ARTES

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

Real-Time Graduate Student Conference 2001

Lund Institute of Technology

LUNDS TEKNISKA HÖGSKOLA
Lunds universitet

UPPSALA UNIVERSITY

Swedish Foundation for Strategic Research

UPPSALA UNIVERSITY

Swedish Foundation for Strategic Research

2001-03-04

ARTES TODAY

Dear participant at the 3:rd ARTES Graduate Student Conference
in Lund, Sweden, March 8-9, 2001,

Starting as an idea in 1996, ARTES has now developed into an internationally well known research network. Almost 100 graduate students, their advisors and industrial partners are included in the network, and more than 40 of the students are currently funded by ARTES.

Last year ARTES was evaluated by its funding organisation, the Swedish Foundation for Strategic Research (SSF). It was a great experience for us to gather all the material you supplied us with. An impressive amount of studies have already been carried out, providing many impressive results. The publications (printed double sided) that were sent to the evaluators completely filled a standard size paper box for copying machine paper. What is really amazing is that this is only the beginning, since in August last year when the evaluation took place the average ARTES student had only carried out an equivalent of one year of studies. What is even more satisfying is that the comments we received from SSF have been very positive, indicating that ARTES is the best programme in the IT field, as well as the importance of the network for creating a critical mass. This has given us courage to proceed with planning of the next stage, provisionally named ARTES++.

The aim is to start ARTES++ already 2002, i.e. before the end of ARTES in the beginning of 2003. The intention is to continue with the networking activities in ARTES++, but to make the research programme more focused by defining a number of sub-programmes (or clusters) with specific research agendas and technical visions. The research groups and industrial networks are currently being formed for these clusters.

Now, in March 2001 ARTES project support has reached its maximum, at a level of 1.6 MSEK per month, corresponding to one PhD carried out every 6 weeks. This corresponds to about 40% of the total Real-Time studies in Sweden. Adding the importance of the networking and mobility activities, it is definitely fair to say that ARTES is the driving force for the real-time research in Sweden. On the other hand, this research would never have been possible without the many talented and dedicated graduate students that performs most of the actual work.

Your contributions are highly appreciated! As is your help to cover all relevant conferences in the world with your presence and travel reports.

This Graduate Student Conference is the largest so far with 42 pre-registerd participants.

Welcome

Hans Hansson and Roland Grönroos

Participants at
ARTES Graduate Student Conference 2001

Number of participants 42, presentations 21.

Name	E-mail	Affiliation
Lars Albertsson	lalle@sics.se	SICS
Björn Andersson	ba@ce.chalmers.se	Chalmers
Bengt Asker	bengt.asker@telia.com	ARTES
Carl Bergenhem	carl.bergenhem@ide.hh.se	Halmstad
Anton Cervin	anton@control.lth.se	Lund
Fredrik Dahlgren	fredrik.dahlgren@ecs.ericsson.se	Ericsson Mobile
Radu Dobrin	radu.dobrin@mdh.se	MDH
Cecilia Ekelin	cekelin@ce.chalmers.se	Chalmers
Magnus Ekman	mekman@ce.chalmers.se	Chalmers
Torbjörn Ekman	torbjorn@cs.lth.se	Lund
Jad El-khoury	jad@md.kth.se	KTH
Petter Ericson		Anoto AB
Sven Gestegård Robertz	Sven.Gestegard@cs.lth.se	Lund
Flavius Gruian	Flavius.Gruian@cs.lth.se	Lund
Roland Grönroos	Roland.Gronroos@DoCS.UU.SE	ARTES
Sanny Gustavsson	sanny@ida.his.se	Skövde
Hans Hansson	hansh@DoCS.UU.SE	ARTES
Dan Henriksson	dan@control.lth.se	Lund
Hoai Hoang	hoang@ide.hh.se	Halmstad
Daniel Häggander	Daniel.Haggander@ipd.hk-r.se	Blekinge
Magnus Jonsson	Magnus.Jonsson@ide.hh.se	Halmstad
Krzysztof Kuchcinski	Krzysztof.Kuchcinski@cs.lth.se	Lund
Tomas Lennvall	tomas.lennvall@mdh.se	MDH
Bo Lincoln	lincoln@control.lth.se	Lund
Birgitta Lindström	birgitta@ida.his.se	Skövde
Sorin Manolache	sorma@ida.liu.se	Linköping
Anders Nilsson	anders.nilsson@cs.lth.se	Lund
Klas Nilsson	klas@cs.lth.se	Lund
Robert Nilsson	robni@ida.his.se	Skövde
Thomas Nolte	thomas.nolte@mdh.se	MDH
Dag Nyström	dag.nystrom@mdh.se	MDH
Anders Orebäck	oreback@nada.kth.se	KTH
Anders Pettersson	anders.pettersson@mdh.se	MDH
Paul Pop	paupo@ida.liu.se	Linköping
Paul Scerri	pausc@ida.liu.se	Linköping
Per Håkan Sundell	phs@cs.chalmers.se	Chalmers
Radoslaw Szymanek	Radoslaw.Szymanek@cs.lth.se	Lund
Yudong Tan	ydtan@ece.gatech.edu	MDH/guest
Mattias Wecksten	Mattias.Wecksten@ratech.se	Halmstad
Zhang Yi	yzhang@cs.chalmers.se	Chalmers
Aleksandra Zagorac	g-aleza@ida.liu.se	Linköping
Karl-Erik Årzén	karlerik@control.lth.se	Lund

Programme: ARTES Real-Time Graduate Student Conference 2001

Thursday March 8

			Page
9.30-10.00	Registration and Coffee		
10.00-10.20	**Introduction**	Hans Hansson, Karl-Erik Årzén	
10.20-11.00	**Session: Computer Control**	**Chair**: Klas Nilsson	
	AIDA II Progress Report: Towards a Co-simulation Environment for Computer Control Systems	Jad El-Khoury, KTH	**9**
	Analyzing the Effects of Missed Deadlines in Control Systems	Anton Cervin, LTH	**17**
11.00-11.15	Break		
11.15-12.15	**Session: Testing & Debugging**	**Chair**: Bengt Asker	
	Simulation-Based Debugging and Profiling of Soft Real-Time Applications	Lars Albertsson, SICS	**27**
	Methods for Increasing Software Testability	Birgitta Lindström, HIS	**39**
	Test Case Generation for Testing of Timeliness	Robert Nilsson, HIS	**43**
12.30-13.30	**Lunch**, Kårhuset		
13.30-14.30	**Plenary**		
	CTechnologies & Anoto: An Embedded System Case Study	Petter Ericson Chief Scientific Officer, Anoto AB	
14.30-15.30	**Session: Scheduling**	**Chair**: Hans Hansson	
	Fixed-Priority Preemptive Multiprocessor Scheduling: To Partition or not to Partition	Björn Andersson, Chalmers	**55**
	Solving Embedded System Scheduling Problems using Constraint Programming	Cecilia Ekelin, Chalmers	**65**
	Memory and Time-Efficient Schedulability Analysis of Task Sets with Stochastic Execution Time	Sorin Manolache, LiU	**81**
15.30-16.00	Coffee		
16.00-17.00	**Session**	**Chair**: Karl-Erik Årzén	
	On Energy Reduction in Hard Real-Time Systems with Dynamic Voltage Supply Processors	Flavius Grubian, LTH	**89**
	Minimizing System Modification in an Incremental Design Approach	Paul Pop, LiU	**95**
	Minimization of Execution Scenarios in Static Priority Preemptive Scheduled Real-Time Systems	Anders Pettersson, MDH	**103**
19.00-...	**Dinner**, Restaurant Stäket		

Friday March 9

			Page
8.45-9.45	**Plenary**		
	Future Mobile Phones - Design Challenges from a Real-Time System's Perspective	Fredrik Dahlgren, Technical Manager Ericsson Mobile Communications	
9.45-10.00	ARTES Patent Award Ceremony		
10.00-10.30	Coffee		
10.30-11.30	**Session**	**Chair**: Karl-Erik Årzén	
	Deterministic Java in Tiny Embedded Systems	Anders Nilsson, LTH	**105**
	Designing Agents for Systems with Adjustable Autonomy	Paul Scerri, LiU	**115**
	On recovery and consistency preservation in distributed real-time database systems	Sanny Gustavsson, HIS	**123**
11.30-11.40	Break		
11.40-12.20	**Session: Communication**	**Chair**: Magnus Jonsson	
	Switched Real-Time Communication for Industrial Applications	Hoai Hoang, HH	**129**
	Fibre-Ribbon Pipeline Ring Network with Distributed Global Deadline Scheduling	Carl Bergenhem, HH	**131**
12.30-13.30	**Lunch**, Kårhuset		
13.30-14.30	**Session: Real-Time Programming**	**Chair**: Klas Nilsson	
	Modeling and Analysis of Message-Queues in Multi-Tasking Systems	Thomas Nolte, MDH	**141**
	Applications of lock and wait-free shared data structures to real-time systems	Håkan Sundell, Chalmers	**149**
	Non-blocking synchronization for soft realtime applications	Zhang Yi, Chalmers	**151**

Presentations at ARTES Graduate Student Conference 2001

Name	Title and Authors
Lars Albertsson	Simulation-Based Debugging and Profiling of Soft Real-Time Applications Lars Albertsson
Björn Andersson	Fixed-Priority Preemptive Multiprocessor Scheduling: To Partition or not to Partition Björn Andersson and Jan Jonsson
Carl Bergenhem	Fibre-Ribbon Pipeline Ring Network with Distributed Global Deadline Scheduling Carl Bergenhem, Magnus Jonsson, and Jörgen Olsson
Anton Cervin	Analyzing the Effects of Missed Deadlines in Control Systems Anton Cervin
Fredrik Dahlgren	Future Mobile Phones - Design Challenges from a Real-Time System's Perspective Fredrik Dahlgren
Cecilia Ekelin	Solving Embedded System Scheduling Problems using Constraint Programming Cecilia Ekelin and Jan Jonsson
Jad El-khoury	AIDA II Progress Report Towards a Co-simulation Environment for Computer Control Systems Jad El-khoury
Petter Ericson	CTechnologies & Anoto: An Embedded System Case Study Petter Ericson
Flavius Gruian	On Energy Reduction in Hard Real-Time Systems with Dynamic Voltage Supply Processors Flavius Gruian
Sanny Gustavsson	On recovery and consistency preservation in distributed real-time database systems Sanny Gustavsson, Sten F. Andler
Hoai Hoang	Switched Real-Time Communication for Industrial Applications Hoai Hoang
Birgitta Lindström	Methods for Increasing Software Testability Birgitta Lindström, Jonas Mellin, and Sten Andler
Sorin Manolache	Memory and Time-Efficient Schedulability Analysis of Task Sets with Stochastic Execution Time Sorin Manolache, Petru Eles, Zebo Peng.
Anders Nilsson	Deterministic Java in Tiny Embedded Systems Anders Nilsson and Torbjörn Ekman
Robert Nilsson	Test Case Generation for Testing of Timeliness Robert Nilsson, Sten F. Andler and Jonas Mellin
Thomas Nolte	Modeling and Analysis of Message-Queues in Multi-Tasking Systems Thomas Nolte
Anders Pettersson	Minimization of Execution Scenarios in Static Priority Preemptive Scheduled Real-Time systems Anders Pettersson
Paul Pop	Minimizing System Modification in an Incremental Design Approach Paul Pop, Petru Eles, Traian Pop, Zebo Peng

Paul Scerri	Designing Agents for Systems with Adjustable Autonomy Paul Scerri and Nancy Reed
Per Håkan Sundell	Applications of lock and wait-free shared data structures to real-time systems Håkan Sundell
Zhang Yi	Non-blocking synchronization for soft realtime applications. Zhang Yi

---

Invitation, Programme, Participants, Presentations

Updated: 03-Mar-2001 09:39 by Roland Grönroos
e-mail: artes@docs.uu.se web: www.artes.uu.se
Location: http://www.artes.uu.se/events/gsconf01/pres.shtml

AIDA II Progress Report

Towards a Co-simulation Environment for Computer Control Systems

Jad El-khoury

KTH, Department of Machine design,
Mechatronics lab, 100 44 Stockholm
jad@md.kth.se

ABSTRACT

Modern machinery, such as automobiles, trains and aircraft, are equipped with embedded distributed computer control systems where the software implemented functionality is steadily increasing. The development of such systems, however, is still a relatively new and young discipline. There is consequently a lack of tools, methods and models to support, in particular, the early architectural design stages.

Closely associated with the design of a distributed control system are design decisions on the overall system structure, function triggering and synchronization, policies for scheduling, communication and error handling. These parameters to a large extent determine the resulting system timing and dependability, and hence, impact the control application performance and robustness.

The primary target for the research described here is the development of executable models to support architectural design. The report describes earlier efforts in this direction including case studies in the DICOSMOS and AIDA projects. Experiences from the studies are discussed including aspects on modelling to support interdisciplinary design, modeling abstraction and accuracy, and tool implementation.

A state of the art survey of related modelling efforts reveals that very little appears to have been done in terms of modelling and simulation that involves simulation of the environment with the computer control system. In addition, the treatment of distributed computer systems, redundant systems, and error scenarios in this context appear to be scarce.

The possibilities for extending the simulation models to arrive at a ready to use library to support the design of distributed control systems are discussed. This work is already under way. Remaining work includes to verify the developed simulation models, and to evaluate the models and their usage in case studies. One related topic for further work includes integrating the simulation models with the AIDA modeling framework. A longer term aim is that the models and simulation features should form part of a larger toolset supporting also earlier and later development stages, thus providing more comprehensive analysis and synthesis support.

1. INTRODUCTION

This report is a summary of an internal report developed at the mechatronics lab in order to identify the short and long-term research work to be done in the AIDA II project. Please refer to (Törngren, 2001) for more detailed information.

1.1. Background: embedded control systems, trends and challenges

Machinery, such as automobiles, trains and aircraft, are increasingly being equipped with em-

bedded control systems that are based on distributed computer systems. In these systems the functionality is steadily increasing due to the possibilities enabled by software and networks for information exchange. The computer system is typically composed of a number of networks that connect the different nodes of the system. A node typically includes sensor and actuator interfaces, a microcontroller, and a communication interface to the broadcast bus. From a control function perspective, such machines can be said to be controlled by a hierarchical system, where subfunctions interact with each other and through the vehicle dynamics to provide the desired control performance.

However, a number of challenges face the developers of such machinery in order to really be able to benefit from the technology advances:

- *The need to cope with the dramatic increase in system (software) complexity*. Apart from the sheer amount of new functionality, complexity also arises from different types of interference between functions partly arising from their usage of shared computer system resources.
- *Managing and exploiting multidisciplinarity*. The development of embedded control systems requires knowledge in, and cooperation between, several different scientific and engineering disciplines.
- *Verification and validation of dependability requirements*. The main challenge here is that of developing mechatronic (software based) systems that are safer and more reliable than their mechanical counterparts. Albeit the advantages, software easily becomes very complex and is difficult to verify and validate. In addition, the distributed computer systems where the software is implemented add “new” failure modes.

1.2. Modelling and simulation in the context of architectural design

The development of embedded computer control systems is still a relatively new and young discipline. Consequently, there is a lack of tools, methods and models to support, in particular, the early so called architectural design stages. The primary target for the research described here is the development of models to support architectural design. In the context of designing a distributed computer control system, architectural design is here used to refer to decisions on the:

- Overall system structure, including both functional, software and computer structure (These issues include deciding on allocation and partitioning)
- Function triggering, synchronization, and policies for scheduling all resources
- Communication principles
- Policies for error handling, and the mechanisms used for error detection.

Choices of these “design parameters” to a large extent determine the resulting system reliability, safety, and timing, and impact the control application performance and robustness.

The very basic idea is to develop models that can be used to analyse different architectural proposals, simulation is the analysis approach taken in this work. In a longer term perspective, the aim is that the models and simulation features should form part of a larger toolset being developed at the mechatronics lab; a toolset that supports the design of embedded control systems in order the meet the main identified challenges: complexity, multidisciplinarity and dependability.

Some of the requirements on the models are as follows (the reader is referred to Törngren (1995) and Redell (1998) for more detail and background):

- The models should allow both functionality, to be implemented in a computer system, and the computer systems to be represented.

- The models should allow co-simulation of functionality, as implemented in a computer system, together with the controlled continuous time processes and the behavior of the computer system.
- The models should support interdisciplinary design, thus taking into account different supporting methods, modelling views, abstractions and accuracy, as required by control, system and computer engineers.
- The models should be useful also for a descriptive framework, visualising different aspects of the system, as well as being useful for other types of analysis such as scheduling analysis.

1.3. Why model based architectural design?

By model based design we primarily refer to models that are sufficiently formal to allow analysis to be carried out. Early architectural analysis allows the solution space to be explored, and at the same time provides a better opportunity for the early detection of erroneous requirements and design bugs. Clearly, such mistakes are very costly if detected late. Compared to traditional testing, the approach allows different failure modes to be approached and analysed one at a time, progressing the verification through the development in an iterative and incremental fashion.

A strong advantage of model based design used in the context of early analysis, is the ability to evaluate alternative architectures with very few restrictions. Compared to traditional integration testing, or hardware in-the-loop simulation where the complete environment of a node is simulated in real-time, model based design and analysis provides additional advantages including the ability to easily and with low cost change the design, and the possibility to instrument and analyse the internals of the distributed computer system. This is not to say that model based design can replace the others, but the tasks of for example the system integrator can be made much easier given that some aspects of the system, such as its timing behavior, have been analysed thoroughly earlier. A strong point of the modelling activity is that it is a very useful process that can reveal a number of aspects, such as missing requirements and design bugs.

The investigation of computer system models, at different levels of abstraction with different accuracy/complexity, is an educative process that we hope can provide additional understanding to support interdisciplinary design. The model and tool prototypes are also very important as part of the research because they enable different forms of feedback.

2. ACCOMPLISHMENTS IN MODELLING AND SIMULATION AT THE MECHATRONICS LAB

2.1. Early work on modelling and simulation in the DICOSMOS project

An early investigation of “timing problems” was carried out in the DICOSMOS project, see Wittenmark et al. (1995), Törngren (1995), Nilsson (1996). Here the effects of time-varying feedback delays, sampling period jitter and data loss in feedback control systems were investigated. Inspired by earlier work along the lines of Ray (1988), cosimulation was one approach taken in DICOSMOS towards analysing the timing problems. Special Simulink (www.mathworks.com) blocks were developed to model time-varying delays, sampling instant jitter, vacant sampling and sample rejection (data-loss through over-writing of data), Törngren (1995). A feedback control system, as modelled in Simulink, could then be instrumented with these blocks to come one step towards modelling a distributed computer implementation. The approach turned out to be useful for illustrating the effects of the timing problems and for analysing them, e.g. for comparing the effects of sampling period jitter vs. a time varying delay, or for analysing the sensitivity of a particular control design.

2.2. The AIDA modelling framework

The basis for developing the modelling framework was to determine the information needed in the context of the early stages of design of distributed control systems. The models were targeted towards motion control applications implying the need to be able to model time- and event-triggered, multirate, control systems with different modes of operation. These control systems are to be implemented on distributed heterogeneous hardware with both serial and parallel communication links interconnecting the processing elements. There is a need for modelling the various system specific overheads, scheduling policies, error handling etc. The derived requirements and the AIDA modelling framework are thoroughly described by Redell (1998), Törngren and Redell (2000).

2.3. Modelling and simulation of the SMART satellite fault-tolerant computer system

During very early design stages of the development of the distributed computer control system of the SMART satellite, to be launched in 2002 by the European Space Agency, the Swedish Space Corporation needed to analyse and verify the use of the Controller Area Network (CAN) in the on-board satellite computer control system. In particular the error handling of CAN in conjunction with the chosen design for a fault tolerant network was of concern. This work was partly carried out by the authors of this report, and included assessing the design by means of modelling and simulation. The main aim of the simulation was to verify that the given rules for redundancy are not conflicting, and that the system can recover from a single permanent fault. For more information on the simulation models the reader is referred to Törngren and Fredriksson (1999).

2.4. Cosimulation within the DICOSMOS2 project

Sanfridson (1999 & 2000) describes a co-simulation of a truck and semi-trailer using a CAN bus for the on-board distributed control system in order to investigate the performance of control applications and adjust control periods and message priorities on-line. The simulations have been implemented in Matlab/Simulink. The model of the CAN bus is fairly detailed which gives a realistic timely behaviour of the network communication. The major drawback is the amount of processor time required to carry out the simulation at this detailed level.

3. THE STATE OF THE ART

Various simulation tools have been developed to tackle some of the issues discussed in this report. A small survey has been performed in order to evaluate these tools, with an earlier complementing survey given by Redell (1998). What was common between these tools is the fact that they are developed with the aim of simulating real-time computer systems. However, each tool is focused on particular aspects of such systems.

The following tools were evaluated: (See Törngren, 2001 for further details)

- RT/CS Co-design: A Matlab Toolbox for Real-time and Control Systems Co-design
- DRTSS: A Simulation Framework for Complex Real-time Systems
- STRESS - A Simulator for Hard Real-time Systems
- HaRTS: Design and Simulation of Hard Real-time Applications

4. SOME ESSENTIAL ISSUES IN MODELLING AND SIMULATION

4.1. Interdisciplinary design: modeling purpose, abstraction and accuracy

Good cooperation and interfaces between the involved engineers is essential to maintain con-

sistency between specifications, design and implementation. When developing a motion control system for a machine, somehow the functions and elements thereof need to be allocated to the nodes. This principally means that an implementation independent functional design needs to be enhanced with new “system” functions that:

- Perform communication between parts of the control system.
- Perform scheduling of the computer system processors and networks.
- Perform additional error detection and handling.

This mapping will change the timing behavior of the functions due to effects such as delays and jitter.

4.2. Issues related to system development and tool implementation

While developing distributed control systems, it would be advantageous to have a simulation toolbox or library, in which the user can build the system based on prebuilt modules to define things such as the network protocols and the scheduling algorithms. With such a tool, the user can focus on the application details instead. Such a tool is possible since components like different types of schedulers and CAN network are well defined and standardised across applications. Such a tool will enforce a boundary between the application and the rest of the system. This will speed up the development process, and gives the developers extra flexibility in developing the application.

Also, to be useful and cost-effective for system development, the usage of the models and the simulation facilities need to be integrated into the development process.

5. FUTURE WORK

Investigating and extending the AIDA modeling framework. Some further ideas for developing the models are as follows:

- The developed simulation models should be integrated with the AIDA modelling framework. Ideally, the same basic models should be useful for static analysis and for simulation.
- The simulation models do describe both system structure and behavior but they are not always very descriptive since they sometimes lack graphical views. Some proposals in this direction have been made in the AIDA models (Redell, 1998).
- The semantics of pure simulation models (such as the ones in Simulink) inherently differ from the timing behaviour of real-time execution, (Törngren et al., 1997). How can this semantic gap be bridged?
- The simulation models and the AIDA framework need to be extended both upwards, to models used in earlier design stages, and downwards, towards the implementation. Basically, the motivation is given by the need for a holistic design framework that enables models (and work accomplished) to be reused, and used efficiently. Thus it would be desirable to be able to refine the models such that they can be used not just in early design. In addition, work carried out in configuring a computer node for example, should be reusable in later prototyping and implementation stages.
- The use of the Unified Modelling Language (UML, 1997) and CODARTS models (Gomaa, 1993, a development of structured analysis models) in conjunction with the modeling framework will be investigated. Key aspects here include assessing when and why to use object models vs. functional (structured analysis) models, and how they map to other entities such as tasks.

Developing a toolset for architectural analysis. This topic builds upon the ideas of the AIDA project. The idea is to develop a toolset that provides comprehensive analysis and modelling support for embedded control systems.

The following are some ideas for the implementation of a prototype toolset for research purposes:

- The envisioned toolset concept is based on a number of well established engineering tools that are complemented and extended with a number of functionalities. These functionalities include additional models to enable important analysis and synthesis to be carried out. They will also include necessary interfaces between tools. There are two strong reasons for this structure; given limited research efforts, energy should not be spent on reinventing the wheel. In addition, using available tools will make it easier to demonstrate and apply the toolset concept in a realistic setting. It will also facilitate the pinpointing of the opportunities provided by the complementing tools. As an example, there are tools around that can deal with modelling and analysis of reliability, safety, control system design, finite state machines, timing analysis, etc., but these tools are not integrated; and even if they were, can not provide the functionality required for appropriately supporting the design of distributed real-time control systems.
- The functionalities considered include support for overall system structuring in early design stages (where reliability, safety, resource load and costs are evaluated), hazard and safety analysis integrated with control system design, and control design complemented with support for distributed real-time system implementation where timing analysis is one important part.
- Model reuse must be enabled by the envisioned toolset. Reuse can come in different forms. One aspect of this is to be able to use a developed model throughout the life cycle of a product in order to support product maintenance including upgrades. As discussed earlier, this requires models to be refined during the development. Having developed a simulatable model, it would be valuable to reuse this information, for example the computer system configuration, in further prototyping and development work.

Many interesting issues exist and will arise in the development of this type of toolset.This includes how to manage the different types of models and how to use the toolset facilities in system development. The toolset should be complemented by an appropriate design methodology. One important piece of this methodology should be a verification framework that promotes the development of dependable embedded systems.

Extending the functionality of the toolset also requires the models to be extended (as partly discussed in the previous section). For example, the AIDA models currently do not include explicit fault models and attributes such as criticality.

6. REFERENCES

Gomaa (1993). Hassan Gomaa. *Software design methods for concurrent and real-time systems*. Addison-Wesley publishing company, 1993.

Nilsson (1996). Johan Nilsson. *Real-Time Control Systems with Delays*. PhD thesis. ISRN LUTFD2/TFRT--1049--SE, Lund Institute of Technology, Sweden.

Ray and Halevi (1988). Asok Ray and Y. Halevi. Integrated Communication and Control Systems: Part II - Design Considerations. *ASME Journal of Dynamic Systems, Measurements and Control*, Vol 110, Dec. 1998, pp 374-381.

Redell (1998). Ola Redell. *Modelling of Distributed Real-Time Control Systems, An Approach for Design and Early Analysis*, Licentiate Thesis, Department of Machine Design, KTH, 1998, TRITA-MMK 1998:9, ISSN 1400-1179, ISRN KTH/MMK--98/9--SE, Stockholm, Sweden.

Sanfridson (1999). Martin Sanfridson. QoS in Distributed Control of Safety-Critical Motion Systems. Work in progress paper at the *20th Real-time Systems Symposium*, Phoenix, December 1999.

Sanfridson (2000). Martin Sanfridson. *Timing problems in distributed control.* Licentiate Thesis, TRITA-MMK 2000:14, ISSN 1400-1179, ISRN KTH/MMK--00/14--SE, May 2000.

Törngren (1995). Martin Törngren. *Modelling and design of distributed real-time control applications*. Doctoral thesis, Department of Machine Design, KTH, TRITA-MMK 1995:7, ISSN1400-1179, ISRN KTH/MMK--95/7--SE.

Törngren and Fredriksson (1999). Martin Törngren and Peter Fredriksson. *SMART-1. CAN and Redundancy logic simulation of the SMART SU*. Swedish Space Corporation, Report S80-1-SRAPP-1.

Törngren and Redell (2000). Martin Törngren and Ola Redell. A Modelling Framework to support the design and analysis of distributed real-time control systems. *Journal of Microprocessors and Microsystems* 24 (2000) 81-93, Elsevier, special issue based on selected papers from the Mechatronics 98 proceedings.

Törngren (2001). Martin Törngren, Jad El-khoury, Martin Sanfridson and Ola Redell. *Modelling and Simulation of Embedded Computer Control Systems: Problem formulation*. Internal Report, Department of Machine Design, KTH, TRITA-MMK 2001:3, ISSN1400-1179, ISRN KTH/MMK--01/3--SE.

Törngren et al. (1997). Martin Törngren, Christer Eriksson, Kristian Sandström (1997). Real-time issues in the design and implementation of multirate sampled data systems. In Preprints of *SNART 97 -Swedish National Association on Real-Time Systems Conference*, Lund, 21-22 August 1997.

UML (1997). UML notation guide. Version 1.1. Sept. 1997. Object Management Group, doc. no. ad/97-08-05. http://www.rational.com/uml

Wittenmark et al. (1995). Björn Wittenmark, Johan Nilsson, and Martin Törngren. Timing Problems in Real-time Control Systems. P*roceedings of the 1995 American Control Conference*, Seattle, WA, USA.

Analyzing of the Effects of Missed Deadlines in Control Systems

Anton Cervin

Department of Automatic Control
Lund Institute of Technology
Box 118, SE 221 00 Lund, Sweden
anton@control.lth.se

Abstract

Design based on worst-case assumptions and hard deadlines can give low resource utilization. In control systems, this corresponds to slow sampling and low control performance. By allowing temporary overloads and overruns, we can increase both the average CPU utilization and the control performance.

As a first step toward real-time control design that tolerates overruns, this paper gives two examples which explore the impact of missed deadlines on control performance. In the first example, it is illustrated that control performance and scheduling performance are totally different things. In the second example, it is shown that in some cases we can actually gain control performance by setting tight deadlines which are then sometimes missed. The key parameters that link the areas of scheduling analysis and control analysis are the sampling interval, the input-output latency, and the jitter.

1. Background

Ever since the seminal paper on real-time scheduling theory, [Liu and Layland, 1973], computer-controlled systems have been used as the primary example of *hard real-time systems*. In a hard real-time system, the computer responds to periodic or non-periodic *events* by executing *tasks* that must finish within *hard deadlines*—or else, the system will fail. This description has shaped much of the real-time systems research during the past thirty years. It is questionable, however, whether this description really fits the large majority of computer-controlled systems.

Typically, a controller is implemented as a task that should execute periodically in the computer. In each period, the controller should sample the process output, execute the control algorithm, and send the new control signal, to the process. The controller is often designed using sampled-data control theory, see e.g. [Åström and Wittenmark, 1997], which assumes that the samples are taken at equidistant points in time.

The performance and stability of the closed-loop system depend not only on the control algorithm, but also on the actual implementation and run-time behavior of the controller in the computer system. The computing hardware, the execution-time properties of the control algorithm, the real-time operating system, the scheduling algorithm, and the possible network delays all introduce various amounts of delay and jitter in the control system. In the end, from a control perspective, it is

the *actual sampling period* (including jitter) and the *actual input-output latency* (including jitter) that influence the performance and stability of the closed-loop system. For further discussion on the sources and effects of timing variations, see [Törngren, 1998].

2. Where Do Deadlines Come From?

In the hard real-time scheduling literature, it is often unclear where the timing constraints—including deadlines—actually come from [Ramamritham, 1996]. The paper [Liu and Layland, 1973] is an exception to this, however: One of the assumptions clearly states that "Deadlines consist of run-ability constraints only—i.e. each task must be completed before the next request occurs." This implies that $D = T$ for all tasks. The connection to control theory is then made by mentioning that "Any control loops closed within the computer must be designed to allow at least an extra unit sample delay." Later research has extended the schedulability analysis to handle the cases $D < T$ and $D > T$ as well, see e.g. [Tindell *et al.*, 1994] and [Stankovic *et al.*, 1998], but the origin of the deadlines and their relation to control theory are rarely mentioned.

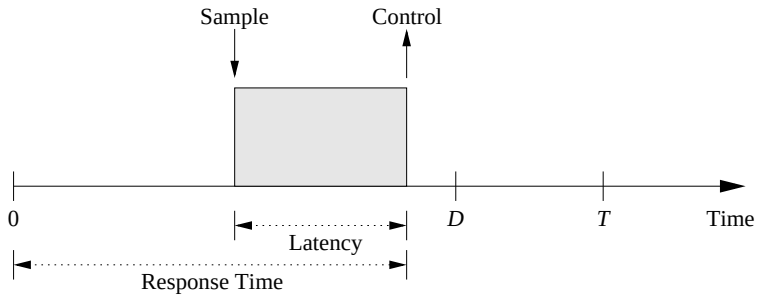


Figure 1 Illustration of the relationship between deadline (D), response time, and input-output latency for a control task. The latency is bounded by the response time, which in turn is bounded by the deadline. The task is assumed to be released at time zero.

From a control perspective, deadlines are primarily used to bound the input-output latency of the controllers. Figure 1 illustrates the relationship between deadline, response time, and input-output latency for a control task. Here, it is assumed that the A-D conversion takes place when the control task starts its execution, and that the D-A conversion takes place when the task completes its execution. A common alternative is to sample the process at the beginning of the period. Furthermore, it is common practice to divide the control algorithm into two parts—*Calculate Output* and *Update State*—and send out the control signal when the first part has completed.

For controlled plants with unstable dynamics, bounded latency is absolutely necessary to guarantee the stability of the closed-loop system. For all plants, bounded latency is necessary to guarantee some minimum level of control performance. Also, in general, the smaller the latency can be made, the better control performance and robustness can be achieved. Thus, typical deadlines assigned to control tasks ($D \leq T$) are often much tighter than what is dictated by pure stability considerations.

Deadlines may also be derived from other real-time constraints in the implementation. For instance, consider the case in Figure 2, where a dedicated, high-priority output task τ_{DA} is used to minimize the output jitter of control task τ_1, see [Locke,

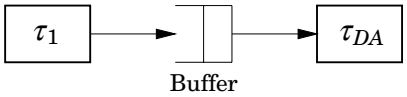


Figure 2 In fixed-priority scheduling with offsets, a high-priority, dedicated output task τ_{DA} can be used to minimize the output jitter of control task τ_1. This enforces a deadline on τ_1.

1992]. The tasks have the same period, but τ_{DA} is released with a fixed *offset* O relative to the release of τ_1. A deadline $D < O$ must be assigned to τ_1 to ensure that fresh output data will be available in the buffer when τ_{DA} starts its execution. See [Audsley *et al.*, 1993] for further details on offset scheduling. Different control structures, such as synchronized loops, cascaded loops, etc., may also enforce additional time constraints, see [Sandström, 1999].

Truly *hard* deadlines in real-time control systems are studied in [Shin *et al.*, 1985]. There, hard constraints on the controlled variables (e.g. physical constraints) are used to derive maximum allowable control latencies in different regions of the state-space. It is also noted that the hard deadline may be a random variable due to stochastic disturbances acting on the process. The approach is extended in [Shin and Kim, 1992], where the stability of the closed-loop system is also considered. In the examples given, the hard deadlines are typically found to be several times longer than the sampling interval. This may not be so surprising. As pointed out, the sampling period of a controller is not only chosen to satisfy Shannon's sampling theorem, but also to achieve the desired performance. Thus, a controller always has some degree of robustness against variations in the sampling interval due to, for instance, missed deadlines.

3. Analysis—A Two-Step Procedure

Investigating the effects of missed deadlines in control systems is a two-step procedure, see Figure 3. Since control performance is a function of the sampling period, the input-output delay, and the jitter, these, possibly stochastic, variables are first obtained by scheduling analysis. The resulting control performance is then found by control analysis. In the general case, both steps are very difficult to carry out, and some approximations may be necessary.

Previous work [Eker and Cervin, 1999; Palopoli *et al.*, 2000] has relied on co-simulations of the real-time kernel and the continuous dynamics to determine the performance of the controllers in the real-time system. While this approach can provide important insight, it is also very time-consuming, especially when the performance should be evaluated for a wide range of parameters. Also, simulations cannot be used to prove stability or a minimum level of performance.

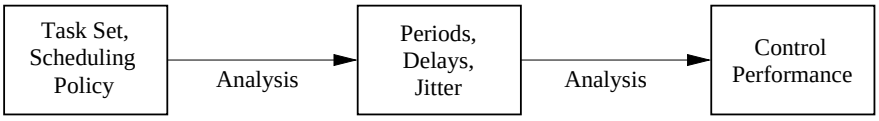


Figure 3 Analysis of the effects of missed deadlines on the control performance is a two-step procedure: scheduling analysis followed by control analysis.

3.1 Scheduling Analysis

The effect of missed deadlines depends not only on the scheduling policy used, but also on the details in the real-time operating system. Often, deadlines are derived and used in the scheduling design phase, but they are not necessarily enforced during run-time. Hence, an overrun in one task may or may not disrupt the timing of the other tasks.

Consider for instance a set of periodic control tasks that are described by the following pseudo-code:

```
t = CurrentTime;
LOOP
  AD-Conversion;
  ControlAlgorithm;
  DA-Conversion;
  t = t + T;
  WaitUntil(t);
END
```

Here, the only time constraint enforced during run-time is, that the task will never be released prior to its nominal release time. If the control task does not meet its design specifications during run-time, e.g. if the execution time of the control algorithm is larger than the predicted worst-case execution time, other task instances will be delayed.

The scheduling analysis methods available today do not allow for derivation of the distribution of the input-output delay and the jitter, except for in very simple cases. One solution could be to rely on simulations to find the approximate distributions and on analysis to find the best-case and the worst-case values.

3.2 Deterministic Control Analysis

If the pattern of sampling periods and delays is deterministic, the control analysis is straight-forward. A sampled plant, including a latency τ, where $0 < \tau \leq T$, can be written as (see e.g. [Åström and Wittenmark, 1997])

$$x(k+1) = \Phi x(k+1) + \Gamma_0 u(k) + \Gamma_1 u(k-1).$$

A controller can be written on general state-space form:

$$\begin{aligned} x_r(k+1) &= \Phi_r x_r(k) + \Gamma_r y(k) \\ u(k) &= C_r x_r(k) + D_r y(k) \end{aligned}$$

Forming the closed-loop system, we get

$$\begin{bmatrix} x(k+1) \\ u(k) \\ x_r(k+1) \end{bmatrix} = \underbrace{\begin{bmatrix} \Phi + \Gamma_0 D_r C & \Gamma_1 & \Gamma_0 C_r \\ D_r C & 0 & C_r \\ \Gamma_r C & 0 & \Phi_r \end{bmatrix}}_{\Phi_{cl}} \begin{bmatrix} x(k) \\ u(k-1) \\ x_r(k) \end{bmatrix}$$

For a constant Φ_{cl} matrix, the closed-loop system is stable if and only if $\bar{\sigma}(\Phi_{cl}) < 1$, i.e. iff the spectral radius of Φ_{cl} is less than one. When Φ_{cl} varies according a repeating pattern, $\Phi_{cl1}, \Phi_{cl2}, \ldots, \Phi_{cln}$, the closed-loop system is stable if and only if $\bar{\sigma}(\Phi_{cl1}\Phi_{cl2}\cdots\Phi_{cln}) < 1$.

3.3 Stochastic Control Analysis

When the actual sampling periods and delays are stochastic (as they are in the general case, due to random execution times, etc.), the analysis is considerably harder. One approach is to numerically evaluate a linear-quadratic cost function of the form

$$J = \lim_{T\to\infty} \frac{1}{T} \mathrm{E}\left\{ \int_0^T (x^T Q_1 x + u^T Q_2 u) dt \right\},$$

where x is the state vector, u is the control signal, and Q_1 and Q_2 are weighting matrices. The cost function measures the variance of different signals in the control system. With proper choices of weighting matrices, stability can be concluded if $J < \infty$. Furthermore, J is a measure of the performance of the controller.

Theory developed in [Nilsson, 1998; Lincoln and Bernhardsson, 2000] permits evaluation of the cost J of any linear controller for stochastic, independent delays from sampling to actuation. Aborted computations (modeled as infinite delays) can also be accounted for. Work in progress aims at evaluating the cost also in the presence of sampling jitter.

4. Two Examples

Two examples are given that demonstrate the type of analysis that can be carried out today. The first example assumes fixed execution times and investigates stability under overload conditions under different scheduling policies. The second example assumes a random execution time and demonstrates how a soft deadline can be assigned to maximize the control performance.

4.1 Example 1: Overloaded System

Consider a set-up where two mechanical servos,

$$G_{1,2}(s) = \frac{1000}{s(s+1)},$$

should be controlled by two PD (proportional-derivative) controllers that execute as two tasks in a computer. The controllers have been tuned to give good performance in simulations, and suitable sampling periods have been found to be $T_1 = 6$ ms and $T_2 = 4$ ms. It is assumed that $D = T$ for both tasks. Unfortunately, the execution time of the control algorithm is longer than anticipated, $C = 3$ ms. The requested CPU utilization is

$$U = \frac{C}{T_1} + \frac{C}{T_2} = 1.25.$$

Since $U > 1$, the system is overloaded, and no scheduling algorithm can guarantee that all deadlines will be met. This says very little about the performance of the controllers, however.

We analyze stability of the controllers under rate-monotonic (RM) and earliest-deadline-first (EDF) scheduling. Assuming that the tasks are implemented according to the pseudo-code in the previous section, we first derive the actual sampling periods and the actual input-output latency of the controllers. We then apply stability analysis to the controllers.

Rate-Monotonic Scheduling The performance under rate-monotonic (RM) scheduling is investigated first. Task 2 is given priority over Task 1 since $T_2 < T_1$. The first part of the resulting schedule, when both tasks are released at time zero, is shown in Figure 4. Because of preemption, Task 1 misses all its deadlines. The

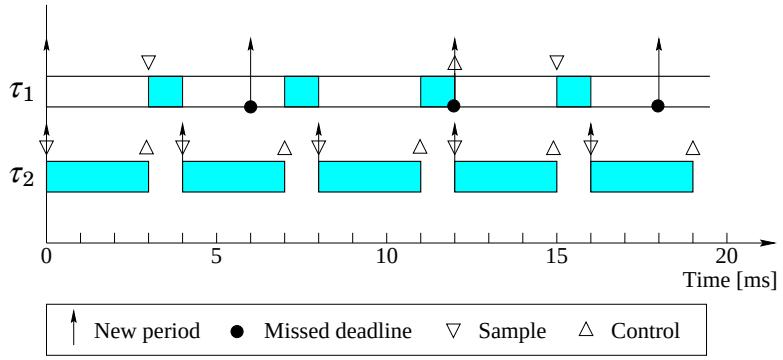


Figure 4 The first part of the schedule under RM scheduling. Task 1 misses all its deadlines.

regularity of the schedule makes it easy to determine the actual sampling period, $\overline{T}$, and the actual input-output latency, $\overline{L}$, of the controllers. By inspection, they are found to be constant (jitter-free) and equal to $\overline{T}_1 = 12$ ms, $\overline{L}_1 = 9$ ms, $\overline{T}_2 = 4$ ms, and $\overline{L}_2 = 3$ ms.

Applying the deterministic stability analysis to Controller 1, we find that $\bar{\sigma}(\Phi_{cl}) > 1$, so the controller is unstable. For Controller 2, we have $\bar{\sigma}(\Phi_{cl}) < 1$, so the controller is stable.

A simulation of the complete set-up is shown in Figure 5. The performance of Controller 2 is good, as expected. Controller 1, however, loses stability due to the long sampling period and the long latency, which the controller has not been designed for.

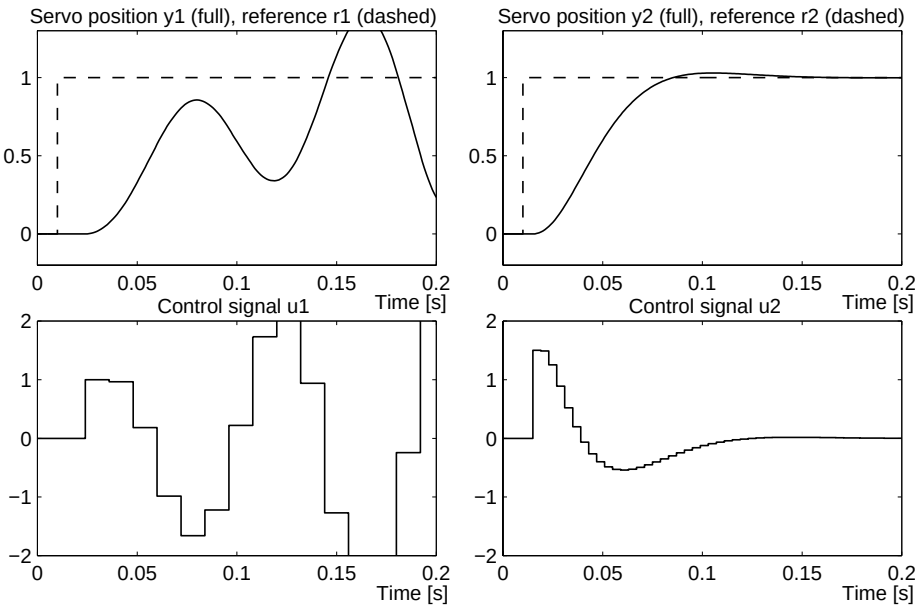


Figure 5 Step responses under RM scheduling for Controller 1 (left) and Controller 2 (right). Controller 1 loses stability due to the long actual sampling period and the long actual input-output latency.

Earliest-Deadline-First Scheduling Next, the performance under earliest-deadline-first (EDF) scheduling is investigated. The tasks execute in the order of their absolute deadlines. Ties can be broken arbitrarily—assume that Task 1 gets to execute before Task 2 in those cases. The first part of the resulting schedule, when both tasks are released at time zero, is shown in Figure 6. After an initial

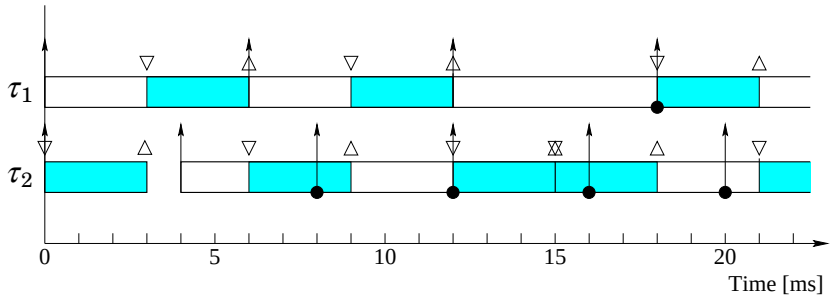


Figure 6 The first part of the schedule under EDF scheduling. After an initial transient, all deadlines are missed.

transient, *all* deadlines are missed. This is due to the well-known *domino effect*. There is no preemption in the resulting schedule so the actual input-output latency is always equal to $\overline{L} = 3$ ms for both controllers. The actual sampling periods are no longer constant, but they exhibit periodic behavior. By inspection, $\overline{T}_1$ is found to exhibit the cycle $\{6, 9\}$ ms while $\overline{T}_2$ exhibits the cycle $\{6, 6, 3\}$ ms.

In this case, we are switching between different sampling intervals, which gives different Φ_{cl} matrices. Applying the stability analysis, we find that for Controller 1, $\bar{\sigma}(\Phi_{cl1}\Phi_{cl2}) < 1$, so this controller is stable. For Controller 2, we have $\bar{\sigma}(\Phi_{cl1}\Phi_{cl1}\Phi_{cl2}) < 1$. so this controller is also stable.

A simulation of the set-up is shown in Figure 7. The systems are stable and the performance is satisfactory for both controllers, despite the missed deadlines. The jitter due to the varying sampling periods is clearly visible in the control signals.

The example shows that control performance and scheduling performance are completely different things. It also displays that EDF scheduling distributes the available computing resources more evenly in overload situations than RM scheduling does.

4.2 Example 2: Optimal Deadline Assignment

Consider a set-up where an inverted pendulum,

$$G(s) = \frac{1}{s^2 - 1},$$

should be controlled by a linear-quadratic controller. The sampling period is chosen as $T = 0.2$ s. The execution-time of the control task is described by the probability distribution function shown in Figure 8. The long tail of the distribution could for instance be the result of hardware interrupts, data dependencies, or cache misses.

An output buffer (a high-priority task) is used to minimize output jitter, see Figure 2. The output task executes with a fixed offset L compared to the controller. This enforces the deadline $D = L$ on the control task. If the output value is put in the buffer after the deadline has expired, the control signal will be lost.

The controller is designed taking into account a fixed delay L and minimizing the cost function

$$J = \lim_{T \to \infty} \frac{1}{T} \mathrm{E} \left\{ \int_0^T (y^2(t) + u^2(t)) dt \right\}. \tag{1}$$

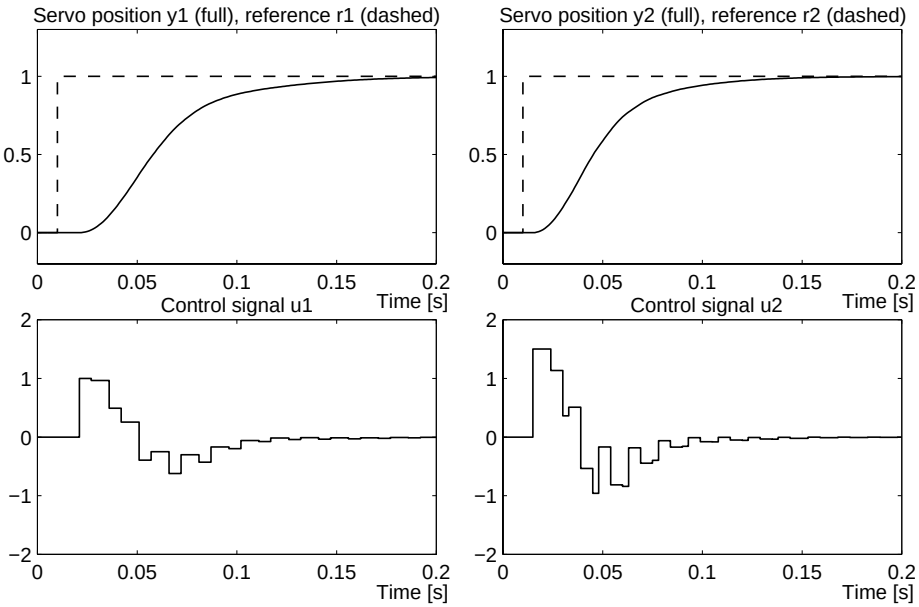


Figure 7 Step responses under EDF scheduling for Controller 1 (left) and Controller 2 (right). Both systems are stable despite the fact that, after an initial transient, all deadlines are missed. The sampling interval jitter is clearly visible in the control signals.

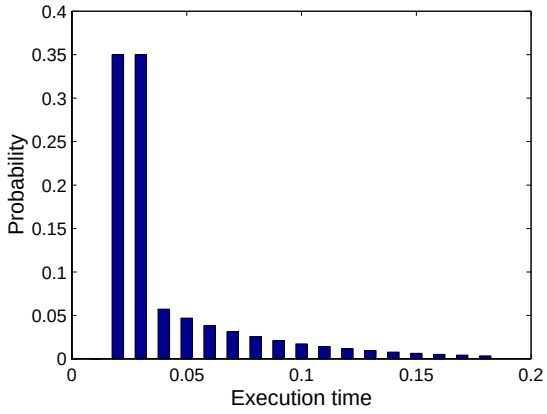


Figure 8 The probability distribution of the execution time of the pendulum controller.

The problem is to find the optimal value of L. If we choose $L = T$, all computations will finish in time, but we will also have a long delay, which is bad from a control performance perspective. If we choose a value $L < T$, the delay may be shorter on average, but some computations will also be aborted. If we choose $L = 0$, the task will never finish before its deadline and, without any control signals, the system will of course be unstable.

The optimal deadline assignment can be found by computing the cost J for different values of the deadline L. The plot is shown in Figure 9. In this case the optimal deadline assignment turns out to be $L = 0.08$, in which case about 10% of the deadlines are missed. The result depends strongly on the execution-time distribution, however. The penalty for aborted computations is quite high, so a more uniform execution-time distribution would push the deadline towards $L = T$. Still, the example shows that, even in the case of aborted computations, we can gain control performance by allowing some percentage of the deadlines to be missed.

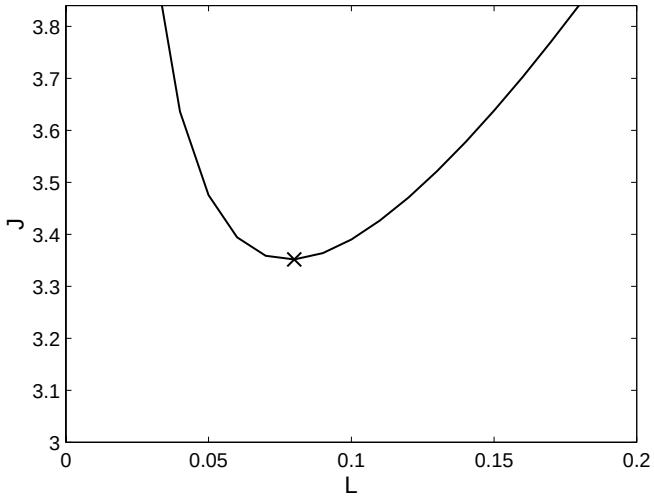


Figure 9 The cost J as a function of the deadline L for the inverted pendulum controller. The optimal deadline assignment is $L = 0.08$.

5. Conclusions

Relaxing the nominal requirements on hard deadlines in real-time control systems is motivated from a resource utilization viewpoint. Modern computing hardware tends to be optimized for high average-case performance rather than guaranteed worst-case performance. Also, many control algorithms display considerable variations in their execution-time demands. Taken together, scheduling based on worst-case execution times and hard deadlines may be infeasible for a large set of control applications.

Exploring the consequences of overruns is necessary in order to make the correct co-design trade-offs. For instance, it may be beneficial to decrease the average period of a controller, and allow it to miss a few deadlines, as long as the worst-case latency does not exceed a certain bound.

In the suggested analysis, the first step is to determine what the actual sampling period and the actual input-output latency will be for the different control tasks. These quantities will be random variables in the general case, depending on the task execution-time distributions, the scheduling algorithm, the mechanisms in the real-time operating system, the controller structure, etc. The next step is to determine what the impact on the control performance will be. Future work will extend the stochastic control performance analysis to handle sampling jitter.

6. References

Åström, K. J. and B. Wittenmark (1997): *Computer-Controlled Systems*, third edition. Prentice Hall.

Audsley, N., K. Tindell, and A. Burns (1993): "The end of the line for static cyclic scheduling." In *Proceedings of 5th Euromicro Workshop on Real-Time Systems*.

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.

Lincoln, B. and B. Bernhardsson (2000): "Optimal control over networks with long random delays." In *Proceedings of the International Symposium on Mathematical Theory of Networks and Systems*.

Liu, C. L. and J. W. Layland (1973): "Scheduling algorithms for multiprogramming in a hard-real-time environment." *Journal of the ACM*, **20:1**, pp. 40–61.

Locke, C. D. (1992): "Software architecture for hard real-time applications: Cyclic vs. fixed priority executives." *Real-Time Systems*, **4**, pp. 37–53.

Nilsson, J. (1998): *Real-Time Control Systems with Delays*. PhD thesis ISRN LUTFD2/TFRT--1049--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Palopoli, L., L. Abeni, and G. Buttazzo (2000): "Real-time control system analysis: An integrated approach." In *Proceedings of the IEEE Real-Time Systems Symposium, Orlando, Florida*.

Ramamritham, K. (1996): "Where do time constraints come from and where do they go?" *International Journal of Database Management*, **7:2**.

Sandström, K. (1999): *Modeling and Scheduling of Control Systems*. Licentiate thesis ISRN KTH/MMK/R--99/5--SE, Mechatronics Laboratory, Department of Machine Design, Royal Institute of Technology, Stockholm, Sweden.

Shin, K. G. and H. Kim (1992): "Derivation and application of hard deadlines for real-time control systems." *IEEE Transactions on Systems, Man, and Cybernetics*, **22:6**, pp. 1403–1413.

