Preface

This volume contains the papers presented at the 4th ARTES Graduate Student Conference, held at Uppsala University, from April 18-19, 2002.

As of today, more than 110 graduate students have joined the ARTES network by registering as Real-Time Graduate Students, and thereby getting access to the benefits provided by ARTES, including free-of-charge participation at the ARTES Summer School, as well as at other ARTES conferences and meetings, not to mention the mobility support provided by ARTES. The 110+ real-time graduate students clearly indicate that a substantial amount of real-time research is conducted in Sweden. Due to ARTES and other efforts, it is fair to say that Sweden is one of the world-leaders in real-time systems research. However, the future of ARTES is highly uncertain, as the current funding period will end in December 2002. Fortunately enough, the ARTES board have set aside funding to ensure continued operation of the network activities also in 2003. This includes the graduate student conference and the summer school. For prolongation after 2003 additional funding will be required, something which we are currently working on. So much for the uncertain future; returning to the present, we have an exiting conference ahead of us.

The main idea with the ARTES Graduate Student Conference is to provide a forum for technical presentations and discussions among the Swedish graduate students active in the real-time area. For newly recruited graduate students it will provide an opportunity to experience “a real conference situation” (maybe) for the first time. For everyone, the conference will be an excellent opportunity to, in a relatively short time, get an overview of the current state of the national research. This year we have even included a dissertation in the programme, as the real-time graduate students Jakob Engblom will defend his thesis Friday afternoon. In addition to the approximately 20 technical presentations by graduate students, we will enjoy invited talks by Profs. Peter Puschner from Vienna and Sang Son from Virginia. Further stimulation will be provided by Prof. Lars Asplund from MdH and Uppsala, who will tell the story of the Senseboard, as well as presenting Football playing robots developed by students at Uppsala University.

This year, ARTES deputy programme director Roland Grönroos has in addition to providing the usual invaluable assistance in organizing the event, also taken care of the local arrangements. Without his efforts it would not have been possible to organize this conference.

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

Enjoy it!

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

Participants at ARTES Graduate Student Conference 2002

Tobias	Amnell	tobias.amnell@docs.uu.se	Uppsala
Anita	Andler	anita.andler@usa.net	Skövde
Sten F.	Andler	sten.f.andler@ida.his.se	Skövde
Lars	Asplund	lars.asplund@mdh.se	Västerås
Marcus	Brohede	marcus@ida.his.se	Skövde
Anton	Cervin	anton@control.lth.se	Lund
Julien	D'Orso	juldor@docs.uu.se	Uppsala
Alexandre	David	adavid@DoCS.UU.SE	Uppsala
Radu	Dobrin	radu.dobrin@mdh.se	Västerås
Magnus	Ekman	mekman@ce.chalmers.se	Göteborg
Jad	El-khoury	jad@md.kth.se	Stockholm
Jakob	Engblom	jakob@docs.uu.se	Uppsala
Andreas	Ermedahl	ebbe@docs.uu.se	Uppsala
Elena	Fersman	elenaf@docs.uu.se	Uppsala
Roland	Grönroos	Roland.Gronroos@it.uu.se	Uppsala
Hans	Hansson	Hans.Hansson@it.uu.se	Uppsala
Jörgen	Hansson	jorha@ida.liu.se	Linköping
Dan	Henriksson	dan@control.lth.se	Lund
Damir	Isovic	damir.isovic@mdh.se	Västerås
Martin	Karlsson	martink@docs.uu.se	Uppsala
Rikard	Land	rikard.land@mdh.se	Västerås
Birgitta	Lindström	birgitta@ida.his.se	Skövde
Sorin	Manolache	sorma@ida.liu.se	Linköping
Leonid	Mokrushin	leom@docs.uu.se	Uppsala
Nguyen-Thai	Nguyen-Phan	thai@imit.kth.se	Stockholm
Robert	Nilsson	robni@ida.his.se	Skövde
Jonas	Norberg	jonas@yurisnight.net	Stockholm
Anders	Pettersson	anders.pettersson@mdh.se	Västerås
Paul	Pop	paupo@ida.liu.se	Linköping
Peter	Puschner	peter@vmars.tuwien.ac.at	Wien
Ola	Redell	ola@md.kth.se	Stockholm
Sven	Robertz	sven@cs.lth.se	Lund
Andreas	Sjögren	andreas.sjogren@mdh.se	Västerås
Sang Hyuk	Son	son@virginia.edu	Charlottesville
Daniel	Sundmark	daniel.sundmark@mdh.se	Västerås
Aleksandra	Tesanovic	alete@ida.liu.se	Linköping
Martin	Törngren	martin@damek.kth.se	Stockholm
Elisabeth	Uhlemann	elisabeth.uhlemann@ide.hh.se	Halmstad
Jonny	Vinter	vinter@ce.chalmers.se	Göteborg
Thiemo	Voigt	thiemo@sics.se	Uppsala
Mattias	Wecksten	Mattias.Wecksten@ide.hh.se	Halmstad
Zhang	Yi	yzhang@cs.chalmers.se	Göteborg

The number of participants is 41.

Programme at ARTES Graduate Student Conference 2002

Programme at ARTES Graduate Student Conference 2002

Friday April 19			**Page**
09.00	**Session: Scheduling and control**		
	Flexibility Driven Scheduling and Mapping for Distributed Real-Time Systems	Paul Pop	101
	Feedback Scheduling of Model Predictive Controllers	Dan Henriksson	111
	A Computational Model for Real-Time Control Tasks	Anton Cervin	119
10.00	Coffe at Cafe Alma		
10.15	QoS Management in Real-Time Data Services	Sang Hyuk Son	125
11.00	**Session: Testing and Wait-free Ques**		
	Efficient and Simple Implementations of the Wait-Free Queue Classes of the Real-Time Specification for Java	Zhang Yi	127
	Towards a Framework for Automated Testing of Transaction-Based Real-Time Systems	Robert Nilsson	137
	Exact Best-Case Response Time Analysis of Fixed Priority Scheduled Tasks	Ola Redell	143
12.00	Lunch at Cafe Alma		
13.00	Dissertation at Ångströmlab room 10134	Jakob Engblom	151
15.00	Coffe at Cafe Alma		
15.30	**Session: Resource Handling**		
	The Potential of Resource Budgets in Complex Real-Time System Design	Mattias Wecksten	153
	Resource-based Web Server Overload Protection	Thiemo Voigt	163
16.20	Closing		
18 dk	Optional dissertation party		

Presentations at
ARTES Graduate Student Conference 2002

Presentations at
ARTES Graduate Student Conference 2002

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

Graduate Student Conference 2002

Making Real-Time Tasks Temporally Predictable

Prof. Peter Puschner
Technische Universitaet Wien
Real-Time Systems Group
Austria
Web: http://www.vmars.tuwien.ac.at/~peter

Abstract
This talk introduces a new programming paradigm for writing hard real-time code: single-path programming. Every program following this paradigm has only a single possible execution path. The execution time of single-path code is constant and therefore fully predictable. We explain the software and hardware architectures used in single-path programming and show that the single-path paradigm provides a universal solution towards writing temporally predictable code -- every piece of code that has a boundable execution time can be transformed into single-path code.

Peter Puschner is a professor in computer science at Vienna University of Technology. His main research focus is on worst-case execution time (WCET) analysis for real-time programs. Puschner has been working on WCET analysis for more than ten years and has strongly influenced the state of the art in this field. He has published numerous papers on WCET analysis and was a guest editor for the special issue on WCET analysis of the Kluwer International Journal on Real-Time Systems. Further interests of Peter Puschner include real-time programming languages and hardware/software architectures for real-time systems.

Invitation, Schedule

Updated: 14-Mar-2002 14:45 by Roland Grönroos
e-mail: artes@docs.uu.se web: www.artes.uu.se
Location: http://www.artes.uu.se/events/gsconf02/pp.shtml

Swedish Foundation for Strategic Research

COMET: a COMponent-based Embedded real-Time database*

Aleksandra Tešanović
Jörgen Hansson
Linköping University
Department of Computer Science
Linköping, Sweden
{alete,jorha}@ida.liu.se

Dag Nyström
Christer Norström
Mälardalen University
Department of Computer Engineering
Västerås, Sweden
{dag.nystrom,christer.norstrom}@mdh.se

Abstract

In this paper we introduce the concept of the tailorable embedded real-time database called COMET. Designing a database to be tailorable, and at the same time satisfy real-time (temporal) requirements and minimal resource (space) requirements of an embedded and a real-time system, require exploiting the concept of components and aspects in embedded real-time database development.

1. Introduction

In the last years the deployment of embedded and real-time systems has increased dramatically [13]. The amount of data that needs to be managed by the real-time systems is increasing, thus requiring an efficient and structured data management. Hence, database functionality is needed to provide support for storage and manipulation of data in real-time and embedded systems, but it must also fulfill requirements both from embedded systems and from real-time systems[1].

Real-time systems are typically constructed out of concurrent programs, called tasks. The correctness of a real-time system depends both on the logical result of the computation, and the time when the results are produced, expressed explicitly as temporal constraints [12]. The most common type of temporal constraint that a real-time system must satisfy is the completion of task deadlines. Moreover, the data in the system is normally associated with temporal constraints, e.g., absolute and temporal validity intervals [10]. Thus, a real-time database must ensure that the data in the database is both logically and temporally consistent.

*This work is supported by ARTES (A network for Real-time and graduate education in Sweden).

[1]We distinguish between embedded and real-time systems, since there are some embedded systems that do not enforce real-time behavior, and there are real-time systems that are not embedded.

In contrast to an application-embedded database hidden inside an application, a device-embedded database is a database that resides in an embedded system. We focus on device-embedded databases, and refer to those as embedded databases. The main objectives for an embedded database are low memory usage, i.e., small memory footprint, portability to different operating system platforms, efficient resource management , e.g., minimization of the CPU usage, and ability to run for long periods of time without administration [9].

Two dimensions are of interest when designing a database for real-time and embedded systems: time and space. This can be further refined to include: (i) functionality vs size trade-offs, (ii) effects of the temporal requirements on the functionality, and (iii) the cost of the production.

Hence, to cope with these challenges we propose a real-time database platform based on the concept of components and aspects: COMET (COMponent-based Embedded real-Time database system). To reduce the cost of production, reuse is the key word, and reuse of components from a component library is the form of reuse that is gaining momentum, e.g., COM, CORBA and JavaBeans. Having aspects in addition to components implies incorporating basic ideas of aspect-oriented programming (AOP) [6] into the database development. Thus, in COMET we distinguish between *functional and aspectual decomposition* of the database system. Functional decomposition is a way of decomposing a database system into components. *Components* are functional units that contain the primary structure of the database system and carry the core functionality of the system. Aspectual decomposition is a way of separating cross-cutting concerns in the system, e.g., code that cannot be encapsulated within one functional unit but is tangled over the entire system. Hence, *aspects* are non-functional units that contain the secondary structure of the database and refer to components and other aspects.

The paper is organized as follows. Motivation for com-

ponentization of an embedded real-time database system is given in section 2. Section 3 presents major challenges for the development of a component-based embedded real-time database system. Related work is discussed in section 4. The paper finishes with summary containing main conclusions and directions to our future research.

2. Motivation

2.1. Embedded and Real-Time Databases

Existing commercial embedded database systems, e.g., Polyhedra, RDM, Velocis, Pervasive.SQL, Berkeley DB, and TimesTen, have different characteristics and are designed with specific applications in mind. They support different data models, e.g., relational vs object-relational model, and different operating system platforms. Moreover, they have different memory requirements and provide different types of interfaces for users to access data in the database. Application developers must carefully choose the embedded database their application requires, and find the balance between required and offered database functionality. Hence, finding the right embedded database is a time consuming, costly and difficult process, often with a lot of compromises. Additionally, the designer is faced with the problem of database evolution, i.e., the database must be able to evolve during the life-time of an embedded system, with respect to new functionality. However, traditional database systems are hard to modify or extend with new required functionality, mainly because of their monolithic structure and the fact that adding functionality results in additional system complexity.

Although a significant amount of research in real-time databases has been done in the past years, it has mainly focussed on various schemes for concurrency control, transaction scheduling, and logging and recovery, and less on configurability of software architectures. Research projects that are building real-time database platforms, such as ART-RTDB [7], BeeHive [14], DeeDS [1] and RODAIN [8], have monolithic structure, and are built for a particular real-time application. Hence, the issue of how to enable development of an embedded database system that can be tailored for different embedded and real-time applications arises.

2.2. Customization by Composition

Having embedded real-time database systems that would allow adding or replacing functionality in its architecture in a component-based manner would be beneficial for several reasons: (i) complexity of the database system and maintenance cost would be reduced; (ii) applications do not have to pay performance and cost penalty for using unneeded functionality, since unnecessary components do not have to be added to a system; and (iii) evolution of a system would be simplified, since new components with new required functionality could be plugged into the system.

Although some major database vendors (e.g., Oracle, Informix, Sybase, and Microsoft) have recognized that component-based development offers significant benefits, their component-based solutions are limited in terms of tailorability with, in most cases, no support for analysis of the composed system. To the best of our knowledge, the only existing research aimed to build completely configurable database management system (DBMS) is KIDS [4]. KIDS introduces a configurable DBMS composed out of components (DBMS subsystems), e.g., object management and transaction management. Customization of this system is improved by having reusable architectures, as well as components stored in a library. KIDS has a well-defined development process, available configuration support and optional analysis tools (which are missing in other component-based databases solutions, e.g., Garlic [3], Navajo [2]). From a real-time point of view all approaches discussed do not enforce real-time behavior. Issues related to embedded systems such as low-resource consumption are not addressed at all.

Tools to support the designer in the composition and analysis of the composed system are essential, and most of the component-based systems discussed do not provide adequate configuration and especially analysis support (vital for real-time systems).

3. COMET

3.1. Methodology

The COMET platform consists of two parts. The first part is a component library, which holds a set of components and aspects. The second part are tools that, based on the application requirements, support the designer when building an embedded database using components and aspects from the library. Our approach consists of extracting relevant database features by studying a number of application case studies in the first phase, followed by an implementation in the next phase, and evaluation of the design tools and database architecture on number of case studies in the final phase.

3.2. Challenges

A starting point towards the platform is to design a component-based database. In this respect, we are faced with the following challenges.

1. How do we encapsulate different database functionalities/services into components suitable for embedded

and real-time systems? This also includes: (i) defining the database component model appropriate for real-time and embedded systems; (ii) specifying component's real-time attributes as aspects such that the composed system is predictable; and (iii) defining rules for connecting components, such that composed system is reliable and fulfills the system's requirements.

2. How do we ensure predictability of the composed database system, with respect to response time, memory usage and CPU utilization? Furthermore, the challenge is to ensure easy integration of the composed database system into the run-time environment of the real-time system ensuring temporal behavior of the system.

Coping with these challenges is a difficult task, for several reasons. First, the COMET database should be suitable for low resource systems, primarily systems with constrained memory and power consumption. Thus, appropriate decomposition of the database functionality should be made, such that the functionality most likely (not) to be needed by most embedded systems is identified and encapsulated into components. Second, the COMET database should ensure predictable transaction execution. Additionally, most embedded systems are implemented in main-memory, thus requiring the COMET database to be a main-memory database.

We illustrate these challenges with an example of data management issues in a class of automotive control applications. Discussion in the example is primarily based on a case study within the project performed at Volvo Construction Equipment Components AB, Sweden, where we analyzed data management in existing real-time systems used to control wheel loaders and articulated haulers.

Example *Vehicle Control Systems*

Typically, control systems in the automotive industry are hard real-time safety-critical systems consisting of several distributed nodes. Each node implements specific functionality and can be viewed as a stand-alone real-time system, e.g., nodes can implement transmission, engine, or instrumental functions. The size of the nodes can vary significantly, from very small nodes, e.g., 32 Kb RAM, to larger nodes, e.g., 64 Kb RAM and 512 Kb Flash. Depending on the functionality of a node and the available memory, different database implementations are needed. For example, in safety-critical nodes tasks are often non-preemptive and scheduled off-line, avoiding concurrency by allowing only one task to be active at any given time. This, in turn, influences functionality of a database in a given node with respect to concurrency control. At this point we need to clarify the relationship between the task and the transaction. In control applications, such as this one, tasks are performing all updates on data. Therefore, from a database point of view, a task performing an update can be treated as a database transaction, as it encapsulates (one or more) transactions. Less critical nodes, having preemptable tasks, would require concurrency control mechanisms. Furthermore, some nodes require critical data to be logged, e.g., warning and errors, and require backups on startup and shutdown, while other nodes only have RAM (main-memory), and do not require non-volatile backup facilities from the database.

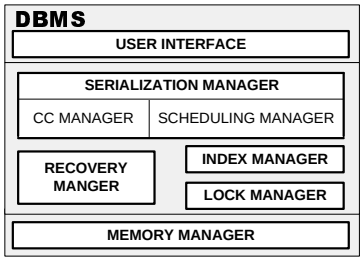


Figure 1. COMET functional decomposition

The discussion on different functionality needed from a database by different nodes could be continued further, but our goal with this short discussion is to show that introducing a component-based database has premise, since it would allow the database to be tailored to suit the needs of a real-time application in each node with respect to memory consumption, concurrency control, recovery, different scheduling techniques, transaction and storage models. This helps to optimize memory consumption in every node and allows databases to be integrated more easily with the run-time environment.

3.3. Functional and Aspectual Decomposition

In order to meet the demands for customization and fine-tuning of the COMET database for specific real-time and embedded applications, the decomposition of a database system must be done both on a functional level (functional decomposition), and on an aspect level (aspectual decomposition). Using aspects as separation of concerns, allows designing functional components that are cross-cut with real-time-specific and database-specific aspects, e.g., real-time aspect and transaction aspect, respectively. For example, a component that controls access to the data in the memory will not only have to manage retrieval and storage of data in a particular memory type (memory type could be viewed as an aspect), but also manage real-time properties of the data objects stored in memory (real-time constraints on data could also be viewed as an aspect).

The functional and aspectual decompositions of COMET are shown in figure 1 and 2, respectively. As-

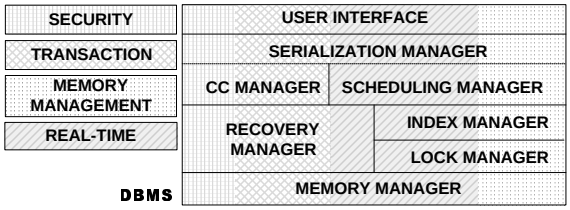


Figure 2. COMET aspectual decomposition on an application level (application aspects)

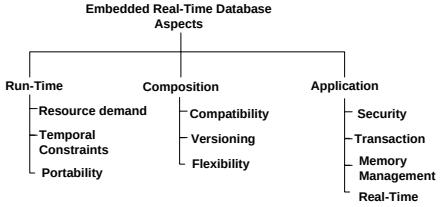


Figure 3. Classification of aspects in an embedded real-time database system

pectual decomposition also implies that components used for system composition are no longer traditional black box components[2], rather they are grey in that we can modify their internal behavior by applying different aspects.

Since aspects are considered to be a property of a system that affects its performance or semantics, and that cross-cuts the system's functionality [6], aspects that can be identified in an embedded real-time database system could be numerous. Thus, we must carefully define both components implementing certain functionality and different aspects that cross-cut these components.

3.4. COMET Functional Decomposition

In the initial design of the COMET we have identified six components (see figure 1): (i) user interface, a component that enables user to access data in the database and is doing query processing; (ii) serialization manager, a component that is in charge of ensuring serialization of transactions, and performs scheduling and concurrency control (CC); (iii) locking manager, a component that deals with locking of data; (iv) index manager, a component that deals with the indexing of the data; (v) recovery manager, a component that is in charged of recovery and logging of data in the database; and (vi) memory manager, a component that allows access to data possibly stored in different memory media.

The principle that lead us to this functional decomposition of the COMET database is primarily the need to have functionally exchangeable units that are loosely coupled, but with strong cohesion, i.e., internal strength.

Somewhat natural is the choice to have a user interface as a component, since different applications may require different ways of accessing data within the system.

In the control applications discussed previously, there are scenarios not requiring sophisticated transaction scheduling and concurrency control, e.g., the hard real-time system allows only one transaction to access the database at a time. Therefore, it would be helpful to have this functionality as a flexible part of the database architecture. Given that scheduling and concurrency control are database functionality that are tightly coupled, we choose to encapsulate the two into one component, a serialization manager. On the other hand, we must decouple these two functionality into two distinct sub-components within the serialization manager to allow flexible exchange of scheduling mechanisms and/or concurrency control mechanisms when applicable.

As COMET stores data in main-memory, there is a need for different recovery and logging techniques, depending on the type of the storage, e.g., non-volatile EEPROM or Flash. Thus, enabling the exchange of different recovery strategies through the recovery manager component would be beneficial for different applications. Similarly, memory management is required to be a flexible part of the architecture since there is a need for a component that would take care of the access to different types of memory. Same reasoning could be applied to decisions on index manager and locking manager, since indexing and locking are performed by the database fairly independent of, for example, serialization and recovery.

3.5. COMET Aspectual Decomposition

In order to classify the complexity of requirements in the database design, and address them in the design of the COMET in a systematic way, we classify aspects as follows (see figure 3):

- application aspects, which are aspects of the database towards the application,
- run-time aspects, which are aspects of the database towards the run-time system, and
- composition aspect, which are aspects that influence composition of the system.

Application aspects can change the internal behavior of components as they cross-cut them in the database system (as shown in figure 2). The application in this context refers

[2]The internal behavior and attributes of the black box component are strongly encapsulated and cannot be changed or modified.

to the embedded and real-time application towards which the database should be tailored. Application aspects can be divided into: memory management, real-time, security and transaction aspects. We view memory management as an application aspect of a database system, since size and allocation of memory influences the system's structure. Additionally, real-time properties are viewed as an application aspect as they influence the overall structure of the database system. Real-time properties could be further divided into categories, e.g., absolute and relative validity. Security is another application aspect that influences the system behavior and structure, e.g., user interface must be able to distinguish users with different security clearance. Depending on the application requirements, ACID (atomicity, consistency, isolation, and durability) properties of a transaction, for some applications, need to be relaxed. Thus, the transaction properties could be viewed as a transaction application aspect.

Run-time aspects are the most critical as they refer to aspects of the monolithic database system that need to be considered when integrating the database system into the run-time environment of a real-time system. Run-time aspects give information which is needed by the run-time system to ensure that integrating a database would not compromise timeliness, or memory consumption of the overall system. Therefore, each component could have declared resource demands in its resource demand aspect, and could have information of its temporal behavior, contained in the temporal constraints aspect, e.g., worst-case execution time (WCET), deadline and period. Additionally, each component must contain information of the platform with which it is compatible, e.g., real-time operating system supported, and other hardware related information. This information is contained as a portability aspect. It is imperative that this information is provided, to ensure predictability of the composed database system, ease the integration of the database into a real-time system and the underlying run-time system, and ensure portability to different hardware and/or software platforms.

Composition aspects do not cross-cut components, rather they are descriptive. Composition aspects describe with which components a component can be combined (compatibility aspect), version of the component (version aspect), and possibilities of extending the component with additional aspects (flexibility aspect).

While conventional AOP has focused predominantly on application aspects, having separation of aspects in different categories eases reasoning about different embedded and real-time related requirements, as well as the composition of the database and its integration into a real-time system. For example, a run-time system can define which (run-time) aspect the database must fulfill, so that proper database components can be chosen from the library, and composed into a monolithic system. This approach offers a significant flexibility, since additional aspect types can be added to components, and therefore, to the monolithic database system, further improving the integration of the database into a run-time system.

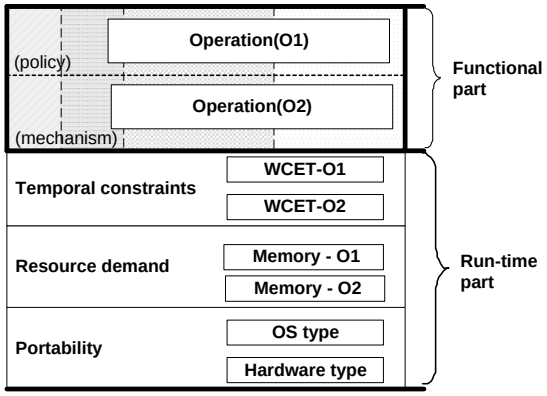


Figure 4. COMET unified component model

3.6. COMET Unified Component Model

Having described components, and categorized aspects that cross-cut or describe a component, we present a more uniform view of a component using a unified component model that unifies functional and aspectual decomposition. A unified component can be viewed as a component colored with aspects, both inside (application aspects), and outside (run-time and composition aspects). Due to space limitations of this paper, in the unified component model we do not consider composition aspects.

A unified component consists of the run-time system dependent part, and the functional part (see figure 4). This simplified view of a uniform component presents application aspects as vertical layers on the functionality, whereas run-time aspects are horizontal parallel layers to the functionality. Every component provides a set of database operations to the real-time application. These operations are contained in the functional layer of the unified component model. If needed, the functional part could be separated further on the policy layer, in the higher level, the mechanism layer, in the lower level. This is useful, for example, in cases when different scheduling (or CC) policies must be exchanged that work on the same mechanisms, e.g., `SetPriority()`.

When a transaction enters the database system, it will execute a number of operations provided by different components. To ensure predictability of transaction execution, WCET (run-time temporal aspect) of each operation needed by the transaction must be known. The total WCET of the

transaction could then be determined by aggregating the WCETs of different operations. WCET information for a specific operation within the component is contained in the run-time layer, as it depends on the operating system used, and the hardware platform on which the operation is to be executed, i.e., it depends on the platform aspects of the component. Of course, the scenario explained is a simplified one. In a more realistic situation we must consider composition aspects, and the complexity of all categories of aspects, i.e., the WCET is only one of the run-time temporal aspects.

4. Related Work

In this section we recognize the research in the area of component-based embedded and real-time systems, and the database and real-time research projects that are using aspects to separate concerns.

The focus in existing component-based real-time systems is enforcement of real-time behavior. In these systems a component is usually mapped to a task, e.g., passive component [13], binary component [5], and port-based object component [15]. Therefore, analysis of real-time components in these solutions addresses the problem of temporal scopes at a component level as task attributes [5, 13, 15]: worst case execution time, release time, deadline.

While there are some projects in the area of real-time systems and database systems that use the aspect-oriented programming paradigm, to the best of our knowledge, there is no project that looks on both these issues. Normally, these projects either focus on providing component-based platforms for development of real-time systems but without database functionality [13], or focus on providing a non-real-time database with limited tailorability using only aspects (i.e., no components) [11].

5. Summary

Designing an embedded database that can be tailored for different real-time applications implies careful balance between application requirement and run-time system requirements. The COMET database aims to balance these requirements, by having exchangeable components encapsulating different functionalities, and use three distinct types of aspects. The design of COMET is flexible, since initial classification of aspects could be, if needed, extended to include other (types of) aspects.

Our future work will focus on validation of the outlined design, and implementation of COMET. This includes: (i) developing a set of components and aspects, (ii) defining rules for composing these components into a real-time database system, and (iii) developing a set of tools to support the designer when composing and analyzing the database system.

References

[1] S. Andler, J. Hansson, J. Eriksson, J.Mellin, M. Berndtsson, and N.Elftring. DeeDS towards a distributed and active real-time database system. *ACM Sigmon Record*, 25, 1996.

[2] H. Bobzin. *Component Database Systems*, chapter The Architecture of a Database System for Mobile and Embedded Devices. Morgan Kaufmann Publishers, 2000.

[3] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. T. II, J. H. Williams, and E. L. Wimmers. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proceedings of the RIDE-DOM*, pages 124–131, Taipei, Taiwan, March 1995. IEEE Computer Society.

[4] A. Geppert, S. Scherrer, and K. R. Dittrich. KIDS: Construction of database management systems based on reuse. Technical Report ifi-97.01, Department of Computer Science, University of Zurich, September 1997.

[5] D. Isovic, M. Lindgren, and I. Crnkovic. System development with real-time components. In *Proceedings of ECOOP Workshop - Pervasive Component-Based Systems*, France, June 2000.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[7] Y. Kim, M. Lehr, D. George, and S. Son. A database server for distributed real-time systems: Issues and experiences. In *Proceedings of the 2nd Workshop on Parallel and Distributed Real-Time Systems*, Cancun, Mexico, April 1994.

[8] J. Lindström, T. Niklander, P. Porkka, and K. Raatikainen. A Distributed Real-Time Main-Memory Database for Telecommunication. In *Lecture Notes in Computer Science*, volume 1819.

[9] M. A. Olson. Selecting and implementing an embedded database system. *IEEE Computers*, 33(9):27–34, Sept. 2000.

[10] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, 1993.

[11] A. Rashid and E. Pulvermueller. From object-oriented to aspect-oriented databases. In *Proceedings of the DEXA 2000*, volume 1873 of *Lecture Notes in Computer Science*, pages 125–134. Springer-Verlag, 2000.

[12] J. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19, October 1988.

[13] J. Stankovic. VEST: A toolset for constructing and analyzing component based operating systems for embedded and real-time systems. Technical Report CS-2000-19, Department of Computer Science, University of Virginia, May 2000.

[14] J. Stankovic, S. Son, and J. Liebeherr. BeeHive: Global multimedia database support for dependable, real-time applications. In *2nd Workshop on Active Real-Time Databases*, 1997.

[15] D. S. Stewart. Designing software components for real-time applications. In *Proceedings of Embedded System Conference*, San Jose, CA, September 2000. Class 408, 428.

Contract Specifications of Software Components

Andreas Sjögren
Department of Computer Engineering
Mälardalen University
andreas.sjogren@mdh.se

28 March 2002

Abstract

Before Component-Based Development, the practice of building applications from pre-existing parts can be trusted in mission critical applications we have to be able to determine how it will behave. The key to be able to determine the behavior when we do not have access to the source code is the specification of the software component. This paper discusses different levels of specifications, with particular focus on what we call contracts.

1 Introduction

Ideally components are black boxes which provides the implementation of a set of named interfaces. A specification of a component is therefore a specification of its interfaces. The specification should provide all the information a client needs to know about the component's services and context dependencies [3].

The properties which we can define for a component can be divided into four different levels of increasingly negotiable properties [1, 3]. At the first level we have syntactic specifications, which are the minimum required to be able to use a component. At the second level we have possibilities to describe sequential behavioral properties between components. At the third level we can describe the synchronization of services. At the fourth level we have possibilites to specify quality of service. In different stages of a system's development process we have different usage and need of specifications.

The reminder of this paper is organized as follows. The next section discusses four different levels of precision of specifications of software components. Section 3 discusses different aspect of the usage of specifications. Section 4 introduces a specification technique based on UML. Section 5 discusses future work and finally, Section 6 summarizes the paper.

2 The Specification Levels

The properties which we can define for a component can be divided into four different levels of increasingly negotiable properties [1, 3]. We will examine these levels in this section.

Sometimes the word *contract* is used to describe a component however we specify them, but most often, as in this paper, the word contract is restricted to mean interfaces specified at the second or higher levels.

2.1 Syntactical Specification

At the first level we have syntactic specifications, the basic way of specifying a component which are the minimum required to be able to use a component. Here we can use Interface Definition Languages (IDL:s), or other typed programming languages. The specifications used in practice today are mostly limited to this level. Different IDL:s are used in component technologies such as Microsoft's COM/DCOM and in Object Management Group's CORBA. In Sun's Enterprise JavaBeans (EJB) and JavaBeans (JB) the ordinary Java programming language are used to define interfaces. In these techniques we can specify the following

- the services (operations) a component provides

- possible input and output parameters for the services

- possible exceptions (or errorcodes as in COM) that might be raised during the execution of the services

The information we can obtain from a syntactic specification is limited to what services the component offers, and the number and types of its parameters. Below is an example of a interface specification of an audioplayer as a EJB component, the audioplayer has two interfaces one for the player functionality and another for the track list (in EJB we are only able to define one interface for the services of a bean, therefore we have to define a super interface which derives from the two interfaces).

```
public interface IAudioplayer
  extends javax.ejb.EJBObject
  ,IPlayer
  ,IPlayList {
}

interface IPlayer {
  void play() throws RemoteException;
  void stop() throws RemoteException;
  void fwd() throws RemoteException;
  void prev() throws RemoteException;
}

interface IPlayList {
  void addTrack(Track t)
    throws RemoteException;
  void delTrack(int id)
    throws RemoteException;
  void moveTrack(int id)
    throws RemoteException;
  void selectTrack(int id)
    throws RemoteException;
}
```

The primary use of these specifications is for static or dynamic type checking that a client uses a component properly, and the limit for enable interoperability between independent components. Of course there is no guarantee that a service offered by a component behaves as expected from the name of it.

2.2 Behavioral Specification

Since syntactic specification does not says anything about the outcome of execution of one of its provided services there is no way to be protected against stupid implementations, such as letting the stop function in an audioplayer component instead change to the next track in the playlist. On the second identified level of specification we can extend the syntactical interface specifications with some semantic information.

Inspired by ideas from axiomatic program theory (see for instance [4]) Bertrand Meyer developed the design by contract theory as a part of the Eiffel language [5]. Here a contract specification can be associated with a software element. If we use these contracts we can specify a component's behaviour by use of boolean assertions, called pre- and post-conditions on the component's services, and constraints on component's state, called invariants. The pre-conditions specify assertions that a client must fulfil before the invocation of a service. The post-conditions are assertions a component guarantee will hold after a service has been invoked, provided the pre-condition was fulfilled. An operation will often depend on the state of the component. Therefore the contract has to include a partial model of the state which may be affected of the operation. Furthermore, a set of invariants may be associated with the specification. An invariant is a constraint on the state model which always will hold. In some techniques there is also possible to specify intra-interface dependencies.

On this level of specifications we can specify conditions that could go wrong and assign explicitly responsibilities to the client, or the component (the contractor). If a client respects the contract it can only call a service from a component in a state where the pre-condition and the invariants holds. Likewise, with respect to the contract, a component guarantees a service will perform the work specified in the post-condition, and in a state where the invariants holds.

With use of the iContract [6] extension tool for Java we can add contracts to the syntactical specification of the audioplayer component. The syntax for the specifications are a combination of a own specification language which is a subset of OCL and ordinary Java syntax. The semantic information is added as Javadoc comments. Here is an example of

pre- and post-conditions for the `addTrack` method.

```
/**
 *@pre  t != null
 *@post playlist.size()
        = playlist.size()@pre + 1;
 *@post playlist.includes(t);
*/
void addTrack(Track t)
  throws RemoteException;
```

The pre-condition states that `addTrack` has to be invoked with an ingoing parameter that is not `null`. The post-condition ensures that, if the pre-condition holds, the track will be added to the playlist and the size of the playlist will be increased with one track after the invocation. We can refer to the state of the pre-condition in the post-condition by using the `@pre` postfix.

In some specification techniques it is also possible to specify that components are dependent on interfaces implemented by other components. These interface dependencies are often specified, as the required and offered interfaces of the component, respectively.

2.3 Synchronizational Specification

At the third identified specification level we are able to describe dependencies between services provided by a component in terms of synchronization between method calls. When specifying operations for distributed component systems transactional behavior becames very important. Most component technologies (such as COM/DCOM, CORBA, EJB) provides rich schemes for the definition of transactional behavior. Therefore a specification needs to describe a mapping onto these technologies.

The specification issues to cover here is whether a service starts needs its own transaction or if it runs in an excisting transaction. This is a binary choice to take care of, so in the specification we only need a mechanism for call out the forced new transaction for those operations, and absence of this mechanism implies that the operation runs in an exising transaction. Java provides this functionality through the `synchronization` keyword. E.g. if we always needs to know that, whatever service other clients request, the `addTrack` method will execute behave correctly we can define this method to be mutual exclusive.

```
synchronized void addTrack(Track t)
  throws RemoteException;
```

2.4 Quality-of-Service Specification

At the fourth and final level we have possibilities to specify quality–of-service, such as maximum response time, precision of results, etc. How to specify these kind of extra-functional properties and environmental constraints is largely an open issue, and has recently started to be discussed in the software engineering comunity and it will likely impact the future of software component specification. It is of course of great concern in system with high requirements on reliability, such as embedded systems and realtime systems.

When we specify the quality of service we can statically specify the features the component will respect. Alternatively, we can have a more dynamic solution where a server and a client can negotiate about the features. Examples of quality-of-service attributes are

- maxium response time
- average response time
- precision of the result (the bounds)
- memory usage

3 Usage of Contracts

A contract is a formal agreement between two or more parties. It specifies the detail of the agreement in a non-ambiguous form. This involves stating the responsibilities of each party. Contracts do not have to be complete. Specifications are of nature nondeterministic. If we over-specify and make them deterministic we have put to strict bounds of how to do things.

We can distinguish between two different needs of contracts for software components [2]:

Usage the contract between a component's interface and the clients.

Realization the contract between a component specification and its implementation.

The people which use components are often not the same as who use them. This is the fact, even if we are far from independent third component markets and the most components which are used today are developed internally in an organization. Not all aspects of a specification is relevant to a client.

A usage contract describes the relationship between a component's interfaces and a client. This is specified by a specification of the operations and eventual semantic information, depending on which level of specification technique we using. A realization contract is a contract between a component specification and an implementation of the component. While an interface specification defines a set of behavior, a component specification defines an implementation and deployment boundary.

3.1 Monitoring Contracts

What happens when a system violates the contract is a question not directly related to the specification problem. This depends of course of the implementation. However, many techniques like iContract [?] and Eiffel [5] provides a whole environment for both possibilities of specifying contracts and monitoring them. But, on the other hand is these techniques more focused on providing ways of implementing contracts and not on specification of them. In iContract we can add an exception to the assertions which will be raised if the system violates the contract. Here is an example for the `addTrack` method of the audioplayer component.

```
/**
 *@post playlist.includes(t) #TrackException;
*/
void addTrack(Track t)
  throws RemoteException;
```

Then we need to monitor du execution and catch eventual raised exceptions.

4 Specifications in UML

In this section we look closer into one particular specification technique. This technique is based on UML and OCL, with some extensions. Here we can specify a contract on level two for the audioplayer component. The operation signatures will

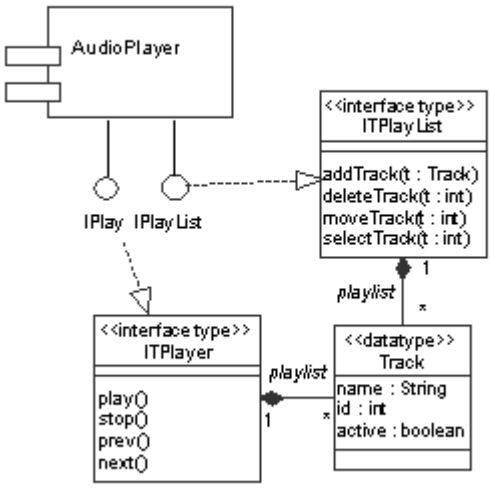


Figure 1: An UML specification of the Audioplayer component

look more or less the same as in Java specification on level 1, see Figure 1. We have associated a partial model of the components state to the component. The `<<interface type>>` stereotype is used to state that this is an interface type, interface type will be realized by an interface. Interface types can have a model of there state associated to them. We use the convention to prefix interface types with `IT`, and interfaces with an `I`. The audioplayer has two interfaces, which both is a realization of a corrsponding interface type. Both these interface types is associated to a number of tracks. We associates pre- and post-conditions, defined in OCL to each of the provided services. Here is an example of the pre- and post-conditions for the `addTrack` method of the `ITPlayer` interfacetype. The syntax for this example looks very similar to the iContract specification in Section 2, and has the same semantics.

```
context ITPlay::addTrack(in t:Track)
pre:
  t <> null
post:
  (playlist-playlist@pre)->size = 1
  and playlist.includes(t)
```

We can also add an inter-interface constraint, specifying that the set of tracks in the state model of the two interface must be the same.

```
context AudioPlayer
IPlay::playlist=IPlayList::playlist
```

In this technique we can also adds stereotypes which states that operations has to be executed in a own transaction and also add information about which interface a component uses and offers [2].

