]. Several tasks are competing for finite resources, and each task is associated with a utility value, which is a function of the assigned resources. The system distributes the resources between the tasks to maximize the total utility of the system. In [Abdelzaher *et al.*, 1997] a QoS renegotiation scheme is proposed as a way to allow graceful degradation in cases of overload, failures or violation of pre-run-time assumptions. The mechanism permits clients to express, in their service requests, a range of QoS levels they can accept from the provider, and the perceived utility of receiving service at each of these levels.

The third area relates to the wealth of flexible scheduling algorithms available. An interesting alternative to linear task rescaling is given in [Buttazzo *et al.*, 1998], where an elastic task model for periodic tasks is presented. The relative sensitivity of tasks to rescaling are expressed in terms of elasticity coefficients. Schedulability analysis of the system under EDF scheduling is given. Closely related to our work, [Stankovic *et al.*, 1999] presents a scheduling algorithm that explicitly uses feedback. A PID controller regulates the deadline miss-ratio for a set of soft real-time tasks with varying execution times, by adjusting their requested CPU utilization. It is assumed that tasks can change their CPU consumption by executing different versions of the same algorithm. An admission controller is used to accommodate larger changes in the workload.

1.2 Outline

The rest of the paper is outlined as follows. Section 2 describes a hybrid controller for a double-tank process. Section 3 applies the feedback scheduling principle to a system with three hybrid controllers. The design and implementation are discussed and simulation results are given. Finally, Section 4 contains the conclusions.

2. A Hybrid Controller

A hybrid controller for the double-tank process, see Figure 2, is described. The controller was designed and implemented in [Eker and Malmborg, 1999]. The goal is to control the level of the lower tank to a

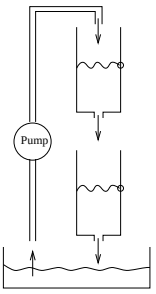


Figure 2 The double-tank process.

desired setpoint. The measurement signals are the levels of both tanks, and the control signal is the inflow to the upper tank. Choosing state variables $x_1(t)$ for the upper tank level and $x_2(t)$ for the lower tank level, we get the nonlinear state-space description

$$\frac{dx}{dt} = \begin{bmatrix} -\alpha\sqrt{x_1(t)} + \beta u(t) \\ \alpha\sqrt{x_1(t)} - \alpha\sqrt{x_2(t)} \end{bmatrix} \quad (1)$$

The process constants α and β depend on the cross-sections of the tanks, the outlet areas, and the capacity of the pump. The control signal $u(t)$ is limited to the interval [0,1].

Traditionally there is a trade-off in design objectives when choosing controller parameters. It is usually hard to achieve the desired step-change response and at the same time get the wanted steady-state behavior. An example of contradictory design criteria is tuning a PID controller to achieve both fast response to setpoint changes, fast disturbance rejection, and no or little overshoot. In process control it is common practice to use PI control for steady state regulation and to use manual control for large setpoint changes. One solution to this problem is to use a hybrid controller consisting of two sub-controllers, one PID controller and one time-optimal controller, together with a switching scheme. The time-optimal controller is used when the states are far away from the reference point. Coming close, the PID controller will automatically be switched in to replace the time optimal controller.

The sub-controller designs are based on a linearization of Equation (1).

$$\frac{dx}{dt} = \begin{bmatrix} -a & 0 \\ a & -a \end{bmatrix} x(t) + \begin{bmatrix} b \\ 0 \end{bmatrix} u(t) \quad (2)$$

The new process parameters a and b are functions of α, β and the current linearization level.

2.1 PID Controller

The PID parameters (K, T_i, T_d) are calculated to give the closed-loop characteristic polynomial

$$(s+\omega_0)(s^2 + 2\zeta\omega_0 s + \omega_0^2) \quad (3)$$

where $(\omega_0, \zeta) = (6, 0.7)$ are chosen to give good rejection of load disturbances. The following discrete-time implementation, which includes low-pass filtering of the derivative part ($N = 10$), is used:

$$\begin{aligned} P(t) &= K(y_{sp}(t) - y(t)) \\ I(t) &= I(t-h) + \tfrac{Kh}{T_i}(y_{sp}(t) - y(t)) \\ D(t) &= \tfrac{T_d}{Nh+T_d}D(t-h) + \tfrac{NKT_d}{Nh+T_d}(y(t-h) - y(t)) \\ u(t) &= P(t) + I(t) + D(t) \end{aligned}$$

2.2 Time-Optimal Controller

The time-optimal control signal is of bang-bang type. For the linearized process it is possible to derive the switching curve

$$x_2(x_1) = \frac{1}{a}((ax_1 - b\overline{u})(1 + \ln(\frac{ax_1^R - b\overline{u}}{ax_1 - b\overline{u}})) + b\overline{u})$$

where $\overline{u}$ takes values in $\{0, 1\}$, and x_1^R is the target state for x_1. The control signal is $u = 0$ above the switching curve and $u = 1$ below. A closeness criterion on the form

$$V_{close} = \begin{bmatrix} x_1^R - x_1 \\ x_2^R - x_2 \end{bmatrix}^T P(\theta, \gamma) \begin{bmatrix} x_1^R - x_1 \\ x_2^R - x_2 \end{bmatrix}$$

where $P(\theta, \gamma)$ is positive definite matrix, is evaluated at each sample, to determine whether the controller should switch to PID mode.

2.3 Implementation

The controller implementation is outlined below.

```
y = analogIn(yChan);
ysp = analogIn(yspChan);
if (getMode() == PID) {
  if (ysp != ysp_old) {
    setMode(OPT);
    signal(FBS_sem); /* feedforward, see Sec 3.3 */
    u = calculateOPT();
  } else {
    u = calculatePID();
  }
} else { /* OPT */
  Vclose = computeVclose();
  if (Vclose < Vregion) {
    setMode(PID);
    u = calculatePID();
  } else {
    u = calculateOPT();
  }
}
analogOut(uChan,u);
```

2.4 Real-Time Properties

The execution-time properties of the hybrid controller were investigated in [Persson *et al.*, 2000].

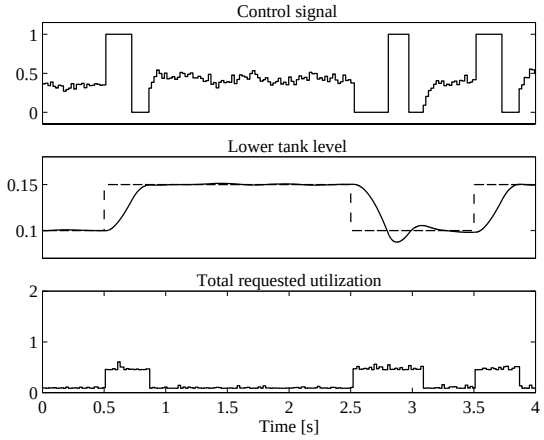


Figure 3 The single-controller case.

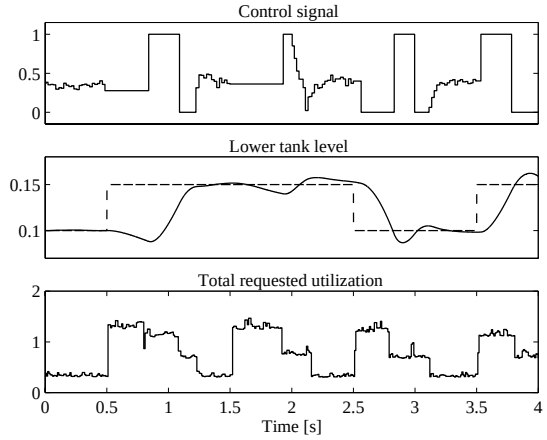


Figure 4 Open-loop scheduling.