Shin, K. G., C. M. Krishna, and Y.-H. Lee (1985): "A unified method for evauating real-time computer controllers and its applications." *IEEE Transactions on Automatic Control*, **30:4**, pp. 357–366.

Stankovic, J., M. Spuri, K. Ramamritham, and G. Buttazzo (1998): *Deadline Scheduling for Real-Time Systems*. Kluwer Academic Publishers.

Tindell, K., A. Burns, and A. Wellings (1994): "An extendible approach for analyzing fixed priority hard real-time tasks." *Real-Time Systems*, **6:2**, pp. 133–151.

Törngren, M. (1998): "Fundamentals of implementing real-time control applications in distributed computer systems." *Real-time systems*, **14:3**.

Simulation-Based Debugging and Profiling of Soft Real-Time Applications

Lars Albertsson
Computer and Network Architectures Laboratory
Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden
lalle@sics.se

Abstract

We present a temporal debugger, capable of examining time flow of soft real-time applications. The debugger is attached to a simulator modelling an entire workstation in sufficient detail to run commodity operating systems and workloads. Whereas traditional debuggers need to interfere with program execution, thereby changing the temporal behaviour, a debugger operating on a simulated system does not disturb the timing of the target program. The temporal debugger therefore allows non-intrusive and reproducible debugging of real-time applications in general-purpose operating systems.

We have implemented the temporal debugger by modifying the GNU debugger to operate on applications running in a simulated machine. Debugger implementation is difficult because the debugger expects application-related data, whereas the simulator provides low-level data. We introduce a technique, virtual machine translation, for mapping simulator data to the debugger by parsing operating system data structures in the simulated system.

The debugger environment allows execution time measurement and collection of statistics from multiple levels of the system: hardware, operating system, and application. We show how this profiling data can help a programmer explain missed deadlines; the statistics are correlated to deadline violations, thereby exposing differences between executions when deadlines are met and when they are missed.

The temporal debugger is demonstrated with a debugging session of an MPEG video decoder running in a Linux system. We show how the debugger can be used to detect that the decoder fails to render some frames in time, and how the causes for deadline violations are found by correlating rendering time with runtime statistics.

Keywords: Complete System Simulation, GDB, Linux, Operating Systems, Real-Time Profiling, Simics, Soft Real Time Systems, Temporal Debugging

1 Introduction

Applications demanding short response times and throughput guarantees are increasingly common in general purpose computing environments. These so called soft real-time applications include video decoders, games, and online trading systems. They need to provide both high throughput performance and service requests in predictable time. Unfortunately, it is difficult to program applications to meet these requirements. The difficulty of the task is compounded by the fact that there are few tools available to aid the programmer.

Conventional tools in mainstream systems have no support for analysing real-time requirements and generally ignore the fact that application correctness depends on execution time. Tools specifically tar-

geting soft real-time applications are needed in order to help the programmer minimise the number of missed deadlines occurring during execution. Such tools should locate missed deadlines, explain why the program did not meet performance expectations and provide guidelines for correcting the problem. As the perceived quality of soft real-time applications depends on temporal correctness, they are sensitive to time variations during execution. The tools must therefore keep time distortion low and should be able to display time flow in detail.

The debugger is one of the most important tools for computer programming. Traditional debuggers, however, are inadequate for real-time applications, as they lack a notion of temporal correctness and interfere with program execution. In embedded real-time system design, these problems have been addressed by simulator-based debuggers [2, 22]. Historically, embedded systems have been so much slower than engineering workstations that temporally accurate simulation has been feasible. Until recently, it has not been practical to use simulators to study commodity desktop and server computer systems, as the simulators have been too slow.

Advances in simulation technology have resulted in simulators providing an approximate, but reasonably accurate timing model while executing roughly 50-200 times slower than native execution [8, 11]. These simulators, referred to as complete system simulators, model an entire workstation at the instruction set level, and run unmodified operating systems and workloads. A complete system simulator that is deterministic addresses the two major problems in real-time analysis: lack of reproducibility and time distortion resulting from intrusion. The characteristics of complete system simulators make them excellent candidates for building a temporally correct debugger for real-time systems. Furthermore, as the simulators execute both application and operating system, they provide an authentic model of performance and timing factors caused by the operating system.

In this paper, we present a temporal debugger for soft real-time applications. The debugger is based on the GNU debugger [6], modified to debug Linux applications running on a complete system simulator. As the target program and operating system are executed in an artificial environment, they can be suspended and probed without affecting the execution. Thus, the debugger allows non-intrusive and reproducible application debugging. The debugger is also able to present the time flow of the simulated system using the simulator's time model. Figure 1 shows the debugger user interface. The window in the background is the simulated console, showing output from UltraSPARC Linux during boot.

Figure 1: Temporal debugger user interface.

We have earlier studied temporal debugging of operating systems [1]. An operating system executes directly on the physical machine, which corresponds to the model provided by the simulator. It is therefore straightforward to debug operating systems by using services provided by the simulator. Debugging user level applications, however, is non-trivial. Unix programs execute inside protected environments, provided by the operating system. These environments are referred to as *virtual machines*. As a debugger expects information related to a process, it is necessary to map physical data, as presented by the simulator,

to process-related data. The information necessary to perform this mapping in a robust manner exists only in the operating system running in the simulated machine. We have therefore invented a technique, *virtual machine translation* (VMT), for reading memory and register contents of Linux processes by parsing kernel data structures.

We also describe how the services of the temporal debugger can be used to profile real-time applications. The debugger facilitates the explanation of deadline misses by providing measurements of system metrics, for example counts of hardware and operating system events, application variables, and execution time of functions. This allows a programmer to compare measurements and find differences in execution between successful iterations and iterations where a deadline was missed.

Section 2 provides some background on complete system simulation and the benefits of using it for real-time analysis. A description of the design and implementation of the temporal debugger, including the virtual machine translation, is presented in Section 3. It also contains an example debugging session of an MPEG video decoder running in a Linux system. In Section 4, we describe how the debugger allows measurement of system metrics and how the results are used for profiling the video decoder. Section 5 contains a short survey of related work in real-time debugging and simulation. Conclusions and future work are presented in Section 6.

2 Complete system 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 an execution environment including operating system and libraries. The simulator used in this paper is a complete system simulator. It provides a model of a machine at the instruction set level, which represents the well-defined border between hardware and software. It models all the hardware in a system, and only the hardware. As the simulation model is functionally identical to a real system, the simulator runs unmodified operating system and application software. This limits the sources of errors to those introduced by the hardware model, and by models of simulation input feed.

2.1 Simulation model

The model provided by a simulator can be thought of as having two components, functional and temporal. The simulator must provide an almost exact functional model to be able to run unmodified software. In contrast, the accuracy of the temporal model can be compromised without breaking the operating system and applications. It is therefore possible to trade accuracy for speed by using approximative models. The appropriate degree of approximation depends on the size of the workload and the time scale of its deadlines. For large workloads, a reasonably accurate time model is usually sufficient to obtain a coarse understanding of the timing behaviour in the system.

2.2 Reproducibility

A simulated computer is purely artificial and deterministic. Thus, a simulated system starting execution in a known state will always execute along the same path. This property is essential, both for experiments and debugging, as it is possible to reproduce a state reached in execution. A user of a temporal debugger may detect that excessive time has passed at one point in execution, and restart the simulation to examine recently executed routines more carefully. This technique is similar to the methodology used for debugging logical correctness of conventional programs. As time is part of the state the user wishes to verify, it is crucial that temporal behaviour is preserved between debugging sessions.

Execution is reproducible only if all input to the simulator is deterministic. As the physical world is inherently unpredictable, the simulator must not communicate with real, unpredictable input sources. Instead, all input sources must be modelled, with traces, synthetic models, or other simulators. In order to obtain a realistic input feed, the simulator may be connected to the real world once, with recording enabled. Further experiments can then use the

recorded trace.

2.3 Probe immunity

In physical systems, measurement of the system generally affects its behaviour. This is referred to as *probe effect*. Although measurements change time flow in the system, soft real-time applications tend to have deadlines separated by long intervals, and are therefore not very sensitive to these changes. The probe effect does, however, put a limit on the amount of measurement. Nevertheless, a temporal debugger requires non-intrusive probing to achieve reproducible execution. Even a minimal change in execution time flow could affect decisions in the applications and the operating system, resulting in a completely different execution path.

In a simulated system, the time scale of the system under study is decoupled from the time scale of the system running the simulator. When the user stops execution, simulation is suspended, and simulated time is frozen. Time distortion due to the probe effect is thereby eliminated.

2.4 Simics

The simulator used for this work is Virtutech Simics [21]. Simics simulates the SPARC V9 instruction set and models single or multiprocessor systems corresponding to the sun4u architecture from Sun Microsystems, for example the Enterprise 3500.

Simics consists of a core interpreter that offers basic services such as an instruction set interpreter, a general event model, and a module for simulating and profiling memory activity. A programming interface allows the addition of device models, which may be connected to the "real world" or models thereof.

Simics supports a simple time model in its default configuration. This model approximates time by defining a cycle as either an executed instruction or a part of a memory or device stall. In this mode, Simics has a rather simple view of the timing of a modern system, and assumes a linear penalty for events such as translation look-aside buffer (TLB) misses, data cache misses, and instruction cache misses. Simics supports efficient programming of models for the components most important for performance modelling: cache hierarchies, synchronisation in multiprocessor machines, and I/O devices.

Simics's programming interface allows the user to add more detailed timing models. If an application is constrained by a particular bottleneck in the system, the user can program a detailed model of that part and trade some simulation performance for a better model. The performance impact can be significant if the user adds an inefficient timing model for a central component, such as the CPU pipeline. In this case, performance can be improved by switching models at runtime. Examples of simulation model tradeoffs and dynamic switching of models have been presented by Herrod [8].

3 Debugging real-time applications

A debugger allows a programmer to inspect program state. In order for the debugger to be useful, it must not affect correctness by changing program behaviour. Also, as debugging is a repetitive task, the user must be able to repeat sessions, and observe identical execution each time. For programs whose correctness depend only on predictable input, meeting these requirements is straightforward. A debugger for real-time programs, however, must be able to capture and replay program time flow without changing it. In this section we describe how a debugger based on complete system simulation allows correct debugging of real-time applications.

3.1 Simulation-based debugging

A complete debugger setup consists of the debugger program and a target machine running the debugged program. Figure 2 shows the different parts of a debugger setup. We refer to the debugger program itself as front-end and to the target machine/program tuple as back-end. Examples of such tuples are: Unix programs running in the virtual machine provided by the operating system, or an embedded operating system on a separate target board.

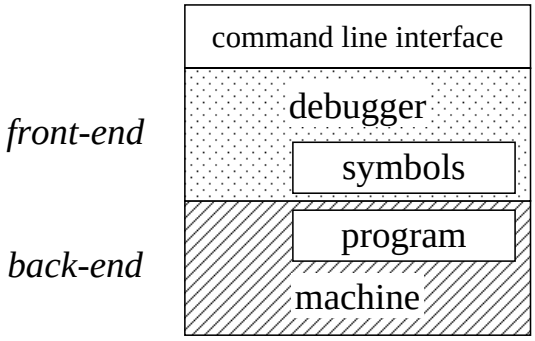


Figure 2: Debugger structure.

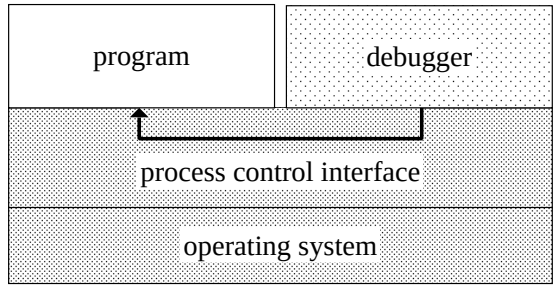


Figure 3: Traditional debugger.

A debugger requires a certain set of primitives for probing and controlling the target machine. Such primitives include reading memory, reading registers, single stepping, and setting breakpoints. Adapting a simulator to the primitives required by a specific debugger enables symbolic debugging of the simulator workload.

In addition to primitives required by a debugger front-end, the simulator provides services not normally available in a debugger. The service most relevant for real-time analysis is the ability to present current time with cycle count granularity. It enables the user to step through a portion of code, checking for both functional and temporal errors.

3.2 User process debugging

In time sharing operating systems, such as Linux, each program runs in a protected environment, with private registers, memory, and operating system resources. This is referred to as a *virtual machine*. In order for a debugger to debug a program, it needs to control execution and probe state of the corresponding virtual machine. A traditional debugger does not implement this controlling mechanism. Instead, it uses an interface provided by the operating system, illustrated in Figure 3. The operating system performs necessary administrative tasks, such as virtual memory translation and virtual register lookups. It thereby exports an image of a consistent virtual machine to the debugger.

As a simulation-based debugger runs in a different operating system than the target process, it cannot use operating system services to probe the target, and is restricted to simulator services. The simulator provides primitives for probing hardware state, such as register, memory, and disk contents. This information is useful for probing the operating system itself, which is the program running directly on the hardware. Without post-processing, however, the information is not useful for analysing user space processes.

In order to support debugging of processes in the simulated system, we introduce an intermediate filter between the debugger and the simulator, the *virtual machine translator* (VMT). The VMT answers a debugger's queries for virtual machine state. Queries for memory content refer to virtual addresses, and must be translated to physical addresses. The VMT performs this translation, which is normally provided by the operating system. It starts by looking up the head of the process list, which is a global variable whose address is found in the kernel symbol table. It finds the appropriate process entry by following pointers referring to data structures in the simulated memory. It proceeds in the same way in the process's page table until the mapping for the virtual address is found. If the page resides in physical memory, the VMT queries the simulator for the contents and responds to the debugger. If the sought page is paged out to disk, the VMT needs to walk the kernel data structures further to find which disk block it was written to and query the simulator for disk contents.

The debuggers are separate programs, communi-

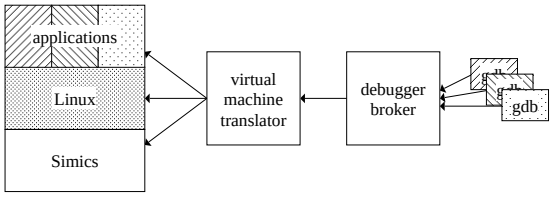


Figure 4: Simulation-based debugger. Arrows represent directions of data queries.

cating with the VMT via a proxy, a *debugger broker*, which supports concurrent debugging of multiple processes in the simulated system. The broker keeps state for each debugger connected and maps debugger queries to the corresponding virtual machine. It controls simulator execution, stepping forward only when all connected debuggers are ready to execute. The setup is shown in Figure 4. The debugger broker also makes sure all debugger breakpoints refer to physical memory. It maintains a list of breakpoints in use and inserts (removes) simulation breakpoints when code pages are mapped (unmapped) to physical memory.

3.3 Virtual machine translator implementation

Writing code for parsing data structures of a simulated system is a tedious and error-prone process. In order to simplify and automate the task, a few engineering tricks have been employed.

The symbol table for the operating system kernel contains information on the size and memory layout of all types defined in the kernel source code. We have implemented a meta-tool that uses a symbolic debugger to parse a symbol table, and then generates a C++ class for each type found in the table. The code generated includes routines for constructing objects by probing simulated memory. This code generator provides a programming environment with strong typing, enabling compile time checking for simple mistakes. It also facilitates performance optimisations, for example cached variable lookups or lazy memory probing. Moreover, the Linux kernel is written in C, which is a subset of C++. Therefore, definitions from the kernel source code can be reused with little or no modification. The dependencies between VMT modules are illustrated in Figure 5.

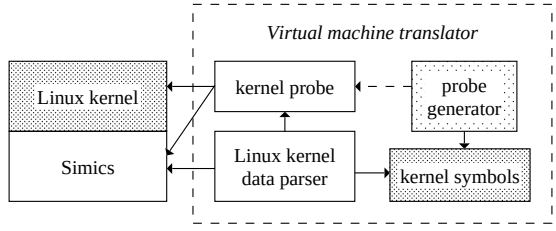


Figure 5: Implementation of the virtual machine translator. Solid arrows represent data queries. The dashed arrow represents code generation.

The Linux-specific module for parsing kernel structures uses the generated code whenever it needs to probe kernel memory. It is therefore independent of the exact code layout, and needs to query the simulator directly only occasionally. This independence provides some portability and robustness when porting to new architectures or kernel versions.

3.4 Real-time debugging example

As a demonstration of debugging soft real-time applications, we present examples from a debugging session of the MPEG video decoder `mpeg_play` [13]. The decoder displays a video clip with a frame rate of 30 frames per second. In this example, Simics is used to model an UltraSPARC workstation. The simulated machine boots from a disk image containing an installation of UltraPenguin Linux 1.1. The system is configured to run the MPEG decoder at boot. The system also runs standard Unix daemons and, in order to make the workload more complex, a CPU-bound background task with low priority. The startup sequence is shown in Figure 1.

We have executed simulation forward until the video decoder is started, and attached GDB to the `mpeg_play` process. In order to present simulated time flow, the modified GDB provides a magic variable, `sim_time`. Whenever the variable is referenced, GDB queries the simulator for the number of cycles executed since boot. As the variable is integrated into

GDB, it can be used as any other program variable, for example in GDB scripting or conditional breakpoints. Other types of simulator information, such as hardware or operating system statistics, can be presented in a similar fashion.

Listing 1 shows how we set a breakpoint at the routine `ExecuteDisplay`, which is called at the end of the rendering loop. We use the `sim_time` variable to check whether the deadline was met for each frame. The machine runs at 225 simulated megahertz, and must therefore render a frame in less than 7.5 million cycles to meet the deadline. A short GDB command sequence locates the first missed deadline for us.

Listing 1 Example of locating a missed deadline.

```
(gdb-sim) break ExecuteDisplay
Breakpoint 1 at 0x200cc: file gdith.c, line 942.
(gdb-sim) continue
Continuing.

Breakpoint 1, ExecuteDisplay (vid_stream=0x11a160,
    frame_increment=1, xinfo=0x116654) at gdith.c:942
942       if (xinfo!=NULL) display=xinfo->display;
(gdb-sim) display $sim_time
$1 = 979949561
(gdb-sim) set $start = $sim_time
(gdb-sim) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>silent
>printf "Frame completed at %d, in %d cycles\n", $sim_time,
    $sim_time - $start
>if $sim_time - $start > 7500000
 >printf "Deadline missed\n"
 >else
 >set $start = $sim_time
 >continue
 >end
>end
(gdb-sim) continue
Continuing.
Frame completed at 980856912, in 907351 cycles
Frame completed at 981744876, in 887964 cycles
Frame completed at 989462277, in 7717401 cycles
Deadline missed
(gdb-sim)
```

When a missed deadline is found, the user can restart the session and investigate the unsatisfactory behaviour in detail. As the simulated machine is deterministic, an identical execution will be observed.

4 Profiling real-time applications

A profiling tool assists a programmer in deciding where in a program to make optimisations. The most common type of profiling tool is the performance profiler. It measures execution time spent in different code sections, and shows the programmer where the majority of execution time is spent. This gives the programmer a hint on where to concentrate his efforts to improve program performance. Certain advanced profilers also collect statistics on how well hardware or operating systems resources are used. Such statistics may explain why a program fails to meet performance expectations, and are used to suggest to the programmer how to modify the program.

When optimising soft real-time applications, the programmer aims to minimise the number of missed deadlines. Traditional profiling tools are inadequate for this purpose, as they present total time spent in a code block, measured over the whole execution. Instead, a real-time profiler should present differences in execution times between iterations when the deadline was missed and iterations when the deadline was met. This presentation gives the programmer a hint as to which routines he should modify to eliminate deadline misses.

4.1 Source code profiling example

In order to explain why the video decoder missed some deadlines in the example in Section 3.4, we will measure and compare time spent in a subroutine. By running `mpeg_play` in a conventional performance profiler, we observe that most of the execution time is spent in the function `OrderedDitherImage`. We set breakpoints at the start and end of this function and measure execution time for each iteration with a simple script (similar to that in Listing 1). The results are shown in Figure 6. The axes correspond to execution time spent in `OrderedDitherImage` and rendering time for the whole frame. Surprisingly, `OrderedDitherImage` executes in constant time — except for a few odd values, caused by interrupts and page faults — and there is no correlation between a

deadline miss and execution time spent in this function.

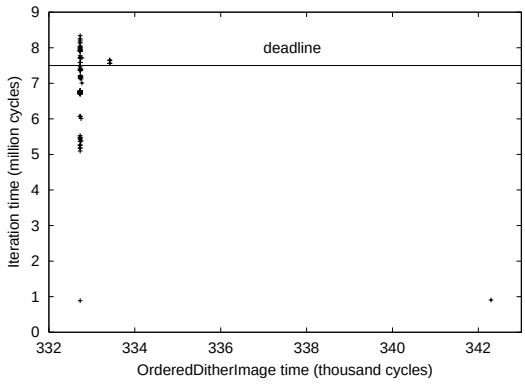


Figure 6: Correlation to OrderedDitherImage execution time.

We could continue searching for causes of missed deadlines by measuring execution time spent in other functions, hoping they would show better correlation. Instead, we will present how to perform instrumented profiling by correlating deadline violations with other metrics.

4.2 Instrumented profiling

Execution time for a code section may differ between iterations, even though the program executes along identical paths. In a general purpose operating system, the stochastic behaviour of the memory hierarchy, I/O devices, and competing processes affects execution time in unpredictable ways.

Variations in execution time between iterations may be explained by counting system events, and comparing event statistics for each iteration with execution time. We collect statistics from different levels in the system using different methods.

- Hardware level. The simulator counts performance related hardware events occurring in the simulated machine. Events counted include cache misses, TLB misses, and I/O transactions.
- Operating system level. The virtual machine translator is able to collect three types of statistics.
  - Event count. Some operating system events, such as page faults and context switches, have great impact on performance. These are counted by inserting a breakpoint in a corresponding kernel routine, for example the scheduler. Whenever the breakpoint is triggered, the event counter is incremented.
  - Kernel variable values. Kernel variables may be sampled by reading simulated memory, and the addresses are found in the kernel symbol table. The process id of the current task is an example of a useful kernel variable.
  - Computed kernel variables. Variable values that are not available immediately in memory are calculated using a specific routine for each variable. A traditional debugger would execute code in the target machine to perform such computations. This method is unacceptable for a simulation-based debugger, as it would modify the simulation and make it non-deterministic. Instead, the user must implement a lookup routine for each computed variable. The length of the run queue is an example of a computed kernel variable.
- Application level. The user may choose to include statistics from application events and variables in the same way as for the operating system. Routines for collecting application statistics are best implemented on top of the debugger interface, as the debugger services are necessary. The code generation techniques used for implementing the virtual machine translator (described in Section 3.3) are applicable also for programming application level probing.

The ability to collect execution statistics from multiple levels in a computer system is very useful. It allows efficient problem solving without requiring knowledge about the nature of the problem. Most

performance tools collect data from a single level, and therefore require that the programmer knows which tool to use and where to look.

4.3 Instrumented profiling example

We can now proceed to try to explain the missed deadlines in the MPEG video decoder by correlating statistics from the application and operating system level.

Listing 2 Reading application variables.

```
(gdb-sim) list video.h:100
97      /* Macros for picture code type. */
98
99      #define I_TYPE 1
100     #define P_TYPE 2
101     #define B_TYPE 3
(gdb-sim) break ExecuteDisplay
Breakpoint 1 at 0x200cc: file gdith.c, line 942.

(gdb-sim) set $start = $sim_time
(gdb-sim) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>silent
>printf "Frame number %d", TotalFrameCount
>printf ", type %d", vid_stream->picture.code_type
>printf ", %d cycles\n", $sim_time - $start
>set $start = $sim_time
>continue
>end
(gdb-sim) continue
Continuing.
Frame number 9, type 1, 7933769 cycles
Frame number 10, type 3, 7179460 cycles
Frame number 11, type 3, 6730860 cycles
Frame number 12, type 2, 7364510 cycles
Frame number 13, type 3, 6062254 cycles
Frame number 14, type 3, 6742310 cycles
Frame number 15, type 1, 7897064 cycles
```

An MPEG stream contains a compressed representation of video frames. For some frames, the whole frame is encoded as a JPEG picture. These are referred to as I frames. Other frames are represented only by the difference to past or future frames (P and B frames). The decoder handles each frame type in a different way. Thus, decoding time is probably dependent on the frame type. In order to find out whether most deadline violations occur for a particular frame type, we instruct the debugger to break at each iteration and print the variable containing the frame type code. The debugger commands are shown in Listing 2.

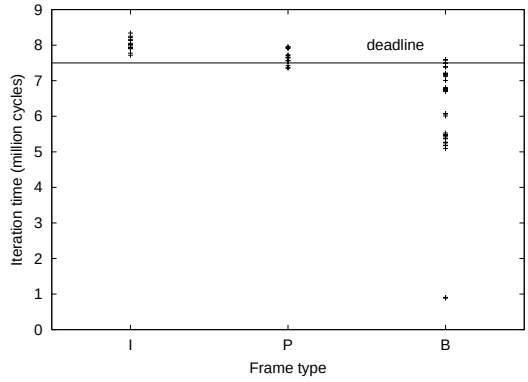


Figure 7: Correlation to MPEG frame type.

The results, shown in Figure 7, indicate that most deadline violations occur while decoding I and P frames. This implies that the programmer of `mpeg_play` should concentrate optimisation efforts on the code executed during I and P frame decoding. Optimising code performance is often difficult, however, and the programmer may want to tune other system parameters to improve application quality. Figure 7 shows that rendering time varies, even for frames of the same type. We suspect that this variance is caused by the background load. In order to measure the amount of CPU resources consumed by the competing program, we instruct the simulator to insert a breakpoint at the point where Linux returns from kernel to user space, and inform us whenever a new process is scheduled. This procedure is shown in Listing 3. A comparison of rendering time and time spent in other processes is shown in Figure 8, indicating a clear correlation. Therefore, limiting CPU usage of other processes, for example by using a real-time scheduling policy, may improve application performance in the scenario presented. The correlation may also be a side-effect, if the application spends a large amount of time waiting for I/O. By measuring time spent on blocking system calls, the programmer can find out whether the application misses deadlines because of I/O wait. He can continue validating his theories this way, through further measurements and correlations, until he has obtained sufficient under-

standing of the causes of deadline violations.

Listing 3 Measuring context switches.

```
(gdb-sim) break ExecuteDisplay
Breakpoint 1 at 0x200cc: file gdith.c, line 942.
(gdb-sim) set sim os-context-switches on
(gdb-sim) continue
Continuing.
Context switch from 105 to 106 at 981749467.
Context switch from 106 to 105 at 987309788.

Breakpoint 1, ExecuteDisplay (vid_stream=0x11a160,
    frame_increment=1, xinfo=0x116654) at gdith.c:942
q942        if (xinfo!=NULL) display=xinfo->display;
(gdb-sim)
```

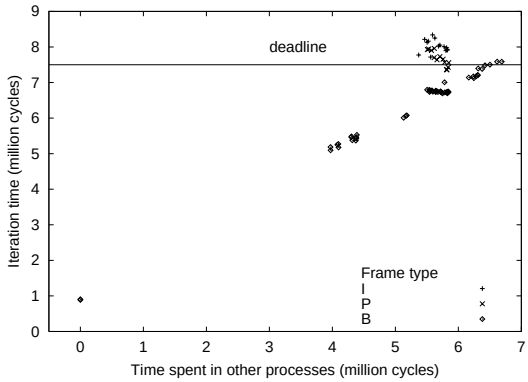


Figure 8: Correlation to time spent in other processes.

5 Related work

In many existing real-time operating systems and environments, only conventional, non-real-time debugging tools are available. These systems may only be used for validating and debugging functional, not temporal, correctness. There are however vendors providing support for alternative debugging methods. Some of these methods are discussed below.

Developers of programs for small embedded systems commonly use an emulator (tool for executing programs in foreign environments) as debugging back-end. Emulators generally focus on the functional model and do not provide a time model, which is necessary to avoid intrusion and to reproduce sessions. Some vendors, for example Motorola [2] and Wind River [22], provide simulators with models of caches and CPU pipeline, resulting in good execution time modelling. The tools generally available are either too incomplete to run general-purpose operating systems or too slow to run desktop applications.

Support for non-interactive debugging of real-time programs may be provided by logging execution to a trace, which is sent over a network to a separate system. The trace may be generated by dedicated hardware [7, 16, 20] or additional program code [4, 15]. Both approaches are inconvenient, and the amount of monitoring is limited. Furthermore, data exploration is limited to the data subset collected.

The R2D2 debugger [17] 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. Sessions cannot be authentically repeated, however, and the target system may be unable to provide information when it is under stress.

Mueller and Whalley [14] 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. This prediction does not take operating system effects into account and works best for small programs.

The work in this paper is made possible by many different advances in simulator implementation technology [3, 5, 8, 9, 10, 19]. 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 [5, 11]. The SimOS project [8] has made similar achievements, although the simulator presented does not model a complete binary interface and requires operating system modifications. Due to the accurate timing model provided, complete system simulators have proven to be effective tools for performance profiling [8, 12, 18].

6 Conclusions

The primary contribution of this paper is the temporal debugger, an important tool for development of soft real-time applications. The temporal debugger implementation is based on the GNU debugger, modified to connect to a simulator modelling a complete workstation. The simulator runs unmodified operating system and application software and allows reproducible analysis of system time flow.

The major difficulty in debugger implementation is the mapping of low-level simulation data to application-related data useful to the debugger. We address this problem using a new technique, virtual machine translation, which operates by traversing data structures in the operating system kernel. This technique has been implemented for UltraSPARC Linux systems.

We demonstrate how the temporal debugger can be used to detect missed deadlines in an MPEG video decoder, running in Linux. We also show how the debugger makes application and operating system internals visible during simulation, enabling collection of runtime statistics from different system levels. The statistics are correlated to deadline misses in the video decoder, thereby explaining the causes of deadline violations.

Currently, the temporal debugger environment supports manual examination, or simple automated operations using GDB scripts. A programmer would benefit more from an automated profiler, visualising correlations between missed deadlines and many different types of statistics, for example cache misses, function call arguments, page faults, and branch decisions. As the debugger environment allows non-intrusive probing, an automated tool can measure large amounts of data eagerly, without compromising accuracy.

References

[1] Lars Albertsson and Peter S. Magnusson. Using complete system simulation for temporal debugging of general purpose operating systems and workloads. In *Proceedings of MASCOTS 2000*. IEEE Computer Society, IEEE Computer Society Press, August 2000.

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

[3] Robert C. Bedichek. Some efficient architecture simulation techniques. In USENIX Association, editor, *Proceedings of the Winter 1990 USENIX Conference, January 22–26, 1990, Washington, DC, USA*, pages 53–64, Berkeley, CA, USA, January 1990. USENIX.

[4] 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.

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

[6] The GNU debugger, version 5.0. http://sources.redhat.com/gdb.

[7] 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.

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

[9] Peter Magnusson and Bengt Werner. Efficient memory simulation in SimICS. In *Proceedings of the 28th Annual Simulation Symposium*, 1995.

[10] Peter S. Magnusson. Efficient instruction cache simulation and execution profiling with a threaded-code interpreter. In *Proceedings of Winter Simulation Conference 97*, 1997.

[11] 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.

[12] 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.

[13] The Berkeley MPEG player, version 2.3. http://bmrc.berkeley.edu/frame/research/-mpeg/mpeg_play.html.

[14] 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.

[15] 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.

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

[17] The R2D2 debugger, Zentropix. http://www.zentropix.com.

[18] 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.

[19] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.

[20] 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.

[21] Virtutech Simics v0.97/sun4u. http://www.simics.com.

[22] Wind River. http://www.windriver.com.

Methods for Increasing Software Testability Extended abstract

Birgitta Lindström, Jonas Mellin, and Sten Andler

Testing is the act of executing a system or application, in order to show the presence of faults. The confidence in the system can thereby increase as faults are removed. It can, however, never be used to prove that the system is correct. The reason for this is that it is generally infeasible to conduct exhaustive testing, since the number of test cases to test all possible outcomes is too high. It is a well-known fact in software engineering that the cost for testing of software often is 50% or more of the development costs. Hence, methods to improve testability, i.e., to reduce the effort required for testing, have a very large potential to decrease development costs.

In this paper, controllability and observability, (Schütz 1993), are considered necessary prerequisites for testability. Without the ability to control test execution, and the ability to observe important outcomes such as state transitions, it is not feasible to reach a high level of testability. A system, which lacks controllability and observability, is always difficult to test. However, testability also depends on the number of possible execution orders (i.e., interleavings of threads). An often neglected circumstance which is especially important for distributed or real-time systems. The reason for this dependency is that observability and controllability only apply to test cases that are actually executed. Given a certain amount of test effort, the test coverage decreases with an increase in the number of distinct execution orders and hence, test cases. Achieving higher test coverage (i.e., executing a higher ratio of all possible test cases) increases the required test effort, since it requires designing, executing, and analyzing more test cases (Birgisson, Mellin and Andler 1999).

As we have shown that testability is dependent of the number of distinct execution orders, it is a natural step to investigate the execution environment and its impact on the number of execution orders. Which execution orders are allowed depends to a large degree on processor scheduling and concurrency control policies. In a survey and analysis of current methods for improving testability (Lindström 2000), several such methods are investigated and their properties with respect to processor scheduling and concurrency control analyzed. Especially, their impact on the number of distinct execution orders is discussed. The survey reveals that (*i*) there are few methods which explicitly address software testability, and (*ii*) methods that concern the execution en-

vironment for real-time systems require or favor a time-triggered design.

Kopetz (1991) points out that testability depends on the architecture and must be considered during the design phase. The first comparison of time-triggered and event-triggered systems with respect to testability is made by Schütz (1993). Schütz shows that the effort to test an event-triggered real-time system is inherently higher than that of a time-triggered real-time system. This is due to the dynamic nature of event-triggered systems, which makes the number of distinct execution orders high in comparision to time-triggered systems.

A time-triggered design is, however, not always suitable. The reasons to choose an event-triggered design include the need for a flexible system in unpredictable environments. Usually there is a need for graceful degradation. Therefore, it is an interesting issue to investigate which properties of the execution environment in event-triggered systems that lead to high or low levels of testability. Some of these properties are identified by Lindström (2000) and others remain as open problems. These properties are used to define categories, which form a basis of taxonomy for testability.

It is important to note here that we have no intention to discriminate between good and bad solutions based on testability alone. We are, however, interested in the impact on testability from different properties of the execution environment, as design is always a matter of trade-off decisions. The important thing is that these decisions should be based on as much information about the consequences of the choices as possible.

The scheduling properties that are investigated by Lindström (2000) are preemption policy, task priority policy, and observation policy. It is shown that there is a significant influence on testability from the choice of scheduling properties. One example is the priority policy, which has two dimensions of complexity that do have an impact on the test effort:

1. Whether the task priority is unique during the execution of the task,
2. Whether the priority ordering between two tasks can switch during task execution (e.g., priority ceiling protocol)

A priority policy that guarantees unique priorities during execution does not increase the test effort, since for each given set of tasks we will have exactly one distinct execution order. However, if the priorities are non-unique, the number of potential execution orders for a set of k tasks equals $k!$ given a non-preemptive scheduling policy. The proof is trivial, consider the case when all k tasks are assigned the same priority. In this worst case, the execution order is totally arbitrary which means that all permutations of k are possible. In a more average case, we might have a distribution of the tasks over levels of priorities. Suppose we have three levels of priorities, then for each set of tasks we will have $x!*y!*z!$ distinct execution orders, where x, y, and z are the number of tasks in the set with priority level 1, 2, and 3 respectively. The question of whether priorities are guaranteed to be unique is of significant importance for the system testability. It separates the systems into two different degrees of testability where the number of test cases either grows

with combinations of the number of tasks or permutations of the number of tasks.

A priority policy that allows the priority order between tasks to be changed arbitrarily during the execution has a negative influence on testability. The change can be regarded as an event that changes the execution order for the set of current tasks. If these changes are arbitrary, all execution orders are possible for a given set of tasks. Hence, this is also a property that separates systems into two different levels of testability.

Similar results have been shown for several other execution environment properties. The conclusion is that there is an obvious impact on testability from the execution environment. Further investigations are necessary to identify other properties that are important for testability. It is also of central importance to determine the magnitude of influence from the different properties on testability and any interrelationships between them that might affect their influence. Of course, some combinations of properties may not be meaningful.

The completion of the initiated research into properties that affect testability will lead to a taxonomy of testability. The benefit of such a taxonomy is that it enables designers of event-triggered systems to regard testability when making informed trade-off decisions.

References

Birgisson, R., Mellin, J. and Andler, S. (1999). Bounds on Test Effort for Event-Triggered Real-Time Systems, *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*.

Kopetz, H. (1991). The Design of Real-Time Systems, *Software Enginering Journal* **6**(3): 72–82.

Lindström, B. (2000). *Methods for increasing software testability*, Master's thesis, HS-IDA-MD-00-017, University of Skövde.

Schütz, W. (1993). *The Testability of Distributed Real-Time Systems*, Kluwer Academic Publishers.

Test-Case Generation for Testing of Timeliness*

Robert Nilsson, Sten F. Andler and Jonas Mellin

February 27, 2001

Abstract

In complex distributed real-time systems the temporal correctness is imperative for dependability. Industrial practice has few methods for testing of temporal correctness and the methods that exist are often ad-hoc. A problem associated with testing real-time is that their timeliness depends on the execution order of tasks. This is particularly problematic for event-triggered real-time systems where the system continuously is notified of events that influence the execution order. Further, real-time systems may behave differently depending not only on time dependencies in program logic but also on differences in execution times. This paper investigates how existing test-case generation methods take these factors into consideration and explains the opportunity to construct better methods and metrics for their evaluation.

An analysis of current methods for generating test cases for testing temporal properties of real-time systems is presented. The methods are classified according to their characteristics and analyzed in relation to the requirements of an event-triggered system model. The aim of the classification is to detect similarities between different test-case generation methods so that the characteristics that have an impact on the applicability of the method when testing timeliness can be determined. We conclude that existing work only consider a subset of the factors that influence timeliness, and therefore propose research activities that take these factors into consideration during test-case generation.

1 Introduction

Modern real-time systems tend to be increasingly complex. This is particularly true for distributed real-time systems, where the complexity of both distribution and real-time issues need to be considered. Moreover, real-time systems generally must be dependable and therefore temporal correctness of the software is imperative. These characteristics imply that there is a need for rigorous verification methods to detect temporal faults that could lead to critical failures. Industrial practice has few methods for verifying temporal correctness of complex systems such as distributed real-time applications, and the methods that exist are often case-specific or ad-hoc [Sch94, BJ00].

Testing is a method to dynamically verify software by execution in order to detect errors and failures [Lap94]. *Test-case generation* is the process of selectively generating test cases that exercise system behaviors likely to reveal errors.

This paper focuses on selective generation of test cases for distributed real-time systems. There exist generally accepted methods for generation of test cases for sequential software, based on, for example, specifications or code structure [Bei90]. Real-time systems are often concurrent; this complicates generation and selection of suitable test cases because system behavior is dependent on the non-deterministic order in which tasks execute (cf. [TH99]). Further, real-time systems may behave differently depending on time dependencies in program logic and differences in execution times. In particular, we investigate how existing test-case generation methods to a varying degree take these factors into consideration for testing *timeliness*, which is the ability of the system to fulfill its time constraints.

The investigated test-case generation methods have been classified according to their characteristics and analyzed in relation to the requirements of an event-triggered system model.

*This work is funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Strategic Research.. Submitted for publication

More specifically, our model of the target system adopts the event-triggered design paradigm as presented by Kopetz et al. [KZF+91], and constrains the execution environment to increase testability, as described by Birgisson et al. [BMA99]. The aim of our classification is to reveal similarities between different test-case generation methods and allow determination whether a certain type of characteristics has an impact on the applicability of the method to testing of timeliness. Furthermore, the result of the classification is analyzed against the requirement of our model of the target system in order to highlight issues that have not been covered in existing research.

2 Test-Case Generation Model

Test-case generation is based on application knowledge such as specification, code structure, and system design (see figure 1). This information is used to selectively generate a set of test cases, sometimes referred to as a *test suite*, that has high probability of revealing a specific class of errors or deviations from the expected behavior [Lap94].

An *execution environment*, in this context, is the architecture in which the tested applications run. This includes real-time operating system services, communication primitives, and properties of underlying hardware. The execution environment gives certain constraints on applicability of different test methods, e.g. required contents of the test cases. Hence, we suggest that properties of the execution environment should be considered in the test-case generation process.

Test cases are generated in accordance with the test-case selection criterion, which depend on the specific test method that is being used. In this paper a *test coverage criterion* denote the level of ambition for a test method, i.e., test coverage criterion sets a goal of when an application has been tested sufficiently. A *test-case selection criterion* is related to this – a criterion for selecting test cases that eventually will fulfill the test coverage criterion.

When a test suite has been prepared, it is executed on the system under test. The test-case execution phase requires that the execution environment is sufficiently controllable and observable so that the desired test scenario can be enforced and executed. The term *observability* denotes how well the system provides facilities for monitoring or observing the execution of application programs during testing [Sch93]. *Controllability* is associated with how well a tester can control the behavior of the system under test [Sch93].

Test evaluation is the process of comparing the result of test-case execution with the expected result. Test evaluation also incorporates evaluation against the test coverage criterion to determine if sufficient test coverage has been attained.

3 Generating Test Cases for Testing of Timeliness

This section presents the state-of-the-art in test-case generation for testing of time constraints in real-time systems. We argue a study of this area is interesting and what problems are solved by the analysis. The methods that are relevant, from a timeliness perspective, are classified according to a set of attributes in section 4. These attributes and the criteria for classification are explained in section 3.2.

3.1 The need for Testing of Timeliness

There is a plethora of articles describing different methods for generating or selecting tests for various target systems and frameworks. As a developer in the domain of complex distributed real-time systems, it is hard to know what test-case generation methods are meaningful for the properties that one wants to test in this domain. Timeliness is one of the most important properties of real-time systems. Few articles consider testing of timeliness, but refer to formal proofs, static analysis and scheduling theory to guarantee timeliness. Such analysis often requires exact estimations of worst-case execution times, known load patterns, and off-line scheduling. However, event-triggered real-time systems often allow a task load with mixed criticality and dynamic scheduling policies to be more flexible. This complicate analysis and proofs while testing becomes necessary in order to achieve confidence in the temporal correctness. By investigating and classifying methods for generating test cases from the timeliness perspective it is possible to determine what aspects already have

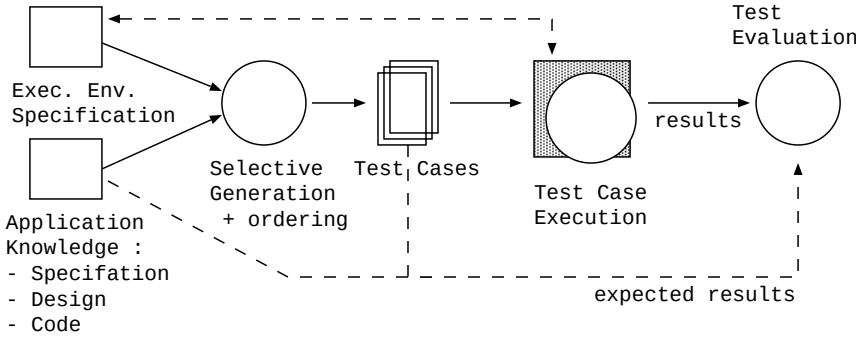


Figure 1: High-level testing model

been addressed and what aspects needs further consideration and research when generating test cases for dependable real-time systems (see also [Nil00]).

3.2 Classification Attributes

The methods that have been encountered in literature are classified in relation to each other, forming a taxonomy over test-case generation methods for testing of temporal properties. The attributes have been selected so that they should apply to all test generation methods that consider time constraints.

3.2.1 Specification Type

This attribute classifies the methods by the type of specification from which test cases are derived. A specification in this context is any model that incorporates the expected temporal behavior of a real-time system. Four main categories of specification type have been identified; process algebra, finite state machines, temporal logic, and Petri-net specification. The aim for this classification is to identify if a test-case generation method requires a specification type that has inherent advantages or problems when used for testing of timeliness.

Some test-case generation methods, used for testing temporal properties, are independent of the specification. These methods often rely on some other, possibly manual, method for verifying temporal correctness. Examples of such methods are included in this article for completeness but will not be classified by this attribute.

3.2.2 Test-Case Contents

Different methods supply various degree of test-case detail. Some methods supply only "test-data" and rely on some other mechanism to verify the implementation against a temporal specification. These methods are not specifically aimed at testing temporal correctness; thus when the execution time of a task depends on the input data, appropriate input data for such testing must be specified.

Other methods aim at generating complete execution sequences with series of input events and expected output events. This class of test data is denoted as "event sequences" in this work. A special version of event-sequence representation is executable "test processes", which are run in parallel with the application, supplying inputs at certain points in the execution and responses to outputs from the system. A test process can potentially incorporate timed input events, input data, and expected results.

Another category is test cases that include a specification of the initial internal state of the system from which the event sequence should be applied as discussed by Birgisson et al. [BMA99].

3.2.3 Time Base

From the timeliness perspective, the time base used in test-case generation is interesting. A continuous time base makes it possible to specify time constraints in a more precise way than in its discrete counterpart. However, this ability makes the number of test cases infinite.

Some methods assume a discrete time base, where there is a constant period between time

instants, which simplifies testing. This may be a valid assumption, since all currently used computer systems operate with a discrete time base at the lowest level, i.e., at the level of hardware clock cycles.

3.2.4 Degree of Automation

From a timeliness testing perspective a high degree of automation is desirable to reduce the test effort, and thus, to be able to execute more system-level tests in a limited time. Most methods for test-case generation recognize the need for automation and tool support for generating test-suites. In some cases, the methods support fully automated test-case generation. In other methods a tool allows the tester to specify aspects that needs to be tested, and the generation of test cases is then performed in semi-automated interaction with this tool. Some methods do not consider automation explicitly but do have the required properties needed to support an automated solution. Hence, four possible categories have been identified; "Automated", "Semi-Automated", "Potentially Automated", and "Not Automated".

4 Test-Case Generation Methods

This section presents the most significant test-case generation methods from the timeliness perspective. The layout of this section follows the categories of the "specification type" attribute.

4.1 Methods Based on Process Algebra Specifications

An example of a method based on process algebra is presented by Clarke and Lee [CL97]. In this paper they introduce a framework for testing time constraints of real-time systems. The time constraints are specified in a constraint graph that specifies intervals in which events can occur and dependencies between events. The graph is translated to processes in the algebra of communicating shared resources (ACSR). Test processes produced from the process algebraic specifications can be used to verify the specification model of the system as well as be converted to test cases for the actual software implementation. The test processes are run in parallel with the system under test to supply timed inputs to the system and intercept outputs. The article provides a taxonomy of timing-constraint faults and a set of coverage criteria for detecting various types of implementation faults. The recommended test case generation method focuses on deriving test cases close to the end-points of the intervals at which events are specified to occur.

The target system for the presented is real-time systems and protocols. The generated test cases are primarily aimed at testing of ASCR specifications, but the authors conjecture that test cases can be reused for testing of actual software implementations. This method is interesting from the perspective of our target model since input occurs continuously and time constraints are specified in a well-formed way. However, the focus of this test-case generation method is to test "behavioral constraints" – time constraints on the input to the system. No method for generation of test cases for testing of time constraints on the output values is presented.

Cleveland and Zwarico [CZ91] propose a method that incorporates a framework for generating "behavioral preorders", which is a specification in process algebra that is suitable for reasoning about real-time characteristics of reactive systems. They are mainly concerned with the problems arising when describing timed systems in a process algebraic notation. No concrete method for test-case generation for real-time software systems is provided.

4.2 Methods Based on State Machine Specifications

A finite state machine is a well-known abstraction of computer systems. An interesting property of real-time systems is that a point in time can be viewed as an event that may cause a transition in a finite state model. When testing systems for non-temporal properties, methods based on finite-state-machine representations have already been exploited for test-case generation, e.g., [FBK+91]. In this section, methods for testing temporal properties based on three different flavors of state machine models are presented.

A common problem in finite-state-machine representations is the state explosion problem. This problem often occurs when the number of states in the actual implementation is greater than the number of states in the finite state

model. Another problem with these models are events that could lead to more than one state, i.e., non-determinism [FBK+91].

4.2.1 Timed I/O Automaton

Petitjean and Fochal [PF99a] propose a test architecture for testing of timed systems. A timed system is described as a timed automaton that can be expressed as a region graph. This work is built on the timed- I/O automaton theory proposed by Alur and Dill [AD94]. In their model every automaton has a set of states and a set of clocks. The clocks are represented by real numbers that proceed at the same rate and can be reset by any transition in the system. The refinement of this model by Petitjean and Fochal are clock regions that allow time constraints to be represented by a region graph where the number of clock regions becomes smaller than in the original model by Alur and Dill.

The Petitjean and Fochal paper further describes how test cases are exercised in the system under test. The architecture takes the management of clocks away from the implementation and lifts it to the level of the tester, because it must use timers that correspond to the real-valued clocks in the specification. Thus, real-value clocks are kept within a read-only module of the test-driver, equivalent with the clock representation in the system. When the system under test generates output events, the test driver verifies these against the clock zones. However, it is inconclusive how efficient this test case generation method is

In order to get sufficient test coverage, test cases are selected from each vertex that makes up a clock region in the region graph. This policy guarantees that at least one test case has been exercised for each combination of time constraints. We believe this approach to be promising for generating test suits for testing timing constraints. However, the method abstract away from properties in the execution environment and it is inconclusive if the generated test cases can be used in our target model, this issue needs investigation.

4.2.2 Timed Transition Systems

Another method which takes a finite-state-machine approach is Cardell-Oliver and Glover [COG98]. The purpose of their article is to present a practical and complete method for generating conformance tests for real-time systems. The paper starts from a formal-methods perspective, where software specifications is iteratively refined and formally verified at each step. The authors note that the last step from symbol manipulation to an actual hardware and software implementation requires testing since the implementation no longer is a provable description.

According to Cardell-Oliver and Glover, a formal test method has four stages; checking of the test hypothesis, generating test cases, running the tests, and evaluating the test results. The test hypothesis defines the assumptions about the property under test so that correct conclusions from the test results can be drawn. This method requires that the system under test can be viewed, at some level of abstraction, as a deterministic finite state automaton. According to the paper, a test method is complete if "its test generation algorithm determines a finite set of test cases sufficient to demonstrate that the formal model of the implementation under test is equivalent to its specification". The specification language used in this method is Timed Transition Systems (TTS), but the authors claim that the method is adaptable to other formal languages such as timed CSP.

The basic idea of the method is to generate test cases that exercise all transitions between states and compares observable variables with their specification counterparts. Each transition in the TTS specification has an associated clock guard in discrete time units that is used for expressing time constraints. The required input for this method is a specification of the system as a TTS process as well as a specification of its environment. From these models, a timed action automaton is derived by symbolic execution. The test-case generation algorithm takes the timed action automaton associated with a TTS specification and constructs a finite set of test cases that will test each timed action edge of the automata. Each test is a sequence of timed actions.

The authors note that their method results in a large number of test cases, but suggest that other methods should be applied to eliminate redundancy. They also claim that if a unique sequence of output events can identify each state, the number of required test cases for conformance testing can be decreased.

4.2.3 Extended Timed Input-Output State Machines

Koné [Kon95] introduces a method for designing tests for time constraints. However, the main contributions of his work are a formal approach to modeling and a theory for testing of time dependencies in system behavior. The approach will not be described in detail here since it is similar to the methods described above and does not handle test-case generation for software explicitly. However the article is mentioned in this context since it is one of the seminal works in the area.

Laurencot and Castanet [LC97] show different existing methods to integrate time in existing formal modeling languages so that it is possible to test time constraints. The article also aims to show how the notion of time can be integrated in test cases for distributed systems and protocols. The importance of time in test cases for time-dependent systems is highlighted. Three formal specifications from other authors are presented in their overview and the notion of time in each of them is discussed. The different specifications are the automata of Alur and Dill [AD94], Timed Transition Models (e.g., [COG98]), and Extended Time Input Output State Machine (ETIOSM) [Kon95]. Based on the latter notation, a canonical "tester" that handles time is presented. The method decomposes timed formal models, in this case ETIOSM, into sub-models for each real-valued timer. These are then analyzed according to a set of rules and used for building the tester that incorporates time. According to the paper, test cases given by this method can be formulated in TTCN test language. An interesting issue is it that the authors claim that their tester is very well suited for testing distributed systems or protocols, because of the use of a transition for each transmission and reception. In addition, propagation delay is part of the models.

4.3 Methods Based on Temporal Logic Specifications

Mandrioli et al. [MMM95] suggest a method for functional testing of real-time systems based on specifications of system behavior in TRIO. TRIO is an extension of a temporal-logic language defined to deal explicitly with strict time requirements. A specification of behaviors is decomposed into elemental test-case fragments that later can be recombined to form test cases consisting of combinations of timed inputs and corresponding outputs. A test generation tool helps a test engineer to device test cases and includes history generator/checker components. The history generator is used for deriving all elementary test cases satisfying a TRIO formula at a given time instant. The elemental test cases are timed input-output pairs. These pairs can be combined and shifted in time to create a large number of partial test cases. In order to get a manageable test suite, a heuristic function combined with human guidance is used for selecting which elemental test cases should be combined in order to satisfy some test coverage criterion. The authors claim that most coverage criteria can be used with this method. A problem with this method, however, is how to relate actual output events instances to multiple specified output events of the same type. This problem is partly solved by a history checker that analyzes the generated history from the system under test. Further, this method assumes that all system behavior can be specified with logic specifications; for system level testing the number and complexity of logic formulas will increase and it is inconclusive how the method scales.

In a more recent work, SanPietro et al. [SMM00] expand the work by Mandriolli et al. to incorporate high-level, structured specifications that can be combined with the low-level specifications proposed in the earlier work. The main difference in this work is that the language allows modularization and presents a graphical notation of modules and components. The structured model is translated to a graph which is traversed by an algorithm to construct execution sequences for the overall system (cf. [MMM95]). The test-case generation semi-automatically constructs execution sequences in accordance with some coverage criterion. In our opinion, this extension increases the applicability of the method presented by Mandrioli et al. It would be interesting to further evaluate this method in the context of testing of timeliness and for example the effect of constraining the observation granularity.

4.4 Methods Based on Timed Petri-Net Specifications

Braberman et al. [BFM97] introduces a method for generating test cases for real-time systems based on timed Petri-nets. The method aims at

extracting knowledge not only from the specification or code, but from all steps during development, e.g. design. The article suggests the use of a design notation SA/SD-RT for specifying the behavior of a real-time system. For test purposes, this design specification is translated to a high level, timed Petri-net notation from which a timed reachability tree (TRT) can be derived. Each path from root to leaf in this tree represent a "situation" that in itself represent a potentially unlimited number of test cases due to a continuous time base. These "situations" are the base for generating test cases. The authors present different reduction schemes that can be applied to reduce the number of test cases resulting from each situation. The higher-level adequacy criteria presented in the paper are based on "situation" coverage, e.g. "one is enough" is satisfied if one test execution from each situation is executed. Listed future work is to create a tool which helps to manage and automate the method, an extension for applying the method on isolated components, and also to investigate how architectural information, such as scheduling policies, can be taken into account during test-case generation. An identified but unsolved problem with this method is that there is no way for deriving actual input test data, only temporal information about sequences of events can be produced, e.g. what timed input sequences are needed to test all situations. We believe that the method by Braberman et al. is one of the most promising methods in this survey. We share the opinion with the authors that it is of great importance to take advantage of the additional architectural design information during test-case generation.