5 Future Work

As discussed above current component specification techniques are mostly focused on syntactic specifications. We will therefore investigate how we can use syntactic specifications which we have extended with some semantic information. Eventually, we will also specify extra-functional requirements in these specifications (this is a open issue, which has not been much research on so far). We will use well-known, techniques as UML to specify semantics (and maybe extra-functional properties) for components. By the use of standard techniques, and by providing tool support for the use of them, we hope to contribute to bridge the gap between formal specifications and practical implementations of these.

The modelling method presented in UML Components [2], and briefly discussed in Section 4, uses UML and OCL to write syntactic specifications, extended with semantic information in form of contracts. Currently we investigate how to use a variant of this method to specify components in an existing UML tool, and then generate stubs for the implementation of components. We will then experiment with specifications of components, and test how to specify both in the design of new components and to specify contracts for existing components. Contract can for example be added to existing components by use of wrappers, i.e. an "external contract". We will also perform analysis of examples of components with and without contracts (with internal or external contracts) to see how much this extra code will affect different properties (e.g. performance, time and space complexity, etc.).

6 Summary

We have in this paper discussed the for Component-Based Development very important concept of specification of software components, in particular to be able to use a CBD strategy in systems with high requirements, e.g. on reliability. We discussed four identified levels of specification and showed some examples of different specifications. We discussed how we can have different use of the specifications in different stages. We shortly introduced a particular specification technique based on UML, and finally discussed some further work with this method.

References

[1] A. Beugnard, J-M. Jezequel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, Vol. 32(7):38–45, July 1999.

[2] J. Chessman and J. Daniels. *UML Components - A simple Process for Specifying Component-Based Software.* Addison-Wesley, 2001.

[3] Ivica Crnkovic and Magnus Larsson, editors. *Building Reliable Component-Based Systems.* Artech House, June 2002.

[4] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, Vol. 12:576–580, 1969.

[5] B. Meyer. Applying design by contract. *Computer*, Vol. 25(10):40–51, Oct 1992.

[6] R. Kramer R. icontract - the java design by contract tool. In *Proceedings Technology of Object-Oriented Languages*, number TOOLS26. IEEE Computer Society, 1998.

Measurements of Software Maintainability

Rikard Land
Mälardalen University
Department of Computer Engineering
Box 883
SE-721 23 Västerås, Sweden
+46 (0)21 10 70 35
rikard.land@mdh.se
http://www.idt.mdh.se/~rld

ABSTRACT

In this position paper, we describe the research we have just initiated. We will investigate how the "maintainability" of a piece of software changes as time passes and it is being maintained by performing measurements on industrial systems. We present the notion of "maintainability", our hypotheses, and our approach.

Keywords

Software metrics, Halstead measure, maintainability, modifiability, software architecture, software deterioration.

1. INTRODUCTION

Many resources are spent on software maintenance. Thus, producing software that is easy to maintain may potentially save large costs. The problem of maintaining software is widely acknowledged in industry, and much has been written on how maintainability can be facilitated by e.g. tools and processes (see e.g. the IEEE International Conference on Software Maintenance, ICSM). However, you cannot control what you cannot measure, and there is yet no universal measure of maintainability. Some proposals have indeed been presented, but the very idea of measuring maintainability has inherent problems (these issues are discussed in section 2).

We can in research and practice discern two ways of discussing the term maintainability[1]: either it is used very informally, or it is considered possible to derive a measure directly from source code. To be fair, those adopting the second view admit that any formula describing maintainability as a function of e.g. "relative number of commented lines of source code" is of limited use, and those having the first view have a feeling that maintainability has something to do with program size and complexity.

[1] Although "maintainability" and "modifiability" are similar but by some not considered equivalent terms, will use the term maintainability exclusively throughout the paper.

We have identified two areas where there is little research done. First, it is well known that software systems deteriorate as time passes and changes are made to it. We intend to describe this in terms of how *maintainability changes as a system is being maintained* (see section 3.1), rather than verifying a measure using expert judgment as is usually done. Second, we only know of measurements on code level, and will thus perform *measurements on the architectural level* and compare measurements made on both levels (see section 3.2).

We are therefore about to start performing measures on the history of a number of industrial systems to see how maintainability has changed as changes are implemented. We hope to be able to identify "bad" and "good" types of changes, and learn from that how a system should be maintained.

2. WHAT IS MAINTAINABILITY?

Maintainability has previously been described mainly in two ways, either informally or as a function of directly measurable attributes.

2.1 Informal Descriptions

There are many text descriptions available, which are in essence very similar. We quote the IEEE Standard Glossary of Software Engineering Terminology:

> **maintainability.** […] The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. [15]

There are other examples of such descriptions [3,4,36]. Such descriptions capture what is intuitively meant with maintainability, but have some problems. They do not in any way guide in how to estimate or measure maintainability. Another problem is that if we follow this approach and try to measure maintainability as "effort", we should bear in mind that the common unit for effort, "man-month", is in itself very dubious [8].

We can also note that Pfleeger describes maintainability as "the *probability* that […] a maintenance activity can be

carried out within a stated time interval […] [it] ranges from 0 to 1" ([26], italics added).

2.2 Maintainability Measures

Despite the subjectivity of any attempt to measure maintainability, great effort has been put into constructing formulas for describing maintainability. Following the opinion that maintainability "is the set of attributes that bear on the effort needed to make specified modifications" [16], we describe maintainability according to this approach as a function of directly measurable attributes A_1 through A_n, that is:

$$M = f(A_1, A_2, \ldots, A_n) \quad \textbf{(1)}$$

On an informal level, this approach is quite appealing – it is intuitive that a maintainable system must be e.g. consistent and simple. However, there may be great difficulties in measuring those attributes and weighting them against each other and combine them in a function *f*. Any such attempt is therefore bound to a quite limited context – a particular programming language, organization, type of system, type of project; the skill and knowledge of the people involved must also be considered then drawing conclusions. Many researchers have tried to quantify maintainability in different types of measures [1,2,10,23,24,36], of which the most noticeable probably is the Maintainability Index, MI [24,32]. The Halstead source code measures proposed in the seventies [13,31] have been used for describing maintainability [31,32] (see section 3.4).

Typically, maintainability measures are validated using expert judgments about the *state* of different systems and modules [10,36], while we rather investigate how the measures *change* as the software is maintained. However, the value of performing such an evaluation before and after a change is implemented has already been discussed [2,10,27] (although the objective of such studies has been to verify cost predictions); we adopt this approach but pursue it even longer by investigating a complete evolution history (very long at least) of industrial systems, measuring such functions after each change.

2.3 Maintainability and Software Architecture

Our research interests includes software architecture [3,6,14,33] and component-based systems [35], in connection with "change". Within the software architecture literature, the terms "maintainability" and "modifiability" are often used informally as a desired feature [3,6,14,33] – indeed, it is one of the very goals with software architecture to make a system understandable, and thus maintainable, by providing abstractions on an appropriate level.

One important goal for research is to make it possible to accurately estimate maintenance costs; such estimations should be done early in the development, i.e. during the architectural design. Such prediction models are often based both on the argument that maintainability must be discussed in the context of particular changes – it might be easy to perform one particular change, while another is virtually impossible. The use of scenarios to evaluate maintainability has therefore been discussed, particularly on the architectural level [3-7,9,17-20]. SAAM [3,19] and ATAM [20] are general scenario-based evaluation techniques with which any quality attributes can be estimated on the architectural level; these have been reported useful in practice [3,18,21]. Bengtsson has suggested one cost estimation model where the type of change (new components, modified components, or new "plug-ins") of each change scenario is taken into account to calculate the estimated change effort [4].

3. OUR RESEARCH

We do not propose a new formula for maintainability, but will rather measure attributes A_i that have been suggested in literature to affect maintainability. We are not interested in comparing systems (i.e. answering whether system A with $M = 82$ is more maintainable than system B with $M = 81$) but rather discuss around how the maintainability of a single system change, thus connecting the notion of maintainability to the recognized problem of software aging and deterioration [25] (see section 3.1). The idea of comparing a measure before and after a change is made has been discussed [2,10]; however, we have not seen a study like the one we are describing in this paper, investigating changes over a long sequence of changes.

We will perform measurements on the architectural level as well (this is described in section 3.2).

3.1 Software Deterioration

It has been noticed that software deteriorates as it ages and is being maintained [3,6,25,33,34]; using the vocabulary of Fred Brooks, a system's *conceptual integrity* [8] degrades as changes are made to it. Our hypothesis is that this is discernible when measuring maintainability of a system during a long sequence of changes. If it is true that the system decays continuously, we should get a graph similar to the one in Figure 1.

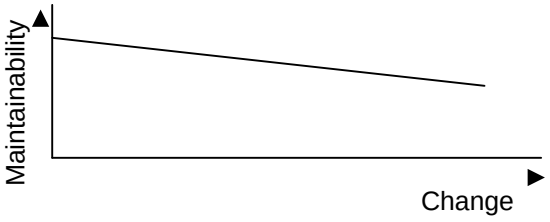


Figure 1. The system deteriorates (measured in maintainability) as changes are implemented[2].

[2] The "change" dimension could be thought of as "discrete time", with which we mean that a number of changes are implemented sequentially to a system.

We will perform measurements on a long sequence of versions of the same system, and investigate if it is possible to discern any trends. However, we do not expect to find a graph that is so easy to interpret as Figure 1; rather we expect to find that the maintainability sometimes increase and sometimes decrease. We hope to be able to correlate increases and decreases in maintainability measures to the descriptions of the changes made (as described by change requests). For example, the logical change "system restructuring" should cause the maintainability measure to increase, as Figure 2 describes.

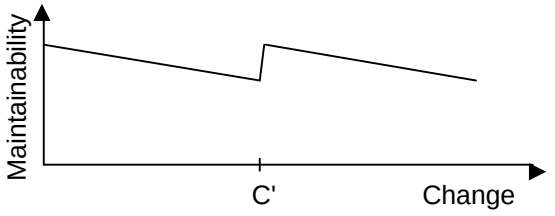


Figure 2. Change C' (a restructuring of the system) causes maintainability to (temporarily) increase.

Since there is no universally accepted definition of how to measure maintainability, we will measure a number of different measures that has been proposed as affecting maintainability. After we have collected data throughout a long history of at least two systems, we will focus on several related questions:

- Can the changes in maintainability be correlated to descriptions of the logical changes done? Are there any specific types of changes that cause the measure to increase or decrease significantly? Are there differences between "maintenance" changes and "modifications"?
- Do different measures display the same trend for the same change? It is quite possible that we cannot find any always-valid correlation – but is it possible to discern any types of changes that make all (or most) measures to show the same trend? For example, do "fault corrections" make two measures decrease or increase simultaneously? When do different measures show the opposite trends?

3.2 Architecture-Level Maintainability Measures

In the field of Component-Based software engineering, a future is pictured where more and more software is built from *components*, meaning binary executables (EXEs, DLLs, etc.) possibly developed out-of-house [35] (this trend is already discernible – we daresay that the absolute majority of systems developed today use existing components such as operating systems, databases, and graphical packages). Therefore, a big question is how to make component-based systems maintainable, and as part of that we must be prepared to measure and estimate maintainability on the architectural level when source code is no longer available. We will thus compare architectural and code-level measures to see if there is any correlation between these.

3.3 Which Systems?

The properties we wish the systems under investigation to have are listed below.

- The system should be much used and much maintained so that there is a long sequence of changes, which means much data.
- To make it possible to investigate the changes made to a system, it should have been developed using some sort of revision control system, from which any earlier version of the system can be retrieved. At least, it should be possible to retrieve all released versions, but this does not allow us to investigate individual changes.
- There should also be a way of tracking how physical changes correspond to logical changes; i.e. which lines in which files the correction of a certain bug affected.

However, a mature use of a revision control system and change requests implies that the developing organization is relatively mature; therefore, we can only expect our results to apply to other development projects with similar maturity.

3.4 Measures

This section lists the candidate measures we will use, and measures we have decided not to use. We have, for convenience, focused on attributes directly measurable from program code (we have therefore neither included measures including subjective ranking techniques, nor measures that includes documentation [1])

There is an abundance of proposed measures of program complexity and maintainability. We have not yet analyzed fully which measures will make sense, but have in literature identified the following candidate measures on "lexical level": Lines Of Code (LOC)[3], number of commented lines, Halstead Length, Halstead Volume, Halstead Effort, purity ratio, number of executable semicolons (the same as executable statements [22]), average variable span, average number of statements between two successive references to the same variable, number of blank lines, number of tokens, number of lines of data declarations, control structure nesting level, average number of commented lines per module, average number of LOC per module [13,22,36]; cyclomatic complexity [29]; readability of source code

[3] There are a number of ways to count LOC – with or without comments, blank lines, compiler directives etc. [11,28]; however, since we are mainly interested in the changes, we can expect the choice of definition of LOC to be of minor importance.

(defined as the ratio between lines of code and number of commented lines) [1]; number of knots [22]. We will also investigate the maintainability measures taxonomy by Oman et al where 92 measures are listed and classified [24].

In addition, we have some ideas of the measurements we will perform on the architectural level, but have yet to perform a thorough literature search. This could include counting dependencies between components; “fan-in” and “fan-out” [12]; number of calls in, number of calls out [22]; and other measures.

There are other “complexity measures”, which we will not use: neither the Function Point measure of software complexity [11,30], the Object Point measure included in the COCOMO 2 method [11], nor DeMarco’s specification weight metrics (“bang metrics”) [11], are directly measurable from source code. Each of these requires a manual moment since not all parameters are measurable from source code. One example it the rating of items as “simple”, “average”, or “complex”. It is of course highly impractical to include manual work to evaluate a large number of subsequent versions, and there is a high risk of mistakes in counting or unfairness in rating. Fenton and Pfleeger list 11 limitations with the Function Point measure [11] (although all of these need not be disadvantages in our case). Also, these measures were rather designed for cost estimations (before source code is available) than of performing measurements.

4. SUMMARY

We will, through measurements, investigate the maintainability of at least two industrial systems. Although there are proposals on how to measure “maintainability” on a given piece of software, we are mainly interested how such measures have changed over time as the software is being maintained. We hope to be able to discern patterns in why some changes make the maintainability decrease and others make it increase. We will also compare measurements on the lexical level and on the architectural level.

We will use this description in our future work on software architecture and components, and hope to be able to describe how to design a system to make it maintainable.

REFERENCES

[1] Aggarwal K. K., Singh Y., and Chhabra J. K., "An Integrated Measure of Software Maintainability", In *Proceedings of Annual Reliability and Maintainability Symposium*, IEEE, 2002.

[2] Ash D., Alderete J., Yao L., Oman P. W., and Lowther B., "Using software maintainability models to track code health", In *Proceedings of International Conference on Software Maintenance*, IEEE, 1994.

[3] Bass L., Clements P., and Kazman R., *Software Architecture in Practice*, Addison-Wesley, 1998.

[4] Bengtsson P., "Architecture-Level Modifiability Analysis", Ph.D. Thesis, Blekinge Institute of Technology, Sweden, 2002

[5] Bengtsson P. and Bosch J., "Architecture Level Prediction of Software Maintenance", In *Proceedings of 3rd European Conference on Software Maintenance and Reengineering (CSMR'99)*, IEEE Computer Society, 1999.

[6] Bosch J., *Design & Use of Software Architectures*, Addison-Wesley, 2000.

[7] Bosch, J. and Bengtsson, P., An Experiment on Creating Scenario Profiles for Software Change, report ISSN 1103-1581, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, 1999.

[8] Brooks F. P., *The Mythical Man-Month - Essays On Software Engineering, 20th Anniversary Edition*, Addison-Wesley Longman, 1995.

[9] Clements P., Kazman R., and Klein M., *Evaluating Software Architectures: Methods and Case Studies*, Addison Wesley, 2000.

[10] Coleman D., Ash D., Lowther B., and Oman P., Using Metrics to Evaluate Software System Maintainability, *IEEE Computer*, volume 27, issue 8, 1994.

[11] Fenton N. E. and Pfleeger S. L., *Software Metrics - A Rigorous & Practical Approach*, PWS Publishing Company, 1997.

[12] Grady R.B., Successfully Applying Software Metrics, *IEEE Computer*, volume 27, issue 9, 1994.

[13] Halstead M. H., *Elements of Software Science, Operating, and Programming Systems Series Volume 7*, Elsevier, 1977.

[14] Hofmeister C., Nord R., and Soni D., *Applied Software Architecture*, Addison-Wesley, 2000.

[15] IEEE, IEEE Standard Glossary of Software Engineering Terminology, report IEEE Std 610.12-1990, IEEE, 1990.

[16] ISO/IEC, Information technology - Software product quality - Part 1: Quality model, report ISO/IEC FDIS 9126-1:2000 (E), ISO, 2000.

[17] Kazman R., Abowd G., Bass L., and Clements P., Scenario-Based Analysis of Software Architecture, *IEEE Software*, volume 13, issue 6, 1996.

[18] Kazman R., Barbacci M., Klein M., and Carriere J., "Experience with Performing Architecture Tradeoff Analysis Method", In *Proceedings of The International Conference on Software Engineering, New York*, 1999.

[19] Kazman R., Bass L., Abowd G., and Webb M., "SAAM: A Method for Analyzing the Properties of Software Architectures", In *Proceedings of The 16th International Conference on Software Engineering*, 1994.

[20] Kazman R., Klein M., Barbacci M., Longstaff T., Lipson H., and Carriere J., "The Architecture Tradeoff Analysis Method", In *Proceedings of The Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), (Monterey, CA)*, 1998.

[21] Land R., "Improving Quality Attributes of a Complex System Through Architectural Analysis - A Case Study", In *Proceedings of 9th IEEE Conference on Engineering of Computer-Based Systems*, IEEE, 2002.

[22] Lanning D.L. and Khoshgoftaar T. M., Modeling the Relationship Between Source Code Complexity and Maintenance Difficulty, *IEEE Computer*, volume 27, issue 9, 1994.

[23] Oman P. and Hagemeister J., "Metrics for Assessing a Software System's Maintainability", In *Proceedings of Conference on Software Maintenance*, IEEE, 1992.

[24] Oman, P., Hagemeister, J., and Ash, D., A Definition and Taxonomy for Software Maintainability, report SETL Report 91-08-TR, University of Idaho, 1991.

[25] Parnas D. L., "Software Aging", In *Proceedings of The 16th International Conference on Software Engineering*, IEEE Press, 1994.

[26] Pfleeger S. L., *Software Engineering, Theory and Practice*, Prentice-Hall, Inc., 1998.

[27] Ramil J. F. and Lehman M. M., "Metrics of Software Evolution as Effort Predictors - A Case Study", In *Proceedings of International Conference on Software Maintenance*, IEEE, 2000.

[28] Rozum, J. A. and Florac, W. A., A DoD Software Measurement Pilot: Applying the SEI Core Measures, report CMU/SEI-94-TR-016, Software Engineering Institute, 1995.

[29] SEI Software Technology Review, *Cyclomatic Complexity*, URL: http://www.sei.cmu.edu/, 2002

[30] SEI Software Technology Review, *Function Point Analysis*, URL: http://www.sei.cmu.edu/, 2002

[31] SEI Software Technology Review, *Halstead Complexity Measures*, URL: http://www.sei.cmu.edu/, 2002

[32] SEI Software Technology Review, *Maintainability Index Technique for Measuring Program Maintainability*, URL: http://www.sei.cmu.edu/, 2002

[33] Shaw M. and Garlan D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[34] Sommerville I., *Software Engineering*, Addison-Wesley, 2001.

[35] Szyperski C., *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[36] Zhuo F., Lowther B., Oman P., and Hagemeister J., "Constructing and testing software maintainability assessment models", In *Proceedings of First International Software Metrics Symposium*, IEEE, 1993.

TIMES - A Tool for Modelling and Implementation of Embedded Systems

Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi*

Uppsala University, Sweden.
E-mail: {tobiasa,elenaf,leom,paupet,yi}@docs.uu.se.

1 Introduction

TIMES is a modelling and schedulability analysis tool for embedded real-time systems, developed at Uppsala University in 2001. It is appropriate for systems that can be described as a set of preemptive or non-preemptive tasks which are triggered periodically or sporadically by time or external events. It provides a graphical interface for editing and simulation, and an engine for schedulability analysis.

The main features of TIMES are:

- A graphical editor for timed automata extended with tasks [FPY02], which allows the user to model a system and the abstract behaviour of its environment In addition the user may specify a set of preemptive or non-preemtive tasks with parameters such as (relative) deadline, execution time, priority, etc.
- A simulator, in which the user can validate the dynamic behaviour of the system and see how the tasks execute according to the task parameters and a given scheduling policy. The simulator shows a graphical representation of the generated trace showing the time points when the tasks are released, invoked, suspended, resumed, and completed.
- A verifier for schedulability analysis, which is used to check if all reachable states of the complete system are schedulable that is, all task instances meet their deadlines. A symbolic algorithm has been developed based on the DBM techniques and implemented based on the verifier of the UPPAAL tool [LPY97].
- A code generator for automatic synthesis of C-code on LegoOS platform from the model. If the automata model is schedulable according to the schedulability analyser the execution of the generated code will meet all the timing constraints specified in the model and the tasks.

An screen-shot of the TIMES tool is shown in Fig. 1. In section 3 we describe the tool and its main functionalities in more details. The modelling language and the theoretical foundation of TIMES is based on the model of timed automata with tasks, described in the following section.

* Corresponding author: Wang Yi, Department of Information Technology, Uppsala University, Box 325, 751 05, Uppsala, Sweden. Email: yi@docs.uu.se

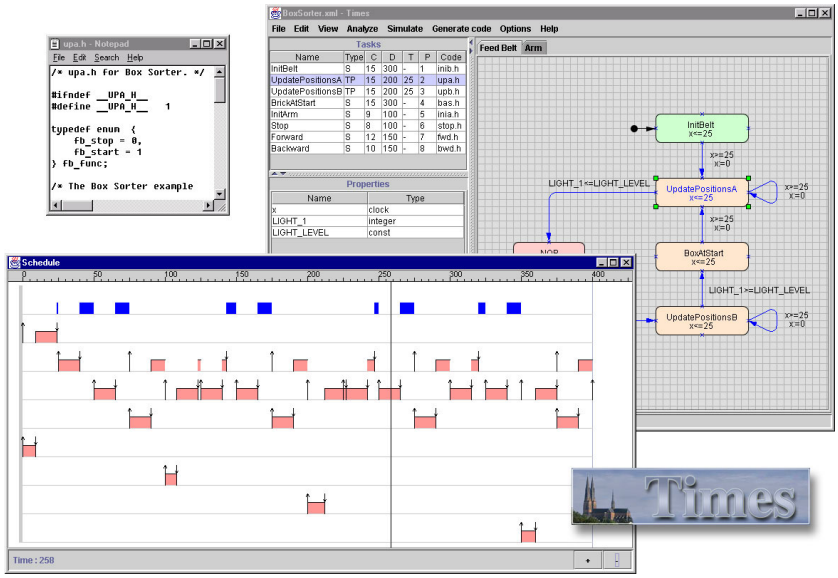


Fig. 1. Screen-shot of the TIMES tool.

2 Input Language

The core of the input language of TIMES is timed automata with real time tasks (TAT), described in details in [FPY02] included in this volume, with one major addition that shared variables between automata and tasks are allowed. Here we give a brief introduction to the model. A TAT is a timed automaton extended with tasks triggered by events. A task is an executable program (written in a programming language e.g. C) characterized by its worst execution time and deadline, and possibly other parameters such as priorities etc. for scheduling. A task may update a set of variables using assignments in the form $x := E$ where x is a variable and E is an expression (computed by the task and the value of E is returned when the task is finished). The variables may also be changed and tested by an automaton. Intuitively an edge leading to a location in the automaton denotes an event triggering the task, and the guard (clock constraints) on the transition specifies the possible arrival times of the event. This allows us to describe concurrency and synchronization, and real time tasks which may be periodic, sporadic, preemptive and (or) non-preemptive, with or without precedence constraints. An automaton is schedulable if there exists a (preemptive or non-preemptive) scheduling strategy such that all possible sequences of events accepted by the automaton are schedulable in the sense that all associated tasks can be computed within their deadlines.

Semantically, an extended timed automaton may perform two types of transitions just as standard timed automata. But the difference is that delay transitions correspond to the execution of running tasks with highest priority (or earliest

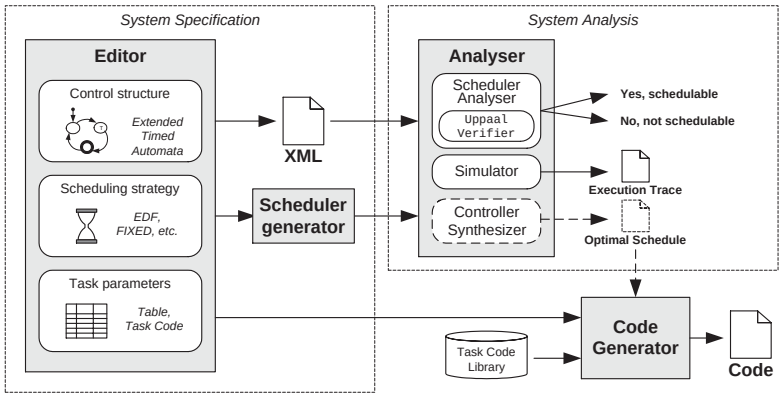


Fig. 2. Overview of the TIMES tool.

deadline) and idling for the other tasks waiting to run. Discrete transitions corresponds to the arrival of new task instances. Whenever a task is triggered, it will be put in a task queue for execution (corresponding to the ready queue in operating systems). Thus there may be a number of processes (released tasks) running logically in parallel during the execution of an automaton.

The scheduling problem of TAT is to verify that all released tasks are guaranteed to always meet their deadlines when executed according to a given scheduling policy. In TIMES the analysis is performed by transforming a TAT system into ordinary timed automata extended with subtraction operations on clocks, and encoding the schedulability problem to a reachability problem as described in [FPY02].

3 Tool Overview

An overview of the TIMES tool is shown in Fig. 2. The tool is divided in three parts: a system specification part, a system analysis part, and a code generator[1].

In the system specification part, the user models a system to be analysed. A system specification in TIMES consists of three parts: the control automata modelled as a network of timed automata extended with tasks [FPY02], a task table with information about the processes triggered (released) when the control automata changes location, and a scheduling policy.

The System Editor tool, shown in Fig. 1, is a an editor for drawing the control automata of the system model. It also provides a table for defining the task parameters. The task parameters currently supported are: (relative) deadline, execution time, period, priority, a reference to the task code, and a field indicating the task behaviour. A task has one of the following three behaviours: "S"

[1] The components indicated with dashed lines in Fig. 2 are planed extension not yet included.

for sporadic, "TP" for temporarily periodic, and "P" for periodic. The currently supported scheduling polices are: first-come first-served, fixed priority (i.e. according to the priorities assigned in the task table), rate monotonic, deadline monotonic, and earliest-deadline first. All polices can be either preemptive or non-preemptive.

The output of the System Editor is an XML representation of the control automata. The information from the task table and the scheduling policy, are used by the Scheduler Generator to generate a scheduler automaton that is composed in parallel with the controller automata to ensure that the system behaves according to the scheduling policy and the task parameters. If the scheduling policy is non-preemptive, the scheduler automaton is an ordinary timed automaton. If the scheduling policy is preemptive, the scheduler automaton is modelled as a variant of timed automata in which clock variables may be updated by subtractions. For more information about how to solve scheduling problems of timed automata with tasks using timed automata, see [FPY02].

The parallel composition of the control automata and the scheduler automaton is used as input to the System Analyser that consist of two main components: a Simulator, and a Schedule Analyser. In the Simulator the user may debug the system by exploring the dynamic behaviour of the system model and observe how the tasks execute according to the chosen scheduling policy. Screen-shots of the Simulator are shown in Fig. 1 and 4. As shown, the Simulator displays a diagram with $n + 1$ lines, where n is the number of tasks. On the upper line, it is indicated in blue (or black) when no task is executing. The lower n lines are associated with one task each, and used to show in red (or gray) when the corresponding task is executing. As time progresses the diagram grows to the right (i.e. the time line goes from left to right). Note that at any moment in time either, one or zero tasks are executing, or a switch takes place.

From the System Analyser it is also possible to invoke the Schedule Analyser that performs schedulability analysis of the system. The analysis is performed by rephrasing scheduling to a reachability problem that is solved with an extended version of the verifier of the UPPAAL tool [LPY97]. In case the output of the analysis is negative, the analyser generates a trace of the system that ends in a state in which one of the system task fails to meet its deadline.

The Code Generator of TIMES uses the control automata and the task programs to synthesise executable C-code. Currently the only supported platform is the LegOS operating system. Support for other platforms will be included in future versions.

4 Example: Box Sorter

In this section, we describe how TIMES is applied to the box sorter example of [LPY97]. The system sorts red and black boxes arriving on a belt by kicking the black boxes of the belt. The physical components are the feed belt, a light sensor, and a kick-off arm. The programs controlling the feed belt and the arm are modelled using two timed automata with tasks, as shown in Fig. 3.

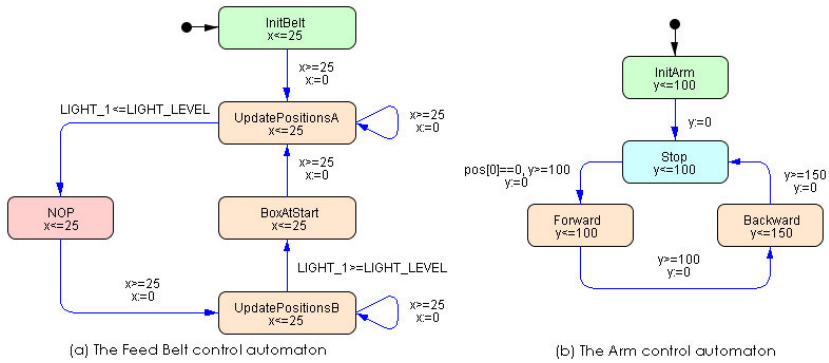


Fig. 3. The two Control Automata of the Box Sorter.

The tasks of the automaton in Fig. 3(a) uses a global array named pos to store the positions of the black boxes on the belt. The automaton in Fig. 3(b) controls the kick-off arm.

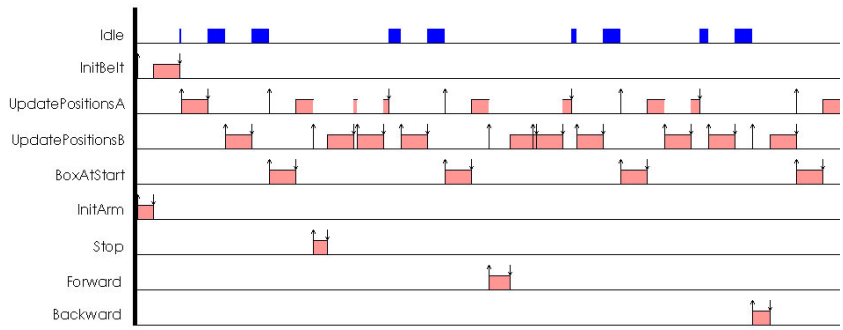


Fig. 4. The Box Sorter Schedule.

Fig. 4 shows an execution trace of the box sorter system, as it appears in the simulator window of the TIMES tool. Note how the simulator indicates that e.g. the second invocation of task UpdatePositionsA is delayed and preempted twice by higher priority tasks.

References

1. Elena Fersman, Paul Pettersson, and Wang Yi. Timed Automata with Asynchronous Processes: Schedulability and Decidability. In *Proc. of the 8*th *Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2002.
2. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

Timed Automata with Asynchronous Processes: Schedulability and Decidability

Elena Fersman, Paul Pettersson and Wang Yi*
Uppsala University, Sweden

Abstract. In this paper, we exend timed automata with asynchronous processes i.e. tasks triggered by events as a model for real-time systems. The model is expressive enough to describe concurrency and synchronization, and real time tasks which may be periodic, sporadic, preemptive or non-preemptive. We generalize the classic notion of schedulability to timed automata. An automaton is schedulable if there exists a scheduling strategy such that all possible sequences of events accepted by the automaton are schedulable in the sense that all associated tasks can be computed within their deadlines. We believe that the model may serve as a bridge between scheduling theory and automata-theoretic approaches to system modeling and analysis. Our main result is that the schedulability checking problem is decidable. To our knowledge, this is the first general decidability result on dense-time models for real time scheduling without assuming that preemptions occur only at integer time points. The proof is based on a decidable class of updatable automata: timed automata with subtraction in which clocks may be updated by subtractions within a bounded zone. The crucial observation is that the schedulability checking problem can be encoded as a reachability problem for such automata. Based on the proof, we have developed a symbolic technique and a prototype tool for schedulability analysis.

1 Introduction

One of the most important issues in developing real time systems is *schedulability analysis* prior to implementation. In the area of real time scheduling, there are well-studied methods [8] e.g. rate monotonic scheduling, that are widely applied in the analysis of periodic tasks with deterministic behaviours. For *non-periodic* tasks with non-deterministic behaviours, there are no satisfactory solutions. There are approximative methods with pessimistic analysis e.g. using periodic tasks to model sporadic tasks when control structures of tasks are not considered. The advantage with automata-theoretic approches e.g. using timed automata in modeling systems is that one may specify general timing constraints on events and model other behavioural aspects such as concurrency and synchronization. However, it is not clear how timed automata can be used for schedulability analysis because there is no support for specifying resource requirements and hard time constraints on computations e.g. deadlines.

* Corresponding author: Wang Yi, Department of Information Technology, Uppsala University, Box 325, 751 05, Uppsala, Sweden. Email: yi@docs.uu.se

Following the work of [11], we study an extended version of timed automata with asynchronous processes i.e. tasks triggered by events. A task is an executable program characterized by its worst case execution time and deadline, and possibly other parameters such as priorities etc for scheduling. The main idea is to associate each location of an automaton with a task (or a set of tasks in the general case). Intuitively a transition leading to a location in the automaton denotes an event triggering the task and the guard (clock constraints) on the transition specifies the possible arrival times of the event. Semantically, an automaton may perform two types of transitions. Delay transitions correspond to the execution of running tasks (with highest priority) and idling for the other waiting tasks. Discrete transitions correspond to the arrival of new task instances. Whenever a task is triggered, it will be put in the scheduling queue for execution (i.e. the ready queue in operating systems). We assume that the tasks will be executed according to a given scheduling strategy e.g. FPS (fixed priority scheduling) or EDF (earliest deadline first). Thus during the execution of an automaton, there may be a number of processes (released tasks) running in parallel (logically).

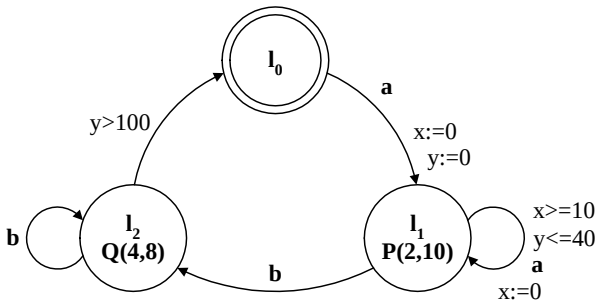


Fig. 1. Timed automaton with asynchronous processes.

For example, consider the automaton shown in Figure 1. It has three locations l_0, l_1, l_2, and two tasks P and Q (triggered by a and b) with computing time and relative deadline in brackets $(2, 10)$, and $(4, 8)$ respectively. The automaton models a system starting in l_0 may move to l_1 by event a at any time, which triggers the task P. In l_1, as long as the constraints $x \geq 10$ and $y \leq 40$ hold and event a occurs, a copy of task P will be created and put in the scheduling queue. However, in l_1, it can not create more than 5 instances of P because the constraint $y \leq 40$ will be violated after 40 time units. In fact, every copy will be computed before the next instance arrives and the scheduling queue may contain at most one task instance and no task instance will miss its deadline in l_1. In l_1, the system is also able to accept b, trigger Q and then switch to l_2. In l_2, because there is no constraints labelled on the b-transition, it may accept any number of b's, and create any number of Q's in 0 time. This is the so-called

zeno-behavior. However, after more than two copies of Q, the queue will be non-schedulable. This means that the system is non-schedulable. Thus, zeno-behaviour will correspond to non-schedulability, which is a natural property of the model.

We shall formalize the notion of schedulability in terms of reachable states. A state of an extended automaton will be a triple (l, u, q) consisting of a location l, a clock assignment u and a task queue q. The task queue contains pairs of remaining computing times and relative deadlines for all released tasks. Naturally, a state (l, u, q) is schedulable if q is schedulable in the sense there exists a scheduling strategy with which all tasks in q can be computed within their deadlines. An automaton is schedulable if all reachable states of the automaton are schedulable. Note that the notion of schedulability here is relative to the scheduling strategy. A task queue which is not schedulable with one scheduling strategy, may be schedulable with another strategy. In [11], we have shown that under the assumption that the tasks are non-preemptive, the schedulability checking problem can be transformed to a reachability problem for ordinary timed automata and thus it is decidable. The result essentially means that given an automaton it is possible to check whether the automaton is schedulable with any *non-preemptive* scheduling strategy. For *preemptive scheduling* strategies, it has been suspected that the schedulability checking problem is undecidable because in *preemptive scheduling* we must use stop-watches to accumulate computing times for tasks. It appears that the computation model behind preemptive scheduling is stop-watch automata for which it is known that the reachability problem is undecidable. Surprisingly the above intuition is wrong. In this paper, we establish that the schedulability checking problem for extended timed automata is decidable for preemptive scheduling. In fact, our result applies to not only preemptive scheduling, but any scheduling strategy. That is, for a given extended timed automata, it is checkable if there exists a scheduling strategy (preemtive or non-preemtive) with which the automaton is schedulable. The crucial observation in the proof is that the schedulability checking problem can be translated to a reachability problem for a decidable class of updatable automata, that is, timed automata with subtraction where clocks may be updated with subtraction only in a bounded zone.

The rest of this paper is organized as follows: Section 2 presents the syntax and semantics of timed automata extended with tasks. Section 3 describes scheduling problems related to the model. Section 4 is devoted to the main proof that the schedulability checking problem for preemptive scheduling is decidable. Section 5 concludes the paper with summarized results and future work, as well as a brief summary and comparison with related work.

2 Timed Automata with Tasks

Let $\mathcal{P}$, ranged over by P, Q, R, denote a finite set of task types. A task type may have different instances that are copies of the same program with different inputs. We further assume that the *worst case execution times* and *hard deadlines* of

tasks in $\mathcal{P}$ are known [1]. Thus, each task P is characterized as a pair of natural numbers denoted $P(C, D)$ with $C \leq D$, where C is the worst case execution time of P and D is the relative deadline for P. We shall use $C(P)$ and $D(P)$ to denote the worst case execution time and relative deadline of P respectively.

As in timed automata, assume a finite alphabet $\mathcal{A}ct$ ranged over by a, b etc and a finite set of real-valued clocks $\mathcal{C}$ ranged over by x_1, x_2 etc. We use $\mathcal{B}(\mathcal{C})$ ranged over by g to denote the set of conjunctive formulas of atomic constraints in the form: $x_i \sim C$ or $x_i - x_j \sim D$ where $x_i, x_j \in \mathcal{C}$ are clocks, $\sim \in \{\leq, <, \geq, >\}$, and C, D are natural numbers. The elements of $\mathcal{B}(\mathcal{C})$ are called *clock constraints*.