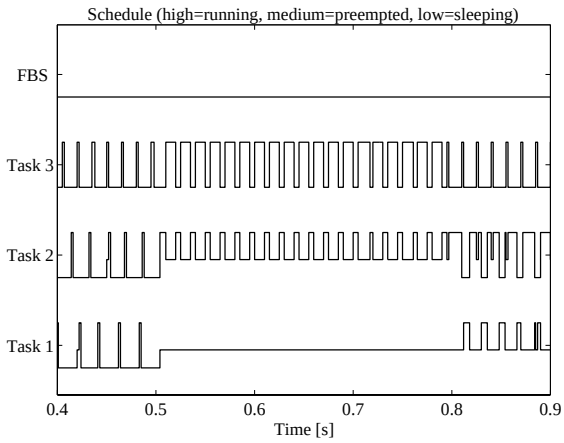


Figure 5 Close-up of the schedule under open-loop scheduling.

It was found that the optimal-control mode had considerable longer execution time than the PID mode. In each mode, the execution time was close to the best case most of the time, but it also exhibited random bursts. For purposes of illustration, assume that the execution-time characteristics in the different modes can be described by $C_{PID} = 1.8 + 0.2\varepsilon_i^2$ ms and $C_{Opt} = 9.5 + 0.5\varepsilon_i^2$ ms, where $\{\varepsilon_i\}$ is unit-variance Gaussian white noise.

The nominal sampling interval is chosen to be one tenth of the rise time, T_r, of the closed-loop system. Our first example process has $T_{r1} = 210$ ms which gives $h_{nom\,1} = 21$ ms. A simulation of the computer-control system is found in Figure 3. The controller displays very good set-point response and steady-state regulation. It is seen that the requested CPU utilization is very low in PID mode, on average $\overline{U} = \overline{C}_{PID}/h_{nom\,1} = 9\%$. In Opt mode, it is significantly higher, on average $\overline{U} = \overline{C}_{Opt}/h_{nom\,1} = 45\%$.

3. Feedback Scheduling Example

Now assume that two additional hybrid double-tank controllers should execute on the same CPU as the first one. The tanks have slightly different process parameters. Based on the rise-times, $(T_{r2}, T_{r3}) = (180, 150)$ ms, they are assigned the nominal sampling intervals $(h_{nom\,2}, h_{nom\,3}) = (18, 15)$ ms. To consider scheduling, some assumptions about the real-time operating system must be made. Throughout this example, we assume a fixed-priority real-time kernel with the possibility to measure task execution time. The tasks are assigned rate-monotonic priorities, i.e., the task with the shortest period gets the highest priority.

First, open-loop scheduling is attempted. Then, a feedback scheduler is added to the system. Finally, feedforward is introduced in the scheduler. The systems are evaluated by co-simulation of the real-time kernel and the plant dynamics [Eker and Cervin, 1999]. A 4-second simulation cycle of is constructed as follows. At time $t = 0$, all controllers start in the PID mode. At time $t = 0.5$ s, the worst-case case scenario appears: all controllers receive new setpoints and should switch to Opt mode. Following this, the controllers get new setpoints pairwise, and then one by one. For each simulation, the behavior of Controller 1, now being the lowest-priority controller, is plotted. Also plotted is the total requested utilization, $\sum_i c_i/h_i$, where c_i is the current actual execution time of task i, and h_i is the current period of task i. Notice that this signal cannot be directly measured and used for feedback, but must be estimated.

3.1 Open-loop scheduling

We first consider open-loop scheduling, where the controllers are implemented as tasks with fixed periods equal to their nominal sampling intervals.

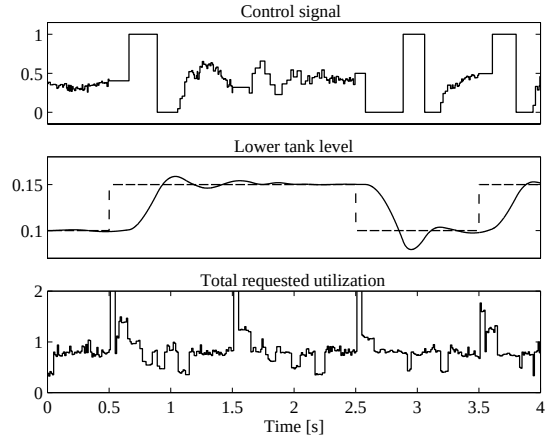


Figure 6 Feedback scheduling.

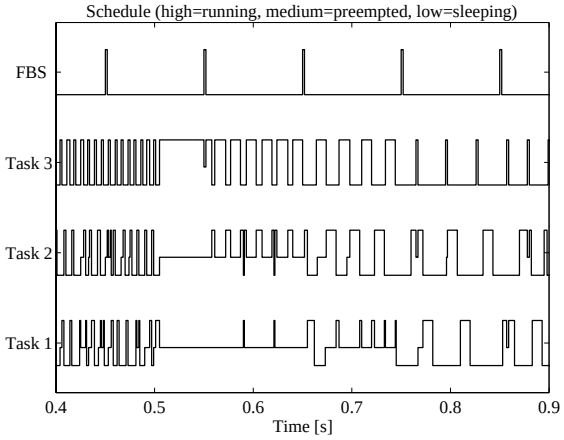


Figure 7 Close-up of the schedule under feedback scheduling.

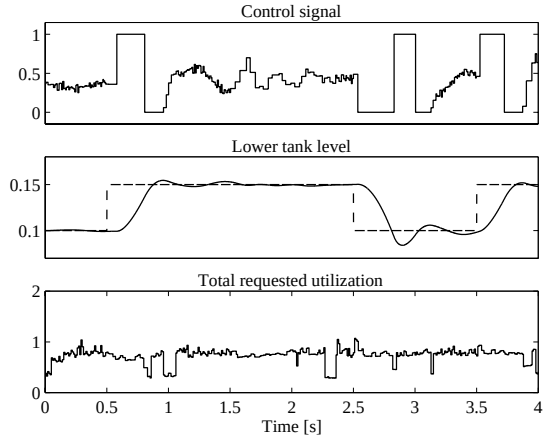


Figure 8 Feedback + feedforward scheduling.

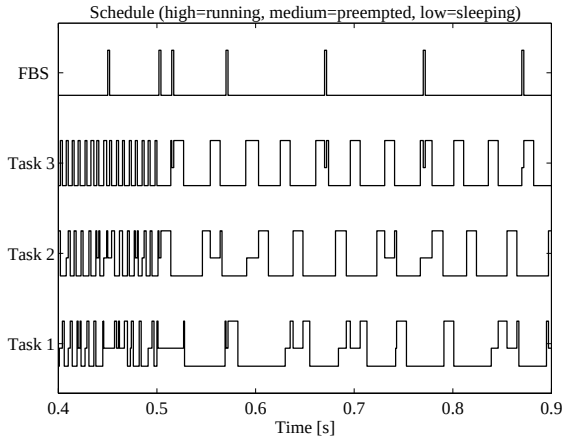


Figure 9 Close-up of the schedule under feedback + feedforward scheduling.