In an earlier paper, Morasca and Pezze [MP90] propose a method for testing concurrent and real-time systems that uses high-level Petri-nets for specification and implementation. The Petri-nets proposed in their article are Er-nets, which can handle concurrency, where time is modeled as a special property associated with the transitions in the model. Problems with testing of concurrent and real-time systems are presented, high-lighting the problems with non-determinism and combinations of test-scenarios. Also, this paper presents a number of test criteria for selecting test cases for this kind of systems. This article takes problems with concurrency into consideration, but testing of the time constraints is not explicitly considered. Further, it is hard to determine how applicable it is for implementations that do not use Er-nets.

4.5 Methods for Enforcing Temporal Behaviors

Methods that are not based on a specification are generally not applicable for test-case generation aimed at testing temporal properties. The reason for this is that in order to determine if some transaction violates a time constraint, e.g. misses its deadline, the constraint must be specified in some way.

From our perspective, such test-case generation methods can still be helpful in finding test data that influence the temporal behavior of a transaction in a specific way. There exist other methods than test-case generation for finding data that cause the maximum execution time of a task, for example static analysis of code [PS91] and measurement [PF99b]. However, in static analysis, the value of the estimated maximum execution time is often pessimistic [MW98] and assumes that components such as caches and pipelines are turned of.

Wegener et al.[WSJE97] propose a method that uses genetic algorithms to generate test data for testing temporal properties of real-time systems. The heuristic method aims to find the longest and shortest execution paths of real-time programs. The paper shows that the genetic algorithm method easily outperforms a random method in finding the longest and shortest execution paths, measured in the number of clock cycles of program execution. The fitness function is supplied by a simulation tool, which counts the number of executed machine level instructions and summarizes the associated cycle times. This is only an approximation of the real execution time and the article states that temporal testing must be repeated for each platform where the software will run; this further emphasize the importance of automation. One problem is that there is no guarantee that a heuristic approach can find the extremes, the decision on when to stop the search for them is arbitrary.

4.6 Analysis and Classifications

Papers describing methods for test-case generation aimed for testing temporal properties are listed above. The result of our classification of these methods is shown in table 1.

Assuming that these papers represent all or most current methods aimed at testing time

Authors [Reference]	Specification	Test Cases	Time Base	Automation
Clarke and Lee [CL97]	Process Algebra	Test Process	Discrete	Automated
Cleaveland and Zwarico [CZ91]	Process Algebra	Test Process	Discrete	Not Automated
Mandrioli et al. [MMM95]	Temporal logic	Event Sequence	Discrete	Semi-Automated
SanPietro et al. [SMM00]	Temporal logic	Event Sequence	Discrete	Semi-Automated
Braberman et al. [BFM97]	Petri-nets	Event Sequence	Continuous	Semi-Automated
Morasca and Pezze [MP90]	Petri-nets	Event Sequence	Continuous	Potentially Automated
Laurencot and Castanet [LC97]	ETIOSM	Test Process	Continuous	Semi-Automated
Petitjean and Fochal [PF99a]	Timed Input Output Automata	Test Process	Continuous	Potentially Automated
Cardell-Oliver and Glover [COG98]	Timed Transition Systems	Test Process	Discrete	Automated
Koné [Kon95]	ETIOSM	Test Process	Continuous	Not Automated
Wegener et al. [WSJE97]	N/A	Test Data	Discrete	Automated

Table 1: Classification of Test-case Generation Methods

constraints, an observation is that the type of specification has influence on the contents of produced test cases. For example, temporal logic and Petri-net specifications generally generates test cases that consist of event sequences whereas test cases generated from some finite state machine or process algebra often are specified as test processes. None of the methods include information about internal states of the execution environment in the test cases and, thus, it is impossible to get full test coverage of non-deterministic internal behaviors.

Another observation from the table is that the time base used in these methods appears to be unrelated to the specification language or test case-type. A note to this classification is that in the method that use genetic algorithms for deriving input data [WSJE97], we have assumed that they use a discrete time base, e.g., clock cycles for measuring execution times. However, it is not relevant to these methods, since they rely on some other mechanism to verify time constraints and are only concerned with forcing a task to a specific temporal behavior.

One mayor conclusion that can be drawn from the presented methods is that very few consider internal states of the system when generating test cases. The only approaches that consider any form of such states at all are methods based on finite-state machines or Petri-nets. However, in these representations it is difficult to maintain a relationship between the states of the automata and the internal states of the computer system because the approach with these modeling methods often abstract too far away from mechanisms in the execution environment. Further, if non-determinism is allowed, finite-state-machine methods seldom apply due to state explosion problems.

Other methods consider the system as a black-box and only verify that the time from when input events are supplied to the time where output is signaled corresponds to the constraints in the specification. This is indeed an applicable method in some systems, but for dependable systems with non-deterministic execution times and execution orders, it is generally not sufficient to supply different sets of timed inputs to guarantee timeliness. Parameters and internal states are bound to have an impact on timeliness, i.e. different paths in the code are executed for different classes of input data, and thus, must be tested.

A related, important, observation is that very few of the specification based testing methods supply input parameters. The only input that is assumed is events and timing between events. This may be sufficient in some control applications, but in general, a real-time system has to read data values from sensors in its environment and may act differently upon different data values.

An interesting observation is that most methods assume an event-triggered paradigm; none of the test-case generation methods assume that input events are constrained in any way. The only constraint on system behavior in this context is that some of the works assume a discrete time base. A disadvantage with this is that it is hard to determine the impact of a constrained execution environment in existing methods. Hence, it would be interesting to investigate if constraints, such as the ones presented in [Mel98] could be used with these test-case generation methods to reduce the size of the test suite. For example, a lot of input sequences may be considered equivalent in a more sparsely observed environment.

The only method that considers distributed systems explicitly is Laurencot and Castanet [LC97]. Their approach is finite state machine based. It is inconclusive how well this method scales to large real-time systems and how the states in the specification can be mapped to internal states of the system. Nevertheless, it is in our opinion a promising research direction. The fact that this is the only test-case generation method found in the survey that considers distribution explicitly is surprising, since distribution is an inherent aspect of real-time systems.

5 Conclusions

This section summarizes the material presented in this article. First, the state-of-the-art and general impressions of the area are briefly discussed. Second, the contributions from the classification and analysis are highlighted. Third, some future research directions are suggested that would remedy a few of the identified problems.

5.1 Discussion

The area of testing of real-time properties is slowly beginning to get more attention in academia and industry; the reasons for this we believe is that real-time systems are becoming increasingly complex. The static analysis methods and formal verification methods have trouble keeping up with the required flexibility, diversity of applications, immense parallelism, and distribution of the next generation of embedded systems. Hence, practitioners and researchers turn to testing in hope of finding a simpler way to verify overall correctness. Unfortunately, testing cannot provide absolute guarantees of correctness; the number of test cases needed to completely verify even a medium-sized real-time system is too large to even consider. However, if we take testing in consideration when designing systems by avoiding mechanisms that result in unnecessary test complexity, we believe that it is possible to construct test-case generation methods that exploit this to give confidence in any given degree of correctness while not loosing the desired properties of flexible real-time systems.

One may ask if testing is a useful technique for verifying the correctness of real-time systems in safety-critical environments. In such environments, it can be argued that functional and temporal correctness is so important that well-known, time-triggered systems with static off-line scheduling and formal proofs should always be used to guarantee timeliness. It is true that static architectures are more suitable for safety-critical environments. However, we believe there are many application areas of dependable systems where timeliness is important, but where flexibility and performance requirements call for a flexible, event-triggered architecture.

5.2 Contributions

This article presents an analysis of approaches for generating test cases for testing of time constraints. All encountered methods for testing of time constraints seems to be based on some structured or formal specification. Different specification notations give different advantages but also limitations and problems. Methods based on different notations are analyzed and their properties evaluated from the perspective of testing timeliness in distributed event-triggered real-time systems.

A classification of test-case generation methods relating to testing of temporal properties has been presented, complementing the classification of test methods presented by Laprie et al. [Lap94]. The classification is general enough so that all methods that aim to test time constraints should be easy to add as they arise.

The characteristics and contents of different test-cases produced by the different methods are described and evaluated against the requirement of our target model. The conclusion is that none of the methods consider internal states of the execution environment in the

test-cases, and hence, they allow temporal non-determinism. That is, the same test case may result in very different response-times in consecutive executions.

A related contribution is the identification of a need for special test criteria for selecting test cases for testing of real-time systems. None of the encountered selection criteria take advantage of constraints in the execution environment. Instead selection criteria are often based on the properties in the specification notation, e.g., transition coverage.

Furthermore, the methods has been classified according to their degree of automation and time-base, these attributes indicate which of the methods that are most probable of producing test suites with a realistic associated test-effort.

5.3 Future Research Directions

This section summarizes, in our opinion, the most important of the identified problems and proposes future research directions in this area that aim to solve these problems.

5.3.1 A Selective Test-Case Generation Method for Real-Time Systems

A test generation method for real-time systems must take time constraints into consideration, both for testing that input events are handled in a correct way and that the output of a system always is within a specified interval, i.e., timeliness. To verify the latter property, we believe that testing must be conducted on two levels. On a higher level, the tested event scenario must cause the worst-case internal behavior in blocking, synchronization, and arrival time of events. On a lower level the input parameters and data from shared resources must cause the worst-case execution time of transactions.

Our analysis has concluded that existing methods for test-case generation produce either event sequences or test data, but there exist very few methods that can supply both. An interesting problem for research is to combine a specification-based testing technique, where event sequences and their time constraints are generated from a formal specification, with a temporal behavior enforcing testing technique, such as described in section 4.5.

Furthermore, the test cases produced by the method should support test execution under specific internal conditions, e.g., by providing the internal state from where test execution begins (cf. [Nil99]).

5.3.2 A Test Coverage Criterion for Testing of Timeliness

No work encountered in the survey takes advantage of constraints in the execution environment for limiting the number of test cases. We believe that it is possible to eliminate many redundant test cases by taking such constraints into consideration. Hence, a future work is to further refine these ideas into more formal coverage criteria for testing timeliness in real-time systems.

Preferably, this should result in a hierarchy of coverage criteria attaining increasing degrees of test coverage. Such criteria must be evaluated against other approaches on real-life applications in order to determine it the criteria ensures high quality applications.

References

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[Bei90] B. Beizer. *Software Testing Techniques.* Von Nostrand Reinhold, 1990.

[BFM97] V. Braberman, M. Felder, and M. Marré. Testing timing behavior of real-time software. International Software Quality Week, 1997.

[BJ00] B. Bereza-Jarocinski. Automated testing in daily build. Technical Report ISSN 1493-6444, Swedish Engineering Industries, 2000.

[BMA99] R. Birgisson, J. Mellin, and S. F. Andler. Bounds on test effort for event-triggered real-time systems. In *In Proc. 6th Int'l Conference on Real-Time Computing, Systems and Applications (RTCSA'99)*, pages 212–215. Department of Computer Science, University of Skövde, December 1999.

[CL97] D. Clarke and I. Lee. Automatic generation of tests for timing constraints from requirements. In *Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems*, Newport Beach, California, February 1997.

[COG98] R. Cardell-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. *Lecture Notes in Computer Science*, 1486:251–261, 1998.

[CZ91] R. Cleveland and A. E. Zwarico. A theory of testing for real-time. *IEEE Computer Society Press*, pages 110–119, 1991.

[FBK+91] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on software engineering*, 17(6):591–603, june 1991.

[Kon95] O. Koné. Designing tests for time dependent systems. Seoul, South Korea, 1995. IFIP International conference on Computer Communications.

[KZF+91] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schütz. An engineering approach to hard real-time system design. pages 166–188, Milano, Italy, 1991. In proc. of the Third European Software Engineering Conference, ESEC ' 1991.

[Lap94] J. Laprie, editor. *Dependability : Basic Concepts and Terminology.* Springer-Verlag for IFIP WG 10.4, August 1994.

[LC97] P. Laurencot and R Castanet. Integration of time in canonical testers for real-time systems. California, 1997. 1997 Workshop on Object-Oriented Real-time dependable Systems, IEEE Computer Society Press.

[Mel98] J. Mellin. Supporting system level testing of applications by active real-time databases. In Proc. 2nd Int'l Workshop on Active, Real-Time, and Temporal Databases, ARTDB-97, number 1553 in LNCS. Springer-Verlag., 1998.

[MMM95] D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Transactions on Computer Systems*, 4(13):365–398, Nov 1995.

[MP90] S. Morasca and M. Pezze. Using high level petri-nets for testing concurrent and real-time system. *Real-Time Systems : Theory and Applications*, pages 119–131, 1990. Amsterdam North-Holland.

[MW98] F. Müeller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. 1998.

[Nil99] R. Nilsson. Automated test case execution for real-time systems that are constrained for improved testability. Technical Report HS-IDA-EA-99-118, Department of Computer Science, University of Skövde, 1999.

[Nil00] R. Nilsson. Automated selective test case generation methods for real-time systems. Master's thesis, University of Skövde, September 2000.

[PF99a] E. Petitjean and H. Fochal. A realistic architecture for timed testing. In *Proc. of Fifth IEEE International Conference on Engineering of Complex Computer Systems*, USA, Las Vegas, October 1999.

[PF99b] S. M. Petters and G. Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. Hong Kong, 1999. RTCSA99.

[PS91] P. P. Puschner and A. V. Schedl. Computing maximum task execution times - a graph based approach. 1991. Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

[Sch93] W. Schütz. *The Testability of Distributed Real-Time Systems.* Kluwer Academic Publishers, 1993.

[Sch94] W. Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, September 1994.

[SMM00] P. SanPietro, A. Morzenti, and S. Morasca. Generation of execution sequences for modular time critical systems. *IEEE Transactions on Software Engineering*, 26(2):128–149, feb 2000.

[TH99] H. Thane and H. Hansson. Towards deterministic testing of distributed real-time systems. Swedish National Real-Time Conference SNART'99, August 1999.

[WSJE97] J. Wegener, H. H. StHammer, B. F. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.

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. 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 preemptive scheduling of periodically arriving tasks on m equally powerful processors. We compare the performance of the best algorithms of the partitioned and non-partitioned method, from two different aspects. First, an average-case comparison, using an idealized architecture, shows that, if a system has a small number of processors, then the non-partitioned method offers higher performance than the partitioned method. Second, an average-case comparison, using a realistic architecture, shows that, for several combinations of preemption and migration costs, the non-partitioned method offers higher performance.

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.

Fixed-priority scheduling of tasks on such systems is typically solved using one of two different methods based on how tasks are assigned to the processors at run-time. In the *partitioned* method, all instances of a task are executed on the same processor. The processor used for the execution of a task is determined before run-time by a partitioning algorithm. In the *non-partitioned* method, a task is allowed to execute on any processor, even when resuming after having been preempted. For the partitioned method, a plethora of mature algorithms and analyses have been proposed, something that has contributed to the widespread use of the method in multiprocessor-based real-time systems. In contrast, the non-partitioned method has received much less attention, mainly because it is believed to suffer from scheduling- and implementation-related shortcomings, but also because it lacks support for more advanced system models, such as the management of shared resources.

Many operating systems for shared-memory multiprocessors support fixed-priority scheduling with both the partitioned and the non-partitioned method. In order to know which of the methods should be used in practice, it is important to know their ability to schedule an arbitrary task set on a fixed number of processors. So far, the only previous evaluation of fixed-priority preemptive multiprocessor scheduling claims that the average-case performance of the non-partitioned method is lower than the average-case performance of the partitioned method [2]. It has also been shown that the worst-case performance of the non-partitioned method using the rate-monotonic priority scheme is poor [3]. We believe that these comparisons are not complete, because (i) recently a novel priority as-

signment for the non-partitioned method [4] was proposed which alleviates the scheduling-related shortcoming, and (ii) the previous evaluation considered the cost of preemption and migration to be zero. It is not clear how preemption and migration costs will affect performance of the partitioned and non-partitioned methods.

In this paper, we compare the novel non-partitioned priority-assignment scheme against the best partitioning algorithms. We demonstrate that the partitioned method is *not necessarily the best approach*. To this end, we make two main research contributions.

C1. We show that, on an idealized architecture, the non-partitioned method offers higher average-case performance than the partitioned method. We also show that, even if a necessary and sufficient schedulability test is used for the partitioning algorithms, then the best partitioned method performs only slightly better than the non-partitioned method.

C2. We show that, on a realistic architecture, the non-partitioned method can provide higher performance than the partitioned method when both an improved dispatcher is used and the additional cost of a migration is no greater than the cost of a preemption.

The rest of this paper is organized as follows. In Section 2, we review previous work in fixed-priority preemptive multiprocessor scheduling and define concepts and system models used. Section 3 shows the average-case behavior of the partitioned and non-partitioned method, and in Section 4, we discuss and show the architectural impact on the average-case performance. We discuss other aspects of the scheduling problem in Section 5, and summarize our results in Section 6.

2 Background

Recall that there are two methods that are used to solve the multiprocessor scheduling problem, namely the partitioned and the non-partitioned method[1]. In this section, we first discuss in more detail the capabilities and problems of these methods, and then define concepts and system models used in this paper.

2.1 Previous work

For fixed-priority preemptive scheduling of periodically-arriving tasks on a multiprocessor system, both the partitioned and the non-partitioned method have been addressed in previous research. Important properties were presented in a seminal paper by Leung and Whitehead [5]. In particular, they showed that the problem of deciding whether 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 no method dominates the other in the sense that there are task sets which are schedulable with an optimal priority assignment with the non-partitioned method, but are unschedulable with an optimal partitioning algorithm and conversely.

Among the two methods, the partitioned method has received the most attention in the research literature. The main reason for this is 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 partitions, each having its own dedicated processor. Since an optimal solution to the problem of partitioning the tasks is believed to be computationally intractable, many heuristics for partitioning have been proposed (see for example [3, 6, 7, 8, 9, 10]). 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 following limitations. First, no efficient schedulability tests currently exist for the non-partitioned method. The only known necessary and sufficient schedulability test for the non-partitioned method has an exponential time-complexity [11]. The complexity can be reduced with sufficient schedulability tests to a polynomial [12, 2, 13, 14] or pseudo-polynomial [13] time complexity. However, the test in [14] becomes pessimistic when the number of tasks increases, and the tests in [12, 2, 13] become pessimistic when the number of processors increases. Second, no efficient optimal priority-assignment scheme has been found for the non-partitioned method. The rate-monotonic priority assignment (RM) [15], which is optimal on a uniprocessor, is not optimal for multiprocessors using the non-partitioned method [3, 5]. Even worse, task sets with a very low utilization can be unschedulable with RM [3]. We refer to the latter as *Dhall's effect*.

In a recent comparison of the partitioned and non-partitioned method, it was claimed that, even if one is willing to use a necessary and sufficient schedulability test for the non-partitioned method, the non-partitioned rate-monotonic is inferior to the partitioning algorithms [2, Section 3.5]. We believe that this comparison is not complete for two reasons. First, the previous evaluation did not consider a recently-proposed priority assignment scheme, *adaptiveTkC* [4], in which the highest priority is assigned to the task τ_i which has the least $T_i - k * C_i$, where $k = \frac{1}{2} \cdot \frac{m-1+\sqrt{5\,m^2-6\,m+1}}{m}$ (m is the number of processors). As shown in [4], the value of k is selected to counter two effects that occur at $k = 0$ and $k \to \infty$. The case

[1] Some authors refer to the non-partitioned method as "dynamic binding" or "global scheduling".

where $k = 0$ is equivalent to RM, which is known to suffer from Dhall's effect. The other case, $k \to \infty$, gives highest priority to the task with the longest execution time, which can make task sets unschedulable at arbitrary low utilization. The second reason why the previous evaluation is not complete is that it considered the cost of preemption and migration to be zero. It is not clear how preemption and migration costs will affect performance of the partitioned and non-partitioned methods. Because of these shortcomings, we believe that it is necessary to take a new look at the question of whether to partition or not to partition.

2.2 Concepts and System model

We consider the problem of scheduling a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n independent[2], periodically-arriving real-time tasks on m identical processors. A task arrives periodically with a period of T_i. Each time a task arrives, a new *instance* of the task is created. Each instance has 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 *meta period* of the task set is the least common multiple of $T_1, T_2, \ldots, T_n$.

For the partitioned method the system behaves as follows. Each task is assigned to a processor, and then assigned a local (for the processor), unique and fixed priority. With no loss of generality, we assume that the tasks on each processor are numbered in the order of decreasing priority, that is, τ_1 has the highest priority. On each processor, the task with the highest priority of those tasks which has arrived, but not completed, is executed.

For the non-partitioned method the system behaves as follows. Each task is assigned a global, unique and fixed priority. With no loss of generality, we assume that the tasks in τ are numbered in the order of decreasing priority, that is, τ_1 has the highest priority. Of all tasks that have arrived, but not completed, the m highest-priority tasks are executed[3] in parallel on the m processors.

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, $U = \sum_{i=1}^{n} C_i/T_i$. A task is *schedulable* if all its instances completes no later than their deadlines. A task set is schedulable if all its tasks are schedulable.

To understand the basic performance characteristics, we initially assume the following simple system model.

A1. Tasks are independent, arrive periodically, and can always be preempted. Hence, at every moment, a dispatcher determines which task to execute. In practice, some operating systems use tick driven scheduling, where the ready queue is only inspected at certain times.

A2. Tasks do not require exclusive access to any other resource than a processor.

A3. 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 (that is, the cost of migration is also assumed to be zero).

A4. The cost when a task arrives is zero because we consider cache misses that occur when a task arrives to be included in the execution time. We will use this assumption even if a task executes after arrival on a processor on which it has never executed. These assumptions should be reasonable since worst-case execution time analysis tools typically consider uninterrupted execution of a task that starts in an "empty state" (e.g., all cache lines are empty).

To investigate more realistic characteristics of the system, we will relax A3 later in this paper (in Section 4).

3 Average-Case Behavior

In this section, we will conduct an average-case performance evaluation of the non-partitioned and partitioned methods, assuming an idealized architecture with no overhead of task preemption or migration. Our evaluation methodology, which will be described in Section 3.1, is based on simulation experiments using randomly-generated task sets. The rationale for using simulation of synthetic task sets is that it more easily reveals the average-case performance and robustness of a scheduling algorithm than can be achieved by scheduling a single application benchmark. Section 3.2 presents the results from the simulations.

3.1 Experimental setup

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 $m = 4$ processors. The number of tasks, n, follows a uniform distribution with an expected value of $E[n] = 8$, a minimum of $0.5\,E[n]$, and a maximum of $1.5\,E[n]$. The period, T_i, of a task τ_i is taken from a set $\{100, 200, 300, 400, 500, \ldots, 1600\}$, 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 the execution time is rounded down to the next lower integer. If the execution time becomes zero, then the task is generated again.

[2] That is, there are no precedence constraints on the arrival time of the tasks.

[3] At each instant, the processor chosen for each of the m tasks is arbitrary. If less than m tasks should be executed simultaneously, some processors will be idle.

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 one meta period[4]. Scheduling decisions are only taken when a task arrives or completes.

We use *success ratio* as the performance measure for average-case performance. The success ratio is the fraction of all generated task sets that are successfully scheduled with respect to an algorithm. The success ratio is computed for each point in a plot as an average of 2,000,000 task sets[5]. In the evaluation of the partitioned method, we consider a task set as successfully scheduled if and only if the number of processors required according to the bin-packing algorithm is no greater than m. In the evaluation of the non-partitioned method, we consider a task set as successfully scheduled if the task set is schedulable, that is all task instances in the task set simulated during a meta period completed no later than their deadlines.

Two non-partitioned priority-assignment schemes are evaluated, namely RM [15] and adaptiveTkC [4]. Two bin-packing-based partitioning algorithms are studied, namely RM-FFDU [8], and R-BOUND-MP [9]. The reason for selecting these two latter algorithms is that we have found that they are the partitioning algorithms which provide the best performance in our experimental environment (other algorithms, such as RRM-BF [7] and RMGT [6] offer lower performance). For all partitioning algorithms, we use the corresponding sufficient schedulability test.

We have also evaluated a hybrid partitioned/non-partitioned algorithm, which we will call *RM-FFDU+adaptiveTkC*. The reason for considering a hybrid solution is that we may be able to increase processor utilization with the use of non-partitioned tasks, without jeopardizing the guarantees given to partitioned tasks. The RM-FFDU+adaptiveTkC scheme operates in the following manner. First, as many tasks as possible are partitioned with RM-FFDU on the given number of processors, and are given local priorities. Then, the remaining tasks (if any) are assigned global priorities according to the adaptiveTkC priority-assignment scheme. Each processor has a local ready queue for the partitioned tasks and there is a global ready queue for the non-partitioned tasks. A processor executes a task from its local ready queue, if the local ready queue of that processor is non-empty. A processor executes a task from the global ready queue, if the local ready queue of that processor is empty.

[4]The reason for scheduling during a meta period is because for the non-partitioned method, the response time of a task is not necessarily maximized when a task arrives at the same time as its higher priority tasks [4]. We therefor select small values of $E[n]$ to avoid that the meta period grows too large, causing simulations to take too long time. We also select small values of m since m must be less than n to make the scheduling problem non-trivial.

[5]With 95% confidence, we obtain an error of the success ratio that is less than 0.1%.

3.2 Performance comparison

Figure 1 shows the results of the simulation experiment. We make three observations.

First, RM offers the worst performance. However, it is not as bad as suggested by previous studies [3]. The reason for this is of course that Dhall's effect, although it exists, does not occur frequently. This observation also corroborates a recent study [2].

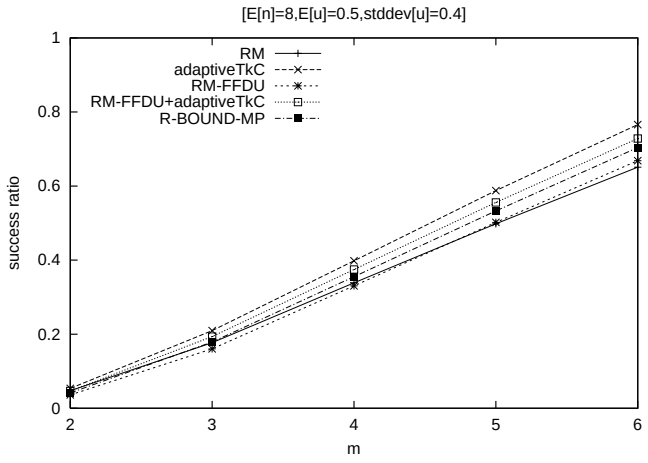
Second, adaptiveTkC offers the best performance, no matter how we vary the parameters (the number of processors, the expected value of the number of tasks, the expected value of task utilization and the standard deviation of the task utilization). The reason for this is that adaptiveTkC takes advantage of the salient property of the non-partitioned method, namely the ability to schedule tasks in slots of unused time on different processors (as RM does).

Third, the hybrid partitioned/non-partitioned algorithm consistently outperforms the corresponding partitioning algorithms. This indicates that such a hybrid scheme is a viable alternative to use in multiprocessor systems that mixes real-time tasks of different criticality. The reason for the good performance is that the hybrid partitioned/non-partitioned algorithm can schedule all task sets that the corresponding partitioned method can schedule, but the hybrid partitioned/non-partitioned algorithm can also schedule some tasks in slots of unused time on different processors.

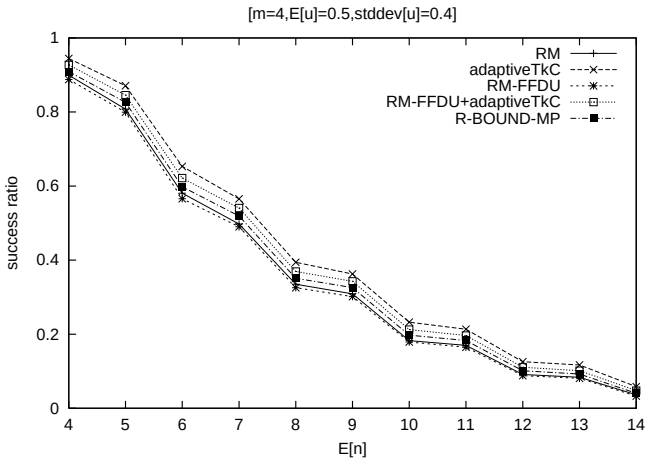
For all partitioning algorithms, sufficient schedulability tests were originally proposed to be used. Now, if we instead use a necessary and sufficient schedulability test based on response-time analysis [16] for the partitioning algorithms, the success ratio can be expected to increase, albeit at the 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. Figure 2 shows the simulation results when the partitioning algorithms use response-time analysis. Here, we make the following observation. RM-FFDU with response-time analysis only provides a slightly higher success ratio than adaptiveTkC while other partitioning algorithms still have a lower success ratio than adaptiveTkC. Note that this means that, even if the best partitioning algorithm (R-BOUND-MP) use response-time analysis, it still performs worse than adaptiveTkC. This should not come as a surprise, since for R-BOUND-MP, the performance bottleneck is the partitioning and not the schedulability test [9].

4 Architectural Impact

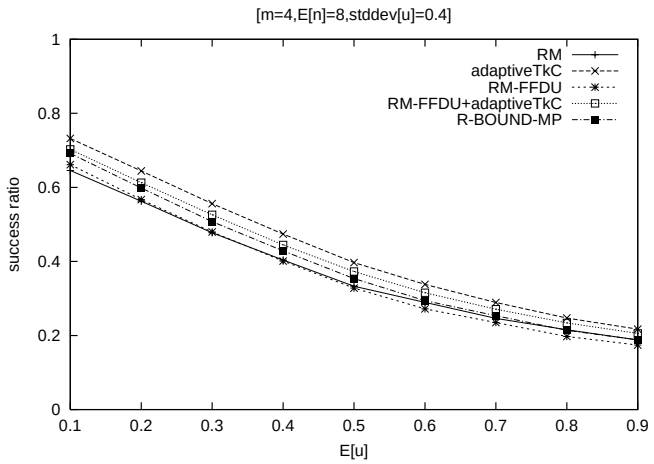
From the results in Section 3, we observed that adaptiveTkC performed well under the assumption of an ideal-



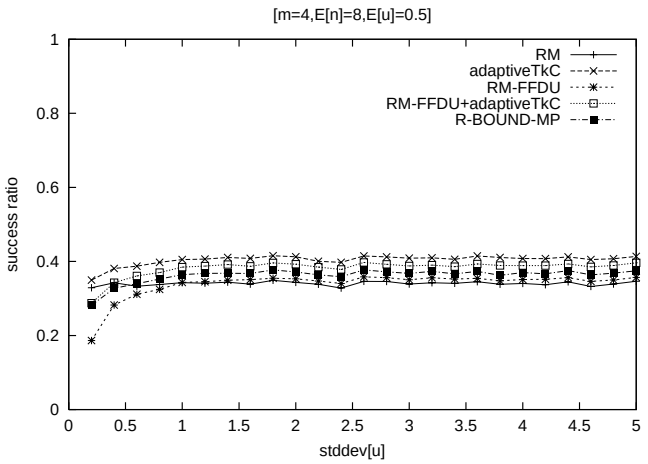
(a) Success ratio as a function of the number of processors.



(b) Success ratio as a function of the number of tasks.



(c) Success ratio as a function of the expected value of utilization of tasks.



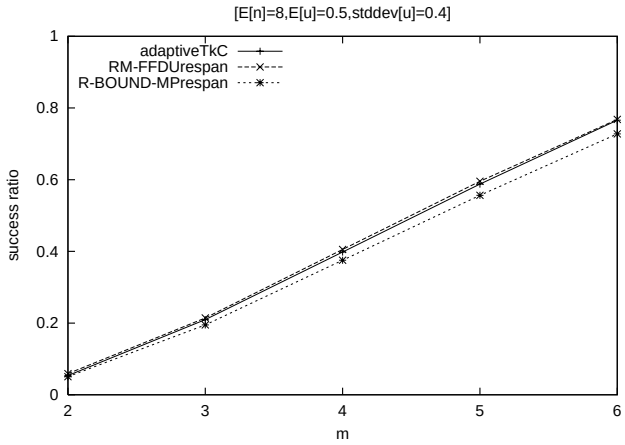
(d) Success ratio as a function of the standard deviation of utilization of tasks.

Figure 1: Success ratio for different scheduling algorithms when the partitioning algorithms use a sufficient schedulability test (polynomial time complexity).

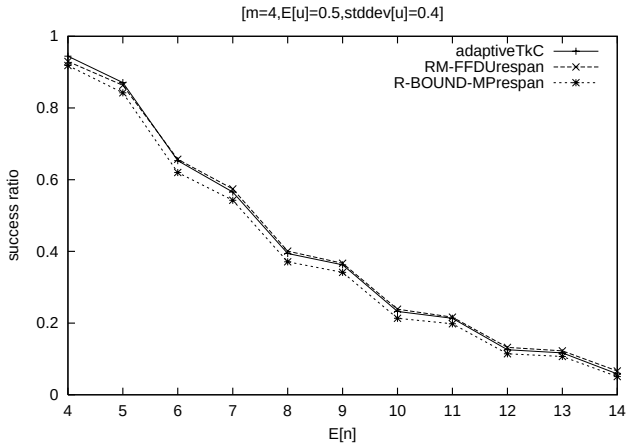
ized architecture. Now, in order to evaluate the performance of a realistic system we must incorporate some architectural costs. In this section, we will study the impact of two architectural costs, namely *preemption* and *migration*.

It is tempting to believe that the non-partitioned method causes a larger amount of (and more costly) preemptions[6] than the partitioned method, and that this makes more task sets schedulable with the partitioned method than with the non-partitioned method. We will now show that such statements cannot easily be made. First, we will propose a dispatcher for the non-partitioned method that reduces the number of preemptions by analyzing the current state of the schedule. We will then show that the average number of preemptions generated by the best non-partitioned method using the new dispatcher is significantly less than that of the best partitioning algorithm. Finally, we will evaluate the schedulability of the non-partitioned and partitioned method when the costs of preemption and migration are accounted for.

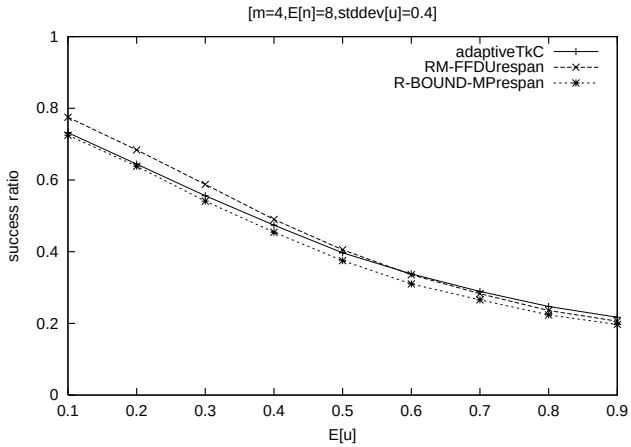
[6] A task is preempted if it has remaining execution time but does not immediately continue to execute on the same processor.



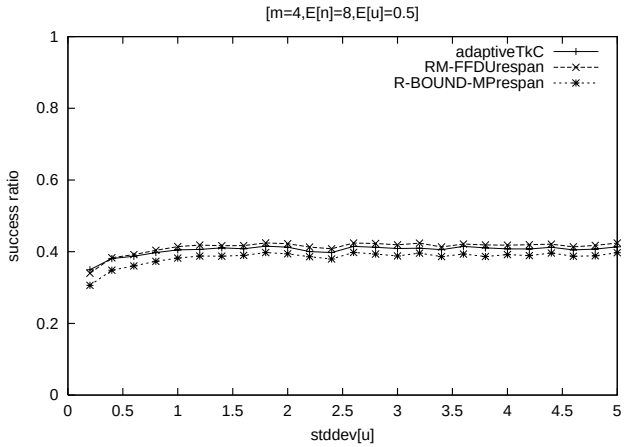
(a) Success ratio as a function of the number of processors.



(b) Success ratio as a function of the number of tasks.



(c) Success ratio as a function of the expected value of utilization of tasks.



(d) Success ratio as a function of the standard deviation of utilization of tasks.

Figure 2: Success ratio for different scheduling algorithms when the partitioning algorithms use a necessary and sufficient schedulability test (pseudo-polynomial time complexity).

4.1 Improved Dispatcher

Recall that the non-partitioned method using fixed-priority preemptive scheduling only requires 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. As long as the cost of a preemption is zero, the task-to-processor assignment at each instant does not affect schedulability. However, on real computers the time required for a preemption is non-negligible. If the task-to-processor assignment is selected arbitrarily, it could happen 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 same ones that executed last. With the original dispatcher, this would have serious consequences for the schedulability of the system. Hence, to reduce the risk of unnecessary preemptions, 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 preemptions is minimized.

We now propose a heuristic for the task-to-processor assignment that will reduce the number of preemptions for the non-partitioned method. The basic idea of the task-to-processor assignment algorithm is to determine which of the

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 selected for execution.

Output: On each processor p_i, a (possibly non-existing) task $\tau_{ij,after}$ should execute.

1: $E = \{p_i : (\tau_{ij,before} \neq \text{non} - \text{existing}) \wedge (\tau_{ij,before} \in \tau_{highest})\}$
2: for each $p_i \in E$
3: remove $\tau_{ij,before}$ from $\tau_{highest}$
4: $\tau_{ij,after} \leftarrow \tau_{ij,before}$
5: for each $p_i \notin E$
6: *if* $\tau_{highest} \neq \emptyset$
7: select an arbitrary τ_j from $\tau_{highest}$
8: remove τ_j from $\tau_{highest}$
9: $\tau_{ij,after} \leftarrow \tau_j$
10: *else*
11: $\tau_{ij,after} \leftarrow \text{non} - \text{existing}$

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 in their previous execution. The algorithm for this is described in Algorithm 1. In the remainder of this paper, we will refer to this dispatcher as the *preemption-aware* dispatcher. We let *adaptiveTkCunaware* denote adaptiveTkC together with a dispatcher where the task with the highest priority is assigned to the processor with the least index, unaware of the preemptions it will generate. Correspondingly, we let *adaptiveTkCaware* denote adaptiveTkC together with the preemption-aware dispatcher. In the remainder of this section, we will assume that, whenever the non-partitioned method is used, adaptiveTkCaware is used. AdaptiveTkCunaware will only be used as a baseline algorithm.

4.2 Number of Preemptions

To demonstrate that the preemption-aware dispatcher is effective, we will start by showing its performance in terms of the number of preemptions generated. To this end, it would be natural to simulate scheduling and count the total number of preemptions generated during a meta period and use that number as a measure of the number of preemptions. However, if many task sets are simulated, and we take the sum of the preemptions in all task sets, then task sets with a large meta period would be biased in that they would have a higher impact on the total number of preemptions. To make each task set equally important, we have therefore chosen to use *preemption density* as the measure of the number of preemptions. Preemption density of a scheduled task set is defined as the number of preemptions generated during a meta period divided by the length of the meta period.

To reveal which of the two methods (partitioned or non-partitioned) that generates the highest number of preemptions, we have simulated scheduling of randomly-generated task sets and computed the preemption density for each of these task sets. Since, in this subsection, we are only interested in the number of preemptions — not their impact on schedulability — we assume that the cost of preemption and the cost of migration is zero. We simulate scheduling using adaptiveTkCaware, adaptiveTkCunaware and R-BOUND-MP because they are the best (as demonstrated in Section 3.2) schemes for the non-partitioned method and the partitioned method, respectively. We use the same experimental setup as described in Section 3.1. We varied the number of tasks and simulated 5,000 task sets for each point in a plot. We then computed the preemption density only for those task sets for which both adaptiveTkCaware, adaptiveTkCunaware and R-BOUND-MP were schedulable.

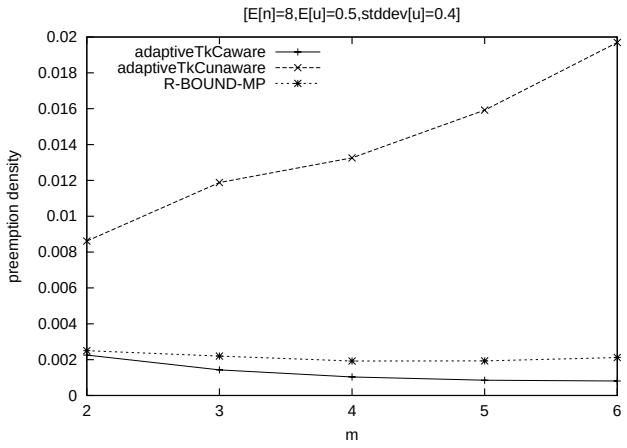
The results from the simulations are shown in Figure 3. We observe that, on average, the preemption density of the best non-partitioned method (adaptiveTkCaware) is lower than the preemption density of the best partitioned method (R-BOUND-MP). The reason for this is that, for the partitioned method, an arriving higher-priority task must always preempt an executing lower-priority task on the same processor. With the non-partitioned method, a higher priority task can sometimes execute on another idle processor, thereby avoiding a preemption. Figure 4 illustrates a situation where the non-partitioned method with the preemption-aware dispatcher causes the number of preemptions to be less than the number of preemptions of a partitioned method.

Note that, in Figure 3(a), increasing the number of processors gives a different effect on the preemption density for adaptiveTkCaware than for adaptiveTkCunaware. It can be explained as follows. When the number of processors increases, there will be more tasks executing in parallel and it is more likely that, during a time interval, a task completes its execution. Consider which processor a low-priority task executes on when a higher-priority task completes it execution. For adaptiveTkCaware, the lower-priority task will continue to execute on the same processor as it did before the higher priority task completed. Hence no preemption occurs. However, for adaptiveTkCunaware (as defined in Section 4.1), the lower priority task will continue its execution on another processor with a lower index. That is, with our definition, a preemption.

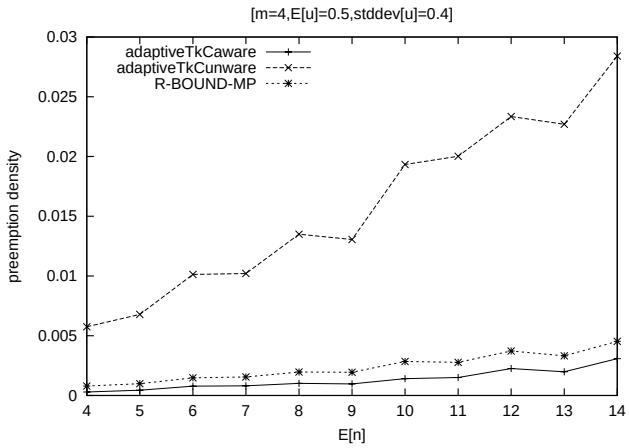
4.3 Average-Case Behavior

With the new dispatcher at hand, we are now in position to assess the schedulability of a system with non-negligible preemption and migration cost. We will model the preemption cost and migration cost of a uniform-memory access shared-memory multiprocessor with caches.

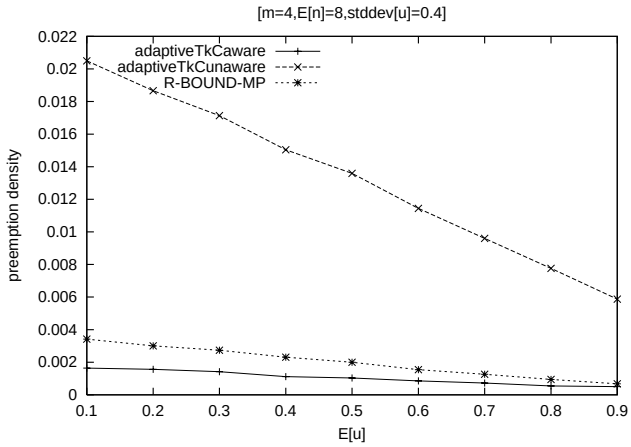
For the assumed system, the cost of a preemption includes the following terms: time to acquire the ready queue, time to save and restore a task's environment (e.g., registers,



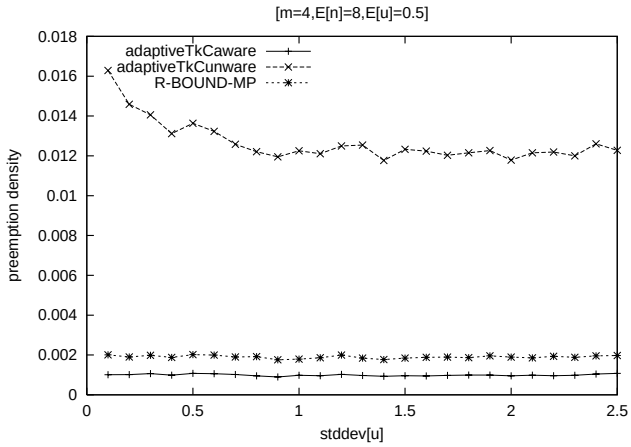
(a) Preemption density as a function of the number of processors.



(b) Preemption density as a function of the number of tasks.

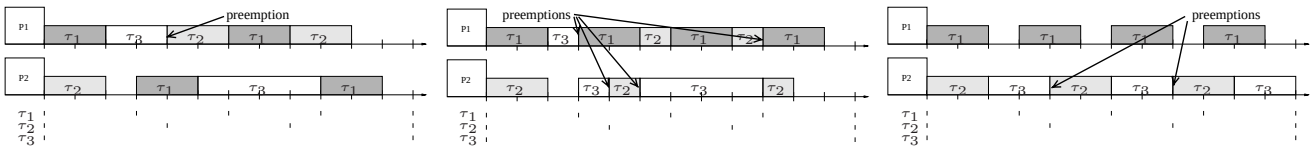


(c) Preemption density as a function of the expected value of utilization of tasks.



(d) Preemption density as a function of the standard deviation of utilization of tasks.

Figure 3: Preemption density of different scheduling algorithms.



(a) AdaptiveTkCaware (b) AdaptiveTkCunaware (c) R-BOUND-MP

Figure 4: Preemptions for the non-partitioned method and partitioned method when scheduling the task set $\{(T_1 = 3, C_1 = 2), (T_2 = 4, C_2 = 2), (T_3 = 12, C_3 = 6)\}$ on $m = 2$ processors. The non-partitioned method, with adaptiveTkCaware, generates the least number of preemptions.

memory mapping) and the cost of cache misses due to cache reloading when the task resumes. In the evaluation in this section, we will assume that the time to acquire the ready queue and time to save and restore a task's state is zero. We do this for three reasons, namely (i) these costs depend heavily on the implementation (hardware/software) and mechanism to protect the ready queue (fine-grained/course-grained locking/wait-free/lock-free), (ii) it has been found that the time to execute kernel code constitute only a fraction of the total cost of a context switch [17], and (iii) the trend of faster processors and (relatively) slower memories will make the cache reload effect an even more dominant term in the cost of preemption.

Our simulated system model is now changed as follows. When a task is resumed, we add a cost of $preempt.ratio \cdot E[u] \cdot E[T]$ to its remaining execution time because of cache reloading. If a task is resumed on another processor than it was preempted on we add yet another cost of $mig.ratio \cdot E[u] \cdot E[T]$ to its remaining execution time because of cache reloading due to the migration. We use the same experimental setup as described in Section 3.1, but simulated 5,000 task sets for each possible combination of preemption ratio, migration ratio, and the number of processors.

The results from our simulations are shown in Figure 5. We observe that, when the migration and preemption costs are high, adaptiveTkCaware has higher success ratio than adaptiveTkCunaware. This indicates that the preemption-aware dispatcher is effective. We can also see that the partitioned method becomes more favorable when the migration ratio is large, because no migrations can occur for that method. On the other hand, the non-partitioned method becomes more favorable when the preemption ratio increases, because the preemption-aware dispatcher can reduce the number of preemptions. However, when there are only two processors available (see Figure 5(a)), the preemption-aware dispatcher becomes less effective, because it has only one alternative processor to try to schedule a task on to reduce the number of preemptions.

5 Discussion

From the previous sections, we have learned that the relative performance of the partitioned and non-partitioned method varies depending on the system models and performance measures used. However, the decision whether to partition or not to partition may also need to consider other aspects than comparing success ratio. Below we list some interesting aspects.

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 [2]. 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 execution 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 hence offers one solution for exploiting processors efficiently, 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 alternative approach would be to partition only the tasks that have hard real-time constraints, and then let the tasks with soft constraints be scheduled by the non-partitioned method.

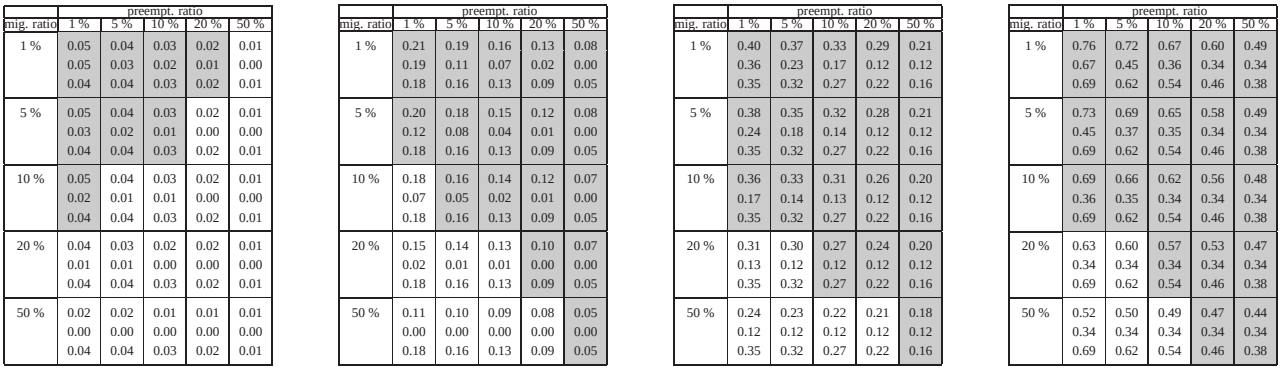
The partitioned method has the advantage of having a low computational complexity of the schedulability test, something that is important for, e.g., online admission tests and QoS negotiation [18]. There is also a complexity associated with the algorithm that reschedules a task set that has changed. The non-partitioned method has lower computational complexity for rescheduling than the partitioned method if the partitioned method use the response-time analysis. However, even if sufficient schedulability tests are used for the partitioned method, the non-partitioned method has a slightly lower computational complexity of rescheduling ($O(n \log n)$ for adaptiveTkC) than the partitioned method ($O(nm + n \log n)$ for R-BOUND-MP and $O(nm+n \log n)$ for RM-FFDU). For systems that reschedule frequently, but do not rely on schedulability tests, e.g., feedback control scheduling [19], the non-partitioned method, with its slightly higher success ratio and its slightly lower computational complexity of rescheduling, becomes a viable alternative. Also, since these systems are likely to be used when the execution time varies, the advantage of reclaiming resources in the non-partitioned method gives a further performance increase.

6 Conclusions

It has been believed that the non-partitioned method is inferior to the partitioned method. In this paper, we have shown through two evaluations that by using a better priority assignment scheme and an improved dispatcher, such a belief is not necessarily true. First, an average-case comparison using an idealized architecture, showed that, if a system has a small number of processors, then the non-partitioned method offers higher performance than the partitioned method. Second, an average-case comparison using a realistic architecture, showed that, for several combinations of preemption and migration costs, the non-partitioned method offers higher performance.

Acknowledgement

This work is funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Strategic Research.



mig. ratio	preempt. ratio 1 %	5 %	10 %	20 %	50 %
1 %	0.05	0.04	0.03	0.02	0.01
	0.05	0.03	0.02	0.01	0.00
	0.04	0.04	0.03	0.02	0.01
5 %	0.05	0.04	0.03	0.02	0.01
	0.03	0.02	0.01	0.00	0.00
	0.04	0.04	0.03	0.02	0.01
10 %	0.05	0.04	0.03	0.02	0.01
	0.02	0.01	0.01	0.00	0.00
	0.04	0.04	0.03	0.02	0.01
20 %	0.04	0.03	0.02	0.02	0.01
	0.01	0.01	0.00	0.00	0.00
	0.04	0.04	0.03	0.02	0.01
50 %	0.02	0.02	0.01	0.01	0.01
	0.00	0.00	0.00	0.00	0.00
	0.04	0.04	0.03	0.02	0.01

(a) $m = 2$

mig. ratio	preempt. ratio 1 %	5 %	10 %	20 %	50 %
1 %	0.21	0.19	0.16	0.13	0.08
	0.19	0.11	0.07	0.02	0.00
	0.18	0.16	0.13	0.09	0.05
5 %	0.20	0.18	0.15	0.12	0.08
	0.12	0.08	0.04	0.01	0.00
	0.18	0.16	0.13	0.09	0.05
10 %	0.18	0.16	0.14	0.12	0.07
	0.07	0.05	0.02	0.01	0.00
	0.18	0.16	0.13	0.09	0.05
20 %	0.15	0.14	0.13	0.10	0.07
	0.02	0.01	0.01	0.00	0.00
	0.18	0.16	0.13	0.09	0.05
50 %	0.11	0.10	0.09	0.08	0.05
	0.00	0.00	0.00	0.00	0.00
	0.18	0.16	0.13	0.09	0.05

(b) $m = 3$

mig. ratio	preempt. ratio 1 %	5 %	10 %	20 %	50 %
1 %	0.40	0.37	0.33	0.29	0.21
	0.36	0.23	0.17	0.12	0.12
	0.35	0.32	0.27	0.22	0.16
5 %	0.38	0.35	0.32	0.28	0.21
	0.24	0.18	0.14	0.12	0.12
	0.35	0.32	0.27	0.22	0.16
10 %	0.36	0.33	0.31	0.26	0.20
	0.17	0.14	0.13	0.12	0.12
	0.35	0.32	0.27	0.22	0.16
20 %	0.31	0.30	0.27	0.24	0.20
	0.13	0.12	0.12	0.12	0.12
	0.35	0.32	0.27	0.22	0.16
50 %	0.24	0.23	0.22	0.21	0.18
	0.12	0.12	0.12	0.12	0.12
	0.35	0.32	0.27	0.22	0.16

(c) $m = 4$

mig. ratio	preempt. ratio 1 %	5 %	10 %	20 %	50 %
1 %	0.76	0.72	0.67	0.60	0.49
	0.67	0.45	0.36	0.34	0.34
	0.69	0.62	0.54	0.46	0.38
5 %	0.73	0.69	0.65	0.58	0.49
	0.45	0.37	0.35	0.34	0.34
	0.69	0.62	0.54	0.46	0.38
10 %	0.69	0.66	0.62	0.56	0.48
	0.36	0.35	0.34	0.34	0.34
	0.69	0.62	0.54	0.46	0.38
20 %	0.63	0.60	0.57	0.53	0.47
	0.34	0.34	0.34	0.34	0.34
	0.69	0.62	0.54	0.46	0.38
50 %	0.52	0.50	0.49	0.47	0.44
	0.34	0.34	0.34	0.34	0.34
	0.69	0.62	0.54	0.46	0.38

(d) $m = 6$

Figure 5: Success ratio for scheduling algorithms as a function of the number of processors, preemption costs and migration costs. Each square shows adaptiveTkCaware (upper), adaptiveTkCunaware (middle), R-BOUND-MP (lower). The shaded regions indicate where the non-partitioned method (with adaptiveTkCaware) performs better than the partitioned method (R-BOUND-MP).