Definition 1. *A timed automaton extended with tasks, over actions $\mathcal{A}ct$, clocks $\mathcal{C}$ and tasks $\mathcal{P}$ is a tuple $\langle N, l_0, E, I, M\rangle$ where*

- *$\langle N, l_0, E, I\rangle$ is a timed automaton where*
  - *N is a finite set of locations ranged over by l, m, n,*
  - *$l_0 \in N$ is the initial location, and*
  - *$E \subseteq N \times \mathcal{B}(\mathcal{C}) \times \mathcal{A}ct \times 2^{\mathcal{C}} \times N$ is the set of edges.*
  - *$I : N \mapsto \mathcal{B}(\mathcal{C})$ is a function assigning each location with a clock constraint (a location invariant).*
- *$M : N \hookrightarrow \mathcal{P}$ is a partial function assigning locations with tasks [2].*

Intuitively, a discrete transition in an automaton denotes an event triggering a task annotated in the target location, and the guard on the transition specifies all the possible arrival times of the event (or the annotated task). Whenever a task is triggered, it will be put in the scheduling (or task) queue for execution (corresponding to the ready queue in operating systems).

Clearly extended timed automata are at least as expressive as timed automata; in particular, if M is the empty mapping, we will have ordinary timed automata. It is a rather general and expressive model. For example, it may model time-triggered periodic tasks as a simple automaton as shown in Figure 2(a) where P is a periodic task with computing time 2, deadline 8 and period 20. More generally it may model systems containing both periodic and sporadic tasks as shown in Figure 2(b) which is a system consisting of 4 tasks as annotation on locations, where P_1 and P_2 are periodic with periods 10 and 20 respectively (specified by the constraints: x=10 and x=20), and Q_1 and Q_2 are sporadic or event driven by event a and b respectively.

In general, there may be a number of released tasks running logically in parallel. For example, an instance of Q_2 may be released before the preceding instance of P_1 is finished because there is no constraint on the arrival time of b_2. This means that the queue may contains at least P_1 and Q_2. In fact, instances of all four task types may appear in the queue at the same time.

[1] Tasks may have other parameters such as fixed priority for scheduling and other resource requirements e.g. on memory consumption. For simplicity, in this paper, we only consider computing time and deadline.

[2] Note that M is a partial function meaning that some of the locations may have no task. Note also that we may also associate a location with a set of tasks instead of a single one. It will not introduce technical difficulties.

Shared Variables. To have a more general model, we may introduce data variables shared among automata and tasks. For example, shared variables can be used to model precedence relations and synchronization between tasks. Note that the sharing will not add technical difficulties as long as their domiains are finite. For simplicity, we will not consider sharing in this paper. The only requirement on the completion of a task is given by the deadline. The time when a task is finished does not effect the control behavior specified in the automaton.

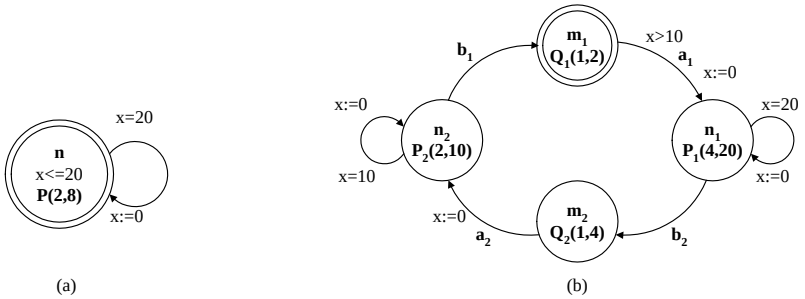


Fig. 2. Modeling Periodic and Sporadic Tasks.

Parallel Composition. To handle concurrency and synchronization, a parallel composition of extended timed automata may be defined as a product automaton in the same way as for ordinary timed automata (e.g. see [16]). Note that the parallel composition here is only an operator to construct models of systems based on their components. It has nothing to do with multi-processor scheduling. A product automaton may be scheduled to run on a one- or multi-processor system.

Semantically, an automaton may perform two types of transitions. Delay transitions correspond to the execution of running tasks with highest priority (or earliest deadline) and idling for the other tasks waiting to run. Discrete transitions corresponds to the arrivals of new task instances.

We represent the values of clocks as functions (i.e. clock assignments) from $\mathcal{C}$ to the non–negative reals $\mathcal{R}_{\geq 0}$. We denote by $\mathcal{V}$ the set of clock assignments for $\mathcal{C}$. Naturally, a semantic state of an automaton is a triple (l, u, q) where l is the current location, $u \in \mathcal{V}$ denotes the current values of clocks, and q is the current task queue. We assume that the task queue takes the form: $[P_1(c_0, d_0), P_2(c_1, d_1)...P_n(c_n, d_n)]$ where $P_i(c_i, d_i)$ denotes a released instance of task type P_i with remaining computing time c_i and relative deadline d_i.

Assume that there are a fixed number of processors running the released task instances according to a certain scheduling strategy Sch e.g. FPS (fixed priority scheduling) or EDF (earliest deadline first) which sorts the task queue whenever new tasks arrives according to task parameters e.g. deadlines. An action tran-

sition will result in a sorted queue including the newly released tasks by the transition. A delay transition with t time units is to execute the task in the first positon of the queue with t time units. Thus the delay transition will decrease the computing time of the first task with t. If its computation time becomes 0, the task should be removed from the queue. Moreover, all deadlines in the queue will be decreased by t (time has progressed by t). To summarize the above intuition, we introduce the following functions on task queues:

- Sch is a sorting function for task queues (or lists), that may change the ordering of the queue elements only. For example, $\mathsf{EDF}([P(3.1,10),Q(4,5.3)]) = [Q(4,5.3),P(3.1,10)])$. We call such sorting functions a scheduling strategy that may be preemptive or non-preemptive [3].
- Run is a function which given a real number t and a task queue q returns the resulted queue after t time units of execution according to available resources. For simplicity, we assume that only one processor is available [4]. Then the meaning of $\mathsf{Run}(q,t)$ should be obvious and it can be defined inductively as follows: $\mathsf{Run}(q,0) = q$, $\mathsf{Run}([P_0(c_0,d_0),P_1(c_1,d_1)...P_n(c_n,d_n)],t)$ $= \mathsf{Run}([P_1(c_1,d_1-c_0)\ ...\ P_n(c_n,d_n-c_0)],t-c_0)$ if $c_0 \leq t$ and $\mathsf{Run}([P_1(c_0,d_0)$ $...\ P_n(c_n,d_n)],t) = [P_1(c_0-t,d_0-t)\ ...\ P_n(c_n,d_n-t)]$ if $c_0 > t$. For example, let $q = [Q(4,5),P(3,10)]$. Then $\mathsf{Run}(q,6) = [P(1,4)]$ in which the first task is finished and the second has been executed for 2 time units.

We use $u \models g$ to denote that the clock assignment u satisfies the constraint g. For $t \in \mathcal{R}_{\geq 0}$, we use $u+t$ to denote the clock assignment which maps each clock x to the value $u(x)+t$, and $u[r \mapsto 0]$ for $r \subseteq \mathcal{C}$, to denote the clock assignment which maps each clock in r to 0 and agrees with u for the other clocks (i.e. $\mathcal{C}\backslash r$). Now we are ready to present the operational semantics for extended timed automata by transition rules:

Definition 2. *Given a scheduling strategy* Sch*, the semantics of an automaton* $\langle N, l_0, E, I, M\rangle$ *with initial state* (l_0,u_0,q_0) *is a labelled transition system defined by the following rules:*

- $(l,u,q) \xrightarrow{a}_{\mathsf{Sch}} (m, u[r\mapsto 0], \mathsf{Sch}(M(m) :: q))$ *if* $l \xrightarrow{gar} m$ *and* $u \models g$
- $(l,u,q) \xrightarrow{t}_{\mathsf{Sch}} (l, u+t, \mathsf{Run}(q,t))$ *if* $(u+t) \models I(l)$

where $M(m) :: q$ *denotes the queue with* $M(m)$ *inserted in* q*.*

Note that the transition rules are parameterized by Sch (scheduling strategy). and Run (function representing the available computing resources). According to the transition rules, the task queue is growing with action transitions and

[3] A non-preemptive strategy will never change the position of the first element of a queue and a preemtive strategy may change the ordering of task types only, but never change the ordering of task instances of the same type.

[4] The semantics may be extended to multi-processor setting by modifying the function Run according the number of processors available.

shrinking with delay transitions. Multiple copies (instances) of the same task type may appear in the queue.

Whenever it is understood from the context, we shall omit Sch from the transition relation. Consider the automaton in Figure 2(b). Assume that preemptive earliest deadline first (EDF) is used to schedule the task queue. Then the automaton with initial state $(m_1, [x = 0], [Q_1(1, 2)])$ may demonstrate the following sequence of typical transitions:

$$
\begin{aligned}
&(m_1, [x = 0], [Q_1(1, 2)]) \\
&\xrightarrow{1} (m_1, [x = 1], [Q_1(0, 1)]) = (m_1, [x = 1], []) \\
&\xrightarrow{9.5} (m_1, [x = 10.5], []) \\
&\xrightarrow{a_1} (n_1, [x = 0], [P_1(4, 20)]) \\
&\xrightarrow{0.5} (n_1, [x = 0.5], [P_1(3.5, 19.5)]) \\
&\xrightarrow{b2} (m_2, [x = 0.5], [Q_2(1, 4), P_1(3.5, 19.5)]) \\
&\xrightarrow{0.3} (m_2, [x = 0.8], [Q_2(0.7, 3.7), P_1(3.5, 19.2)]) \\
&\xrightarrow{a_2} (n_2, [x = 0], [Q_2(0.7, 3.7), P_2(2, 10), P_1(3.5, 19.2)]) \\
&\xrightarrow{b_1} (m_1, [x = 0], [Q_2(0.7, 3.7), Q_1(1, 2), P_2(2, 10), P_1(3.5, 19.2)]) \\
&\xrightarrow{10} (n_1, [x = 10], []) \\
&\ldots
\end{aligned}
$$

This is only a partial behaviour of the automaton. A question of interest is whether it can perform a sequence of transitions leading to a state where the task queue is non-schedulable.

3 Schedulability Analysis

In this section we study verification problems related to the model presented in the previous section. First, we have the same notion of reachability as for timed automata.

Definition 3. *We shall write* $(l, u, q) \longrightarrow (l', u', q')$ *if* $(l, u, q) \xrightarrow{a} (l', u', q')$ *for an action* a *or* $(l, u, q) \xrightarrow{t} (l', u', q')$ *for a delay* t*. For an automaton with initial state* (l_0, u_0, q_0)*,* (l, u, q) *is reachable iff* $(l_0, u_0, q_0) \longrightarrow^* (l, u, q)$*.*

In general, the task queue is unbounded though the constraints of a givn automaton may restrict the possibility of reaching states with infinitely many different task queues. This makes the analysis of automata more difficult. However, for certain analysis, e.g. verification of safety properties that are not related to the task queue, we may only be interested in the reachability of locations. A nice property of our extension is that the location reachability problem can be checked by the same technique as for timed automata [14, 19]. So we may view the original timed automaton (without task assignment) as an abstraction of its extended version preserving location reachability. The existing model checking tools such as [20, 17] can be applied directly to verify the abstract models.

But if properties related to the task queue are of interests, we need to develop new verification techniques. One of the most interesting properties of extended automata related to the task queue is schedulability.

Definition 4. *(Schedulability) A state* (l, u, q) *with* $q = [P_1(c_1, d_1) \dots P_n(c_n, d_n)]$ *is a failure denoted* (l, u, Error) *if there exists* i *such that* $c_i \geq 0$ *and* $d_i < 0$*, that is, a task failed in meeting its deadline. Naturally an automaton* A *with initial state* (l_0, u_0, q_0) *is non-schedulable with* Sch *iff* $(l_0, u_0, q_0)(\longrightarrow_{\mathsf{Sch}})^*(l, u, \mathsf{Error})$ *for some* l *and* u*. Otherwise, we say that* A *is schedulable with* Sch*. More generally, we say that* A *is schedulable iff there exists a scheduling strategy* Sch *with which* A *is schedulable.*

The schedulability of a state may be checked by the standard test. We say that (l, u, q) is schedulable with Sch if $\mathsf{Sch}(q) = [P_1(c_1, d_1) \dots P_n(c_n, d_n)]$ and $(\sum_{i \leq k} c_i) \leq d_k$ for all $k \leq n$. Alternatively, an automaton is schedulable with Sch if all its reachable states are schedulable with Sch.

Checking schedulability of a state is a trivial task according to the definition. But checking the relative schedulability of an automaton with respects to a given scheduling strategy is not easy, and checking the general schedulability (equivalent to finding a scheduling strategy to schedule the automaton) is even more difficult.

Fortunately the queues of all schedulable states of an automaton are bounded. First note that a task instance that has been started can not be preempted by another instance of the same task type. This means that there is only one instance of each task type in the queue whose computing time can be a real number and it can be arbitrarily small. Thus the number of instances of each task type $P \in \mathcal{P}$, in a schedulable queue is bounded by $\lceil D(P)/C(P) \rceil$ and the size of schedulable queues is bounded by $\sum_{P \in \mathcal{P}} \lceil D(P)/C(P) \rceil$.

We will code schedulability checking problems as reachability problems. First, we consider the case of non-preemptive scheduling to introduce the problems. We have the following positive result.

Theorem 1. *The problem of checking schedulability relative to non-preemptive scheduling strategy for extended timed automata is decidable.*

Proof. A detailed proof is given in [11]. We sketch the proof idea here. It is to code the given scheduling strategy as a timed automaton (called the scheduler) denoted $E(\mathsf{Sch})$ which uses clocks to remember computing times and relative deadlines for released tasks. The scheduler automaton is constructed as follows: Whenever a task instance P_i is released by an event $\mathsf{release}_i$, a clock d_i is reset to 0. Whenever a task is started to run, a clock c is reset to 0. Whenever the constraint $d_i = 0$ is satisfied, and P_i is not running, an error-state (non-schedulable) should be reached. We also need to transform the original automaton A to $E(A)$ to synchronize with the scheduler that P_i is released whenever a location, say l to which P_i is associated, is reached. This is done simply by replacing actions labeled on transitions leading to l with $\mathsf{release}_i$. Finally we construct the product

automaton $E(\mathsf{Sch})||E(A)$ in which both $E(\mathsf{Sch})$ and $E(A)$ can only synchronize on identical action symbols namely $\mathsf{release}_i$'s. It can be proved that if an error-state of the product automaton is reachable, the original extended timed automaton is non-schedulable.

For *preemptive scheduling* strategies, it has been conjectured that the schedulability checking problem is undecidable. The reason is that if we use the same ideas as for non-preemptive scheduling to encode a preemptive scheduling strategy, we must use stop-watches (or integrators) to add up computing times for suspended tasks. It appears that the computation model behind preemptive scheduling is stop-watch automata for which it is known that the reachability problem is undecidable. Surprisingly this conjecture is wrong.

Theorem 2. *The problem of checking schedulability relative to a preemptive scheduling strategy for extended timed automata is decidable.*

The rest of this paper will be devoted to the proof of this theorem. It follows from Lemma 3, 4, and 5 established in the following section. Before we go further, we state a more general result that follows from the above theorem.

Theorem 3. *The problem of checking schedulability for extended timed automata is decidable.*

From scheduling theory [8], we know that the preemptive version of Earliest Deadline First scheduling (EDF) is optimal in the sense that if a task queue is non-schedulable with EDF, it can not be schedulable with any other scheduling strategy (preemptive or non-preemptive). Thus, the general schedulability checking problem is equivalent to the relative schedulability checking with respects to EDF.

4 Decidability and Proofs

We shall encode the schedulability checking problem as a reachability problem. For the case of non-preemptive scheduling, the expressive power of timed automata is enough. For preemptive scheduling, we need a more expressive model.

4.1 Timed Automata with Subtraction

Definition 5. *A timed automaton with subtraction is a timed automaton in which clocks may be updated by subtraction in the form $x := x - C$ in addition to reset of the form $x := 0$, where C is a natural number.*

This is the so called updatable automata [7]. It is known that the reachability problem for this class of automata is undecidable. However, for the following class of suspension automata, location reachability is decidable.

Definition 6. *(Bounded Timed Automata with Subtraction) A timed automaton is bounded iff for all its reachable states (l, u, q), there is a maximal constant C_x for each clock x such that*

1. $u(x) \geq 0$ *for all clocks* x*, i.e. clock values should not be negative and*
2. $u(x) \leq C_x$ *if* $l \xrightarrow{gar} l'$ *for some* l' *and* C *such that* $g(u)$ *and* $(x := x - C) \in r$.

In general, it may be difficult to compute the maximal constants from the syntax of an automaton. But we shall see that we can compute the constants for our encoding of scheduling problems.

Because subtractions on clocks are performed only within a bounded area, the region equivalence is preserved by the operation. We adopt the standard definition due to Alur and Dill [5].

Definition 7. *(Region Equivalence denoted* $\sim$*) For a clock* $x \in \mathcal{C}$*, let* C_x *be a constant (the ceiling of clock* x*). For a real number* t*, let* $\{t\}$ *denote the fractional part of* t*, and* $\lfloor t \rfloor$ *denote its integer part. For clock assignments* $u, v \in \mathcal{V}$*,* u, v *are region-equivalent denote* $u \sim v$ *iff*

1. *for each clock* x*, either* $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ *or* $u(x) > C_x$ *and* $v(x) > C_x$*), and*
2. *for all clocks* x, y *if* $u(x) \leq C_x$ *and* $u(y) \leq C_y$ *then*
   (a) $(\{u(x)\} = 0$ *iff* $\{v(x)\} = 0$ *and*
   (b) $\{u(x)\} \leq \{u(y)\}$ *iff* $\{v(x)\} \leq \{v(y)\}$

It is known that region equivalence is preserved by the delay (addition) and reset. In the following, we establish that region equivalence is also preserved by subtraction for clocks that are bounded as defined in Definition 6. For a clock assignment u, let $u(x - C)$ denote the assignment: $u(x - C)(x) = u(x) - C$ and $u(x - C)(y) = u(y)$ for $y \neq x$.

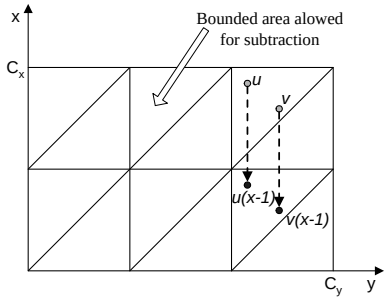


Fig. 3. Region equivalence preserved by subtraction when clocks are bounded.

Lemma 1. *Let* $u, v \in \mathcal{V}$*. Then* $u \sim v$ *implies*

1. $u + t \sim v + t$ *for a positive real number* t*, and*
2. $u[x \mapsto 0] \sim v[x \mapsto 0]$ *for a clock* x *and*
3. $u(x - C) \sim v(x - C)$ *for all natural numbers* C *such that* $C \leq u(x) \leq C_x$.

Proof. It is given in the full version of this paper [13].

In fact, region equivalence over clock assignments induces a bisimulation over reachable states of automata, which can be used to partition the whole state space as a finite number of equivalence classes.

Lemma 2. *Assume a bounded timed automaton with subtraction, a location l and clock assignments u and v. Then $u \sim v$ implies that*

1. *whenever* $(l,u) \longrightarrow (l',u')$ *then* $(l,v) \longrightarrow (l',v')$ *for some* v' *s.t.* $u' \sim v'$ *and*
2. *whenever* $(l,v) \longrightarrow (l',v')$ *then* $(l,u) \longrightarrow (l',u')$ *for some* u' *s.t.* $u' \sim v'$.

Proof. It follows from Lemma 1.

The above lemma essentially states that if $u \sim v$ then (l,u) and (l,v) are bisimular, which implies the following result.

Lemma 3. *The location reachability problem for bounded timed automata with subtraction, whose clocks are bounded with known maximal constants is decidable.*

Proof. Because each clock of the automaton is bounded by a maximal constant, it follows from lemma 2 that for each location l, there is a finite number of equivalence classes of states which are equivalent in the sense that they will reach the same equivalence classes of states. Because the number of locations of an automaton is finite, the whole state space of an automaton can be partitioned into finite number of such equivalence classes.

4.2 Encoding of Schedulability as Reachability

Assume an automaton A extended with tasks, and a *preemptive scheduling* strategy Sch. The aim is to check if A is schedulable with Sch. As for the case of *non-preemptive scheduling* (Theorem 1), we construct $E(A)$ and $E(\mathsf{Sch})$, and check a pre-defined error-state in the product automaton of the two. The construction is illustrated in figure 4.

$E(A)$ is constructed as a timed automaton which is exactly the same as for the non-preemptive case (Theorem 1) and $E(\mathsf{Sch})$ will be constructed as a timed automaton with subtraction.

We introduce some notation. Let $C(i)$ and $D(i)$ stand for the worst case execution time and relative deadline respectively for each task type P_i. We use P_{ij} to denote the jth instance of task type P_i.

For each task instance P_{ij}, we have the following state variables: $\mathsf{status}(i,j)$ initialized to free. Let $\mathsf{status}(i,j) = \mathsf{running}$ stand for that P_{ij} is executing on the processor, $\mathsf{preempted}$ for that P_{ij} is started but not running, and $\mathsf{released}$ for that P_{ij} is released, but not started yet. We use $\mathsf{status}(i,j) = \mathsf{free}$ to denote that P_{ij} is not released yet or position (i,j) of the task queue is free.

According to the definition of scheduling strategy, for all i, there should be only one j such that $\mathsf{status}(i,j) = \mathsf{preempted}$ (only one instance of the same task type is started), and for all i,j, there should be only one pair (k,l) such that $\mathsf{status}(k,l) = \mathsf{running}$ (only one is running for a one-processor system).

We need two clocks for each task instance:

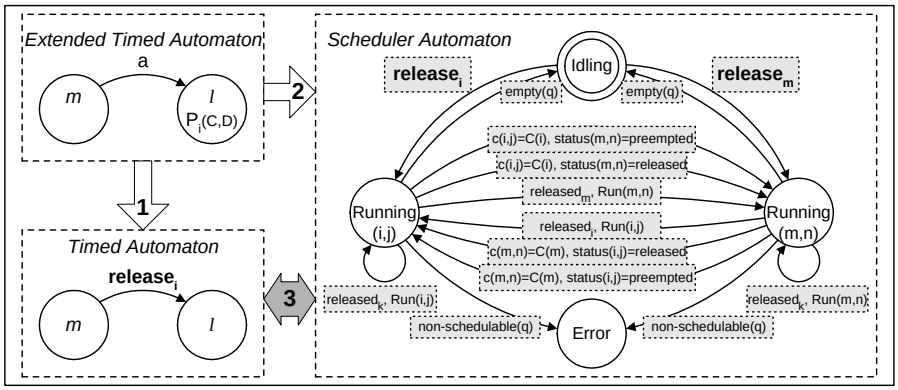


Fig. 4. Encoding of schelulability problem.

1. $c(i,j)$ (a computing clock) is used to remember the accumulated computing time since P_{ij} was started (when $\mathsf{Run}(i,j)$ became true) [5], and subtracted with $C(k)$ when the running task, say P_{kl}, is finished if it was preempted after it was started.
2. $d(i,j)$ (a deadline clock) is used to remember the deadline and reset to 0 when P_{ij} is released.

We use a triple $\langle c(i,j), d(i,j), \mathsf{status}(i,j)\rangle$ to represent each task instance, and the task queue will contain such triples. We use q to denote the task queue. Note that the maximal number of instances of P_i appearing in a schedulable queue is $\lceil D(i)/C(i)\rceil$. We have a bound on the size of queue as claimed earlier, which is $\sum_{P_i\in\mathcal{P}} \lceil D(i)/C(i)\rceil$. We shall say that queue is empty denoted $\mathsf{empty}(q)$ if $\mathsf{status}(i,j) = \mathsf{free}$ for all i,j.

For a given scheduling strategy Sch, we use the predicate $\mathsf{Run}(m,n)$ to denote that task instance P_{mn} is scheduled to run according to Sch. For a given Sch, it can be coded as a constraint over the state variables. For example, for EDF, $\mathsf{Run}(m,n)$ is the conjunction of the following constraints:

1. $d(k,l) \le D(k)$ for all k,l such that $\mathsf{status}(k,l) \neq \mathsf{free}$: no deadline is violated yet
2. $\mathsf{status}(m,n) \neq \mathsf{free}$: P_{mn} is released or preempted
3. $D(m) - d(m,n) \le D(i) - d(i,j)$ for all (i,j): P_{mn} has the shortest deadline

$E(\mathsf{Sch})$ contains three type of locations: Idling, Running and Error with Running being parameterized with (i,j) representing the running task instance.

1. Idling denotes that the task queue is empty.

[5] In fact, for each task type, we need only one clock for computing time because only one instance of the same task type may be started.

2. Running(i,j) denotes that task instance P_{ij} is running, that is, status(i,j) = running. We have an invariant for each Running(i,j): $c(i,j) \leq C(i)$ and $d(i,j) \leq D(i)$.
3. Error denotes that the task queues are non-schedulable with Sch.

There are five types of edges labeled as follows:

1. Idling to Running(i,j): there is an edge labeled by
   - guard: none
   - action: release$_i$
   - reset: $c(i,j) := 0$, $d(i,j) := 0$, and status(i,j) := running
2. Running(i,j) to Idling: there is only one edge labeled with
   - guard: empty(q) that is, status(i,j) = free for all i,j (all positions are free).
   - action: none
   - reset: none
3. Running(i,j) to Running(m,n): there are two types of edges.
   (a) The running task P_{ij} is finished, and P_{mn} is scheduled to run by Run(m,n). There are two cases:
      i. P_{mn} was preempted earlier:
         - guard: $c(i,j) = C(i)$, status(m,n) = preempted and Run(m,n)
         - action: none
         - reset: status(i,j) := free, $\{c(k,l) := c(k,l) - C(i) | \text{status}(k,l) = \text{preempted}\}$, and status$(m,n)$:= running
      ii. P_{mn} was released, but never preempted (not started yet):
         - guard: $c(i,j) = C(i)$, status(m,n) = released and Run(m,n)
         - action: none
         - reset: status(i,j) := free, $\{c(k,l) := c(k,l) - C(i) | \text{status}(k,l) = \text{preempted}\}$, $c(m,n) := 0$, $d(m,n) := 0$ and status(m,n) := running
   (b) A new task P_{mn} is released, which preempts the running task P_{ij}:
      - guard: status(m,n) = free, and Run(m,n)
      - action: released$_m$
      - reset: status(m,n) := running, $c(m,n) := 0$, $d(m,n) := 0$, and status(i,j) := preempted
4. Running(i,j) to Running(i,j). There is only one edge representing the case when a new task is released, and the running task P_{ij} will continue to run:
   - guard: status(k,l) = free, and Run(i,j)
   - action: released$_k$
   - reset: status(k,l) := released and $d(k,l) := 0$
5. Running(i,j) to Error: for each pair (k,l), there is an edge labeled by $d(k,l) > D(k)$ and status$(k,l) \neq$ free meaning that the task P_{kl} which is released (or preempted) fails in meeting its deadline.

The third step of the encoding is to construct the product automaton $E(Sch) \parallel E(A)$ in which both $E(Sch)$ and $E(A)$ can only synchronize on identical action symbols. Now we show that the product automaton is bounded.

Lemma 4. *All clocks of* $E(\mathsf{Sch})$ *in* $E(Sch) \parallel E(A)$ *are bounded and non-negative.*

Proof. It is given in the full version of this paper [13].

Now we have the correctness lemma for our encoding. Assume, without losing generality, that the initial task queue of an automaton is empty.

Lemma 5. *Let* A *be an extended timed automaton and* Sch *a scheduling strategy. Assume that* (l_0, u_0, q_0) *and* $(\langle l_0, \mathsf{Idling}\rangle, u_0)$ *are the initial states of* A *and the product automaton* $E(A)||E(\mathsf{Sch})$ *respectively where* l_0 *is the initial location of* A, u_0 *and* v_0 *are clock assignments assigning all clocks with* 0 *and* q_0 *is the empty task queue. Then for all* l *and* u*:*

$(l_0, u_0, q_0) \longrightarrow^* (l, u, \mathsf{Error})$ *iff* $(\langle l_0, \mathsf{Idling}\rangle, u_0 \cup v_0) \longrightarrow^* (\langle l, \mathsf{Error}\rangle, u \cup v)$*for some* v

Proof. It is by induction on the length of transition sequence.

The above lemma states that the schedulability analysis problem can be solved by reachability analysis for timed automata extended with subtraction. From Lemma 4, we know that $E(\mathsf{Sch})$ is bounded. Because the reachability problem is decidable due to Lemma 3, we complete the proof for our main result stated in Theorem 2.

5 Conclusions and Related Work

We have studied a model of timed systems, which unifies timed automata with the classic task models from scheduling theory. The model can be used to specify resource requirements and hard time constraints on computations, in addition to features offered by timed automata. It is general and expressive enough to describe concurrency and synchronization, and tasks which may be periodic, sporadic, preemptive and (or) non-preemptive. The classic notion of schedulability is naturally extended to automata model.

Our main technical contribution is the proof that the schedulability checking problem is decidable. The problem has been suspected to be undecidable due to the nature of preemptive scheduling. To our knowledge, this is the first decidability result for preemptive scheduling in dense-time models. Based the proof, we have developed a symbolic schedulability checking algorithm using the DBM techniques extended with a subtraction operation. It has been implemented in a prototype tool [6]. We believe that our work is one step forward to bridge scheduling theory and automata-theoretic approaches to system modeling and analysis. A challenge is to make the results an applicable technique combined with classic methods such as rate monotonic scheduling. We need new algorithms and data structures to represent and manipulate the dynamic task queue consisting of time and resource constraints. As another direction of future work, we shall study the schedule synthesis problem. More precisely given an automaton, it is desirable to characterize the set of schedulable traces accepted by the automaton.

Related work. Scheduling is a well-established area. Various analysis methods have been published in the literature. For systems restricted to periodic tasks, algorithms such as rate monotonic scheduling are widely used and efficient methods for schedulability checking exist, see e.g. [8]. These techniques can be used to handle non-periodic tasks. The standard way is to consider non-periodic tasks as periodic using the estimated *minimal* inter-arrival times as *task periods*. Clearly, the analysis based on such a task model would be pessimistic in many cases, e.g. a task set which is schedulable may be considered as non-schedulable as the inter-arrival times of the tasks may vary over time, that are not necessary minimal. Our work is more related to work on timed systems and scheduling.

A nice work on relating classic scheduling theory to timed systems is the controller synthesis approach [2, 3]. The idea is to achieve schedulability by construction. A general framework to characterize scheduling contstraints as invariants and synthesize scheduled systems by decomposition of constraints is presented in [3]. However, algorithmic aspects are not discussed in these work. Timed automata has been used to solve non-preemptive scheduling problems mainly for job-shop scheduling[1, 12, 15]. These techniques specify pre-defined locations of an automaton as goals to achieve by scheduling and use reachability analysis to construct traces leading to the goal locations. The traces are used as schedules. There have been several work e.g. [18, 10, 9] on using stop-watch automata to model preemptive scheduling problems. As the reachability analysis problem for stop-watch automata is undecidable in general [4], there is no guarantee for termination for the analysis without the assumption that task preemptions occur only at integer points. The idea of subtractions on timers with integers, was first proposed by McManis and Varaiya in [18]. In general, the class of timed automata with subtractions is undecidable, which is shown in [7]. In this paper, we have identified a decidable class of updatable automata, which is precisely what we need to solve scheduling problems without assuming that preemptions occur only at integer points.

Acknowledgement: Thanks to the anonymous referees for their insights and constructive comments.

References

1. Y. Abdeddam and O. Maler. Job-shop scheduling using timed automata. In *Proceedings of 13th Conference on Computer Aided Verification, July 18-23, 2001 Paris, France*, 2001.
2. K. Altisen, G. Gößler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proceedings of the 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, 1-3 December, 1999*, pages 154–163. IEEE Computer Society Press, 1999.
3. K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In *Proceedings of FTRTFT 2000, Pune, India, September 2000, LNCS 1926, pp.106-120*, 2000.

4. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
5. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
6. T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES - a tool for modelling and implementation of embedded systems. In *Proceedings of TACAS02*. Springer–Verlag, 2002.
7. P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Are timed automata updatable? In *Proceedings of the 12th International Conference on Computer-Aided Verification*, Chicago, IL, USA, July 15-19, 2000, 2000. Springer-Verlag.
8. G. C. Buttazzo. *Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications*. Kulwer Academic Publishers, 1997.
9. F. Cassez and F. Laroussinie. Model-checking for hybrid systems by quotienting and constraints solving. In *Proceedings of the 12th International Conference on Computer-Aided Verification*, pages 373–388, Standford, California, USA, 2000. Springer-Verlag.
10. J. Corbett. Modeling and analysis of real-time ada tasking programs. In *Proceedings of 15th IEEE Real-Time Systems Symposium, San Juan, Puerto Rico, USA*, pages 132–141. IEEE Computer Society Press, 1994.
11. C. Ericsson, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, 1999.
12. A. Fehnker. Scheduling a steel plant with timed automata. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, 1999.
13. E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: Schedulability and decidability. Technical report, Department of Information Technology, Uppsala University, Sweden, 2002.
14. T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.
15. T. Hune, K. G. Larsen, and P. Pettersson. Guided Synthesis of Control Programs using UPPAAL. *Nordic Journal of Computing*, 8(1):43–64, 2001.
16. K. G. Larsen, P. P., and W. Yi. Compositional and symbolic model-checking of real-time systems. In *Proceedings of 16th IEEE Real-Time Systems Symposium, December 5-7, 1995 Pisa, Italy*, pages 76–89. IEEE Computer Society Press, 1995.
17. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
18. J. McManis and P. Varaiya. Suspension automata: a decidable class of hybrid automata. In *Proceedings of the 6th International Conference on Computer-Aided Verification*, pages 105–117, Standford, California, USA, 1994. Springer-Verlag.
19. W. Yi, P. Pettersson, and M. Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Proceedings of the 7th International Conference on Formal Description Techniques*, 1994.
20. Sergio Yovine. A Verification Tool for Real Time Systems. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

The Next Generation of UPPAAL

Alexandre David Wang Yi

April 2, 2002

Abstract

We present the design and internal data structures for the next generation of UPPAAL. Early experimental results demonstrate that the new implementation based on these structures improves the efficiency of UPPAAL by about 80% in both time and space. In addition, the new version is built to handle hierarchical models. The challenge in handling hierarchy comes from the very dynamic structure of hierarchical systems: the levels of concurrency and the scope of data variables and clocks are changing. We present data structures and a searching scheme for state space exploration of hierarchical models.

Memory Utilization in Software DSM for Embedded Systems

Nguyen-Thai Nguyen-Phan[1] and Mats Brorsson
Department of Microelectronics and Information Technology.
Royal Institution of Technology
Electrum 229, SE-164 40 Kista, Sweden
Email: thai@imit.kth.se, Mats.Brorsson@imit.kth.se
Fax: +46-8-751 1793

Abstract: Software Distributed Shared Memory (S-DSM) systems support parallel programming by implementing a shared memory on top of distributed system. It frees programmer from communication complexities to concentrate to parallel algorithms. However, there is a drawback: current S-DSM systems waist memory on all processors. Therefore it is hard to be implemented to embedded systems, which have small resources e.g. memory. In this project, we modified an implemented S-DSM system named CVM – Coherent Virtual Machine – to support an embedded system and investigate memory consumed, in order to implement an S-DSM system supports embedded systems. The results shows that with a good selection of initial values for an S-DSM system and carefully programmed, S-DSM system can be implemented for embedded system with less memory overhead and high speedup.

Keywords: Software distributed shared memory, S-DSM, embedded system, parallel programming**.**

I. INTRODUCTION

Even though processors for embedded computer systems are becoming more and more powerful there is still a significant performance gap to PC processors and the like [11]. At the same time, we ask that the embedded systems can perform more performance demanding tasks. One prominent example is image processing. In network attached cameras used for surveillance systems—such as the ones produced by Axis Communications [2]—it would save an enormous amount of network bandwidth if intelligent processing, for instance face recognition, could be done in the embedded systems serving the cameras and then only send the important data, e.g. the identity of the person recognized, over the network.

In order to achieve this we need more computational power than normally is present in embedded systems. However, we do not want to put more computational power in the system than needed and it is also desirable that increased performance can be added incrementally and in a modular fashion.

One relatively simple way to increase performance is to use parallelism. Several identical embedded platforms can be joined together to form nodes in a parallel platform that can be programmed as one parallel computer with distributed memory (DM). This has the advantage that we do not need to modify the existing platform to make use of parallelism. The interconnection medium between the individual nodes can be the standard network or preferably a dedicated high-performance network for inter-node communication.

A DM parallel platform is normally programmed with a message-passing programming mode such as MPI [3]. However, it is widely recognized that a shared address space programming model is to prefer since they present a more natural model to the programmer. Unfortunately, these models normally require substantial hardware support. In order to build cost-effective platforms for shared memory programming, S-DSM systems have been developed, e.g. TreadMarks [1] and CVM [7]. In short, an S-DSM system provides the application program with the illusion of a shared memory on a collection of nodes with distributed memory.

In this paper, we describe the adaptation of a well-known S-DSM system, CVM [7], to be used in an embedded computer system environment based on platforms from Axis Communications, and an analysis of the memory usage for a few parallel applications. As far as we are aware, this is the first time a S-DSM system has been used in embedded platforms where the available memory is scarce and it is therefore important to study the memory usage in particular.

Our conclusion of this work is that it is indeed possible to adapt CVM, which was developed for desktop and server type computers, to be used also in embedded systems. The memory usage analysis also shows that it is important to consider where and how much memory is allocated. For instance, through careful selection of the home location for some memory pages we are able to lower the total memory requirement leading to larger data sets for the applications.

In the rest of the paper we first present some background information about S-DSM systems in section II. Section III describes a version of S-DSM system and memory usage in initialization stage. In section IV, results from applying our system in the Axis developer boards are demonstrated and analyzed. Finally, our conclusion is given in the last section.

[1] Presenting author, if accepted

II. Software Distributed Shared Memory

A. Overview

A software distributed shared memory (S-DSM) system provides the illusion of coherent memory sharing for machines connected only by a message-passing network, or by a non-cache-coherent memory system. Traditional S-DSM system provides a conceptually appealing programming model for processes that have been spread across a locally distributed cluster for the purpose of parallel speed-up.

The idea of implementing an S-DSM system is to implement a shared memory layer on top of a message-passing layer. Many memory consistency protocols have been introduced and implemented to achieve this. It has been shown that the home-based lazy release consistency (HLRC) protocol is robust and that it provides good performance for many different types of applications [9]. Therefore, we have chosen HLRC as the memory consistency protocol for our system. In HLRC, as in most other S-DSM protocols, the shared memory is divided into pages with fixed size. However, specific to HLRC is that each page is assigned with a processor called *home* for that page. Memory synchronization takes place at barriers or locks, thus reducing communication and ping-pong effect. At a synchronization point, modifications of shared memory pages since the last synchronization are stored in a data structure, called *diff*, and are sent to the memory in the home node.

B. Shared address space protocols

In most S-DSM systems that run on Linux OS, accesses to the shared memory are handled by catching a page fault (SEGV) generated when a memory access is made to a page, which is read or write protected.

As introduced before, HLRC was selected as a protocol for our system. We have chosen to base our system on CVM, a flexible and well-documented S-DSM system supporting HLRC [7]. While the protocol is well described in publications [9], in this part we only concentrate on memory consumed in the protocol it self.

In HLRC, shared memory is divided into chunks of pages and each page has a home in one of the processor nodes. The home node keeps track of changes in the content of the page and a list of other nodes, which has a copy of that page. Each non-home node must ask the home node before accessing the page. Since all processors may request any page of the shared memory for reading or writing, they should keep list of properties of all pages, e.g. home address, its current version etc.

When a node—which is not the home—performs the first read access, it will experience a page fault (SEGV) and the S-DSM system will request the page, through messages, from the home. When a write fault occurs, if the node does not already have a copy of the page, it sends a request to the home for a copy of the page. Then it creates another copy of the page, called *twin*, in order to keep track of the modification it will make to the page that later will be communicated to the home node. While the memory for the twin is dynamically allocated, the system also maintains a heap for holding *write notices* and *diffs*. A write notice is a message from the home with information about which pages have been modified and when. This provides information to the HLRC protocol about when it needs to request new information from the home node. A diff is an encoding of the modifications made to a page by that node created from the modified page and the twin.