The simulation results are shown in Figures 4 and 5. The system easily becomes overloaded, since in the worst case, $\overline{U} = \sum_i \overline{C}_{Opt}/h_{nom\,i} = 170\%$. Controller 1 is for instance temporarily turned off in the intervals $t = [0.5, 0.8]$ s and $t = [1.5, 1.8]$ s because of preemption. The result is low control performance.

3.2 Feedback scheduling

Next, a feedback scheduler is introduced. In its first version, it is implemented as a high-priority task with a period $T_{\mathrm{FBS}} = 100$ ms. The utilization setpoint is set to $U_{sp} = 80\%$. At each invocation, the feedback scheduler estimates the current total requested utilization of the tasks by computing $\hat{U} = \sum_i \hat{C}_i/h_i$. The estimate $\hat{C}_i$ is obtained from filtered execution-time measurements,

$$\hat{C}_i(k) = \lambda \hat{C}_i(k-1) + (1-\lambda)c_i$$

where λ is a forgetting factor. Setting λ close to 1 results in a smooth, but slow estimate. In this case, $\lambda = 0$, which gives fast detection of overloads, was preferred. Finally, new task periods are assigned according to the linear rescaling

$$h_i = h_{nom\,i}\hat{U}/U_{sp}$$

The execution time of the feedback scheduler is assumed to be 2 ms. The simulation results are shown in Figures 6 and 7. The scheduler tries to keep the workload close to 80%. However, there is a delay from a change in the requested utilization until it is detected by the feedback scheduler. This results in overload peaks at some of the mode change instants. For instance, Controller 1 is preempted in the interval $t = [0.5, 0.6]$ s. The result is slightly degraded control performance.

3.3 Feedback + feedforward scheduling

A feedforward mechanism is added to the scheduler. The basic period of the scheduler is kept at $T_{\mathrm{FBS}} = 100$ ms. However, when a task in PID mode

detects a new setpoint, it notifies the feedback scheduler, which is released immediately. The task periods are adjusted before the notifying task can continue to execute in the Opt mode. The execution-time estimation can also benefit from the mode-change information, by running separate estimators in the different modes. A forgetting factor of $\lambda = 0.9$ was chosen to give smooth estimates in both modes. The result is a more responsive and accurate feedback scheduler. The simulation results are shown in Figures 8 and 9. It is seen that the delay for Controller 1 at time $t = 0.5$ s has been reduced, and that the control performance is slightly better.

3.4 Performance Evaluation

The performance of the controllers under different scheduling policies are evaluated using the criterion

$$V_i(t) = \int_0^t (y_{nom\,i}(\tau) - y_{actual\,i}(\tau))^2 d\tau$$

where $y_{nom\,i}$ is the process output when the controller is running unpreempted at its nominal sampling interval, and $y_{actual\,i}$ is the actual process output when the controller is running in the multi-task real-time system. V is referred to as the additional loss due to scheduling. 25 cycles (100 s) are simulated and the final loss for the different controllers are summarized below:

Scheduling	$V_1(100)$	$V_2(100)$	$V_3(100)$
Open-loop	$4.2 \cdot 10^{-2}$	$2.0 \cdot 10^{-3}$	$5.1 \cdot 10^{-4}$
Feedback	$5.0 \cdot 10^{-3}$	$3.2 \cdot 10^{-3}$	$1.0 \cdot 10^{-3}$
Feedback+ feedforward	$1.5 \cdot 10^{-3}$	$1.1 \cdot 10^{-3}$	$1.0 \cdot 10^{-3}$

The evolution of the additional loss for Controller 1 is shown in Figure 10. There is a big improvement when introducing feedback, and feedforward gives even further improvement.

4. Conclusions

The feedback scheduler presented increases the control performance and relaxes the requirements of known executions times for multi-task control systems. The controllers are allowed to miss an occasional deadline, and are hence not treated as hard real-time tasks. In case of overload, the scheduler calculates new sampling periods for all control tasks. The estimate of the current workload is based on execution-time measurements. The new sampling periods are given by simple linear rescaling of the nominal sampling periods, i.e., the relative importance order of the controllers is preserved. A more elaborate rescaling procedure would most likely give better control performance but also require more computational power. The feedback scheduler itself is implemented as a task, and its period is an important design parameter.

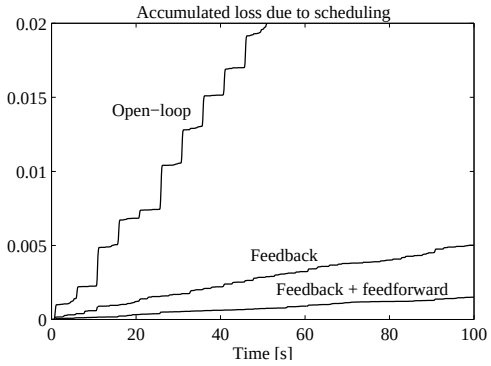


Figure 10 The accumulated additional loss due to scheduling for Controller 1, $V_1(t)$.

5. References

Abdelzaher, T., E. Atkins, and K. Shin (1997): "QoS negotiation in real-time systems, and its application to flight control." In *Proceedings of the IEEE Real-Time Systems Symposium*.

Buttazzo, G., G. Lipari, and L. Abeni (1998): "Elastic task model for adaptive rate control." In *Proceedings of the IEEE Real-Time Systems Symposium*.

Eker, J. and A. Cervin (1999): "A Matlab toolbox for real-time and control systems co-design." In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pp. 320–327. Hong Kong, P.R. China.

Eker, J. and J. Malmborg (1999): "Design and implementation of a hybrid control strategy." *IEEE Control Systems Magazine*, **19:4**.

Li, B. and K. Nahrstedt (1998): "A control theoretic model for quality of service adaptations." In *Proceedings of Sixth International Workshop on Quality of Service*.

Persson, P., A. Cervin, and J. Eker (2000): "Execution time properties of a hybrid controller." Technical Report. Department of Computer Science, Lund Institute of Technology, Lund, Sweden.

Rajkumar, R., C. Lee, J. Lehoczky, and D. Siewiorek (1997): "A resources allocation model for QoS management." In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*.

Ryu, M., S. Hong, and M. Saksena (1997): "Streamlining real-time controller design: From performance specifications to end-to-end timing constraints." In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*.

Seto, D., J. P. Lehoczky, L. Sha, and K. G. Shin (1996): "On task schedulability in real-time control systems." In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pp. 13–21.

Shin, K. and C. Meissner (1999): "Adaptation of control system performance by task reallocation and period modification." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 29–36.

Stankovic, J. A., C. Lu, S. H. Son, and G. Tao (1999): "The case for feedback control real-time scheduling." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 11–20.

Schedulability Analysis for Systems with Data and Control Dependencies

Paul Pop, Petru Eles, and Zebo Peng
Dept. of Computer and Information Science,
Linköping University, Sweden
{paupo, petel, zebpe}@ida.liu.se

Abstract

In this paper we present an approach to schedulability analysis for hard real-time systems with control and data dependencies. We consider distributed architectures consisting of multiple programmable processors, and the scheduling policy is based on a static priority preemptive strategy. Our model of the system captures both data and control dependencies, and the schedulability approach is able to reduce the pessimism of the analysis by using the knowledge about control and data dependencies. Extensive experiments as well as a real life example demonstrate the efficiency of our approach.