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. 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.

[3] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, January/February 1978.

[4] B. Andersson and J. Jonsson. Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling. In *Proc. of the IEEE Real-Time Systems Symposium – Work-in-Progress Session*, Orlando, Florida, November 27–30, 2000. Also in TR-00-10, Dept. of Computer Engineering, Chalmers University of Technology.

[5] 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.

[6] 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.

[7] Y. Oh and S. H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Systems*, 9(3):207–239, November 1995.

[8] 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.

[9] 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.

[10] 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.

[11] J. Y.-T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(2):209–219, 1989.

[12] 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.

[13] 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.

[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] 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.

[16] M. Joseph and P. Pandya. Finding response times in a real-time system. *Computer Journal*, 29(5):390–395, October 1986.

[17] V. Padmanabhan and D. Roselli. The real cost of context switching. Technical report, CS Division, University of California at Berkeley, November 1994.

[18] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin. QoS negotiation in real-time systems and its application to automated flight control. In *Proc. of the IEEE Real-Time Technology and Applications Symposium*, pages 228–238, Montreal, Canada, June 9–11, 1997.

[19] 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, 1999.

Solving Embedded System Scheduling Problems using Constraint Programming

Cecilia Ekelin and Jan Jonsson

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

Abstract

Static scheduling of tasks in embedded distributed real-time systems often implies a tediuos iterative design process. The reason for this is the lack of flexibility and expressive power in existing scheduling frameworks, which makes it difficult to both model the system accurately and provide correct optimization guidelines in a first attempt. The most detrimental effect this has on the scheduling process is that the real-time tasks are over-constrained due to a careless use of rules-of-thumb.

We have developed a scheduling framework based on constraint programming which attempts to tackle these deficiencies. First, the framework allows for expressing the scheduling problem in terms much closer to the actual system requirements. This is because of the support provided by the framework for modeling constraints that are usually not handled by existing scheduling tools, for example, different types of task allocation constraints such as clustering and proximity to resources. A second strong feature of the framework is that it supports different strategies for single- and multi-objective optimization, something that is very important in the design of embedded real-time systems. An evaluation study encompassing a set of applications with typical embedded system constraints suggests that our approach does in fact provide the salient features necessary.

1 Introduction

Real-time system design is becoming increasingly dependent on flexible scheduling frameworks to cope with the size and complexity of contemporary applications. This is most apparent in the design of embedded distributed real-time systems where the scheduling framework must possess two salient features, namely *(a) support for a large variety of application constraints* to handle scheduling of distributed tasks and dedicated resources, and *(b) support for advanced optimization capabilities* to concurrently account for system requirements on predictability, reliability, and hardware performance, as well as various aspects of total system cost. As a basis for constructing a scheduling framework, real-time literature proposes several scheduling algorithms that present solutions to different aspects of the scheduling problem. Unfortunately, most existing scheduling algorithms only solve stand-alone problems (for example, focusing only on uniprocessor systems, one type of task model, and a single optimization criterion). This makes construction of a scheduling framework particularly difficult because there may exist a discrepancy between the theoretical scheduling problem and the practical aspects of the real-time system being designed, which gives rise to two major problems.

First, there is the problem of using constraints that are as close as possible to the original system requirements. Since many scheduling algorithms are defined with "hard-coded" assumptions regarding task and system models, it is easy to believe that it suffices to devise constraints that fit the scheduling algorithm. However, such an approach would modify the semantics of the original system requirements, and typically lead to the introduction of *artificial* constraints without any direct connection to the original scheduling problem [1, 2]. Such design approaches are very error-prone since the *ad hoc* constraint construction often leads to an over-constrained and infeasible system. This turns scheduling into a tedious design process since the designer must suggest suitable changes in the specification, implementation and/or system models.

Hence, instead of adapting the problem to the scheduling algorithm, the scheduling algorithm should be adapted to the problem in the sense that it should be capable of accounting for as many *natural* and *implementation-based* constraints as possible.

Second, even if a feasible schedule is found by the scheduling framework, the suggested solution may still be considered invalid by the system designer since requirements like cost and hardware resources usually cannot be accounted for in most scheduling algorithms. This makes it necessary for the designer to manually verify a schedule with respect to these requirements, and suggest suitable design changes. Hence, a scheduling framework with a notion of multi-objective optimization would shorten the process cycle. Moreover, it would increase the ability for real-time system developers to provide cost-effective solutions. This is particularly useful as the trend in embedded real-time system design is moving towards open systems [3]. Apart from making the scheduling process more effective, we also believe that by using constraints that are semantically closer to the requirements, the scheduling search space can be reduced. The rationale for this is that, when constraints are more close to the original requirements, the scheduling algorithm can operate in a more intelligent way since it has more knowledge of the problem (see, for example, recent results for branch-and-bound algorithms [4, 5]).

In this paper, we propose a scheduling framework that possesses all the salient features discussed above. While many other proposed scheduling approaches for distributed real-time systems (for example, simulated annealing or branch-and-bound) have similar features, it is well-known that they require a large amount of fine-tuning to perform well. In contrast, our framework is based on the concept of *constraint programming*, which means that constraints can be introduced in the framework in a flexible and natural way, and that the optimization features do not require tedious manual interaction.

Organization of this paper: This paper addresses how to build a scheduling framework based on constraint programming. The rest of our presentation is organized as follows. In Section 2, we describe typical requirements for embedded distributed real-time systems, and recollect methods for automated constraint derivation and scheduling of distributed systems. In Section 3, we summarize a set of constraints typically found in embedded distributed real-time system design. We present the underlying practical motivation for these constraints, and identify problems in using the constraints with existing scheduling algorithms. In Section 4, we present different generic strategies for single- and multi-objective optimization. We demonstrate how these techniques can be applied to scheduling of embedded distributed real-time systems, and how different strategies for constraining the scheduling problem can lead to improved scheduling performance. In Section 5, we introduce the constraint-programming paradigm and show how it can be used for building a scheduling framework that supports all identified constraints (from Section 3) as well as single- and multi-objective optimization. In Section 6, we demonstrate the practical usefulness of the framework by performing several case studies and evaluating different aspects of the framework's scheduling performance. Finally, in Section 7, we discuss our results and suggest future directions, while our findings are summarized in Section 8.

2 Background

The design of real-time systems encompasses three important phases. From the specification of the system, it must be possible to identify the *requirements* on the system in terms of functionality, performance, and cost. Based on the system specification, an implementation phase then follows wherein the designer makes a choice of programming environment, run-time system and hardware architecture. From the choice of implementation, it is possible to identify the concrete tasks that the application is required to perform, and the resource that are available for its execution. Finally, the designer schedules the application tasks on the available resources. This is done by using a scheduling framework that automates the scheduling and allocation of the task, and also verifies that all requirements of the application are met. In order to do this, however, it is necessary to use models of the application tasks and the system architecture.

In the scheduling phase, it is possible to identify two potential caveats in the modeling of the tasks and the architecture. First, it is necessary (for practical reasons) to abstract away some implementation details and replace them with quantitative measures; for example, program code stretches and processor hardware mechanisms are typically analyzed separately in order to derive a worst-case execution time of

each task. These abstractions suffer from a potential risk of being overly pessimistic since embedded real-time systems often contain strict timing constraints. Second, in order to convey the system requirements to the scheduling framework, it is necessary to introduce a set of *constraints* that models system requirements into a manageable form for the scheduling algorithm. Since the scheduling framework is used to validate that the system requirements are actually met by the implementation, the methodology used for constructing constraints must be cognizant of the practical application domain as well as the theoretical scheduling domain. In our methodology, the constraints must *(a) reflect the intended behaviour of the system*, *(b) be derived in natural way without overconstraining the system*, and *(c) be fully supported by the scheduling framework*. In the following subsections, we describe the state-of-the art in practical real-time system design with respect to these three features.

2.1 Requirements

The system requirements for an embedded distributed real-time system can be divided into different groups which each impose a number of constraints on the scheduling problem.

The *functional* behavior of a real-time system is determined by the tasks and resources that constitute the system. Typical functional behavior requirements are those that control task execution order or task allocation, that is, *how* a task should execute. A real-time system also have *temporal* behavior requirements in addition to the functional ones. The temporal behavior of a task depends mainly on the environment (sensors, actuators or other tasks) that the task interacts with, that is, *when* a task should execute. These requirements directly affect the modeling of the application tasks and consequently the construction of the scheduling constraints.

Most real-time systems are also safety-critical in the sense that a failure in a processor could be catastrophic. To avoid such failures the embedded system must often also be *fault tolerant*. Fault tolerance is achieved by introducing redundancy in the system, for example, using additional processors and/or copies of tasks. Hence, it is clear that the requirement of making a system fault tolerant affects the implementation of the system, and consequently also the construction of scheduling constraints.

Apart from mere software requirements, it is also a practical consideration that the development of embedded real-time systems is made *cost-effective* so as to allow for mass-production of the system. That is why development using off-the-shelf hardware components has become a viable alternative in modern designs. Other practical aspects of embedded system design encompass the introduction of *weight* and *power-consumption* requirements. This means that cost, performance and various physical characteristics of the hardware components (processors, memory and busses) need to be conveyed to the constraint construction process.

2.2 Constraint derivation

Recent analyses [1, 2] have indicated that *most constraints are artifacts of the design*. This does not come as a surprise since they are a necessary consequence of the models used for tasks, run-time system and architecture. However, since the modeling process often has to be done manually, constraints should be constructed according to suitable guidelines or otherwise serious drawbacks will result. For example, an *ad hoc* constraint derivation is likely to lead to an over-constrained system which may prevent the scheduling algorithm from finding the best (or any) solution. On the other hand, if the system is too loosely constrained, finding the optimal solution might by too computationally intractable in practice. Furthermore, the semantics of the original requirements may be lost in the process, which from a practical point of view means that it will be hard to detect performance bottlenecks in the case of a failed scheduling attempt [6].

To overcome these problems, many researchers have recently proposed automated methods for constraint derivation. An example (taken from [1]) of a design flow for constraint derivation is illustrated in Figure 1. As indicated in the figure, there are basically two types of constraint derivation that can be performed: (a) derivation of system-level end-to-end constraints from performance requirements, and (b) derivation of task-level constraints from the end-to-end constraints.

The first type of constraint derivation means translating performance requirements, such as maximum steady-state error or maximum transient overshoot, into system-level end-to-end constraints that describe,

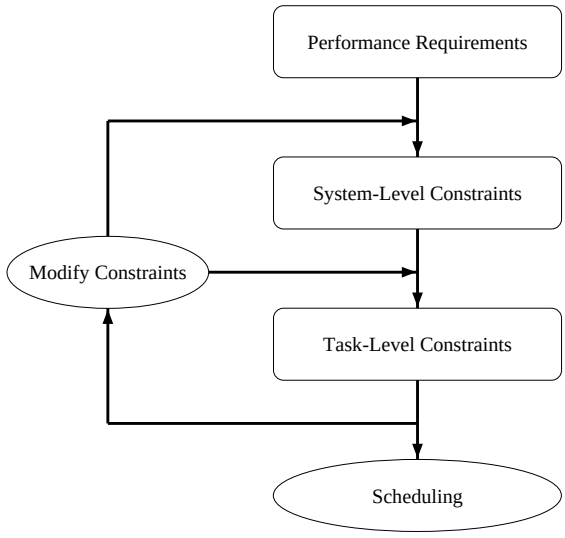


Figure 1: Constraint-derivation flow.

for example, maximum sensor-to-actuator latency, minimum sampling periods, or maximum output jitter. To that end, several techniques for system-level constraint derivation have been proposed in the context of control systems (see, for example, [7, 8, 9]).

The second type of constraint derivation means translating the end-to-end constraints into task-level timing and execution constraints. One of the most refined techniques for this type of constraint derivation is the period calibration method (PCM) proposed by Gerber, Hong and Saksena [10, 11]. The PCM works in a two-stage fashion. First, task periods are derived from given end-to-end system constraints. Deadlines and release times are then assigned to individual tasks using the derived periods[1]. While several other techniques exist for deriving task-level deadlines and release times (see, for example, [13, 14, 15, 16, 17, 18]), few techniques actually exist for deriving other types of task-level constraints. However, Ramamritham [19] and Abdelzaher & Shin [20] propose techniques that generate task clustering recommendations based on heuristic analyses of inter-task communication needs and task periods, respectively.

All of these constraint derivation techniques assume a subsequent scheduling stage which means that constraint derivation is primarily performed with the objective of increasing the likelihood of succeeding with the scheduling attempt. In fact, most techniques are tailored for a specific scheduling policy in order to generate good results[2]. It should also be noted that, with few exceptions, all techniques for derivation of task-level *timing* constraints work under the assumption that tasks have been assigned to processors in advance, which clearly limits their applicability to larger distributed systems. As we will see later in this paper, the constraint-programming paradigm does not suffer from this limitation.

2.3 Scheduling approaches

Recall that a scheduling framework for embedded system design should include a scheduling algorithm that is capable of accounting for various types of constraints. It is also desired that the algorithm should find a "good" or even optimal schedule using advanced single- or multi-objective optimization strategies. Unfortunately, most scheduling algorithms proposed in the real-time research literature fail to satisfy both of these requirements. In fact, there are very few scheduling approaches that even claim to solve this complex problem. However, there are three scheduling approaches that do have the potential to provide the capabilities necessary.

Simulated annealing is a local search technique that has been applied to real-time allocation and scheduling by Tindell, Burns and Wellings [22]. The method attempts to minimize a so-called energy function which describes the quality of a schedule. The algorithm offers great flexibility but since the

[1] In a recent paper, Nilsson *et al.* [12] presented an application of PCM on an avionics control software using constraint programming.

[2] An integrated constraint-derivation and scheduling approach is proposed in the holistic approach by Tindell *et al.* [21].

energy function contains both the constraints and the scheduling objective it can be troublesome to balance these factors correctly. The stochastic nature of simulated annealing reduces the computational complexity since only portions of the search space are examined but this also means that there is no guarantee that the resulting schedule is optimal.

Branch-and-bound (B&B) systematically explores the search space in a tree-like fashion (branching). The search space is reduced by pruning branches in the tree that can not improve the solution found so far (bounding). For this method to be effective the search-tree should be subject to a lot of pruning. However, the computation of an accurate pruning rule is about as complex as solving the problem in the first place. The use of B&B in the context of scheduling for distributed real-time systems have been investigated by several researchers (see, for example, [23, 24, 25, 26, 4]).

Constraint programming is a technique which has been used with great success to solve scheduling problems within operations research and artifical intelligence. The scheduling problem is expressed as variables and constraints which are handled by a constraint solver. The method has been shown to be promising also for real-time scheduling by Schild and Würtz [27]. We extend this work by considering additional real-time constraints and also adding support for different scheduling objectives. This is discussed in more detail in Section 5.

3 Identification of Constraints

An important issue in the design of embedded real-time systems is what task-level constraints exist for that particular domain. In this section, we look closer at these constraints and attempt to justify the presence of each constraint construct by examining its origin. We also discuss how constraints relate to each other, and, based on this information, attempt to identify a minimal set of necessary constraints to be implemented in a scheduling framework.

The system model assumed in this paper has been chosen to reflect a typical embedded system. The hardware architecture consists of a number of *nodes* which are connected via a *bus*, and each node contains one or more *processors*. The application includes a set of *tasks* that execute on the processors and possibly communicate by message passing on the bus. Each node has a number of *resources* that can be used locally by tasks at that node, or globally by all tasks in the system.

3.1 System constraints

System constraints are imposed by the selected hardware architecture of the system. How the hardware is composed depends on functional or economical reasons, which means that the system configuration is typically fixed. The configuration includes information on the number of processors and other resources in the system (that is, a model) as well as their performance and capacities (that is, properties). Hence, the system configuration can be expressed using *model* and *property* constraints.

Processors have different speed which means that the worst-case *execution time* of a task depends on which processor it is scheduled to execute on. Hence, in a distributed system the actual execution time for a task has to be dynamically calculated during the scheduling. Each processor is also subject to a *context-switch cost* which usually is assumed to be small enough to be neglected or, in the case of non-preemptive scheduling, assumed to be included in the task's worst-case execution time. However, for preemptive scheduling, where it is not known beforehand how many times a task will be preempted, this approximation is not desirable. In this case, worst-case execution time and context-switch cost have to be handled separately during scheduling.

Resources other than processors have different limited *capacities* which restrict the number of tasks that can simultaneously access a resource. For example, a processor might have a RAM of 16 kbytes. Assume there are three tasks T_1, T_2 and T_3 which each use 4, 8 and 12 kbytes, respectively, during execution. Then T_1 and T_2 are allowed to execute concurrently as well as T_1 and T_3, whereas T_2 and T_3 are not. A special case is a critical section which can be modelled as a resource with capacity 1. Tasks which share a particular critical section are then defined to require usage of 1 entity from this resource during their execution.

The communication on the bus can be modeled using a *transmission delay* which reflects the time to transmit messages. The transmission delay $c_{transmit}$ is expressed as:

$$c_{transmit} = c_{speed} \cdot c_{size} + c_{overhead} + c_{delay}$$

where c_{speed} is the data rate of the bus, c_{size} the size of a message to transmit, and $c_{overhead}$ and c_{delay} are overheads pertaining to the message queuing/retrieval and bus scheduling strategy, respectively. There are a number of bus scheduling strategies which each gives different c_{delay} in the formula above. A *linear* model causes no delay since it is assumed that the message always can be delivered whenever requested because there is no contention from other messages. In a *contention-based* model (such as the one used by Mecel's BASEMENT architecture [28]) the bus is regarded as an additional processor and the messages as tasks be to scheduled on the bus. The delay then depends on what other messages that simultaneously contend for the bus. If the bus uses a *time-triggered* protocol, for example, DACAPO [29] or TTP [30], messages can only be sent in dedicated time slots. The delay then depends on how long time it is to the next available time slot. In a *priority-based* communication protocol, such as CAN, the delay is caused by higher-priority messages being sent [31]. It is clear from this discussion that the transmission delay for each message has to be dynamically calculated during the scheduling.

3.2 Task constraints

Task constraints are the restrictions that control the *timing* and *execution* behavior of the tasks and concern both the behavior of a single task (*intra-task*) as well as interaction between tasks (*inter-task*).

3.2.1 Intra-task timing constraints

In this group, we find constraints that restrict the time limits within which a single task should execute.

The *period* determines how frequently a task should execute and is linked to how accurate the system needs to be in the interaction with its environment. From a pure functionality point of view it is likely that the required period not is completely fixed, but rather is expressed as an interval (sometimes called a *separation* constraint). To guarantee the responsiveness required by the system, a task is also subject to a *deadline* that defines an upper limit on the finish time of the task. On the other hand, a task must not start too soon. For example, a task might require a sensor value which is not immediately available at the beginning of the task's period. Hence, the *release time* of the task also has to be constrained. Note that periods are related to both deadlines and release times in that the k^{th} invocation of a task must typically be completed before invocation $k+1$, thus imposing deadlines and release times for each invocation.

Besides release time and deadline, the start and finish times of a task can also be constrained by input or output *jitter* constraints. Input jitter is the difference between the release time and the actual start time of the task. Similarly, output jitter is the difference between the deadline and the actual finish time of the task. The main reason to limit the amount of jitter is when a task is required to execute at regular intervals, which is the case in control applications.

As mentioned earlier, many scheduling algorithms require that deadlines and release times are defined for each task in order to solve a certain scheduling problem. If only end-to-end deadlines are given for the application, these deadlines must be split into shorter deadlines which are assigned to each task. In a similar fashion, release time constraints are typically imposed as a way to obtain mutual exclusion between tasks that access the same resource. However, the introduction of such forced artificial task-level constraints may put an unnecessary strain on the task set, and hence affect schedulability. To avoid this, the semantics of mutual exclusion and end-to-end deadlines should be modeled using more flexible constraints, such as resource-usage and precedence constraints.

3.2.2 Inter-task timing constraints

This group includes constraints that express time limits within which two tasks should execute in relation to each other.

Interacting tasks are subject to three types of constraints that originate from the application requirements, namely *distance* constraints (due to transmission delay or input/output delays in a control system),

freshness constraints (due to aging of data in, for example, database applications) and *correlation* constraints (due to limits on the allowed time-skew in concurrent operations, for example fault-tolerance voting). There is also another type of inter-task timing constraint, but which originates from the implementation of the system, namely *harmonicity* constraints. These constraints are mainly imposed to simplify the implementation of communicating tasks. It is desired that the period of the receiver task is exactly divisible by the period of the sender task since this simplifies the procedure of identifying received messages. Depending on the run-time scheduler used, this constraint may or may not be needed. For example, in a priority-driven scheduler the harmonicity constraints are necessary since the order of execution for two tasks of different periodicity is not predictable. On the other hand, if a time-driven scheduler is used, it is possible to guarantee (using precedence constraints) that the order of execution between two such tasks is always the same.

3.2.3 Intra-task execution constraints

This group includes constraints that are local to a single task and determine on what processor and with what resources the task should execute.

If a task *uses a resource*, the set of nodes that the task can be allocated to is automatically restricted since the resource must be present at the node and have enough capacity. We distinguish between the cases where the resource is always required (*static*) and where the resource is required only during execution (*dynamic*). Examples of these cases are ROM and RAM, respectively. Tasks can also be directly allocated to a certain node using *locality* constraints which often are implementation recommendations made by the designer, and less frequently given in the specification. Another implementation issue is whether all invocations of a task have to execute on the same processor (*affinity*), which is most likely desired in a distributed system because the cost of migrating the task is too high. It also has to be decided whether tasks are allowed to interrupt each other (*preemption*). By allowing preemption it might be possible to find schedules for designs that are otherwise infeasible.

3.2.4 Inter-task execution constraints

This group includes constraints that determine in what order, on what processor, and with what resources two or more tasks should execute in relation to each other.

A specific function in the system is often modeled as a sequence of operating tasks. Hence, tasks should execute in a certain order which is typically expressed by *precedence* constraints. In case data should be exchanged between tasks, *communication* constraints are used to model a precedence constraint as well as an amount of data to communicate[3]. It is worth noting that distance constraints can actually be used to model precedence (assume a "distance" equal to 0) as well as communication constraints (assume a "distance" equal to a fixed transmission delay).

In some distributed systems, *clustering* is used to assign communicating tasks onto the same node in order to limit the cost for the communication network or to increase the schedulability. Another rationale for clustering is when the system has different modes for which different task sets are active. Now, if only a few tasks are active in one mode, it could be a good idea to save power by turning off all nodes that do not contain any active tasks. Hence, tasks that are active in the same mode should be allocated to the same node. Such an example is the computer system of a car which operates differently depending on whether the car is turned on (driving) or off (parked) and where the system should not consume too much power while parked. The opposite of clustering, *anti-clustering*, refers to the case when a set of tasks *cannot* execute on the same node. The typical application for this constraint is when tasks have been replicated to achieve fault-tolerance and the replicas must be allocated to different nodes.

In systems where it is necessary to prevent simultaneous access to indivisible resources, such as critical sections and I/O-devices, *exclusion* constraints determine whether two tasks are allowed to execute concurrently. It should be noted that this is an artificial constraint, the existence of which is merely due to the lack of support for more natural constraints (such as resource-usage constraints) in existing scheduling algorithms.

[3]Note that the communication constraint becomes a pure precedence constraint if the tasks are located on the same node since the transmission delay will be zero.

Constraint	System		Task				Origin		
	Model	Property	Timing		Execution		Natural	Implementation	Artificial
			Inter	Intra	Inter	Intra			
Processors	X							X	
Resources	X							X	
Context switch		X						X	
Transmission delay		X						X	
Resource capacity		X						X	
Execution times		X						X	
Deadlines				X			X		x
Release times				X					X
Periods				X			X		x
Jitter				X			X		
Distance			X				X		
Freshness			X				X		
Correlation			X				X		
Harmonicity			X						X
Preemption						X		X	x
Affinity						X		x	X
Locality						X		X	x
Resource usage						X		X	
Precedence					X		X		
Communication					X		X		
Clustering					X		x		X
Anti-clustering					X			X	
Exclusion					X				X

Table 1: Constraint taxonomy showing which groups a constraint mostly (X) and sometimes (x) belongs to.

3.3 Constraint taxonomy

The constraints and the groups they are divided into constitute a constraint taxonomy which is summarized in Table 1. The table shows whether a constraint is *natural*, *implementation-based* or *artificial*. Natural constraints are directly derived from the system requirements while implementation-based constraints are imposed as a consequence of the choice of hardware architecture and run-time scheduling strategy. Artificial constraints are typically a result of adapting to limitations in existing scheduling algorithms or to control the scheduling of the tasks.

We believe that the constraint taxonomy can aid the system designer in the process of modeling the scheduling problem. With knowledge of common constraint definitions, it becomes easier to identify constraints from the system requirements. Furthermore, the identification becomes more exact which makes the model more accurate, thus increasing schedulability.

4 Definition of Optimality

Optimization of a schedule is possible and desirable since the specification does not completely determine the system. To be able to determine whether a schedule is optimal or not, we need to identify a measure for the quality of a schedule. This measure can involve one or more metrics which each provide information about the schedule's quality. We refer to these two cases as *single-objective* and *multi-objective* optimization problems.

4.1 Single-objective optimization

Single-objective optimization requires that we define an *object function* which computes the value of a schedule. That is, if x is a feasible schedule and f is the object function, then $f(x)$ is the value of the

schedule. In this section, we describe some object functions for single-objective optimization that are relevant in the design of embedded distributed real-time systems.

A traditionally-used optimization criterion for hard real-time systems is the *maximum lateness*, which is the same as the shortest slack in the final schedule. The rationale for using lateness as the optimization criterion is made clear by noting that the lateness will never exceed 0 for a feasible schedule.

In a system with a fixed number of processors, a commonly-used optimization criterion is the *load balance*. This optimization strategy means distributing the tasks between the processors such that they have approximately the same amount of load[4], which makes the most use of the available resources.

If the hardware architecture has not been fixed, on the other hand, it could instead be important to minimize the *number of processors* used in order to keep hardware costs down. To this end, the number of processors could be regarded as unknown and be subject to minimization. However, in reality parameters such as number of processors and other resources are not flexible enough to allow for optimization. Instead, an incremental approach, where various system configurations are manually evaluated, is likely to be more suitable.

For distributed systems, it is sometimes also necessary to optimize with respect to *communication*, that is, the amount of messages sent on the network. In embedded systems, the main arguments for minimizing the time required for message passing are that a low bus utilization may (a) enable the system designer to use a cheaper bus with less communication bandwidth, and (b) reduce the total amount of cabling in the system, thus reducing weight. If contention-based message scheduling is used it is typically desired to group the messages into frames, of a certain size, which are then used to transmit the messages. The objective of such a strategy is that it will minimize the overhead associated with sending/receiving messages.

In control systems, minimizing *jitter* is typically used as the optimization criterion. Note that jitter can be considered as a scheduling objective as well as a constraint. Minimizing jitter means minimizing the drift in the task invocations.

Since many embedded real-time systems are also dependable, it may be necessary to maximize the *reliability* of the system. That is, given the failure rate of each hardware component, the probability of a system failure is minimized.

4.2 Multi-objective optimization

Multi-objective optimization is necessary when we have several object functions which we want to optimize at the same time. In contrast to the single-objective case, we are now confronted with the problem of combining these functions such that a returned solution will be in line with what is considered optimal. We will now discuss some possible combinations.

The simplest approach is to make a single-objective function which is the sum of the participating functions, that is, $f(x) = \sum f_i(x)$. The major disadvantage with this approach is that, if the variation in the values of the different functions is large, some functions may dominate others. For example, assume $f_1(x) \in [0, 2]$ and $f_2(x) \in [0, 42]$. Then, if $f_1(x) = 2, f_2(x) = 21$ we obtain $f(x) = 23$ which should be considered to be better than the case where $f_1(x) = 1, f_2(x) = 28$ (which gives $f(x) = 29$) because $f_1(x)$ is maximized for the former case. However, the latter case has a greater single-objective value (assuming maximization) and will be considered the best solution for this optimization approach. A way to overcome this disadvantage is to assign weights to the functions [32]. It can be difficult, though, to come up with well-working weights.

Another way to solve the unbalance in the summarization approach is to make sure that the value ranges of the object functions are the same [33]. This can be achieved by mapping the original values into ranges of equal size, for example, $[0, 100]$. Our previous example would then give $f_1(x) = 2 \rightarrow 100, f_2(x) = 21 \rightarrow 50, f(x) = 150$ which is better than $f_1(x) = 1 \rightarrow 50, f_2(x) = 28 \rightarrow 67, f(x) = 117$. The size of the mapped range should correspond to the largest value range among the object functions. Otherwise, a (small) increase in its value might fail to increase its mapped value.

If the value ranges are equal yet another approach can be taken. Instead of a sum, the combined value is chosen to be the minimum of all object function values. This value is then to be maximized. This approach tries to balance the function values even further since a function far away from its optimum is more likely to be increased than one that is close.

[4]By 'load' of a processor, we mean the sum of the task execution times on that processor.

It could also be argued that the optimization should be performed in priority order. That is, first optimize on the most important function, and then optimize on the next second most important function under the assumption that value of the first function is maintained. This procedure is then repeated for all object functions. An example where this approach may be applicable is in communication scheduling. In this case the bus utilization is first minimized to determine which messages should be sent. Then, these messages should be formed in as few groups as possible in order to make the transmission less expensive.

5 Constraint Programming Framework

Most proposed real-time scheduling algorithms are "hard-coded" with respect to the constraints, task properties, resources and optimization criteria they can handle. Although a specific method in most cases outperforms a generic one, the former soon becomes quite complex and it becomes difficult to include new features. The constraint programming approach allows constraints to be specified in terms closer to the requirements. The included constraint solver then provides techniques to automatically perform the constraint derivation as well as the scheduling.

5.1 Introduction to constraint programming

The problems addressed using the constraint programming paradigm are called *constraint satisfaction problems (CSP)*. A CSP consists of variables with associated domains and constraints between the variables. A solution to a CSP is an assignment of values to the variables such that all constraints are satisfied. The programming of a CSP can be described in three steps:

(1) Declare the variables (and their domains)
(2) Post the problem constraints
(3) Search for a feasible or optimal solution

The constraint solver uses *propagation* in order to reduce the search space. That is, by considering the constraints the domains of the variables are narrowed by removing values that cannot be part of a solution. However, propagation alone is usually not enough to find a solution, but backtrack searching is also required.

The tool that we have based our framework on is SICStus Prolog [34] and its associated constraint solver for finite domains [35]. An interesting feature of this tool is the possibility to guide the search algorithm by varying the search parameters. By using suitable parameters the search time can be significantly reduced. The translation of the constraints into the code format internal to the solver is fairly straightforward and several examples of how constraints and applications are modeled can be found in [36].

5.2 Problem feasibility

In our case, a schedule is feasible if there exists a solution to the corresponding CSP. It is desired that a feasible schedule is found quickly. Not only from the users point of view, but also in the context of optimization. This schedule is then used as a starting point for further refined attempts to find an optimal solution. In order to speed up the search we aid the constraint solver by supplying heuristics. Intuitively, tasks should be balanced between the nodes in order to increase the chances of meeting their deadlines. However, this is not visible to the solver. Therefore, we pre-assign the tasks to different nodes before the actual search begins. This assignment is undone if it turns out to make the problem infeasible.

5.3 Problem optimality

The optimization algorithm (for minimization) used in the framework can be described as follows:

(1) Find a feasible schedule, x_i
(2) **If** a schedule is found **then** add the constraint $f(x) < f(x_i)$ **else** x_{i-1} is optimal
(3) Increase i and go to (1)

Solution	Loosely constrained	Tightly constrained
Feasible	Easy since many solutions exist	Harder since propagation usually can not reduce the search space enough
Optimal	Can be harder since many candidate solutions exist	Can be easier since the search space is smaller

Table 2: Complexity relationship (of original problem).

This algorithm applies directly to single-objective optimization. For the multi-objective case the object functions are mapped into new ones with range $[0, 100]$ which represents percentage of the optimal solution. We then use a combination of summarization and maximizing minimum. That is, for each function f_j the inequality $f_j(x) \geq \min\{f_j(x_i)\}$ should hold as well as the inequality $\sum f_j(x) > \sum f_j(x_i)$. The framework also supports priority-ordered optimization which is treated as consecutive single-objective optimizations.

Hence, once a feasible schedule is found the search is restarted but with additional constraints. These optimality constraints does not only state that a better solution is required but also prunes the search space due to constraint propagation.

5.4 Problem complexity

The complexity involved in finding a feasible solution versus finding an optimal solution to a CSP depends on the tightness of the problem, that is, the relation between the number of solutions and the size of the search space. The relationship is illustrated in Table 2 which was presented by Tsang in [37].

We can see that, since our constraint derivation approach does not make the problem tighter than necessary, we should be able to quickly locate of a feasible schedule. Furthermore, once a solution is found, our optimization algorithm increasingly tightens the problem and thus continuously decreases the search space.

6 Performance Evaluation

To validate the quality of our approach we will now evaluate the scheduling framework using a set of realistic examples. This will stress both the modeling and scheduling capabilities of the framework. To this end we have studied the following applications:

- A mobile base-station [38] which includes 3 processors and 17 tasks with precedence, communication, clustering, locality and release time constraints. There is one end-to-end deadline.
- A control application [16] which includes 4 processors and 22 tasks with precedence, communication, clustering and locality constraints. There are 5 end-to-end deadlines.
- A safety-critical application [19] which includes 3 processors and 12 tasks with communication and anti-clustering constraints. There are 2 end-to-end deadlines.

6.1 Framework setup

We configured the scheduling framework to model time-driven, non-preemptive task scheduling with affinity (no migration) and contention-based message scheduling.

Each application was tested for feasibility as well as for applicable aspects of optimality. For the single-objective optimization, we have used the objectives that were originally proposed in conjunction with the applications, namely lateness (for the base-station and control applications) and communication (for the safety-critical application). The multi-objective optimization was performed on lateness, load balance and communication. The estimated proximity to the optimum for lateness and communication were given by the constraint solver, while, for load balance, we optimistically divided the tasks between the processors such that the maximum load was minimized. The priority-order optimization was performed in the following order: communication, load balance and lateness. This choice of optimization order is used to reflect that, in a distributed embedded system, the communication may be considered to be the major bottle-neck. Once the communication has been determined, it could then be interesting to make the

Problem	Feasible	Single-objective			Multi-objective	Priority order
		objective	result	performance		
Base-station	0.11/6	lateness	-199	0.14/11	0.18/11	0.29/25
Control	0.10/0	lateness	-21	0.19/5	0.20/5	0.32/7
Safety-critical	0.11/7	communication	20	2.5/1463	1.8/345	4.0/2298

Table 3: Scheduling performance (secs/backtracks).

Solution	Function values			Estimated proximity to optimum			
	communication	load balance	lateness	communication	load balance	lateness	sum
1	32	32	-1	46	92	10	148
2	26	44	-4	56	69	40	165
3	22	44	-4	63	69	40	172
4	20	39	-9	66	78	90	234

Table 4: Multi-objective search trace for the safety-critical example.

best use of the available resources. Finally, lateness can be used to distinguish between otherwise equal solutions.

The examples were scheduled using an Ultra Sparc 10 with 128 M bytes of primary memory.

6.2 Results

The scheduling results are listed in Table 3. Performance is given as x/y where x is the time in seconds and y is the number of backtracks in the constraint solver.

Mainly, the number of backtracks gives information about the difficulty of the problem. The time, however, also depends on the number of constraints and variables which were created, that is, the size of the problem. The differences in the number of backtracks for an example reflect the tightness of the problem. For instance, the number of backtracks required to find an optimal solution for the control application using single-objective optimization (5) is only slightly higher than for a feasible solution (0). Hence, the application constraints almost completely determine the problem which results in a small search space. In contrast, the safety-critical application is subject to a large difference between the number of backtracks used to find the feasible (7) and single-objective (1463) solutions. Hence, this problem is less constrained which results in a large search space. These observations consequently corroborate the relationships listed in Table 2.

Table 4 shows how the search progresses in the multi-objective optimization of the safety-critical application. In the table it can be seen that in the first found solution, the load-balancing objective is estimated to have reached 92 percent of its optimal value while the lateness only has reached 10 percent. The constraint solver then tries to find a solution where all objectives have reached at least 10 percent and where the sum of the estimated proximity values is larger than the current sum (148 percent). In the next found solution, the lateness and the communication have increased their percentage at the cost of the load balance. However, since the total sum is greater (165 percent) than before this new solution is regarded as a better one. After four iterations optimum (of the estimated proximity) is obtained despite a rather low proximity of optimum for the individual functions. The main reason for this is that the estimation mechanisms used are too weak, something we will elaborate further on in Section 7.

6.3 Complexity comparison

To demonstrate how a problem is affected when additional constraints are introduced, we solve the scheduling problem for the control application using the DACAPO communication model [29]. The DACAPO model is more restrictive than the contention-based model used so far since communication must now be synchronized with time slots. The DACAPO model assumes that each processor is given dedicated time slots where tasks on that processor are allowed to send messages. The time slots are of fixed size and

Strategy	Slot size	Backtracks	Optimal result
Contention	-	5	-21
DACAPO	1	13	-20
DACAPO	2	1	-16
DACAPO	5	1	no solution

Table 5: Effects of over-constraining in the control application.

are divided equally between the processors. It is assumed that the least common period (length of the schedule) is a multiple of the number of processors. Furthermore, the total amount of transmission time for a processor should be a multiple of the size of the time slots. A message must fit into one time slot. The slots are assigned to the four processors in a round-robin order[5].

As can be seen in Table 5, the use of the DACAPO model transforms the original problem into a non-optimal (or even infeasible) one. The table also demonstrates that, for a slot size of 2, the complexity was reduced, while, for a slot size of 1, the complexity was increased. Whether a constraint decreases or increases the complexity depends on its strength, that is, how much the constraint solver can reduce the search space due to constraint propagation in relation to the number of solutions. For example, a DACAPO slot size of 1 probably only restricts the number of solutions, while a slot size of 2 also restricts the search space as indicated in the table. A slot size of 5 seems to restrict the search space as well, but unfortunately also over-constrains the problem so that no solution is found at all.

7 Discussion

Constraint programming is a declarative approach where it suffices to express what constitutes a solution. The constraint solver then handles the actual search. However, this search can be significantly improved by using knowledge about the problem as guidance. The SICStus Prolog constraint solver offers the possibility to supply search heuristics. So far, we have not looked much into this, but we believe that by tailoring the search algorithm to the specific problem, complexity can be significantly reduced. Previous work for the B&B algorithm has shown that this is a viable approach [4, 5].

A CSP can usually be modeled in many different ways. How a problem is modeled affects the complexity of solving it. By remodeling or supplying redundant constraints the constraint propagation can be more effective which reduces the search space. It would be interesting to see if such reformulation is possible.

As can be seen in Table 4, the multi-objective optimization strategy obtains an optimal value (20) of the communication. However, the estimated proximity of the communication in the optimal solution is only 66 percent. Since this is the lowest value among all estimated values in the fourth iteration, no solution with a lower estimated proximity value for the communication will be accepted. On the other hand, if that value had been more accurate (for example, close to 100 percent), a decrease in the estimated proximity for the communication would be tolerable if the other proximity values increased, resulting in a better solution. Hence, it would be of great value to have a tight estimation of optimum since this is essential to make our multi-objective optimization approach work well and also reduce the problem search space. Thus, further investigation on how to obtain fast, but accurate, estimations of the optimal value is desired.

8 Conclusions

Scheduling of real-time tasks in an embedded distributed system is a difficult problem since such systems not only have standard task-level timing constraints (such as periods and deadlines) but also have more advanced constraints such as locality (what processor to execute on) and clustering (what other tasks to execute with). Unfortunately, few existing scheduling algorithms are capable of accounting for such diverse set of constraints. Moreover, scheduling of embedded systems often requires multi-objective optimization

[5]In our test run we used the slot-order $\{3, 4, 1, 2\}$. However, other slot orders were found to produce similar results.

in order to simultaneously account for requirements such as schedulability, reliability, power consumption and economic cost.

In this paper, we have proposed a scheduling framework based on constraint programming that is capable of providing these features. One main advantage of the constraint programming paradigm is that it suffices to specify what defines an acceptable solution, instead of having to provide guidelines on how to find it. We have argued that, by using this framework, it is easy to model relevant constraints as well as perform single- and multi-objective optimization for embedded distributed real-time systems. To this end, the applicability of the framework was evaluated using a set of realistic applications with non-trivial constraints.

References

[1] M. Saksena, "Real-Time System Design: A Temporal Perspective," *Proc. of IEEE Canadian Conference on Electrical and Computer Engineering*, Waterloo, Canada, May 1998, pp. 405–408.

[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, "Strategic Direction in Real-Time and Embedded Systems," *ACM Computing Surveys, 50th Anniversary Issue*, vol. 28, no. 4, pp. 751–763, Dec. 1996.

[4] 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.

[5] 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.

[6] G. Fohler, "Dynamic Timing Constraints — Relaxing Overconstraining Specifications of Real-Time Systems," *Proc. of the IEEE Real-Time Systems Symposium – Work-in-Progress Session*, San Francisco, California, Dec. 3–4, 1997, pp. 27–30.

[7] M. Ryu, S. Hong, and M. Saksena, "Streamlining Real-Time Controller Design: From Performance Specifications to End-to-End Timing Constraints," *Proc. of the IEEE Real-Time Systems Symposium*, San Francisco, California, Dec. 3–5, 1997, pp. 91–99.

[8] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin, "On Task Schedulability in Real-Time Control Systems," *Proc. of the IEEE Real-Time Systems Symposium*, Washington, D.C., Dec. 4–6, 1996, pp. 13–21.

[9] D. Seto, J. P. Lehoczky, and L. Sha, "Task Period Selection and Schedulability in Real-Time Systems," *Proc. of the IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec. 2–4, 1998, pp. 188–198.

[10] 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.

[11] M. Saksena and S. Hong, "Resource Conscious Design of Distributed Real-Time Systems: An End-to-End Approach," *Proc. of the IEEE Int'l Conf. on Engineering of Complex Computer Systems*, Montreal, Canada, Oct. 21–25, 1996, pp. 306–313.

[12] U. Nilsson, S. Streiffert, and A. Törne, "Detailed Design of Avionics Control Software," *Proc. of the IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec. 2–4, 1998, pp. 82–91.

[13] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic Scheduling of Real-Time Tasks under Precedence Constraints," *Real-Time Systems*, vol. 2, no. 3, pp. 181–194, Sept. 1990.

[14] B. Kao and H. Garcia-Molina, "Deadline Assignment in a Distributed Soft Real-Time System," *Proc. of the IEEE Int'l Conf. on Distributed Computing Systems*, Pittsburgh, Pennsylvania, May 25–28, 1993, pp. 428–437.

[15] R. Bettati and J. W.-S. Liu, "End-to-End Scheduling to Meet Deadlines in Distributed Systems," *Proc. of the IEEE Int'l Conf. on Distributed Computing Systems*, Yokohama, Japan, June 9–12, 1992, pp. 452–459.

[16] 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.

[17] J. J. Gutiérrez García and M. González Harbour, "Optimized Priority Assignment for Tasks and Messages in Distributed Hard Real-Time Systems," *Proc. of the IEEE Workshop on Parallel and Distributed Real-Time Systems*, Santa Barbara, California, Apr. 25, 1995, pp. 124–132.

[18] J. Jonsson and K. G. Shin, "Robust Adaptive Metrics for Deadline Assignment in Distributed Hard Real-Time Systems," *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, 2000, (to appear).

[19] K. Ramamritham, "Allocation and Scheduling of Precedence-Related Periodic Tasks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 4, pp. 412–420, Apr. 1995.

[20] T. F. Abdelzaher and K. G. Shin, "Period-Based Load Partitioning and Assignment for Large Real-Time Applications," *IEEE Trans. on Computers*, vol. 49, no. 1, pp. 81–87, Jan. 2000.

[21] K. Tindell and J. Clark, "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems," *Microprocessing and Microprogramming*, vol. 40, no. 2/3, pp. 117–134, Apr. 1994.

[22] K. W. Tindell, A. Burns, and A. J. Wellings, "Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy," *Real-Time Systems*, vol. 4, no. 2, pp. 145–165, June 1992.

[23] T. Shepard and J. A. M. Gagné, "A Pre-Run-Time Scheduling Algorithm for Hard Real-Time Systems," *IEEE Trans. on Software Engineering*, vol. 17, no. 7, pp. 669–677, July 1991.

[24] J. Xu, "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Trans. on Software Engineering*, vol. 19, no. 2, pp. 139–154, Feb. 1993.

[25] D. Peng, K. G. Shin, and T. Abdelzaher, "Assignment and Scheduling of Communicating Periodic Tasks in Distributed Real-Time Systems," *IEEE Trans. on Software Engineering*, vol. 23, no. 12, pp. 745–758, Dec. 1997.

[26] C.-J. Hou and K. G. Shin, "Allocation of Periodic Task Modules with Precedence and Deadline Constraints in Distributed Real-Time Systems," *IEEE Trans. on Computers*, vol. 46, no. 12, pp. 1338–1356, Dec. 1997.

[27] 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.

[28] H. A. Hansson, H. W. Lawson, M. Strömberg, and S. Larsson, "BASEMENT: A Distributed Real-Time Architecture for Vehicle Applications," *Real-Time Systems*, vol. 11, no. 3, pp. 223–244, Nov. 1996.

[29] B. Rostamzadeh, H. Lönn, R. Snedsböl, and J. Torin, "DACAPO: A Distributed Computer Architecture for Safety-Critical Control Applications," *Proc. of the IEEE Int'l Symposium on Intelligent Vehicles*, Detroit, Michigan, Sept. 25–26 1995, pp. 376–381.

[30] H. Kopetz and G. Grünsteidl, "TTP – A Protocol for Fault-Tolerant Real-Time Systems," *IEEE Computer*, vol. 27, no. 1, pp. 14–23, Jan. 1994.

[31] K. W. Tindell, H. Hansson, and A. J. Wellings, "Analysing Real-Time Communications: Controller Area Network (CAN)," *Proc. of the IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, Dec. 7–9, 1994, pp. 259–263.

[32] C. C. Amaro, R. Nossal, and A. D. Stoyen, "On Cost Function Synthesis For Multi-Objective Design Decisions in Complex Real-Time Systems," *Proc. of Int'l Conference on Engineering of Complex Computer Systems*, Las Vegas, Nevada, Oct. 18–21, 1999, pp. 86–97.

[33] P. J. Bentley and J. P. Wakefield, "An Analysis of Multiobjective Optimization within Genetic Algorithms," Tech. Rep. ENGPJB96, Division of Computing and Control Systems Engineering, The University of Huddersfield, Huddersfield HD1 3DH, U. K., 1996.

[34] Intelligent Systems Laboratory, *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, 1995.

[35] 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.

[36] C. Ekelin and J. Jonsson, "A Modeling Framework for Constraints in Real-Time Systems," Tech. Rep. 00-9, Dept. of Computer Engineering, Chalmers University of Technology, S-412 96 Göteborg, Sweden, May 2000, Available at http://www.ce.chalmers.se/~cekelin/constraints.us.ps.gz.

[37] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[38] Y. Zhao, "Derivation of Local Timing Constraints in Early Design Stages," Master's thesis, Dept. of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1999.

Memory and Time-Efficient Schedulability Analysis of Task Sets with Stochastic Execution Time

Sorin Manolache, Petru Eles, Zebo Peng
{sorma, petel, zebpe}@ida.liu.se
Department of Computer and Information Science,
Linköping University, Sweden

Abstract

This paper presents an efficient way to analyse the performance of task sets, where the task execution time is specified as a generalized continuous probability distribution. We consider fixed task sets of periodic, possibly dependent, non-pre-emptable tasks with deadlines less than or equal to the period. Our method is not restricted to any specific scheduling policy and supports policies with both dynamic and static priorities. An algorithm to construct the underlying stochastic process in a memory and time efficient way is presented. We discuss the impact of various parameters on complexity, in terms of analysis time and required memory. Experimental results show the efficiency of the proposed approach.

1 Introduction

Embedded systems are digital systems meant to perform a dedicated function inside a larger system. Most embedded systems are also real-time systems. Hence, their validation has to take into consideration not only the functional aspect but also timeliness.

A typical digital systems design flow starts from an abstract description of the functionality and a set of constraints. Subsequent steps partition the functionality, allocate processing units, map the functionality to the allocated processing units and, finally, select a scheduling policy and perform timing analysis.

In safety-critical applications missing a deadline can have disastrous consequences. Hence, a conservative model for task scheduling is adopted, the worst-case execution time (WCET) model. Such systems are designed to perform according to hard time constraints even in rare borderline cases. This leads to a processor underutilization for most of the operation time.

There are several opportunities to relax some of the conservative assumptions typical to hard real-time systems. This is the case for applications where missing a deadline causes an overall quality degradation, but it is still acceptable provided that the probability of such misses is below a certain limit. Such applications are, for example, multimedia and telecommunication systems.

Another opportunity to relax the WCET model is during the early design stages. For instance, in transformational design approaches, the design phases may not be performed in the simplistic waterfall sequence described above. It may happen that information about a schedule fitness would be needed without knowing exactly the processor on which a certain task will be executed. The processor itself could be still under design and, thus, no exact information concerning execution time would be available.

Tia et al. [11] provide an example where the maximum processor utilizations are around 145%, whereas the average utilizations do not exceed 25%. The large variation in utilizations stems from the large variation of task execution times. This could be due to several reasons: architectural factors (dynamic features like caches and pipelines), causes related to functionality (data dependent branches and loops), external causes (network delays), and dependencies on input data (very strong in multimedia applications). Another source of non-exact execution time specification is the limited amount of information available. This could be the case at early design phases or with designs integrating third party components, or other customer blocks with secret functionality and insufficiently specified non-functional interfaces.

Typically, by using variable execution time models, considerable savings in system (hardware) cost can be expected. The functionality can be implemented on less powerful and cheaper processors, leading to a higher processor utilization. In the case of overload, some tasks will, most likely, fail their deadlines. The designer has to be provided with analysis tools in order to guide his or her decisions and to estimate the trade-off between cost and quality, the failure likeliness and the failure consequences. Thus, new execution time models and analysis techniques have to be developed.

Approaches based on the average case or on probabilities uniformly distributed between the best and worst case execution time have the advantage of simplicity. However, their use is limited, as they do not give information on the likeliness of particular cases. More accurate models are based on execution time probability distributions. Those distributions can be derived from statistical models of the variation sources, from legacy designs, code analysis, simulations and profiling. One of the main difficulties with probabilistic models is their solving complexity.

The aim of this work is to provide a performance analysis method for task schedules considering probabilistic models of task execution times. The methodology is not specific to any particular execution time probability distri-

bution class or scheduling policy and, thus, it is adaptable to various applications. The result of the application analysis is the ratio of missed deadlines per task or task graph. In order to cope with the complexity problem typical to such an analysis, we have considered both execution time and memory aspects. The algorithm is efficient in both regards and can be applied to the analysis of large task sets. We have also investigated the impact of the task set parameters (e.g. periods, dependencies, and number of tasks) on the complexity of the analysis in terms of time and memory.

The rest of the paper is structured as follows. Section 2 surveys some related work. Section 3 details our assumptions and gives the problem formulation. Section 4 introduces the underlying stochastic process and illustrates its construction and analysis on an example. Section 5 presents the analysis algorithm and in section 6 we evaluate our approach experimentally. The last section draws some conclusions.

2 Related work

Much work has been done in the field of task scheduling with fixed parameters (periods, execution times, etc.). Results are summarized in several surveys such as those by Stankovic et al. [10], Fidge [5], and Audsley et al. [3]. However, only recently researchers have focused on scheduling policies, schedulability analysis and performance analysis of tasks with stochastic parameters.

Atlas and Bestavros [2] extend the classical rate monotonic scheduling policy with an admittance controller in order to handle tasks with stochastic execution times. They analyse the quality of service of the resulting schedule and its dependence on the admittance controller parameters. The approach is limited to rate monotonic analysis and assumes the presence of an admission controller at run-time.

Abeni and Butazzo's work [1] addresses both scheduling and performance analysis of tasks with stochastic parameters. Their focus is on how to schedule both hard and soft real-time tasks on the same processor, in such a way that the hard ones are not disturbed by ill-behaved soft tasks. The performance analysis method is used to assess their proposed scheduling policy (constant bandwidth server), and is restricted to the scope of their assumptions.

Spuri and Butazzo [9] propose five scheduling algorithms for aperiodic tasks. The task model they consider is one with aperiodic tasks but with fixed, worst-case execution time.

Tia et al. [11] assume a task model composed of independent tasks. Two methods for performance analysis are given. One of them is just an estimate and demonstrated to be overly optimistic. In the second method, a soft task is transformed into a deterministic task and a sporadic one. The latter is executed only when the former exceeds the promised execution time. The sporadic tasks are handled by a server policy. The analysis is carried out on this model.

Zhou et al. [12] root their work in Tia's. However, they do not intend to give per-task guarantees, but characterize the fitness of the entire task set. Because they consider all possible combinations of execution times of all requests up to a time moment, the analysis can be applied only to small task sets due to complexity reasons.

De Veciana et al. [4] address a different type of problem. Having a task graph and an imposed deadline, they determine the path that has the highest probability to violate the deadline. The problem is then reduced to a non-linear optimization problem by using an approximation of the convolution of the probability densities.

Lehoczky [8] models the task set as a Markovian process. The advantage of such an approach is that it is applicable to arbitrary scheduling policies. The process state space is the vector of lead-times (time left until the deadline). As this space is potentially infinite, Lehoczky analyses it in heavy traffic conditions, when the system provides a simple solution. The main limitations of this approach are the non-realistic assumptions about task inter-arrival and execution times.

Kalavade and Moghe [7] consider task graphs, where the task execution times are arbitrarily distributed over discrete sets. Their analysis is based on Markovian stochastic processes too. Each state in the process is characterized by the executed time and lead-time. The analysis is performed by solving a system of linear equations. Because the execution time is allowed to take only a finite (most likely small) number of values, such a set of equations is small.

Besides the differences in assumptions, our work diverges from Kalavade and Moghe's in the sense that we use pseudo-continuous execution time distributions (discretized continuous distributions) instead of being restricted to discrete sets. As the number of possible execution times becomes very high, it is infeasible to consider individual times as states. Our solution is to group execution times in equivalence classes. As a consequence, we have to use probability density convolutions for the analysis. In order to reduce the complexity in terms of memory space, the stochastic process is never stored entirely in memory, but we both construct and analyse the process at the same time.

3 Preliminaries and problem formulation

The system to be analysed is represented as a set of task graphs. A task graph is an acyclic graph with nodes representing tasks and edges capturing the precedence constraints among tasks. Precedences can be induced, for example, by data dependencies (a task processes the output of its predecessor). All tasks are executed on one single processor. They are assumed to be non-pre-emptable.

Let N be the number of tasks and let t_i, $0 \leq i < N$, denote a task. Let M be the number of task graphs and let g_i, $0 \leq i < M$ denote a task graph.