Because of the different nature of these data structures, the heap is in CVM separated into two heaps: the home-heap for holding write notices and the diff-heap for creating diffs.

The execution of an application is divided into *intervals*. An interval starts at the acquisition of a lock and finishes at the releasing point. A barrier synchronization is from a memory consistency point of view semantically equivalent to a release immediately followed by an acquire-operation. HLRC is a multiple-writer protocol, which means that several nodes may modify the same page simultaneously in order to avoid performance problems. Therefore, new intervals only start at a barrier or a lock request. Barriers and locks are thus the only two synchronization points in this protocol. While barriers synchronize all modified pages to their home, locks only synchronize pages with the owner of the lock at that time. Therefore, like pages, locks also need memory for the lock manager and the next node requesting the lock.

III. Adapting an S-DSM for embedded systems

For this study, we have built a small cluster system with 4 Etrax developer boards, connected via 100 Mbits/s Ethernet. The memory system for the Etrax board consists of 2 Mbytes Flash ROM and 8 Mbytes RAM. The OS (Linux) is stored in compressed format and saved on the Flash ROM and is uncompressed and installed in RAM during the boot process. When running, a maximum of 3 MByte of RAM is left available for applications on each board. We used the Linux 2.4.5 kernel version and gcc-cris[2] compiler version 2.96.

Etrax itself is a custom-made processor used for communication-oriented applications. The version we are using is clocked with 100 MHz and has a unified 8 KBytes cache memory.

CVM is a research tool for S-DSM systems and contains many different consistency protocol variations. In order to adapt it to an embedded environment we first removed all different protocols but HLRC, modified architecture dependent points to be suitable for the Etrax architecture. By doing that, the CVM library size was reduced from 2 MByte to 0.2 MByte. We also implemented a very simple rsh server and client since these services are not present in the developer board version of Linux. Additionally, we inserted more tracing information in order to track memory usage in applications. The result is a lightweight S-DSM system for the Etrax developer boards that we call *eCVM* for *embedded* CVM.

[2] gcc-cris is a port of gcc for the Etrax processor.

A. Application run-time phases

The runtime of a parallel application in this system can be divided into 3 phases: *initialize*, *run*, *finalize*.

At beginning, an execution command is sent to the OS in the node 0, the master node. The OS loads the executable file to main memory, initializes the environment and points the program counter to the starting address. Then the program starts executing and goes to initialize state. This very first process is called *parent*. The parent executes appropriate instructions to invoke the same application on other machines. After this point, all processors go to the initialize state; execute exactly the same instructions, like the parent. Therefore, they are called parent simulation.

In this step, the shared address space is created using virtual memory mapping; the memory fault handler is established; the communication mechanism is generated. Then, shared memory variables location is located and initialized. Since all processors are running exactly the same code, and eCVM is running in a homogeneous system, all parallel processes have the same status, including the memory variables' virtual addresses and values, except for information on how to recognize the process itself in the system. Finally, connections to others processes are established.

After the initialization, all processes go to running phase and really execute in parallel. Memory consistency is automatically handled by the system.

In the finalize phase, statistics information is collected; memory is released and the application is finished. However, in CVM, and most of the other system, the memory releasing is skipped. The job is passed to OS do the garbage collection.

B. The effect of system parameters on memory usage

There are four parameters specified in an S-DSM system—using the HLRC protocol—that affect the memory resource. They are:

- The maximum number of shared memory pages that the system can manage: MAX_PAGES.
- The number of pages which the system pre-allocates to use each time: NUM_PAGES
- The maximum number of processors the system can use: MAX_PROCS
- The maximum number of locks to be used: MAX_LOCKS

We used two different sets of parameter to investigate the memory usage according to Table 1.

Table 1. Different sets of system parameters used.

	Set 1	Set 2
MAX_PAGES	8192	512
NUM_PAGES	200	200
MAX_PROCS	32	32
MAX_LOCKS	4110	500

The first set takes common parameters using in standard CVM system. The second set was taken considering the maximum free memory that can be used in our system.

As shown in Table 2, CVM takes less than 1MB of memory for management purpose, which is very small for workstations because they have large memory and virtual memory. However, in embedded system like ours, it could be over 10% of the total memory resource and is about 30% of maximum free memory for an application.

Table 2. Memory needed at initialization of the system.

Memory for	Size (KBytes)		Affected by parameters
	Set 1	Set 2	
Page management	49.6	49.6	NUM_PAGES
Page copy sets	32.0	2.0	MAX_PAGES
Home management	80.3	12.8	MAX_PROCS, MAX_PAGES
Locks	722.5	87.9	MAX_LOCKS
Total	**884.4**	**152.3**	

We observed that the maximum problem size decreases when the number of processors increases (see Table 3) and it is independent on number of iterations in iterative algorithms. A reason for this is that the main process, process id 0, uses a system command to invoke a shell to execute rsh. It thus allocates memory to initialize the environment and does not return it when done. Besides that, resources for holding connections and preparing the exchange of messages increase when the number of nodes increases.

Table 3. Maximum problem size vs. number of processors for two different versions of SOR.

Number of nodes	1	2	3	4
SOR	590x590	540x540	520x520	520x520
ID_SOR[3]	590x590	775x775	885x885	1000x1000

IV. APPLICATION STUDIES

A. Speedup

Lacking good parallel applications for high-performance embedded systems, we have used five applications from the CVM distribution to evaluate our new system [7]. They are SOR, Water, FFT, QS and TSP. The programs originally come from the SPLASH benchmark suite and have been described in detail elsewhere so we do not explain the algorithms here [6].

Because of the limited memory resource on the developer boards, the applications can only run with relatively small data sets as given by Table 4. Most of applications got speedup of about 3.5 or higher with four processors. The performance speedup will be discussed in more detail in the next section.

[3] ID_SOR is a modified version of SOR, in which the shared memory is initialized by it' home only.

Table 4. Shared memory usage and execution time of applications

Application	Shared memory used (pages)	Execution time for 1 processor (seconds)
SOR (500x500x5)	247	82.5
Water (125 molecules)	11	1245.1
FFT (16x16x16)	26	49.5
QS (150000 elements)	386	27.8
TSP (19 cities)	99	735.6

Figure 1 shows the relative speedup of our applications. All applications except QS exhibit a relatively good speedup. One reason for this good speedup is that the Etrax processor does not have hardware support for floating point arithmetic. Therefore, even with a small problem size, the computation-to-communication ratio (CCR) is still quite large; the shared memory exchange between processors is small (see Table 4) causing low overhead from the underlying system. This is confirmed in QS, which was running with large problem size but the CCR is low and the shared memory is accessed heavily.

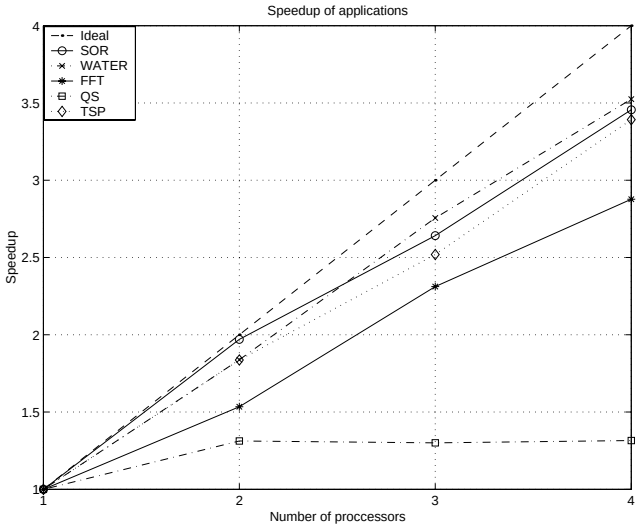


Figure 1. Speedup of benchmark applications.

B. FFT

Figure 2 shows a typical diagram of the memory usage for the four nodes in eCVM after the initialization stage. From now on, with *memory* without further explanation we mean the local memory used in the system only. The shared memory space used in the application is not shown in the figures.

The four diagrams in the figure show the amount of memory allocated in eCVM for the four different processors. We can see that node 0 always uses 32 KByte more memory than the other does after the initialization stage (time 0). The graphs for nodes 1-3 are show a characteristic with spikes of high memory usage. A spike indicates a synchronization point—i.e., a new consistency protocol interval—, in which memory is allocated for creating diffs that are sent to home. Then that memory and the corresponding twins are deallocated. The increase of memory usage between spikes shows that a write fault on a remote page occurred and a twin was made. There is also a slight steady increase of allocated memory after each synchronization point. This small amount of memory is used to store the time stamp of pages (the version of page) to keep track of how pages are modified. However, we would only need to keep the last time stamp but since this memory leakage is quite small, we have chosen to ignore it for now.

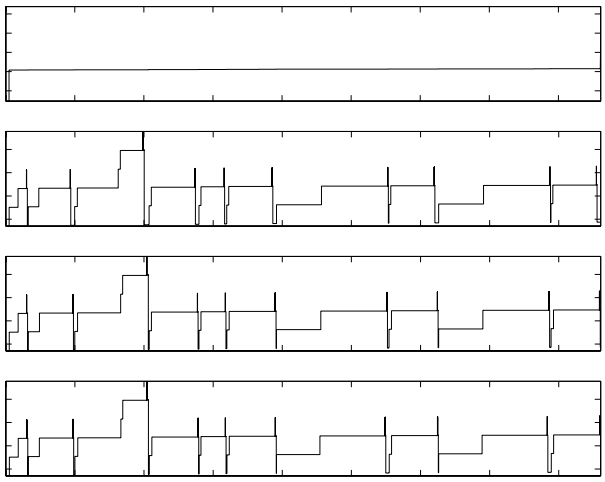


Figure 2. Memory usage vs. time in FFT (16x16x16)

C. SOR

In SOR, the memory used for matrix elements is allocated in shared memory. However, communication only occurs across the boundary rows between bands and it is for read only.

Figure 3 shows that many twins and diffs are generated regularly when executing SOR. This is because the matrix is allocated continuously in the shared memory space; therefore, it may happen that a processor owns a row over a page boundary and the next page is assigned to another processor. It also may happen that a boundary row spans over two pages although its size was less than a page size. This causes more memory access faults than expected.

Although the overhead for creating twins and to generate diffs is low [10], it is simple to insert some pads before and after a boundary row to make sure that it spans over one or more entire pages. We observed that this effectively removed the overhead of creating twins and generating diffs. It also reduced the number of page faults.

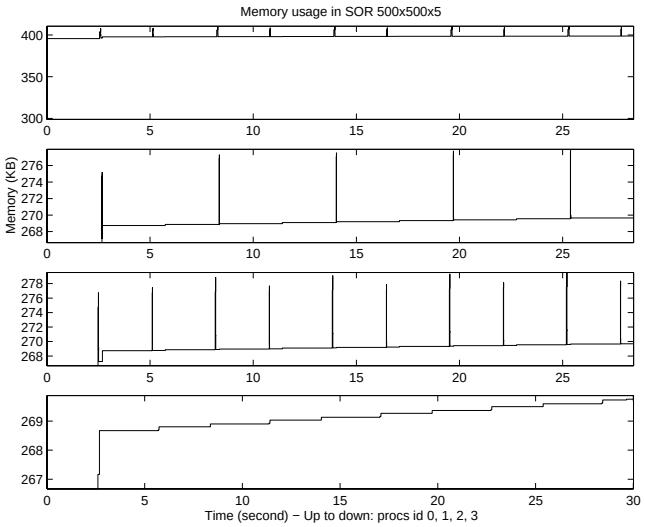


Figure 3. Memory usage in eCVM for SOR before inserting pads.

In both cases, with and without inserting pads, although the matrix was divided into balanced parts for each of processors, there was still load imbalance (see Figure 4). Processors 0 and 2 had to wait longer time than processors 1 and 3 at barriers.

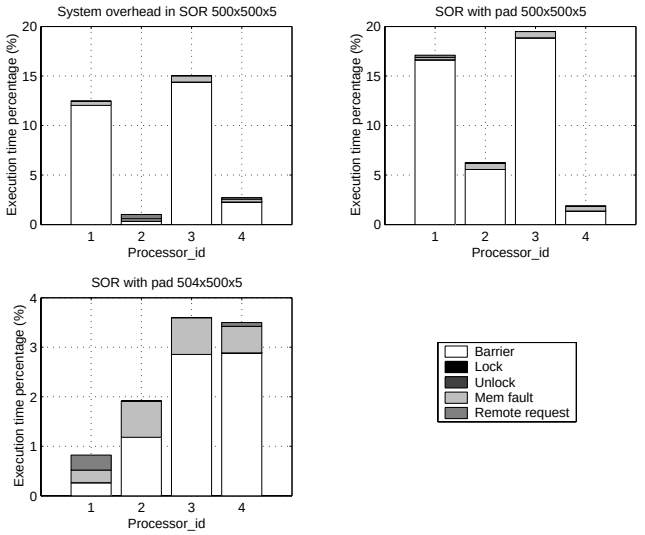


Figure 4. System overhead time specification in SOR.

In each iteration, the computation on each processor is divided into two parts separated by two barriers. The first part computes all black elements; the second computes all red elements. If the number of rows is not divisible with eight, when running on four processors, then processors 0 and 2 start with a black element and the processors 1 and 3 start with a red element. Although the total number of red and black elements assigned to each processor is equal, the number of red elements is different for processors 0 and 2 compared with 1 and 3, and so does the number of black elements. This caused a load imbalance between two parts on a processor (see Table 5). We call it local load imbalance.

Table 5. Average time spent on each part of each processor in one iteration.

	Processor time (ms)			
	0	1	2	3
500, black	2782	2841	2838	2829
500, red	2749	3398	2781	3223
504, black	2815	2828	2846	2797
504, red	2772	2783	2802	2786

By using a number of rows divisible by 8, e.g. m=504, this local load imbalance was isolated and we thus got a speedup of 3.9 on 4 processors instead of 3.5 (see Figure 5).

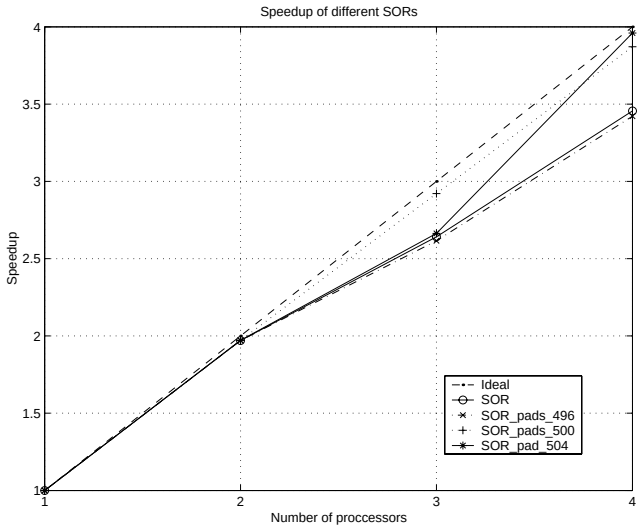


Figure 5. Speedup of a: SOR m = 500, b: SOR padded m=500, c: SOR padded m=504

D. QS and TSP

The quick sort (QS) and traveling salesman problem (TSP) algorithms are well described in [5]. They parallel versions use the same algorithms. However, recursion is replaced by a shared pool of jobs in which all processors can take out a job in turns and put new jobs. Load balancing is implied in this sort of algorithms, because the one, which served heavier jobs, was served less number of jobs than others. Locks are used to protect and synchronize the shared memory among processors instead of barriers.

When running on 4 processors, the overhead of S-DSM system took average of 61% of the execution time of QS and about 8% in that of TSP. We gained a speedup of 1.3 with 2 processors and the same speedup with 4 processors in QS. This can be explained by the workload of QS was small (see Table 4).

As shown in Figure 6, the memory used for making twins and diffs was as much as double the amount of shared memory allocated. Intuitively, one might think that distributing pages among processors may reduce the local memory usage because it will reduce the number of diffs and twins. However, this did not help. The overhead of the S-DSM system even increased. One reason is that although the number of diffs and twins was reduced, the number of exchanged messages was increased and the overhead of

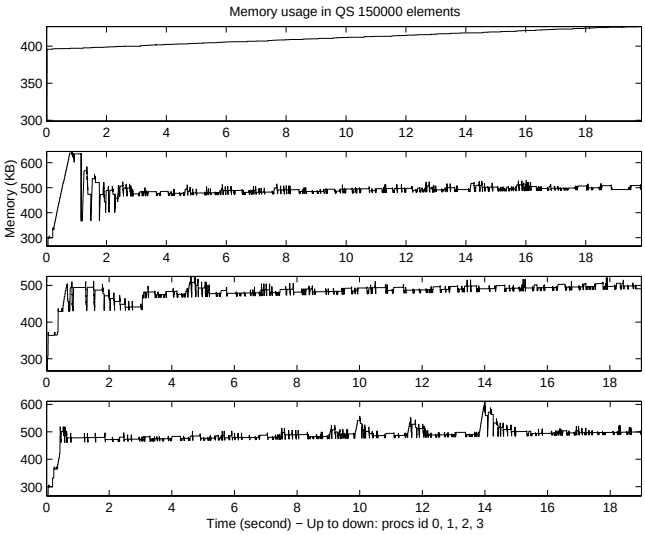


Figure 6. Memory usage in QS, 150000 elements

sending messages is well-known from other studies on S-DSM systems [8].

V. DISCUSSION AND CONCLUSION

One way of reducing the duplication of the initialization step is to implement a "fork-like" function. As we all know, fork creates a new process and copies the environment and parent page tables to create a unique task structure for the child. Then the "parent simulation" is unnecessary, the child processes will only access memory pages it really needs. It thus reduces the memory consumed in those processes for shared memory. However, with this solution, the parent process still suffers from the initialization step. It accesses all shared memory pages during the initialization; therefore lack of memory is still a problem.

A large number of shared pages requested in QS were because of its upper bound parameters:
MAX_SIZE = 768K elements which is about 384 pages. The amount of memory really used was 150000 elements, which corresponds about 74 pages. However, by a property of Linux, a memory page is only allocated at the first reference, therefore this large amount of pages did not affect the amount of shared memory. The only negative impact it had was the increase of local memory for managing pages. This cost 49 Kbytes extra. Thus, a carefully selection of the initial parameters for each architecture in an S-DSM system will reduce large amount of memory usage, especial in embedded systems.

The local memory used for making twins and generating diffs may as large as double the shared memory used. Distribution of home pages among all processors may reduce this memory but it may increase the memory used for producing messages at synchronization points. This was confirmed in figures of memory used in TSP and TSP_D (home pages distributed over processors).

For producer/consumer problems, like the boundary rows in SOR, dynamic mapping of shared memory will reduce a large amount of memory for creating twin and making diffs in the consumer. Clearly, every time a page fault occurs in the consumer, it fetches the page from home (producer) into its local memory only. Whenever the producer updates its data, it should announce that to the consumer. This is an adaptation of the invalidation protocol.

We have shown in this study that it is possible to achieve high speedup with a software distributed shared memory system in a cluster of embedded processing nodes. However, lack of memory resources is a major problem in S-DSM systems on such platforms. However, we have only studied with a maximum of four processors. A study with 16 or more processors for scalability would give better useful information.

Due to the difficulty of finding and parallelizing applications to run on embedded system, all benchmark applications were taken from scientific computing world. They are rarely run in this kind of systems. A study on classifications of common applications run on embedded system and their parallelized versions would be of interest.

ACKNOWLEDGMENT

This research has been funded by the Swedish Foundation of Strategic Research. Devices have been graciously provided by Axis Communications [2].

REFERENCES

[1] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel. *TreadMarks: Shared Memory Computing on Networks of Workstations*, IEEE Computer, Vol. 29, no. 2, pp. 18-28, February 1996.

[2] Axis Communications, http://www.axis.com

[3] J. W. Chung, B.H.Seong, K. H. Park, D. Park. Moving Home-Based Lazy Release Consistency for Shared Virtual Memory Systems, in *proceedings of the 1999 International Conference on Parallel Processing*, Septembet 1999.

[4] L. Clarke, I. Glendinning and R. Hempel, *The MPI Message Passing Interface Standard*, The MPI Forum, March 1994

[5] T.H.Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[6] A. Gupta, J. Hennessy, C. Holt. *Stanford Parallel Applications for Shared Memory*. http://www-flash.stanford.edu/apps/SPLASH/

[7] P. Keleher, *CVM: The Coherent Virtual Machine*, Tech. Report, Department of Computer Science, University of Maryland, July 1997.

[8] E. W. Parsons, M. Brorsson and K. C. Sevcik, *Predicting the Performance of Distributed Virtual Shared Memory Applications*, IBM Systems Journal, Volume 36, No. 4, 1997, pp. 527-549.

[9] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. *HomeBased SVM Protocols for SMP Clusters: Design and Performance*. In Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture, pages 113--124, February 1998.

[10] W. Shi, W. Hu, Z. Tang. *Where does the time goes in S-DSM systems: Experiences with JIAJIA*, Journal of Computer Science and Technology, Vol 14, No.3, pp.193-205, May 1999

[11] C. D. Snyder, *Faster & Wider Performance Parts*, Microprocessor report, 2/19/02-03.

Applying Priorities to Memory Allocation*

Sven Robertz
Department of Computer Science, Lund University
sven@cs.lth.se

ABSTRACT

In embedded systems, memory is a scarce resource and great attention must be given to memory management. A novel approach of applying priorities to memory allocation is presented and it is shown how this can be used to enhance the robustness of real-time applications. Focus is on systems with automatic memory management, but the ideas are also applicable to manually managed memory. In systems with automatic memory management, the proposed mechanisms can also be used to increase performance by limiting the amount of garbage collection work. Furthermore, a way of introducing priorities for memory allocation in a Java system without making any changes to the syntax of the Java language is proposed. This has been implemented in an experimental Java virtual machine.

**This is a shorter version of a paper to appear at the International Symposium on Memory Management (ISMM), Berlin, Germany, June 20-21, 2002.*

1. INTRODUCTION

With the recent development in small, cheap and fast processors for embedded systems and the emerging trend of writing embedded applications in high level object oriented languages, the performance limiting bottleneck may no longer be CPU time but rather memory and memory management. This is accentuated by the high relative cost of memory in embedded systems and systems on chip.

Memory management is a system-global problem and currently puts a great responsibility on programmers. For instance, a memory leak or excessive memory allocation in one module of a system will eventually cause the entire system to run out of memory and fail. Therefore it is interesting to study whether it is possible to apply priorities to memory as well as CPU time allocation; just as we don't want an important process to be delayed because a less important one is executing we don't want an unimportant memory allocation to cause a critical process to fail or be delayed, because the system runs out of memory or has to do a large amount of garbage collection work to satisfy its allocation needs.

We propose a novel approach which addresses two problems: firstly, how to increase program robustness by avoiding out-of-memory problems and secondly, to increase application performance in systems with automatic memory management by reducing the garbage collection (GC) workload. Section 3 briefly describes both aspects, whereas the rest of the paper will focus on the robustness issue.

While this paper focuses on object oriented systems with garbage collection, especially Java, the robustness issues should be equally applicable to any memory allocator.

A note on terminology; in order to avoid confusion we will use the terms *high priority* (HP) and *low priority* (LP) to denote the CPU time priority of a process and the terms *critical* and *non-critical* for our new notion of priorities for memory allocations.

2. BACKGROUND

It has been shown that it is possible to schedule GC work in such a way that high priority processes are not disturbed by using a technique called semi-concurrent garbage collection scheduling [3]. The fundamental idea of this technique is that since we don't want the high priority processes to be delayed by garbage collection, we suspend the garbage collector when they are executing. The GC work neglected during the execution of the high priority processes is then performed in the pauses between the activations of high priority processes. The remaining CPU time will be divided between executing low priority processes and performing GC work motivated by the actions of the LP processes, using traditional incremental techniques [5].

Basically, a system using this strategy can be described as having three levels of priority:

1. High priority processes
2. Garbage collection required to satisfy the high priority process
3. Low priority processes and traditional incremental garbage collection

Figure 1 shows how the CPU time will be used in a system

with one periodic high priority process and one low priority process.

Coupled with good worst case execution time and memory requirements estimates and a good garbage collection work metric, semi-concurrent garbage collection scheduling allows us to make hard real-time guarantees for the high-priority threads by using traditional schedulability analysis.

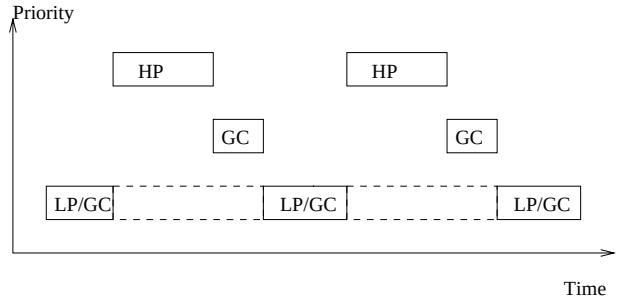


Figure 1: Dividing the CPU time between processes. The system consists of one periodic high priority process (*HP*) and one low priority process (*LP*). Whenever a high priority process is suspended, and no other HP process is eligible for execution, the garbage collector (*GC*) is run. GC work is also interleaved with the low priority process using traditional incremental garbage collection.

3. APPLYING PRIORITIES TO MEMORY ALLOCATIONS

We like to view memory allocation as any other resource allocation. Our goal is to provide run-time system support for doing the most important memory allocation if the system has limited memory in analogy with how the process scheduler makes sure that the most important process is run and less important ones are delayed if CPU time is scarce.

3.1 Avoiding out-of-memory situations

A high priority process in an embedded system may perform other tasks[1] in addition to its core functionality. For example, a digital controller process may produce log data in addition to calculating and outputting its control signal. In such a process, memory allocations by the less important tasks (e.g., producing log data) must never interfere with the core functionality (calculating the control signal).

This can, of course, be achieved by manually ensuring that the amount of log data never exceeds a certain value, e.g., by using a bounded buffer for delivering it to the logger process. Doing this manually has the drawback that the size of the buffer has to be calculated and this calculation is highly platform and application dependent. (I.e., each time a change that affects the application's memory allocation behaviour is made, the maximum amount of non-critical memory has to be recalculated.) If more than one process does unrelated non-critical memory allocations, the complexity of managing this increases rapidly. Thus, manual solutions require a lot of work and risk beeing unnecessarily conservative, error prone, or both.

Our approach to this problem is to transfer the responsibility for making the decisions about when to allow non-critical memory allocations from the programmer to the run time system. Then, the only a priori calculation that has to be done is to calculate the amount of critical allocations done by each (high priority) process during its period and this depends only on the application and not on any properties of the target platform.

This approach can also be used to provide a "limp home" mode, i.e., a mode of operation with lesser performance but radically lower memory consumption that will allow the application to continue executing in an out of memory situation, facilitating a more graceful degradation. This may be useful for adding some amount of predictability to applications with non-predictable memory allocation behaviour.

Finally, non-critical memory allocation gives programmers the possibility to add more features to a system without risking that these additions cause the system to run out of memory and jepoardise the core functionality of the system even if it is moved to a smaller platform. E.g., a low priority process with only non-critical memory allocations cannot cause a system to fail since, if the CPU load is dangerously high it will not get any CPU time and if the amount of memory is too low, it will not be allowed to allocate any memory.

This also has the advantage that it makes it easier to make hard real-time guarantees since worst case and schedulability analysis only has to be done on the critical parts of the system. Such analysis still has to be done using existing techniques [6, 10, 7].

3.2 Improving performance by reducing the GC workload

Another reason to limit non-critical memory allocations is to reduce the amount of garbage collection work needed and thereby increaseing the amount of CPU time available to the application. This can, in turn, improve the application's performance by e.g., allowing more accurate calculations or a higher sampling rate.

Furthermore, in a real-time GC system, such as the one devised by Henriksson [3], additional memory allocations done by a high priority process may cause starvation of low priority processes; either directly, through increased execution time, or indirectly, due to the increase in GC work caused by these allocations (since the garbage collector for the high priority processes run at a higher priority than the system's low priority processes). In complex systems, however, the LP process may be more important for good system performance than a secondary task of the high priority process. By using priorities for memory allocations, the application may be written so that, if the system runs low on memory, the primary tasks of both the HP and the LP processes are executed, but the less important task of the HP process is not.

[1]The word task is used in the sense "a piece of work to be done" and not in the real-time programming sense. For the latter, the words process and thread are used.

4. NON-CRITICAL MEMORY ALLOCATIONS

The semi-concurrent garbage collection scheduling model introduces a special garbage collection scheduling for the high priority processes in order to guarantee that they are never delayed. In this work we take this a step further by also considering the behaviour of the memory allocator and the risk of running out of memory, due to, for instance, unpredictable application behaviour or even wrong worst case estimates. This is done by introducing the notion of non-critical memory allocation requests, i.e., requests for memory that the run-time system may choose to deny without causing the program to fail.

Ultimately, what we want to do is to keep the amount of live non-critically allocated memory below a certain limit in order to make guarantees that critical allocations never will fail. Unfortunately, live memory amount is not a very suitable measurement, since keeping track of this is not always practically possible [2]. In automatically managed memory systems, where we have the problem with floating garbage[3], there is no real way of knowing how much live memory there is in the system. The only factor we can be sure of is the amount of memory available for allocation, so we need to base our decisions on this.

4.1 Non-critical allocation limit

The decision whether to grant or deny a non-critical memory allocation request has to be as simple as possible if it is to be used in high performance applications. We do this by introducing an allocation limit for non-critical allocations; if there is less free, or *allocatable*[4], memory than this limit, no non-critical allocations may be done. This limit will vary over time; at the start of a GC cycle, we have to reserve memory for all the HP memory allocations needed during this GC cycle and then, as the HP process runs and does its allocations, the amount of reserved memory is reduced accordingly. Figure 2 shows schematically how the amount of allocated, reserved and free memory varies over a GC cycle.

When deciding whether to grant or deny a non-critical memory request, we look at how much allocatable memory there is, and how much memory we need to reserve for the HP process so that all its remaining memory allocations during this GC cycle will succeed. Let n be the number of HP periods in a GC cycle, and m_{HP} the amount of memory allocated during each period by the HP process. Then, i HP periods into a GC cycle we need to reserve $R_{HP_i} = (n - i)\ m_{HP}$ bytes for the remaining HP periods during this GC cycle.

[2]In systems with manual memory management it would be trivial to keep track of the amount of live non-critical memory, since objects are explicitly deallocated. The only problem here is the possibility of fragmentation.

[3]Floating garbage is memory that is no longer reachable from the application but has not yet been reclaimed by the garbage collector.

[4]Allocatable memory is memory that is immediately available for allocation. We prefer the term allocatable memory to free memory since, depending on the memory allocator or garbage collection algorithm used, the term free memory may be difficult to define or even irrelevant. E.g., in a non-compacting system, the amount of free memory may be much larger than the amount of allocatable memory due to fragmentation.

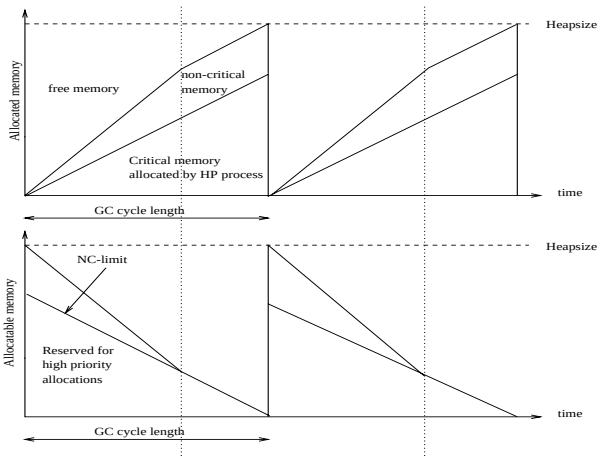


Figure 2: A schematic figure showing the limit for non-critical allocations. The dotted lines indicate the times where the non-critical limit is equal to the amount of allocatable memory, i.e., when the system starts to deny non-critical allocation requests.

Non-critical memory allocations should only be allowed if they won't cause the amount of allocatable memory to drop below R_{HP}.

4.2 Fixed GC cycle length

In order to be able to guarantee that the HP process always will get the memory it requests, we need to make sure that the GC always keeps up with the application. I.e., after each invocation of the HP process, the GC must do enough GC work so that all the allocations during the next HP process invocation will succeed. Given the amount of memory allocated by the HP process each period and the amount of memory reserved for HP allocations we can calculate the GC cycle time expressed in number of HP process periods. We call this time the nominal GC cycle time.

To ensure that no HP allocation fails, we need to complete each GC cycle within this time, even if the actual amount of allocations done during the current GC cycle are less than the worst case. Otherwise, the situation may arise that there is allocatable memory left, but not enough for another complete HP process invocation. If a HP process is started at that time, it will require more memory than currently available and thus, that HP process will be delayed by panic garbage collection.

5. NON-CRITICAL MEMORY IN JAVA

The main objective when implementing these ideas in a Java environment was that no changes to the syntax of the Java language should be made, and that programs written for our system should work on any Java platform (but, of course, without the added sematics of non-critical memory allocations.)

Our proposed approach is to use the exception mechanism of Java, so we define a special exception class, `NoNonCriticalMemoryException`, with the added semantics that all allocations that are done in a block which catches

that exception are non-critical. Figure 3 shows a simple program which does both critical and non-critical memory allocations. This program will run on any Java platform with the only addition of an (empty) exception class.

```
void example(){
        Object aCriticalObject = new Object();
        foo(aCriticalObject); // do something important
        try{
                Object aNonCriticalObject = new Object();
                foo(aNonCriticalObject);
                doSomething();
                        // do something
                        // if the non-critical
                        // allocation was successful
        } catch(NoNonCriticalMemoryException e){
                        // non-critical allocation failed
        }
}
```

Figure 3: Small example program. The allocation of `aCriticalObject` is always done, but the allocation of `aNonCriticalObject` may be denied. If the allocation fails, a `NoNonCriticalMemoryException` is thrown and may be handled in the catch-clause.

Non-criticality is transitive, i.e., memory allocations done in a method that is called from a non-critical region, like the calls `foo(aNonCriticalObject)` and `doSomething()` in Figure 3, are also non-critical. Note, however, that the first call to `foo()`, `foo(aCriticalObject)` is *not* non-critical since the call is not made from a non-critical block. This behaviour is preferable since an auxilliary function could be called both from critical and non-critical regions of the same program.

The exception class `NoNonCriticalMemoryException` is an unchecked exception in order to make such transitivity possible without having to litter the code with `try` and `catch` clauses. An instance of this class can be statically allocated to avoid wasting memory.

We have made an experimental implementation using the IVM (Infinitesimal Virtual Machine) [4], a very compact real-time Java virtual machine currently being developed at the Department of Computer Science, Lund University. Currently, we explicitly turn non-critical allocations on and off using a native method `IVM.setMemoryPriority()`. This is not fundamentally different from our proposed approach since the `setMemoryPriority()` calls could be inserted automatically by the class loader as the exception catching table is set up (much in the same way as `monitorenter` and `monitorexit` are generated for synchronized blocks). This is, we believe, a better approach than dynamically checking whether `NoNonCriticalMemoryException` is caught at each allocation, since such a run-time check would be more expensive.

6. EXPERIMENTAL RESULTS

We have implemented these ideas in a simple control system. For the experiments, we used a lab process with a ball on a beam. The angle of the beam is controlled in order to roll the ball to a given position on the beam, see Figure 4.

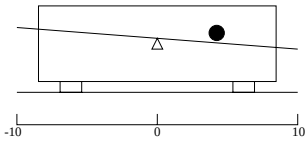


Figure 4: The ball-on-beam process. The beam can be rotated to roll the ball to the desired position. The position of the ball is in the interval $[-10, 10]$.

The control was done by a Java application consisting of three threads; a user interface (low priority), a reference generator (high priority) and a controller (high priority). In addition to doing the actual control, the controller thread sends log data back to the user interface thread.

6.1 Avoiding out-of-memory situations

We have encountered two scenarios where non-critical memory allocations can help making sure that a change to a previously working system doesn't risk breaking it: increasing the sampling rate of the controller and reducing the amount of memory available to the application.

When the sampling rate is increased, the controller both uses a greater part of the CPU time and allocates log data at a higher rate until we get to a point where the user interface thread doesn't get the CPU time needed for consuming all the log data and the application runs out of memory and fails. By making the log data allocations non-critical, this cannot happen and the control is not affected.

Reducing the available memory[5] will, obviously, at some point cause the application to fail. However, by making the allocation of log data non-critical, the minimum memory requirement for the application may be significanty reduced compared to the original version.

The following traces illustrate the first scenario. In these experiments, the period of the reference generator and the controller was both 20 ms, and a log data object about 60 bytes. Figure 5 shows a run of the ball-on-beam system without non-critical memory. The high allocation rate causes a large GC workload and the UI process is starved, eventually leading to failure. Figure 6 shows the same system where the allocation of log data has been made non-critical. The majority of the log data allocations are still made, but the allocation is kept at a sustainable level. Figure 7 shows a close-up of Figure 6 where you can see the non-critical behaviour more clearly.

6.2 Improving performance

Our experiments also indicate that it is possible to achieve better control performance by limiting the amount of non-critical memory allocations. The plots in Figure 8 show two runs of the ball-on-beam application without and with non-critical memory allocations enabled, respectively.

In the version without non-critical allocations, the high al-

[5]This could occur either by actually running the system on a smaller platform or, perhaps more likely, by adding more threads to the system.

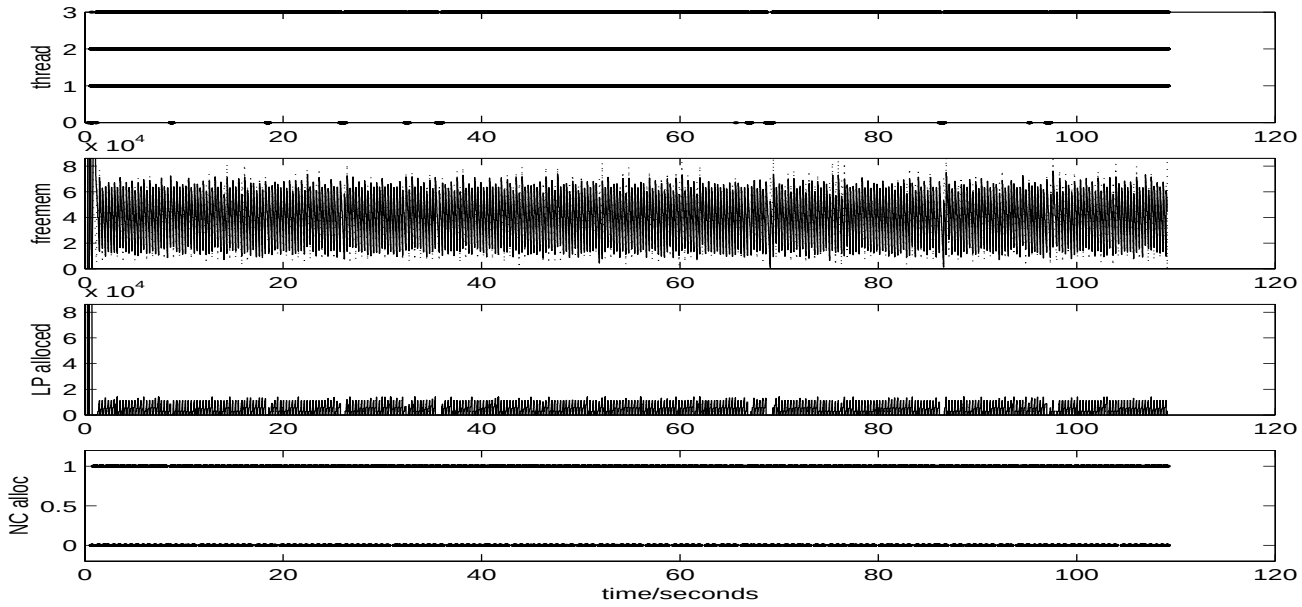


Figure 6: A run of the ball-on-beam system with log-data allocations made non-critical. In the thread plot you see that the UI thread gets CPU time throughout the run. The third plot shows the amount of memory allocated by low priority processes during this cycle. The fourth plot shows if non-critical allocations succeed or not; high level means success and low level is deny.