1. Introduction

Depending on the particular application, a real-time system has certain requirements on performance, cost, dependability, size, etc. For hard real-time applications the timing requirements are extremely important. Thus, in order to function correctly, a real-time system implementing such an application has to meet its deadlines.

In this paper we present an approach to schedulability analysis for hard real-time systems that have both data and control dependencies. We consider systems that are implemented on distributed architectures consisting of multiple programmable processors and, in our approach, the system is modeled through a so called *conditional process graph* (CPG) [3]. Such a graph captures both the flow of data and that of control. Process scheduling is based on a static priority preemptive approach.

Process scheduling for performance estimation and synthesis of real-time systems has been intensively researched in the last years. The existing approaches differ in the scheduling strategy adopted, system architectures considered, handling of the communication, and process interaction aspects.

Static non-preemptive scheduling of a set of processes on a multiprocessor system has been discussed in [3, 5, 7, 12]. Preemptive scheduling of independent processes with static priorities running on single processor architectures has its roots in [9]. The approach has been later extended to accommodate more general system models and has been also applied to distributed systems [15]. The reader is referred to [1] for a survey on this topic.

In many of the previous scheduling approaches researchers have assumed that processes are scheduled independently. However, this is not the case in reality, where process sets can exhibit both data and control dependencies. Moreover, knowledge about these dependencies can be used in order to improve the accuracy of schedulability analyses and the quality of produced schedules.

Static cyclic scheduling of processes with both data and control dependencies has been addressed by us in [2, 3]. We have discussed the particular aspects concerning communication in such distributed systems in [11, 12].

One way of dealing with data dependencies between processes with static priority based scheduling has been indirectly addressed by the extensions proposed for the schedulability analysis of distributed systems through the use of the *release jitter* [15]. Release jitter is the worst case delay between the arrival of a process and its release (when it is placed in the run-queue for the processor) and can include the communication delay due to the transmission of a message on the communication channel.

[14] and [16] use *time offset* relationships and *phases*, respectively, in order to model data dependencies. Offset and phase are similar concepts that express the existence of a fixed interval in time between the arrivals of sets of processes. The authors show that by introducing such concepts into the computational model, the pessimism of the analysis is significantly reduced when bounding the time behaviour of the system.

When control dependencies exist then, depending on conditions, only a subset of the set of processes is executed during an invocation of the system. *Modes* have been used to model a certain class of control dependencies [4]. Such a model basically assumes that at the starting of an execution cycle, a particular functionality is known in advance and is fixed for one or several cycles until another mode change is performed. However, modes cannot handle fine grained control dependencies, or certain combinations of data and control dependencies. Careful modeling using the *periods* of processes (lower bound between subsequent re-arrivals of a process) can also be a solution for some cases of control dependencies [6]. If, for example, we know that a certain set of processes will only execute every second cycle of the system, we can set their periods to the double of the period of the rest of the processes in the system. However, using the worst case assumption on periods leads very often to unnecessarily pessimistic schedulability evaluations. A more refined process model can produce much better schedulability results, as will be shown later.

We propose in the next section a system model based on a conditional process graph that is able to capture both

data and control dependencies. Then, we introduce a less pessimistic schedulability analysis technique in order to bound the response time of a hard real-time system modeled in such a way. In this paper we insist on various aspects concerning dependencies between processes. Other issues like communication protocol, bus arbitration, packaging of messages, clock synchronization, as discussed by us in [11], can easily be included in the analysis.

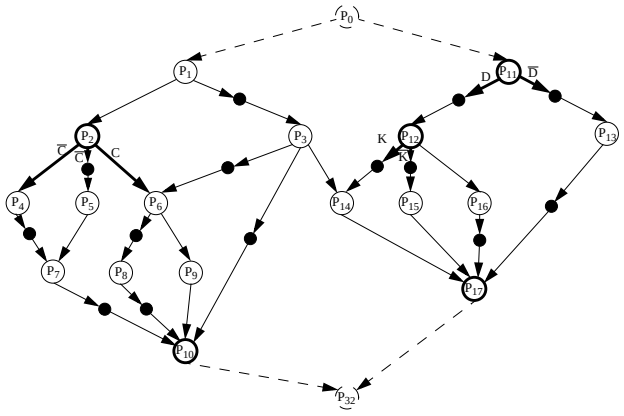
This paper is divided into 7 sections. The next section presents our graph-based abstract system representation. Section 3 formulates the problem and sections 4 and 5 present the schedulability analyses proposed. The techniques are evaluated in section 6, and section 7 presents our conclusions.

2. Conditional Process Graph

As an abstract model for system representation we use a directed, acyclic, polar graph $\Gamma(V, E_S, E_C)$. Each node $P_i \in V$ represents one process. Such a process can be an "ordinary" process specified by the designer or a so called *communication process* which captures the message passing activity. E_S and E_C are the sets of simple and conditional edges respectively. $E_S \cap E_C = \emptyset$ and $E_S \cup E_C = E$, where E is the set of all edges. An edge $e_{ij} \in E$ from P_i to P_j indicates that the output of P_i is the input of P_j. The graph is polar, which means that there are two nodes, called *source* and *sink*, that conventionally represent the first and last process. These nodes are introduced as dummy processes so that all other nodes in the graph are successors of the source and predecessors of the sink respectively.

We consider a distributed architecture consisting of several *processors* connected through *busses*. These busses can be shared by several communication channels connecting processes assigned to different processors.

We assume that each process is assigned to a processor and each communication channel which connects processes assigned to different processors is assigned to a bus.



Process mapping
Processor pe_1: P_1, P_2, P_4, P_6, P_9, P_{10}, P_{13}
Processor pe_2: P_3, P_5, P_7, P_{11}, P_{14}, P_{15}, P_{17}
Processor pe_3: P_8, P_{12}, P_{16}
Communications are mapped to a unique bus

Figure 1. Conditional Process Graph

The mapping of processes to processors and busses is given by a function M: $V \rightarrow PE$, where $PE=\{pe_1, pe_2, .., pe_{Npe}\}$ is the set of processing elements (processors and busses). For any process P_i, $M(P_i)$ is the processing element to which P_i is assigned for execution.

Each process P_i, assigned to processor or bus $M(P_i)$, is characterized by a worst case execution time C_i. In the process graph depicted in Figure 1, P_0 and P_{32} are the source and sink nodes respectively. Nodes denoted $P_1, P_2, .., P_{17}$, are "ordinary" processes specified by the designer. Figure 1 also shows the mapping of processes to three different processors. The communication processes are represented in Figure 1 as solid circles and are introduced for each connection which links processes mapped to different processors. In this paper we do not consider the message passing aspects which we have analyzed in [11, 12].

An edge $e_{ij} \in E_C$ is a *conditional edge* (thick lines in Figure 1) and it has an associated condition. Transmission on such an edge takes place only if the associated condition is satisfied. We call a node with conditional edges at its output a *disjunction node* (and the corresponding process a *disjunction process*). Alternative paths starting from a disjunction node, which correspond to a certain condition, are disjoint and they meet in a so called *conjunction node* (with the corresponding process called *conjunction process*). *Conditions are dynamically computed by disjunction processes and their value is unpredictable at the start of an execution cycle of the conditional process graph*. In Figure 1 circles representing conjunction and disjunction nodes are depicted with thick borders. We assume that conditions are independent.

According to our model, we assume that a process, which is not a conjunction process, can be activated only after all its inputs have arrived. A conjunction process can be activated after messages coming on one of the alternative paths have arrived. All processes issue their outputs when they terminate. If we consider the activation time of the source process as a reference, the finishing time of the sink process is the delay of the system at a certain execution.