Each task is characterized by its period (inter-arrival time), assumed to be fixed, its deadline, and its execution time probability density. Let p_i and d_i, $0 \leq i < N$, denote the period and deadline of task t_i, where $d_i \leq p_i$. P, the applica-

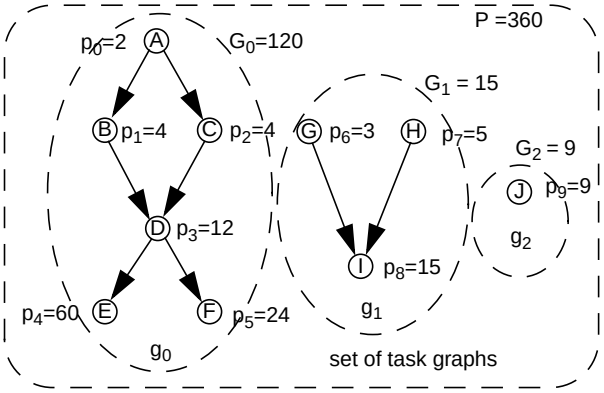


Figure 1. Set of task graphs

tion period, is the least common multiple of all task periods. The period of a task has to be a common multiple of the periods of its predecessors. The period of a task graph equals the least common multiple of the periods of its composing tasks. Let G_i, $0 \le i < M$, denote the period of the task graph g_i. A task graph g_i is activated every G_i time units.

A task consists of an infinite sequence of activations called jobs. In the sequel, we will say that a task is running when one of its jobs is running. Similarly, a task is ready when one of its jobs is ready, and a task is discarded when one of its jobs was discarded.

The task $t_k \in g_i$ is ready (pending, waiting) if and only if each of its predecessors t_j has run p_k/p_j times during the current activation of task graph g_i. If any of the jobs in a current activation of a task graph has missed its deadline, then the current task graph activation is said to have failed.

A probabilistic guarantee given for a task *t* is expressed as the ratio between the number of jobs belonging to *t* that miss their deadlines and the total number of jobs of the task *t*. A probabilistic guarantee given for a task graph *g* is expressed as the ratio between the number of the task graph's activations that fail and the number of all activations of the task graph *g*.

Figure 1 depicts a task set consisting of three task graphs g_0, g_1, and g_2. For each task and task graph the respective period is shown.

Let e_i, $0 \le i < N$, be the execution time probability density function (ETPDF) of task t_i. e_i is represented as a set of samples resulting from the discretization of the density curve. The discretization resolution is left to the designer's choice. We assume generalized probability densities of the task execution times. Figure 2 illustrates some possible ETPDFs. Density e_1 could happen if, for example, the task has only three computation paths and the variation around them is caused by hardware architecture

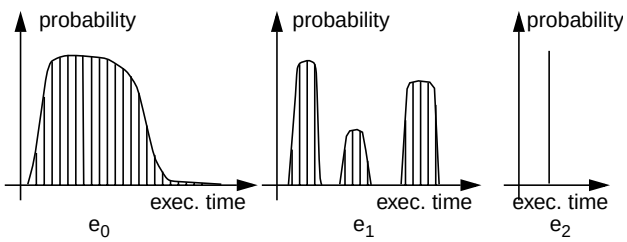


Figure 2. Exec. time probability density functions

factors. e_2 is a deterministic density (a Dirac impulse).

When a job of a task t misses its deadline it is discarded. If t has a successor in the task graph then two different policies are considered for the analysis among which the designer can choose:

1. The whole task graph is discarded. This means that all the jobs belonging to the current activation of the task graph are discarded. These are the ones already arrived but not yet completed and the ones that will arrive before a new activation of the task graph. This strategy is adopted in the case when the computed value of a job is critical for the continuation of the tasks in the task graph and it is a meaningful value only if the job met its deadline. In this case, it is meaningless to give per task guarantees and only per task graph guarantees will be produced as a result of the analysis. In the example in Figure 1 assume that two jobs of task A have successfully executed and B and C are now ready. Assume that B executes and misses its deadline. Then g_0 is discarded and no jobs of any of its tasks are anymore accepted until a new arrival of the entire task graph at a time moment multiple of G_0.
2. Only the missing job is discarded, but the rest of the task graph is activated normally. The successor tasks will consume either a partial result or the result from a previous execution of the discarded task. In this case, it is important to provide not only per task graph but also per task guarantees.

Task execution is considered to be non-pre-emptable. However, a method to work this around, if needed, is to define pre-emption points inside a task. For analysis, the task will be replaced by several dependent tasks with the same period and deadline as the original one, as shown in Figure 3. Let τ_i^q, $0 \le i < S$ be S tasks that resulted from task t_q, and π_i^q be their respective periods, δ_i^q their deadlines and ε_i^q their ETPDFs. Then $\pi_i^q = p_q$ and $\delta_i^q = d_q$, $0 \le i < S$. The convolution (denoted by *) of the execution time probability density functions ε_i^q has to be equal to e_q. Due to the fact that the tasks τ_i^q are dependent and have the same period, the consequence of such a task decomposition on the analysis complexity is limited, as will be shown later.

Problem formulation

The input to the analysis algorithm is a set of task graphs and a scheduling policy. The task graphs are given

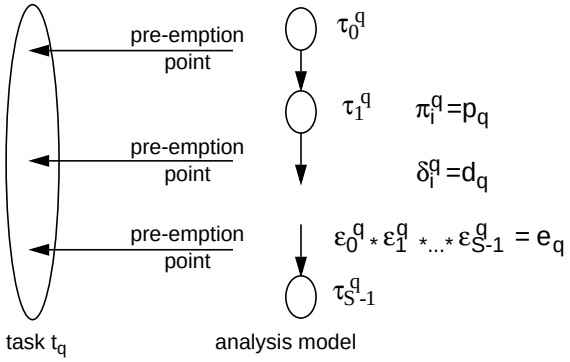


Figure 3. Introducing pre-emption points

according to the assumptions discussed previously. The scheduling policy is given as an algorithm to choose a next job knowing the set of ready jobs, their deadlines and the current time.

The analysis produces per task and per task graph probabilistic guarantees, as defined above.

4 The stochastic process

The analysis methodology for task sets with stochastic execution times relies on the underlying stochastic process. A stochastic process is a mathematical abstraction that characterizes a random process which proceeds in stages. The set of all possible stage outcomes in every stage forms the stochastic process state space. A stochastic process can be represented graphically in a similar manner as finite state machines. The difference is that a next state is known only probabilistically. It is assumed that the next state transition probabilities are known once the current and past states are known. If the next states and their corresponding transition probabilities depend only on the current state, then the process exhibits the Markovian property. The discrete time stochastic process that results from sampling the state space of the underlying continuous time stochastic process, at the time moments immediately following a job arrival or a job completion, forms the embedded stochastic process.

For the sequel, "process" will refer to the underlying stochastic process, and "state" will refer to the stochastic process state.

In order for the embedded process to be Markovian, certain information have to be available in a process state. A straightforward solution would be to characterize a state by the currently running task, the ready tasks and the start time of the running job. A state change would occur if the running task finished execution for some reason. The ready tasks can be deduced from the old ready tasks and the jobs arrived during the old task's execution time. The new running job can be selected considering the particular scheduling policy. In principle, there may be as many next states as many possible execution times the running job has. Hence, one factor that influences the stochastic process size is the task execution time span. This is the approach followed by Kalavade and Moghe [7] and leads to tree-like stochastic processes. Except the case that only a very small number of discrete execution times are allowed for each task, such an approach leads to an extremely huge state explosion. In our approach, we have grouped time moments into equivalence classes and, by doing so, we limited the process size explosion. Thus, practically a set of equivalent states is represented as a single state in the stochastic process. Even so, the application size is still limited by the amount of memory available for analysis. Therefore, we propose a way to perform the construction and the analysis of the process simultaneously. Consequently only a part of the process is stored in memory at any time during the analysis.

Due to the assumption that the tasks are discarded when they miss their respective deadlines, the analysis is performed over the interval [0, P), where P is the application period (the least common multiple of all task periods).

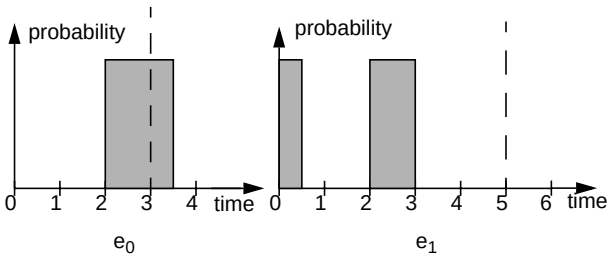


Figure 4. ETPDFs for tasks t_0 and t_1

In the following, an example is used in order to illustrate the construction and the analysis of the stochastic process. Let t_0 and t_1 be two independent tasks scheduled according to an earlier deadline first (EDF) policy. Let $p_0 = 3$ and $p_1 = 5$ be their periods and let $d_0 = p_0$ and $d_1 = p_1$. P is then 15. The execution time probabilities are distributed as depicted in Figure 4. For simplicity, the densities were not depicted as discretized. Note that e_0 contains execution times larger than the deadline.

As a first step to the analysis, the interval [0, P) is divided in disjunct intervals, the so-called *priority monotonicity intervals (PMI)*. A PMI is delimited by the time moments a job may arrive or may be discarded. Figure 5a depicts the PMIs for the example above. If the deadlines were $d_0 = 2$ and $d_1 = 4$, then the PMIs would be as depicted in Figure 5b.

Next, the stochastic process is constructed and analysed at the same time. Let us assume the straightforward approach mentioned earlier. In this case, a stochastic process state would be characterized by the index of the task the currently running job belongs to, the start time of this job and the indexes of the waiting tasks (see Figure 6a). τ_1, τ_2, …, τ_q in Figure 6a are possible finishing times for the job of task t_0 and, implicitly, possible starting times of the waiting job of task t_1. The number of next states equals the number of possible execution times of the running job in the current state. The resulting process is extremely large (theoretically infinite, practically depending on the discretization resolution) and, in practice, unsolvable. Therefore, we would like to group as many states as possible in one equivalent state and still preserve the Markovian property.

Consider a state s_0 characterized by {i, t, w}: the current

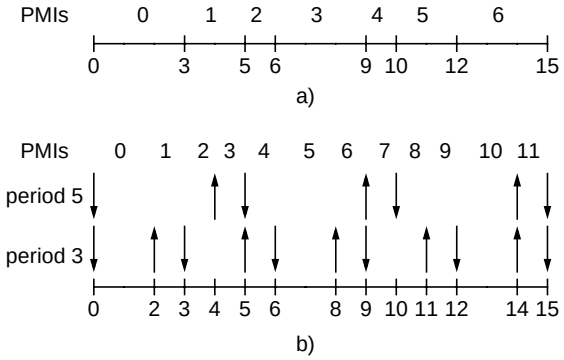


Figure 5. Priority monotonicity intervals

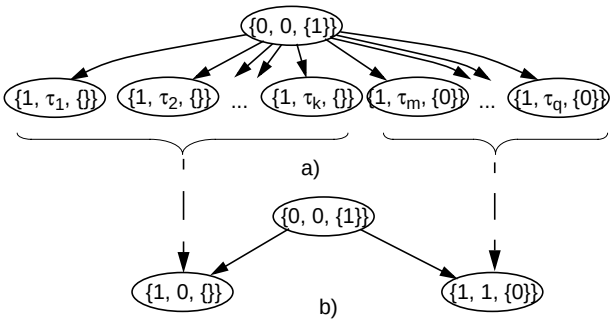


Figure 6. State encoding

job belongs to task t_i, it has been started at time t, and the waiting jobs belong to the tasks in set w. Let us consider the next states derived from s_0: s_1 characterized by {j, τ_1, w_1} and s_2 with {k, τ_2, w_2}. Let τ_1 and τ_2 belong to the same PMI. This means that no job has arrived or finished in the time interval between τ_1 and τ_2, no one has missed its deadline, and the relative priorities of the tasks inside the set w have not changed (see Section 5.1). Thus, j=k= the index of the highest priority task in the set w; $w_1=w_2=w\backslash\{t_j\}$. It follows that all states derived from state s_0 that have their time τ belonging to the same PMI have an identical current job and identical sets of waiting jobs. Therefore, instead of considering individual times we consider time intervals, and we group together those states that have their associated start time inside the same PMI. With such a representation, the number of next states of a state *s* equals the number of PMIs the possible execution time of the job that runs in state *s* is spanning over.

We propose a representation in which a stochastic process state is a triplet {r, pmi, w}, where r is the running task index, pmi is the index of the PMI containing the running job's start time, and w the set of ready task indexes. When there is no running task, then r = -1. In our example, the execution time of task t_0 (which is in the interval [2, 3.5]) is spanning over the PMIs 0 and 1. Thus, there are only two states emerging from the initial state, as shown in Figure 6b.

Let $\wp_i$, the set of predecessor states of a state s_i, denote the set of all states that have s_i as a next state. The set of successor states of a state s_i consists of those states that can be reached directly from state s_i. With our proposed stochastic process representation, the time moment a transition to a state s_i occurred is not determined exactly, as the task execution times are known only probabilistically. However, a probability density of this time can be deduced. Let z_i denote this density function. Then z_i can be computed from the functions z_j, where $s_j \in \wp_i$, and the ETPDFs of the tasks running in the states $s_j \in \wp_i$.

Figure 7 depicts a part of the stochastic process constructed for our example. The initial state is s_0: {0, 0, {1}}. The first field indicates that a job of task t_0 is running. The second field shows the current pmi (0), and the third field denotes that task t_1 is waiting. If the job of task t_0 does not complete until time moment 3, then it will be discarded. The state s_0 has two possible next states. The first one is state s_1: {1, 0, {}} and corresponds to the case when the job completes before time moment 3. The second one is state s_2: {1, 1, {0}} and corresponds to the case when the job was discarded at time moment 3. State s_1 indicates that a job of task t_1 is running (it is the job that was waiting in state s_0), that the pmi is 0 and that no job is waiting. Consider state s_1 to be the new current state. Then the next states could be state s_3: {-1, 0, {}} (task t_1 completed before time moment 3 and the processor is idle), state s_4: {0, 1, {}} (task t_1 completed at a time moment somewhere between 3 and 5), or state s_5: {0, 2, {1}} (the execution of task t_1 reached over time moment 5, and hence it was discarded at time moment 5). The construction procedure continues until all possible states corresponding to the time interval [0, P) have been visited.

The transition time probability density functions z_1, z_2, z_3, z_4, and z_5 are shown in Figure 7 to the left of their corresponding states. The transition from state s_3 to state s_4 occurs at a precisely known time instant, time 3, at which a new job of task t_0 arrives. Therefore, z_4 will contain a Dirac impulse at the beginning of the corresponding PMI. The probability density function z_4 results from the superposition of $z_1 * e_1$ (because task t_1 runs in state s_1) with $z_2 * e_1$ (because task t_1 runs in state

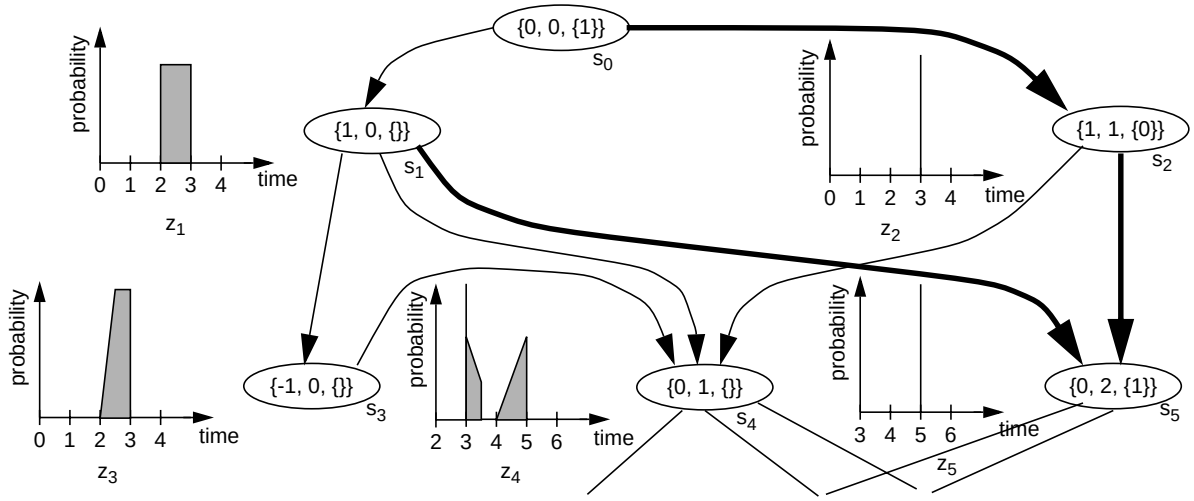


Figure 7. Stochastic process example

s_2 too) and with the aforementioned Dirac impulse over the PMI 1, i.e. over the time interval [3, 5).

The embedded process being Markovian, the probabilities of the transitions out of a state s_i are computed exclusively from the information stored in that state s_i. For example, the probability of the transition from state s_1 to state s_4 (see Figure 7) is given by the probability that the transition occurs at some time moment in the PMI of state s_4 (the interval [3, 5)). This probability is computed by integrating $z_1 * e_1$ over the interval [3, 5). The probability of a task missing its deadline is easily computed from the transition probabilities of those transitions that correspond to a job discarding (the thick arrows in Figure 7).

As it can be seen, by using the PMI approach, some process states have more than one incident arc, thus keeping the graph "narrow". This is because, as mentioned, one process state in our representation captures several possible states of a representation considering individual times (see Figure 6a).

Because the number of states grows rapidly and because each state has to store its probability density function, the memory space required to store the whole process can become prohibitively large. Our solution to mastering memory complexity is to perform the stochastic process construction and analysis simultaneously. As each arrow updates the time probability density of the state it leads to, the process has to be constructed in topological order. The result of this procedure is that the process is never stored entirely in memory but rather that a sliding window of states is used for analysis. For the example in Figure 7, the construction starts with state s_0. After its next states (s_1 and s_2) are created, their corresponding transition probabilities determined and the possible discarding probabilities accounted for, state s_0 can be removed from memory. Next, one of the states s_1 and s_2 is taken as current state, let us consider state s_1. The procedure is repeated, states s_3, s_4 and s_5 are created and state s_1 removed. At this moment, the arcs emerging from states s_2 and s_3 have not yet been created. Consequently, one would think that any of the states s_2, s_3, s_4, and s_5 can be selected for continuation of the analysis. Obviously, this is not the case, as not all the information needed in order to handle states s_4 and s_5 are computed (in particular those coming from s_2 and s_3). Thus, only states s_2 and s_3 are possible alternatives for the continuation of the analysis in topological order. In Section 5 we discuss the criteria for selection of the correct state to continue with.

5 Stochastic process construction and analysis

The analysis of the stochastic task set is performed in two phases:

1. Divide the interval [0, P) in PMIs.
2. Construct the stochastic process in topological order and analyse it.

5.1 Priority monotonicity intervals

The concept of PMI (called in their paper "state") was introduced by Zhou et al. [12] in a different context, unrelated to the construction of a stochastic process.

Let A_i denote the set of time moments in the interval [0, P) when a new job of task t_i arrives and let A denote the union of all A_i. Let D_i denote the set of absolute deadlines[1] of the jobs belonging to task t_i in the interval [0, P), and D be the union of all D_i. Consequently,

$$A_i = \{x | x = k \cdot p_i, 0 \le k < P/p_i\}$$
$$D_i = \{x | x = d_i + k \cdot p_i, 0 \le k < P/p_i\}$$

If the deadline of a certain task t_i equals its period, then $A_i = D_i$ (the time moment 0 is considered, conventionally, to be the deadline of the job arrived at time moment $-p_i$).

Let $H = A \cup D$. If H is sorted in ascending order of the time moments, then a priority monotonicity interval is the interval between two consecutive time moments in H. The last PMI is the interval between the greatest element in H and P.

The only restriction imposed on the scheduling policies accepted by our approach is that inside a PMI the ordering of tasks according to their priorities is not allowed to change. The consequence of this assumption is that the next state can be determined no matter when the currently running task completes within the PMI. All the widely used scheduling policies we are aware of (RM, EDF, FCFS, etc.) exhibit this property.

5.2 The construction and analysis algorithm

The algorithm proceeds as discussed in Section 4. An essential point is the construction of the process in topological order, which allows only parts of the states to be stored in memory at any moment.

The algorithm for the stochastic process construction is depicted in Figure 8. All states belonging to the sliding window are stored in a priority queue. The key to the process construction in topological order lies in the order in which the states are extracted from this queue. First, observe that it is impossible for an arc to lead from a state with a PMI number u to a state with a PMI number v so that

```
put first state in the queue;
while queue not empty do
   s_j = extract state from the queue;
   t_i = s_j.running; -- field r of state s_j
   distribution = convolute(e_i, z_j);
   nextstatelist = next_states(s_j);
            -- consider task dependencies!
   for each s_k ∈ nextstatelist do
     compute probability of the transition
          from s_j to s_k using distribution;
     update discarding probabilities;
     update z_k;
     if s_k is not in the queue then
       put s_k in the queue;
     end if;
   end for;
   delete state s_j;
end while;
```

Figure 8. Construction and analysis algorithm

1. Except here, whenever we use the term "deadline", we consider relative deadlines.

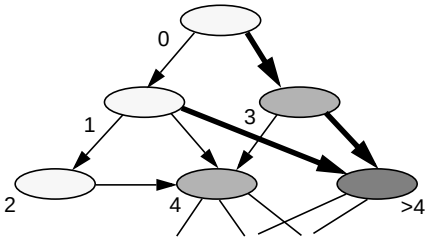


Figure 9. State selection order

v < u (there are no arcs back in time). Hence, a first criterion for selecting a state from the queue is to select the one with the smallest PMI number. A second criterion determines which state has to be selected out of those with the same PMI number. Note that inside a PMI no new job can arrive, and that the task ordering according to their priorities is unchanged. Thus, it is impossible that the next state s_k of a current state s_j would be one that contains waiting tasks of higher priority than those waiting in s_j. Hence, the second criterion reads: among states with the same PMI, one should choose the one with the waiting task of highest priority.

Figure 9 illustrates the algorithm on the example given in Section 4 (Figure 7). The shades of the states denote their PMI number. The lighter the shade, the smaller the PMI number. The numbers near the states denote the sequence in which the states are extracted from the queue and processed.

6 Experimental Results

The most computation intensive part of the analysis is the computation of the convolutions. In our implementation we used the FFTW library [6] for performing convolutions based on the Fast Fourier Transform. The number of convolutions to be performed equals the number of states of the stochastic process. The memory required for analysis is determined by the maximum number of states in the sliding window. The main factors on which the stochastic process depends are P (the least common multiple of the task periods), the number of PMIs, the number of tasks, and the task dependencies.

As the selection of the next running task is unique, given the pending jobs and the time moment, the particular scheduling policy has a reduced impact on the process size. On the other hand, the task dependencies play a significant role, as they strongly influence the set of ready tasks and by this the process size. Additionally, they generate a smaller number of PMIs as they impose a certain harmony among the task periods.

In the following, we report on three sets of experiments. The aspects of interest were the stochastic process size, as it determines the analysis execution time, and the maximum size of the sliding window, as it determines the memory space required for the analysis. All experiments were performed on an UltraSPARC 10 at 450 MHz.

In the first set of experiments we analysed the impact of the number of tasks on the process size. We considered task sets of 10 up to 19 independent tasks. P, the least common multiple of the task periods, was 360 for all task sets. We repeated the experiment four times for average

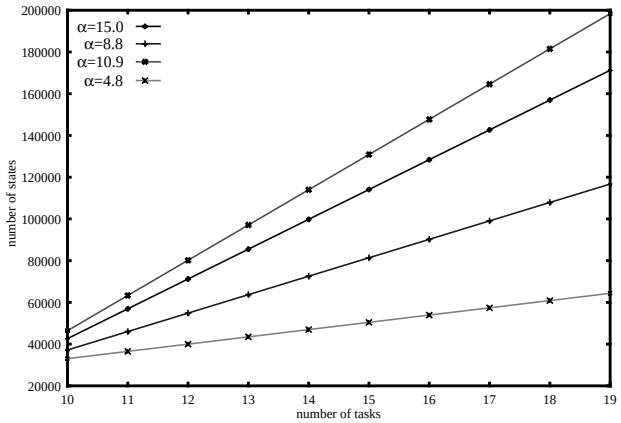


Figure 10. Experiment 1, stochastic process size

values of the task periods α = 15.0, 10.9, 8.8, and 4.8 (keeping P=360). The results are shown in Figure 10. Figure 11 depicts the maximum size of the sliding window for the same task sets. As it can be seen from the diagram, the increase, both of the process size and of the sliding window, is linear. The steepness of the curves depends on the task periods (which influence the number of PMIs). It is important to notice the big difference between the process size and the maximum number of states in the sliding window. In the case for 19 tasks, for example, the process size is between 64356 and 198356 while the dimension of the sliding window varies between 373 and 11883 (16 to 172 times smaller). The reduction factor of the sliding window compared to the process size was between 15 and 1914, considering all our experiments. Because of space limitation, for the rest of paper we will concentrate only on the process size.

In the second set of experiments we analysed the impact of the application period P (the least common multiple of the task periods) on the process size. We considered 784 sets, each of 20 independent tasks. The task periods were chosen such that P takes values in the interval [1, 5040]. Figure 12 shows the variation of the average process size with the application period.

With the third set of experiments we analysed the impact of task dependencies on the process size. A task set of 200 tasks with strong dependencies (28000 arcs) among the tasks was initially created. The application period P was 360. Then 9 new task graphs were successively

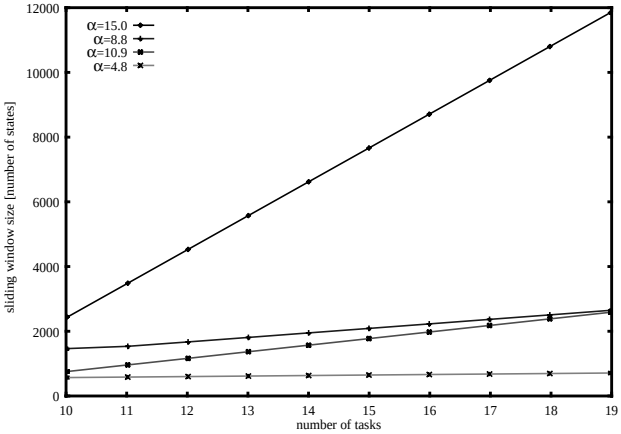


Figure 11. Experiment 1, sliding window size

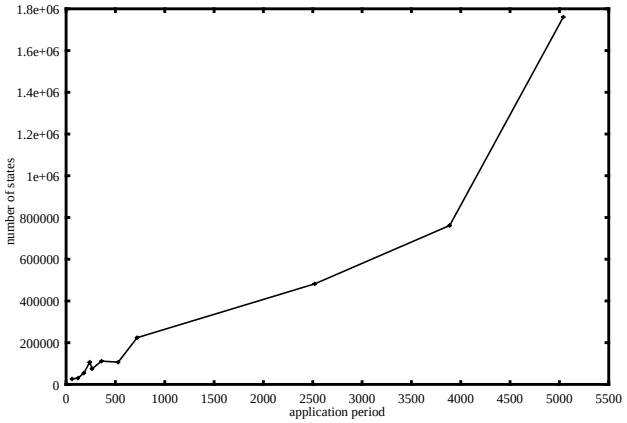


Figure 12. Experiment 2, stochastic process size

derived from the first one by uniformly removing dependencies between the tasks until we finally got a set of 200 independent tasks. The results are depicted in Figure 13 with a logarithmic scale for the y axis. The x axis represents the degree of dependencies among the tasks (0 for independent tasks, 9 for the initial task set with the highest amount of dependencies). In this set of experiments, we used the first of the two policies defined in Section 3 for handling task graphs with dependencies.

As mentioned, the execution time for the analysis algorithm strictly depends on the process size. Therefore, we showed all the results in terms of this parameter. For the set of 200 independent tasks used in the last experiment (process size 1126517) the analysis time was 745 seconds. In the case of the same 200 tasks with strong dependencies (process size 2178) the analysis took 1.4 seconds.

Finally, we considered an example from the mobile communication area. A set of 8 tasks co-operate in order to decode the digital bursts corresponding to a GSM 900 signalling channel. In this example, there are two sources of variation in execution times. One task has both data and control intensive behaviour, which can cause pipeline hazards on the deeply pipelined DSP it runs on. Its execution time probability density is derived from the input data streams and measurements. Another task will finally implement a deciphering unit. Due to the lack of knowledge about the deciphering algorithm (its specification is not publicly available), the deciphering task execution

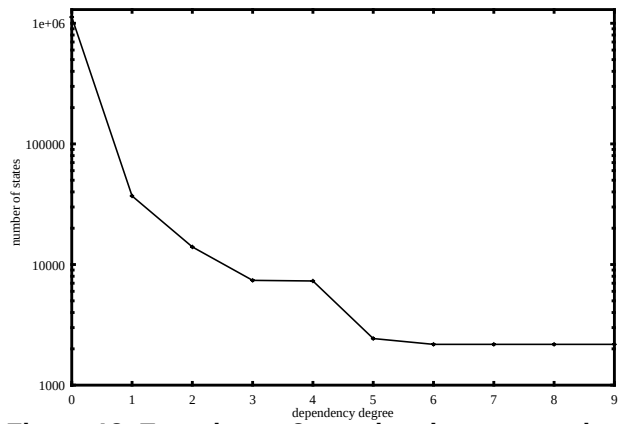


Figure 13. Experiment 3, stochastic process size

time is considered to be uniformly distributed between an upper and a lower bound. When two channels are scheduled on the same DSP, the ratio of missed deadlines is 0 (all deadlines are met). Considering three channels assigned to the same processor, the analysis produced a ratio of missed deadlines, which was below the one enforced by the required QoS. It is important to note that using a hard real-time model with WCET, the system with three channels would result as unschedulable on the selected DSP. The underlying stochastic process for the three channels had 130 nodes and its analysis took 0.01 seconds. The small number of nodes is caused by the strong harmony among the task periods, imposed by the GSM standard.

7 Conclusions

This work proposes a method for performance analysis of task sets with probabilistically distributed execution times. The tasks are scheduled according to an arbitrary scheduling policy. The method is based on the construction of the underlying stochastic process and the analysis of this process. The stochastic process is constructed and analysed in a memory- and time-efficient way making the method applicable to large task sets. Experimental results show a very good scaling of the algorithm both in terms of memory space and execution time.

As a future work, we intend to extend our approach in order to handle sets of tasks distributed over multiprocessor systems.

References

[1] L. Abeni, G. Butazzo, *"Integrating Multimedia Applications in Hard Real-Time Systems"*, Proc. of the Real-Time Systems Symposium, 1998, pp. 4–13

[2] A. Atlas, A. Bestavros, *"Statistical Rate Monotonic Scheduling"*, Proc. of the Real-Time Systems Symposium, 1998, pp. 123–132

[3] N. C. Audsley, A. Burns, R. I. Davies, K. W. Tindell, A. J. Wellings, *"Fixed Priority Pre-emptive Scheduling: An Historical Perspective"*, Journal of Real-Time Systems, v8, 1995, pp. 173–198

[4] G. De Veciana, M. Jacome, J.H. Guo, *"Assessing Probabilistic Timing Constraints on System Performance"*, Jour. of Design Autom. for Emb. Systems, v5, 2000, pp. 61–81

[5] C. J. Fidge, *"Real-Time Schedulability Tests for Preemptive Multitasking"*, Journal of Real-Time Systems, v14, 1998, pp. 61–93

[6] M. Frigo, S. G. Johnson, *"FFTW: An Adaptive Software Architecture for the FFT"*, Proc. Intl. Conf. of Acoustics, Speech and Signal Processing, 1998, v. 3, p. 1381

[7] A. Kalavade, P. Moghe, *"A Tool for Performance Estimation for Networked Embedded Systems"*, Proc. of the Design Automation Conference, 1998, pp. 257–262

[8] J. P. Lehoczky, *"Real-Time Queueing Theory"*, Proc. of the Real-Time Systems Symposium, 1996, pp. 186–195

[9] M. Spuri, G. Butazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems", Journal of Real-Time Systems, v. 10, no. 2, 1996, pp. 1979–2012

[10] J. A. Stankovic, M. Spuri, M. Di Natale, G. Butazzo, *"Implications of Classical Scheduling Results for Real-Time Systems"*, IEEE Computer, June 1995, pp. 16–25

[11] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, J. W.-S. Liu, *"Probabilistic Performance Guarantee for Real-Time Tasks with Varying Computation Times"*, Proc. of IEEE Real-Time Techn. and Applic. Symp., 1995

[12] T. Zhou, X. Hu, E. H.-M. Sha, *"A probabilistic performance metric for Real-Time System Design"*, Proc. of the Hardw/Softw. Co-Design Symposium, 1999, pp. 90–94

On Energy Reduction for Hard Real-Time Tasks with Stochastic Execution Times

Flavius Gruian
Department of Computer Science, Lund University, Box 118,
S-221 00 Lund, Sweden
<Flavius.Gruian@cs.lth.se>

Abstract — This paper addresses energy reduction in hard real-time systems, containing independent tasks running on dynamic voltage supply processors. Since tasks with probabilistic run times are more realistic, we focus on systems containing such tasks. We show that both off-line and on-line scheduling may contribute in reducing the energy consumption, if the probabilistic behavior is taken into consideration. Stochastic data can be used in deriving schedules both for task sets and individual tasks. We also present a task-level scheduling method which reduces the average energy consumption independent of the other tasks in the system. The experimental results, although in an incipient phase, are promising.

I. Introduction

Low energy consumption is today an important design requirement for digital systems, with impact on operating time, on system cost, and, of no lesser importance, on the environment. Reducing power and energy dissipation in digital systems has long been addressed by several research groups. With the advent of dynamic voltage supply (DVS) processors [6,20], highly flexible systems can be designed, while still taking advantage of supply voltage scaling to reduce the energy consumption. Since the supply voltage has a direct impact on processor speed, classic task scheduling and supply voltage selection have to be addressed together. Scheduling offers thus yet another level of possibilities for achieving energy/power efficient systems, especially when the system architecture is fixed or the system exhibits a very dynamic behavior. For such dynamic systems, various power management techniques exist and are reviewed for example in [4,5]. Yet, these mainly target soft real-time systems, where deadlines can be missed if the Quality of Service is kept. Several scheduling techniques for soft real-time tasks, running on DVS processors have already been described [7,8,9]. Energy reductions can be achieved even in hard real-time systems, where no deadline can be missed, as shown in [10,11,12,13,14]. In this paper, we focus on hard real-time scheduling techniques, where all deadlines have to be met.

Task level voltage scheduling decisions can reduce even further the energy consumption. Some of these intra-task scheduling methods use several re-scheduling points inside a task, and are usually compiler assisted [16,17,18]. Alternatively, fixing the schedule before the task starts executing as in [11,14,19] eliminates the internal scheduling overhead, but with possible affects on energy reduction. Statistics can be used to take full advantage of the dynamic behavior of the system, both at task level [18] and at task-set level [10]. We show the importance of employing stochastic data in deriving efficient voltage schedules at both task and task-set levels.

The rest of the paper is organized as follows. Section II presents the scheduling alternatives taken at task-set level, for three cases, increasing in generality. A task-level scheduling method for low average energy is described in section III. We give examples and some experimental results throughout the paper, in specific sub-sections, hoping to make our point even clearer. Our conclusions and future work make the subject of section IV.

II. Task-Set Level Scheduling Decisions

In this paper we address only independent tasks running on a single dynamic supply voltage processor. Each of the N tasks in the task set is defined by a 4-tuple $\tau_i = (X_i, C_i, T_i, D_i)$ composed of the execution pattern, worst case execution time, period, and deadline for task τ_i. The execution pattern X_i is a random variable describing the number of clock cycles required to execute the task. We will denote by $\overline{X_i}$ the expected value of the random variable X_i. At the fastest processor speed (clock frequency), the maximal value for X_i is C_i.

The focus of this discussion is on the influence of considering tasks with probabilistic execution pattern on the task-set level scheduling strategies. The main goal of all the strategies presented here is reducing the energy consumption. We show that scheduling policies considering only the tasks WCET can be improved when stochastic data is used. We compute therefore the following three measures. The worst case execution speed, s^{WCE}, which does not consider the stochastic behavior of the tasks. The expected ideal speed, s^{ideal}, which assumes that all tasks will always run at their expected time and can take full advantage of this knowledge. And, finally, the average real speed, s, which is a estimate of the realistic case behavior, when the actual execution times are not known a priori. We can use these three measures to compute the energy consumptions for the worst, ideal, and average cases. The energy consumption for a given time interval $[t0, t1]$ is defined as the integral of power over time. Assuming a quadratic dependency between power and speed, as in [10], we obtain the energy as:

$$E = \int_{t0}^{t1} P(s(t))dt \cong \int_{t0}^{t1} (s(t))^2 dt \qquad (1)$$

What remains to be done is to find out the processor speeds in every time moment.

Because of the dynamic behavior of the system, we need to address not only off-line scheduling methods, but also on-line re-scheduling policies. Depending on the generality of the problem, the complexity of the off-line and on-line policies varies.

We address three cases, with increasing generality. First we consider task sets with tasks having the same period and deadline. Next we analyze tasks having the same period, but different deadlines. Finally, we consider task sets which have both different periods and possibly different deadlines.

A. Unique Period, Unique Deadline

This is the simplest possible case, where the set of tasks has to finish before a certain deadline, in no particular order. Formally we consider $T_i = D_i = T_j = D_j = A, \forall(\tau_i, \tau_j)$. When the actual execution time of each task is known beforehand, the optimal schedule from the energy point of view is given by uniformly stretching the tasks to exactly meet the deadline [14]. The worst case processor speed is then computed as:

$$s^{WCE} = \left(\sum_{1 \le i \le N} C_i\right) / A \qquad (2)$$

while the ideal expected speed:

$$s^{ideal} = \left(\sum_{1 \le i \le N} \overline{X}_i\right) / A . \qquad (3)$$

Using only off-line decisions, one can always guarantee the deadlines only by using the worst case speed. An on-line re-scheduling algorithm may be used every time a task completes its execution. In each of these scheduling points, a new expected optimal processor speed can be computed as:

$$s_j = \left(\sum_{j \le i \le N} C_i\right) / \left(A - \sum_{1 \le k < j} (\overline{X}_k / s_k)\right) \qquad (4)$$

Note that the execution order of the tasks in this case becomes important. Tasks which exhibit a large discrepancy between their worst case execution and their expected (average) execution should be executed first. In this way, the tasks executing later know if they can use up more processor time. To eliminate the biggest uncertainties early, the tasks should therefore be executed in an order according to the following priorities:

$$p_i = 1/(C_i - \overline{X}_i) \qquad (5).$$

Having decided on the schedule, we can now compute the energy consumption for each of the cases presented here. Note that the processor speed is constant on intervals, and for the WCE and ideal case there is a unique speed:

$$E^{WCE} = (s^{WCE})^2 \sum_{1 \le i \le N} (\overline{X}_i / s^{WCE}) = \left(\left(\sum_{1 \le i \le N} C_i\right) \cdot \left(\sum_{1 \le i \le N} \overline{X}_i\right)\right) / A \qquad (6)$$

$$E^{ideal} = A(s^{ideal})^2 = \left(\sum_{1 \le i \le N} \overline{X}_i\right)^2 / A \qquad (7)$$

$$E^{real} = \sum_{1 \le i \le N} \frac{\overline{X}_i}{s_i} \cdot s_i^2 = \sum_{1 \le i \le N} \overline{X}_i \cdot s_i \qquad (8)$$

Note that the last energy value is dependent on the task ordering. This is were one can reduce the energy consumption by using an optimal ordering.

Consider a simple task set, composed of three tasks $\{\tau_1 = (X_1, 20, 100, 100),\ \overline{X_1} = 16;\ \tau_2 = (X_2, 30, 100, 100),\ \overline{X_2} = 20;\ \tau_3 = (X_3, 40, 100, 100),\ \overline{X_3} = 32\}$. The processor speeds for four cases, plus the energy consumptions are gathered in Table 1. The first two columns refer to the worst and the ideal cases. The "worst case" decides an off-line speed such that the tasks will always meet their deadlines, and uses onöy that speed at runtime. The "ideal case" assumes that the exact execution times are known beforehand and decides an optimal processor speed for each period. The real case implies using a runtime re-scheduling policy. The processor speed is recomputed each time a task finishes execution, that is why we give three speed values for the real case. The "best order" in Table 1 is the order given by the priorities computed as in (5). The "reverse best" is when the task assume an inverse order. From the table we can deduce that an on-line strategy reduces further the energy consumption. Moreover, a good ordering also contributes to energy reduction.

Table 1: An example of case A

$\{\tau_1, \tau_2, \tau_3\}$	WCE	ideal	real expected	
			best order: 2, 3, 1	reverse best: 1, 3, 2
speed, s	0.90	0.68	0.90, 0.77, 0.55	0.90, 0.85, 0.67
normalized energy, E	1.324	1	1.114	1.182

B. Unique Period, Different Deadlines

The problem becomes more complex when the deadlines for the tasks differ: $T_i = T_j = A$, $D_i \ne D_j \le A$, $\forall(\tau_i, \tau_j)$. A deadline monotonic scheduling strategy would, in this case, guarantee feasible schedules up to full processor utilization. Considering the tasks ordered according to their deadlines, the processor speed for each task can be computed as in [10]:

$$s_j^{WCE} = \sum_{j \le i \le N} \left(C_i / \left(D_i - \sum_{1 \le k < j} (C_k / s_k)\right)\right) \qquad (9)$$

and the ideal speed:

$$s_j^{ideal} = \sum_{j \le i \le N} \left(\overline{X}_i / \left(D_i - \sum_{1 \le k < j} (\overline{X}_k / s_k)\right)\right) \qquad (10).$$

As for the on-line scheduling method, the processor speed is recomputed whenever a task finishes execution. The time moment when task j finishes is:

$$t_j = \sum_{1 \le k \le j} (\overline{X}_k / s_k) \qquad (11)$$

The following speed is then re-computed for the next task:

$$s_{j+1} = \sum_{j < i \le N} (C_i / (D_i - t_j)) \qquad (12)$$

As in the previous case, the lowest average energy consumption would be achieved if the tasks with big uncertainties execute first. On the other hand, the deadlines have to be met. With this in mind, we present here an off-line preemptive scheduling strategy which attempts to obtain a low average energy consumption. The algorithm, presented in Fig. 1, uses an initial step in which WCE processor speeds are computed. The actual algorithm assigns time intervals for each task or task part, starting from the lowest priority task. We always schedule

```
begin procedure OfflinePreemptiveSchedule
empty WorkList
  forall deadlines Di, longest..shortest do
    time := Di, empty NextWorkList
    * put all tasks with deadline Di in WorkList
    * order WorkList descending
      using priorities pj := sj/(Cj - Xj)
    forall tasks τk in WorkList do
      if Ck/sk > (time - Di-1) then
        * split τk:
          NextWorkList.add(task with
            C := Ck - (time - Di-1)*sk,
            D := Di-1, X := min(C, Xk))
          Also update Ck := time - Di-1
      end if
      * schedule τk between [time - Ck/sk, time)
      time := time - Ck/sk
    end forall
    * make NextWorkList the new WorkList
  end forall
end procedure
```

Fig. 1. Off-line scheduling algorithm for reduced energy in the case of task sets with unique period and different deadlines

tasks in bursts, between two consecutive deadlines, called scheduling intervals. The tasks having the same deadline are sorted according to priorities first introduced in sub-section A. The tasks exhibiting the smallest discrepancy between their WCET and their expected execution time are scheduled last. Whenever a task does not fit entirely in the current scheduling interval, it is split in two parts such that one of them fits. The other part is treated as a new task, which has to be completed before the current scheduling interval. The on-line re-scheduling algorithm is the same as the one described previously, by equation (12).

Having decided on the schedule, we can now compute the expected energy consumption for all the cases presented here. As in the previous case, the processor speed is constant on intervals. For brevity we will not expand the expressions: $E^{WCE} = \sum_{1 \le i \le N} \overline{X_i} \cdot s_i^{WCE}$, $E^{ideal} = \sum_{1 \le i \le N} \overline{X_i} \cdot s_i^{ideal}$, $E^{real} = \sum_{1 \le i \le M} \overline{X_i} \cdot s_i$.

Note that in the last case the number of intervals with constant speed can increase to *M* as a result of our off-line preemptive scheduling algorithm.

C. Different Periods

Dealing with tasks which have different periods is an even more complex problem. Formally, now $T_i \neq T_j$, $\forall(\tau_i, \tau_j)$. There are several approaches to scheduling in this situation. One could use task hyperperiods, as in [11], and practically return to the case described in sub-section B. Yet, for certain task sets the complexity of the problem, given by the number of instances that have to be analyzed, becomes huge. In the following we present an alternative approach, developed on the fixed priority (rate or deadline monotonic) scheduling framework. Our method consists as before, from an off-line step and an on-line policy.

The off-line scheduling step. The scheduling condition proposed by Liu and Layland [1] is a sufficient one and covers the worst possible case for the task group characteristics. Yet, an exact analysis as proposed in [2] may reveal possibilities for stretching tasks and still keeping the deadlines. Based on this, [12] describes a method to compute the minimal required frequency (speed) for a task set. In similar way, we go further and compute the minimal achievable processor speed $\{s_i\}_{1 \le i \le n}$ for each task τ_i in the task group $\{\tau_i\}_{1 \le i \le n}$. We consider that the tasks in the group are indexed according to their priority.

We compute the speeds in an iterative manner, from the higher to the lower priority tasks. An index *q* points to the latest task which has been assigned a speed. Initially, $q = 0$. Each of the tasks $\tau_i, q < i \le n$ has to be executed before one of its scheduling points S_i as defined in [2]: $S_i = \{kT_j \mid 1 \le j \le i; 1 \le k \le \lfloor T_i/T_j \rfloor\}$, if $T_i = D_i$. If $T_i \neq D_i$, we only need to change the set of scheduling points according to $S_i' = \{t \mid (t \in S_i) \wedge (t < D_i)\} \cup \{D_i\}$. For each of this scheduling points $S_{ij} \in S_i$, task τ_i exactly meets its deadline if:

$$\sum_{1 \le r \le q} \frac{1}{s_r} C_r \cdot \left\lceil \frac{S_{ij}}{T_r} \right\rceil + \frac{1}{s_{ij}} \cdot \sum_{q < p \le i} C_p \cdot \left\lceil \frac{S_{ij}}{T_p} \right\rceil = S_{ij} \qquad (13)$$

Note that for the tasks which already have assigned a speed we used that one, s_r, while for the rest of the tasks we assumed they will all use the same and yet to be computed speed, s_{ij}, which is dependent on the scheduling point. For the task τ_i the best scheduling choice, from the energy point of view, is the smallest of its s_{ij}. At the same time, from (13), this has to be the equal for all tasks $\tau_i, q < i \le n$. There is a task with index *m* for which its lowest speed is the largest among all other tasks. Note that this in not necessarily the last task, *n*. If $q = 0$, this task sets the minimal clock frequency as computed in [12]. Obtaining the index *m*, all tasks between *q* and *m* will use the same speed as *m*. With this an iteration of the algorithm for finding the task speeds is complete. The next iteration then proceeds for $q = m$. Finally the process ends when *q* reaches *n*, meaning all tasks have been given their own off-line speed.

A numerical example is given in Table 2. Note that tasks 3 and 4 can use a lower speed than 1 and 2, while 5 has the lowest processor speed. As a result of slowing down the tasks, the processor utilization changes from 0.687 to 0.994.

Table 2: Numerical Example for Off-line Scheduling

Task τ			Off-line Speed s	
No.	WCET (C)	Period (T)	value	iterations needed
1	1	5	0.70	1
2	5	11	0.70	1
3	1	45	0.56	2
4	1	130	0.56	2
5	1	370	0.42	3

The on-line strategy. During runtime we exploit the stochastic nature of the system. It is important to use the variations in execution length of the various task instances to be able to execute at lower processor speed and, thus, consume less energy. In [12] the only situation when a task executes slower is when

it is the only one running and has enough time until the next task instance arrives. In all other situations tasks are executed at the speed dictated by the off-line analysis. In [16] tasks are slowed down to cover their WCET at runtime, independent of other tasks, using several checking/re-scheduling points during a task instance. Our method is perhaps most resemblant to the optimal scheduling method OPASTS presented in [11]. Yet, OPASTS performs analysis over task hyperperiods, which may lead to working on a huge number of task instances for certain task sets. Our method, briefly described next, keeps a low and the same computational complexity, regardless of the task set.

An early finishing task may pass on its unused processor time for any of the tasks executing next. But this time slack can not be used by any task at any time since deadlines have to be met. We solve this by considering several levels of slacks, with different priorities, as in the slack stealing algorithm [3]. The slack in each level is a cumulative value, the sum of the unused processor times remaining from the tasks with higher priority. Whenever a task finishes its execution early, it contributes to the corresponding slack level with the amount of unused time. This is then distributed to the lower priority tasks present in the system, according to their expected execution time. The slack distribution is performed such that the highest processor speeds are reduced, lowering the energy consumption. We present a more detailed description of the on-line strategy in [15]. We also give a proof that our scheduling method using slack levels meets all the deadlines.

III. Task Level Scheduling Decisions

Task-level voltage scheduling has captured the attention of the research community rather recently [19]. Fine grain scheduling, where several re-scheduling points are used inside a task were presented in [16,18]. In [18] statistical data is used to improve the task level schedule, by slowing down different regions of a task according to their average execution time. Our approach produces voltage schedules only when a task starts executing, while using stochastic data more aggressively.

In our model a task τ_i can be executed in phases, at different available voltages, depending on its allowed execution time A_i. The ideal case states that the most energy is saved when the processor uses the voltage for which the task exactly covers its allowed execution time. This corresponds to an ideal voltage which may not overlap with the available voltages. A close to optimal solution is to execute the task in two phases at two of the available voltages. These two voltages are the ones bounding the ideal voltage [14,19].

An important observation is that tasks may finish, and in many cases do finish, before their worst case execution time (WCET). Therefore it makes sense to execute first at a low voltage and accelerate the execution, instead of executing at high voltage first and decelerate. In this manner, if a task instance is not the worst case, one skips executing high voltage regions.

In the following we will distinguish between three modes of execution for a task, as depicted in Fig. 2. The ideal case (mode 1) is when the actual execution pattern (the number of clock cycles) becomes known when the task arrives. We can stretch then the actual execution time of the task to exactly fill the allowed time. This mode requires rather accurate execution pattern estimates, depending on the input data, and therefore is rarely achievable in practice. The second mode (mode 2) is the WCE stretching - the voltage schedule for the task is determined as if the task will exhibit its worst case behavior. These first two modes use at most two voltage regions, and therefore at most one DC-DC switch. The third mode (mode 3), described in more detail next, uses stochastic data to build a multiple voltage schedule. The purpose for using stochastic data is to minimize the average case energy consumption. Note that the voltage schedules in all these three modes are decided at a task instance arrival. Unlike in [16,17] no rescheduling is done while the task is executing. The only overhead during task execution is the one given by the changes in the supply voltage.

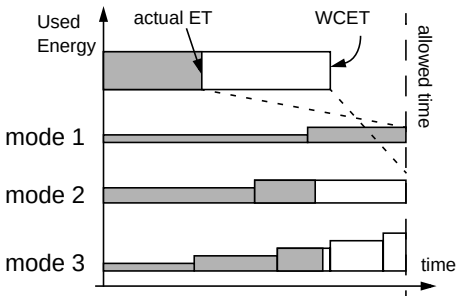


Fig. 2. Voltage scheduling modes for tasks: 1) ideal schedule, 2) WCET oriented schedule, 3) stochastic schedule.

The stochastic voltage schedule (mode 3 in Fig. 2) for a task is obtained using the probability distribution of the execution pattern for a task (the number of clock cycles used). This probability distribution can be obtained off-line, via simulation, or built and improved at runtime. The actual voltage schedule is obtained by minimizing the expected value of the energy consumption. We present adetailed description of computing such a schedule in [15].

Two examples of stochastic voltage schedules are given in Fig. 3. We assumed a normal probability distribution with the mean of 70 cycles, and standard deviation of 10. WCE is 100. Assuming we only have four available clock frequencies f, f/2, f/3, and f/4, we give two voltage schedules obtained for two different values of the allowed execution time. The schedules are given in number of clock cycles executed at each available frequency. The allowed execution time is reported in percentage of the time needed for executing the worst case behavior at the highest clock frequency (f).

Next we present an experiment that examines the energy gains of using a stochastic voltage schedule at task level. For this we considered a single task with execution time varying between a best case (BCE) and a worst case (WCE) according to a normal distribution. All distributions have the mean (BCE+WCE)/2 and standard deviation (WCE-BCE)/6. For a several cases ranging from highly flexible execution time (BCE/WCE is 0.1) to almost fixed (BCE/WCE is 0.9) we built stochastic schedules for a range of allowed execution times (from WCE to 3x WCE). We assumed that our processor has 9 different voltage levels, equally distributed between f and f/3.

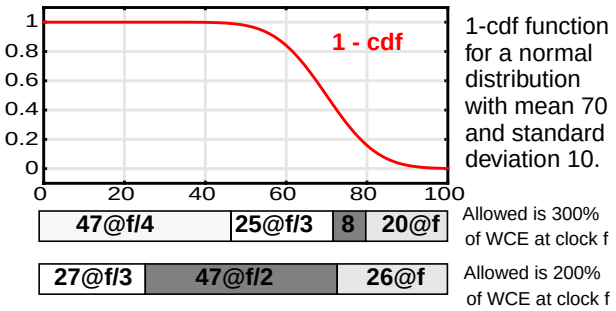


Fig. 3. Examples of stochastic voltage schedules for a task with normal distribution execution time and worst case behavior of 100 cycles

For a large number of task instances generated according to the given distribution we computed both the energy of the stochastic schedule (mode 3 in Fig. 2) and the WCE-stretch schedule (mode 2 in Fig. 2). We depict in Fig. 4 the average energy consumption of the stochastic schedule as a part of the WCE-stretch schedule. Note that when the allowed time approaches either WCE or 3-times WCE, the energy consumptions become equal. The lowest possible clock frequency is f/3 which anyway means 3-times WCE, so there is no better schedule for these cases. On the other hand when the allowed time closes WCE, there is no other way but to use the fastest clock. Somewhere between the slowest and the fastest frequencies (Allowed/WCE = 2) is the largest energy gain since the stochastic schedule can use the whole spectrum of available frequencies. Note that the energy gains become more important when the task execution time varies much (BCE/WCE closes 0.1). Notice also that WCE-stretch already gains very much energy compared to the non-scaling case. For example when the allowed time is twice the WCE, the WCE-stretch energy is around 25% of the no-scaling energy. A stochastic approach contributes even more to these gains, as the figure shows.