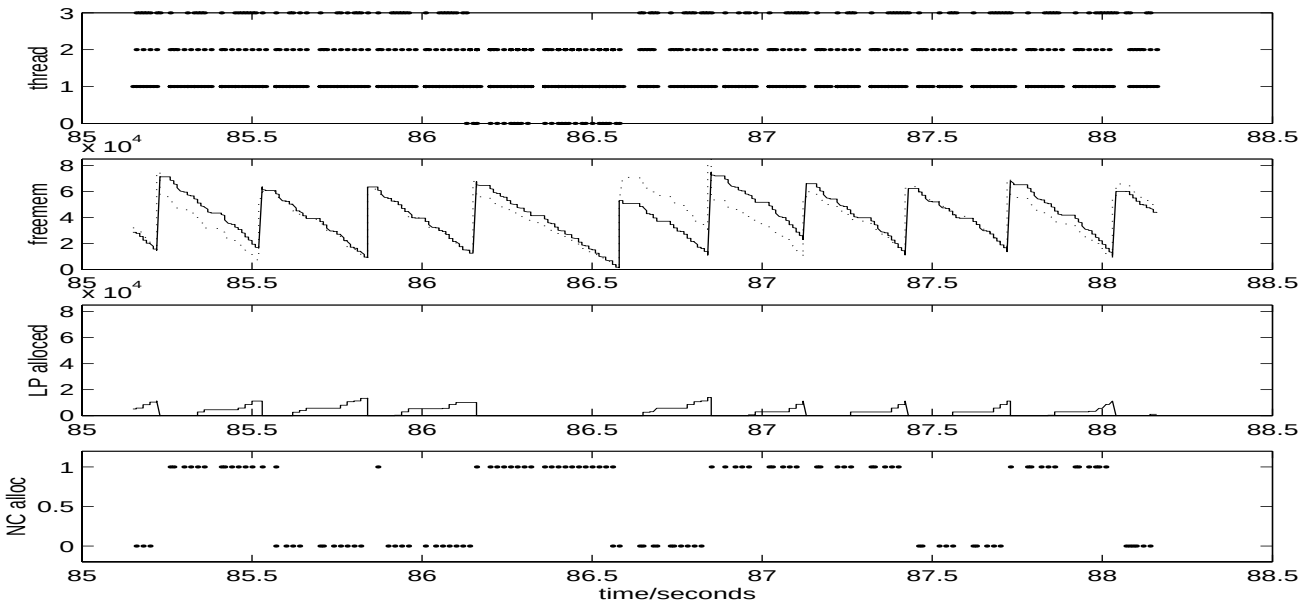


Figure 7: Closeup to show the non-critical memory behaviour.The dotted line in the freemem plot is the non-critical limit.

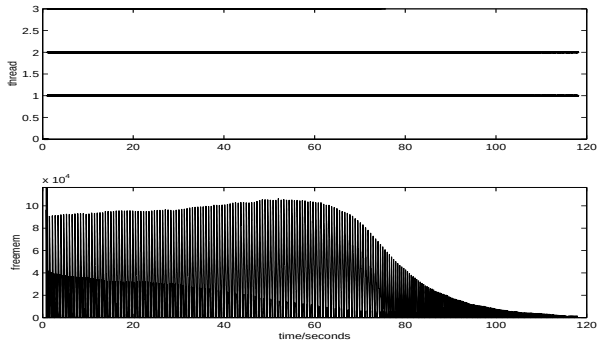


Figure 5: A sample run of the ball-on-beam system without non-critical memory. The UI thread *(3)* doesn't get enough CPU time to consume all plot data that is produced. After $t = 75$ it is totally starved by the GC. Then, less and less memory is available and more and more CPU time is spent doing (panic) GC. In the first half of the run the controller *(1)* and reference generator *(2)* threads run unimpeded, and the control was OK until $t = 90$. After that the amount of panic stop-the-world GC caused so long delays that the controller dropped the ball. The CPU load is almost 100% and the idle thread *(0)* is not run except in the very beginning. The reason that the maximum amount of allocatable memory increases in the middle is that when the GC cycles get shorter there is less floating garbage.

location rate occasionally forces the garbage collector to do a full garbage collection cycle in order to reclaim enough memory to satisfy the allocation needs. This delays the high priority controller process so that it misses its deadline which, in turn, degrades the control performance.

When the allocation of log data is made non-critical, the allocation is kept below the safe limit and the system runs as designed, with more consistent control performance.

7. RELATED WORK

7.1 Memory Management in Real-Time Java

There are two specifications for real-time Java; The Real-Time Specification for Java (RTSJ) [2] and the Real-Time Core Extensions (RTCE) [1]. They both try to solve the real-time garbage problem by avoiding it, using region based approaches to memory management for the real-time threads. The non-real-time threads do their memory allocation on a heap with traditional garbage collection.

RTSJ uses *scoped memory areas* for the high priority threads. Objects allocated in scoped memory areas are not garbage collected but instead the whole memory area is reclaimed when the program exits the scope in which the memory area was allocated.

The access restrictions associated with scoped memory (e.g., objects allocated on the heap may not reference objects in scoped memory, and real time threads aren't allowed to access the heap[6]) make inter-thread communication more difficult. Real-time threads, however, may share scoped memory areas.

[6]Since the heap is garbage collected, real-time threads with hard time constraints must be of the type *NoHeapRealTimeThread* in order to avoid interference from the garbage collector.

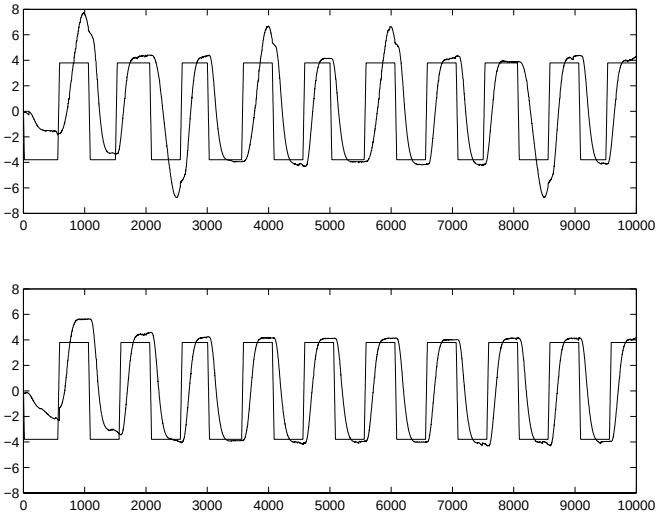


Figure 8: Plots showing the reference value and the measured position for the ball-on-beam process. The upper plot shows the system without non-critical memory allocations and the lower plot shows the system where the allocation of plot data is non-critical. The irregular behaviour in the upper plot, around samples 2500, 4000, 6000, and 8500, is caused by the controller process being delayed by the garbage collector due to the program running out of allocatable memory and forcing a complete garbage collection cycle.

In RTCE, real-time objects are allocated in *core memory*, and may not access objects on the garbage collected *baseline* heap. Objects on the heap may, with some restrictions, access core objects through special method calls. Core objects are allocated in an *allocation context*. When an allocation context is released, all objects in it may be eligible for reclamation but, since there might be references from the baseline heap, the actual reclamation is done by the baseline garbage collector when all of the objects in the allocation context are unreachable. Thus, a non-real-time garbage collector is used to reclaim the memory used by the real-time processes.

In RTCE, there are no limitations on which allocation contexts objects may reference so it is up to the programmer not to release an allocation context when it is still referenced.

RTCE also specifies stack allocation of real-time objects, which are to be automatically reclaimed as the scope is exited. To allocate stack objects, a set of restrictions apply and the reference must explicitly be declared stackable.

Under both of these specifications, the same behaviour as our system can be achieved by using one memory area (or

allocation context) for critical memory and another (or the heap) for the non-critical objects. The drawbacks of these approaches compared to ours are firstly that a much higher responsibility is placed on the programmer and that garbage collection cannot be used for the real-time threads and secondly that the access restrictions make communications between e.g. low and high priority threads more complicated.

7.2 Worst case analysis

Good worst case estimates for execution time[8] and memory usage[7] are crucial for making any kind of real-time guarantees. The experimental tool Skånerost[9] developed at our department provides interactive worst case execution time and memory consumption analysis based on timing schema and source code annotations for (currently a subset of) the Java language.

8. CONCLUSIONS

We have introduced the idea of applying priorities to memory allocation and shown how this can be used to enhance the robustness of real-time applications. The advantage this approach gives is twofold; firstly, it provides run-time support for prioritizing memory allocations if there is not enough memory for all allocation requests. Secondly, but equally important, it makes it easier to provide hard guarantees since the worst case memory usage calculations only has to be done for the critical parts of the system as non-critical allocations cannot cause the system to fail. Furthermore, we also suggest that the same mechanisms could be used to increase performance by limiting the amount of memory allocation and, consequentially, GC work.

Our approach is based on the notion of non-critical memory allocation requests, which can be used by the programmer to indicate that the memory allocations done in a certain part of the program are less important than the rest. Such non-critical allocations may be allowed to fail if the run-time system decides that that memory could be of better use elsewhere or that the increased garbage collection work would degrade system performance.

We also propose a way of introducing non-critical memory allocation in a Java system without making any changes to the syntax of the Java language and we have implemented this in an experimental Java virtual machine.

Preliminary experiments show that this mechanism is fairly easy to implement and can improve the robustness and performance of a control application by restricting its operation to the critical tasks if the system runs low on memory. It allows the programmer to write a system that performs better if run on a faster and larger system but whose critical tasks won't fail if it is run on a system with less than ideal amount of memory. Instead, the non-critical features of the system will automatically be turned off if there isn't enough memory for them to be safely executed.

9. REFERENCES

[1] Real-time core extensions. International J Consortium Specification, 2000.

[2] G. Bollella, et al. *The Real-Time Specification for Java*. Addison-Wesley, 2001.

[3] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 1998.

[4] A. Ive. *Implementation of an Embedded Real-Time Java Virtual Machine Prototype*. Lic. eng. thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 2002. (to be published).

[5] R. Jones and R. Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley & sons, 1996.

[6] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5), 1986.

[7] P. Persson. Live memory analysis for garbage collection in embedded systems. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, Atlanta, Georgia, May 1999.

[8] P. Persson and G. Hedin. Interactive execution time predictions using reference attributed grammars. In *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*, Amsterdam, The Netherlands, March 1999.

[9] P. Persson and G. Hedin. An interactive environment for real-time software development. In *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe 2000)*, St. Malo, France, June 2000.

[10] L. Sha, R. Rajkumar, and J. P. Lehoczky. Generalized rate-monotonic scheduling theory. *Proceedings of the IEEE*, 82(1), 1994.

Deadline Dependent Coding with Concatenated Hybrid ARQ Schemes for Wireless Real-Time Communication*

Elisabeth Uhlemann[1,2], Tor M. Aulin[2] , Lars K. Rasmussen[2,3] and Per-Arne Wiberg[1]

[1]Halmstad University, Box 823, SE-301 18 Halmstad, Sweden. E-mail: {bettan, per-arne.wiberg}@ide.hh.se
[2]Chalmers University of Tech., Dept. of Computer Eng., SE-412 96 Göteborg, Sweden. E-mail: tor@ce.chalmers.se
[3]Univ. of South Australia, Inst. for Telecom. Research, Mawson Lakes, Australia. E-mail: larsr@ce.chalmers.se

Abstract

A protocol for real-time communication over a wireless channel, based on concatenated codes using iterative decoding is proposed. The concept of deadline dependent coding (DDC), previously suggested by the authors, to maximize the probability of delivering the information before a given deadline, is further developed to include concatenated codes. The strategy of DDC is to combine different coding and decoding methods with automatic repeat request (ARQ) in order to fulfill the application requirements. These requirements are formulated as two Quality of Service (QoS) parameters: deadline (t_{DL}) and probability of correct delivery before the deadline (P_d), leading to a probabilistic view of real-time communication. An application can negotiate these QoS parameters with the DDC protocol, thus creating a flexible and reliable scheme.

1 Introduction

The tremendous development in wireless communication has provided opportunities in many related fields, such as remote sensing, monitoring, and control. New technologies and products are introduced into the market at an ever-increasing rate. This wireless evolution offers improvements for industrial applications, where traditional wireline solutions have prohibitive problems in terms of cost and feasibility. Wireline implementations are for example not cost efficient for large, temporary production lines, and may not be feasible at all for systems including rotating or high mobility machinery such as measurement and control of moving objects. Another sample application, requiring wireless access, is communication to and from different kinds of vehicles in factory automation situations.

Monitoring and control of industrial applications often introduce strict real-time constraints in order to guarantee proper operation. A time-critical system or a real-time system is characterized by the fact that it has deadlines to meet. In this work, we are interested in real-time communication systems where the timeliness of the delivered data is equally important as the correctness of the delivered data. This provides a challenge for *wireless* real-time communication due to the harsh communication environment encountered when transmitting over a wireless channel. The inherent consequence is a relatively high error rate, making the wireless channel unreliable in comparison to optical wireline channels. This has limited the extensive use of wireless access in real-time systems. Implementations of wireless communication systems for industrial use, guaranteeing real-time delivery, have been attempted, however, no general framework has been suggested. As a consequence, it is not straightforward to evaluate the general dependability of such systems with respect to real-time delivery and reliability.

1.1 QoS in Real-Time Communication

The literature in the real-time field often discusses two different classes of systems, *hard* and *soft* real-time systems. In a hard real-time system, late delivery cannot be tolerated. In contrast, in a soft real-time system a specified low probability of late delivery is tolerated, while permitting performance degradation by relaxing the real-time constraints. In this work we use a probabilistic view of the real-time constraints. This means it is no longer meaningful to talk about hard or soft real-time systems. We rather talk about deadline for delivery and the probability of succeeding in delivering correct information before this deadline. We therefore use two parameters: deadline (t_{DL}) and probability of correct delivery before deadline (P_d). They can be viewed as Quality of Service (QoS) parameters of the real-time communication system. It follows that a protocol can negotiate these parameters with an upper or a lower layer. The protocol can then either reject or accept a request, guaranteeing the delivery based on the given values of the QoS parameters. If the request cannot be accepted the application can possibly re-negotiate. One of the objectives of the real-time communication protocol is to maximize the probability that the communication system will be able to accept the request. Note that a hard real-time system defined using these QoS parameters will have P_d=1, a situation that cannot be achieved in any physical system due to noisy channel conditions, [1].

* 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.

1.2 Deadline Dependent Coding

The main idea behind the concept of Deadline Dependent Coding (DDC) [2] is to make all aspects of the communication protocol deadline dependent. The protocol should also attempt to minimize the required bandwidth, the transmitted energy and the time required to successfully deliver the information. The QoS parameters t_{DL} and P_d are mapped onto a retransmission protocol, which plays the role of maximizing the probability of correct delivery before deadline, using a minimum of resources. In a multiple access radio system, multiple access interference (MAI) is encountered. Hence, it is important to limit the interference created by the acting nodes. Consequently, another advantage of a carefully designed retransmission scheme is that we only use the required amount of redundancy suitable for the current channel condition and thus the amount of MAI is reduced.

The DDC scheme differs from existing communication protocols in a number of ways. Most importantly, DDC explicitly uses the deadline to control the transmission suite. There are a number of real-time communication protocols, e.g., [3] and [4] that are best effort protocols and consequently do not give any guarantees or explicit predictions on the probability of delivery. Other protocols have been developed, which guarantee hard deadlines [5], [6]. They depend, however, on a reliable channel, while DDC in contrast strives to maximize the probability of correct delivery over an unreliable channel.

In this paper the concepts of DDC are further developed to incorporate serially concatenated block codes using iterative decoding.

2 Iterative Decoding

Concatenated codes using iterative decoding is a way of providing long codes with manageable decoding complexity, [7],[8]. The iterative decoder is sub-optimal and hence less complex, but for certain channels, it approaches the performance of the optimal decoder in an iterative fashion.

A time and safety critical application benefits from the long powerful code yielding reliable communication, while the processing time of the decoder is kept low. Within a DDC protocol, the iterative decoding algorithm also gives the opportunity to always deliver something to the receiver just before the deadline. We can offer a fast tentative response and progressively provide iterative refinements. This last-minute delivery can be complemented by a measure of reliability of the delivered data based on *a posteriori* probability (APP) information [8].

A common approach for iterative decoding is to allow for a fixed number of iterations in the process. This may lead to unnecessary iterations, or to performance degradation if the process is terminated prematurely. Applying a performance based stopping criterion these problems can be addressed. Such stopping rules are connected to the convergence behavior of the iterative decoder. The convergence behavior of concatenated codes using iterative APP decoding algorithms has been paid considerable attention recently, e.g. [9], where a stopping rule based on thresholding the average log-likelihood ratio (LLR) of the so-called extrinsic APPs over a packet is found to be efficient. In this work, we consider this approach for serially concatenated codes. It should be noted that other methods, such as using an additional error detection code or using the error detection capabilities of the constituent codes might also be used. We have chosen the average LLR because it is simple and provides information about the current error rate.

The stopping criterion is intended to stop the iterative process once the mean LLR is above the threshold, however, if the decoding algorithm does not converge for a specific packet, this may never happen. As we are dealing with real-time communication we must have an upper limit on the time to decode a packet. Since the decoding complexity is directly related to the number of iterations, we consequently need to have an upper limit on the number of iterations. Examining the convergence behavior in a concatenated system, it is generally noticed that for a majority of packets, convergence is observed after a fixed number of iterations. Consequently, in this work the maximum allowed number of iterations is set accordingly. A non-negligible number of packets may, however, converge after less iterations.

3 Concatenated Hybrid ARQ

By concatenated hybrid automatic repeat request (CHARQ) we mean a HARQ scheme using concatenated codes as the error control code, but also a concatenation between retransmissions. Now, soft information may also be passed between retransmissions to be used in the iterative decoding process.

The convergence behavior of the iterative detector yielding the stopping criterion is also used to define the retransmission criterion that will be applied in this work. Consequently, we will not use an additional code for the sole purpose of error detection, but instead request a retransmission if the iteration-stopping criterion is not fulfilled after the maximum allowed number of iterations.

Since the retransmission scheme is intended for wireless real-time applications, the number of retransmissions is limited and the ARQ scheme becomes truncated [10]. Note that, once a request is accepted, the final decoder results are always delivered prior to the deadline, regardless of quality.

In a pure ARQ scheme, rejected packets are discarded. Previously received packets may, however, be used for so-called packet combining, in order to improve performance. There are two major types of packet combining, diversity combining, [11], and code combining, [12]. Diversity combining is in general done before the decoder whereas code combining is generally done within the decoder.

Since the CHARQ scheme is based on serially concatenated block codes using iterative decoding, some of the packet combining techniques may alternatively be viewed as different decoding strategies. The extrinsic APP information used in the iterative decoding process can be saved and used when decoding after a retransmission, similar to the work in [13]. The strategy of saving the extrinsic information can be seen as a packet combining technique, i.e. some sort of turbo code combining or concatenated code combining. The process of saving the extrinsic information requires no extra memory since this information is already saved to be passed and updated between the decoders. It is simply not reset when a new copy of a packet arrives. Saving the extrinsic information may also be seen as a doping criterion yielding faster convergence [14]. The concept of doping is based on providing side information to the decoding process. The extrinsic information from the decoding process of the previous transmission constitutes useful side information. The use of prior knowledge leads to a constructive bias, speeding up convergence. Applying this point of view, the extrinsic information can be used in conjunction with traditional diversity combining schemes. The basic scheme used in this work is a type-I HARQ scheme, hence we can directly apply the diversity combining technique equal gain combining [15]. This implies that the receiver just averages over the demodulator output metrics from all received copies of a packet to produce a combined packet for decoding. Hence, the code rate will remain the same.

3.1 Performance Results

We have used binary phase shift keying (BPSK) modulation [1] throughout this work, as it is a commonly used method in many existing hardware platforms. Further, we consider serially concatenated block codes based on two binary Reed Solomon RS(7,3) codes with a pseudo-random interleaver of size 945 used in conjunction with the ARQ protocol [16]. The communication channel is the additive white Gaussian noise (AWGN) channel [1], which models thermal noise present in all electronic equipment. An error free feedback channel is assumed throughout this work. The maximum number of iterations was chosen, based on simulation results, to be seven and the maximum number of retransmissions allowed was chosen to be four. Consequently, the maximum number of iterations that can ever occur is $7 \cdot 5$=35, yielding an upper bound on the execution time or communication time of each task. The LLR threshold was set to ± 10.

We consider the following procedure:

1. Use iterative decoding on the received packet, checking the LLR threshold after each iteration. If the stopping criterion is fulfilled go to step 3. If seven iterations are reached go to step 2.
2. If four retransmissions are made, go to step 3, else request a retransmission. Do packet combining, if any. Go to step 1
3. Output hard quantized values.

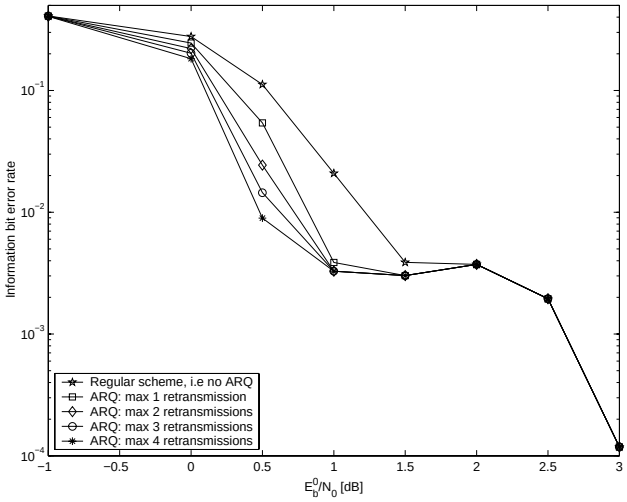


Figure 1: *BER as a function of E_b^0/N_0 for different ARQ schemes. Erroneous packets are discarded.*

In Figure 1, schemes allowing from 0 to 4 retransmissions are considered and compared in terms of bit error rate (BER) plotted against E_b^0 / N_0, where E_b^0 is the bit energy for the regular scheme with no retransmissions. For high signal to noise ratio, SNR= E_b^0 / N_0, all the curves tend towards the regular scheme without ARQ, since the probability of a retransmission tends to zero. For very low SNR, the curves of the different schemes are also on top of each other, since the probability of a retransmission tends to one. Virtually all transmissions are rejected and thus there is nothing to gain from a retransmission since the final packet must be accepted. The region between these two extreme values, however, is the working area of the ARQ schemes and a noticeable gain can be observed. The reason for showing the different truncation lengths in one figure is to see the improvements when more time is allocated to allow for additional retransmissions.

At high SNR, a loss in performance due to the stopping criterion can be observed. This is a result of the fact that the threshold is fixed for all values of SNR. The performance in terms of BER for a particular stopping criterion should, however, be compared to the number of iterations actually needed. It can be concluded that, although we have a considerable performance loss for high SNR, the average number of iterations needed to fulfill the stopping criterion is low [16]. It follows that we gain real-time benefits at the expense of BER performance. A truncated ARQ system offers most improvement in the region where the initial probability of retransmission is relatively high at the same time as the probability for accepting a packet within the allowed number of retransmissions is high. This is the case for the region 0-1.5 dB [16]. Consequently, in this region, the stopping criterion is good since we obtain faster convergence at virtually no performance loss. It follows that even though the chosen stopping criterion has an

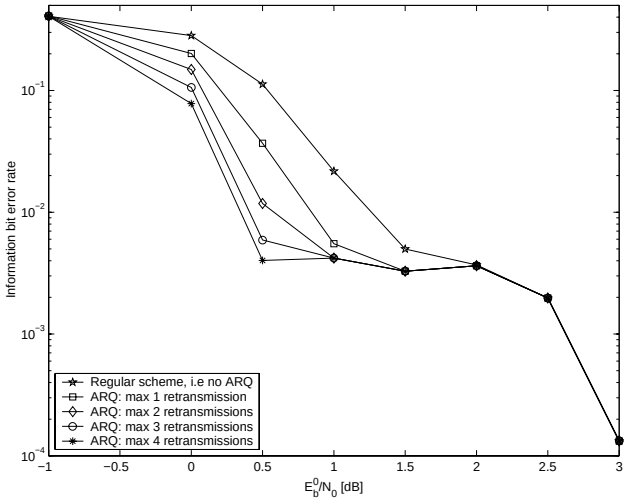


Figure 2: *BER as a function of E_b^0/N_0 for different ARQ schemes. The extrinsic information is used for doping.*

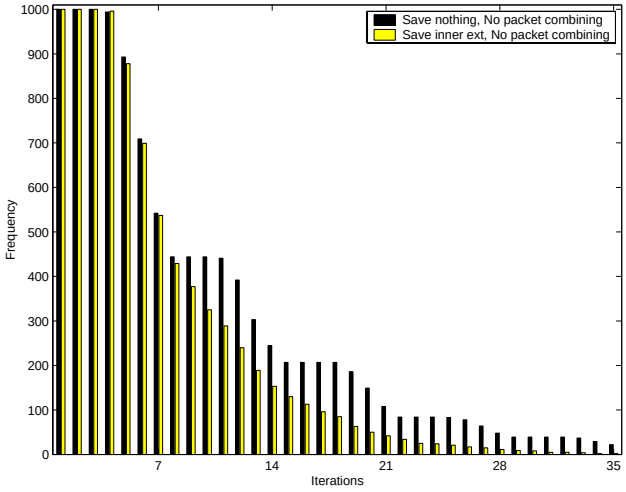


Figure 3: *Relative frequency of occurrence for each iteration. A scheme without packet combining is compared to a scheme saving the extrinsic information and is plotted at E_b^0/N_0 =0.5 dB.*

error plateau at 1.5-2 dB, it is still relevant in the SNR area where we expect to operate.

In Figure 2 the BER for the same schemes are shown, but this time the extrinsic information is saved and used when decoding after a retransmission. It can be seen that the BER for low SNR is improved as compared to the case with no packet combining in Figure 1. Note that the curves representing a regular scheme with no ARQ in both Figure 1 and Figure 2 are the same since no packet combining is made for these curves and the same stopping criterion is used.

When equal gain combining is applied, additional memory is required to store erroneous copies used in the combining process, but the gain in terms of BER is considerable. Saving the extrinsic information yields minor additional improvements to the resulting BER performance for the equal gain combining technique. Due to space limitations, no BER plots of equal gain combining are included. Results can be found in [16].

Whenever an ARQ scheme is considered its performance is usually measured both in terms of BER and throughput. In our case we are not only concerned with the number of retransmissions, but also the number of iterations made for each retransmission. We want to know if the packet combining technique considered improves both the throughput and the convergence speed. We will compare the scheme saving the extrinsic information to that where no packet combining technique is used. The relative frequency of occurrence of each iteration is plotted in Figure 3. Note that after 7 iterations a retransmission occurs. Consequently, iteration number 8 in Figure 3 corresponds to the first iteration of a newly received retransmission. The black columns in Figure 3 correspond to the scheme discarding erroneous copies of a packet. Hence, when a new packet arrives the iterative process is restarted. This can be seen as a staircase characteristic with a step every seventh iteration, since each new packet requires a few iterations to converge. The lighter columns correspond to the scheme saving the extrinsic information. The number of iterations decreases smoothly due to the fact that the decoder does not have to restart every time a retransmission arrives. Hence, convergence is faster. Note that the first seven iterations are the same for the two different schemes, as no retransmissions have yet occurred

In Figure 4 the equal gain combining schemes are compared in a similar way. Here, one scheme is using the saved extrinsic information and the other scheme is not. The most noticeable difference between Figure 3 and Figure 4 is that equal gain combining significantly reduces the number of iterations. For the first seven iterations in Figure 4 the two schemes are the same, since no retransmissions have been made and hence no packet combining is performed. However, we can still see an improvement in the convergence speed when the

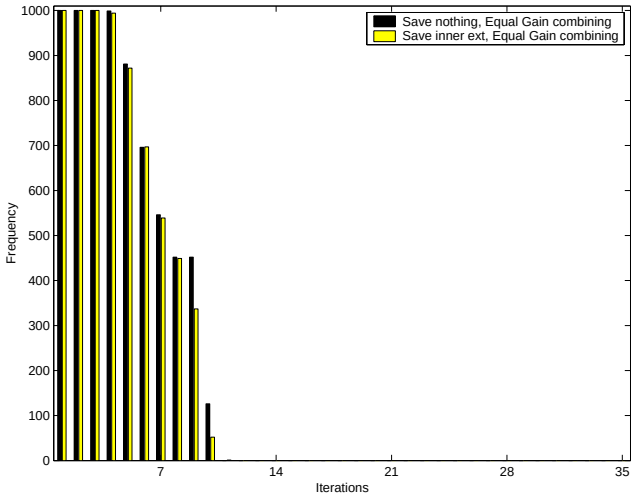


Figure 4: *Relative frequency of occurrence for each iteration, plotted at E_b^0/N_0 =0.5 dB. Both schemes employ equal gain combining, but one is saving the extrinsic information whereas the other is not.*

extrinsic information is saved and used, since the columns for iteration nine and ten have a lower frequency for this scheme. Consequently, since saving the extrinsic information does not require any major changes in the decoder it is reasonable to include it.

Additional results, together with a more detailed analysis can be found in [16].

When a real-time scheme is considered, the transmission propagation time has to be taken into account. Consequently, an iteration cannot be compared directly to a retransmission. In the bar charts on relative frequency of occurrence for each iteration, one may use one time unit to make one iteration, while a retransmission with a following iteration takes, for example, five time units.

4 Conclusions

The resulting DDC protocol presented here is based on concatenated hybrid ARQ techniques, creating a flexible and reliable scheme to meet real-time constraints. Each packet has a certain t_{DL} and P_d required by the user or the application. These two parameters will be translated into a maximum number of retransmissions allowed and a maximum number of iterations allowed for the sub-optimal iterative decoding algorithm per transmission.

If simple, cheap transmitters and receivers are required, e.g., a mobile sensor with limited battery supply, the mapping of the QoS parameters onto the CHARQ-DDC protocol may be done using a look up table. If the transmitter and receiver can be more costly, the mapping can be done adaptively based on the current estimated channel conditions.

Two different packet combining techniques for the CHARQ-DDC scheme were presented and evaluated. The diversity combining technique equal gain combining was applied. It is concluded that equal gain combining gives substantial performance improvements. However, since the hybrid ARQ system is based on serially concatenated codes using iterative decoding, a different category of packet combining techniques can be used. The iterative decoder passes extrinsic information between its constituent component decoders, which can be saved from one transmission and then used in the decoding process of a retransmission. This technique can be viewed as a code combining technique, and using it leads to performance improvement in terms of BER and convergence speed at very low additional cost in decoder complexity. Saving the extrinsic information for use in the decoding process of a retransmission can also be viewed as a doping operation of the iterative decoder. With this observation in mind, equal gain combining can be used in conjunction with the saved extrinsic information. It is concluded that the inclusion of the extrinsic information obtained from a previous transmission does speed up the convergence process of the iterative decoder.

References

[1] J. G. Proakis, *Digital Communications*, McGraw-Hill Book Co., New York, 1995, 3rd edition.

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

[3] W. Zhao, J. A. Stankovic and K Ramamritham, "A window protocol for transmission of time-constrained messages," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1186-1203, September 1990.

[4] S.-K. Kweon, K. G. Shin and Q. Zheng, "Statistical real-time communication over Ethernet for manufacturing automation systems," *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Phoenix, Arizona, pp. 192-202, May 1999.

[5] T. F. Abdelzaher, E. M. Atkins and K. G. Shin, "QoS negotiation in real-time systems and its application to automated flight control," *IEEE Transactions on Computers*, pp. 1170-1183, Nov. 2000.

[6] J. Kim and K. G. Shin, "Performance evaluation of dependable real-time communication with elastic QoS," *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 295-303, 2001.

[7] G. D. Forney, Jr., *Concatenated Codes*, M.I.T. Press, Cambridge, MA, 1966.

[8] C. Berrou, A. Glavieux and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: turbo codes," *Proceedings of the International Conference on Communications*, Geneva, Switzerland, pp. 1064-1070, May 1993.

[9] A. C. Reid, T. A. Gulliver and D. P. Taylor, "Convergence and errors in turbo-decoding," *IEEE Transactions on Communication*, pp. 2045-2051, December 2001.

[10] E. Malkamäki, *Performance of Error Control over Block Fading Channels with ARQ Applications*, Ph.D. dissertation, Helsinki University of Technology, October 1998.

[11] P. Sindhu, "Retransmission error control with memory," *IEEE Transactions of Communications*, vol. 25, no. 5, pp. 423-429, May 1977.

[12] D. Chase, "Code combining - a maximum-likelihood decoding approach for combining an arbitrary number of noisy packets," *IEEE Transactions of Communications*, vol. 33, no. 5, pp. 385-393, May 1985.

[13] K. R. Narayanan and G. L. Stüber, "A novel ARQ technique using the turbo coding principle," *IEEE Communications Letters*, vol. 1, no. 2, pp. 49-51, March 1997.

[14] S. ten Brink, "Code doping for triggering iterative decoding convergence," *Proceedings of the IEEE International Symposium on Information Theory*, ISIT 2001, Washington D.C., USA, p. 235, June 2001.

[15] M. Schwartz and W. R. Bennett, *Communication Systems and Techniques*, McGraw-Hill, Inc. 1966.

[16] E. Uhlemann, *Hybrid ARQ using serially concatenated block codes for real-time communication - an iterative decoding approach*, Licentiate Thesis, Chalmers University of Technology, Göteborg, Sweden, October, 2001. Available at http://www.hh.se/staff/bettan.

Regular Model Checking made Simple and Efficient

Parosh Aziz Abdulla[1], Bengt Jonsson[1], Marcus Nilsson[1], and Julien d'Orso[1]

Dept. of Computer Systems, P.O. Box 337, S-751 05 Uppsala, Sweden
{parosh,bengt,marcusn,juldor}@docs.uu.se

Abstract. We present a new technique for computing the transitive closure of a regular relation characterized by a finite-state transducer. The construction starts from the original transducer, and repeatedly adds new transitions which are compositions of currently existing transitions. Furthermore, we define an equivalence relation which we use to merge states of the transducer during the construction. The equivalence relation can be determined by a simple local check, since it is syntactically characterized in terms of "columns" that label constructed states. This makes our algorithm both simpler to present and more efficient to implement, compared to existing approaches. We have implemented a prototype and carried out verification of a number of parameterized protocols.

1 Introduction

Regular model checking has been proposed as a uniform paradigm for algorithmic verification of several classes of infinite-state systems; in particular *parameterized systems* [KMM+97,ABJN99,BJNT00,PS00]. Such systems arise naturally in many applications. For instance, the specification of a protocol may be parameterized by the number of components which may participate in a given session of the protocol. In such a case, it is interesting to verify the correctness of the protocol, regardless of the number of participants in a particular session. The idea of regular model checking is to perform symbolic reachability analysis, using words over a finite alphabet to represent states, and using finite-state transducers to describe transitions between states. Such an approach has been advocated by, e.g., Kesten et al. [KMM+97], Boigelot and Wolper [WB98], and implemented, e.g., in the Mona [HJJ+96], MoSel [KMMG97], or LASH [BFL] packages.

A generic task in most symbolic model checking paradigms is to compute a representation for the transitive closure of the transition relation. Such a characterization can then be used to compute the set of reachable states (e.g. for verifying safety properties), or to find loops when verifying liveness properties [BJNT00,PS00].

A central problem in regular model checking is that the standard iteration-based methods for computing transitive closures, which are used for finite-state systems (e.g., [BCMD92]), are guaranteed to terminate only if there is a bound on the distance (in number of transitions) from the initial configurations to

any reachable configuration. In general, a parameterized or infinite-state system does not have such a bound. For instance, consider a transition of a parameterized system in which a process passes a token to its neighbour. The transitive closure of such a transition relation will be to pass the token to any other process through an arbitrary sequence of neighbours. Therefore, an important challenge in the design of algorithms for computing transitive closures, is to invent techniques in order to enhance the performances of iteration-based methods. One such a technique is that of *accelerations*: try to calculate the effect of arbitrarily long sequences of transitions. Although such an effect is in general not computable, accelerations have successfully been applied for several classes of parameterized and infinite-state systems, e.g., systems with unbounded FIFO channels [BG96,BGWW97,BH97,ABJ98], systems with stacks [BEM97,Cau92,FWW97,ES01] systems with counters [BW94,CJ98], and several classes of parameterized systems [ABJN99,PS00].

In our work [JN00], we gave an explicit representation of a finite-state transducer accepting the transitive closure, for the case that the transition relation satisfies a condition of *bounded local depth*. A related automata-based construction was presented in [BJNT00]. Both these works employ a direct construction of some form of "column transducer", whose states are sequences (columns) of states of the original transducer.

In this paper, we present a technique for computing transitive closures, which is more light-weight than our previous automata-based solutions. The technique uses straight-forward post-image computation augmented with identification of "equivalent" states. Roughly, the construction of transitive closure proceeds by starting from the original transducer, then repeatedly adding new transitions by simple matching of already constructed transitions. During the construction, equivalent states are merged, using an equivalence relation which preserves the set of traces of the transducer. More precisely, our equivalence relation is the combination of a forward simulation and a backward simulation relation. This makes sure that no prefix/suffix combinations are added to the set of traces. The technique represents a substantial simplification over the previous approaches [JN00,BJNT00], where several layers of automata-theoretic constructions were used. An important property of the equivalence relation is that it can be syntactically characterized in terms of "columns" that label constructed states, and therefore it can be determined by a simple local check. This allows for a much more efficient implementation of the algorithm. In fact, a first implementation of the new, simplified, technique improves the running times of examples by up to a factor of ten. At the same time, the technique does not substantially sacrifice completeness. Completeness results, similar to those in [JN00,BJNT00] can be proven.

Related Work Previous work on the general aspects of regular model checking, and on analyzing classes of systems, e.g., pushdown systems, parameterized systems, systems with FIFO channels, or with counters, has already been mentioned earlier in this introduction.

In [BJNT00], we present a technique for computing the transitive closure of a regular transducer. The technique relies on several potentially expensive operations on automata such as checking language equivalence, computing post-images of regular sets, and saturating regular sets with respect to members of the alphabet. These operations are not needed in the present algorithm, leading to a much more efficient implementation (see Section 5).

Dams et al. [DLS01] present a related approach, which differs from ours in the way states are merged. Dams et al. use an extensional equivalence, which is computed by a global analysis of the current approximation of the transitive closure. It appears that this calculation is very expensive, and the paper does not report successful application of the techniques to examples of similar complexity as the more complex examples in Section 5. In contrast, we base the equivalence on a relation defined in terms of the "columns" that label constructed states, which can be determined by a simple local check. The technique in our proof of Theorem 2 is inspired by the proof technique of their paper.

Caucal [Cau00] presents a class of rewriting systems, called *right-overlapping systems* (and symmetrically also *left-overlapping systems*) for which the transitive closure can be computed as a transducer. A simple instance is the token-passing example mentioned at the beginning of this introduction. Our algorithm is guaranteed to terminate on all overlapping rewriting systems.

Touili [Tou01] presents a technique for computing transitive closures of regular transducers based on *widening*, and shows that the method is sufficiently powerful to simulate earlier constructions described in [ABJN99] and [BMT01]. However, these are substantially weaker than for the automata-based techniques. Another approach, based on second order monadic logic [PS00], covers some commonly occurring patterns of successive transduction.

Outline In the next section, we present a simple example which we will use to illustrate our algorithm. In Section 3 we describe the algorithm for computing transitive closures. In Section 4, we show soundness and completeness of the algorithm and present sufficient conditions for termination. Section 5 contains a description of an implementation of the algorithm and the result of applying it to a number of mutual exclusion protocols. Concluding remarks and directions for future research are given in Section 6.