3. Problem Formulation

An application is modeled as a set Ψ of n conditional process graphs Γ_i, $i = 1..n$. Every process P_i in such a graph is mapped to a certain processor, has a known worst-case execution time C_i, a deadline D_i, and a uniquely assigned priority. All processes belonging to the same CPG Γ_i have the same period $T_{\Gamma i}$ which is the period of the respective conditional process graph. Each CPG in the application has its own independent period. Typically, global deadlines $\delta_{\Gamma i}$ on the delay of each CPG are imposed and not individual deadlines on processes.

We consider a priority based preemptive execution environment, which means that higher priority processes will interrupt the execution of lower priority processes. A lower priority process can block a higher priority process (e.g., it is in its critical section), and the blocking time is

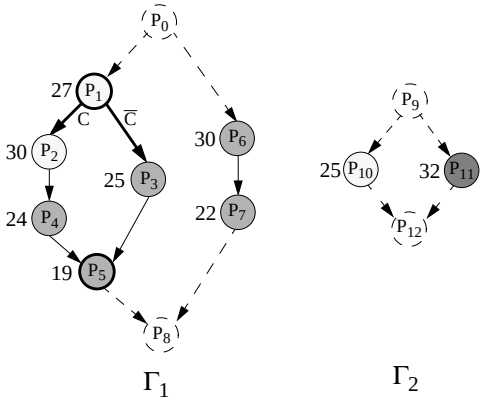


Figure 2. System with Control and Data Dependencies

computed according to the priority ceiling protocol [13].

We are interested to develop a schedulability analysis for a system modeled as a set of conditional process graphs. For the rest of the paper we will consider that global deadlines are imposed on each CPG. The approach can be easily extended if individual deadlines are imposed on processes.

To show the relevance of our problem, let us consider the example depicted in Figure 2, where we have a system modeled as two conditional process graphs Γ_1 and Γ_2 with a total of 9 processes (processes P_0, P_8, P_9 and P_{12} are dummy processes and are not counted), and one condition. The processes are mapped on three different processors as indicated by the shading in Figure 2, and the worst case execution time in milliseconds for each process on its respective processor is depicted to the left of each node. Γ_1 has a period of 200 ms, Γ_2 has a period of 150 ms. The deadlines are 100 ms on Γ_1 and 90 ms on Γ_2.

Table 1 presents the estimated worst case delay on the two graphs. In the column labeled “no conditions” we have the results for the case when the analysis is applied to the set of processes, ignoring control dependencies. This results in a worst case delay of 120 ms for Γ_1 and 82 ms for Γ_2. Thus, the system is considered to be not schedulable.

However, this analysis assumes as a worst case scenario the possible activation of all nine processes for each execution of the system. This is the solution which will be obtained using a dataflow graph representation of the system. However, considering the CPG Γ_1 in Figure 2, it is easy to observe that process P_3 on the one side and processes P_2 and P_4 on the other side will not be activated during the same period of Γ_1.

Making use of this information for the analysis we obtain a worst case delay of 100 ms, for Γ_1, as shown in Table 1 in the column headed “conditions”, which indicates that the system is schedulable.

CPG	Worst Case Delays	
	no conditions	conditions
Γ_1	120	100
Γ_2	82	82

Table 1: Worst Case Delays for the System in Fig. 2

4. Schedulability Analysis for Task Graphs with Data Dependencies

Methods for schedulability analysis of data dependent processes with static priority preemptive scheduling have been proposed in [14] and [16].

They use the concept of *offset* or *phase*, respectively, in order to handle data dependencies. [14] shows that the pessimism of the analysis is reduced through the introduction of offsets. The offsets have to be determined by the designer.

[16] provides a framework that iteratively finds the phases (offsets) for all processes, and then feeds them back into the schedulability analysis which in turn is used again to derive better phases. Thus, the pessimism of the analysis is iteratively reduced.

We have used the framework provided by [16] as a starting point for our analysis. The response time of a process P_i is:

$$r_i = C_i + \sum_{\forall j \in hp(P_i)} C_j \left\lceil \frac{r_i - O_{ij}}{T_j} \right\rceil \quad (1)$$

where $hp(P_i)$ is the set of processes that have higher priority than P_i, and O_{ij} is the phase of P_j relative to P_i.

As a first step we have extended this analysis to real-time systems that use the time-triggered protocol as the underlying communication infrastructure [12]. However, for the sake of simplicity, we do not consider the communication of messages in this paper.

In [16] a system is modeled as a set S of n *task graphs* G_i, $i = 1..n$. The system model assumed and the definition of a task graph are similar to our CPG , but without considering any conditions. The aim of the schedulability analysis in [16] is to derive an as tight as possible worst

```
DelayEstimate(task graph G, system S)
-- derives the worst case delay of a task graph G considering
-- the influence from all other task graphs in the system S
   for each pair (Pi, Pj) in G
      maxsep[Pi, Pj] = ∞
   end for
   step = 0
   repeat
      LatestTimes(G)
      EarliestTimes(G)
      for each Pi ∈ G
         MaxSeparations(Pi)
      end for
   until maxsep is not changed or step < limit
   return the worst case delay δG of the graph G
end DelayEstimate

SchedulabilityTest(system S)
-- derives the worst case delay for each task graph in the system
-- and verifies if the deadlines are met
   for each task graph Gi ∈ S
      DelayEstimate(Gi, S)
   end for
   if all task graphs meet their deadline system S is schedulable
end SchedulabilityTest
```

Figure 3. Delay Estimation and Schedulability Analysis for Task Graphs

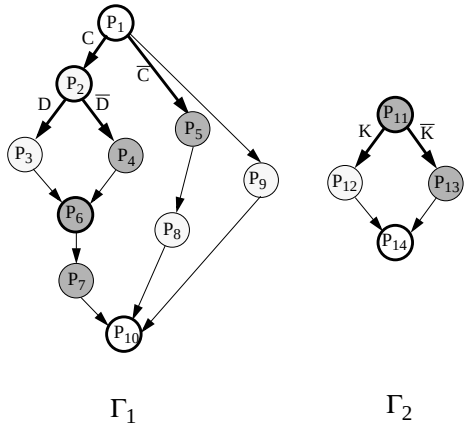


Figure 4. Example of two CPGs

case delay on the execution time of each of the task graphs in the system. This delay estimation is done using the algorithm DelayEstimate described in Figure 3.

At the core of this algorithm is a worst case response time calculation based on offsets, similar to the analysis in [14]. Thus, in the LatestTimes function worst case response times and upper bounds for the offsets are calculated, while the EarliestTimes function calculates the lower bounds of the offsets.

The LatestTimes function is a modified critical-path algorithm that calculates for each node of the graph the longest path to the sink node. Thus, during the topological traversal of the graph *G* within LatestTimes, for each process P_i, the worst case response time r_i is calculated according to the equation (1). This value is based on the values of the offsets known so far. Once an r_i is calculated, it can be used to determine and update offsets for other successor processes. Accordingly, the EarliestTimes function determines the lower bounds on the offsets. The influence on graph G from other graphs in the system is considered in both of the functions mentioned earlier.