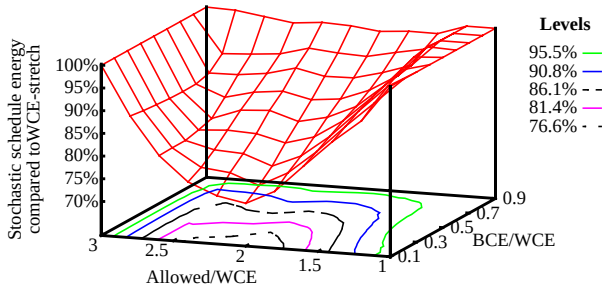


Fig. 4. The average energy consumption of a stochastic voltage schedule compared to the energy consumption of a WCE- stretch schedule.

IV. Conclusions and Future Work

In this paper we addressed scheduling methods targeting energy reduction in hard real-time systems containing dynamic supply voltage processors. Both task-set and individual task level scheduling decisions were described. For task sets we started from the simple case of tasks having the same unique deadline and period. We then generalized the problem by considering different deadlines, and then different periods. In all cases we showed that more efficient schedules, from the energy point of view, can be derived when stochastic data is taken into consideration. We also presented a task level stochastic voltage schedule which is able to reduce the average energy consumption of a task even more, compared to classic approaches.

As future work, we plan to do extensive experiments using all the scheduling methods presented here, focusing more on the task-set level policies. We also want to perform a thorough theoretical examination of the most general case and eventually come up with realistic lower bounds for the energy consumption of systems containing stochastic task sets.

References

[1] C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprograming in a hard real time environment," *JACM* 20 (1), 1973, pp. 46-61.

[2] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," *Proc. of the Real-Time Systems Symposium* 1989, pp. 166-171.

[3] J. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," *Proc. of the Real-Time Systems Symposium* 1992, pp. 110-123.

[4] L. Benini and G. DeMicheli, "System-level power optimization: techniques and tools," *ACM Trans. on Design Automation of Electronic Systems*, No. 2, Vol. 5, April 2000, pp. 115-192.

[5] Pedram M., "Power optimization and management in embedded systems," *Proc. of ASP-DAC 2001*, pp. 239-244.

[6] K. Suzuki, S. Mita, T. Fujita, F. Yamane, F. Sano, A. Chiba, Y. Watanabe, K. Matsuda, T. Maeda, and T. Kuroda, "A 300MIPS/W RISC core processor with variable supply-voltage scheme in variable threshold-voltage CMOS," *Proc. of the IEEE Custom Integrated Circuits Conference 1997*, pp. 587-590.

[7] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms,"*Proc. of the '98 ISLPED*, pp 76-81.

[8] A. Chnadrakasan, V. Gutnik, and T. Xanthopoulos, "Data driven signal processing: an approach for energy efficient computing," *Proc. of the '96 ISLPED*, pp. 347-352.

[9] M. Weiser, B. Welch, A Demers, and S. Shenker, "Scheduling for reduced CPU energy," *Proc. of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[10] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced CPU energy," *Proc. of the 36th Symposium on Foundations of Computer Science*, pp. 374-382, 1995.

[11] I. Hong, M. Potkonjak, and M.B. Srivastava, "On-line scheduling of hard real-time tasks on variable voltage processor," *Digest of Technical Papers of the 1998 ICCAD*, pp. 653-656.

[12] Y. Shin and K. Choi, "Power conscious fixed priority scheduling for hard real-time systems," *Proc. of the 36th DAC, 1999*, pp. 134-139.

[13] Y.-H. Lee and C.M. Krishna, "Voltage-clock scaling for low energy consumption in real-time embedded systems," *Proc. of the 6th International Conference on Real-Time Computing Systems and Applications*, 1999, pp. 272-279.

[14] F. Gruian and K. Kuchcinski, "LEneS: task scheduling for low-energy systems using variable voltage processors," *Proc. of ASP-DAC2001*, pp. 449-455.

[15] F. Gruian, "Hard real-time scheduling using stochastic data and DVS Processors," submitted for review.

[16] S. Lee and T. Sakurai, "Run-time voltage hopping for low-power real-time systems," *Proc. of the 37th DAC*, 2000, pp. 806-809.

[17] D. Shin, J. Kim, and S. Lee, "Intra-task voltage scheduling for low-energy hard real-time applications," *Special Issue of IEEE Design and Test of Computers*, October 2000.

[18] D. Mossé, H. Aydin, B. Childers, and R. Melhem, "Compiler-assisted dynamic power-aware scheduling for real-time applications," *Worksop on Compilers and Operating Systems for Low-Power*, October 2000.

[19] T. Ishihara,H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," *Proc. of the '98 ISLPED*, pp 197-202.

[20] http://www.transmeta.com

Minimizing System Modification in an Incremental Design Approach

Paul Pop, Petru Eles, Traian Pop, Zebo Peng
Dept. of Computer and Information Science, Linköping University
{paupo, petel, trapo, zebpe}@ida.liu.se

ABSTRACT

In this paper we present an approach to mapping and scheduling of distributed embedded systems for hard real-time applications, aiming at minimizing the system modification cost. We consider an incremental design process that starts from an already existing system running a set of applications. We are interested to implement new functionality so that the already running applications are disturbed as little as possible and there is a good chance that, later, new functionality can easily be added to the resulted system. The mapping and scheduling problem are considered in the context of a realistic communication model based on a TDMA protocol.

Keywords*: design space exploration, design reuse, distributed real-time systems, process mapping and scheduling, methodology.*

1. INTRODUCTION

Distributed embedded systems with multiple processing elements are becoming common in various application areas [4]. In [12], for example, allocation of processing elements, as well as process mapping and scheduling for distributed systems are formulated as a mixed integer linear programming (MILP) problem. A disadvantage of this approach is the complexity of solving the MILP problem. Therefore, alternative problem formulations and solutions based on efficient heuristics have been proposed [1, 2, 8, 14].

Although much of the above work is dedicated to specific aspects of distributed systems, researchers have often ignored or very much simplified issues concerning the communication infrastructure. One notable exception is [13], in which system synthesis is discussed in the context of a distributed architecture based on arbitrated busses. Many efforts dedicated to communication synthesis have concentrated on the synthesis support for the communication infrastructure but without considering hard real-time constraints and system level scheduling aspects [6, 10, 9].

Another characteristic of research efforts concerning the codesign of embedded systems is that authors concentrate on the design, from scratch, of a new system optimized for a particular application. For many application areas, however, such a situation is extremely uncommon and only rarely appears in design practice. It is much more likely that one has to start from an already existing system running a certain application and the design problem is to implement new functionality (including also upgrades to the existing one) on this system. In such a context it is very important to make as few as possible modifications to the already running applications. The main reason for this is to avoid unnecessarily large design and testing times. Performing modifications on the (potentially large) existing applications increases design time and, even more, testing time (instead of only testing the newly implemented functionality, the old application, or at least a part of it, has also to be retested). However, this is not the only aspect to be considered. Such an incremental design process, in which a design is periodically upgraded with new features, is going through several iterations. Therefore, after new functionality has been implemented, the resulting system has to be structured such that additional functionality, later to be mapped, can easily be accommodated.

We consider mapping and scheduling for hard real-time embedded systems in the context of a realistic communication model. Because our focus is on hard real-time safety critical systems, communication is based on a time division multiple access (TDMA) protocol as recommended for applications in areas like, for example, automotive electronics [7]. For the same reason we use a non-preemptive static task scheduling scheme.

In this paper, we have considered the design of distributed embedded systems in the context of an incremental design process as outlined above. This implies that we perform mapping and scheduling of new functionality so that certain design constraints are satisfied and:

a. already running applications are disturbed as little as possible;
b. there is a good chance that new functionality can, later, easily be mapped on the resulted system.

In [11] we have discussed an incremental design strategy which excludes any modifications on already running applications. In this paper we extend our approach in the sense that remapping and scheduling of currently implemented applications are allowed, if they are needed in order to accommodate the new functionality. In this context, we propose a heuristic which finds the set of old applications which have to be remapped together with the new one such that the disturbance on the running system (expressed as the total cost implied by the modifications) is minimized. Once this set of applications has been determined, mapping and scheduling is performed according to the requirements stated above.

Supporting such a design process is of critical importance for current and future industrial practice, as the time interval between successive generations of a product is continuously decreasing, while the complexity due to increased sophistication of new functionality is growing rapidly.

The paper is divided into 6 sections. The next section presents some preliminary discussion. Section 3 introduces the detailed problem formulation and the quality metrics we have defined. Our mapping and scheduling strategies are outlined in Section 4, and the experimental results are presented in Section 5. The last section presents our conclusions.

2. PRELIMINARIES

2.1 System Architecture

We consider architectures consisting of processing nodes connected by a broadcast communication channel. Communication between nodes is based on a TDMA protocol such as the TTP [7] which integrates a set of services necessary for fault-tolerant real-time systems. The communication channel is a broadcast channel, so a message sent by a node is received by all the other nodes. Each node N_i can transmit only during a predetermined time interval, the so called TDMA slot S_i (Figure 1). In such a slot, a node can send several messages packaged in a frame. A sequence of slots corresponding to all the nodes in the architecture is called a TDMA round. A node can have only one slot in a TDMA round. Several TDMA rounds can be combined together in a cycle that is repeated periodically.

We have designed a software architecture which runs on the CPU in each node, and which has a real-time kernel as its main component. Each kernel has a schedule table that contains all the information needed to take decisions on activation of processes and transmission of messages, based on the current value of time [3].

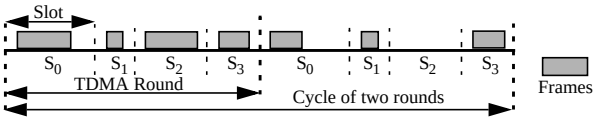


Figure 1. Buss Access Scheme

2.2 The Process Graph

As an abstract model for system representation we use a directed, acyclic, polar graph $G(V, E)$. Each node $P_i \in V$ represents one process. 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. A process can be activated after all its inputs have arrived and it issues its outputs when it terminates. Once activated, a process executes until it completes. Each process graph G is characterized by its period T_G and its deadline $D_G \leq T_G$. The functionality of an application is described as a set of process graphs.

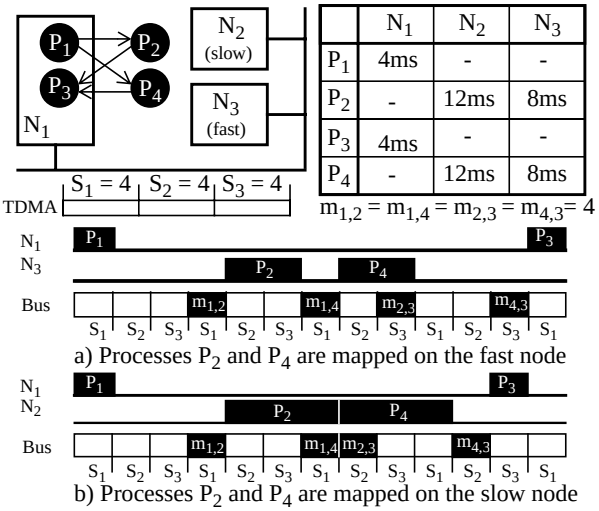
2.3 Application Mapping and Scheduling

Considering a system architecture like the one presented in section 2.1, the mapping of a process graph $G(V, E)$ is given by a function $M: V \rightarrow PE$, where $PE=\{N_1, N_2, .., N_{npe}\}$ is the set of nodes (processing elements). For a process $P_i \in V$, $M(P_i)$ is the node to which P_i is assigned for execution. Each process P_i can potentially be mapped on several nodes. Let $N_{Pi} \subseteq PE$ be the set of nodes to which P_i can potentially be mapped. For each $N_i \in N_{Pi}$, we know the worst case execution time $t_{P_i}^{Ni}$ of process P_i, when executed on N_i.

In order to implement an application, represented as a set of process graphs, the designer has to map the processes to the system nodes and to derive a schedule such that all deadlines are satisfied. We first illustrate some of the problems related to mapping and scheduling, in the context of a system based on a TDMA communication protocol, before going on to explore further aspects specific to an incremental design approach.

Let us consider the example in Figure 2 where we want to map an application consisting of four processes P_1 to P_4, with a period and deadline of 50 ms. The architecture is composed of three nodes that communicate according to a TDMA protocol, such that N_i transmits in slot S_i. According to the specification, processes P_1 and P_3 are constrained to node N_1, while P_2 and P_4 can be mapped on nodes N_2 or N_3, but not N_1. The worst case execution times of processes on each potential node, the size $m_{i,j}$ of the messages passed between P_i and P_j, and the sequence and size of TDMA slots, are presented in Figure 2 (message and slot sizes are in time units).

In [3] we have shown that by considering the communication protocol during scheduling, significant improvements can be made to the schedule quality. The same holds true in the case of mapping. Thus, if we are to map P_2 and P_4 on the faster processor N_3, the resulting schedule length (Figure 2a) will be 52 ms which does not meet the deadline. However, if we map P_2 and P_4 on the slower processor N_2, the schedule length (Figure 2b) is 48 ms, which is the best possible solution and meets the deadline. Note that the total traffic on the bus is the same for both mappings and the initial



	N_1	N_2	N_3
P_1	4ms	-	-
P_2	-	12ms	8ms
P_3	4ms	-	-
P_4	-	12ms	8ms

Figure 2. Mapping and Scheduling Example

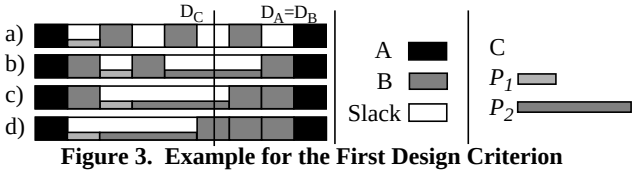


Figure 3. Example for the First Design Criterion

processor load is 0 on both N_2 and N_3. This result has its explanation in the impact of the communication protocol. P_3 cannot start before receiving messages $m_{2,3}$ and $m_{4,3}$. However, slot S_2 corresponding to node N_2 precedes in the TDMA round slot S_3 on which node N_3 communicates. Thus, the messages which P_3 needs are available sooner in the case P_2 and P_4 are, counter-intuitively, mapped on the slower node.

But finding a valid schedule is not always possible, either because there are not enough available hardware resources, or the resources are not intelligently allocated to the already running applications. Thus, in order to produce a valid solution, the resources have to be reallocated through rescheduling and remapping of some of the already running applications or, in the worst case, the architecture has to be modified by adding new resources.

In Figure 3 we consider a single processor system with three applications, A, B and C, each with a deadline D_A, D_B and D_C. Application C is depicted in more detail, showing the two processes P_1 and P_2 it is composed of. Let us suppose that the already running applications are A and B, and we have to implement C as a new application. If A and B have been mapped and schedules like in Figure 3a, we will not be able to map application C (in particular, process P_2). With a mapping of A and B like in Figure 3b and c, we are able to map both processes of C, but no schedule can be produced which meets the deadline D_C. If A and B are implemented like in Figure 3d, application C can be successfully implemented. Two aspects can be highlighted based on this example:

1. If applications A and B are implemented like in Figure 3a (or like in 3b or 3c), it is possible to correctly implement application C only with modifying the implementation of application B.
2. If during implementation of application B we would have taken into consideration that sometimes in the future an application like C will have to be implemented, we could have produced a schedule like the one in Figure 3d. In this case, application C could be implemented without any modification of an existing application.

3. PROBLEM FORMULATION

We model an application $\Gamma_{current}$ as a set of process graphs $G_i \in \Gamma_{current}$, each with a period T_{Gi} and a deadline $D_{Gi} \leq T_{Gi}$. For each process P_i in a process graph we know the set N_{Pi} of potential nodes on which it could be mapped and its worst case execution time on each of these nodes. The underlying architecture is as presented in section 2.1. We consider a non-preemptive static cyclic scheduling policy for both processes and message passing.

Our goal is to map and schedule an application $\Gamma_{current}$ on a system that already implements a set ψ of applications, considering the following requirements:

Requirement a: constraints on $\Gamma_{current}$ are satisfied and minimal modifications are performed to the applications in ψ.

Requirement b: new applications Γ_{future} can be mapped on the resulting system.

If it is not possible to map and schedule $\Gamma_{current}$ without modifying the already running applications, we have to change the scheduling and mapping of some applications in ψ. However, even with serious modifications performed on ψ, it is still possible that certain constraints are not satisfied. In this case the hardware architecture has to be changed by, for example, adding a new processor. In this paper we will not discuss this last case, but will concentrate on the situation where a possible mapping and

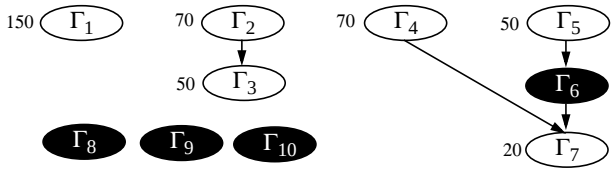


Figure 4. Characterizing the Set of Existing Applications

scheduling which satisfies requirement a) can be found, and this solution has to be further improved by considering requirement b).

In order to achieve our goals we need certain information to be available concerning the set of applications ψ as well as the possible future applications Γ_{future}. We consider that $\Gamma_{current}$ can interact with the previously mapped applications ψ by reading messages generated on the bus by processes in ψ. In this case, the reading process has to be synchronized with the arrival of the message on the bus, which is easy to solve during scheduling of $\Gamma_{current}$.

3.1 Characterizing Existing Applications

To perform the mapping and scheduling of $\Gamma_{current}$, the minimum information needed on the existing applications ψ consists of the local schedule tables for each node. Thus, we know the activation time for each process on the respective node and its worst case execution time. As for messages, their length as well as their place in the particular TDMA frame are known. However, if the initial attempt to schedule and map $\Gamma_{current}$ does not succeed, we have to modify the schedule and, possibly, the mapping of applications belonging to ψ, in the hope to find a valid solution for $\Gamma_{current}$.

Our goal in this paper is to find that minimal modification to the existing system that leads to a correct implementation of $\Gamma_{current}$. In our context, such a minimal modification means remapping and rescheduling a subset of old applications $\Omega \subseteq \psi$ so that the total cost of reimplementing Ω is minimized. We represent a set of applications as a directed acyclic graph *G(V, E)*, where each node $\Gamma_i \in V$ represents an application. An edge $e_{ij} \in E$ from Γ_i to Γ_j indicates that any modification to Γ_i would trigger the need to also remap and schedule Γ_j. Such a relation can be imposed by certain interactions between applications[1]. In Figure 4 we present the graph corresponding to a set of ten applications. Applications Γ_6, Γ_8, Γ_9 and Γ_{10}, depicted in black, are frozen: no modifications are possible to them. The rest of the applications have the remapping cost R_i depicted on their left. Γ_7 can be remapped with a cost of 20. If Γ_4 is to be reimplemented, this also requires the modification of Γ_7, with a total cost of 90. In the case of Γ_5, although not frozen, no remapping is possible as it would trigger the need to remap Γ_6 which is frozen. Given a subset of applications $\Omega \subseteq \psi$, the total cost of modifying the applications in Ω is $R(\Omega) = \sum_{\Gamma_i \in \Omega} R_i$.

To each application $\Gamma_i \in V$ the designer has associated a cost R_i of reimplementing Γ_i. Such a cost can typically be expressed in hours needed to perform retesting of Γ_i and other tasks connected to the remapping and rescheduling of the application. Remapping of Γ_i and the associated rescheduling can only be performed if the process graphs that capture the applications and their deadlines are available. However, this is not always the case, and in such situations the application is considered frozen.

3.2 Characterizing Future Applications

What do we suppose to know about the family Γ_{future} of applications which do not exist yet? Given a certain limited application area (e.g. automotive electronics), it is not unreasonable to assume that, based on the designers' previous experience, the nature of expected future functions to be implemented, profiling of previous applications, available uncomplete designs for future versions of the product, etc., it is possible to characterize the family of applications which could possibly be added to the current implementation. This is an assumption which is basic for the concept of incremental design. Thus, we consider that, concerning the future applications, we know the set $S_t=\{t_{min},...t_i,...t_{max}\}$ of possible worst case execution times for processes, and the set $S_b=\{b_{min},...b_i,...b_{max}\}$ of possible message sizes. We also assume that over these sets we know the distributions of probability $f_{St}(t)$ for $t \in S_t$ and $f_{Sb}(b)$ for $b \in S_b$. For example, we might have worst case execution times $S_t=\{50, 100, 200, 300, 500$ ms$\}$. If there is a higher probability of having processes of 100 ms, and a very low probability of having processes of 300 ms and 500 ms, then our distribution function $f_{St}(t)$ could look like this: $f_{St}(50)=0.20$, $f_{St}(100)=0.50$, $f_{St}(200)=0.20$, $f_{St}(300)=0.05$, and $f_{St}(500)=0.05$.

Another information is related to the period of process graphs which could be part of future applications. In particular, the smallest expected period T_{min} is assumed to be given, together with the expected necessary processor time t_{need}, and bus bandwidth b_{need}, inside such a period T_{min}. As will be shown later, this information is used in order to provide a fair distribution of slacks.

The execution times in S_t as well as t_{need} are considered relative to the slowest node in the system. All the other nodes are characterized by a speedup factor relative to this slowest node.

3.3 Quality Metrics

A designer will be able to map and schedule an application Γ_{future} on top of a system implementing ψ and $\Gamma_{current}$, only if there are sufficient resources available. In our case, the resources are processor time and the bandwidth on the bus. In the context of a non-preemptive static scheduling policy, having free resources translates into having free time slots on the processors and having space left for messages in the bus slots. We call these free slots of available time on the processor or on the bus, *slack*. It is the size and distribution of the slacks that characterizes the quality of a certain design alternative from the point of view of its potential to accommodate future applications. In this section we introduce two criteria in order to reflect the degree to which one design alternative meets the requirement b) presented at the beginning of section 3.

The first criterion reflects how well the resulted slack sizes fit to a future application. The slack sizes resulted after implementation of $\Gamma_{current}$ on top of ψ should be such that they best accommodate a given family of applications Γ_{future}, characterized by the sets S_t, S_b and the probability distributions f_{St} and f_{Sb}, as outlined before. Let us consider the example in Figure 3, where we have a single processor with the applications *A* and *B* implemented and a future application *C* which consists of the two processes, P_1 and P_2. It can be observed that the best configuration, taking in consideration only slack sizes, is to have a contiguous slack. Such a slack, as depicted in Figure 3c and d, will best accommodate any future application. However, in reality, it is almost impossible to map and schedule the current application such that a contiguous slack is obtained. Not only is it impossible, but it is also undesirable from the point of view of the second design criterion, discussed below.

The second criterion expresses how well the slack is distributed in time. Let P_i be a process with period T_{P_i} that belongs to a future application, and $M(P_i)$ the node on which P_i will be mapped. The worst case execution time of P_i is $t_{P_i}^{M(Pi)}$. In order to schedule P_i we need a slack of size $t_{P_i}^{M(Pi)}$ that is available periodically, within a period T_{Pi}, on processor $M(P_i)$. If we consider a group of processes with period *T*, which are part of Γ_{future}, in order to implement them, a certain amount of slack is needed which is available periodically, with a period *T*, on the nodes implementing the respective processes. During implementation of

[1] If a set of applications have a circular dependence, such that the modification of any one implies the remapping of all the others in that set, the set will be represented as a single node in the graph.

$\Gamma_{current}$ we aim for a slack distribution such that the future application with the smallest expected period T_{min} and with the expected necessary processor time t_{need}, and bandwidth b_{need}, can be accommodated.

We have defined two metrics, C_1 and C_2, which quantify the degree to which the first and second criterion, respectively, are met. A detailed discussion about these metrics is given in [11].

3.4 Cost Function and Exact Problem Formulation

In order to capture how well a certain design alternative meets the requirement b) stated in section 3, the metrics discussed before are combined in an objective function, as follows:

$$C = w_1^P (C_1^P)^2 + w_1^m (C_1^m)^2 + w_2^P max(0, t_{need} - C_2^P) + w_2^m max(0, b_{need} - C_2^m)$$

C_1^P and C_2^P are those components of the two metrics that capture the slack properties on processors, while C_1^m and C_2^m are calculated for the slacks on the bus. Our mapping and scheduling strategy will try to minimize this function.

The first two terms measure how well the resulted slack sizes fit to a future application (first criterion), while the second two terms reflect the distribution of slacks (second criterion). We call a *valid solution* that mapping and scheduling which satisfies all the design constraints (in our case the deadlines) and meets the second criterion ($C_2^P \geq t_{need}$ and $C_2^m \geq b_{need}$)[1].

At this point we can give an exact formulation to our problem. Given an existing set of applications ψ which are already mapped and scheduled, and an application $\Gamma_{current}$ to be implemented on top of ψ, we are interested to find the subset $\Omega \subseteq \psi$ of old applications to be remapped and rescheduled such that we produce a valid solution for $\Gamma_{current} \cup \Omega$ and the total cost of modification $R(\Omega)$ is minimized. Once such an Ω is found, we are interested to minimize the objective function C for the set $\Gamma_{current} \cup \Omega$, considering a family of future applications characterized by the sets S_t and S_b, the functions f_{St} and f_{Sb} as well as the parameters T_{min}, t_{need}, and b_{need}.

4. MAPPING AND SCHEDULING STRATEGY

As shown in Figure 5, our mapping and scheduling strategy (MS) has two steps. In the first step we try to obtain a valid solution for $\Gamma_{current} \cup \Omega$ so that $R(\Omega)$ is minimized. Starting from such a solution, a second step iteratively improves on the design in order to minimize the objective function C.

4.1 The Initial Mapping and Scheduling

The first step of MS consists of an iteration that tries subsets $\Omega \subseteq \psi$ with the intention to find that subset $\Omega=\Omega_{min}$ which produces a valid solution for $\Gamma_{current} \cup \Omega$ such that $R(\Omega)$ is minimized. Given a subset Ω, the InitialMappingScheduling function (IMS) constructs a mapping and schedule for $\Gamma_{current} \cup \Omega$ that meets the deadlines, without worrying about the two criteria in section 3.3. For IMS we used as a starting point the Heterogeneous Critical Path (HCP) algorithm, introduced in [5]. HCP is based on a list scheduling algorithm. We have modified the HCP algorithm to consider, during mapping and scheduling, a set of previous applications that have already occupied parts of the schedule table, and to schedule the messages according to the TDMA protocol. Furthermore, for the selection of processes we have used, instead of the CP (critical path) priority function, the (modified partial critical path) MPCP priority function introduced by us in [3]. MPCP takes into consideration the particularities of the communication protocol for calculation of communication delays. These delays are not estimated based only on the message length, but also on the time when slots assigned to the particular node which generates the message, will be available.

However, before using the IMS algorithm, two aspects have to be addressed. First, the process graphs $G_i \in \Gamma_{current} \cup \Omega$ are merged into a single graph $G_{current}$, by unrolling of process graphs and insertion of dummy nodes [11]. In addition, we have to consider during scheduling the mismatch between the periods of the already existing system and those of the current application. The schedule table into which we would like to schedule $G_{current}$ has a length of $T_{\psi\setminus\Omega}$ which is the global period of the system ψ after extraction of the applications in Ω. However, the period $T_{current}$ of $G_{current}$ can be different from $T_{\psi\setminus\Omega}$. Thus, before scheduling $G_{current}$ into the existing schedule table, the schedule table is expanded to the least common multiplier of the two periods. A similar procedure is followed in the case $T_{current} > T_{\psi\setminus\Omega}$.

4.2 The Basic Strategy

If IMS succeeds in finding a mapping and schedule which meet the deadlines, this is not yet a valid solution. In order to produce a valid solution we iteratively try to satisfy the second design criterion. In terms of our metrics, that means a mapping and scheduling such that $C_2^P \geq t_{need}$ and $C_2^m \geq b_{need}$. Potential moves can be the shifting of processes inside their [*ASAP, ALAP*] interval in order to improve the periodic slack. The move can be performed on the same node or to other nodes. Similar moves are considered for messages. SelectMoveC_2 evaluates these moves with regard to the second design criterion and selects the best one to be performed. Any violation of the data dependency constraints is rectified by moving processes or messages concerned in an appropriate way.

If Step 1 has succeeded, a mapping and scheduling of $\Gamma_{current} \cup \Omega$ has been produced which corresponds to a valid solution. In addition, Ω is such that the total modification cost is as small as possible. Starting from this valid solution, the second step of the MS strategy, presented in Figure 5, tries to improve on the design in order to minimize the objective function C. In a similar way as during Step 1, we iteratively improve the design by successive moves.

In [11] we introduced a heuristic with the goal of guiding the moves discussed above. Its intelligence lies in how the moves are selected. For each iteration a set of potential moves is selected by the PotentialMove function. SelectMove then evaluates these moves with regard to the respective metrics and selects the best one to perform.

4.3 Minimizing the Modification Cost

The first step of our mapping strategy described in Figure 5 iterates on subsets Ω searching for a valid solution which also minimizes the total modification cost $R(\Omega)$. As a first attempt, the algorithm searches for a valid implementation of $\Gamma_{current}$ without disturbing the existing applications ($\Omega=\emptyset$). If no valid solution is found successive subsets Ω produced by the function NextSubset are considered, until a terminating condition is met. The performance of the algorithm, in terms of runtime and quality of the solutions produced, is strongly influenced by the implementation of the function NextSubset and the termination condition. They determine how the design space is explored while testing different subsets Ω of applications.

4.3.1 Exhaustive Search (ES)

In order to find Ω_{min}, the simplest solution is to try successively all the possible subsets $\Omega \subseteq \psi$. These subsets are generated in the ascending order of the total modification cost, starting from $\emptyset$. The termination condition is fulfilled when the first valid solution is generated. Since the subsets are generated in ascending order, according to their cost, the subset Ω that first produces a valid solution is also the subset with the minimum modification cost.

The generation of subsets is performed according to the graph G that characterizes the existing applications (see section 3.1). Finding the next subset Ω, starting from the current one, is achieved by a branch and bound algorithm that in the worst case grows exponentially in time with the number of applications. For the example in Figure 4, the call to NextSubset($\emptyset$) will generate

[1] This definition of a valid solution can be relaxed by imposing only the satisfaction of deadlines. In this case, the algorithm in Figure 5 will look after a solution which satisfies the deadlines and R(Ω) is minimized; the two additional criteria are only considered optionally.

```
MappingSchedulingStrategy
  Ω=∅
  -- Step 1: try to find a valid schedule for Γcurrent that minimizes R(Ω)
  repeat
    succeeded=InitialMappingScheduling(ψ \ Ω, Γcurrent∪Ω)
    -- compute ASAP-ALAP intervals
    ASAP(Γcurrent∪Ω); ALAP(Γcurrent∪Ω)
    if succeeded then
      repeat -- try to satisfy the second design criterion
        -- find moves with highest potential to maximize C2
        move_set=PotentialMoveC2(Γcurrent∪Ω)
        -- select and perform move which improves most C2
        move = SelectMoveC2(move_set); Perform(move)
        succeeded = C2^p ≥ tneed and C2^m ≥ bneed
      until succeeded or limit reached
    end if
    if succeeded and R(Ω) smallest so far then
      Ωvalid=Ω; solutionvalid=solutioncurrent
    end if
    -- try another subset
    Ω=NextSubset(Ω)
  until termination condition
  if not succeeded then modify architecture; go to step 1; end if
  -- Step 2: try to improve the cost function C
  solutioncurrent=solutionvalid; Ωmin=Ωvalid
  repeat
    -- find moves with highest potential to minimize C
    move_set=PotentialMoveC(Γcurrent∪Ωmin)
    -- select move which improves C
    -- and does not invalidate the second design criterion
    move = SelectMoveC(move_set); Perform(move)
  until C1 has not changed or limit reached
end MappingSchedulingStrategy
```

Figure 5. MS Strategy to Support Iterative Design

$\{\Gamma_7\}$ which has the smallest nonzero modification cost. The next generated subsets, in order, together with their corresponding total modification cost are: $R(\{\Gamma_3\})$=50, $R(\{\Gamma_3, \Gamma_7\})$=70, $R(\{\Gamma_4, \Gamma_7\})$=90 (the inclusion of Γ_4 triggers the inclusion of Γ_7), $R(\{\Gamma_2, \Gamma_3\})$=120, $R(\{\Gamma_3, \Gamma_4, \Gamma_7\})$=140, $R(\{\Gamma_1\})$=150, and so on. The total number of possible subsets according to the graph G is 16.

This approach, while finding the optimal subset Ω, requires a large amount of computation time and can be used only with a small number of applications.

4.3.2 Ad-hoc Solution (AH)

If the number of applications is larger, a possible ad-hoc solution could be based on a greedy strategy which, starting from Ω=∅, progressively enlarges the subset until a valid solution is produced. The algorithm looks at all the non-frozen applications and picks that one which, together with its dependencies, has the smallest modification cost. If the new subset does not produce a valid solution, it is enlarged by including, in the same fashion, the next application with its dependencies. This greedy expansion of the subset is continued until the set is large enough to lead to a valid solution or no application is left. For the example in Figure 4 the call to NextSubset(∅) will produce $R(\{\Gamma_7\})$=20, and will be successively enlarged to $R(\{\Gamma_7, \Gamma_3\})$=70, $R(\{\Gamma_7, \Gamma_3, \Gamma_2\})$=140 ($\Gamma_4$ could have been picked as well in this step because it has the same modification cost of 70 as Γ_2 and its dependence Γ_7 is already in the subset), $R(\{\Gamma_7, \Gamma_3, \Gamma_2, \Gamma_4\})$=210, and so on.

While this approach finds very quickly a valid solution, if one exists, it is possible that the total modification cost is much higher than the optimal one.

4.3.3 Subset Selection Heuristic (SH)

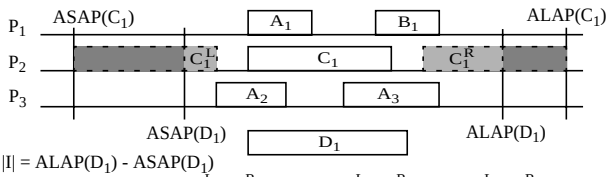
An intelligent selection heuristic should be able to identify the reasons due to which a valid solution has not been found. Such a failure can have two possible causes: an initial mapping which meets the deadlines has not been produced, or the second criterion is not satisfied.

Let us investigate the first reason. If an application Γ_i is to meet its deadline D_i, all its processes $P_j \in \Gamma_i$ have to be scheduled inside their [*ASAP*, *ALAP*] intervals. InitialMappingScheduling (IMS) fails to schedule a process inside its [*ASAP*, *ALAP*] interval if there is not enough slack available on any processor, due to other processes scheduled in the same interval. In this situation we say that there is a *conflict* with processes belonging to other applications. We are interested to find out which applications are responsible for conflicts encountered by our $\Gamma_{current}$, and not only that, but also which ones are *flexible* enough to move away in order to avoid these conflicts.

IMS determines a metric Δ_i that characterizes the degree of conflict and the flexibility of application Γ_i in relation to $\Gamma_{current}$. A set of applications Ω will be characterized, in relation to $\Gamma_{current}$, by $\Delta(\Omega) = \sum_{\Gamma_i \in \Omega} \Delta_i$. The metric Δ(Ω) will be used by our subset selection heuristic if IMS has failed to produce a solution which satisfies the deadlines. An application with a larger Δ_i is more likely to lead to a valid schedule if included in Ω. In Figure 6 we illustrate how this metric is calculated. Applications *A*, *B* and *C* are scheduled on three processors P_1, P_2 and P_3, and our goal is to implement the current application *D*. At a certain moment IMS comes to the point to place process $D_1 \in D$. However, it is not able to place D_1 inside its [*ASAP*, *ALAP*] interval *I*, because there is not enough free slack available inside *I* on any of the processors. We are interested to determine which of the applications *A*, *B* and *C* are more likely to lend free slack for D_1 if remapped. Therefore, we calculate the slack resulted after we move away processes from the interval *I*. For example, the resulted slack available after remapping application *C* (moving process $C_1 \in C$ either to the left or to the right inside its own [*ASAP*, *ALAP*] interval) is of size $|I| - \min(|C_1^L|, |C_1^R|)$. Thus, we increment Δ_C with $\delta_C = |I| - \min(|C_1^L|, |C_1^R|) - |D_1|$. The increments δ_B and δ_A to be added to Δ_B and Δ_A respectively, are also presented in Figure 6. IMS continues with the other processes of application D (after assuming that process D_1 has been scheduled at the beginning of interval *I*). As result of the failed attempt to map *D*, IMS will produce the metrics Δ_A, Δ_B, and Δ_C.

If the initial mapping was successful, the first step of MS could fail during the attempt to satisfy the second criterion. In this case, the metric Δ_i is computed in a different way. It will capture the potential of an application Γ_i to improve the metric C_2 if remapped together with $\Gamma_{current}$. Thus, for the improvement of C_2 we consider a total number of moves from all the non-frozen applications (determined using Potential-MoveC$_2$(ψ)). For each move that has as subject $P_j \in \Gamma_i$, we increment the metric Δ_i with the predicted improvement on C_2.

MS starts by trying an implementation of $\Gamma_{current}$ with Ω=∅. If this attempt fails, because of one of the two reasons mentioned above, the corresponding metrics Δ_i are computed for all $\Gamma_i \in \psi$. Our heuristic SH will then start by finding the ad-hoc solution Ω_{AH} produced by the AH algorithm (this will succeed if there exists any solution) with a corresponding cost $R_{AH}=R(\Omega_{AH})$ and a $\Delta_{AH}=\Delta(\Omega_{AH})$. SH now continues by trying to find a solution with a more favorable Ω (a smaller total cost R). Therefore, the thresholds $R_{max}=R_{AH}$ and $\Delta_{min}=\Delta_{AH}/n$ (for our experiments we considered n=2) are set. For generating new subsets Ω, the function NextSubset now follows a similar approach like ES but in a reverse direction, towards smaller subsets, and it will consider only subsets with a smaller total cost then R_{max} and a larger Δ then Δ_{min} (a small Δ means a reduced potential to eliminate the cause of the



$|I| = ALAP(D_1) - ASAP(D_1)$

$\delta_A = \max(|I| - |B_1| - \min(|A_1^L|, |A_1^R|), |I| - \min(|A_2^L|, |A_2^R|) - \min(|A_3^L|, |A_3^R|)) - |D_1|$

$\delta_B = |I| - |A_1| - \min(|B_1^L|, |B_1^R|) - |D_1|$

$\delta_C = |I| - \min(|C_1^L|, |C_1^R|) - |D_1|$

Figure 6. Metric for the Subset Selection Heuristic

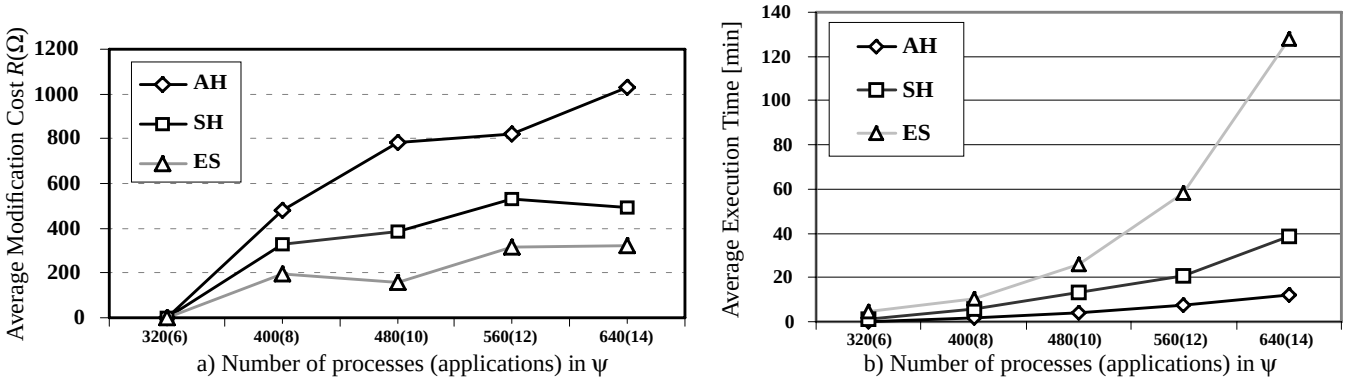


Figure 7. Average Modification Cost (a) and Execution Time (b) for MS with the AH, SH and ES Approaches to Subset Selection

initial failure). Each time a valid solution is found, the current values of R_{max} and Δ_{min} are updated in order to further restrict the search space. The heuristic stops when no subset can be found with $\Delta>\Delta_{min}$, or a certain imposed limit has been reached (e.g. on the total number of attempts to find new sets).

5. EXPERIMENTAL RESULTS

For evaluation of the proposed strategies we first used process graphs of 80, 160, 240, 320 and 400 processes, representing the application $\Gamma_{current}$, generated for experimental purpose. 30 graphs were generated for each graph dimension, resulting in a total of 150 graphs. We considered an architecture consisting of 10 nodes. For the communication channel we considered a transmission speed of 256 kbps and a length below 20 meters. The maximum length of the data field in a bus slot was 8 bytes. Experiments were run on a SUN Ultra 10.

The first results concern the quality of the solution obtained with our mapping strategy MS using the search heuristic SH compared to the case when the ad-hoc approach AH and the exhaustive search ES are used. For each of the five graph dimensions for $\Gamma_{current}$ we have considered a set of existing applications ψ consisting of 320, 400, 480, 560 and 640 processes, respectively. The sets contained 6, 8, 10, 12 and 14 applications, each application with an associated modification cost assigned manually in the range 10 to 100. The available slack is of about 50% of the total schedule size. The dependencies between applications were such that the total number of possible subsets Ω resulted for each set ψ were 32, 128, 256, 1024 and 4096. We have considered that the future applications Γ_{future} consist of a process graph of 80 processes, randomly generated according to the following specifications: S_t={20, 50, 100, 150, 200 ms}, $f_t(S_t)$={10, 25, 45, 15, 5%}, S_b={2, 4, 6, 8 bytes}, $f_b(S_b)$={20, 50, 20, 10%}, T_{min}=250 ms, t_{need}=100 ms and b_{need}=20 ms.

MS has been used to produce a valid solution for each of the 150 process graphs representing $\Gamma_{current}$ on the target system ψ using the ES, AH and SH approaches to subset selection. Figure 7a compares the three approaches based on the total modification cost needed in order to obtain a valid solution. The exhaustive approach ES is able to obtain valid solutions with an optimal (smallest) modification cost, while the ad-hoc approach AH produces in average 3.12 times more costly modifications in order to obtain valid solutions. However, in order to find the optimal solution ES needs large computation times, as shown in Figure 7b. For example, it can take more than 2 hours in average to find the smallest cost subset to be remapped that leads to a valid solution in the case of 14 applications (640 processes). We can see that the proposed heuristic SH performs well, producing close to optimal results with a good scaling for large application sets. For the results in Figure 7 we have eliminated those situations in which no valid solution could be produced by MS.

Another important aspect to be proven by experiments is the extent to which the mapping strategy proposed in the paper really facilitates the implementation of *future* applications. For these experiments we have considered that no modifications are allowed to the applications in ψ. We have used an existing set of applications ψ consisting of 400 processes, with a schedule table of 6s on each processor, and a slack of about 50% of the total schedule size. Then, we have mapped graphs of 40, 80, 160 and 240 nodes representing the $\Gamma_{current}$ application on top of ψ.

After mapping and scheduling each of these graphs we have tried to add a new application Γ_{future} to the resulted system (for Γ_{future} we used the same experimental set as presented before). The experiments have been performed two times, using first MS* (we call MS* the version of MS in which no modification of applications in ψ is allowed), and then an *ad-hoc mapping* approach (AM), for mapping $\Gamma_{current}$. In both cases we were interested if it is possible to find a valid implementation for Γ_{future} on top of $\Gamma_{current}$, using the initial mapping algorithm IMS. The AM approach is a simple, straight-forward solution to produce designs which, to a certain degree, support an incremental process. Starting from the initial valid schedule of length S obtained by IMS for the graph G with N processes, AH uses a simple scheme to redistribute the processes inside the [0, D] interval, where D is the deadline of the process graph G. AH starts by considering the first process in topological order, let it be P_1. It introduces after P_1 a slack of size min(smallest process size of Γ_{future}, $(D-S)/N$), thus shifting all P_1's descendants to the right. The insertion of slacks is repeated for the next process, with the current larger value of S, as long as the resulted schedule has an $S \leq D$.

Figure 8 shows the number of successful implementations in the two cases. In the case $\Gamma_{current}$ has been mapped with MS*, this means using the design criteria and metrics proposed in the paper, we were able to find a valid solution for 65% of the total process graphs considered. However, using AM to map $\Gamma_{current}$ has led to a situation where IMS is able to find schedules which satisfy the deadlines for only 27.5% cases. When $\Gamma_{current}$ grows to 160 processes, only MS* is able to find a mapping of $\Gamma_{current}$ that supports an incremental design process, accommodating more that 60% of the future applications. If the remaining slack is very small, after we map a $\Gamma_{current}$ of 240, it becomes practically impossible to map new applications without modifying the current system.

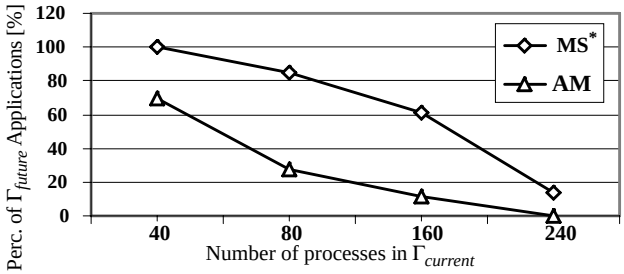


Figure 8. Percentage of Γ_{future} Apps. Successfully Mapped

If the mapping heuristic is allowed to modify the existing system, as discussed in this paper, then we are able to increase the total number of successfully mapped applications Γ_{future} from 65% with MS* to 77.5% with MS. For a $\Gamma_{current}$ with 160 processes the increase is from 60% to 92%. Such an increase is, of course, expected. The important aspect, however, is that it is obtained not by randomly selecting old applications to be remapped, but by performing this selection such that the total modification cost is minimized.

Finally, we considered an example implementing a vehicle cruise controller (CC) modeled using one process graph. The graph has 32 processes and it was to be mapped on an architecture consisting of 4 nodes, namely: Anti Blocking System, Transmission Control Module, Engine Control Module and Electronic Throttle Module. The period was 300 ms, equal to the deadline. In order to validate our approach, we have considered the following setting. The system ψ consists of 80 processes generated randomly, with a schedule table of 300 ms and about 40% slack. The CC is the $\Gamma_{current}$ application to be mapped. We have also generated 30 future applications of 40 processes each with the characteristics of the CC, which are typical for automotive applications. By mapping the CC using MS* we were able to later map 21 of the future applications, while using AM only 4 of the future applications could be mapped. When modifications of the current system were allowed, using MS, we are able to map 24 of the 30 future applications.

6. CONCLUSIONS

We have presented an approach to the incremental design of distributed hard real-time embedded systems. Such a design process satisfies two main requirements when adding new functionality: already running applications are disturbed as little as possible, and there is a good chance that, later, new functionality can easily be mapped on the resulted system. Our approach assumes a non-preemptive static cyclic scheduling policy and a realistic communication model based on a TDMA scheme.

We have introduced two design criteria with their corresponding metrics that drive our mapping strategy to solutions supporting an incremental design process. Three algorithms have been proposed to produce a minimal subset of applications which have to be remapped and scheduled in order to implement the new functionality. ES is based on a, potentially slow, branch and bound strategy which finds an optimal solution. AH is very fast but produces solutions that could be of too high cost, while SH is able to quickly produce good quality results. The approach has been validated through several experiments.

REFERENCES

[1] T. Blicke, J. Teich, L. Thiele, “System-Level Synthesis Using Evolutionary Algorithms”, Des. Aut. for Emb. Syst., V4, N1, 1998, 23-58.

[2] B.P. Dave, G. Lakshminarayana, N.K. Jha, “COSYN: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded Systems”, IEEE Trans. on VLSI Systems, March 1999, 92 -104.

[3] P. Eles, A. Doboli, P. Pop, Z. Peng, “Scheduling with Bus Access Optimization for Distributed Embedded Systems”, IEEE Transactions on VLSI Systems, October 2000.

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

[5] P. B. Jorgensen, J. Madsen, “Critical Path Driven Cosynthesis for Heterogeneous Target Architectures,” Proc. Int. Workshop on Hardware-Software Co-design, 1997, 15-19.

[6] P.V. Knudsen, J. Madsen, "Integrating Communication Protocol Selection with Hardware/Software Codesign", IEEE Trans. on CAD, V18, N8, 1999, 1077-1095.

[7] H. Kopetz, G. Grünsteidl, “TTP-A Protocol for Fault-Tolerant Real-Time Systems,” IEEE Computer, 27(1), 1994, 14-23.

[8] C. Lee, M. Potkonjak, W. Wolf, "Synthesis of Hard Real-Time Application Specific Systems", Des. Aut. for Emb. Syst., V4, N4, 1999, 215-241.

[9] S. Narayan, D.D. Gajski, "Synthesis of System-Level Bus Interfaces", Proc. Europ. Des. & Test Conf, 1994, 395-399.

[10] R.B. Ortega, G. Borriello, "Communication Synthesis for Distributed Embedded Systems", Proc. Int. Conf. on CAD, 1998, 437-444.

[11] P. Pop, P. Eles, T. Pop, Z. Peng, “An Approach to Incremental Design of Distributed Embedded Systems,” Submitted to the Design Automation Conference, 2001.

[12] S. Prakash, A. Parker, “SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems”, Journal of Parallel and Distrib. Comp., V16, 1992, 338-351.

[13] D. L. Rhodes, Wayne Wolf, “Co-Synthesis of Heterogeneous Multiprocessor Systems using Arbitrated Communication”, Proceeding of the 1999 International Conference on CAD, 1999, 339 - 342.

[14] T. Y. Yen, W. Wolf, "Hardware-Software Co-Synthesis of Distributed Embedded Systems", Kluwer Academic Publ., 1997.

Minimization of Execution Scenarios in Static Priority Preemptive Scheduled Real-Time systems

Anders Pettersson
Email : apo@mdh.se
Department of Computer Science and Computer Engineering
Mälardalen University
P.O. Box 883, SE-721 23 Västerås, SWEDEN

Abstract

In static priority preemptive real-time systems a large number of execution scenarios can occur. This can lead to that a lot of effort is put into testing of the system. The reason for this can be that a certain degree of confidence must be assured, and the coverage criteria is set to test all possible execution scenarios. In distributed systems the complexity increases with the amount of the scenarios produced. The effort in testing of such system could be decreased if the number of execution scenarios were reduced. In this paper we show results from simulations of jitter minimization in execution time of tasks in static priority preemptive scheduled real-time systems. By simulation of a real-time micro-kernel with support for jitter minimization we have observed that reduction of execution scenarios can be acheived. This has been done by adapting the worst case and best case execution time parameters. We have also made some basic observation when there is only one execution scenario. Algorithms revealing preemption points that affects the number of execution scenarios are presented. These algorithms can assist real-time systems developer to consider testability at design time.

Deterministic Java in Tiny Embedded Systems

Anders Nilsson
Torbjörn Ekman
Department of Computer Science
Lund University, Sweden
{andersn | torbjorn}@cs.lth.se

Abstract

As embedded systems become more and more complex, and the time to market becomes shorter, there is a need in the embedded systems community to find better programming languages that let the programmers develop correct code faster. The programming languages used today—typically C and/or Assemblers—are just too error-prone. The Java technology has therefore gained a lot of interest from developers of embedded systems in the last few years.

We propose an approach based on compiling Java into native machine code via C as an intermediate language. The C code generation process should also add close interaction with a fully pre-emptive incremental garbage collector and a small and efficient real-time kernel. Tests performed on a small 8-bit microprocessor show that it is possible to use a modern object-oriented language with automatic memory management—such as Java—and yet generate fully predictable code that can be run in very small devices with severe memory constraints.

1. Introduction

Embedded systems are traditionally programmed in C or Assembler, but as the systems grow more complex and the time to market decrease, the need for a more secure and structured language has increased.

In the last few years, the programming language Java [11] has gained more and more interest among embedded systems developers. The object-orientation, strict type checking, and automatic memory management, are features that make it easier to write large and complex systems with less risk of introducing difficult bugs. It's C-like syntax also appeals to C programmers as it makes it easier for them learn, as well as makes it easier to port legacy C code to Java. In the area of automatic control, where domain-specific languages and run-time systems [9, 1] are preferably used, our work supports more robust porting of such systems to even small embedded devices.

However, some serious problems arise when one wants to use Java in small embedded systems, where the most significant ones have to do with the inevitable speed- and memory constraints imposed by using a Java Virtual Machine (JVM). If hard real-time demands are to be fulfilled, there are also problems with most garbage collectors not being predictable with regard to non-preemptable execution time. This can result in too much jitter in the accomplished sampling interval of high priority threads, or even missed samples or deadlines.

1.1. Problem area

Our work is directed towards a certain class of applications and systems where a traditional Java environment cannot fulfill our application demands. Examples of such applications can be found in tiny embedded control devices used in industrial processes. The demands on that kind of embedded system are:

Correctness: The system must not crash due to some bug causing a memory leak or pointer arithmetic failure, for instance once every three months. Of course a programming error can result in too much memory being consumed, but in such a case controlled error handling (exceptions) should enable graceful degradation.

Hard Real-Time: The physical process may be very sensitive to jitter in the sampling interval of the controller. Too much jitter may cause the process to perform badly, or even possibly become unstable[2].

Speed: Applications, such as a servo control, may need a sampling interval down to a few milliseconds to perform well.

Cost: To be able to sell such systems, in some cases, we cannot use anything more powerful than, say, a simple 8 bit micro-controller with less than 128KB of RAM.

The **Correctness** demand indicates that we should use a modern language with strong typing and automatic memory management, and only use C and assembler for isolated functions and hardware interfaces. Of course an application needs to be correctly written and tested in order to really be correct. Using a safe[1] language drastically improves the development process. The **Hard real-time** demand requires predictability in both run-time and garbage collector. The **Speed** and **Cost** demands indicates that we must look for a reasonably effective solution so that cheap micro-controllers can be used.

The correctness demand and the other three demands have so far been contradictory. The topic of this paper is to resolve that contradiction! On one hand, we want to use Java, representing a safe object-oriented programming language with many other nice features. On the other hand, Java in it's traditional—interpreted byte code—form is neither predictable regarding timing nor can it be run on small micro-controllers with very limited amounts of RAM.

We are also committed to not making any extensions to the Java language as that would make it more difficult to simulate the software with standard Java tools on a standard workstation where all sorts of debugging tools are available.

From a pessimistic point of view, the Java language does not support predictability and real-time programming, and Java VMs and standard class libraries does not fit into very small systems. From an optimistic point of view, however, the Java language supports concurrency and it does not explicitly hinder utilization of real-time support from the underlying system, and for development of embedded systems we are free to use an appropriate (possibly our own) runtime system. The following is based on the optimistic approach.

1.2. Approach

We propose an approach consisting of three parts working together:

Compiled Java: By compiling Java—via C as an intermediate language—we should be able to make typical applications sufficiently fast and memory effective.

Real-Time Kernel: A small real-time kernel which is tailored for small micro-controllers and object-oriented applications.

Predictable Garbage Collector: A predictable garbage collector, which is integrated with the kernel, helps us to fulfill also the hard real-time demand.

[1]By a *safe language* we mean a language that ensures that all possible executions are expressed by the program itself. Specifically C and C++ are unsafe, whereas Java is safe. C# is safe except where declared unsafe.