2 An Example

In this section, we present informally, through a simple example, an algorithm which computes R^+, for a regular relation R. It computes successively larger under-approximations of R^+, starting from R. The algorithm consists of repeatedly performing a small basic step which combines two matching transitions of the current approximation. It also uses an equivalence relation on states, for on-the-fly identification of newly produced states.

Let Σ be a finite alphabet of symbols. Let R be a regular relation on Σ, represented by a deterministic finite-state *transducer* $T = \langle Q, q_0, \longrightarrow, F\rangle$ where

Q is the set of states, q_0 is the initial state, $\longrightarrow$: $(Q \times (\Sigma \times \Sigma)) \mapsto Q$ is the transition function, and $F \subseteq Q$ is the set of accepting states. We use $q_1 \stackrel{(a,b)}{\longrightarrow} q_2$ to denote that $(q_1, (a, b), q_2) \in \longrightarrow$. We use a similar infix notation also for the other types of transition relations introduced later in the paper.

Our goal is to construct a transducer that recognizes the relation R^+, where $R^+ = \cup_{i>0} R^i$.

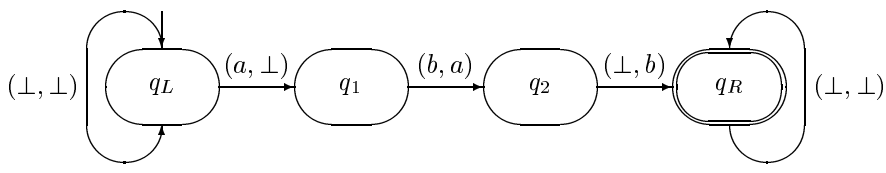
Starting from R, we can in a straight-forward way construct a transducer for R^+ whose states, called *columns*, are sequences of states in Q, where runs of transitions between columns of length i accept pairs of words in R^i. More precisely, define the *column transducer* for T as the tuple $T^+ = \langle Q^+, q_0^+, \Longrightarrow, F^+ \rangle$ where

- Q^+ is the set of non-empty sequences of states of T,
- q_0^+ is the set of non-empty sequences of initial states of T,
- $\Longrightarrow$: $(Q^+ \times (\Sigma \times \Sigma)) \mapsto 2^{Q^+}$ is defined as follows: for any columns $q_1 q_2 \cdots q_m$ and $r_1 r_2 \cdots r_m$, and pair (a, a'), we have $q_1 q_2 \cdots q_m \stackrel{(a,a')}{\Longrightarrow} r_1 r_2 \cdots r_m$ iff there are $a_0, a_1, \ldots, a_m$ with $a = a_0$ and $a' = a_m$ such that $q_i \stackrel{(a_{i-1}, a_i)}{\longrightarrow} r_i$ for $1 \leq i \leq m$,
- F^+ is the set of non-empty sequences of accepting states of T.

It is easy to see that T^+ accepts exactly the relation R^+: runs of transitions from q_0^i to columns in F^i accept transductions in R^i. The problem is that T^+ has infinitely many columns.

We will use x, y, etc. to denote columns in Q^+, and X, Y, etc. to denote sets of columns (i.e., subsets of Q^+). We use regular expressions notation for representing sets. In this paper, we present a procedure for incrementally generating a transducer which accepts the same relation as T^+. The procedure starts from T; by successively adding transitions of T^+ we compute a sequence of successively larger (in terms of sets of accepted pairs of words) transducers, all of which under-approximate R^+. Each new approximation is generated through performing a basic step. The step constructs transitions by combining already constructed transitions. Furthermore, all the time during this procedure, "equivalent" columns will be merged, in order to hopefully arrive at a finite-state result.

As a running example, consider the transducer below over the alphabet $\{\bot, a, b\}$. It relates a word of the form $\bot^i ab \bot^j$ with $\bot^{i+1} ab \bot^{j-1}$, moving the sequence ab one step to the right.

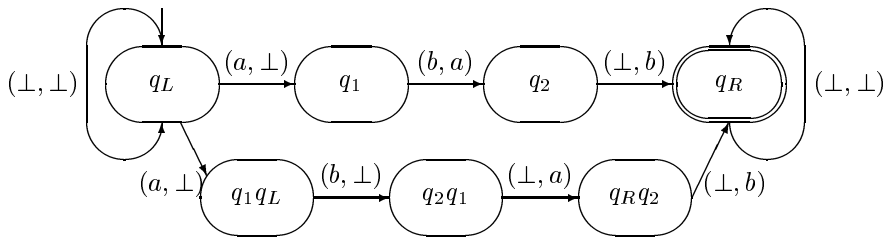


Our algorithm identifies pairs of transitions and combines them in the following way. When we have a transition from x to x' on (a, b), and a transition

from y to y' on (b, c) we add the transition xx' to yy' on (a, c). Furthermore, we define an equivalence relation which enables us to merge columns in the following way. A state in Q is *left-copying* if all words in its prefix consist of pairs of identical symbols. A state in Q is *right-copying* if all words in its suffixes consist of pairs of identical symbols. In the above example, the states q_L and q_R are left- and right-copying, respectively. Now, two columns are *equivalent* if they can be made equal by repeatedly deleting identical neighbours which are either left- or right copying. For instance the columns $q_L q_L x q_R$ and $q_L x q_R q_R$ are equivalent. Applying this to our example, we get the following transitions.

- $q_L \overset{(a,\perp)}{\Longrightarrow} q_1$ and $q_L \overset{(\perp,\perp)}{\Longrightarrow} q_L$ give us $q_L q_L \overset{(a,\perp)}{\Longrightarrow} q_1 q_L$. We merge $q_L q_L$ and q_L.
- $q_1 \overset{(b,a)}{\Longrightarrow} q_2$ and $q_L \overset{(a,\perp)}{\Longrightarrow} q_1$ give us $q_1 q_L \overset{(b,\perp)}{\Longrightarrow} q_2 q_1$.
- $q_2 \overset{(\perp,b)}{\Longrightarrow} q_R$ and $q_1 \overset{(b,a)}{\Longrightarrow} q_2$ give us $q_2 q_1 \overset{(\perp,a)}{\Longrightarrow} q_R q_2$.
- $q_R \overset{(\perp,\perp)}{\Longrightarrow} q_R$ and $q_2 \overset{(\perp,b)}{\Longrightarrow} q_R$ give us $q_R q_2 \overset{(\perp,b)}{\Longrightarrow} q_R q_R$. We merge $q_R q_R$ and q_R.

The new transducer thus becomes:



3 Algorithm

In this section, we will formally present our algorithm. In particular, we define an equivalence relation on the set Q^+ of columns of T^+, which can be used to merge columns during the computation of R^+. The equivalence relation is not just language equivalence: equivalent columns may in general have different left quotients or right quotients. Instead, we shall prove that merging equivalent columns of T^+ does not increase the relation computed by T^+.

Formally, denoting the equivalence relation on Q^+ by $\simeq$, we define the *quotient transducer* $T_\simeq$ as $T_\simeq = \langle Q^+/\simeq, \{q_0\}^+, \Longrightarrow_\simeq, F^+/\simeq\rangle$ where

- $Q^+/\simeq$ is the set of equivalence classes of columns,
- q_0^+ is the initial equivalence class (this will indeed be one equivalence class of $\simeq$),
- $\Longrightarrow_\simeq : (Q^+/\simeq \times (\Sigma \times \Sigma)) \mapsto 2^{Q^+/\simeq}$ is defined in the natural way as follows:

$$X \overset{(a,a')}{\Longrightarrow}_\simeq X' \quad \Leftrightarrow \quad \exists x \in X, x' \in X' : x \overset{(a,a')}{\Longrightarrow} x'$$

- $F^+/\simeq$ is the partitioning of F^+ with respect to $\simeq$ (this will be well-defined since, as we shall see later, there are no columns x and y with $x \simeq y$, $x \in F^+$, and $y \notin F^+$).

Now, we define $\simeq$ as follows.

For a set X of columns, let $\mathsf{pref}(X)$ denote the set of prefixes of X, i.e., the set of words $w \in (\Sigma \times \Sigma)^*$ such that $q_0^+ \stackrel{w}{\Longrightarrow} x$ for some x in X. Analogously, let $\mathsf{suff}(X)$ denote the set of suffixes of X, i.e., the set of words $w \in (\Sigma \times \Sigma)^*$ such that there are $x \in X$ and $y \in F^+$ with $x \stackrel{w}{\Longrightarrow} y$.

A state $q \in Q$ is *left-copying* if all words in $\mathsf{pref}(\{q\})$ are of form $(a_1, a_1) \cdot (a_2, a_2) \cdots (a_n, a_n)$, i.e., containing only pairs of identical alphabet symbols. A state $q \in Q$ is *right-copying* if all words in $\mathsf{suff}(\{q\})$ are of form $(a_1, a_1) \cdot (a_2, a_2) \cdots (a_n, a_n)$. In other words, prefixes of left-copying states only copy input symbols to output symbols, and similarly for suffixes of right-copying states.

The equivalence classes of $\simeq$ will now be sets of form $e_1 e_2 \cdots e_n$ where each e_i is one of the following:

1. $\{q_L\}^+$, for some left-copying state q_L,
2. $\{q_R\}^+$, for some right-copying state q_R,
3. $\{q\}$, for some state q which is neither left-copying nor right-copying,

and where two consecutive e_i can be identical only if they are neither left-copying nor right-copying. In the following, we often omit the set notation, using q to denote the set $\{q\}$. Thus, equivalence classes in $Q^+/\simeq$ can be represented as columns $e_1 e_2 \cdots e_n$ where each e_i is of form q_L^+, q_R^+, or q, avoiding successive duplicates as described above.

Define the operator $\star$ as the natural concatenation operator on equivalence classes:

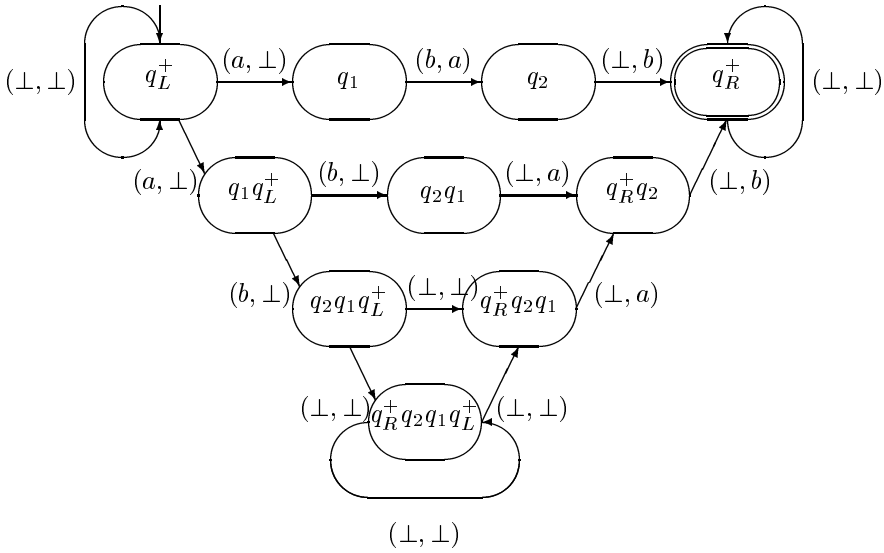
- $e_1 \star e_2$ is defined as
  - e_1 if $e_1 = e_2 = \{q\}^+$ and q is a left- or right-copying state,
  - $e_1 e_2$ otherwise.
- $(E_1 e_1) \star (e_2 E_2)$ is defined as $E_1 (e_1 \star e_2) E_2$ for equivalence classes $E_1 e_1$ and $e_2 E_2$.

Our procedure will now build a sequence $\widetilde{T}_0, \widetilde{T}_1, \widetilde{T}_2, \cdots$ of successively larger (in terms of sets of accepted pairs of words) transducers, all of which underapproximate R^+: the transition relation $\Longrightarrow_i$ of $\widetilde{T}_i$ will be a subset of $\Longrightarrow_\simeq$. The procedure incrementally adds transitions in $\Longrightarrow_\simeq$ between equivalence classes. The initial transducer $\widetilde{T}_0$ is obtained from T by taking all transitions in $\longrightarrow$ and replacing all left-copying states q_L by $\{q_L\}^+$, all right-copying states q_R by $\{q_R\}^+$, and all other states q by $\{q\}$.

In each step of the procedure, $\Longrightarrow_{i+1}$ is obtained from $\Longrightarrow_i$ by adding transitions of form $X \star X' \stackrel{(a,c)}{\Longrightarrow}_{i+1} Y \star Y'$ with $X \stackrel{(a,b)}{\Longrightarrow}_i Y$ and $X' \stackrel{(b,c)}{\Longrightarrow}_0 Y'$.

The algorithm terminates when the relation R^+ is accepted by $\widetilde{T}_i$. This can be tested by checking if the language of $\widetilde{T}_i \circ R$ is included in $\widetilde{T}_i$.

Continuing our example from Section 2, a partial transducer of the algorithm is shown below. Other transitions are also added, but they do not add to the language of the transducer.



4 Correctness

In this section we show correctness (soundness and completeness) of our construction. We do this in two steps. First, we prove (Corollary 1) that $T_{\simeq}$ is equivalent to T^+ in the sense that both transducers accept the same relation on words. Then, we relate the transducer generated by the algorithm in Section 3 to $T_{\simeq}$ proving its soundness (Theorem 3) and completeness (Theorem 4).

We also present sufficient conditions for termination of the algorithm, which implies that our approach is sufficiently general to cover several classes of systems considered in earlier works.

To start with, we need a technical result saying that, since T is deterministic, we can ignore columns containing distinct consecutive left-copying states.

Lemma 1. *Any column x which contains two distinct consecutive left-copying states, i.e., is of form $x = x_1 \cdot q_1 \cdot q_2 \cdot x_2$ where q_1 and q_2 are distinct left-copying states, is unreachable in T^+*

Proof. Follows directly from the fact that T is deterministic, and that the set of initial states of T^+ is q_0^+. □

Lemma 2. *The relation accepted by T^+ remains the same if we remove all columns that contain two distinct consecutive left-copying states. Analogously, the relation accepted by $T_{\simeq}$ remains the same if we remove all equivalence classes that contain two distinct consecutive left-copying states.*

Proof. Follows directly from Lemma 1, and the observation that if some column in an equivalence class contains two distinct consecutive left-copying states, then all columns in this equivalence class will also do so. □

In the rest of this paper, we will thus assume that all columns with two distinct consecutive left-copying states are removed from T^+ and $T_{\simeq}$.

Equivalence of $T_\simeq$ and T^+ We prove equivalence of $T_\simeq$ and T^+ (Corollary 1) by showing (Theorem 1) that the equivalence relation $\simeq$ contains a forward simulation and a backward simulation relation. This ensures that the merging of states will not add prefix/suffix combinations not existing in T^+.

A relation $\leq_F$ on the set of columns is a *forward simulation* if whenever $x \leq_F y$ and $x \overset{(a,b)}{\Longrightarrow} x'$ for some pair (a, b) of symbols and column x', there is a column y' such that $y \overset{(a,b)}{\Longrightarrow} y'$ and $x' \leq_F y'$.

Similarly, a relation $\leq_B$ on the set of columns is a *backward simulation* if whenever $x \leq_B y$ and $x' \overset{(a,b)}{\Longrightarrow} x$ for some pair (a, b) of symbols and column x', there is a column y' such that $y' \overset{(a,b)}{\Longrightarrow} y$ and $x' \leq_B y'$.

Theorem 1. *There is a forward simulation $\leq_F$ and a backward simulation $\leq_B$ with $\leq_F \subseteq \simeq$ and $\leq_B \subseteq \simeq$ such that for all columns x and y with $x \simeq y$, there is some column z such that $x \leq_F z$ and $y \leq_B z$.*

Proof. We must define the forward simulation $\leq_F$ and the backward simulation $\leq_B$ on columns. Let $e_1 e_2 \cdots e_n$ and $f_1 f_2 \cdots f_n$ be two equivalent columns, where for each k, we have either $e_k = f_k = q$ for some state q, or that $e_k = q^i$ and $f_k = q^j$ for some left- or right-copying state q. Define

- $e_1 e_2 \cdots e_n \leq_F f_1 f_2 \cdots f_n$ iff in addition $e_k = f_k$ whenever $e_k = q^i$ for some left-copying state q,
- $e_1 e_2 \cdots e_n \leq_B f_1 f_2 \cdots f_n$ iff in addition $e_k = f_k$ whenever $e_k = q^i$ for some right-copying state q.

We must prove that $\leq_F$ is a forward simulation, and that $\leq_B$ is a backward simulation. Let $x = e_1 e_2 \cdots e_n$ and $y = f_1 f_2 \cdots f_n$ be as above.

$\leq_F$: Assume $x \leq_F y$. If $x \overset{(a,b)}{\Longrightarrow} x'$, then x' is of form $x' = e'_1 e'_2 \cdots e'_n$. We can choose y' as $f'_1 f'_2 \cdots f'_n$, where $f'_k = e'_k$, except when e_k is of the form q^i for a right-copying state q. However, in this case, since T is deterministic, e'_k will be of form q'^i for a right-copying state q', whence we can choose f'_k as q'^j.

$\leq_B$: Assume $x \leq_B y$. If $x' \overset{(a,b)}{\Longrightarrow} x$, then x' is of form $x' = e'_1 e'_2 \cdots e'_n$. We can choose y' as $f'_1 f'_2 \cdots f'_n$, where $f'_k = e'_k$, except when e_k is of the form q^i for a left-copying state q. However, in this case, by lemma 2, e'_k will be of form q'^i for a left-copying state q', whence we can choose f'_k as q'^j.

For each pair $x = e_1 e_2 \cdots e_n$ and $y = f_1 f_2 \cdots f_n$ of equivalent columns, we can now find a z with $x \leq_F z$ and $y \leq_B z$ by taking z as $g_1 g_2 \cdots g_n$, where g_k is

- e_k if $e_k = f_k$,
- $g_k = e_k$ whenever $e_k = q^i$ for some left-copying state q,
- $g_k = f_k$ whenever $e_k = q^i$ for some right-copying state q. □

We are now ready to prove the main theorem of this section, namely that the set of traces of $T_\simeq$ is included in the set of traces of T^+.

Theorem 2. *$T_{\simeq}$ and T^+ have the same set of traces.*

Proof. Notice that since $T_{\simeq}$ is a collapsed version of T^+, it will obviously have more traces. We just need to show the inclusion in the other direction.

We will show that for each sequence of transitions of $T_{\simeq}$

$$X_0 \overset{(a_1,b_1)}{\Longrightarrow_{\simeq}} X_1 \overset{(a_2,b_2)}{\Longrightarrow_{\simeq}} \cdots \overset{(a_{n-1},b_{n-1})}{\Longrightarrow_{\simeq}} X_{n-1} \overset{(a_n,b_n)}{\Longrightarrow_{\simeq}} X_n$$

there is a corresponding sequence of transitions of T^+.

$$x_0 \overset{(a_1,b_1)}{\Longrightarrow} x_1 \overset{(a_2,b_2)}{\Longrightarrow} \cdots \overset{(a_{n-1},b_{n-1})}{\Longrightarrow} x_{n-1} \overset{(a_n,b_n)}{\Longrightarrow} x_n$$

where $x_i \in X_i$ for $i = 0, \ldots, n$. We show this by induction on the length n of the sequence.

•**Base case:** The empty trace is trivially in both $T_{\simeq}$ and T^+.

•**Inductive case:** Assume that the property is true for n. Let $w = w_1 \cdot (a_{n+1}, b_{n+1})$ be a trace of length $n+1$. Then w_1 is a trace of length n, and is thus, by the induction hypothesis, also accepted by T^+ as in the display above. w is accepted in a sequence of transitions

$$X_0 \overset{(a_1,b_1)}{\Longrightarrow_{\simeq}} \cdots \overset{(a_n,b_n)}{\Longrightarrow_{\simeq}} X_n \overset{(a_{n+1},b_{n+1})}{\Longrightarrow_{\simeq}} X_{n+1}$$

meaning that there are $y_n \in X_n$ and $y_{n+1} \in X_{n+1}$ such that $y_n \overset{(a_{n+1},b_{n+1})}{\Longrightarrow} y_{n+1}$. Since $x_n \in X_n$ we have $x_n \simeq y_n$, and hence there is a z_n such that $x_n \leq_B z_n$ and $y_n \leq_F z_n$. From $y_n \leq_F z_n$ we infer that there is a $z_{n+1} \in X_{n+1}$ such that

$$z_n \overset{(a_{n+1},b_{n+1})}{\Longrightarrow} z_{n+1}$$

From $x_n \leq_B z_n$ we infer that there is a sequence

$$z_0 \overset{(a_1,b_1)}{\Longrightarrow} \cdots \overset{(a_n,b_n)}{\Longrightarrow} z_n$$

such that $x_i \leq_F z_i$ for $i = 0, \ldots, n$, implying that $z_i \in X_i$ for $i = 0, \ldots, n$. We can thus conclude that the sequence

$$z_0 \overset{(a_1,b_1)}{\Longrightarrow} \cdots \overset{(a_n,b_n)}{\Longrightarrow} z_n \overset{(a_{n+1},b_{n+1})}{\Longrightarrow} z_{n+1}$$

satisfies the conditions for the inductive step.

□

From this theorem, we can deduce that $T_{\simeq}$ and T^+ accept the same relation.

Corollary 1. *$T_{\simeq}$ and T^+ accept the same relation.*

Proof. We notice that the union of the sets attached to each final state of $T_{\simeq}$ is the set of final columns of T^+ (they form a partition of it w.r.t $\simeq$). Thus we can conclude that any trace that is an accepting run in one automaton is also an accepting run in the other automaton. □

Soundness and Completeness We are now ready to prove the soundness and completeness of the algorithm. For a column x, let $[x]_{\simeq}$ denote the equivalence class for x. We will use the following property of the operator $\star$ to prove the soundness.

Lemma 3. *For any sets X, X' associated with some equivalence classes, we have that $X \cdot X' \subseteq X \star X'$.*

Proof. By observing that $E_1 \cdot q^+ \cdot q^+ \cdot E_2 \subseteq E_1 \cdot q^+ \cdot E_2$. □

The following is the soundness theorem.

Theorem 3. *For every k, $\Longrightarrow_k \subseteq \Longrightarrow_{\simeq}$.*

Proof. For $k = 0$, let X, Y be two sets of columns such that $X \overset{(a,a')}{\Longrightarrow}_0 Y$ for some pair (a, a'). Since $\Longrightarrow_0$ is obtained from T by substituting each state with its equivalence-class, we must have that $X = [q]_{\simeq}$ and $Y = [q']_{\simeq}$ for some states q, q' such that $q \overset{(a,a')}{\longrightarrow} q'$. Thus $X \overset{(a,a')}{\Longrightarrow}_{\simeq} Y$.

Now take $k > 0$ and assume that for all $k' < k$ we have $\Longrightarrow_{k'} \subseteq \Longrightarrow_{\simeq}$. Let X, X', Y, Y' be sets of columns such that $X \overset{(a,b)}{\Longrightarrow}_{k-1} Y$ and $X' \overset{(b,c)}{\Longrightarrow}_0 Y'$. Then we have to show that $X \star X' \overset{(a,c)}{\Longrightarrow}_{\simeq} Y \star Y'$. By induction, we have that $X \overset{(a,b)}{\Longrightarrow}_{\simeq} Y$ and $X' \overset{(b,c)}{\Longrightarrow}_{\simeq} Y'$. Thus, there are $x \in X, y \in Y, x' \in X', y' \in Y'$ such that $x \overset{(a,b)}{\Longrightarrow} y$ and $x' \overset{(b,c)}{\Longrightarrow} y'$, and hence, $x \cdot y \overset{(a,c)}{\Longrightarrow} x' \cdot y'$. Since $x \cdot y \in X \cdot Y$ and $x' \cdot y' \in X' \cdot Y'$, by Lemma 3 we get that $x \cdot y \in X \star Y$ and $x' \cdot y' \in X' \star Y'$, and thus $X \star X' \overset{(a,c)}{\Longrightarrow}_{\simeq} Y \star Y'$. □

The following completeness theorem states that any pair in the transitive closure will eventually be generated by the algorithm.

Theorem 4. *Let (w, w') be a word in R^+. Then there is some k such that $\widetilde{T}_k$ accepts (w, w').*

Proof. Let $x_1, x_2, \cdots, x_n$ be a run of T^+ accepting (w, w'). This run can be organized as columns of the following matrix:

$$\begin{array}{ccccccc}
q_1^1 & \xrightarrow{(a_1^0, a_1^1)} & q_2^1 & \xrightarrow{(a_2^0, a_2^1)} & \cdots & q_{n-1}^1 & \xrightarrow{(a_{n-1}^0, a_{n-1}^1)} q_n^1 \\
q_1^2 & \xrightarrow{(a_1^1, a_1^2)} & q_2^2 & \xrightarrow{(a_2^1, a_2^2)} & \cdots & q_{n-1}^2 & \xrightarrow{(a_{n-1}^1, a_{n-1}^2)} q_n^2 \\
 & & & & \vdots & & \\
q_1^m & \xrightarrow{(a_1^{m-1}, a_1^m)} & q_2^m & \xrightarrow{(a_2^{m-1}, a_2^m)} & \cdots & q_{n-1}^m & \xrightarrow{(a_{n-1}^{m-1}, a_{n-1}^m)} q_n^m
\end{array}$$

where for all i with $1 \leq i < n$ and all j with $1 \leq j \leq m$ we have that $q_i^j \overset{(a_i^{j-1}, a_i^j)}{\longrightarrow} q_{i+1}^j$.

We now prove by induction on the number of rows of this matrix that the pair (w, w') is eventually accepted by the transducer built by the algorithm.

By the definition of $\Longrightarrow_0$ we get that $[q_i^j]_\simeq \overset{(a_i^{j-1}, a_i^j)}{\Longrightarrow}_0 [q_{i+1}^j]_\simeq$, for all i with $1 \leq i < n$ and all j with $1 \leq j \leq m$. Taking $j = 1$, we get that $[q_1^1]_\simeq, [q_2^1]_\simeq, \ldots, [q_n^1]_\simeq$ is a run of $\widetilde{T}_0$ accepting $(a_1^0, a_1^1) \cdot (a_2^0, a_2^1) \ldots (a_{n-1}^0, a_{n-1}^1)$.

Now suppose that in some step i in the algorithm, for some k we have built the transition relation $\Longrightarrow_i$ such that $E_1, E_2, \ldots, E_n$ is a run of $\widetilde{T}_i$ accepting $(a_1^0, a_1^k) \cdot (a_2^0, a_2^k) \cdots (a_{n-1}^0, a_{n-1}^k)$. Then since $[q_1^{k+1}]_\simeq, [q_2^{k+1}]_\simeq, \ldots, [q_n^{k+1}]_\simeq$ is a run of $\widetilde{T}_0$ accepting $(a_1^k, a_1^{k+1}) \cdot (a_2^k, a_2^{k+1}) \cdots (a_{n-1}^k, a_{n-1}^{k+1})$, transitions will be in $\Longrightarrow_{i+1}$ such that $E_1 \star [q_1^{k+1}]_\simeq, E_2 \star [q_2^{k+1}]_\simeq \ldots E_n \star [q_n^{k+1}]_\simeq$ is a run of $\widetilde{T}_{i+1}$ accepting $(a_1^0, a_1^{k+1}) \cdot (a_2^0, a_2^{k+1}) \cdots (a_{n-1}^0, a_{n-1}^{k+1})$. □

Termination The termination of our method is dependent on the number of different equivalence classes of the form $e_1 e_2 \cdots e_n$ which might be generated during construction of the transitive closure. This number in turn depends on two parameters:

1. the number of non-copying states in columns, and
2. the number of alternations of copying states in columns.

Therefore a bound on these two parameters is a sufficient condition for termination.

In [JN00], we introduced a class of systems which satisfied the *bounded local depth* property. Roughly speaking, this property means that there is a bound on the number of times each position in a word is rewritten when applying the transducer to the word an arbitrary number of times. For example, a system passing a token to the right has local depth 2, since each position can be rewritten at most twice; once when passing the token, and once when receiving it. In [JN00], we also assumed that there could only be at most one left-copying state and one right-copying state. For this class of systems, it can be shown that for each pair of words in the transitive closure, there is a run of the column transducer having at most two alternations of the left-copying state and the right-copying state. Thus, our construction will also terminate under the condition of bounded local depth.

In [Cau00], a class of rewriting systems is considered which has the *right-overlapping* (or symmetrically the *left-overlapping*) property. This means that all rewritings must occur in strict order from the left to the right. In such a case, our construction gives columns of the form $q_R^+ q_1 q_2 \cdots q_n q_L^+$ where q_L is a left-copying state, q_R is a right-copying state and each q_i is some state representing a rewriting. This implies that the number of alternations of copying states is at most one, and that the number of non-copying states is bounded.

5 Implementation

We have implemented the technique in this paper and run it on a number of mutual exclusion and termination detection protocols. We have compared the performance of the algorithm with our earlier work [BJNT00].

The technique in [BJNT00] is based on applying subset construction to the column transducer and on-the-fly identification of equivalent (w.r.t. suffixes) states. The subset construction technique represents sets of states (columns) by finite-state automata and involves several operations on regular sets, such as

- computing post-images of sets of columns represented by finite-state automata,
- saturating generated sets of columns, and
- testing saturated sets for equality against all previous sets.

All the above operations can potentially be expensive. In contrast, our new algorithm represents the equivalence classes as vectors of states. The concatenation operator is a variant of concatenation of vectors, and equivalence checking of vectors is fast and can be hashed effectively.

Furthermore, we have implemented an obvious optimization to our new method to avoid generating useless states, namely, in each step, we only merge transitions where $x \stackrel{(a,b)}{\Longrightarrow} x'$ and $y \stackrel{(b,c)}{\Longrightarrow} y'$ only if all x, x', y, y' are both reachable and productive in the transducer obtained in the previous step. Using this technique, we substantially reduce the number of generated useless states.

We have used both methods to compute the transitive closures of a set of transitions in some algorithms. It should be mentioned that it is not clear whether all operations in the subset construction method are implemented in the most efficient way, since there are many ways to represent automata. Nevertheless, the initial experiments indicate that the new method runs several times faster. Furthermore, there are a large number of potential optimizations which have still not been carried out.

The results are shown in Table 1.

Algorithm	Subset construction	Matching	Speedup
Dijkstra	435s	39s	11.2
Szymanski	278s	178s	1.5
Termination detection	47s	22s	2.1
Ticket	17s	20s	0.85

Table 1. Improvements of new method compared to subset construction method

For the ticket algorithm, the new algorithm performed slightly worse. The ticket algorithm is a rather small example, and it seems that the new method scales better than the old one. We expect to be able to improve the new method

by considering different ways of scheduling the matching operations. Also, it may be possible to find ways to remember already tried combinations to avoid repeated work.

6 Conclusions and Future Research

We have presented a new technique for performing regular model checking. More precisely, given a finite-state transducer, our algorithm generates a new transducer corresponding to the transitive closure of the original one. The algorithm involves two ingredients, namely a matching operation which combines existing transitions to add new ones, and an equivalence relation which enables us to merge states. An important property of the equivalence relation is that it is syntactically characterized and hence possible to decide locally.

A crucial aspect in the application of the algorithm is the order in which the matching operation is performed on transitions. By defining appropriate matching strategies, we believe that our algorithm can be made both to uniformly simulate existing algorithms for parameterized protocols [JN00,BJNT00], rewriting systems [Cau00], push-down systems [BEM97,Cau92,FWW97], etc, and to produce more efficient versions of these algorithms. Furthermore, we think that the generality of construction will enable us to extend the algorithm to other classes of relations than those on words, e.g., relations on trees and graphs. This would allow us to verify systems with dynamic behaviours such as data security protocols, mobile protocols, etc.

References

[ABJ98] Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *Proc.* 10^{th} *Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 305–318, 1998.

[ABJN99] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parameterized system verification. In *Proc.* 11^{th} *Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145, 1999.

[BCMD92] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model Checking. In *Proc. Intern. Conf. on Concurrency Theory (CONCUR'97)*. LNCS 1243, 1997.

[BFL] B. Boigelot, J-M. Franois, and L. Latour. The Liége automata-based symbolic handler (lash). Available at `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In Alur and Henzinger, editors, *Proc.* 8^{th} *Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer Verlag, 1996.

[BGWW97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *Proc. of the Fourth International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer Verlag, 1997.

[BH97] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of fifo-channel systems with nonregular sets of configurations. In *Proc. ICALP '97, 24^{th} International Colloquium on Automata, Lnaguages, and Programming*, volume 1256 of *Lecture Notes in Computer Science*, 1997.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla, editors, *Proc. 12^{th} Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418, 2000.

[BMT01] A. Bouajjani, A. Muscholl, and T. Touili. Permutation rewriting and algorithmic verification. In *Proc. LICS' 01 17^{th} IEEE Int. Symp. on Logic in Computer Science*. IEEE, 2001.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc. 6^{th} Int. Conf. on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer Verlag, 1994.

[Cau92] Didier Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1):61–86, Nov. 1992.

[Cau00] Didier Caucal. On word rewriting systems having a rational derivation. In *FOSSACS 2000*, volume 1784 of *Lecture Notes in Computer Science*, pages 48–62, April 2000.

[CJ98] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In *CAV'98*. LNCS 1427, 1998.

[DLS01] D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, 2001.

[ES01] J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *Proc. 13^{th} Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336, 2001.

[FWW97] A. Finkel, B. Willems, , and P. Wolper. A direct symbolic approach to model checking pushdown systems (extended abstract). In *Proc. Infinity'97, Electronic Notes in Theoretical Computer Science*, Bologna, Aug. 1997.

[HJJ+96] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proc. TACAS '95, 1^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, 1996.

[JN00] Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS '00, 6^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, 2000.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proc. 9^{th} Int. Conf. on Computer Aided Verification*, volume 1254, pages 424–435, Haifa, Israel, 1997. Springer Verlag.

[KMMG97] P. Kelb, T. Margaria, M. Mendler, and C. Gsottberger. Mosel: A flexible toolset for monadic second–order logic. In *Proc. of the Int. Work-*

shop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), Enschede (NL), volume 1217 of *Lecture Notes in Computer Science (LNCS)*, pages 183–202, Heidelberg, Germany, March 1997. Springer–Verlag.

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *Proc. 12^{th} Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 328–343, 2000.

[Tou01] T. Touili. Regular Model Checking using Widening Techniques. *Electronic Notes in Theoretical Computer Science*, 50(4), 2001. Proc. Workshop on Verification of Parametrized Systems (VEPAS'01), Crete, July, 2001.

[WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97, Vancouver, July 1998. Springer Verlag.

Performance Analysis of Multiprocessor Schedules of Tasks with Stochastic Execution Times

Extended Abstract

Sorin Manolache, Petru Eles, Zebo Peng
Linköping University
{sorma, petel, zebpe}@ida.liu.se

1 Introduction

Systems controlled by dedicated electronic systems become indispensable in our lives and can be found in avionics, automotive industry, home appliances, medicine, telecommunication industry etc. [3]. Due to the application nature itself, as well as due to the high demands in terms of computation power and constraints such as cost and energy dissipation, these systems are mainly custom designed heterogeneous multiprocessor platforms. Their high complexity makes the design process a challenging activity. An accurate and efficient design process is hence the key to cope with the high time-to-market pressure. The enormous design space implies that accurate performance estimation tools in all design stages are of capital importance for guiding the designer and reducing the design cost and iterations.

The present work focuses on an analytic approach to performance estimation of multiprocessor real-time systems. We consider applications as sets of task graphs with the tasks statically mapped on a set of processors. Most of the work in this area carries out the analysis in the worst case of the task execution times (WCET) [4][13] and provides answers whether the tasks will meet or not all of their deadlines. Such an approach is well suited for critical applications. However, it leads to expensive and inefficient implementations in the case of other application classes like soft real-time systems and multimedia applications. The latter occupy an important market share. According to Tia [14], applications have been reported where the WCET is 145% of the average one and the worst case appears rarely.

Our work considers the case when the task execution times are of stochastic nature and their probabilities are distributed according to given arbitrary continuous time probability distribution functions. Because meeting a deadline in this case becomes also a stochastic event, our method outputs the probability of task deadline misses.

The variability of the task execution time may stem from several sources: input data characteristics (especially in differently coded video frames), hardware architecture hazards (caches and pipelines), environmental factors (network load), or insufficient knowledge regarding the design (task running on a not yet manufactured processor, for example).

Leung and Whitehead [9] showed that the schedulability analysis is an NP-complete problem in the case of fixed task execution times and more than two processors. Obviously, the problem is more challenging in the case of stochastic task execution times.

The sequel of the paper is organized as follows: The next section surveys some related approaches. Section 3 formalizes the problem. Section 4 presents the approach outline. Each of the points in the approach is detailed in the following sections. The last section draws the conclusions.

2 Related work

Lehoczky has pioneered the “heavy traffic” school of thought in the area of real-time queueing [7][8]. The theory was later extended by Harrison [5], Williams [15] and others. The application is modelled as a multiclass queueing network. This network behaves as a reflected Brownian motion with drift under heavy traffic conditions, i.e. when the processor utilizations approach 1, and therefore it has a simple solution. This approach, to our knowledge, fails yet to handle systems where a task has more than one immediate successor task. Moreover, the heavy traffic assumption implies an almost infinite queue in the case of input distributions with non-negligible variation. This leads to an unacceptably high deadline miss ratio and limits the applicability of such an approach in real-time systems.

Manolache et al. [10] considered monoprocessor systems. The analysis is based on solving a generalized semi-Markov process by means of the auxiliary variable

method. Although they concurrently construct and analyse the process, saving a significant amount of memory, the method is of limited applicability to multiprocessor systems due to the exploding complexity.

Shin et al. [6] also modelled the application as a queueing network, but restricted the task execution times to exponentially distributed ones for ease of analysis purposes. The tasks were considered to be scheduled according to a FIFO policy. The underlying mathematic model is then the appealing continuous time Markov chain (CTMC).

Our approach is also based on a CTMC in order to keep the appealing character of the solution procedure and to avoid the time and memory consuming convolutions implied by solving the otherwise generalized semi-Markov process. We consider the tasks to be scheduled according to a fixed priority non-preemptive policy. We overcome the limitation of the exponentially distributed execution time probabilities by approximating arbitrary real-world distributions by means of Coxian distributions, i.e. weighted sums of convoluted exponentials. The resulting CTMC is huge, but because of regularities in its construction, its infinitesimal generator needs not be stored explicitly, making the method applicable to real applications. The infinitesimal generator appears as a sum of Kronecker products of simple matrices.

Plateau and Fourneau [11] address the problem of constructing the global infinitesimal generator of a CTMC modelling a concurrent system from the local infinitesimal generators of the system components. The execution times are still considered exponential. Although we address a different problem, our approach is related to theirs by using a generator given as a sum of Kronecker products of matrices.

3 Problem formulation

Informally, given a set of task graphs where the task execution time probabilities are distributed according to given arbitrary continuous distributions, the analysis outputs the task deadline miss ratios. Formally, the input to the analysis procedure consists of:

- a set TG of task graphs TG = $\{g_1, ..., g_G\}$, formed of the tasks in the set T = $\{t_1, ..., t_N\}$
- a set of processors P = $\{p_1, p_2, ..., p_s\}$
- a surjective mapping Map : T $\rightarrow$ P
- a set of continuous execution time probability distribution functions (ETPDF) E = $\{e_1, ..., e_N\}$, e_i : $[0{:}\infty) \rightarrow \Re$, statistically independent
- a set of periods, A = $\{a_1, a_2..., a_G\}$
- a set of fixed deadlines, deadline equals period
- a fixed priority non-preemptive scheduling policy, Prior : T $\rightarrow$ $\aleph$
- the number of late task graph instantiations allowed in the system, B = $\{b_1, ..., b_G\}$. If $b_i = 0$ then the late instantiations of task graph i are discarded

The analysis outputs the deadline miss ratios of the task graphs, F = $\{f_1, f_2, ..., f_G\}$

4 Approach outline

The underlying mathematical model of the application to be analysed is the stochastic process. The process has to be constructed and analysed in order to extract the desired performance metrics. The task execution times correspond to holding times in the states of the process. When considering arbitrary ETPDFs, the resulting process is a generalized semi-Markov process, making its analysis demanding in terms of memory and time. If the execution time probabilities were exponentially distributed, as assumed for instance by Shin, the process would be a CTMC.