These calculations can be improved by realizing that for a process P_i, there might exist a process P_j mapped on the same processor, with priority(P_i) < priority(P_j), such that their execution windows never overlap. In this case, the term in the equation (1) that expresses the influence of P_j on the execution of P_i can be dropped, resulting in a tighter worst case response time calculation. This situation is expressed through the so called *maxsep* table, computed by the MaxSeparations function, whose value *maxsep*[P_i, P_j] is less than or equal to 0 if the two processes never overlap during their execution. *maxsep* stands for *maximum separation*, an analysis modified from [10] that builds the *maxsep* table based on the worst case execution times and offsets determined in EarliestTimes and LatestTimes.

Having a better view on the maximum separation between each pair of processes, tighter worst case execution times and offsets can be derived, which in turn contribute to the update of the *maxsep* table. This iterative tightening process is repeated until there is no modification to the *maxsep* table, or a certain imposed *limit* on the number of iterations is reached.

Finally, the DelayEstimate function returns the worst-case delay δ_G estimated for a task graph *G*, as the latest time when the sink node of *G* can finish its execution. Based on the delays produced by DelayEstimate, the function SchedulabilityTest in Figure 3 concludes on the schedulability of the system.

5. Schedulability Analysis for CPGs

Before introducing our approach to schedulability analysis of conditional process graphs, two concepts have to be introduced: the *unconditional subgraphs* and the *process guards*.

Depending on the values calculated for the conditions, different alternative paths through a conditional process graph are activated for a given activation of the system. To model this, a boolean expression X_{Pi}, called *guard*, can be associated to each node P_i in the graph. It represents the necessary condition for the respective process to be activated. In Figure 4, for example, $X_{P4}=C\wedge\overline{D}$, $X_{P5}=\overline{C}$, $X_{P9}=true$, $X_{P11}=true$, and $X_{P12}=K$.

We call an alternative path through a conditional process graph, resulting from a combination of conditions, an *unconditional subgraph*, denoted by *g*. For example, the CPG Γ_1 in Figure 4 has three unconditional subgraphs, corresponding to the following three combinations of conditions: $C\wedge D$, $C\wedge\overline{D}$, and $\overline{C}$. The unconditional subgraph corresponding to the combination $C\wedge\overline{D}$ in the CPG Γ_1 consists of processes P_1, P_2, P_4, P_6, P_7, P_9 and P_{10}.

The guards of each process, as well as the unconditional subgraphs resulting from a conditional process graph Γ can be determined through a simple recursive topological traversal of Γ.

5.1 Ignoring Conditions (IC)

A straightforward approach to the schedulability analysis of systems represented as CPGs is to ignore control dependencies and to apply the schedulability analysis as described in section 4 (the algorithm SchedulabilityTest in Figure 3).

This means that conditional edges in the CPGs are considered like simple edges and the conditions in the model are dropped. What results is a system *S* consisting of simple task graphs G_i, each one resulted from a CPG Γ_i of the given system ψ. The system *S* can then be analyzed using the algorithm in Figure 5. It is obvious that if the system *S* is schedulable, the system ψ is also schedulable.

SA/IC(*system* ψ)
-- verifies the schedulability of a system consisting of a set of
-- conditional process graphs
transform each $\Gamma_i \in \psi$ into the corresponding $G_i \in S$
SchedulabilityTest(*S*)
if *S* is schedulable, system ψ is schedulable
end SA/IC

Figure 5. Schedulability Analysis Ignoring Conditions

```
DE/CPG(CPG Γ, system S)
-- derives the worst case delay of a CPG Γ considering
-- the influence from all other task graphs in the system S
   extract all unconditional subgraphs g_j from Γ
   for each g_j
      DelayEstimate(g_j, S)
   end for
   return the largest of the delays, which is
     the worst case delay δ_Γ of CPG Γ
end DE/CPG
```

a) DE/CPG -- Delay Estimate for Conditional Process Graphs

```
SA/BF(system ψ)
-- verifies the schedulability of a system consisting of a set ψ of
-- conditional process graphs
   transform each Γ_i ∈ ψ into the corresponding G_i ∈ S
   for each Γ_i ∈ ψ
      DE/CPG(Γ_i, {G_1, G_2, ...G_i-1, G_i+1, G_n})
   end for
   if all CPGs meet their deadline the system ψ is schedulable
end SA/BF
```

b) SA/BF -- Schedulability Analysis: the Brute Force approach

Figure 6. Brute Force Schedulability Analysis

This approach, which we call IC, is, of course, very pessimistic. However, this is the current practice when worst case arrival periods are considered and classical data flow graphs are used for modeling and scheduling.

5.2 Brute Force Solution (BF)

The pessimism of the previous approach can be reduced by using a conditional process graph model. A simple, brute force solution is to apply the schedulability analysis presented in section 4, after the CPGs have been decomposed into their constituent unconditional subgraphs.

Consider a system ψ which consists of n CPGs Γ_i, $i = 1..n$. Each CPG Γ_i can be decomposed into n_i unconditional subgraphs g_j^i , $j = 1..n_i$. In Figure 4, for example, we have 3 unconditional subgraphs g_1^1, g_2^1, g_3^1 derived from Γ_1 and two, g_1^2, g_2^2 derived from Γ_2.

At the same time, each CPG Γ_i can be transformed (as shown in subsection 5.1) into a simple task graph G_i, by transforming conditional edges into ordinary ones and dropping the conditions. When deriving the worst case delay on Γ_i we apply the analysis from section 4 (algorithm DelayEstimate in Figure 3) separately to each unconditional subgraph g_j^i in combination with the graphs (G_1, G_2, ...G_{i-1}, G_{i+1}, G_n). This means that we consider each alternative path from Γ_i in the context of the system, instead of the whole subgraph G_i as in the previous approach. This is described by the algorithm DE/CPG in Figure 6 a). The schedulability analysis is then based on the delay estimation for each CPG as shown in the algorithm SA/BF in Figure 6 b).

Such an approach, we call it BF, while producing tight bounds on the delays, can be expensive from the runtime point of view, because it is applied for each unconditional subgraph. In general, the number of unconditional subgraphs can grow exponentially. However, for many of the practical systems this is not the case, and the brute force method can be used. Alternatively, less expensive methods, like those presented below, should be applied.

5.3 Condition Separation (CS)

In some situations, the explosion of unconditional subgraphs makes the brute force method inapplicable. Thus, we need to find an analysis that is situated somewhere between the two alternatives discussed in 5.1 and 5.2, which means its should be not too pessimistic and should run in acceptable time.

A first idea is to go back to the DelayEstimate algorithm in Figure 3, and use the knowledge about conditions in order to update the *maxsep* table. Thus, if two processes P_i and P_j never overlap their execution because they execute under alternative values of conditions, then we can update $maxsep[P_i, P_j]$ to 0, and thus improve the quality of the delay estimation. Two processes P_i and P_j never overlap their execution if there exists at least one condition C, so that $C \subset X_{Pi}$ (X_{Pi} is the guard of process P_i) and $\overline{C} \subset X_{Pj}$.