2. Compiling Java

To be able to meet the demands on speed and memory consumption[2], we need to compile our Java code into processor specific machine code. There are basically two ways to do this:

Native compiler: An ordinary compiler taking Java source code, or byte code, and producing machine dependent binary code.

Intermediate language: A tool for converting Java source or byte code to some intermediate language. The intermediate representation can then be compiled with a standard compiler for the specific machine architecture.

2.1. Native Compiler

Using a native Java compiler seems at a first sight to be the most straight-forward way to produce machine code from Java source. There are some native compilers available, both as commercial packages and as open source. Most of those are aimed at speeding up execution of large Java applications, especially on the server side in a client-server solution, see for example TowerJ3[25] or Jove[13]. WindRiver Inc. has developed a native Java compiler, TurboJ[28], that produces object files which can be linked to their real-time operating system VxWorks. They have however not dealt with the predictability problem incurred by the automatic memory management, and thus recommend that all critical real-time parts of an application should be written in C/C++ as usual.

2.2. Intermediate language

There are some tools available today which can translate Java to an intermediate language, usually C. Most of them, for example Toba[20] and Harissa[17] take Java byte code and converts it to C source code. The other type of converter—going from Java source to C source—is represented by jcc[23]. There are good things and bad things in both approaches. Converting byte code makes it easier to use pre-compiled Java libraries or applications which may not be available as source. On the other hand, the byte code is tailored for a generic stack machine with no hardware registers, which we think will harm the performance compared to generating C code from Java source, at least if no special optimization techniques are used. Converting Java source code also produces a somewhat more readable

[2]The Java byte codes actually occupies less memory than machine codes, but a JVM will take some memory space. Both in ROM for itself and some extra RAM for it's runtime. A JVM implemented in hardware would need significantly less ROM, but that solution has other drawbacks.

C code, which makes debugging feasible also for the intermediate code.

The biggest drawback of using Java source as input to the C code converter is that many of the available Java packages are only available as pre-compiled byte code. This introduces the limitation that we can only compile Java programs that are —in itself and for all dependencies—available as source. However, preliminary tests with Java decompilers, such as jad [15], shows that the byte code can quite well be decompiled into Java source, then making our approach feasible.

2.3. Our Compiler

For portability- and efficiency reasons we are focusing on going via C as an intermediate language for compiled Java, rather than implementing a compiler or adapt an existing compiler, for example GCC[7], to meet our needs. A tool that converts Java source to C can also generate object layout information and calls that makes predictable garbage collection possible.

A tool called Java2C has been developed. Given Java source as input, it generates the corresponding C code as well as the necessary GC administrative calls and information about the layout and size of objects.

The compiler: The Java2C tool is built in 100% pure Java2, so it can be used on any platform that can run a Java2 virtual machine. It also implies that the Java2C tool could convert itself for native compilation and better performance. A parser generator is used to build a parser for the Java formal grammar. The parser can then take any valid Java source code file and produce an abstract syntax tree (AST). From this AST, the C source code is then generated in one pair of files for each Java class or interface (one .h and one .c file).

The compiler-compiler: The JavaCC compiler-compiler [16] is a freely available parser generator written in Java. It was originally written by Sun Microsystems but is now freely available from Metamata Inc. Given a grammar in a BNF-like form, it builds a parser in Java and optionally also an AST from the parsed file. Each node of this AST consists of a Java class all inheriting a common ancestor `Simplenode`. This makes it fairly easy to traverse the syntax tree by using method overloading.

Code generation The generation of C code is accomplished by traversing the AST of a class in two passes. During the first pass information about inheritance, field- and method declarations is gathered whereas the actual code generation is accomplished during the second pass.

2.4. Object model

Some works has been carried out at the department on how to model compiled real-time Java [4]. An instance of any object is represented by a pointer to an object instance structure, see Figure 1.

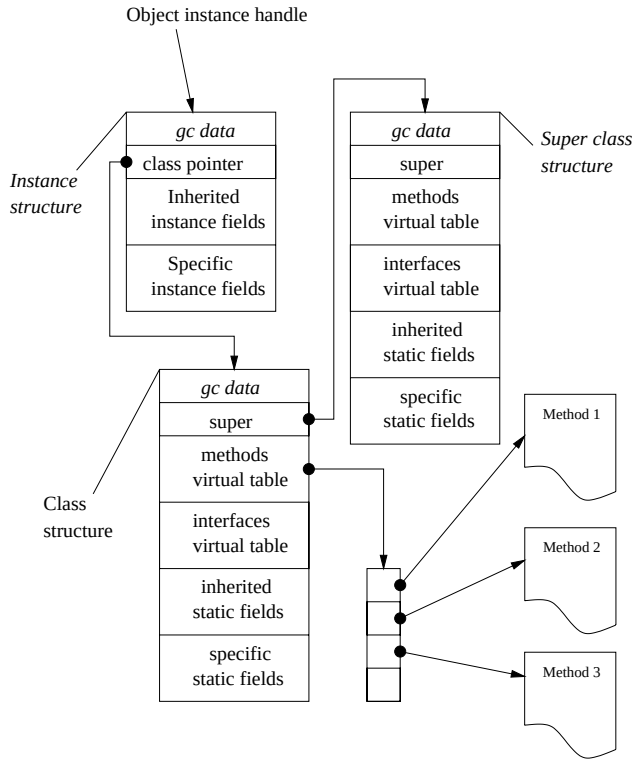


Figure 1. Run-time model of an object.

Consider a very simple Java class with one constructor and one method like:

```
class Dummy {
  int a,b;
  String name;

  Dummy() {
    a = 1;
    b = 2;
    name = new String("dummy");
  }

  String getName() {
    return name;
  }
}
```

This results in the following C code:

```
struct DummyClassStruct {

      // GC data

      // Super class ptr
   struct ObjectClassStruct *super;

      //methods virtual table ptr
   void* (**methodTblPtr)();

      //interfaces virtual table ptr
   void* (**interfaceTblPtr)();

};
typedef struct DummyClassStruct DummyClass;

struct DummyInstanceStruct {

      // GC data

      // Class ptr
   struct DummyClassStruct  *classPtr;

      // Primitive type fields
   int a;
   int b;

      // Field of type String
   REF(struct StringInstanceStruct) name;
};
typedef struct DummyInstanceStruct
                    DummyInstance;
```

The `REF(a)` macro is explained in section 5.

3. Real-Time Kernel

The kernel made is a preemptive multi-threaded kernel with a fixed priority based scheduler. Effort has been made in creating predictable lower and upper bounds on each function in the kernel. Worst case execution times of operations affecting context switch, interrupts, and initialization of threads are made to be affected by the number of priority levels and not the number of currently running threads to lower the jitter. Much of the kernel properties are standard, but the structure of the queues, with respect to priorities and execution time, may not be new but we have not seen it elsewhere.

3.1. Thread model

The kernel supports two different types of threads:

Ongoing threads Each ongoing thread has its own stack space and will upon completion be removed from the system. The thread will execute for a small period of time and then be preempted by another thread. These threads have no limitations on which kernel primitives to use, or how common resources are shared.

Periodic threads These threads have a fixed period and may not be preempted by threads that have the same priority, and hence they may not share synchronization primitives. These limitations makes it possible for all periodic threads of the same priority to share a common stack which improves memory consumption.

These types of threads, but not their implementations, are related to [4] which was inspired by [18].

3.2. Priority Queues

When a thread is in a suspended or in a ready state, it is placed in a queue. We used the data structure depicted in Figure 2. The *next pointer* references the next element in the queue. The *last pointer* references the last thread in a group with threads of the same priority. Within each priority level we use the last-in-first-out strategy. To insert an element we only have to go through all priority levels in the worst case, regardless how many threads there are in the queue. We can dequeue and enqueue all threads with the same priority in constant time.

3.3. Scheduling

A thread is assigned a time-slice to execute by the kernel, and control is transferred to that thread during a context switch. The thread can be preempted either by using a function in the kernel or when it has used its time slice. The scheduler is then invoked and chooses the next thread to execute.

The scheduler uses as many ready queues as there are priority-levels in the kernel. Each queue is of FIFO type. When a thread is preempted, after it has used its time slice, it is inserted last in the queue of its priority class. The scheduler then chooses the next thread to execute, by selecting the first element in the queue of highest priority. If the queue is empty, the queue of second highest priority is used and so forth.

3.4. Time

The kernel is interrupted at a given interval and at this time the tick counter is updated. This period is also used as a timeslice, so the kernel preempts the current running thread and reschedules at this time too. There are three primitives in the kernel for a user program to handle time. The tick counter can be read as well as a fine-grained timer. The kernel also provides a primitive for suspending a thread until a certain time.

When the tick counter is increased the kernel also moves any threads waiting for that tick from their suspended state to ready state. A time queue that consists of several queues,

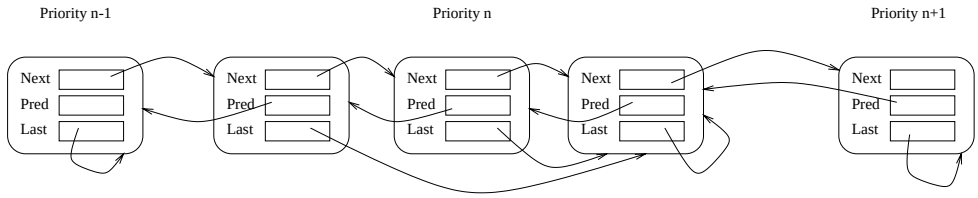


Figure 2. Queue data structure

like the one in Figure 2, are chained together to a long queue. Each subqueue is sorted by priority and the subqueues are chained together in a low-to-hi time order. This way threads of the same priority class that are waiting for the same tick can be handled as a unit. We have balanced the routine that moves threads from the time queue into the different ready queues to always take the worst case execution time. This is done to lower the jitter.

3.5. Synchronization

Synchronization is supported in the kernel by semaphore primitives. Other primitives can in turn be built from semaphores. There exist both binary and counting semaphores. Trying to take a semaphore which counter is zero suspends the current running thread and the kernel reschedules.

To solve the well known problem with priority inversion when using semaphores for mutual exclusion, a mutex form of semaphore is supplied. The most commonly used priority inheritance protocol to defeat priority inversion is the basic priority inheritance protocol. This protocol does, however, permit a thread to be blocked by several lower prioritized threads. Both priority ceiling and immediate inheritance protocol permit only one lower prioritized thread to block another thread and also prohibit deadlock [21]. The mutex primitive uses the immediate inheritance protocol as it is cheaper to implement and has the same worst case behavior as the more elegant priority ceiling protocol.

3.6. Interrupts

To each hardware interrupt a number of threads with different priorities can be attached. When an interrupt is triggered the kernel is invoked and the waiting threads are moved from an interrupt queue to different ready queues based on priority. The same type of queue as in Section 3.2 is used for interrupt queues. The kernel then reschedules, and the selected thread's context is restored.

4. Garbage Collector

The garbage collector (GC) in the run-time system needs to have short predictable worst-case execution times at each invocation to fulfill hard real-time requirements. A new fine-grained incremental mark-compact algorithm, which ensures no fragmentation as well as bounded worst case execution times of all operations, is proposed. An introduction to incremental mark-compact algorithms and GC in general is available in [27]. This GC is described in detail in [10].

4.1. Introduction to the GC

References to the heap stored in processor registers, on the program stack, or in global variables are called *roots*. All objects that are reachable directly through the roots or through a chain of pointers from the roots are considered to be live objects. The GC will trace the reachable objects and mark them. The rest of the objects on the heap are garbage and can be reclaimed.

Work done by the GC is interleaved with normal execution of the user program, often called the *mutator* [26]. We will use the tri-color abstraction introduced in [8] to describe synchronization between the mutator and collector. Each object on the heap is painted in one of three colors:

Black indicates that the object and its immediate descendants have been visited.

Grey indicates that the object must be visited by the collector. Either grey objects have been visited by the collector but not all pointers are scanned, or their connectivity to the rest of the graph has been changed.

White objects are unvisited and at the end of the marking phase considered being garbage.

To make sure that the collector has a coherent view of the heap during the marking phase the following invariant must hold:

No pointer in a black object references a white object.

Coherence is maintained by barriers between the mutator and the heap. These barriers can be either a *read-barrier*, trapping reads, or a *write-barrier*, trapping writes.

4.2. The marking phase

During the marking phase we start by coloring all objects referenced by the root pointers grey. We have a stack of root pointers for each thread which is processed root by root. Each object referenced by a root is inserted into a marking list. The elements in the marking list are the grey objects. We then visit all grey objects and mark the objects referenced by them grey. All pointers in the object are processed and then the visited object is colored black. The procedure is repeated for all grey objects. This way all objects reachable from the roots are visited. In the end of the marking phase all objects are either black or white. As we don't want to risk overflowing the stack by recursively traversing the graph we use Cheney's, [6], non recursive marking strategy with a marking list where the reference to the next object to mark is placed in each object.

If the mutator, during the marking phase, tries to set a pointer in a black object to reference a white object we immediately color the referenced object grey to make the invariant hold. To ensure that this is done a *write barrier* is used that colors the white object grey by inserting it last in the marking list.

4.3. The sweeping phase

During the sweeping phase all black objects are moved into one continuous block at the top of the heap. A *scan pointer* is used, which is initially set to reference the last allocated memory location, from the marking phase of the previous cycle. All objects are processed in a top-down order by decreasing the *scan pointer* with the size of the currently scanned object. Black objects are moved to the top of the heap, and white objects are reclaimed. To be able to traverse the heap in either direction the object size is placed in both the beginning and the end of the objects. When the *scan pointer* reaches the bottom of the heap all objects are processed, the current cycle completed, and the next initiated. In the next cycle the heap is traversed in the opposite direction.

4.4. Allocating new objects

New objects are allocated at the top of the heap. At the end of the current cycle, these objects and the object moved during the sweeping phase, are placed in one continuous block at the top of the heap. These objects will be colored and marked during the following collection cycle. When an object is allocated we need to initialize all pointers in the object to reference *null* through a table. The use of a table is discussed in Section 4.5.

4.5. Moving objects

When an object is moved to a new location we must ensure that all references to that object are changed to reference the new location. Moving the object and changing the references must be done as an atomic operation, to make sure that all accesses to an object are made to the correct copy of the object. As the number of references to an object is not known, we would not get a tight upper bound for worst-case execution time of that operation. To get an upper bound on moving an object we access referenced objects through a table. All pointers reference the target object indirectly through the table. This way only the reference in the table needs to be changed. This is the *read barrier*.

As the entire object needs to be moved all at once, the kernel may be suspended for a too long time to meet hard real-time requirements. We allow preemption during copying of an object, and if we are preempted we restart copying the object when we resume. This is to ensure that the copy of the object contains the most recent data.

4.6. Scheduling of GC work

To be sure that the collector will reclaim garbage at a rate necessary for the mutator never to run out of memory, we a priori calculate a minimum collection rate. As long as the current collection rate is above the a priori calculated worst case, we are ensured never to run out of memory. The operation that can exhaust the heap is memory allocation, and therefore we perform an increment of GC at each allocation. We can in this way make sure that there is space on the heap for the new object. We also want to make sure not to do more collection than necessary to interfere as little as possible with the mutator.

To simplify the discussion we assume that all objects are of the same size. This can quite easily be extended to different sized objects as in our actual implementation. Throughout this discussion we will use the following notation:

S	total number of objects that the heap can hold
W	the amount of GC work done so far during this cycle
W_{max}	the amount of GC work necessary during one cycle in the worst case
E_{max}	maximum number of live objects
A_{max}	maximum number of new objects allocated during one cycle
H_{max}	maximum number of new objects allocated by high-priority threads
A	number of objects allocated so far during this cycle
R_{max}	maximum number of root pointers

We define the minimum GC rate, GCR_{min} as

$$GCR_{min} = \frac{W_{max}}{A_{max}}$$

and current garbage collection rate, GCR as

$$GCR = \frac{W}{A}$$

As long as $GCR > GCR_{min}$ we are ensured not to run out of memory. The work that has to be done during one cycle is divided in three parts: processing the roots, marking all live objects, evacuating the marked objects. The maximum work that has to be done can be written:

$$W_{max} = \alpha \cdot R_{max} + \beta \cdot E_{max} + E_{max}$$

where the coefficients α and β compensate for different costs in processing a root or marking an object compared to evacuating an object. For a given hardware, α and β are constant.

The work W_{max} has to be done during one GC cycle, and how long this cycle is depends on how much memory we have. At each allocation we let the collector perform an increment of GC work. We will start by finding out how long a cycle is. When finishing a collection cycle the maximum number of objects on the heap is the number of live objects from last cycle, E_{max}, plus the maximum number of new objects allocated during that cycle, A_{max}. During the following cycle the mutator may allocate as many as A_{max} new objects. The maximum total number of objects on the heap is therefore $E_{max} + 2 \cdot A_{max}$. As we know how big the heap is, S, we can easily calculate how many allocations we can do in one cycle, without exhausting the heap.

$$A_{max} = \frac{S - E_{max}}{2}$$

We now have an expression for the minimum GC ratio necessary, GCR_{min}:

$$GCR_{min} = 2 \cdot \frac{\alpha \cdot R_{max} + \beta \cdot E_{max} + E_{max}}{S - E_{max}}$$

The current GC ratio, GCR, can be expressed as,

$$GCR = 2 \cdot \frac{\alpha \cdot i + \beta \cdot j + k}{S - E_{max}}$$

where i is the number of processed roots, j the number of marked objects, and k the number of evacuated objects. As long as $GCR \geq GCR_{min}$ we are ensured not to run out of memory.

If we divide the maximum total work that has to be done, W_{max}, by the number of allocations we will do during a cycle, A_{max}, we know how much work that will be done at each allocation. The worst execution time for this work can be calculated and added to the cost for allocating one object.

Even if the amount of work that has to be done during an allocation is small and bounded it can still be too long for us to meet all deadlines. To improve the real-time capabilities of the collector we use the technique proposed by Henriksson, [12], and create a semi-concurrent GC. The threads are divided into groups of high- and low-priority threads. GC work is done interleaved with allocation for the low-priority threads. To improve response time for the high-priority threads we suspend the collector until after the high prioritized threads have executed. We can view the collector as a mid priority thread, that gets to execute after the high prioritized threads. We need to make sure that there is enough free space on the heap for the high-priority threads to allocate new objects without exhausting the heap. We denote the maximum number of objects allocated by high priority threads, assuming they are all released at the same time, H_{max}. If we always have this much space free on the heap we are ensured not to exhaust the heap, even if we don't perform any collection work during high-priority thread allocations. The maximum total number of objects on the heap is now $E_{max} + 2 \cdot A_{max} + H_{max}$. The new minimum GC ratio is:

$$GCR_{min} = 2 \cdot \frac{\alpha \cdot R_{max} + \beta \cdot E_{max} + E_{max}}{S - E_{max} - H_{max}}$$

and the the current GC ratio is changed accordingly.

5. Integration

So far we have designed the kernel, the compiler and the GC. But to make predictable garbage collection possible, we need a tight coupling between the compiled application and the garbage collector.

When using a preemptive mark-compact garbage collector, great care must be taken when handling object references as the currently running thread could be preempted at virtually any time. We must assert that there are no dereferenced object handles whenever the GC starts moving objects around. To fulfill these demands, we must consider all reference manipulations as atomic actions.

We must also inform the GC when new roots of object trees are created, and when they are dismissed. This happens whenever a method is called. Consider the example code below:

```
class Dummy {
  public void aMethod(String s) {
    String aString;
        ...
  }
}
```

To register and unregister these two objects as roots in the GC, we introduce the calls `GC_PUSH(ref)` and `GC_POP(nbr)` which, respectively, pushes a reference *ref* onto the GC stack and pops *nbr* references from the stack. The code example above then results in the following generated code:

```
void Dummy_aMethod(REF(DummyInstance) this,
                   REF(StringInstance) s){
    REF(StringInstance) aString;
    GC_PUSH(this);
    GC_PUSH(s);
    GC_PUSH(aString);
        ...
    GC_POP(3);
}
```

To implement the read-barrier we use a macro `REF(a)`, and for the write-barrier we use a macro `GC_SET(a,b)`.

6. Experimental verification and experiences

We have run two types of tests to verify that our solution is feasible. First, we have measured the predictable timing for a multi-threaded application on a small microprocessor typically found in small embedded systems. Second, we have run some benchmarks on a normal desktop workstation to measure the efficiency of our generated code compared to Java and C++.

6.1. Predictable timing

To verify our techniques, we have implemented a prototype for a very limited target platform; the 8-bit Atmel AVR RISC microcontroller [3]. This platform is very typical for tiny embedded systems, with only 128 KB ROM and 64 KB RAM. The microcontroller used runs at 4 MHz and approaches 1 MIPS per MHz. The kernel and garbage collector allocates a footprint of less than 10 kbytes of ROM and 1 kbyte of RAM. Worst-case execution times of operations in the kernel are summarized in Table 1. If we can predict worst-case execution times and worst-case memory demands of the different threads, in combination with execution times of the kernel operations, we can use generalized scheduling theory [22], to check if the system is schedulable or not.

	Execution time in CPU cycles	
Operation	*Worst*	*Best*
Context switch due to timer interrupt	$963 + 358 \cdot k$	$889 + 346 \cdot k$
Context switch due to voluntary suspension	$740 + 12 \cdot k$	728
Take a mutex	113	113
Give a mutex	$1024 + 12 \cdot k$	1014
Create an object	$234+78 \cdot i+54 \cdot n$	$234+78 \cdot i+54 \cdot n$
Push a root	24	24
Pop j roots	12	12
Reference $null$	8	8
Read barrier	4	4
Write barrier	51	5
Process a root	83	64
Mark an object	$110+78 \cdot i+118 \cdot n$	$110+78 \cdot i+60 \cdot n$
Sweep an object	$169 + 9 \cdot s$	104

Table 1. Measured performance with k priority levels and object size s bytes with n pointers divided into i groups. 1 CPU cycle is 0.25 μs.

6.2. Comparison benchmarks

In order to get some kind of performance measurements of the translated Java code, we have run some benchmark tests on Java2C generated code as well as on a JVM and compiled C++. The benchmarks presented are limited to speed measurements on object allocation and methods calling. So called "number crunching" is not that interesting as Java2C translated arithmetic expressions code will perform very close to the speed of hand-written C-code.

All execution time measurements were performed on a Sun Ultra 5 workstation with 256MB of RAM memory running Solaris 7. The Java compiler used was jikes from IBM and the virtual Java machine was Java HotSpot version 1.3.0. G++ and GCC were of version 2.91.66.

The code generated with Java2C does not contain a garbage collector. This because the GC developed is too tightly coupled to the ARM microprocessor to be easily ported to the SPARC. Adding our predictable garbage collector would give some penalty on the measured execution times, but the magnitude would not change.

The benchmarks were as follows:

Allocation. Allocate 2500 objects. The constructor of each objects also allocates a byte array of size $0 - 5000$.

Virtual methods. Make 1000000 calls to a small virtual method.

Final methods. Make 1000000 calls to a small final method.

Final methods. Make 1000000 calls to a small static method.

The measured execution times are shown in Table 3.

From the results one can see that the performance of the code from our translator is almost as fast as natively compiled C++ code, especially if we could allow some compiler optimizations.

Compiler/Run-time	*Execution time (ms)*			
	Allocation	*Virtual methods*	*Final methods*	*Static methods*
java -Xint	350	13000	13450	11980
java	360	1180	1080	670
g++	90	1720	960	1340
g++ -O2	90	970	70	310
Java2C / gcc	100	2280	2260	1380
Java2C / gcc -O2	100	1040	1000	140

Table 2. Measured execution times for some benchmarks.

What may seem astonishing is that the Java HotSpot performs so well performing method calls. The explanation is that because it is the same method that is called 1000000 times, HotSpot will soon perform JIT compilation.

7. Problems and Future Work

Problems concerning C as the intermediate language is not so much about the C language— which is very allowing, to say the least—but how the C compilers generate machine code. In the current implementation, the mandatory atomicity of object reference manipulations is obtained by using pre-emption points in the code, and by turning off all compiler optimizations. A very interesting problem is to be able to utilize compiler optimization techniques , but that is outside the scope of this paper.

Another problem is to calculate a good upper bound on the maximum number of live objects in the system. A guaranteed upper bound tends to be very pessimistic and will degrade the performance of the system. There is, however, recent work done [19] which provides a much better estimate.

There is still some work to be done in the Java to C translator. Some of the more important features of the Java language that is being implemented are interfaces and exceptions. The implemented object model is also, at the time of writing, a somewhat simplified version of the one depicted in Figure 2.4.

8. Related Work

There has been quite some work done on natively compiling Java, but not much on hard real-time Java for small systems. Sun Microsystems Inc. has published a white paper[24] on using the Java 2 Platform Micro Edition (J2ME) for mobile devices. The J2ME is centered around a small JVM called KVM and aimed at devices with a total memory amount in the range of 128 - 512 KB. A J2ME application can also be compiled to native code and linked to the KVM for better performance. J2ME is, however, not yet suited for use in systems with hard real-time demands.

Various issues concerning real-time behavior in Java are dealt with in *The Real-Time Specification for Java*[5]. There are significant drawbacks in this specification from our point of view, specifically concerning memory management. Instead of adopting a predictable run-time system, it extends Java with a new memory organization. In addition to the normal *HeapMemory*, it adds *ImmortalMemory*, *ImmortalPhysicalMemory* and *ScopedMemory* memory areas which are all treated differently by the automatic memory management system. These additions place responsibility on the programmer to always do the right thing, since a wrongly placed memory allocation type in an application could totally void the real-time behavior of that application.

There has also been some work done on implementing very small and memory efficient Java virtual machines which can be deployed in systems with hard real-time demands [14] but that requires more time and memory.

9. Conclusions

We have shown that it is possible to use Java as a programming language for developing small embedded systems with very limited resources of CPU power and memory. Given a few assumptions on the memory usage of an application, we can also show that hard real-time timing demands are met.

By choosing C as an intermediate language—and choosing a suitable object representation model—we can achieve the efficiency needed for running applications on very small CPUs while still maintaining some platform independency.

By combining natively compiled Java with a very small and efficient RT kernel and a pre-emtive garbage collector we can write and test multi-threaded programs in a normal Java runtime environment which can later be compiled for small hard real-time systems.

10. Acknowledgments

VINNOVA (formerly NUTEK, the Swedish Board for Tech. R&D) is acknowledged for financial support. We are grateful for input and feedback from Anders Blomdell (@control.lth.se), and we thank Klas Nilsson (formerly @control.lth.se and @abb.com, now @cs.lth.se) for inspiring suggestions to work in this direction. We also thank our GC-expert Roger Henriksson for many valuable comments.

References

[1] *IEC 1131-3, Programmable controllers, Part 3: Programming Languages*. International Electrotechnical Commission, 1992.

[2] K. J. Åström and B. Wittenmark. *Computer Controlled Systems: Theory and Design*. Prentice Hall, 3rd edition, January 1997.

[3] *ATmega103(L) Preliminary (Complete)*, Jan. 2000. *http://www.atmel.com/acrobat/doc0945.ps*.

[4] L. A. Bigagli. Real-time java, - a pragmatic approach. Master's thesis, Department of Computer Science, Lund Institute of Technology, October 1998.

[5] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[6] C. J. Cheney. A non recursive list compacting algorithm. *Commmunications of the ACM*, 13(11), 1970.

[7] Cygnus. The GNU compiler for the java language. *http://sourceware.cygnus.com*.

[8] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11), 1978.

[9] J. Eker. A tool for interactive development of embedded control systems. In *Preprints 14th World Congress of IFAC*, Beijing, P.R. China, 1999.

[10] T. Ekman. A real-time kernel with automatic memory management for tiny embedded devices. Master's thesis, Department of Computer Science, Lund Institute of Technology, November 2000.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1st edition, August 1996.

[12] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology / Lund University, 1998.

[13] Jove, super optimizing deployment environment for java, July 1998. Instantiations Inc. *http://www.instantiations.com*.

[14] A. Ive. Implementation of an embedded real-time java virtual machine prototype. Licentiate thesis, Department of Computer Science, Lund Institute of Technology, 2001. In preparation.

[15] P. Kouznetsov. Jad - the fast java decompiler, 2000. http://www.geocities.com/SiliconValley /Bridge/8617/jad.html.

[16] Java-cc parser generator. Metamata Inc. *http://www.metamata.com*.

[17] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient java environment mixing bytecode and compiled code. *http://www.irisa.fr/accueil/index_uk.htm*.

[18] K. Nilsen and S. Lee. Perc real-time api, July 1998.

[19] P. Persson. Predicting time and memory demands of object-oriented programs. Licentiate thesis, Department of Computer Science, Lund Institute of Technology, April 2000.

[20] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and C. A. Watterson. Toba: Java for applications, a *Way ahead of Time* (wat) compiler. *http://www.cs.arizona.edu*.

[21] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), 1990.

[22] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. In *Proceedings of the IEEE*, volume 82, 1994.

[23] N. Shaylor. Jcc - a java to c converter. *http://www.geocities.com/ CapeCanaveral/Hangar/4040/jcc.html*.

[24] Java 2 platform micro edition (j2me) technology for creating mobile devices. http://www.java.sun.com, May 2000. Sun Microsystems Inc. White Paper. *http://www.java.sun.com*.

[25] Deploying high-performance and flexible server-side applications, an introduction to towerj3. Tower Technology Corporation. *http://www.towerj.com*.

[26] P. L. Wadler. Analysis of an algorithm for real time garbage collection. *Communications of the ACM*, 19(9), 1976.

[27] R. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of IWMM'92*. Springer-Verlag, 1992.

[28] Turboj. WindRiver Inc. *http://www.windriver.com*.

Designing Agents for Systems with Adjustable Autonomy

Paul Scerri and Nancy Reed

February 27, 2001

Intelligent agents are starting to assume more responsibility in more critical tasks. Agents do not always work as required, however in some systems it is infeasible to simply "stop" an agent when its behavior is unsatisfactory. Instead it is desirable to reduce the agent's autonomy and give control of the unsatisfactory aspects of the agents task to a more competent entity. Adjustable Autonomy (AA) is a recent idea that means that the autonomy of agents and humans in a system can vary dynamically. In this paper we look at the relationship between the design of agents and the design of AA for an intelligent system.

1 Intelligent Agents

Many of the recent, exciting developments in artificial intelligence (AI) have centered around the concept of an *agent*. An agent is an autonomous entity that senses its environment and acts intelligently and pro-actively towards its goals[17, 2]. An important characteristic of an agent is that it has the ability to take actions that affect its environment[7]. Some agents sense and act in purely software environments (e.g. an operating system monitoring agent[16]), while others have a physical embodiment and inhabit a physical environment (e.g. a museum tour guide robot[3]). Because an agent can act, it can be assigned tasks that can potentially be done more quickly, efficiently, cheaply or safely than a human can do them. Thus humans are freed from menial, dangerous and/or boring tasks. Despite the long list of successful agent applications, it is likely we have only scratched the surface of the possibilities intelligent agents have to change the way we live our lives[6, 8]. In the future, autonomous agents will take on more complex tasks and act more intelligently and more decisively.

An *intelligent system* is one consisting of intelligent agents and, possibly, humans and/or other conventional software. Generally in an intelligent system the assignment of responsibility and authority, i.e. autonomy, is either fixed or switches between a small number of fixed configurations. In an intelligent system with AA the system can flexibly configure the assignment of autonomy between the people and agents to best fit the situation.

2 Adjustable Autonomy

As AI technology develops, the autonomy agents have increases proportionally, i.e. more capable agents are given more authority and more responsibility. In most cases, once an agent's autonomy is determined by a system designer the agent is left to fulfill its responsibilities according to its specification, within the bounds imposed by its authority and capabilities. In complex environments, an agent will be faced with a vast range of situations, in each of which it must act correctly. However, it is unlikely that any agent has appropriate reasoning mechanisms, sensors and actuators to act appropriately in *all* situations it could potentially face[15]. There will be some situations in which the agent will make unacceptable decisions and, hence, take unacceptable actions – potentially causing harm. Some situations are so unlikely that agent designers reasonably ignore them, hence the agent is simply not designed to handle them properly. Other situations might commonly occur but to build software or robotic hardware to properly handle the situation may be considered too expensive or time-consuming. Yet other situations will be unacceptably handled by an agent due to "bugs" in its software. Importantly, the same reasons that cause an agent to fail to handle a situation satisfactorily, may cause it to not even detect that it has encountered a situation it is not capable of handling satisfactorily. For example, an autonomous robot that cannot detect a wall to go around it may not detect that it has bumped into the wall.

The more autonomy an agent has, over more complex and important tasks, the more potentially serious the consequences are when its behavior is unacceptable. However, because the agent is autonomous and because the agent may not detect its own inadequacy, "killing" the agent may be the only way of preventing the incorrect actions. Completely stopping an agent means another entity needs to take over the agent's responsibilities, something that may be unreasonable in a complex physical system (e.g. a spacecraft). A more desirable scenario is if only incorrect *parts* of the agent's activity are taken over while other parts continue to function normally.

Thus, agent developers are faced with a challenging dilemma. For some applications, autonomous agents can be very useful in very many situations, most of the time. But in a small number of situations, a small percentage of the time, agent behavior will be unacceptable and potentially have serious consequences.

Adjustable Autonomy (AA) is a recent idea meaning to *dynamically change* the autonomy of the intelligent entities, both agents and humans, in a system. Instead of the responsibility and authority of entities being fixed at design time, they can be changed to best configure the system's autonomy to the current situation. The idea is to dynamically assign autonomy to best leverage the constituent entities' strengths and avoid their weaknesses. Thus, *an AA system is an intelligent system where the distribution of autonomy is changed dynamically to optimize overall system performance.* For example, if a human pilot notices that a collision with a flock of rare ducks the agent cannot detect is imminent, the pilot might like to slightly alter the aircraft's course without taking over all

the details of its functioning. Flexible assignment of autonomy means a system can deal with a wider range of situations more effectively. Thus, AA allows the intelligence and autonomy of agents to be fully exploited without being stuck with their inadequate decision making when situations occur the agents cannot handle (or humans could handle better).

3 Conceptual Model of AA

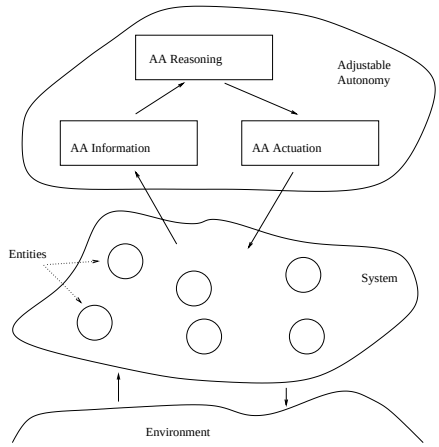


Figure 1: The conceptual relationship between AA and an intelligent system.

A system with AA can dynamically change which entities are responsible for the achievement of which goals by changing decision making responsibility over time. Further, a system with AA can dynamically change the authority constituent entities have to take on particular goals. AA mechanisms manage the changing autonomy by determining appropriate changes in autonomy and implementing those changes.

A key problem to be addressed when building AA is to determine an appropriate distribution of autonomy and provide mechanisms to realize the autonomy changes.

The distribution of autonomy should change according to the current situation and sub-goals, reconfiguring so as to best organize the system resources to achieve the systems goals. Conceptually the task of changing autonomy can be broken into three parts:

- AA Information (AAI) : Collection of the information relevant to the AA decision making.
- AA Reasoning (AAR) : Reasoning about what autonomy changes could or should be made.

- AA Actuation (AAA) : Realization of the decisions made by the AAR.

The conceptual AA model and its relationship to a system are shown in Figure 1. AAI provides information on prevailing environmental conditions and the current system state and potential (referred to as *context* in [4]) as are relevant to AAR. The AA reasoner determines what changes in the autonomy distribution will lead to better system performance with respect to its goals. AAR might be done either by a human or in software. Finally, the AAA provides the mechanisms for implementing the decisions of the AA reasoning, i.e. it provides the mechanisms for realizing changes in authority or transfer of responsibility.

The AAI and AAA tightly constrain the design and potential of AAR. Any information not supplied by AAI cannot be used in determination of appropriate autonomy configurations. Likewise, any change that cannot be realized by AAA should not be decided on by AAR. In turn AAI and AAA are both tightly constrained by the services provided to them by the entities in the system. Any information the entities cannot provide cannot be supplied by AAI to AAR. Similarly, any autonomy change the entities cannot accept could not be implemented by AAA. Hences *the services provided by the entities are critically important to the building of an AA system.* Obviously, we are limited to designing the services that software entities provide (as the services of the human entities are fixed).

Consider an analogy to a management consultant at an organization. The consultant's job consists of collecting information, making decisions about organizational changes and implementing those changes. No matter how good the consultant is at their job they rely on employees in the organization to inform them (either implicitly or explicitly) of the current running, goals, etc. of the organization in order to make decisions. If the employees supply limited, insufficient, incorrect or misleading information the consultant job is significantly harder and their results are likely to be disappointing. Once the consultant makes a decision it needs to be implemented. No matter how good the decision, if it is not accepted and appropriately implemented, the decision is worthless. Implementation of AA works in the same way – if the entities do not supply appropriate information and properly implement decisions, the best AAR is useless.

4 Agent Design and AA

Agent designs differ in the way that information is represented in the agent and how easily it can be extracted in an understandable (to the AAR) manner. Whether the AAI services are simple, just providing access to particular parts of the agent's reasoning, or very complex, needing complex algorithms to extract information from the agent, depends on the design of the agent. The AAI guidelines presented below aim to guide designers to create agents where as much useful, understandable information as possible can be gathered by simply inspecting the data structures of a running agent. The guidelines try to avoid the need for complex algorithms to extract information from a running agent.

AA Actuation services implement autonomy changes decided on by the AAR. Only those autonomy changes that can be realized by AAA services can be decided on by AAR. Hence, the limitations of the AAA services strictly limit the useful conclusions possible by the AAR and the overall AA.

Further, the more elegantly and smoothly the agent incorporates any autonomy changes decided on by AAR into its ongoing behavior, the better the behavior of the overall system. For example, imagine a team of (human) furniture removalists carrying a piano up a staircase. If their foreman yells out from another room that one removalist is being re-assigned to another job he will not (hopefully) simply let go of the piano and walk off (leaving his colleagues squashed under a piano at the bottom of the stairs!) but will make the piano safe before moving on. Thus, the behavior of the overall "furniture removal" system is successful because the worker changing tasks switches between tasks in a reasonable manner. The same idea applies to AA where smooth autonomy changes mean better system behavior. If the agent could exhibit similar "common sense" behavior to the furniture removalist the AAR's task is made simpler because changes can be made without (unnecessary) consideration given to transitions of the agent's behavior.

When designing intelligent agents many competing requirements need to be reconciled. Not all the requirements are related solely to the observable behavior of the agent. For example, there may be particular verifiability, simplicity or computational requirements on an agent[11]. Designers, implicitly or explicitly, follow guidelines when attempting to meet some requirement with a design. A guideline is "a statement or other indication of policy or procedure by which to determine a course of action"[1]. For example, a (simple) guideline to minimize computational requirements for an agent might suggest avoiding algorithms involving significant amounts of search. By following appropriate guidelines designers have a principled, justifiable reason for believing that their design will meet its requirements. For example, if a design avoids extensive use of search algorithms a designer can argue, with justification, that once implemented the design will be computationally efficient (according to the above guideline).

5 Guidelines

We capture our design knowledge of agents for AA systems gained via the implementation of two complete AA systems[13, 10] in a set of guidelines. If followed when designing agents for AA systems, the guidelines should lead to agents with features that makes AA easy to implement.

Three guidelines provide advice for designing agents which will lead to easy to build, high quality AAI:

- *Explicit Information Guideline* : Represent the agent's reasoning process and reasoning state explicitly and in a format close to the format that will used by AAR.

- *Software Engineering Guideline* : Following good software engineering practices in agent design makes it easier to build AA.
- *Design Information Guideline* : Represent information above and beyond the information needed by the reasoning process such that it explains design decisions implicit in the reasoning process.

These guidelines summarize design strategies for building agents whose behavior can be most flexibly and easily changed online:

- *Deterministic Execution Guideline* : Make the reasoning process of the agent as deterministic (and hence predictable) as possible.
- *Explicit Behavior Guideline* : Represent behavior as explicitly as possible and in a format that requires the least translation.
- *Building Blocks Guideline* : Divide overall behavior into small pieces that are related to each other in a very semantically clear and simple way.
- *No Extra Mechanisms Guideline* : The AAA should not use mechanisms other than the normal reasoning mechanisms used by the agents.
- *Design for Failure Guideline* : The agent should be designed so that if any part of it fails at any time, its behavior will degrade gracefully.

6 Discussion

These ideas have been implemented in two systems. The first is a system for creating agents for interactive simulations called EASE[13, 14]. In EASE, the AA is used to give a user runtime control over the agents in a simulation[12]. Two domains are being investigated: air combat using Saab's air-combat simulator, TACSI[9], and football using the RoboCup simulator[5].

The second implemented system is the E-Elves where intelligent agents are used to help human users carry out every day tasks in a human organization[10]. The agents can look after tasks like rescheduling meetings when a user is delayed, ordering lunch and finding presenters for meetings.

Assessing the utility of the guidelines is a two step process. First, we identify the agent features that are a direct consequence of following the guidelines. For example, a particular reasoning mechanism might be designed in a specific way because of the advice of a particular guideline. Then we look at the impact of that agent feature on the ease with which the AA was built. If agent features that are a result of following the guidelines make building AA easier then the utility of the guidelines is demonstrated.

Acknowledgments

This work is supported by The Network for Real-Time Research and Education in Sweden (ARTES), Project no. 0055-22, Saab Corporation, Operational

Analysis Division, the Swedish National Board for Industrial and Technical Development (NUTEK) under grants IK1P-97-09677, IK1P-98-06280 and IK1P-99-6166, and the Center for Industrial Information Technology (CENIIT) under grant 99.7.

References

[1] *The American Heritage Dictionary of the English Language.* Houghton Mifflin Company, 1996.

[2] Jeffery Bradshaw. An introduction to software agents. In *Software Agents*, pages 3–49. MIT Press, 1997.

[3] Wolfram Burgard, Armin Cremers, Dieter Fox, Dirk Hähnel, Gerhard Lakemeyer, Dirk Schulz, Walter Steiner, and Sebastian Thrun. The interactive museum tour-guide robot. In *Proceedings of AAAI'98*, pages 11–18, 1998.

[4] H. Hexmoor. Case studies of autonomy. In *Proceedings of FLAIRS 2000*, pages 246–249, 2000.

[5] Itsuki Noda. Soccer server: A simulator of RoboCup. In *Proceedings of AI Symposium'95*, Japanese Society for Artificial Intelligence, December 1995.

[6] B. Pell, E. Gamble, E. Gat, R. Keesing, J. Kurien, W. Millar, P. Nayak, C. Plaunt, and B. Williams. A hybird procedural/deductive executive for autonomous spacecraft. In *Proceedings of the second international conference on autonomous agents*, pages 369–376, 1998.

[7] S. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Inc., 1995.

[8] P. Rybski, S. Stoeter, M. Erickson, M. Gini, D. Hougen, and N. Papanikolopoulos. A team of robotic agents for surveillance. In *Proceedings of the fourth international conference on autonomous agents*, pages 9–16, 2000.

[9] Saab AB, Gripen, Operational Analysis, Modeling and Simulation. *TACSI - User Guide*, 5.2 edition, September 1998. in Swedish.

[10] P. Scerri, D. Pynadath, and M. Tambe. Adjustable autonomy in real-world multi-agent environments. In *Proceedings of the Fifth international conference on autonomous agents (Agents'01)*, 2001. to appear.

[11] P. Scerri and N. Reed. Engineering characteristics of autonomous agent architectures. *Journal of Experimental and Theoretical artificial intelligence*, 12(2):191–212, 2000.

[12] P. Scerri and N. Reed. Real-time control of intelligent agents. In *Technical Abstracts of Technical Demonstrations at Agents 2000*, 2000.

[13] Paul Scerri and Nancy Reed. Creating complex actors with EASE. In *Proceedings of Autonomous Agents 2000*, pages 142–143, 2000.

[14] Paul Scerri and Nancy Reed. The ease actor development environment. In *Proceedings of the Workshop of the Swedish AI Society, SAIS'2000*, 2000.

[15] B. Shneiderman. *Designing the User Interface.* Addison Wesley, 1998.

[16] Hongjun Song, Stan Franklin, and Aregahegn Negatu. Sumpy: A fuzzy software agent. *on line at : http://www.msci.memphis.edu/ŝongh/publications/sumpy.html*, 1995.

[17] Michael Wooldridge and Nicholas Jennings. *Intelligent Agents*, chapter Agent Theories, Architectures and Languages: A Survey. Springer-Verlag, 1994.

On recovery and consistency preservation in distributed real-time database systems

Sanny Gustavsson, Sten F. Andler
Högskolan i Skövde
{sanny,sten}@ida.his.se

Abstract. Consistency preservation of replicated data in distributed databases is nowadays a well-covered topic. Both pessimistic approaches (such as two-phase commit) and optimistic approaches (such as eventual consistency) have been thoroughly investigated. A current research trend is to move towards optimistic consistency preservation mechanisms, where immediate consistency at each transaction commit is traded off for increased predictability, availability and performance. In this paper, we consider the impact of optimistic consistency preservation techniques on the recovery process in distributed real-time database systems. We identify uninvestigated problems within this field, such as the absence of timely and optimistic recovery mechanisms, and our lack of understanding of the requirements to be met by applications operating in eventually consistent systems. We also suggest approaches for further research on these subjects.

1 Introduction

The distributed real-time database field of research is interesting for several reasons. Not only is the demand in industry for timely and dependable database systems high, the inherent complexity of distributed and concurrent systems coupled with the need for predictability in real-time systems also creates unique and interesting problems. The project described herein addresses the problem of recovering the contents of a crashed node in a distributed real-time database. We are particularly interested in recovery in eventually consistent databases, i.e., databases where availability, predictability and performance are improved by allowing replicas to be temporarily inconsistent. Such databases are interesting in that they must be convergent, which means that if all transactions in the systems are quiesced, all replicas must eventually become consistent. Eventually consistent databases also put restrictions on the applications that use the database, since they must be tolerant of stale data.

We argue that many systems today (e.g., banking systems and base stations for mobile telephony) must allow for temporary inconsistencies in order to provide a reasonable level of availability. These systems must, however, also be able to guarantee that a consistent state will eventually be reached.

In the following section we describe the project background and define the problem area. Section 3 lists the problems that we want to address, while section 4 suggests some possible approaches to solving these problems. Finally, section 5 contains our conclusions, including a discussion on future work and how our research fits in with other projects within the same field.

2 Background

A *distributed system* consists of several autonomous but interconnected processing elements (denoted *nodes*), which work together to achieve a common goal. Some important properties of distributed systems are:

- **Localized processing power**

  Computations can be performed at the site where they are needed, for example close to sensors and/or actuators, which means that less cabling is needed and that communication costs are decreased compared to a centralized system. This property is especially important in embedded systems.

- **High fault tolerance**

  In a distributed system, it is possible to replicate data and services, so that if a node goes down, the data and services of that node can still be available to the user on another node. This is an important property for systems that require high availability.

Many distributed systems contain some form of database. In a *distributed database*, the data is dispersed on several different nodes. Of particular interest to this paper is the concept of data replication. In a distributed database, what is seen by applications as a single, logical database object can in fact be stored as several physical copies, *replicas*, of that object. These replicas may be stored on different nodes to achieve a high degree of fault-tolerance and availability.

In a *real-time system*, correct behavior implies not only functional correctness, but also conformance to timeliness requirements, such as task deadlines and minimum delays. Many real-time systems are also *embedded* systems, which means that their primary function is not information processing (Burns & Wellings 1997). An example of such a system is a microprocessor-controlled washing machine.

A *distributed real-time system* must deal with not only the issues inherited from both these system classes, but also with several unique problems that exist in few other system types, such as predictable communication, multi-node load balancing, and timely consistency management.

2.1 Consistency preservation

As discussed in the previous section, a database object in a distributed database may be represented by several physical replicas on different nodes in the system. For the database to remain consistent, all such replicas must be kept identical and adhere to all constraints in the database specification. This means that any updates on the logical object must be reflected by all the physical replicas.

There are two main approaches to maintaining consistency in a distributed database – pessimistic and optimistic. We say that the pessimistic approaches support immediate consistency, while optimistic approaches only guarantee eventual consistency. These concepts are elaborated in the following two paragraphs.

- **Immediate consistency**

  Pessimistic techniques for consistency preservation ensure that all replicas of all logical database objects accessed by a transaction are consistent when that transaction commits, i.e., consistency is *immediately* achieved. There are several well-defined techniques for this, the most basic and well known of which is the *two-phase commit* protocol (Gray & Reuter 1993).

- **Eventual consistency**

  Optimistic consistency preservation techniques are based on the concept of *eventual* consistency. The idea is to trade off immediate consistency at each transaction commit for increased predictability, availability, and performance. This means that a transaction may commit updates to a local replica of a logical database object without propagating the updates to the nodes containing additional replicas of that object. Thus, the local performance and availability are increased, since all overhead associated with communication protocols and distributed commit protocols is removed. Examples of optimistic consistency preservation protocols are Distributed Optimistic Two-phase Locking (Carey & Livny 1991) and Lazy Replication (Breitbart & Korth 1997).

2.1.1 The consistency trade-off

Although a large research effort has been put into pessimistic consistency preservation techniques, the field is moving towards optimistic techniques for the performance benefits inherent in such techniques. We argue that in many of today's large-scale systems, temporary inconsistencies are inevitable. For example, although banking is a field where the demands on consistency could be assumed to be strict, extracting money from an ATM would often be impossible (due to, e.g., network partitions) if details of the transaction would have to be propagated to all nodes (i.e., bank offices) in the system before any money could be delivered. Thus, to increase the availability of the ATMs, the node at which the transaction was performed must be allowed to be temporarily inconsistent with the rest of the nodes in the system. In this type of system, eventual consistency is required. In the ATM system, for example, money must not be irrecoverably lost.

Since conflicts will occur in any eventually consistent system (e.g. the last $100 may be extracted from a given bank account simultaneously at two different nodes), conflict detection and resolution mechanisms must exist in the system. Also, applications must be aware of (and tolerate) the fact that they may be working with stale or inconsistent data. We say that applications must be convergent.

2.2 Database recovery

When a distributed database node crashes, its main-memory contents are lost, and must somehow be recovered. Traditionally, the most common method is to keep, on stable storage, a *log* of all updates performed on the node's data (Gray & Reuter 1993). When a node recovers, it reads and applies all the log entries to the database. Since the log can grow arbitrarily large, logging is nearly always accompanied by the taking of periodic *checkpoints*, i.e., complete database images written to stable storage. Checkpointing optimizes the time needed for recovery, since only the log records postdating

the most recent checkpoint have to be read during the recovery process.

2.2.1 Distributed recovery

In distributed databases, the consistency of the recovered data must also be considered. If immediate consistency is enforced, once the node is reintegrated in the system, its data must be consistent with the data on the other nodes. If only eventual consistency is required, the reintegrated node must still be in a state that is eventually consistent, i.e., a state that would become consistent after a bounded time period in a quiescent system.

Diskless distributed recovery
For some embedded systems, such as those working in environments with electromagnetic radiation or heavy vibrations, it may not be practical to use disks as permanent storage for checkpoints and logs. Also, many disk drivers are unpredictable, which makes them unsuitable for real-time systems. In addition, disks add to the cost of building the system. We thus see a need for recovery mechanisms that do not depend on disks.

In a replicated database, the inherent redundancy in the system can be used for recovery. Instead of reading a checkpoint from disk, the contents of another node in the system (the *recovery* source) can be copied to the recovering node (the *recovery target*). If updates are performed at the recovery source concurrently with the recovery process, those updates must also be transferred to the recovery target after the database image has been copied (Leifsson 1999).

3 Problem definition

This section gives a brief introduction to the aspects of diskless distributed recovery that we have chosen to focus on. For a more in-depth description of these topics, see Gustavsson (2000).

3.1 Timely recovery

For a recovery technique to be suitable for a real-time system, it must be timely. So far, few recovery algorithms prioritize or even consider timeliness. The only recovery algorithm designed for real-time systems that we have found (Song et al. 1999) does not consider consistency issues. We would like to examine methods for recovery in eventually consistent databases, focusing on their timeliness, i.e., whether they can be made predictable and sufficiently efficient for inclusion in a real-time system.

3.2 Consistency preservation in recovery

Any recovery mechanism in a distributed database must consider consistency preservation to ensure that the node is integrated correctly with the rest of the system once recovery is complete. The easiest way to ensure consistency is to quiesce the rest of the system while recovery is taking place. However, this may not be an option in systems that require high performance and availability.

If a diskless recovery mechanism (in section 2.2.1) is used, special care must be taken to preserve consistency, since the recovery source's view of the database may be updated as it is being sent to the recovery target. We wish to investigate how this affects the recovery process, especially in an eventually consistent system.

3.3 Application constraints

While eventual consistency improves several important attributes of a system, it also restricts the types of applications that may operate in the system. As discussed in section 2.1.1, applications in an eventually consistent system must be able to tolerate the fact that they may be working with stale or inconsistent data. We wish to establish a formal definition of the necessary properties of convergent applications.

4 Approaches

In this section, we present some approaches that we plan to take in tackling the problems of the previous section.

4.1 Creating a taxonomy

Initially, we aim to formulate a taxonomy for distributed recovery in different kinds of systems. For example, recovery (especially diskless recovery) in a quiescent system differs significantly from recovery in a non-quiescent, eventually consistent database. We need to distinguish different recovery scenarios by identifying the system properties that affect the recovery mechanism.

4.2 Modifying existing techniques

We intend to examine already existing recovery techniques to see whether they can be refined for use in a diskless and eventually consistent real-time database system. Some initial work in this direction has already been performed (Leifsson 1999, Gustavsson 2000). For example, according to Leifsson (1999), sending the data from a recovery source to the recovery target in diskless recovery is

analogous to taking a fuzzy checkpoint (Li et al. 1995).

4.3 Analysis and theoretical proofs

To add weight to any claims that we make about a recovery algorithm's timeliness and/or non-interference with conflict detection and conflict resolution protocols, we need to logically prove that it has those properties. For us to be able to perform the required analysis and proofs, we need a way of formally reasoning about these subjects. Finding suitable methods to do this, or, if required, devising new ones will be an important part of our project.

4.4 DeeDS integration

Finally, we wish to verify the feasibility of the diskless recovery algorithm by implementing it in a real-world system and running benchmark tests. DeeDS, a distributed real-time database system prototype developed by the distributed real-time systems research group at university of Skövde (Andler et al., 1996), is well suited for such an implementation.

5 Conclusions

This project is in its early stages. We have completed an initial literature survey and analysis (Gustavsson 2000) and are currently investigating suitable methods. In the remainder of this section, we briefly discuss our future work and how it relates to existing research in this field.