As a first step in our approach, we approximate the arbitrary ETPDFs with Coxian distributions, i.e. weighted sums of convoluted exponentials. This ensures that we still deal with a CTMC. This step is detailed in the next section.

We imagine an application that differs from the one to be analysed by having exponentially distributed execution time probabilities instead of the real ones. The next step is to construct the underlying Markov chain C of the imagined application and to obtain its infinitesimal generator matrix Q. This is done by modelling the application as a Generalized Stochastic Petri Net and constructing its underlying CTMC by using Balbo's algorithm [1].

The third step is to get the generator matrix M of the CTMC resulting from C when considering the Coxian distributions obtained in the first step instead of the imagined exponential ones. The construction is detailed in Section 7.

As a last step, the chain with matrix M is solved with one of the available numerical iterative methods and the performance metrics extracted.

5 Coxian distribution approximation

Coxian distributions were introduced by Cox [2] in the context of queueing theory and we will use its specific terminology. A graphical representation of a Coxian distribution with four stages appears in Figure 1. Imagine a "customer" approaching the Coxian "service station" from the left. The ovals stand for stages with exponentially distributed "service time" probabilities. The service rate (inverse of the average service time) of stage i, $1 \le i \le r$, is μ_i. Upon leaving the stage i, the customer leaves the entire Coxian station with probability α_i or proceeds to the next stage with probability $1-\alpha_i$. The stages are invisible from outside the Coxian station, i.e. no other customer may enter the first stage if another customer has not yet left any

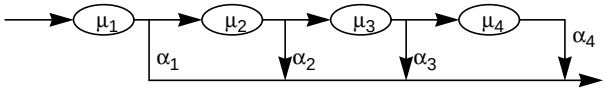


Figure 1 Coxian distribution

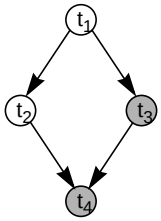


Figure 2 Task graph

of the stages.

The Laplace transform X(s) of the probability density function of a Coxian distribution with r stages is given by the following formula:

$$X(s) = \sum_{i=1}^{r} \alpha_i \cdot \prod_{k=1}^{i-1} (1-\alpha_k) \cdot \prod_{k=1}^{i} \frac{\mu_k}{s+\mu_k}$$

X(s) is a strictly proper rational transform, implying that the Coxian distribution may approximate a fairly large class of arbitrary distributions with an arbitrary accuracy provided a sufficiently large r. The solution to the approximation problem means finding μ_i, $i=\overline{1,r}$, and α_i, $i=\overline{1,r-1}$ ($\alpha_r=1$) such that an error function is minimized. This is done in the complex space by minimizing the distance between the Fourier transform X(jω) of the Coxian distribution and the computed Fourier transform of the distribution to be approximated. The minimization is a typical interpolation problem and can be solved by various numerical methods [12].

6 Application Modelling

The direct algorithmic generation of the underlying stochastic process, though possible, does not give too much insight in the properties of the stochastic process that might be exploited in the analysis. We consider a visual formalism like the Generalized Stochastic Petri Nets (GSPN) an appropriate intermediate representation in the analysis process. Its tangible reachability graph corresponds to the underlying stochastic process.

We illustrate the construction of the GSPN based on an example. Let us consider the task graph in Figure 2. Tasks t_1 and t_2 are mapped on processor p_1 and tasks t_3 and t_4 on processor p_2. The mutual exclusion of the execution of tasks mapped on the same processor is modelled by means

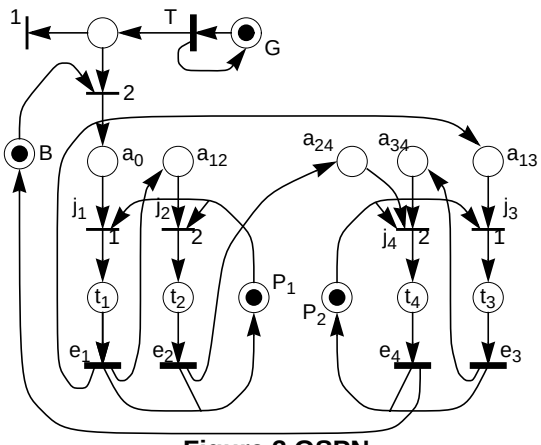


Figure 3 GSPN

of the places P_1 and P_2. The task execution is modelled by means of the timed transitions e_1, ..., e_4. The task priorities are modelled by prioritizing the immediate transitions j_1, ..., j_4. This results in having a Petri Net where the firing of a timed transition in a given marking leads to exactly one next marking, which can be unambiguously determined. In the underlying stochastic process, there will be at most one transition labelled with a task in a given process state. Another important property that is easily detected in the Petri Net is that the net does not contain competitively enabled transitions. Consider the states S_1 and S_2 in the underlying stochastic process and the corresponding sets of tasks, W_1 and W_2 that run in the two states. If, by firing a transition e_k (executing a job of task t_k), the state changes from S_1 to S_2, then $W_1 \setminus \{t_k\} \subseteq W_2$.

The continuous new generation of jobs is modelled by means of place G and transition T, a timed transition with the firing delay equal to the greatest common divisor of task graph periods. The initial marking of place B indicates the tolerated maximum number of late jobs in the system and is controllable by the designer.

7 Analysis

This section details how to construct the infinitesimal generator M of the CTMC with the Coxian distributions starting from the infinitesimal generator Q of the CTMC C where all the execution times probabilities are assumed to be purely exponentially distributed.

The events that may trigger a state transition in C are the arrivals and departures of task instantiations (jobs). We group the states of C in clusters. Two states belong to the same cluster if the same set of events may trigger transitions out of the states. Let R_i denote the set of tasks mapped on processor i, $R_i = \{t : Map(t) = p_i\}$, $i = \overline{1,s}$. The maximum number of clusters is then,

$$\prod_{i=1}^{s} (|R_i| + 1)$$

where $|R_i|$ is the cardinality of R_i. In the worst case, the number of clusters is

$$\left(\frac{N}{s} + 1\right)^s$$

but in reality it is smaller because some tasks may not run concurrently due to data dependencies even if they are mapped on different processors.

The rows and columns of the Q matrix are then shuffled such that the states in the same cluster are consecutively numbered.

Consider an application with four independent tasks, each mapped on a different processor. Figure 4 depicts the matrix Q in such case. The rows and columns in the figure do not correspond to individual rows and columns in Q, but to *clusters* of states. The row labelled with the binary number 1101, for example, indicates that the tasks 1, 3, and 4 are running in the states belonging to the cluster. The shaded cells indicate those submatrices that may contain

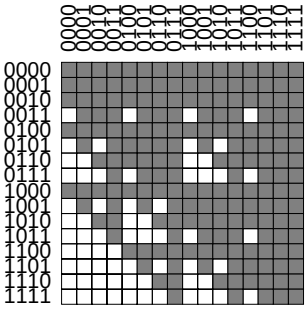


Figure 4 The Q matrix

non-zero elements. The blank ones are null submatrices. One such null submatrix appears for example at the intersection of row 1101 and column 1000. Due to the non-preemption assumption, a task arrival or departure event may not stop the running of another task. If the submatrix (1101, 1000) had non-zero elements it would indicate that an event in a state where the tasks 1, 3, and 4 are running triggers a transition to a state where only the task 4 is running and *two* of the previously running tasks are not running anymore. This is not possible in the case of non-preemption. In the case of k tasks, there are at most $3^{k-1}(k+3)$ non-zero submatrices out of 4^k.

When using the Coxian approximation, a set of new states is introduced for each state in C. We will illustrate the construction of a cell in M from a cell in Q based on an example. We consider a cell on the main diagonal as it is the most complex case.

Figure 5 depicts three states in C. Two tasks, u and v, are running in the states X and Y. These two states belong to the same cluster, 11. Only task v is running in state Z. State Z belongs to cluster 10. If task v finishes running in state X, a transition to state Y occurs in C. When task u finishes running in state X, a transition to state Z occurs in C. Consider that the probability distribution of the execution time of task v is approximated with a three stage Coxian distribution and that of u with a two stage Coxian distribution. The two approximations are depicted in Figure 5 and their arcs were labelled. The resulting stochastic process is depicted in Figure 6. The labels on the transitions in Figure 6 indicate which transition inside the Coxian distribution triggers the particular transition in the expanded Markov chain. Each state in C is replaced by a set of states in the expanded chain and appears in a shaded background in Figure 6. The number of states in such a set is given by the expression

$$\prod_{i \in E} r_i$$

where E is the set of tasks running in the state to be expanded, and r_i indicates the number of stages in the corresponding Coxian distribution. We construct the cell

Figure 5 Part of C

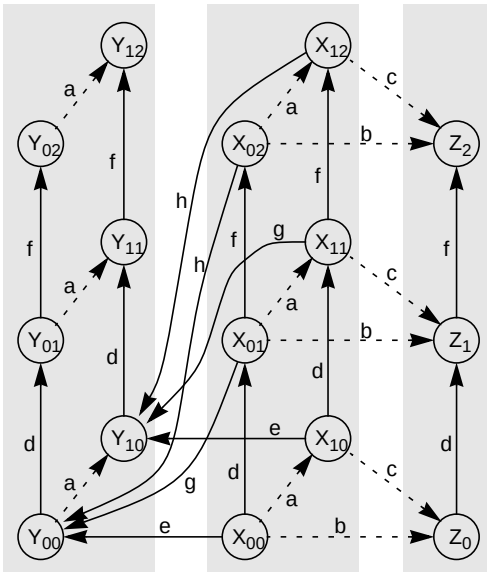


Figure 6 Expanded Markov chain

on the main diagonal, on the row and column of corresponding to the cluster 11, composed of the states X and Y. The observed regularity in the interconnection structure of the set of states is reflected in the expression of the incidence matrix of the cluster 11 as shown in Figure 8. This matrix can be compactly written as

$$(A_u \oplus A_v) \otimes I_{|11|} + I_{r_u} \otimes B_v \otimes e_{r_v} \otimes D_v$$

where

$$A_v = \begin{bmatrix} 0 & d & 0 \\ 0 & 0 & f \\ 0 & 0 & 0 \end{bmatrix}, B_v = \begin{bmatrix} e \\ g \\ h \end{bmatrix}, e_{r_v} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}, D_v = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

and A_u and B_u are similarly defined, |11| denotes the size of the cluster 11 and $\oplus$ and $\otimes$ are the Kronecker sums and products for matrices. In order to obtain a cell in the generator, the labels in the matrices above have to be replaced with the corresponding transition rates. Thus

$$d \leftarrow (1-\alpha_{v1}) \cdot \mu_{v1},\ f \leftarrow (1-a_{v2}) \cdot \mu_{v2},$$
$$e \leftarrow \alpha_{v1} \cdot \mu_{v1},\ g \leftarrow \alpha_{v2} \cdot \mu_{v1},\ h \leftarrow \mu_{v3}$$

In general, a matrix $A_k=[a_{ij}]$ is a $r_k x r_k$ matrix, and is defined as follows:

$$a_{ij} = \begin{cases} (1-\alpha_{ki}) \cdot \mu_{ki} & j = i+1 \\ 0 & j \neq i+1 \end{cases}$$

A $B_k=[b_{ij}]$ matrix is a $r_k x 1$ matrix and $b_{i1}=\alpha_{ki}\mu_{ki}$. An

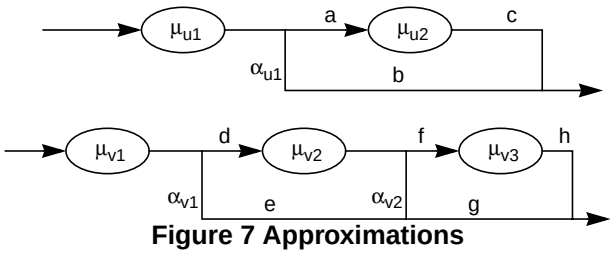


Figure 7 Approximations

	X_{00}	Y_{00}	X_{01}	Y_{01}	X_{02}	Y_{02}	X_{10}	Y_{10}	X_{11}	Y_{11}	X_{12}	Y_{12}
X_{00}		e	d				a					
Y_{00}				d				a				
X_{01}		g			f				a			
Y_{01}						f				a		
X_{02}		h									a	
Y_{02}												a
X_{10}								e	d			
Y_{10}										d		
X_{11}								g			f	
Y_{11}												f
X_{12}								h				
Y_{12}												

Figure 8 Incidence matrix for cluster 11

$e_{rk}=[e_{ij}]$ matrix is a $1xr_k$ matrix and $e_{11}=1$, $e_{1i}=0$, $1<i\le r_k$. A matrix $D_k=[d_{ij}]$ corresponding to a cluster U of size |U| is a |U|x|U| matrix defined as follows:

$$d_{ij} = \begin{cases} 1 & \text{an arc labelled with k leads from i to j in C} \\ 0 & \text{otherwise} \end{cases}$$

Because the cell belongs to the main diagonal, some correction elements have to be introduced such that the entire generator is truly a stochastic matrix (the sum of elements on the rows is 0). The formula for computing a cell on the main diagonal corresponding to a cluster U is given below:

$$(\otimes_{i\in U} I_{r_i}) \otimes (Q_U - \sum_{i\in U} \mu_i D_i) + \oplus_{i\in U} A_i' + \sum_{i\in U} (\otimes_{\substack{j\in U \\ j<i}} I_{r_j}) \otimes B_i \otimes e_{r_i} \otimes (\otimes_{\substack{j\in U \\ j>i}} I_{r_j}) \otimes D_i$$

where μ_i is the transition rate in the artificially constructed chain C and Q_U is the restriction of Q to the states in U.

The matrix $A_k'=[a_{ij}']$ is derived from $A_k=[a_{ij}]$ as follows:

$$a'_{ij} = \begin{cases} a_{ij} & i \neq j \\ \mu_k - \mu_{ki} & i = j \end{cases}$$

We will discuss the construction of the cell at the intersection of the row corresponding to cluster U with the column corresponding to cluster V, U being different from V (cell not on the main diagonal). In this case, the cell will be of the following form:

$$\sum_{i\in U} (\otimes_{j\in V\cup\{i\}} F_{ij}) \otimes D_i$$

The matrices F are given by the following expression:

$$F_{ij} = \begin{cases} e_{r_j} & j \notin U \\ I_{r_j} & j \in U \wedge j \neq i \\ B_i & j \notin V \wedge j = i \\ B_i \otimes e_{r_i} & j \in V \wedge j = i \end{cases}$$

Hence, we may express the generator M of the expanded Markov chain in terms of the generator Q of C, A_i, B_i, and D_i, $1\le i\le N+1$ (+1 for the transition T in Figure 3). All these matrices are both sparse and of small size. Having M as a sum of Kronecker products of matrices allows us not to store M explicitly in memory while performing the analysis.

8 Conclusions and future work

We presented an approach to performance analysis of multiprocessor schedules of tasks with variable execution times. The real-life probability distributions of the execution times are approximated with Coxian distributions, and the expanded underlying Markov chain is constructed in a memory efficient manner exploiting the structural regularities of the chain.

The future work attempts at finding an analytic expression for the number of non-null elements in M, as they determine the analysis speed. Experiments have to be run in order to assess the size of the problems that can be treated and the possible trade-offs between problem size, and approximation accuracy.

References

[1] G. Balbo, G. Chiola, G. Franceschinis, G. M. Roet “On the Efficient Construction of the Tangible Reachability Graph of Generalized Stochastic Petri Nets”, Proc 2nd Workshop on Petri Nets and Performance Models, pp. 85-92, 1987

[2] D.R. Cox “A Use of Complex Probabilities in the Theory of Stochastic Processes”, Proc. Cambridge Philosophical Society, pp. 313-319, 1955

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

[4] C. Fidge “Real-Time Schedulability Tests for Pre-emptive Multitasking”, Real-Time Systems, 14, 61-93 (1998)

[5] J.M. Harrison, V. Nguyen “Brownian Models of Multiclass Queueing Networks: Current Status and Open Problems”, Queueing Systems 13 (1993) 5-40

[6] J. Kim, K.G. Shin, “Execution Time Analysis of Communicating Tasks in Distributed Systems”, IEEE Trans. on Computers, 45 No. 5, May 1996

[7] J.P. Lehoczky “Real-Time Queueing Theory”, Proc. of the 17th IEEE RTSS (1996)

[8] J.P. Lehoczky “Real-Time Queueing Network Theory”, Proc. of the 18th IEEE RTSS (1997)

[9] J.Y.-T. Leung, J. Whitehead “On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks”, Performance Evaluation, 2(4):237-250, Dec. 1982

[10] S. Manolache, P. Eles, Z. Peng “Memory and Time Efficient Schedulability Analysis of Task Sets with Stochastic Execution Time”, Euromicro Conf. on RTS (2001).

[11] B. Plateau, J-M. Fourneau “A Methodology for Solving Markov Models of Parallel Systems”, J. of Parallel and Distributed Computing, 12, 1991

[12] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, “Numerical Recipes in C”, Cambridge Univ. Press, 1992

[13] J. A. Stankovic, M. Spuri, M. Di Natale, G. Butazzo, “Implications of Classical Scheduling Results for Real-Time Systems”, IEEE Computer, June 1995

[14] 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

[15] R.J. Williams “Diffusion Approximations for Open Multiclass Queueing Networks: Sufficient Conditions Involving State Space Collapse”, Queueing Systems 30 (1998) 27-88

Graduate Student Conference 2002

Static Worst-Case Exeuction Time (WCET) Analysis

Andreas Ermedahl
Uppsala University

Static Worst-Case Exeuction Time (WCET) Analysis has been researched for more than ten years, at it finally seems that the technology is ready for deployment in real life. Or is it? In this talk, we will give a short overview of the technology and describe the capabilities of WCET analysis, what can and what can not be analyzed in the current state of the art. Furthermore, we will discuss how the current state of the art matches up with the requirements of potential users and customers. This information is based on a number of informal discussions over the part year with WCET researchers and potential users in industry, and has not been presented previously.

Invitation, Schedule, Participants, Presentations

Updated: 11-Apr-2002 10:43 by Roland Grönroos
e-mail: artes@docs.uu.se web: www.artes.uu.se
Location: http://www.artes.uu.se/events/gsconf02/papers/ebbe.shtml

senseboard TECHNOLOGIES AB

Images

The Product

Senseboard™ Virtual Keyboard (VK) is an exciting new product, designed for the millions of mobile computer users, struggling with their tiny or nonexistent keyboards when trying to communicate or type.

Senseboard™ VK is perfect for businessmen typing e-mails and other documents away from the office. The technology is completely silent allowing e.g. journalists typing articles during a press conference or an interview. The typing speed matches that of standard keyboards allowing quick chatting away from the desktop.

Senseboard™ VK is typically used together with a PDA, a Smart Phone or a Wearable Computer with a Head Mounted Display. The VK hand mounted devices allow the user to type on any surface as if it were a keyboard. Sensors in the units measure the finger movements and artificial intelligence and a language processor determine appropriate keystrokes or mouse movements.

Senseboard™ VK communicates by radio or wire with the PDA, Smart Phone or Wearable Computer.

Senseboard™ is protected by patent pending.

Senseboard Technologies AB | Tel: +46 8 477 98 61 | Fax +46 8 477 98 59 | in

ROBOT FOOTBALL CLUB UPPSALA 2002

UPPSALA UNIVERSITY

The idea of machines looking and acting like humans is not new. As early as two thousand years ago, philosophers imagined creatures, made in bronze, who made life easier for us. We have not come that far yet, despite the time that has passed. The philosophical aspect has been discussed back and forth ever since, though. Can we really create beings to serve us? Can they become more intelligent than humans?

The last question may need clarification. Will computers be better than us at backgammon? Already done; the world's best backgammon player is a computer. Will computers be better than us at chess? Probably, the chess programs of today are almost on par with the best players in the world. Will computers be better than us at football? Maybe, we don't know yet, but the will and resources to create such machines exist.

One may ask whether it is intelligent behavior to develop such machines, but that is mostly a philosophic question. We believe it is.

Why football? Why should machines run around kicking a ball? Why not, we wonder. Football represents all the intricate features we humans are good at; at a physical level: balance, co-ordination, and movement; at an intellectual level: perception of surroundings and opponents, strategy, fast decisions and calculation of risks; at an emotional level; motivation, joy and excitement.

The practical importance of these machines should not be underestimated. Every year a number of disasters occur, where rescue workers put their life on the line. What would be better than a group of robots that come to the rescue, without neither physical nor mental weaknesses, and therto replaceable?

Do you have any thoughts or perhaps you are just curious? Don't hesitate to contact us, we are glad to answer your questions.

www.rfcuppsala.org

1. A robot may not injure a human being, or, through inaction, allow a human being to come to harm.
2. A robot must obey the orders given it by human beings, except where such orders would conflict with the First Law.
3. A robot must protect its own existence, except where such protection would conflict with the First or Second Law.

The Robot Laws, by Isaac Asimov

Humanoid

Means human shaped. Murphy is an humanoid, since he vagely resembles us and walks on two legs, kicks a ball, and so on. Mannequin dummys cannot walk and are thus not humanoids.

Artificial neural networks, ANN

Computers of today perform well in the areas we humans are inefficient, like calculations and text storage. On the other hand, they are bad at everything we can – conclutions, recognition of familiar faces, walking on to legs, etc. This is of course unacceptable.

Computers are naturally inadequate, not letting themselves be programmed to solutions easily. It is of course possible to program a computer to let it solve a specific situation, but the moment that situation changes the computer is again totally incapable of solving the problem. This is where the neural networks enter the scene.

By interconnecting an amount of artificial neurons, it is possible to mimic the functionality of our brains, and make a computer learn just like us. Not entirely like us, since an human brain is infinitely complex, but a little.

Simulator

Reality has a major drawback: it is not customizable. We cannot delete inconveniences like strong wind or limited space. A simulator is a computer program designed to remedy this.

What should a leg look like to let a robot walk up or down stairs? The method of trial and error can be both expensive with all adjustments, and hazardous should the robot lose its balance. A simulator removes all these risks. Assuming it works correctly, of course.

Physics engine

If you have ever played games like Quake and Duke Nukem, you have probably noticed how objects are affected by gravity, that firearms recoil, wooden crates bob in water, etc. All these features are controlled by the physics engine, which is a part of the simulator.

If our simulated Murphy stumbles, he should fall. If a football robot brakes suddenly, it should skid, thanks to the physics engine.

PROJECTS

Games and athletics have always been a part of human culture as long as we have had culture. Since time immemorial, we have been constructing game rules, sports facilities, game strategies, tournaments, and elaborate ways of cheating. Nowadays, we are well on our way of constructing players, too.

Spokesmen from Sony suggest that by the year 2047, we will have robot players that exceed the human original. Whether the four football players in the **Argus** project have reached that far by then remains uncertain.

The literal, romanticized robots have, with few exceptions, walked on two legs. It is without question an advantage to be able to walk the same way as us humans in an human environment, especially since we would want it to aid us in our everyday life.

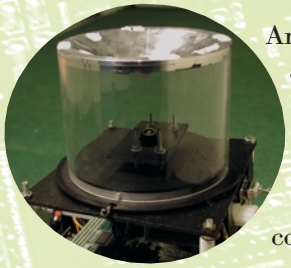
So far, robots have usually been equipped with wheels, for obvious reasons, although biped robots are becoming more common. Our own two-legged fellow **Murphy** has already taken a few tottering steps, and by the summer we plan to have him* walking on his own.

No body is complete without a head. Since Murphy should be able to move according to his surroundings, and since the natural place for eyes and ears is as high up as possible, it is logical that we equip him with a head. Even if Murphy's head **Murray** is supposed to talk, it remains to see if he has something useful to say.

*) He is a man because he is hardheaded, inflexible, and prefers to just hang around.

ARGUS

Argus is a little cube shaped box on three wheels that, together with three other Argus robots, will learn to play football. In June this year, Argus will participate in the robot football world cup, which takes place in Fukuoka, Japan, this year. Argus' parents are very hopeful and aim for the top three places in the tournament. It is still a lot that have to be done in order to make the robots play as a team. They are a little selfish at the moment and do not like to pass the ball when they get hold of it.



Argus has got a lot of eyes and ears. Besides looking forward using a video camera, the robot can see what is going on around it. That is possible using a cone shaped mirror placed at the top of the robot. In order not to run into other players or the goalposts, it keeps track of nearby objects using infrared light. Therefore, one cannot place an Argus robot in front of a TV since the remote control would make the robot go wild.

Like a bat, Argus sends out ultrasound in order to receive a three dimensional image of what is around it. This makes it possible to differentiate the orange coloured ball from other objects looking like the ball, like commercials on the side of the playfield.

Unlike previous years, there are no sideboards surrounding the playfields this year. Therefore, Argus needs to be able to recognize the border lines. This problem, and many others, remain to be solved before the parents of Argus can raise the world cup trophy on June 26th in Fukuoka, Japan.

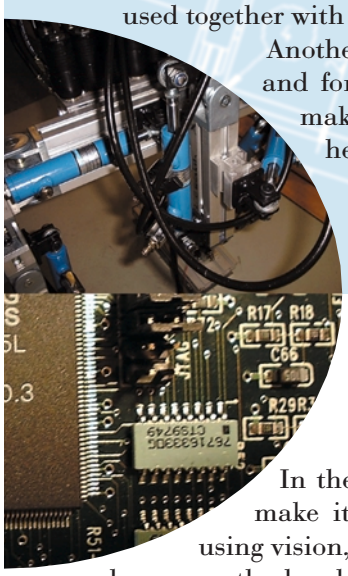
MURPHY

Murphy is a robot resembling humans — an humanoid — located in our basement room at Polacksbacken in Uppsala. He is almost two metres tall, weighs 130 kg, has an incredible shoulder width and is equipped with around 600 kilos of pulling force in his "muscles". He is named after the robot police Murphy in the Robocop movies, and they are about the same size. The difference is that the original Murphy was a robocop and our Murphy aims to participate in the Robocup (robot football world championship).

In the world cup, he will compete with Priscilla from Gothenburg, another humanoid robot. Together, they form a team in the humanoid robot class. One of them will be a goal keeper and the other will shoot penalty kicks, but positions have not been decided yet. Murphy is the bigger one and is perhaps more suitable as a keeper. On the other hand, he has got a lot of power to put behind his penalty kicks.

At the moment, we concentrate on deciding how Murphy is supposed to gain knowledge of how to walk on his own. One way is to build a nerve system, letting Murphy learn from his mistakes, and thereby learning to walk better. It is an approach as good as any, but there is a lot of work before we are there. The solution will probably be this approach used together with a neural network.

Another alternative is to simply say to him "this is how you walk", and force a movement pattern that works. This is a quick way to make him walk, but he will not be able to improve over time, like he would using the first approach.



MURRAY

In the Murray project we are creating Murphy's head, which will make it possible for him to communicate with his surroundings, using vision, hearing and speech. To make it possible for him to use body language, the head has three degrees — three joints, that is — of freedom. This enables him to nod and shake his head.

VISION

The primary task for the head is to identify objects, such as a football, a goal frame, or an opponent. An active research topic is making Murray able to measure the distance to objects using stereo vision, i.e. two cameras as eyes. Another challenge is making him do this while he is moving.

HEARING

To be able to talk to Murphy, two microphones monitor the surroundings in order to record words and sentences, which are transformed into text strings. These strings are then sent to the intelligence module for analyse. The aim is also, by using the difference in time, frequency, and intensity between the two ears (microphones), to localize the source of the sound, allowing the head to turn towards the origin of the sound.

SPEECH

The speech synthesis module will, using a speaker, be able to say the sentence decided by the intelligence module. The speech should sound as realistic as possible and will be the most important way of communication for Murray.

INTELLIGENCE

A central unit processes the sound and video information from the different senses and perform actions based on that information, resulting in speech, head movements, and, in time, that Murphy can play football all by himself. It is the connection point of all the different senses, and later on it will control the movements of Murphy.

Pattern recognition

The most sofisticated feature of our brains', and the hardest to make computers mimic. It is about making computers recognize faces, understand human speech, not to walk against red light, refilling the glass when it is empty, and doing a multitude of tasks we humans perform without thinking.

Reinforcement learning

When we were a few years old, we learnt that oven lids can be very hot. One touch was all we needed to stay well away from the oven. This is a crude example of reinforcement learning; vi perform an action and then learn from it.

Computers and robots do of course not have any means of collecting experience. Here, a *reward system* is used. It tells the computer/robot when it did something good or bad.

Hydraulics

Most machines at a construction site use hydraulics. It is the principle behind the long cylinders that control excavators and lift truck beds. By pumping oil into either of the ends of the cylinders, they expand or contract. The oil can be pumped with much force, and there are few mechanical parts that can break, making the system effective.

Murphy's joints are controlled by hydraulics. The paddles mounted on the front of the football robots move by similar means, using a canister filled with pressurized air as force. Then it is called *pneumatics*.

Speech synthesis

One way to make computers talk is recording words or syllables, and letting a copmuter program merge them into comprehensible phrases. Only it is very exhaustive to listen to. We humans also colour our language according to mood and situation, which is completely lost with today's programs. This is what speech synthesis is about, creating methods that make computers sound just as we do.

Footnote: Murray got his name from the talking, megalomaniac skull Murray in the computer game *Curse of Monkey Island*.

Team building/Kickoff

Launch your project or team building in an inspiring environment connected to the university, and challenge the football playing robots in a game where you control the opponents by remote, or compete in a penalty kick contest. Try manouvering the humanoid robot Murphy. We arrange activities, conferences, and food.

Company lunch or dinner

Introduce your company to Master of Science students with a major in IT, who are to enter life as professionals this autumn. We guarantee you will meet at least 40 sharp minds. A booklet containing the CV:s of all students and their contact information will be handed to you after the presentation, followed by buffé or lunch.

Home page analysis

How does your company's home page work from a user point of view? We perform a thorough analysis using methods developed by scientists at the Institution for Human-Computer Interaction. We can also evaluate its reliability and security devices.

Training

We can offer courses for all versions of MsWindows and MsOffice, web design, Visual Basic, HTML, Java programming, computer security, and more.

Adopt a robot

Your company may become a parent to one or several football playing Argus robots, or why not Murphy himself. Grasp the opportunity to be seen in broadcasting media, in relation to the robot world cup in Japan, or perhaps some youth program on TV.

Help Murphy walk, talk, and see

The humanoid robot Murphy will this year get a head, but we need parts for it. Resources are also needed to make him stand up and walk.

Sponsor the world cup players

We are allowed some advertisement on both Murphy and Argus during the world cup games in Japan's largest indoor arena Fukuoka Dome Stadium, housing 52 000 sitting spectators. You can count on media being there...

All sponsors are of course presented on our well visited home page *www.rfcuppsala.org*. We will arrange a sponsor meeting towards the end of the project period, where we will show our accomplishments and present a final report.

PARTNERS

FlexLink®

MOOG

Nanopuls AB

projektplatsen.se

Rexroth
Bosch Group

STRATO
Information AB

SWEDE COM

Yobotics!

CONTACT US!

Address

RFCU, floor 4 box 7
Box 325
Polacksbacken 1st house
751 05 Uppsala, Sweden
Internet: www.rfcuppsala.org
Fax: +46 (0)18 55 02 25

PR

Patrik Lakhsasi
E-mail: info@rfcuppsala.org
Telephone: +46 (0)18 471 72 16
+46 (0)708 14 24 21

Argus

Anders Taflin
E-mail: anta6963@student.uu.se
Telephone: +46 (0)18 710 30 76
+46 (0)70 441 64 80

Murphy

Mattis Fjällström
E-mail: mafj1085@student.uu.se
Telephone: +46 (0)18 471 72 16
+46 (0)70 768 71 70

Murray

Magnus Andersson
E-mail: maan1947@student.uu.se
Telephone: +46 (0)70 753 41 72

Printing: Eklundshofs Grafiska, Uppsala

Flexibility Driven Scheduling and Mapping for Distributed Real-Time Systems

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

Abstract

In this paper we present an approach to mapping and scheduling of distributed hard real-time systems, aiming at improving the flexibility of the design process. We consider an incremental design process that starts from an already existing system running a set of applications, with preemptive priority based scheduling at the process level, and time triggered static scheduling at the communication level. 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 problems are considered in the context of a realistic communication model based on a TDMA protocol. Extensive experiments as well as a real life example demonstrate the relevance of this problem and the efficiency of our solutions.

1. Introduction

In this paper we concentrate on scheduling and mapping of hard real-time systems which are implemented on distributed architectures. Process scheduling is based on a static priority preemptive approach while the bus communication is statically scheduled.

Preemptive scheduling of independent processes with static priorities running on single processor architectures has its roots in [7]. The approach has later been extended to accommodate more general computational models and has also been applied to distributed systems [17]. The reader is referred to [1, 2, 13] for surveys on this topic. In many of the previous scheduling approaches researchers have assumed that processes are scheduled independently. However, this is not the case in reality, where process sets can exhibit both data and control dependencies. One way of dealing with data dependencies between processes with static priority based scheduling has been indirectly addressed by the extensions proposed for the schedulability analysis of distributed systems through the use of *release jitter* [17].

Currently, more and more real-time systems are used in physically distributed environments and have to be implemented on distributed architectures in order to meet reliability, functional, and performance constraints. We have to mention here some results obtained in extending real-time schedulability analysis so that network communication aspects can be handled. In [16], for example, the CAN protocol is investigated while the work reported in [4] considers systems based on the ATM protocol. Analysis for a simple TDMA protocol is provided in [17], which integrates processor and communication schedulability and provides a "holistic" schedulability analysis in the context of distributed real-time systems. The problem of how to allocate priorities to a set of distributed tasks is discussed in [5], based on the schedulability analysis from [17].

Another characteristic of research efforts concerning the codesign of real-time systems is that researchers 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 [3]. 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 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, the time-triggered protocol (TTP), as recommended for applications in areas like, for example, automotive electronics [6].

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) the existing applications are disturbed as little as possible;

b) there is a good chance that new functionality can, later, be easily mapped on the resulted system.

In [11] we have discussed an incremental design strategy addressed to systems where *both* processes and messages are statically scheduled. However, considering preemptive priority based scheduling at the process level, with time triggered static scheduling at the communication level, can be the right solution under certain circumstances [8]. A communication protocol like TTP provides a global time base, improves fault-tolerance and predictability. At the same time, certain particularities of the application or of the underlying real-time operating system can impose a priority based scheduling policy at the process level.

In this paper we extend our approach to hard real-time systems where process scheduling is based on a static priority preemptive approach while the bus communication is statically scheduled. We accurately take into consideration overheads due to communication and consider, during the mapping and scheduling process, the particular requirements of the communication protocol. We propose a new heuristic, together with the corresponding design criteria, which finds the set of old applications which have to be remapped at the same time 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 a) and b) 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 the architectures considered for system implementation, the computational model assumed together with the schedulability analysis employed, and the process allocation problem. Section 3 introduces the detailed problem formulation and the quality metrics we have defined. The mapping and scheduling strategy is presented in section 4, and the approaches are evaluated in section 5. The last section presents our conclusions.

2. Preliminaries

2.1 System Architecture

We consider architectures consisting of nodes connected by a broadcast communication channel. Every node consists of a TTP controller, a CPU, a RAM, a ROM and an I/O interface to sensors and actuators.

Communication between nodes is based on the TTP [6]. TTP was designed for distributed real-time applications that require predictability and reliability (e.g., drive-by-wire). The communication channel is a broadcast channel, so a message sent by a node is received by all the other nodes. The bus access scheme is time-division multiple-access (TDMA) (Figure 1). Each node N_i can transmit only during a predetermined time interval, the so called TDMA slot S_i. 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. The sequence and length of the slots are the same for all TDMA rounds. However, the length and contents of the frames may differ.

Every node has a TTP controller that implements the protocol services and runs independently of the node's CPU. The TDMA access scheme is imposed by a so called message descriptor list (MEDL) that is located in every TTP controller. MEDL serves as a schedule table for the TTP controller which has to know when to send or receive a frame to or from the communication channel. The TTP controller provides each CPU with a timer interrupt based on a local clock synchronized with the local clocks of the other nodes.

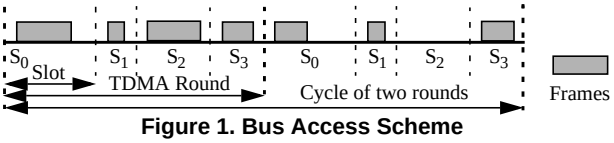


Figure 1. Bus Access Scheme

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. For the exact details of the software architecture and how it is taken into account by the schedulability analysis the reader is directed to [10].

2.2 Computational Model and Schedulability Analysis

We model an application as a set of processes. A process P_i has a period T_i, a deadline D_i, and a uniquely assigned priority. Each process P_i can potentially be mapped on several nodes. Let N_{P_i} be the set of nodes to which P_i can potentially be mapped. For each $N_i \in N_{P_i}$, we know the worst case execution time $C_{P_i}^{N_i}$ of process P_i, when executed on N_i. We consider a preemptive execution environment, which means that higher priority processes can interrupt the execution of lower priority processes. In this paper we use a deadline monotonic priority assignment scheme, but any other more advanced priority schemes like the ones in [5, 14] could as well be used. A lower priority process can block a higher priority process (e.g., it is in its critical section), and the blocking time is computed according to the priority ceiling protocol [13]. Processes exchange messages, and for each message m_i we know its size S_{m_i}. A message is sent once in every n_m invocations of the sending process, with a period $T_m = n_m T_i$ inherited from the sender process P_i, and has a unique destination process.

For a given mapping of processes to processors, Tindell et al. integrate in [17] processor and communication scheduling and provide a "holistic" schedulability analysis in the context of distributed real-time systems with communication based on a simple TDMA protocol. The basic idea in [17] is that the release jitter of a destination process depends on the communication delay between sending and receiving a message. The release jitter of a process is the worst case delay between the arrival of the process and its release (when it is placed in the run-queue for the processor). The communication delay is the worst case time spent between sending a message and the message arriving at the destination process.

Although there are many similarities with the general TDMA protocol, the analysis in the case of TTP is different in several aspects and also differs to a large degree depending on the policy chosen for message scheduling. In [10] we have proposed four approaches for scheduling of messages using TTP that differ in the way the messages are allocated to the communication channel (either statically or dynamically) and whether they are split or not into packets for transmission. For each of these approaches, we have also developed the corresponding schedulability analysis [10].

The first approach, called Static Single Message Allocation (SM), is to statically (off-line) schedule each of the messages into a slot of the TDMA cycle, corresponding to the node sending the message. We also consider that the slots can hold each at maximum one single message. The second approach, called Static Multiple Message Allocation (MM), is an extension of the first one. In this approach we allow more than one message to be statically assigned to a slot and all the

messages transmitted in the same slot are packaged together in a frame. While the previous two approaches have statically allocated one or more messages to their corresponding slots, the third approach, called Dynamic Message Allocation (DM), considers that the messages are dynamically allocated to frames, as they are produced. Finally, the last approach, called Dynamic Packets Allocation (DP), is an extension of the previous one, and allows the messages to be split into packets before they are transmitted on the communication channel. By splitting messages into packets we can obtain a higher utilization of the bus and reduce the release jitter.