In this approach, called CS, we practically use the same algorithm as for ordinary task graphs and try to exploit the information captured by conditional dependencies in order to exclude certain influences during the analysis. In Figure 7 we show the algorithm SA/CS which performs the schedulability analysis based on this heuristic.

```
SA/CS(system ψ)
-- verifies the schedulability of a system consisting of a set ψ of
-- conditional process graphs
   transform each Γ_i ∈ ψ into the corresponding G_i ∈ S
     and keep guard X_Pi for each P_i
   for each G_i ∈ S
      -- derives the worst case delay of a task graph G_i considering
      -- the influence from all other task graphs in the system S
      for each pair (P_i , P_j) in G_i
         maxsep[P_i, P_j] = ∞
      end for
      step = 0
      repeat
         LatestTimes(G_i)
         EarliestTimes(G_i)
         for each P_i ∈ G_i
            MaxSeparations(P_i)
         end for
         for each pair (P_i , P_j) in G_i
            if ∃C, C ⊂ X_Pi ∧ C̄ ⊂ X_Pj then
               maxsep[P_i, P_j] = 0
            end if
         end for
      until maxsep is not changed or step < limit
   δ_Γi is the worst case delay for Γ_i
   end for
   if all CPGs meet their deadline, the system ψ is schedulable
end SA/CS
```

Figure 7. Schedulability Analysis using Condition Separation

```
DelayEstimateRT1(task graph G, system S)
    LatestTimes(G)
end DelayEstimateRT1
```

a) Delay Estimation for RT1

```
DelayEstimateRT2(task graph G, system S)
    for each pair (Pi , Pj) in Gi
        maxsep[Pi, Pj] = ∞
    end for
    LatestTimes(G)
    EarliestTimes(G)
    for each Pi ∈ G
        MaxSeparations(Pi)
    end for
    LatestTimes(G)
end DelayEstimateRT2
```

a) Delay Estimation for RT2

Figure 8. Delay Estimation for the RT Approaches

5.4 Relaxed Tightness Analysis (RT)

The two approaches discussed here are similar to the brute force algorithm (Figure 6) presented in section 5.2. However, they try to improve on the execution time of the analyses by reducing the complexity of the DelayEstimate algorithm (Figure 3) which is called from the DE/CPG function (Figure 6 a). This will reduce the execution time of the analysis, not by reducing the number of subgraphs which have to be visited (like in section 5.3), but by reducing the time needed to analyze each subgraph. As our experimental results show (section 6) this approach can be very effective in practice. Of course, by the simplifications applied to DelayEstimate the quality of the analysis is reduced in comparison to the brute force method.

We have considered two alternatives of which the first one is more drastic while the second one is trying a more refined trade-off between execution time and quality of the analyses.

With both these approaches, the idea is not to run the iterative tightening loop in DelayEstimate that repeats until no changes are made to *maxsep* or until the *limit* is reached. While this tightening loop iteratively reduces the pessimism when calculating the worst case response times, the actual calculation of the worst case response times is done in LatestTimes, and the rest of the algorithm in Figure 3 just tries to improve on these values. For the first approach, called RT1 the function DelayEstimate has been transformed like in Figure 8 a).

However, it might be worth using at least the MaxSeparations in order to obtain tighter values for the worst case response times. For the alternative RT2 in Figure 8 b), DelayEstimateRT2 first calls LatestTimes and EarliestTimes, then MaxSeparations in order to build the *maxsep* table, and again LatestTimes to tighten the worst case response times.

6. Experimental Results

We have performed several experiments in order to evaluate the different approaches proposed. The two main aspects we were interested in are the quality of the schedulability analysis and the scalability of the algorithms for large examples. A first set of massive experiments were performed on conditional process graphs generated for experimental purpose.

We considered architectures consisting of 2, 4, 6, 8 and 10 processors. 40 processes were assigned to each node, resulting in graphs of 80, 160, 240, 320 and 400 processes, having 2, 4, 6, 8 and 10 conditions, respectively. The number of unconditional subgraphs varied for each graph dimension depending on the number of conditions and the randomly generated structure of the CPGs. For example, for CPGs with 400 processes, the maximum number of unconditional subgraphs is 64.

30 graphs were generated for each graph dimension, thus a total of 150 graphs were used for experimental evaluation. Worst case execution times were assigned randomly using both uniform and exponential distribution. All experiments were run on a Sun Ultra 10 workstation.

In order to compare the quality of the schedulability approaches, we need a cost function that captures, for a certain system, the difference in quality between the schedulability approaches proposed. Our cost function is the difference between the deadline and the estimated worst case delay of a CPG, summed for all the CPGs in the system:

$$\text{cost function} = \sum_{i=1}^{n} (D_{\Gamma_i} - \delta_{\Gamma_i})$$

where n is the number of CPGs in the system, $\delta_{\Gamma i}$ is the estimated worst case delay of the CPG Γ_i, and $D_{\Gamma i}$ is the deadline on Γ_i. A higher value for this cost function, for a given system, means that the corresponding approach produces better results (schedulability analysis is less pessimistic).

For each of the 150 generated example systems and each of the five approaches to schedulability analysis we have calculated the cost function mentioned previously, based on results produced with the algorithms described in section 5. These values, for a given system, differ from one analysis to another, with the BF being the least pessimistic approach and therefore having the largest value for the cost function.

We are interested to compare the five analyses, based on the values obtained for the cost function. Thus, Figure 9 a) presents the average percentage deviations of the cost function obtained in each of the five approaches, compared to the value of the cost function obtained with the BF approach. A smaller value for the percentage deviation means a larger cost function, thus a better result. The percentage deviation is calculated according to the formula:

$$\text{deviation} = \frac{\text{cost}_{BF} - \text{cost}_{approach}}{\text{cost}_{BF}} \cdot 100$$

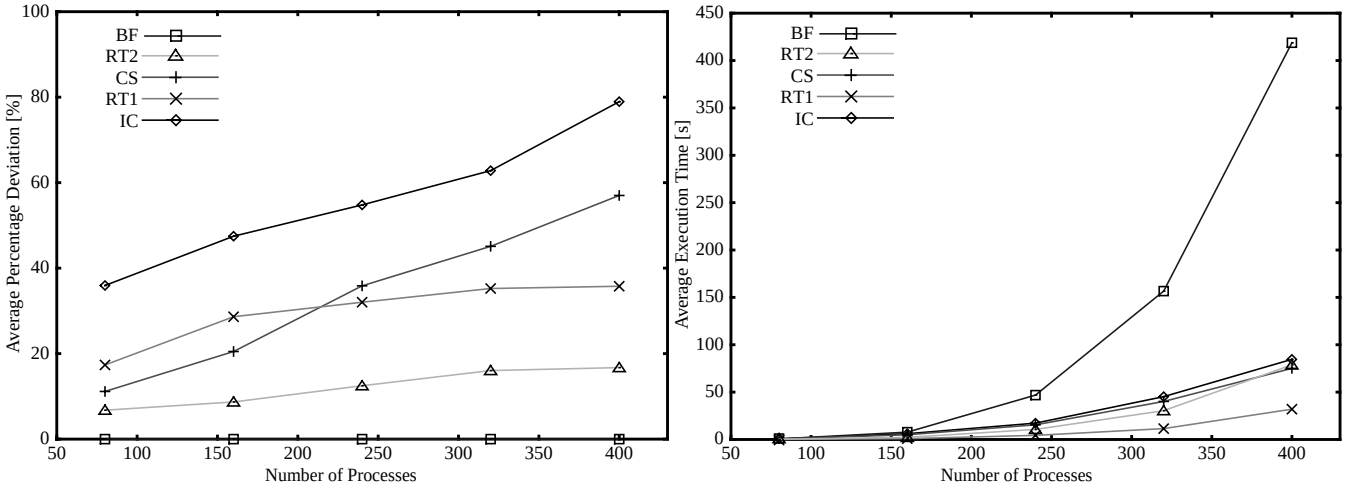


Figure 9. a) Average Percentage Deviation (left) and b) Average Execution Time (right) for each of the five analyses

Figure 9 b) presents the average runtime of the algorithms, in seconds.