5.1 Future work

The research problems that we wish to investigate are in three distinct categories:

- General problems in distributed recovery
  - Finding timely recovery algorithms
  - Exploring the relationship between consistency preservation and recovery
- Eventual consistency problems
  - Devising a recovery method for eventually consistent databases that does not interfere with consistency preservation
  - Defining constraints on applications that operate in an eventually consistent database
- Diskless recovery extensions
  - Relaxing some of the assumptions made by Leifsson (1999), such as allowing multiple node failures or incremental recovery
  - Performing benchmark test of a real-world implementation of the diskless recovery algorithm

5.2 Related work

Work exists on optimistic consistency preservation approaches that are similar to our notion of eventual consistency. For example, the independent update algorithm briefly described by Ceri et al. (1994) shares many aspects with eventual consistency. Unfortunately, little is said about the implications that using the independent update algorithm has on the database system and its applications. The only things mentioned by the authors are that the reconciliation (conflict resolution) algorithm should ensure one-copy serializability, and that the applications must be capable of handling stale data.

Concurrency control, which is similar to consistency preservation, is also a well-covered area. However, we have been able to find little or no such research that focuses on the interaction between these techniques and recovery.

We also argue that diskless recovery is a fresh and interesting area that requires further exploration. Also, there is, to our knowledge, no research on application requirements for eventually consistent databases.

5.3 Contributions

Our work focuses on aspects of distributed recovery that have, so far, received very little attention. We look at timely recovery, which should be increasingly important as the number of systems with demands on timeliness continues to grow. Similarly, many embedded systems operate in hostile environments, where the use of disks may not be an option. In such systems, a diskless recovery mechanism may eliminate the need for stable storage, while decreasing the cost of the system. Finally, the research community as a whole is moving towards optimistic approaches for consistency preservation, as we realize that in many real-world systems, immediate consistency cannot be enforced while meeting the high demands on performance and availability. Therefore, node recovery in an eventually consistent database is a relevant and interesting topic.

Acknowledgments

We extend our thanks to Jonas Mellin and Robert Nilsson, who were both able to give us feedback on short notice.

References

Andler, S., Hansson, J., Eriksson, J., Mellin, J., Berndtsson, M. & Eftring, B. (1996), DeeDS towards a distributed and active real-time database system, SIGMOD Record 25(1), 38-40

Breitbart, Y. & Korth, H. (1997), Replication and consistency: Being lazy helps sometimes, in Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, Tucson, Arizona.

Burns, A. & Wellings, A. (1997), Real-Time Systems and Programming Languages, Addison-Wesley.

Carey, M & Livny, M. (1991), Conflict detection tradeoffs for replicated data, ACM Transactions on Database Systems 16(4), 703-746.

Ceri, S., Houtsma, M., Keller, A. & Samarati, P. (1994), A classification of update methods for replicated databases, Technical Report CS-TR-91-1392, Stanford University, Computer Science Department.

Gray, J. & Reuter, A. (1993), Transaction Processing: Concepts and Techniques, Morgan Kaufmann, chapter 10.

Gustavsson, S. (2000), On recovery and consistency preservation in distributed real-time database systems (HS-IDA-MD-00-015), Master's thesis, University of Skövde.

Li, X., Eich, M., Joseph, V., Gulzar, Z., Corti, C., Nascimento, M. & Peltier, A. (1995), Checkpointing and recovery in partitioned main memory databases, in Proceedings of the International Conference on Intelligent Information Management Systems, Washington, DC, June 7-9, pp. 59-63.

Leifsson, Æ. (1999), Recovery in distributed real-time database systems (HS-IDA-MD-99-009), Master's thesis, University of Skövde.

Song, E-M., Kim, Y-K., Ryu, C., Choi, M., Kim, Y-K., Jin, S-I., Han, M-K. & Choi, W. (1999), No-log recovery mechanism using stable memory for real-time main memory database systems, in Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications.

POSITION STATEMENT

Switched Real-Time Communication for Industrial Applications

Hoai Hoang (*)

School of Information Science, Computer and Electrical Engineering,
Halmstad University, Box 823, S-301 18 Halmstad, Sweden
email: Hoai.Hoang@ide.hh.se, Phone: +46 35 16 71 00, Fax: +46 35 12 03 48

1. Introduction

An important trend in the networking community is to involve more switches in the networks (e.g., LAN, Local Area Networks) and pure switched-based networks becomes more and more common. At the same time, the industrial communication community has a strong will to adapt LAN technology (e.g. Ethernet) for use in industrial systems. The involvement of switches does not only increase the performance; the possibility to offer real-time services is also improved. Now when the cost of LAN switches has reached the level where pure switched-based networks have become affordable, the collision possibility in IEEE 802.3 (Ethernet) networks can be eliminated and methods to support real-time services can be implemented in the switches without changing the underlying wide-spread protocol standard.

This research aims to provide such methods with the focus on industrial applications. The research is motivated by the large interest of using cheap and simple technology (like Ethernet) in industrial and embedded systems. Ethernet has already today been introduced to these applications but has, at the same time, introduced problems with (or lack of) real-time services and analyzability.

The main research question is how to form methods to be able to support typical industrial real-time traffic (e.g., small periodic messages) without changing the underlying protocols and while still supporting existing higher-level protocols for non-real-time traffic (e.g., web based maintenance which is highly desirable to coexist with the real-time traffic). Other important research questions are what degree of service (throughput, latency, delay jitter etc.) one can expect from these communication systems and how to form methods to increase analyzability (e.g., by introducing determinism).

2. Applications

The research efforts on real-time communication over non-real-time LAN technologies have so far been concentrated on multimedia and similar applications while there is a large need of research efforts in the field of industrial systems. Our focus is on industrial and embedded systems. Application examples are industrial automation, Computer Integrated Manufacturing (CIM), radar signal processing systems, airplanes, process control, and telecommunication equipment. Other applications which also might be of interest are home networks and broadband access networks, were a mix of real-time and non-real-time traffic is expected.

(*) A large portion of this document is copied from the ARTES project application authored by Magnus Jonsson and Bertil Svensson.

3. Plan of research

The research work can be described as two parallel tracks that run concurrently:

- Identify industrial application demands on real-time communication services, including survey of real-time communication with focus on LAN-technology and switched communication. Case studies of industrial applications are also planned.
- Develop and analyze how methods to support traffic with industrial real-time demands can be implemented in switches and/or network interfaces to get as much functionality and performance as possible at the same time as the use of standards like TCP/IP and Ethernet is preserved.

4. Expected results and impact

As stated above we have specific ideas and we attempt to focus on real-time support for the wide-spread Ethernet standard. However, it is important to stress that we have the ambition of finding methods with general applicability in the area of real-time networking.

The main expected results are:

- Methods to support traffic with industrial real-time demands over non-real-time LAN-technology, primarily over switched Ethernet, without loss of generality to use common protocol suits like TCP/IP.
- General outlines of how to support traffic with industrial real-time demands over switched system area networks.
- Performance analysis of proposed methods.
- Implementation experiments to demonstrate the practical feasibility of the developed methods and to make performance measurements including implementation aspects.
- Case studies where the industrial applicability of the proposed methods is confirmed.

Fibre-Ribbon Pipeline Ring Network with Distributed Global Deadline Scheduling

Carl Bergenhem[1], Magnus Jonsson[1], and Jörgen Olsson[2]

1. {carl.bergenhem, magnus.jonsson}@ide.hh.se, Computers and Communications lab, Halmstad University, Halmstad, Sweden. Phone: +46-35-167100 Fax: +46-35-120348
2. jool@vtd.volvo.se, Volvo Technological Development Corporation, Göteborg, Sweden. Phone: +46-31-7724681

Abstract

This paper introduces a novel, fair medium access protocol for a pipelined optical ring network. The protocol provides global optimisation of deadline constraints on a packet basis. Requests for sending packets are sent by the nodes in the network to the current master node. The master uses deadline information in the requests to determine which request is most urgent. Arbitration is done in two steps, collection and distribution phases. The protocol is therefore called two-cycle medium access (TCMA). The network is best suited for LANs and SANs (system area networks) such as a high speed network in a cluster of computers or interconnection networks in embedded parallel computers. Services possible in this network include best effort messages, real-time virtual channels, functions used in parallel processing such as barrier-synchronisation, and functions for reliable transmission. These are possible without additional higher level protocol layers. A simulation analysis of the network with the novel protocol is presented. Further analysis shows worst case latency, minimum slot length, and fairness of the protocol.

Keywords: Real-time communications, protocol, global deadline optimisation, packet constraints, fibre-optic interconnection network, embedded systems, parallel processing

1 Introduction

The contribution put forward by this paper is a novel medium access protocol that uses the deadline information of individual packets, queued for sending in each node, to make decisions, in a master node, about who gets to send. The new protocol may be used with a previously presented network topology; the control channel based fibre ribbon pipeline ring (CC-FPR) network [1]. Simulations of the protocol prove its validity.

The proposed medium access protocol provides the user with a service for sending best effort messages for which the timing constraints are globally optimised. Because of this property the protocol is suitable for real time communication. Further more, the global optimisation is a mechanism that is built into the network protocol. No further software in upper layers is required for this service. The scheme of globally optimising deadline constraints presents an advantage over networks that don't arbitrate medium access on this property. Some examples of this may be found in [1], [2], [3], [4], [5]. However, these networks may have upper layer protocol added to them to give them better characteristics for real-time traffic but this increases the complexity of the system. Also, it is hard to get fine deadline granularity using upper layer protocols.

Real-time services in the form of best effort messages, as mentioned above and real-time virtual channels (RTVC) are supported for single destination, multicast and broadcast transmission by the network. A service for slot reservation, used for hard real-time traffic such as RTVCs is also provided [1]. The network also provides services for parallel and distributed computer systems such as short messages, barrier synchronisation and global reduction. Reliable transmission service (flow control and packet acknowledgement) is provided as an intrinsic part of the network [6], [7].

The network with the proposed protocol is best suited for LANs and SANs (system area networks) where the number of nodes and network length is relatively small. This is important since the propagation delay adversely affects the medium access protocol. An example of a suitable application is in an embedded system, e.g., for use as an interconnection network in a radar signal processing system, or as a network for use in cluster parallel computing. A problem with the original CC-FPR protocol [1] is that a node does not consider the time constraints of packets that are queued in downstream

nodes. The novel network presented here does not suffer from this problem.

Novel optical components result in the possibility of new network solutions for the increasing bit rate demands of parallel and distributed systems. Motorola OPTOBUS™ bi-directional links with ten fibres per direction are used but the links are arranged in an unidirectional ring architecture where only $\lceil N / 2 \rceil$ bi-directional links are needed to close a ring of N nodes (assuming that N is an even number). Fibre-ribbon links offering an aggregated bit rate of several Gbits/s have reached the market [8]. The increasingly good price/performance ratio for fibre-ribbon links indicates a great success potential for the proposed type of networks.

The physical ring network is divided into three rings or channels (see Figure 1). For each fibre ribbon link, eight fibres carry data, one fibre is used to clock the data, byte by byte, and one is used for the control channel. Access is divided into slots like in an ordinary TDMA (Time Division Multiple Access) network. The control channel fibre is dedicated for bit-serial transmission of control-packets, which are used for the arbitration of data transmission in each slot. The clock signal on the dedicated clock fibre, which is used to clock data, also clocks each bit in the control-packets. Separating clock- and control-fibres simplifies the transceiver hardware implementation [9]. The control-channel is also used for the implementation of low-level support for barrier-synchronisation, global reduction, and reliable transmission [6].

The ring can dynamically (for each slot) be partitioned into segments to obtain a pipeline optical ring network [2] where several transmissions can be performed simultaneously through spatial bandwidth reuse, thus achieving an aggregated throughput higher than the single-link bit rate (see Figure 2 for an example). Even simultaneous multicast transmissions are possible providing multicast segments do not overlap.

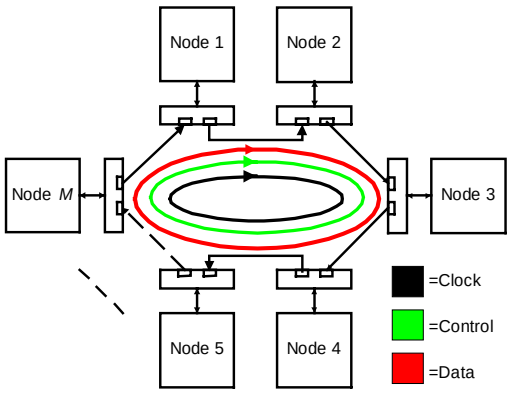


Figure 1: A Control Channel based Fiber Ribbon Pipeline Ring network.

The rest of the paper is organised as follows: Section two briefly describes the original protocol since many ideas are similar to the presented. Section three presents the novel medium access protocol. Section four gives a short description of RTVCs and how they are implemented. Section five presents an analysis of worst case latency and minimum slot time, and describes the simulations carried out. These results are presented and discussed. Conclusions are presented in section six.

2 Original protocol

The novel medium access protocol presented and analysed in the following sections, use many ideas from the CC-FPR protocol presented in [1]. Therefore some details of the CC-FPR protocol are presented here.

The original CC-FPR network protocol is insensitive to propagation delay in the sense that no feedback, from nodes back to the master, is needed during arbitration of the network. Arbitration is decentralised by having nodes take equal, round-robin, turns to be master. The protocol is based on the use of a control-packet that, for each slot, travels almost one lap (over $N–1$ links) round the control-channel ring. In the time domain the control-packet always travels around the ring in the time-slot preceding the time-slot for which it controls the arbitration. The control-packet will hence always pass each node one time-slot before the data-packet it is related to passes.

At the beginning of each slot the master initiates a control packet that contain its own needs for packet transmission. Each node succeeding the master checks the control-packet when it passes the node to see: (*i*) if it will receive a data-packet in the next slot, and (*ii*) if a data-packet will pass the node in the next slot. If no data-packet will pass the node, i.e., the rest of the ring back to the master is free, the node will have the possibility to

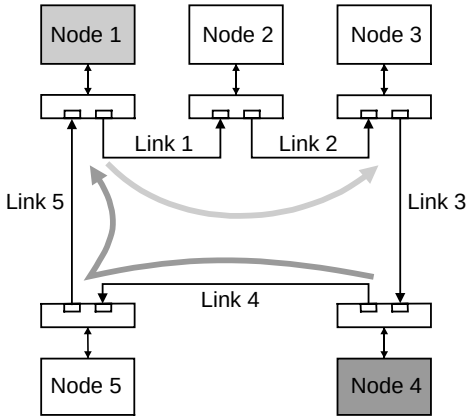


Figure 2: Example where Node 1 sends a single-destination packet to Node 3, and Node 4 sends a multicast packet to Node 5 and Node 1.

transmit a data-packet in the next slot in this segment of the ring. The node signals this by changing the control packet to reflect how it will send. Because all the nodes succeeding the master repeat the procedure of checking the control-packet for the possibility to send, multiple transmissions in different segments of the ring might be possible in the same slot. Observe that the control packet does not return to the master and that a data packet cannot be transmitted across the master node since the clock is interrupted there.

A problem with the original CC-FPR protocol is that a node does not consider the time constraints of packets that are queued in downstream nodes. See Figure 2 for an example described below. Node one decides that it will send and books links one and two, regardless of what Node two may have to send. This means that packets with very tight deadlines may miss their deadlines. For a further presentation of the original protocol, refer to [1].

3 Two-cycle medium access protocol

The greatest difference in function between the new and the old protocol of accessing the network is that arbitration is done in two phases instead of one. The two phases to medium access are collection phase and distribution phase (see Figure 3). Therefore it is referred to as the two-cycle medium access protocol, (TCMA protocol). As can be seen, the protocol is time division multiplexed to share access between nodes. The basic time unit is called a slot and the minimum size of the slot is analysed in section five.

As in the original medium access protocol, the role of network master is cycled equally, round robin, around the ring. Thus all nodes are identical. The role as master is passed on to the next down stream node at the end of the slot. Every node detects when the clock signal is interrupted at the end of the slot and nodes have a counter to determine who is next master.

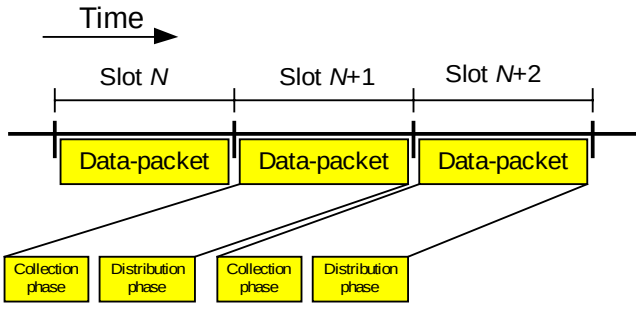


Figure 3: The two phases, collection and distribution, of the TCMA protocol. Notice that the network arbitration information, for data in slot *N+1,* is sent in the previous slot, slot *N.* Observe that the lengths of the phases, and placement in time, in the diagram are not to scale.

There are two types of TCMA control packets, which are used in each of the two phases (see Figure 4). A complete collection phase packet will contain a start bit and total of *N*-1 requests that are added one by one by each node. The master receives it's own request internally. Each request consists of three fields. The "prio"-field contains the priority level of the request. It is further described below. Nodes use the link reservation and destination fields to indicate destination node(s) and which links must be traversed to reach the destination node. Since a node may write several destination nodes into the destination field, transmissions may be multicast or broadcast. In the distribution phase packet the "result of requests"-field contains the outcome of each nodes request. This is the only field, in this phase, which contains network arbitration information. The others are used for services such as, e.g., reliable transmission ("ACK/NACK"- and "flow control"-fields) and, e.g., global reduction and short messages (the "Extra information"-field).

The time until deadline (referred to as laxity) of a packet is mapped, with a certain function, to be expressed within the four-bit limitation of the current version of TCMAs priority field. A lower priority implies a shorter laxity and thus a more urgent message. The priority field is a central mechanism of the TCMA protocol. The result of the mapping is written to the priority field (see Figure 4). A wider field of bits would provide higher resolution of priority and would probably have an advantageous affect on performance. Further

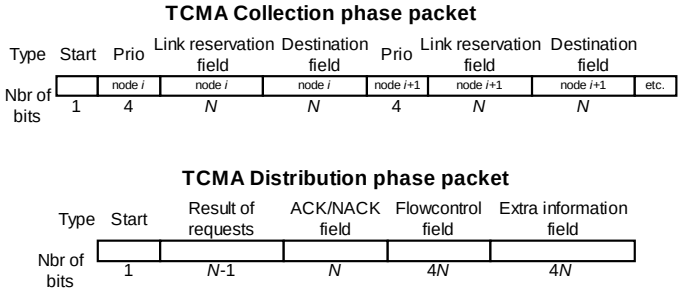


Figure. 4: Contents of the TCM control packets. Notice the possibility in the distribution phase packet to send information other than for medium access arbitration.

evaluation is out of the scope of this paper. Two mappings between deadline and priority, logarithmic and linear (see Figure 5), have been simulated. Results show a negligible difference in performance of throughput, packet-loss, and latency. Further evaluation of how the performance is affected by different mappings is therefore put beyond the scope of this paper. For the simulations presented in section five, logarithmic mapping is used. This mapping gives higher resolution of laxity, the closer to its deadline a packet gets.

All nodes including the master, have information about which slots have been reserved, e.g., for RTVCs between pairs of nodes. This is because a request to establish an RTVC is broadcast to all nodes. Therefore a node will only send a request that may be possible to fulfil regarding RTVCs in the own or other nodes that would use links in the path of the packet that the node would want to send. Observe that a node will not request a transmission that would be "across" the master since the clock signal is interrupted there. This implies that a request will only be rejected if requests from other nodes are more urgent. The node selects its most urgent message as the request. In the case that there are several messages that are equally urgent, the message that is destined furthest and possible to transmit in the next slot is selected. Slots belonging to RTVCs do not need to be "requested" since they are already reserved (see section four).

In the collection phase the node that is currently master generates an empty packet and transmits it on the control channel. Each node adds its request, in turn, for sending in the next slot. A request packet that will be added, by a node, to the packet from the master can be generated and ready to be sent as soon as the node knows the status of its queues so that only a very small delay is incurred. The additional delay is caused when the collection phase packet passes a node. It is assumed to be one bit time. The same delay also applies to the distribution phase packet. The request contains the following information: priority, link-reservation and destination field (see Figure 4). If the node has nothing to send, it signals this to the master by using a reserved priority level (15 in the proposed protocol) and zeros in the other fields.

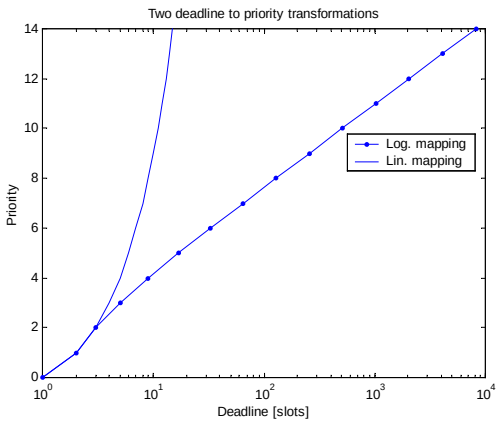


Figure 5: Two different deadline-to-priority mappings that were tested. For the linear transformation, deadlines longer than 14 slots are all mapped to priority level 14.

When the completed collection phase packet arrives back at the master, the requests (including one request from the master) are processed. There can only be *N* requests in the master, as each node gets to send one request per slot. The list of requests is sorted primarily by priority and secondly by the distance to the destination, i.e., furthest within segment (FWS)[2]. The master traverses the list, starting with the request with lowest priority (closest to deadline) and then tries to fulfil as many of the *N* requests as possible. In case of priority ties, the request with the largest distance to its destination is chosen. If there still is a tie then the master's request has priority over other request. Because of spatial slot reuse, several requests may be granted permission to transmit in different segments, during the same slot, providing that segments do not overlap. This is also called pipelining of packets (see Figure 2).

When the master has scheduled the requests it distributes the result to all nodes in the distribution phase. In this phase the master node, and only the master node, has possibility to make use of the "extra" fields in the distribution phase packet such as sending acknowledges for packets sent during previous slots. For further explanation of this, see [6]. When all nodes have received the results of the request, each node is ready for the beginning of the next slot where data may be transmitted. A request was granted if the nodes "request result field"-bit in the distribution phase packet contains a "1". As in the collection phase, the distribution phase packet is delayed when passing each node. Again, this delay is assumed to be one bit time, although not as important as for collection phase since the master does not get feedback from nodes in the distribution phase. The master receives the result of the requests internally.

The advantage of this protocol is that the deadline requirements of all packets are taken into account and considered at a global level. Since the packets deadline information is collected into a "global queue" in the master, packets from each node can be sent in an earliest deadline first (EDF) fashion and the timing constraints of the packets can be seen as globally optimised.

4 Real-time virtual channels

Logical connections with guaranteed bit rate and bounded latency can be realised in the network by using slot reserving. Such connections are referred to as RTVCs. Either the whole ring is reserved for a specific node in a slot, or one or more segments of the ring are dedicated to some specific node(s). Slots are organized into cycles with a set number of slots. Nodes keep track of the current slot index in the cycles. A slot that has been reserved for an RTVC, guarantees that transmission is possible every cycle thus guaranteed bit rate. Several slots may be reserved for an RTVC in order to increase the guaranteed bit rate. Initially, each node has *J* non-reserved slots where it is master, giving a cycle length of $N \cdot J$ slots. So as to always have bandwidth for best effort traffic controlled by the TCMA protocol, described in section three, only *J*-1 slots are reserveable.

When a node wants to reserve a slot for an RTVC, it searches for slots where the required links are free, so allocation of a new segment can be done. First, the node's own slots are searched. If not enough slots could be allocated for the reservation, the search is continued in other nodes. In this case, the node broadcasts a packet containing a request to all other nodes to allocate the desired segment in their slots. The packet contains information about the links required and the amount of slots needed. Each node then checks if any of its own slots have the required free links. All nodes sends a packet back to the requesting node to notify which slots, if any, that have been allocated. When the requesting node has received the answers, it decides if it is satisfied with the number of allocated slots. If not, it sends a release packet. Otherwise, it can start using the reserved slots immediately. If so, the node notifies other nodes by broadcasts the details of the RTVC. It should also send a release packet if more slots than needed were allocated. A node wishing to set up an RTVC can thus "borrow" slots from other nodes. A further, more detailed, description of this is found in [6].

5 Implementation aspects

T_{tcma} is the time required to complete network arbitration according the TCMA protocol. This also sets the minimum possible slot length for the network. T_{tcma}, scales for increasing network length and number of nodes as follows:

$$T_{tcma} = T_{collection} + T_p + T_{selection} + T_{distribution} \quad (1)$$

and is explained below. The master requires a non-infinite time for processing the collected requests to select which requests may be sent. Part of this processing is sorting the incoming requests. This can be done as they arrive by checking them bit by bit and thus the sorting time is incorporated in the time for the collection phase, $T_{collection}$. When requests have been sorted, the list of requests has to be traversed to select which may be sent. The time for this is denoted as $T_{selection}$ and is assumed to be $N \cdot 30$ ns. Propagation delay, T_p, is part of arbitration since the master depends on feedback, i.e., the requests from the other nodes in the collection phase. *L* is the total length of the ring. *P* is the propagation delay through the optical fibre and is assumed to be 5 ns/m. The delay through each node (approximately 1 bit time per node) is neglected in the propagation delay, thus the total propagation delay around the ring is:

$$T_p = L \cdot P \quad (2)$$

From Figure 4 one can see the size of the fields in the control packets which lead to:

$$T_{collection} = \frac{1+(2N+4)(N-1)}{C} \quad (3)$$

and

$$T_{distribution} = \frac{10N}{C} \quad (4)$$

where *C* is the bit rate of the links which is 800Mb/s for the Motorola Optobus and *N* is the number of nodes. The minimum slot length is plotted in Figure 6. As we can see from the figure, the minimum slot length increases with increased network length and number of nodes. This is just as can be expected from a network that requires feedback in its medium access protocol.

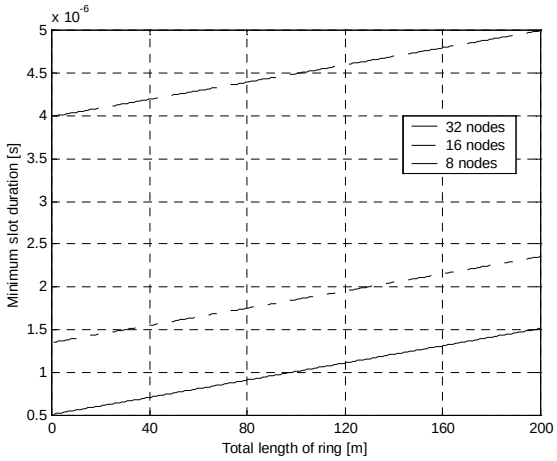


Figure 6: The diagram shows the relation between total network length and minimum slot length, for three different numbers of nodes.

5.1 Worst-case analysis

From the point in time when a packet is generated the delay until the packet may be sent consists of two components. The first component is the delay because other packets in the network are more urgent or have equal priority but are destined further and thus have priority. The second component is when no other packets that are equally or more urgent on other nodes, i.e., the packet is next in line to be sent. Only the second part of the delay is analysed below. This component of the delay is referred to as access to the network latency.

Two cases for worst case access to the network latency are presented: when the packet is destined M=1 hops and when the packet is destined M=N-1 hops. These two cases are explained in Figure 7. As we can see in the figure both latencies are minimised by sending the control phase (CP) packets as close as possible to the next slot. The vertical lines denote the end and beginning of slots. Since the clock is interrupted at the end of each slot, it is impractical to have the CP packets be sent at the end of one slot and continue into the next slot. As can be seen in the figure, slot length also affects the latency.

When a node is master, it is always granted transmission as long as no other node has a packet with a lower priority level (closer to deadline). It is assumed that a node is master every N slots and that no other node has more urgent packets. The equations for the latency is presented below:

$$T_{latency} = M \cdot T_{slot} + T_{cpp\_skew} \quad (5)$$

where

$$T_{cpp\_skew} = T_{tcma} + T_n \quad (6)$$

and

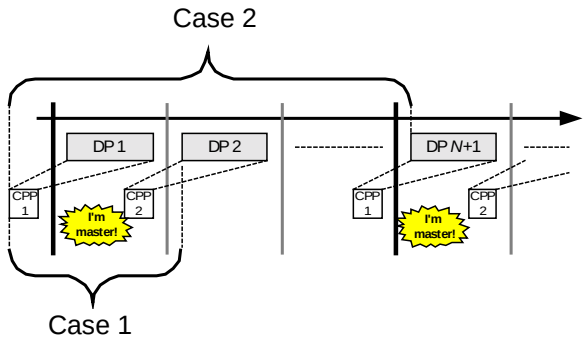


Figure 7: Worst case latency for the two cases. 1: *M*=1 hops and 2: *M*=*N*-1 hops. Control phase packets are denoted CPP and imply both control and distribution phases. Data packets are denoted DP. Note that the diagram is schematic and not to scale.

$$T_n = \frac{N-1}{B} \quad (7)$$

T_{slot} is the slot length, and T_n is the total extra delay incurred to the control packet, compared to the data packet, when passing through N-1 nodes. The extra delay is assumed to be one bit-time in each node. T_{cpp_skew} is the time from the start of the control phase until the start of the data packet. This can be seen in Figure 7. Remember that the control phase for the current slot occurred in the previous slot.

As can be seen in Figure 7 the latency depends on how far the node wishes to transmit, denoted M. The importance of keeping note of M is that a node may not transmit to another node that is "past" the master. This is because the clock signal is interrupted at the master. Transmissions destined for downstream neighbours (M=1) are always possible whereas the chance to be able to send decreases with increased number of hops. This explains the M-term in Equation 5.

Equation 5 is plotted in Figure 8. T_{slot} is chosen to be 5 µs, see the previous section on minimum slot length. As can be seen in the figure, there is a large difference in latency depending on the destination of the transmission. A user of the network can therefore gain in performance by carefully optimising algorithms to take advantage of this property, which is, sending to the next neighbour. The reason that the plots in the figure are almost horizontal, especially the plot for M=N-1 is that the slot delay ($M \cdot T_{slot}$) is a very small part compared to the rest of the arbitration delay (T_{cpp_skew}).

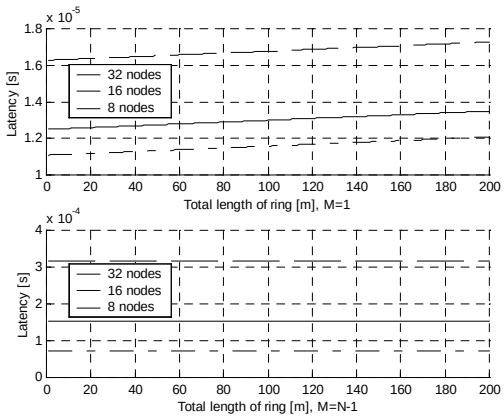


Figure 8: The worst case latency for two cases: *M*=1 hops and *M*=*N*-1 hops. The plots are made using a slot length (T_{slot})of 5 µs, which seems to be a realistic choice, see also Figure 6.

5.2 Simulation analysis

In addition to worst case performance analysis reported on in the previous section, an average case performance analysis for best effort traffic is also presented. The analysis is done by discrete time computer simulation. Networks of 8, 16, 32 and 64 nodes were simulated.

Packet in the system have soft real time constraints, thus are not given any guarantee, at generation, that it will be sent in a timely fashion. The packet is given a relative deadline at generation which is decremented each time slot. The packet is then queued until it is either sent successfully or, when deadline reaches zero, is deemed lost and removed from the queue. We also call this type of communication, best effort messages [10]. The user sending packets with a best effort service should not require any guarantees.

Some further assumptions for the simulations:

- Messages are one packet long and take one time slot to send. The term packet and message is therefore used synonymously.
- Uniform traffic is assumed, i.e., all nodes have equal probability of message generation and uniformly distributed destination addresses. This implies that, on average, it is theoretically possible to transmit two packets each slot, since the packet on average is destined "half way" around the ring, which is $N / 2$ hops. However, this disregards protocol effects which lowers the average utilisation which we will see later. An example of protocol effect is that a node may not send past the master since the clock is interrupted there. The pipelined ring topology of the network suggests that it be very effectively utilised when traffic is mostly destined one hop to the next neighbour such as in some types of radar signal processing [11], [12]. If this special mode of traffic were simulated, which it is not, the effect would simply be higher throughput because of aggregation, i.e., several packets would be sent during one slot.
- Messages were generated according to a poisson process and all messages were of single destination type.
- The deadline of all best effort packets is set at generation to 800 slot times ahead, which would equate to a deadline of 4ms with a slot time of 5 µs.
- Physical effects such as the propagation delay through optical fibre and the time required to detect the end of a slot was assumed to be less than one slot time and is therefore neglected.
- Message latency is defined as the time elapsed from the moment a message is generated until the entire message is received in the receiver.
- Infinite size of message queues is assumed.
- The simulator is run for a total of 100 000 slot times and starts to log statistics at 20 000 slot times.
- The "total packet generation intensity" is the generation intensity for the network regarded as a whole, not of the individual nodes.

Figure 9 shows the best effort packet latency for varying levels of network throughput. At a useful level of packet intensity the network has an average throughput of approximately 1.6 packets per slot. This is 60% better than the theoretical limit of one packet per slot for networks without spatial reuse. Similar results for varying number of nodes are obtained. Figure 9 shows up to which level of throughput the network may be useful. For a clearer view of throughput see also Figure 10.

Figure 10 shows the packet throughput against packet generation intensity. Throughput has a linear relation to packet intensity up to the point of saturation, which can be seen in the figure as the point on the plot where throughput starts to decrease with increasing packet intensity. As packet intensity passes the point of saturation it becomes increasingly difficult for the packet transmission scheduler to effectively utilise the bit rate of the network. This is because it is always increasingly difficult to schedule packets the further their destination

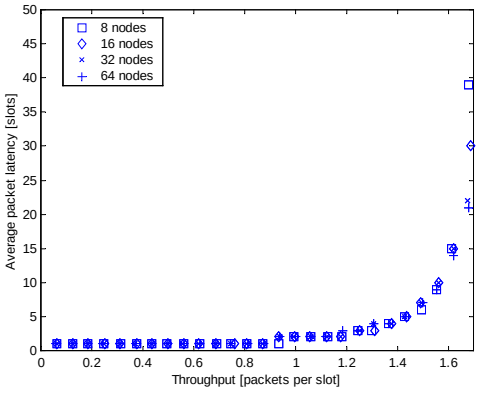


Figure 9. Best effort packet latency vs. throughput for a varying number of nodes.

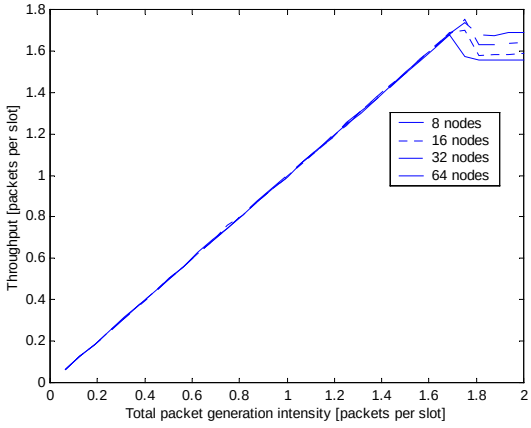


Figure 10: Total throughput of best effort packets vs. packet generation intensity.

[2]. When the network is saturated each node will tend to always contain far destined packets with short deadlines. These cannot always be scheduled together with the shorter destined packets because of conflicting links. When two packets have the same priority level the packets with the further destination has priority. Thus slot utilisation will decrease which is expressed in the decline of the throughput after the "summit" in the figure. The simulation shows that the utilisation of slots does not perform as well as may be theoretically expected (two packets transmitted per slot, because of pipelining). This is attributed to the policy of using deadline to selecting packets for transmission. The policy is not as bandwidth conserving as, e.g., the FWS policy [2] which can achieve higher average throughput. This is the cost of having good support for real-time traffic.

Figure 11 shows packet throughput and average packet latency for each destination distance plotted against packet generation intensity. The simulation is for 16 nodes. Concluded from this simulation is that TCMA treats packets fairly regardless of the distance to the destination even when the network approaches and is saturated. Observe that there are 15 (for *N*-1 distances to destination) plots for latency but that these overlap and appear as one.

6 Conclusions

This paper presents an optical ring network medium access protocol that globally optimises packet timing constraints. Simulation results of the medium access protocol have been presented. The protocol is shown to be fair even when the network is saturated. The network with the presented protocol is suitable for application such as in embedded systems, e.g., for use as interconnection network in a radar signal processing system, or as a high performance network for a LAN. Also worth mentioning is that the network can be built today using fibre-optic off-the-shelf components

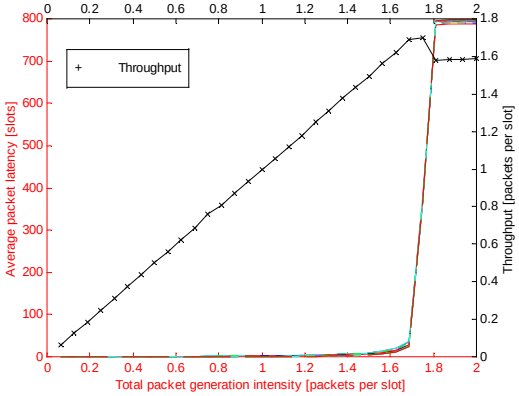


Figure 11. Packet throughput vs. packet generation intensity for 16 nodes. The other curves represent latency for the (15) different distances between source and destination.

Acknowledgement

This work is part of M-NET, a project financed by ARTES: A Real-Time network of graduate Education in Sweden.

References

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

[2] P. C. Wong and T.-S. P. Yum, "Design and analysis of a pipeline ring protocol," *IEEE Transactions on communications*, vol. 42, no. 2/3/4, pp. 1153-1161, Feb./Mar./Apr. 1994.

[3] M. Conti, E. Gregori, L. Lenzini, "DQDB under heavy load: performance evaluation and fairness analysis," *INFOCOM '90, Proceedings of the 9th Annual Joint Conference of the IEEE Computer and Communication Societies*, vol.1, pp. 313 –320, 1990.

[4] B. Raghavan, Y.-G. Kim, T.-Y. Chuang, B. Madhavan, A.F.J. Levi, "A gigabyte-per-second parallel fiber optic network interface for multimedia applications," *IEEE Network*, Volume: 13 Issue: 1 , Jan.-Feb. 1999 pp. 20 –28.

[5] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, Wen-King Su, "Myrinet: a gigabit-per-second local area network," *IEEE Micro*, Volume: 15 Issue: 1, Feb. 1995 pp. 29 –36.

[6] M. Jonsson, C. Bergenhem, and J. Olsson, "Fibre-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.

[7] 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.

[8] D. Bursky, "Parallel optical links move data at 3 Gbits/s," *Electronic Design*, vol. 42, no. 24, pp. 79-82, Nov. 21, 1994.

[9] C. Bergenhem, "A demonstrator for a CC-FPR network," *Techical report 0010, School of Information Science, Computer and Electrical Engineering (IDE), Halmstad University, Sweden*, Feb. 2000. In Swedish.

[10] K. Arvind, K. Ramamritham, and J. A. Stankovic, "A local area network architecture for communication in distributed real-time systems," *Journal of Real-Time Systems*, vol. 3, no. 2, pp. 115-147, May 1991.

[11] M. Jonsson, B. Svensson, M. Taveniku, and A. Åhlander, "Fiber-ribbon pipeline ring network for high-performance distributed computing systems," *Proc. International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'97)*, Taipei, Taiwan, Dec. 18-20, 1997, pp. 138-143.

[12] 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/SPDP98)*, Orlando, FL, USA, Mar. 30 – Apr. 3, 1998, pp. 226-232.

Modeling and Analysis of Message-Queues in Multi-Tasking Systems

Thomas Nolte, Hans Hansson

Mälardalens Högskola/IDT
thomas.nolte@mdh.se, hans.hansson@mdh.se

This paper presents work in progress on an analysis method for message queues in a real-time multi-tasking system. This analysis will later be compared with simulation results, in which variations/distributions of execution times, task periods etc will be considered. The intention is to evaluate the level of pessimism in the analytical worst-case analysis, compared to the execution scenarios that actually occur in the real system. We will use these results as a basis for future work in the ARTES project RATAD, which aims at developing schedulability analysis with a wider applicability, by integrating reliability into real-time scheduling theory. What we want is a method for integrating reliability into classical real-time scheduling, so that we can guarantee, up to some level, that our system is working properly. This is to be compared with the classical 0/1 results of the schedulability theory.

1. Introduction

There is hardly any published work on schedulability analysis of message queues. There is however related work on scheduling messages in communication networks, including [2]. These models are however different from those needed for modeling standard multi-tasking operating system message queues (e.g. those proposed for POSIX [3]), in that the networking queues are separately scheduled resources, whereas the handling of the OS queues is performed by a CPU shared by many types of tasks, as well as the OS kernel.
Outline: In Section 2 we present our model and in Section 3 the method used for analysis is described. Finally our conclusion and future work is described in Section 4.

2. Problem formulation and model

We are assuming a producer-consumer scenario with a pre-emptive fixed priority task model. The inter-process communication is performed using message queues. Initially we assume the queues to be FIFO, but later we will extend the model to also cover priority queues.
The consumer has the following behavior

```
while(1){
    …
```

```
    receive(…); //blocking
    …
}
```

Hence the consumer is blocked if nothing is to be received from the message queue. When the consumer is allowed to execute, it will (in the worst case) execute for C_c time units. This is repeated for every message that it consumes.

The producers are assumed to (in the worst case) behave as periodic tasks that at the end of their execution queue a message.

This paper is restricted to a scenario where all communication and execution is performed locally on a single node. We will for such a system consider a set of message queues, each handled by a single consumer, to which a set of associated producers queue messages as can be seen in Fig 1.

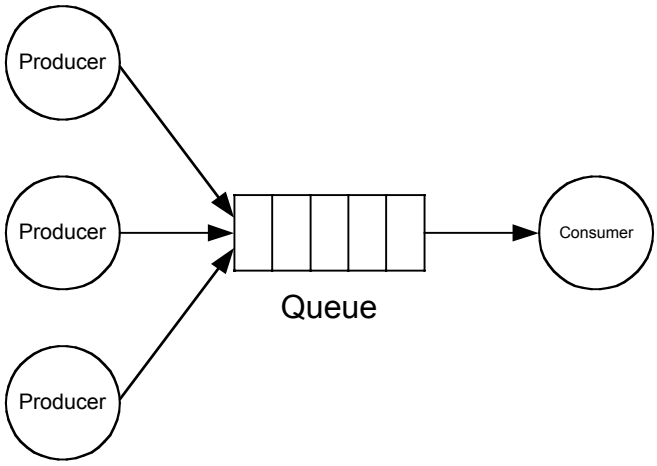


Fig. 1. System containing a set of producers and consumers that are communicating via a message queue.

The problem is here to calculate the worst-case end-to-end delay in a producer – message – consumer transaction. We will additionally calculate an upper bound of the message queue size.

The following terminology is used in this paper:

- T – The period of a task
- C – The worst case execution time of a task
- J – The jitter of a task. $J = r_{max} - r_{min}$, where r is the release time relative to the start of the period associated with each task. If $r_{min} = 0$ then $J = r_{max}$, which is what we will assume in the following.
- P – The priority of a task
- r – The release time
- $\#_p$ – A term associated with the producer
- $\#_c$ – A term associated with the consumer
- $queue_{size}$ – The maximum buffer need for the queue
- R – The worst case end-to-end response time associated with a producer – queue – consumer transaction

- $R_c^{\{0\cdots n\}}$ – The response time to handle the n first messages in a busy period (a busy period being a time period during which the queue is continuously non-empty).

3. Analysis of message queues

We are about to analyse message queues for inter-task communication on a single node. We start this by a simple scenario, which we then extend to be more general and more complex. We have a producer-consumer scenario, where initially the consumer is blocked, waiting for a message to arrive in the message queue. Following standard fixed-priority schedulability analysis practice [4] we assume the producer to be released at most every T_p time units.

We will now derive some simple schedulability criteria and more detailed analysis for the above type of systems. The formulas presented here may not terminate unless load is less than 100%.

We begin with a single producer per queue and then extend to multiple producers per queue.

3.1. Single producer

First, we assume that we have a producer with a period, T, a certain amount of release jitter, J, execution time, C, and a priority, P. Further on we have a consumer with execution time and a priority as can be seen in Fig 2.

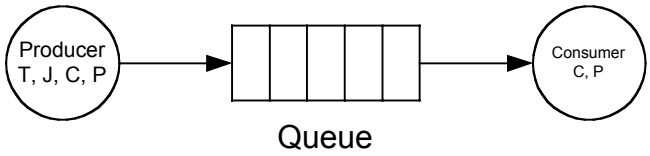


Fig. 2. The properties of the producer and the consumer task used in our model.

What we do now is to find out the worst end-to-end response time for this scenario. In order to find this response time we have to find the maximum response time of all messages produced within a Busy Period, i.e., the time it takes to completely empty the queue in a worst case scenario.

If we use R_n to denote the response time for the consumer to consume the n:th message, then the worst case end-to-end response time, R, is

$$R = \max_{n \in MBP}(R_n) \qquad (1)$$

where MBP is the set of messages sent by all producers of a queue within the Busy Period.

What we need to do is to calculate the end-to-end response times for all the messages sent within the busy period. In order for the busy period to terminate, we require a load that is less than 100%. We use an approach similar to the iterative method used for analysis of task response times when deadline exceeds period [5] to iteratively

search for the maximum response time of the messages queued during the busy period. This amounts to initially assume that a single message is queued, and then calculate the response time for this message. If a second message has been queued during this response time we can calculate the response time for this message by analyzing the (in the response time sense) equivalent scenario when both messages are queued initially (i.e. when the consumer needs to be executed twice). If a third message is queued during the response time of the two messages, then we repeat the procedure for three messages. This continues for four, five, etc. messages until we reach a point when no new message is queued during the response time. That such a situation exists follows by the assumption that we have a utilization that is less than 1.

In the worst case scenario we will number consecutive messages in increasing order, starting with message 0.

We define the release time for a message as r. Then we assume that the message is released as soon as the producer is activated. In this way we cover all cases of messages being sent. The release time of message n is given by

$$r = \left(n * T_p\right) - J_p \tag{2}$$

Then we calculate the response time for the consumer using traditional exact analysis methods [1].

$$\begin{aligned} C &= \left(n+1\right) * C_c \\ w_c &= C + \sum_{j \in hp(tasks)} \left\lceil \frac{w_c + J_j}{T_j} \right\rceil C_j \\ R_c^{\{0 \cdots n\}} &= w_c \end{aligned} \tag{3}$$

When we have the response time for the handling of the n first messages, $R_c^{\{0 \cdots n\}}$, we can calculate the response time associated with message n simply by subtracting the release-time of the current message from the total response time.

$$R_n = R_c^{\{0 \cdots n\}} - r \tag{4}$$

We have now gathered the response times for the n first messages and we must investigate whether or not we have emptied the queue, since this is our termination criterion for the busy period. To do this we investigate whether or not the following condition is met, i.e., if the queue is not empty or not. If the following is true, (2)-(4) is repeated for the next message.

$$\left(n+1\right) < \left\lceil \frac{R_c^{\{0 \cdots n\}} + J_p}{T_p} \right\rceil \tag{5}$$

When we have emptied the queue we use (1) to find out the maximum response time present in the Busy Period. When we have the maximum response time we can use it to calculate the buffer need for the queue. The maximum queue size will be

$$queue_{size} = \left\lceil \frac{R + J_p}{T_p} \right\rceil \quad (6)$$

Intuitively, this captures that the maximum queue size occurs when the worst-case end-to-end response time for a message is present.

3.2. Multiple producers

We have now looked at scenarios where we have a single producer feeding the queue with messages. We now extend this to handle multiple producers to a single queue. We introduce a set M containing all producers associated with each queue, that is all producers belonging to a certain queue. The procedure behaves the same as in the single producer scenario. The only difference is that the queue is receiving messages from multiple producers. If we first try to extract the worst end-to-end response time, we do as in the single producer scenario. What we need to handle is the scenarios where preemption involving several producers occurs, i.e., producers that gets preempted by other producers. If we want to calculate the worst end-to-end response time regardless of producer, all producers that are preempting us will in the worst case always manage to send their messages before us, as illustrated in Fig 3. Here we can see that if all higher priority producers will send their messages before us, we will be consumed as the last one, and therefore we will experience the longest end-to-end response time.

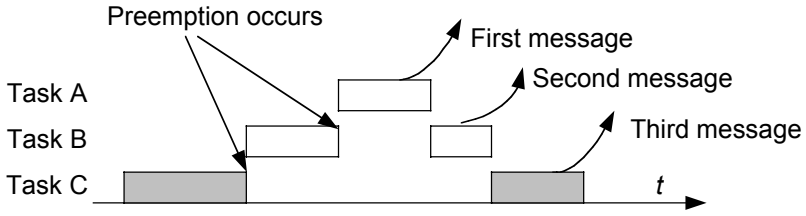


Fig. 3. The worst-case scenario regardless of which producer that is belongs to is always the one created for the producer with the lowest priority.

In this way, it is always the producer, which has the earliest release time of its current message that is interesting. What we need to do though, is to investigate whether or not some other producers will preempt this producer. If this is the case, then we always assume that the other producers, the ones that are preempting us, will produce their message before us. That is, we will always produce the worst scenario this way. If we do this, we can use similar methods to the ones used in the single producer scenario.

The situation becomes slightly more complicated when we are to extract the maximum end-to-end response time associated with a certain producer. This will force us to produce different execution scenarios depending on which producer we are interested in, since the actual time that each message is sent will influence the order of messages in the queue. The different scenarios, when having three producers associated with a queue, is the one described in Fig 3 (which is the worst case scenario for Task C) and the ones described in Fig 4 and Fig 5.

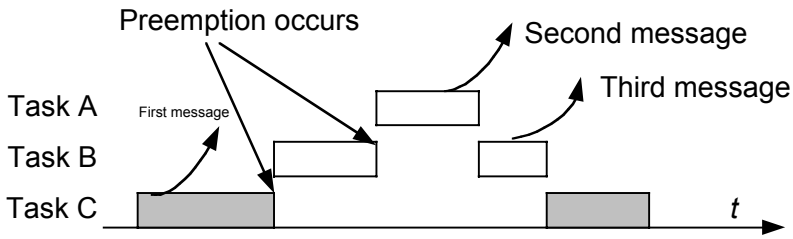


Fig. 4. The worst-case scenario for producer B is that all task with lower priority that we are preempting for the moment, actually has been able to already produce their messages. As usual, all preempting producers with higher priority than us will produce their messages before us.

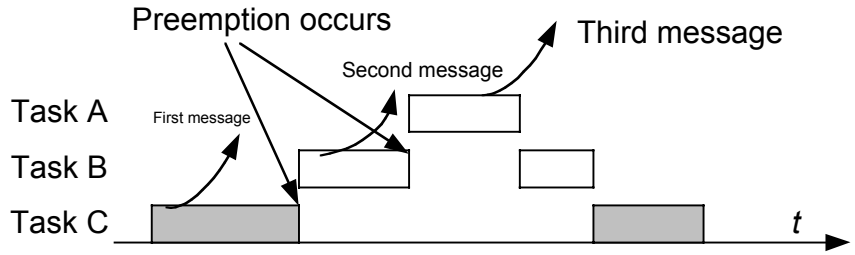


Fig. 5. The last scenario of our task set is when calculating the worst-case for the producer with the highest priority. Here all producers that we have preempted will produce their messages before us.

When we have taken these precautions, we need to modify our formulas used as the termination criteria, i.e., when the queue is empty, and our formula used for calculating the buffer need. What we will do is similar as for the single producer scenario, i.e., we extract all response-times until (5) is not met anymore, but in order to handle multiple producers we need to extend (5) to the following

$$(n+1) < \sum_{j \in M} \left\lceil \frac{R_c^{\{0 \cdots n\}} + J_j}{T_j} \right\rceil \tag{7}$$

When we are done, we can extract R using (1) in order to extract the maximum end-to-end response time so that we can calculate the required queue size. We need to modify (6) to handle multiple producers.

$$queue_{size} = \sum_{j \in M} \left\lceil \frac{R + J_j}{T_j} \right\rceil \tag{8}$$

4. Conclusion and future work

We have presented preliminary results from on-going work on including message queues in schedulability modeling and analysis. Our initial experiments indicate that it is not trivial to construct the worst-case scenario, especially when we are to collect worst-case end-to-end response time associated with each producer. When this is done though, we will extend the model to also cover messages with different priorities. This is essential, since many of the operating systems used in industry today, e.g. those that are POSIX compliant, support priority queues. When we have a good analysis method we will compare the results with simulation results. These results will then be integrated with some probabilistic model making it possible to integrate reliability theory with schedulability analysis.

References

[1] M. Joseph and P. Pandya. “Finding response times in a real-time system”, BCS Computer Journal, 29(5): 390-395, 1986

[2] Mikael Sjödin, “Predictable High-Speed Communications for Distributed Real-Time Systems”, Ph.D. Thesis, Uppsala university, Information Technology, Department of Computer Systems, May 2000

[3] IEEE Standard for Information Technology - Test Methods Specifications for Measuring Conformance to POSIX/sup R/ - Part 1: System Application Program Interface [API] - Amendment 1: Realtime Extension [C Language] IEEE Std 2003.1b-2000 , 2000

[4] Audsley, N.; Burns, A.; Richardson, M.; Tindell, K.; Wellings, A.J., “Applying new scheduling theory to static priority pre-emptive scheduling”, Software Engineering Journal , Volume: 8 Issue: 5 , Sept. 1993, Page(s): 284 –292

[5] Tindell K, Hansson H, “Real Time Systems by Fixed Priority Scheduling”, Department of Computer Systems, Uppsala University, 1996

Applications of Lock and Wait-free shared data structures to Real-Time Systems.

Håkan Sundell
Chalmers University of Technology
Department of Computing Science
phs@cs.chalmers.se

Abstract:

I will talk about my current research with real-time operating systems and also about shared data structures for concurrent programming. I will present the commercial OSE operating system and ideas behind the inter-communication of tasks. The drawbacks with the current approach will be highlighted and some ideas for improvements using lock and wait-free data structures will be presented.

I will briefly present the upcoming library tool-kit of shared data algorithms called NOBLE, and its continuations within embedded systems. NOBLE is mainly designed for fast performance parallel computer systems, but will also be adopted for common microprocessors used in embedded systems.

I will give an overview of the current research of improving current lock and wait-free algorithms, present the different approaches. I have recently looked on how to improve wait-free algorithms using the timing information available in real-time systems, for example to bound time-stamps.

27 Feb 2001

Non-blocking synchronization for soft real-time applications

Zhang Yi
yzhang@cs.chalmers.se

CS Department
Chalmers Technical University

Abstract

Non-blocking synchronization for real-time system has attract a lot of research efforts. However, Previous researches mainly focus on hard real-time system. Previous results show that wait-free mechanism is more favourable for hard real-time system than lock-free mechanism because wait-free
easy the scheduling and provides worst-case execution time for data sharing. In soft real-time application, tasks don't need to always guarantee to meet deadline. Although lock-free has unbounded worst-case execution time, it guarantee the consistency of shared data object during accessing the shared data object. So a task can quit from the critical section anytime it want and without worry about the consistency of the shared data objects. We will look at how to apply lock-free synchronization and what kinds of guarantee can be provided by lock-free synchronization.