The brute force approach, BF, performs best in terms of quality and obtains the largest values for the cost function of the systems at the expense of a large execution time. The execution time can be up to 7 minutes for large graphs of 400 processes, 10 conditions, and 64 unconditional subgraphs. At the other end, the straightforward approach IC that ignores the conditions, performs worst and becomes more and more pessimistic as the system size increases. As can be seen from Figure 9 a), IC has even for smaller systems of 160 processes (3 conditions, maximum 8 unconditional subgraphs) a 50% worse quality than the brute force approach, with almost 80% loss in quality, in average, for large systems of 400 processes. It is interesting to mention that the low quality IC approach has also an average execution time which is equal or comparable to the much better quality heuristics (except the BF, of course). This is because it tries to improve on the worst case delays through the iterative loop presented in DelayEstimate, Figure 3.

Let us turn our attention to the three approaches CS, RT1, and RT2 that, like the BF, consider conditions during the analysis but also try to perform a trade-off between quality and execution time. Figure 9 shows that the pessimism of the analysis is dramatically reduced by considering the conditions during the analysis. The RT1 and RT2 approaches, that visit each unconditional subgraph, perform in average better than the CS approach that considers condition separation for the whole graph. However, CS is comparable in quality with RT1, and even performs better for graphs of size smaller than 240 processes (4 conditions, maximum 16 subgraphs).

The RT2 analysis that tries to improve the worst case response times using the MaxSeparations, as opposed to RT1, performs best among the non-brute-force approaches. As can be seen from Figure 9, RT2 has less than 20% average deviation from the solutions obtained with the brute force approach. However, if faster runtimes are needed, RT1 can be used instead, as it is twice faster in execution time than RT2.

We were also interested to compare the five approaches with respect to the number of unconditional subgraphs in a system. For the results depicted in Figure 10 we have assumed CPGs consisting of 2, 4, 8, 16, and 32 unconditional subgraphs of maximum 50 processes each, allocated to 8 processors. Figure 10 shows that as the number of subgraphs increases, the differences between the approaches grow while the ranking among them remains the same, as resulted from Figure 9. The CS approach performs better than RT1 with a smaller number of subgraphs, but RT1 becomes better as the number of subgraphs in the CPGs increases.

Finally, we considered a real-life example implementing a vehicle cruise controller modeled using a conditional process graph. The graph has 32 processes, two conditions (4 subgraphs), and it was mapped on an architecture consisting of 4 nodes (processors), namely: Anti Blocking System, Transmission Control Module, Engine Control Module and Electronic Throttle Module. The period of the CPG was 200 ms, and the deadline was set to 110 ms. Without considering the conditions, IC obtained a worst case delay of 138 ms, thus the system resulted as being

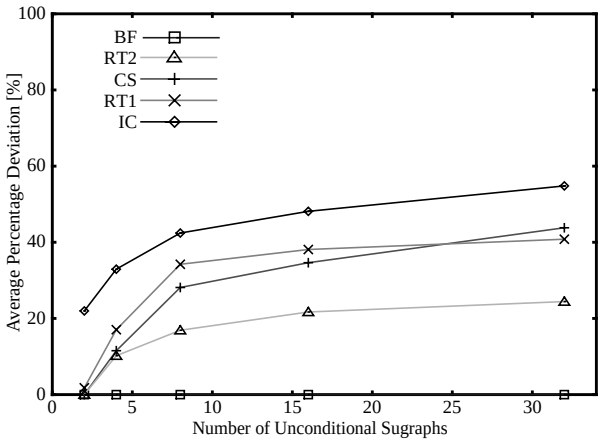


Figure 10. Average Percentage Deviations

unschedulable. The same result was obtained with the CS approach, and this is because the alternative paths were mapped on different processors, thus not influencing each other. However, the brute force approach BF produced a worst case delay of 104 ms which proves that the system implementing the vehicle cruise controller is, in fact, schedulable. Both RT1 and RT2 produced the same worst case delay of 104 ms as the BF.

7. Conclusions

In this paper we proposed solutions to the schedulability analysis of hard real-time systems with control and data dependencies.

The systems are modeled through a set of conditional process graphs that are able to capture both the flow of data and that of control. We consider distributed architectures, and the scheduling policy is based on a static priority preemptive strategy.

Five approaches to the schedulability analysis of such systems are proposed. Extensive experiments and a real-life example show that by considering the conditions during the analysis, the pessimism of the analysis can be drastically reduced.

While the brute force approach BF performed best at the expense of execution time, the RT2 approach is able to obtain results with less than 20% average loss in quality, in a very short time.

References

[1] N. C. Audsley, A. Burns, R. I. Davis, K. Tindell, A. J. Wellings, “Fixed Priority Pre-emptive Scheduling: An Historical Perspective”, *Real-Time Systems*, 8(2/3), 173-198, 1995.

[2] A. Doboli, P. Eles, “Scheduling under Control Dependencies for Heterogeneous Architectures”, International Conference on Computer Design, 602-608, 1998

[3] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, P. Pop, “Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems”, Proceedings of Design Automation & Test in Europe, 23-26, 1998.

[4] G. Fohler, “Realizing Changes of Operational Modes with Pre Run-time Scheduled Hard Real-Time Systems”, *Responsive Computer Systems*, H. Kopetz and Y. Kakuda, editors, 287-300, Springer Verlag, 1993.

[5] G. Fohler, “Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems”, Proceedings of the 16th IEEE Real-Time Systems Symposium, 1995.

[6] R. Gerber, D. Kang, S. Hong, M. Saksena, “End-to-End Design of Real-Time Systems”, *Formal Methods in Real-Time Computing*, D. Mandrioli and C. Heitmeyer, editors, John Wiley & Sons, 1996.

[7] H. Kopetz, *Real-Time Systems-Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997.

[8] H. Kopetz, G. Grünsteidl, “TTP-A Protocol for Fault-Tolerant Real-Time Systems”, *IEEE Computer*, 27(1), 14-23, 1994.

[9] C. L. Liu, J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”, *Journal of the ACM*, 20(1), 46-61, 1973.

[10] K. McMillan and D. Dill, “Algorithms for interface timing verification”, Proceedings of IEEE International Conference on Computer Design, 48-51, 1992.

[11] P. Pop, P., Eles, Z., Peng, “Schedulability-Driven Communication Synthesis for Time-Triggered Embedded Systems”, Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications, 1999.

[12] P. Pop, P., Eles, Z., Peng, “Scheduling with Optimized Communication for Time-Triggered Embedded Systems”, Proceedings of the International Workshop on Hardware-Software Co-design, 78-82, 1999.

[13] L. Sha, R. Rajkumar, J. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”, *IEEE Transactions on Computers*, 39(9), 1175-1185, 1990.

[14] K. Tindell, “Adding Time-Offsets to Schedulability Analysis”, Department of Computer Science, University of York, Report Number YCS-94-221, 1994.

[15] K. Tindell, J. Clark, “Holistic Schedulability Analysis for Distributed Hard Real-Time Systems”, *Microprocessing and Microprogramming*, 40, 117-134, 1994.

[16] T. Yen, W. Wolf, “Performance estimation for real-time distributed embedded systems”, *IEEE Transactions on Parallel and Distributed Systems*, Volume: 9(11), 1125 -1136, Nov. 1998