Comparing these four approaches, in [10] we conclude that while the DP approach is generally the most performant since the dynamic scheduling of messages is able to reduce release jitter because no space is waisted in the slots if the packet size is properly selected, by using the MM approach we can obtain almost the same result if the messages are carefully allocated to slots. Moreover, in the case of larger process sets MM outperforms DP, as DP suffers from large overhead due to its dynamic nature. Also, DM performs worse than DP and MM because it does not split the messages into packets, and this results in a mismatch between the size of the messages dynamically queued and the slot size, leading to unused slot space that increases the jitter. SM performs the worst as it does not permit much room for improvement, leading to large amounts of unused slot space.

Therefore, for the purpose of this paper, we consider that the messages are scheduled using the MM approach, and for the details of the corresponding schedulability analysis the reader is referred to [10]. The discussion can easily be extended to any of the other three message passing approaches presented before.

2.3 Application Mapping and Scheduling

In order to implement an application, represented as a set of processes, the designer has to map the processes to the system nodes and generate the schedule table for the messages (MEDL) such that all deadlines are satisfied. But producing a mapping and scheduling so that the system is schedulable is not enough if we are to support an incremental design process as discussed in the introduction. In this case, starting from a schedulable system, we have to improve the mapping of processes and scheduling of messages so that not only the design constraints are satisfied, but there is also a good chance that, later, new functionality can easily be mapped on the resulted system.

To illustrate the role of mapping and scheduling in the context of an incremental design process, let us consider the example in Figure 2, where we have two processors with the same speed connected by a TTP bus. With black we represent the set of already running applications ψ while the current application $\Gamma_{current}$ to be mapped and scheduled is represented in grey and consists of two processes and three messages. To simplify the discussion, for this particular example we consider that the system is not schedulable if the *utilization factor* of any node is greater than one.We say that the processor can be "filled up" with processes until it reaches an utilization factor of one (the square depicting the processor is full). The utilization factor U_i of a process P_i is the ratio between the worst case execution time C_{P_i} of that process and its period T_i: $U_i = C_{P_i} / T_i$. The utilization factor of a node is the sum of the utilization factors of all processes mapped on that node. The processes and messages that are to be mapped on the processors are depicted as blocks. The height of a process block is equal with its utilization factor, while the length of a message block gives the size of the message. White space on a processor represents available utilization, while white space on the bus represents available slack in the schedule table.

Now, let us suppose that in the future another application, Γ_{future}, has to be mapped on the system. Γ_{future} consists of two processes and two messages represented as hashed blocks.

We can observe that the new application can be scheduled only in the third case, presented in Figure 2c. If $\Gamma_{current}$ has been implemented as in Figure 2b, we are not able to schedule process P_2 and message m_2 of Γ_{future}. The way our current application is mapped and scheduled will influence the likelihood of successfully mapping additional functionality on the system without being forced to modify the implementation of already running applications.

3. Problem Formulation

We model an application $\Gamma_{current}$ as a set of processes as outlined in section 2.2. Thus, for each process P_i we know the set N_{P_i} 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 fixed priority preemptive scheduling for processes and a time-triggered message passing policy, as imposed by the TTP protocol (section 2.1).

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 mapping and scheduling 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 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.

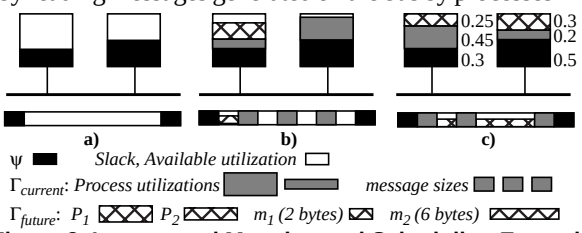


Figure 2. Incremental Mapping and Scheduling Example

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 worst case execution time, period and deadline for each process on its node. As for messages, their length as well as their place in the particular TDMA frame have to be known.

If the initial attempt to schedule and map $\Gamma_{current}$ does not succeed, we have to modify the mapping of processes and schedule of messages belonging to ψ, in the hope to find a valid solution for $\Gamma_{current}$. One of the goals in this paper is to find that minimal modification to the existing system which 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. In Figure 3 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 allowed 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 of the application (moving processes between nodes).

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 incomplete designs for future versions of the product, etc., it is possible to characterize the family of applications which would 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 $S_U=\{U_{min},...U_i,...U_{max}\}$ of possible *processor utilization factors* for processes, and the set $S_b=\{b_{min},...b_i,...b_{max}\}$ of possible message sizes. The processor utilization factor U_i provides a measure of the computational load due to the a process P_i, and is expressed as $U_i=C_{P_i}/T_i$. The utilization factors in S_U are considered relative to the slowest node in the system. All the other nodes are characterized by a speed-up factor relative to this slowest node, which is used to calculate the actual utilization factor due to a process P_i if mapped on a node N_j. The utilization factor for an entire process set is given by $U=\sum_{i=1}^{n} U_i$.

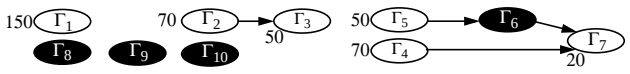


Figure 3. Characterizing the Set of Existing Applications

We also assume that over these sets we know the distributions of probability $f_{S_U}(U)$ for $U \in S_U$ and $f_{S_b}(b)$ for $b \in S_b$. For example, we might have possible utilization factors $S_U=\{0.02, 0.05, 0.1, 0.2\}$ for the future application. If almost half of the processes are assumed to have an utilization factor of 0.1, and there is a lower probability of having processes with utilization factors of 0.2 and 0.02, then our distribution function $f_{S_U}(U)$ could look like this: $f_{S_U}(0.02)=0.15$, $f_{S_U}(0.05)=0.25$, $f_{S_U}(0.1)=0.45$, $f_{S_U}(0.2)=0.15$.

Another information is related to the period of future applications. In particular, the smallest expected period T_{min} is assumed to be given, together with the expected necessary 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 on the bus.

For the sake of simplifying the discussion, we will not address here the memory constraints during process mapping and the implications of memory space in the incremental design process.

3.3 Quality Metrics

A designer will be able to map and schedule a Γ_{future} application on top of a system implementing ψ and $\Gamma_{current}$ only if there are sufficient resources available. In our case, the resources are the processor time and the bandwidth on the bus. In the context where processes are scheduled according to a fixed priority preemptive policy and messages are scheduled statically, having free resources translates into having enough processor capacity, and having space left for messages in the bus slots. We measure the processor capacity using the *available utilization*, while the available resources on the bus are called *slack*.

It is to be noted that the total quantity of computation and communication power available on our system after we have mapped and scheduled $\Gamma_{current}$ on top of ψ is the same regardless of the mapping and scheduling policies used. What depends on the mapping and scheduling strategy is the distribution of the available utilization on each processor, the size of the individual slacks on the bus, and the distribution of slacks along the time line. It is the distribution of available utilization and the size and distribution of the slacks that characterizes the quality of a certain design alternative. In this section we introduce the design criteria which reflect the degree to which one design alternative meets the requirement b) presented in section 3. For each criterion we provide metrics which quantify the degree to which the criterion is met. Relative to processes we have introduced one criterion which reflects how well the resulted available utilization on the nodes fits the requirements of a future application. For messages, there are two criteria. The first one reflects how well the resulted slack sizes fit a future application, and the second criterion expresses how well the slack is distributed over time.

3.3.1 Processes Related Criterion

The distribution of available utilization on the nodes, resulted after implementation of $\Gamma_{current}$ on top of ψ, should be

such that it best accommodates a given family of applications Γ_{future}, characterized by the set S_U and the probability distribution f_{S_U} as outlined before.

Let us consider the example in Figure 2, where we have two processors and the applications ψ and $\Gamma_{current}$ are already mapped. Suppose that application Γ_{future} consists of the two processes P_1 and P_2. If we schedule $\Gamma_{current}$ like in Figure 2b it is impossible to fit Γ_{future} because there is not enough available utilization on any of the processors that can accommodate process P_2. A situation as the one depicted in Figure 2c is desirable, where the resulted available utilization is such that the future application can be accommodated.

In order to measure the degree to which the available utilization in a given design alternative fits the future applications, we provide a metric, C_1^P, which captures to what extent the largest future application (considering the sum of available process utilization) could be mapped on top of the current design. This potentially largest application is determined knowing the total size of the available utilization, and the characteristics of the application: S_U and f_{S_U}. For example, if our *total* available utilization on *all* the processors is of 1.81 then we have to distribute this utilization according to the probabilities in f_{S_U}. Considering the numerical example for processes given in section 3.2, the largest application will result as having a total of 20 processes: 3 processes of utilization 0.02, 5 of 0.05, 9 processes (almost half, $f_{S_U}(0.1)$=0.45) of utilization 0.1, and 3 of 0.2. If the number of processes for a particular dimension is not an integer, then we use the ceiling. After we have determined the largest Γ_{future} we apply a *bin-packing* algorithm [9] using the *best-fit* policy in which we consider processes as the objects to be packed, and the available utilization as containers. The total utilization of unpacked processes relative to the total utilization of the process set gives the C_1^P metric. In the case presented in Figure 2b U_1=0.3 and U_2=0.25, and P_2 represents 45% of the largest possible future application. In this case C_1^P=45. However, in Figure 2c were we were able to completely map the future application C_1^P=0.

3.3.2 Criteria Related to Messages

The first criterion for messages is similar to the one defined for processes. Thus, the slack sizes in the message schedule table MEDL (see section 2.1) 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 set S_b and the probability distribution f_{S_b} for messages.

Let us consider the example in Figure 2, where we have two processors and the applications ψ and $\Gamma_{current}$ are already mapped. Application Γ_{future} has two messages m_1 and m_2. It can be observed that the best configuration, taking in consideration only slack sizes, is to have a contiguous slack. 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. On the other side, as we can see from Figure 2b, if we schedule $\Gamma_{current}$ so that it fragments too much the slack, it is impossible to fit Γ_{future} because there is no slack that can accommodate message m_2. A situation as the one depicted in Figure 2c is desirable, where the resulted slack sizes can accommodate the characteristics of the Γ_{future} application.

In order to measure the degree to which the slack sizes in a given design alternative fit the future applications, we provide the metric C_1^m. C_1^m captures how much of the communications of the largest future application which theoretically could be mapped on the system if the slacks on the bus would be summed, can be mapped on the current design alternative. The messages accounting for the largest amount of communication are determined, as shown above for processes, knowing the total size of the available slack, and the characteristics of the application: S_b and f_b.

C_1^m is calculated similarly to the metric C_1^P but, instead of packing the processes as objects, we try to pack the messages into the available slack on the bus. C_1^m is then the total size of unpacked messages, relative to the total size of messages in the largest future application. For Figure 2b, where m_2 could not be scheduled, C_1^m is 75% because m_2 of 6 bytes represents 75% of the total message sizes of 8 bytes. For the design alternative in Figure 2c C_1^m is 0% because all the messages have been scheduled.

We have just discussed a metric for how well the sizes of the slacks fit a possible future application. A similar metric is needed to characterize the distribution of slacks over time.

During implementation of $\Gamma_{current}$ we aim for a slack distribution such that the future application with the smallest expected period T_{min} and with the expected necessary bandwidth b_{need} inside the period T_{min}, can be accommodated. The minimum over the slacks inside each T_{min} period, which is available *periodically* to the messages of Γ_{future}, is the C_2^m metric.

In Figure 4 we present a message schedule table. We consider a situation with T_{min}=120 ms and b_{need}=40 ms. The length of the schedule table is 360 ms, and the already scheduled messages of ψ and $\Gamma_{current}$ are depicted in black. Let us consider the situation in Figure 4a. In the first period T_{min}, *Period 0*, there are 40 ms of slack available on the bus, in the second period 80 ms, and in the third period no slack is available. Thus, the total slack a future application with a period T_{min} can use on the bus in each period is C_2^m = min(40, 80, 0)=0 ms. In this case, the messages cannot be scheduled. However, if we move m_1 to the left in the schedule table, we are able to create, in Figure 4b, 40 ms of slack in each period, resulting a C_2^m = 40 ms = b_{need}.

3.4 Cost Function and Exact Problem Formulation

In order to capture how well a certain design alternative meets the requirement b) stated at the beginning section 3, the metrics discussed before are combined in a cost function, as follows: $C = w_1^P (C_1^P)^2 + w_1^m (C_1^m)^2 + w_2^m max(0, b_{need} - C_2^m)$, where the metric values are weighted by the constants w_i. Our mapping and scheduling strategy will try to minimize this function.

The first two terms measure how well a future application fits to the available utilization on the processors and slack sizes on the bus, respectively. In order to obtain a balanced solution, that favours a good fitting both on the processors

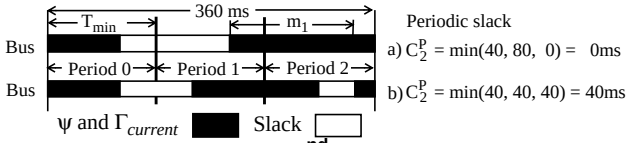


Figure 4. Example for the 2nd Message Design Criterion

and on the bus, we have used the squares of the metrics.

A design alternative that does not meet the second design criterion for messages is not considered a valid solution. Thus, using the last term, we strongly penalize the cost function if b_{need} is not satisfied, by using high values for the w_2 weight.

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 mapped on top of ψ, we are interested to find a mapping and scheduling of $\Gamma_{current}$ which satisfies all deadlines such that the existing applications are disturbed as little as possible. At the same time, the solution should minimize the cost function C, considering a family of future applications characterized by the sets S_U and S_b, the functions f_{SU} and f_{Sb} as well as the parameters T_{min} and b_{need}.

4. Mapping and Scheduling Strategy

As shown in Figure 5, our mapping and scheduling strategy (MH) has two steps. In the first step we try to obtain a valid solution for $\Gamma_{current} \cup \Omega$ so that the total modification cost $R(\Omega)$ is minimized ($\Omega \subseteq \psi$ is the subset of existing applications that have to be modified to accommodate $\Gamma_{current}$). Starting from such a solution, a second step iteratively improves on the design in order to minimize the cost function C. We iteratively improve the design using a transformational approach. A new design is obtained from the current one by performing a transformation called *move*. We consider the following moves: moving a process to a different node, and moving a message to a different slack on the bus. Only those moves are valid moves that result in a schedulable system. The intelligence of the Mapping Heuristic lies in how the potential moves are selected. For each iteration a set of potential moves is generated by the PotentialMove functions. SelectMove functions then evaluate these moves with regard to the respective metrics and selects the best one to be performed.

4.1 The Initial Mapping and Scheduling

The first step of MH 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 (both for processes in $\Gamma_{current}$ and those in Ω), without worrying about the design criteria in section 3.3. For IMS we used as a starting point the mapping algorithm introduced in [15], based on a simulated annealing strategy. We have modified the mapping algorithm in [15] to consider during mapping a set of previous applications that have already been mapped, and to schedule the messages according to the TDMA protocol, using the MM approach [12]. The schedulability test that checks a particular mapping alternative is performed according to our schedulability analysis presented in [10].

If IMS succeeds in finding a mapping and a 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 for messages. In terms of our metrics, that means a mapping and scheduling such that $C_2^m \geq b_{need}$. Potential moves can be the shifting of messages inside their worst case (largest) [ASAP, ALAP] interval in order to improve the periodic slack. In PotentialMoveC_2^m , we also consider movement of processes, trying to place the sender and receiver of a message on the same processor and, thus, reducing the bus load. SelectMoveC_2^m evaluates these moves with regard to the second design criterion and selects the best one to be performed. Consider Figure 4a. In *Period 2* on node N_1 there is no available slack. However, if we move message m_1 with 40 ms to the left into *Period 1*, as depicted in Figure 4b, we create a slack in *Period 2*, thus the periodic slack on the bus will be min(40, 40, 40)=40, instead of 0.

4.2 Incremental Mapping and Scheduling Strategy

If Step 1 of the MH algorithm (Figure 5) 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 (minimization of the modification cost is detailed in section 4.3). Starting from this valid solution, the second step of the MH strategy tries to improve on the design in order to minimize the cost function C. In a similar way as during Step 1, we iteratively improve the design by successive moves, without invalidating the second criterion achieved in the first loop.

The loop ends when there is no improvement achieved on the first two terms of the cost function, or a limit imposed on the number of iterations has been reached. For each iteration, those moves are performed which have the highest chance to improve the cost function. The moves are generated in the PotentialMove functions, and are evaluated and selected based on the respective metrics in the SelectMove functions. We now briefly discuss the PotentialMoveC_1^P and PotentialMoveC_1^m functions (PotentialMoveC_2^m has been discussed in the previous section).

PotentialMoveC_1^P Let U_f be the total utilization factor of the largest future application Γ_{fmax}, and U_0 the utilization of that part which cannot be mapped in the current design alternative. This function is responsible for selecting moves

```
MappingSchedulingStrategy (MH)
Ω=∅ -- Step 1: try to find a valid schedule for Γcurrent that minimizes R(Ω)
repeat
   succeeded=IMS(ψ \ Ω, Γcurrent∪Ω) -- initial mapping and scheduling
   ASAP(Gcurrent∪Ω); ALAP(Gcurrent∪Ω)
   -- compute worst case ASAP-ALAP intervals for messages
   if succeeded then
      repeat -- try to satisfy the second message related design criterion
      -- find moves with highest potential to maximize C2m
      move_set=PotentialMoveC2m (Gcurrent∪Ω)
      -- select and perform move which improves most C2m
      move = SelectMoveC2m (move_set); Perform(move)
      succeeded = C2m ≥ bneed
      until succeeded or limit reached
   end if
   if succeeded and R(Ω) smallest so far then
      Ωvalid=Ω; solutionvalid=solutioncurrent
   end if
   Ω=NextSubset(Ω) -- try another subset
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=PotentialMoveC1P (Gcurrent∪Ωmin)
      ∪ PotentialMoveC1m (Gcurrent∪Ωmin)
   -- select move which improves C
   -- and does not invalidate the second message related design criterion
   move = SelectMoveC1(move_set); Perform(move)
until C1 has not changed or limit reached
end MappingSchedulingStrategy
```

Figure 5. Mapping and Scheduling Strategy (MH)

of processes from one node to another so that $c_1^P = \frac{U_0}{U_f}100$ is reduced. Moving a process P_i with the utilization factor U_i from a node N_j where it is currently mapped to a node N_k will increase the available utilization on node N_j to $U_{Nj}+U_i$ and decrease the available utilization on N_k to $U_{Nk}-U_i$. To find out U_0 in this new case would mean executing the bin-packing with the processes of the future application as objects and the new available utilization configuration as containers. This can take significant execution time, however, since it has to be done for each potential move.

In section 3.3 we have explained how we can determine the processes that make up the largest future application, Γ_{fmax}, based on the total available utilization and the characterization of future applications. Let us assume that Γ_{fmax} consists of the set $P_{fmax}=\{P_{f1}, P_{f2},...P_{fn}\}$ of processes, and that $P_0=\{P_{fi}, P_{fi+1},...,P_{fm}\}$ are the ones that cannot be mapped in the current design alternative. The total utilization requested by the unmapped processes is $U_0=U_{fi} + U_{fi+1} + ... + U_{fm}$. For the potential move of P_i from N_j to N_k we have to recalculate C_1^P which means determining U_0.

In order to reduce the execution time needed by the bin-packing algorithm, we do not consider all the processes of Γ_{fmax} as objects to be packed. We consider for repacking only those processes belonging to Γ_{fmax} that had to be removed from N_k to make room for P_i, together with those that were already left outside. Our heuristic considers that to make room for P_i on node N_k we remove those processes $P_i^{Nk} \subset \Gamma_{fmax}$ mapped on N_k which have the smallest utilization factor, since they are the ones that should be easiest to fit on other nodes. The metric used by SelectMove to rank this move is the sum of the utilization factors of processes which are left out after trying to repack the $P_0 \cup P_i^{Nk}$ set.

Out of the best moves according the previous metric, we encourage those that have the smallest impact on the schedulability analysis, since we would like to keep the system schedulable. This means moving processes that have low priority (do not have a large impact on other processes) and have a response time that is considerably smaller than their deadline (D_i - R_i is large).

PotentialMoveC_1^m In order to avoid excessive fragmentation of the slack on the bus we will consider moving a message to a position that "snaps" to another existing message. A message is selected for potential move if it has the smallest "snapping distance", i.e. in order to attach it to other message it has to travel the smallest distance inside the schedule table. We also consider moves that try to increase the individual slacks sizes. Therefore, we first eliminate slack that is unusable: it is too small to hold the smallest message of the future application. Then, the slacks are sorted in ascending order and the smallest one is considered for improvement. Such improvement of a slack is performed through moving a nearby message, but avoiding to create as a result an even smaller individual slack.

4.3 Minimizing the Modification Cost

In the first step of our mapping strategy, described in Figure 5, we iterate 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 ($\Omega=\varnothing$). 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 $\varnothing$. 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 2, the call to NextSubset($\varnothing$) will generate $\{\Gamma_7\}$ which has the smallest non-zero 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 Subset Selection (AS)

If the number of applications is large, a possible ad-hoc solution could be based on a greedy strategy which, starting from $\Omega=\varnothing$, 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 2 the call to NextSubset($\varnothing$) 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 heuristic should be able to identify the reasons due to which a valid solution has not been found and use this information when selecting applications to be included in Ω. There can be two possible causes for not finding a valid solution: 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 schedulable, this means that all its processes meet their deadlines. If IMS determines that the application is not schedulable this means that at least one of the processes P_i missed its deadline: $R_i > D_i$. Besides the intrinsic properties of the application that can lead to this situation, process P_i can miss its deadline also because of the interference of higher priority processes that are mapped on the same node with P_i, processes that can also belong to other applications. 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 (D_i - R_i is large).

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 $\Delta(\Omega)$ 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.

Basically, Δ_i is the total amount of *interference* caused by higher priority processes of Γ_i to processes in $\Gamma_{current}$. For a process P_i, the interference I_{ji} from a higher priority process P_j mapped on the same node, is the time that P_j delays the execution of P_i, and is given by $I_{ji}=\left\lceil \frac{J_j + R_i}{T_j} \right\rceil C_j$ where J_j is the release jitter of process P_j and a detailed description of how it is calculated in the context of the MM approach for message scheduling over TTP is given in [10]. Figure 6 presents in more detail how Δ_i is calculated.

If the initial mapping was successful, the first step of MH could fail during the attempt to satisfy the second design criterion for messages. 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^m if remapped together with $\Gamma_{current}$. Thus, for the improvement of C_2^m we consider a total number of moves from all the non-frozen applications (determined using PotentialMoveC_2^m (y), see section 4.1). For each move that has as subject $m_j \in \Gamma_i$, we increment the metric Δ_i with the predicted improvement on C_2^m .

MH starts by trying an implementation of $\Gamma_{current}$ with $\Omega=\varnothing$. 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 AS 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 favourable 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 than R_{max} and a larger Δ than Δ_{min} (a small Δ means a reduced potential to eliminate the cause of the 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).

```
DeltaMetrics(Γcurrent, Ω)
for each non frozen Γi ∈ Ω Δi = 0 end for
for each Pi ∈ Γcurrent
    if Ri > Di
        for each non frozen Γk ∈ Ω
            -- hp(Pi) is the set of processes with higher priority than Pi
            for each Pj ∈ Γk ∩ hp(Pi): Δk = Δk + Cj⌈(Jj+Ri)/Tj⌉ end for
        end for
    end if
end for
return Δ
end DeltaMetrics
```

Figure 6. Determinig the Δ metrics

5. Experimental Results

For the evaluation of our mapping strategies we first used process sets of 40, 80, 160, 240 and 320 processes representing the $\Gamma_{current}$ application generated for experimental purpose. 30 applications were generated for each set dimension, thus a total of 150 applications were used for experimental evaluation. We considered an architecture consisting of 10 nodes of different speeds. 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. All experiments were run on a SUN Ultra 10.

The first result concerns the quality of the designs obtained with our mapping strategy MH using the search heuristic SH compared to the case when the ad-hoc approach AS and the exhaustive search ES are used for subset selection. For each of the five application dimensions generated we have considered a set of existing applications ψ consisting of 160, 240, 320, 400 and 480 processes, respectively. The sets contained 4, 6, 8, 10 and 12 applications, each application with an associated modification cost assigned manually in the range 10 to 100. The dependencies between applications were such that the total number of subsets resulted for each set ψ were 8, 32, 128, 256, and 1024. We have considered that the future applications Γ_{future} consist of a process set of 80 processes, randomly generated according to the following specifications: S_U={0.02, 0.05, 0.1, 0.15, 0.2}, $f_{S_U}(S_U)$={10, 25, 45, 15, 5%}, S_b={2, 4, 6, 8 bytes}, $f_{S_b}(S_b)$={20, 50, 20, 10%}, T_{min}=250 ms and b_{need}=20 ms.

MH has been used to produce a valid solution for each of the 150 process sets representing $\Gamma_{current}$ on top of the existing applications ψ using the ES, AS and SH approaches to subset selection. For each of the resulted valid solutions, there corresponds a minimum modification cost $R(\Omega_{min})$. Figure 7a compares the three approaches to subset selection based on the modification cost needed in order to obtain a valid solution. The exhaustive approach ES is able to obtain valid solutions at the optimum (smallest) modification cost, (e.g. less than 400, in average, for systems with 12 applications consisting of a total of 480 processes), while the ad-hoc approach AS needs in average 3.11 times more costly modifications in order to obtain valid solutions (e.g. more than 1100 for 480 processes in Figure 7a). However, in order to find the optimal remapping the ES approach needs large computation times. For example, it can take more than 35 minutes, in average, in order to find the smallest cost subset to be remapped that

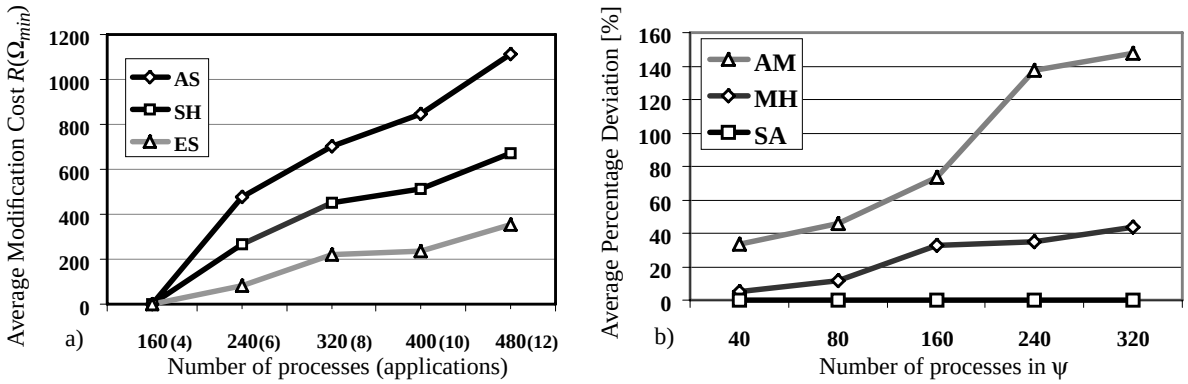


Figure 7. a) Average Modification Costs for AS, SH, ES and b) Percentage Deviations for AH, MH, SA

leads to a valid solution in the case we have 12 applications. From Figure 7a we can see that the proposed heuristic SH performs quite well, needing only 1.84 times larger costs, in average, in order to obtain a valid schedule, and this is achieved at a computation cost comparable with the fast ad-hoc approach AS. For the results in Figure 7a we have eliminated those situations in which a valid solution has not been produced by MH (which means that there is no solution regardless of the modification cost).

Next, we were interested to investigate the quality of the mapping heuristic MH compared to a so called *ad-hoc mapping approach* (AM). To concentrate on this, we have considered that *no modifications* are allowed to the applications in ψ. The AM approach is a simple, straight-forward solution to produce designs which, to a certain degree, support an incremental process. AM tries to evenly balance the available utilization remaining after mapping the current application. The quality of the designs obtained with MH and AM were compared with a near-optimal mapping and schedule obtained with a Simulated Annealing strategy (SA) strategy [12], that minimizes the cost function C (section 3.4). One of the drawbacks of the SA strategy is that in order to find near-optimal solutions it needs very large computation times. Such a strategy, although useful for the final stages of the system synthesis, cannot be used inside a design space exploration cycle.

MH, SA and AM have been used to map each of the 150 process sets representing $\Gamma_{current}$ on the existing applications ψ. For each of the resulted designs, the objective function *C* has been computed. Very long and expensive runs have been performed with the SA algorithm for each process set and the best ever solution produced has been considered as the near-optimum for that process set. We have compared the cost function obtained for the 150 process sets considering each of the three mapping algorithms. Figure 7b presents the average percentage deviation of the cost function obtained with the MH and AH from the value of the cost function obtained with the near-optimal scheme. We have excluded from the results in Figure 7b, 28 solutions obtained with AH for which the second design criterion for messages has not been met, and thus the objective function has been strongly penalized. The SA approach performs best in terms of quality at the expense of a large execution time, which can be up to 40 minutes for large sets of 320 processes. MH performs very well, and is able to obtain good quality solutions in a very short time, e.g., 6.5 seconds for 320 processes. AH is very fast, but since it does not address explicitly the design criteria presented in Section 3 it has the worst quality of solutions, according to the cost function.

The most important aspect of the experiments is determining to which extent the mapping strategies proposed in the paper really facilitate the implementation of future applications. To find this out, we have mapped process sets of 40, 80, 160 and 240 processes representing the $\Gamma_{current}$ application on top of the previously generated existing applications ψ. After mapping and scheduling each of these applications we have tried to add a new application Γ_{future} to the resulted system. Γ_{future} consists of a process set of 80 processes, randomly generated according to the same specifications presented before. The experiments have been performed two times, using first MH*[1] and then 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. Figure 8a shows the number of successful implementations in the two cases. In the case $\Gamma_{current}$ has been mapped with MH*, this means using the design criteria and metrics proposed in the paper, we were able to find a valid schedule for 56% of the total mapping attempts with IMS using Γ_{future}. However, using AH to map $\Gamma_{current}$ has led to a situation where IMS is able to find valid schedules in only 31% of the cases. Another observation from Figure 8 is that when the available utilization is large, as in the case $\Gamma_{current}$ has only 40 processes, it is easy for both MH* and AM to find a mapping that allows adding future applications. However, as $\Gamma_{current}$ grows to 80, only MH* is able to find a mapping of $\Gamma_{current}$ that supports an incremental design process, accommodating more than 60% of the future applications, while using AM only less than 25% are accommodated. If the remaining utilization is very small, after we map a $\Gamma_{current}$ of 240, it becomes practically impossible to map new applications without modifying the current system.

However, in the case the mapping heuristic is *allowed to modify* the existing system as discussed in this paper then we are able to increase the number of successfully mapped Γ_{future} applications to 73% from the total instead of only 56%. The percentage of accommodated Γ_{future} applications, for different dimensions of $\Gamma_{current}$, if modifications are allowed on the existing system, is shown by the diagram MH in Figure 8b. After mapping a $\Gamma_{current}$ with 80 processes using MH we

1. MH* is the same mapping heuristic as in Figure 5, but in which we do not allow modifications to the existing applications.

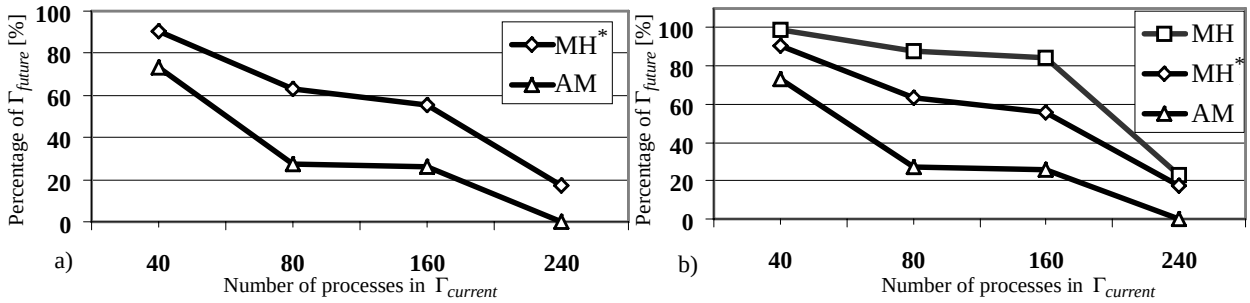


Figure 8. Percentage of Γ_{future} Apps. Successfully Mapped: a) No modifications b) Modifications Allowed

are able to accommodate 88% of the future applications, compared to only 61% in the case we do not allow modifications to the existing system (MH*). 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) modelled as a process set. The CC 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 system ψ consists of 80 processes generated randomly. 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 MH* we were able to later map 18 of the future applications, while using AH only 6 of the future applications could be mapped. MH* and AH do not consider modifications to the existing system. When modifications are allowed, using the MH approach, we are able to map 26 of the 30 future applications.

6. Conclusions

We have presented an approach to the incremental design of distributed hard real-time systems. Such a design process satisfies two main requirements when adding new functionality: the already running functionality is 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 was considered in the context of a fixed priority scheduling policy for processes and a static cyclic scheduling policy for messages. Scheduling of messages has been done using a realistic communication model based on a TDMA scheme.

We have introduced several design criteria with their corresponding metrics, that drive our mapping strategies to solutions supporting an incremental design process. For constructing an initial valid solution, we have shown that it is needed to take into account the features of the communication protocol.

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. AS is very fast but produces solutions that could be of too high cost, while SH is able to quickly produce good quality results. The approaches have been validated through several experiments.

References

[1] Audsley, N.C., Burns, A., Davis, R.I., Tindell, K., Wellings, A.J. 1995. Fixed Priority Preemptive Scheduling: An Historical Perspective. Real-Time Systems, 8(2/3), 173-198.

[2] Balarin, F., Lavagno, L., Murthy, P., Sangiovanni-Vincentelli, A. 1998. Scheduling for Embedded Real-Time Systems. IEEE Design and Test of Computers, 71-82, January-March.

[3] Dobrin R., Özdemir, Y., Fohler, G. Task Attribute Assignment of Fixed Priority Scheduled Tasks to Reenact Off-Line Schedules, In Proc. of RTCSA 2000 Korea , December 2000.

[4] Ermedahl, H., Hansson, H., Sjödin, M. 1997. Response-Time Guarantees in ATM Networks. Proceedings of the 18th IEEE Real-Time Systems Symposium, 274-284.

[5] Gutiérrez García, J.J., González Harbour, M. 1995. Optimized Priority Assignment for Tasks and Messages in Distributed Hard Real-Time Systems, Proc. 3d Workshop on Parallel and Distributed Real-Time Systems, 124-132.

[6] Kopetz, H., Grünsteidl, G. 1994. TTP-A Protocol for Fault-Tolerant Real-Time Systems. IEEE Computer, 27(1), 14-23.

[7] Liu, C.L., Layland, J.W. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM, 20(1), 46-61.

[8] Lonn, H., Axelsson, J. 1999. A Comparison of Fixed-Priority and Static Cyclic Scheduling for Distributed Automotive Control Applications. Proceedings of the 11th Euromicro Conference on Real-Time Systems, 142-149.

[9] Martello, S., Toth, P. 1990. Kanpsack Problems: Algorithms and Computer Implementations. Wiley.

[10] Pop P., Eles P., Peng Z. 1999. Schedulability-Driven Communication Synthesis for Time Triggered Embedded Systems. Proc. of the 6th International Conference on Real-Time Computing Systems and Applications, 287-294.

[11] Pop P., Eles P., Pop T., Peng Z. 2001. Minimizing System Modification in an Incremental Design Approach. Proceedings of the 9th Int. Symp. on Hardware/Soft. Codesign, 183-188.

[12] Reevs, C.R. 1993. Modern Heuristic Techniques for Combinatorial Problems. Blackwell Scientific Publications.

[13] Stankovic, J. A., Ramamritham, K. 1993. Advances in Real-Time Systems. IEEE Computer Society Press.

[14] Sha, L., Rajkumar, R., Lehoczky, J. 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers, 39(9), 1175-1185.

[15] Tindell, K., Burns, A., Wellings, A.J. 1992. Allocating Real-Time Tasks (An NP-Hard Problem made Easy). Real-Time Systems, 4(2), 145-165.

[16] Tindell, K., Burns, A., Wellings, A.J. 1995. Calculating Controller Area Network (CAN) Message Response Times. Control Eng. Practice, 3(8), 1163-1169.

[17] Tindell, K., Clark, J. 1994. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems, Microprocessing and Microprogramming, 40, 117-134.

Feedback Scheduling of Model Predictive Controllers

Dan Henriksson, Anton Cervin, Johan Åkesson, Karl-Erik Årzén

Department of Automatic Control
Lund Institute of Technology
Box 118, SE-221 00 Lund, Sweden
{dan,anton,jakesson,karlerik}@control.lth.se

Abstract

The paper presents some preliminary results on dynamic scheduling of model predictive controllers (MPC's). In model predictive control, the control signal is obtained by optimization of a cost function in each sample, and the MPC task may experience very large variations in execution time. Unique to this application, the cost function also offers an explicit on-line quality-of-service measure for the task. Based on this insight, a feedback scheduling strategy is proposed, where the scheduler allocates CPU time to the tasks according to the current values of the cost functions. Since the MPC algorithm is iterative, the feedback scheduler may also abort a task prematurely to avoid excessive input-output latency. A case study is presented, where the new approach is compared to conventional fixed-priority and earliest-deadline-first scheduling.

1. Introduction

Flexible scheduling of tasks with unknown or varying execution times has attracted considerable attention in the real-time research community during the last decade. The main motivation has been to reduce some of the pessimism that is inevitable when applying traditional real-time scheduling theory to such tasks. Design based on worst-case assumptions would in these cases lead to under-utilization of the computing resources. The research has resulted in a number of general approaches such as scheduling of imprecise computations [Liu *et al.*, 1991], statistical scheduling, e.g. [Tia *et al.*, 1995], and value-based scheduling, e.g. [Burns *et al.*, 2000].

In this work, we instead take the application as the starting point. Many controllers, including hybrid controllers and model predictive controllers, exhibit large variations in their execution time, e.g. [Årzén *et al.*, 1999]. In this paper, we will concentrate on scheduling of model predictive controllers (MPC's), which in the terminology of [Liu *et al.*, 1991] can be viewed as "milestone" tasks. In an MPC, the control signal is determined by solving a convex optimization problem in every sample. The result is gradually refined for each iteration in the optimization algorithm, up to a certain bound.

The model predictive control strategy has won widespread industrial use in recent years, e.g. [Richalet, 1993]. Since MPC is based on on-line optimization, it has traditionally been applied to plants with slow dynamics in the process industry. Today, faster computers have allowed MPC to be applied also to plants with faster dynamics. In [Dunbar *et al.*, 2002], for instance, MPC is applied to high-performance flight control experiments. The main advantages of MPC are its ability to handle constraints and its straightforward applicability to large, multi-variable processes.

The main problem with MPC, however, is the long and highly varying execution time of the control algorithm. The execution time in each sample depends on a number of factors: the state of the plant, the current and future reference values, the current disturbances acting on the plant, the number of active constraints on control signals and outputs, etc. The industrial practice has been to run the MPC algorithm on a dedicated computer, and to decrease the complexity of the problem so that overruns are avoided. The problem of scheduling the MPC as a real-time task has not been adequately studied, and scheduling of several MPC tasks on the same computer has never been considered.

In this paper, we take a first step towards a strategy for dynamic scheduling of model predictive controllers. The main idea is to use feedback from the optimization algorithms to determine (a) when to terminate the optimization and output the control signal, and (b) which of several MPC's that should be scheduled at a given time. One very nice feature of MPC is that it is not only *possible* to extract a real-world quality-of-service measure from the controller, but the control algorithm is indeed *based* on the same measure. This enables a very tight and natural connection between the control and the scheduling. In the paper, a simulation study is performed, where the new approach is compared to conventional fixed-priority and EDF scheduling. The results show that large improvements in control performance can be achieved by more dynamic scheduling.

1.1 Outline

The rest of this paper is outlined as follows. An introduction to model predictive control is given in Section 2. Section 3 discusses the concept of feedback scheduling, and also contrasts the current approach to some related work. Section 4 contains a case study, where an MPC for a quadruple-tank

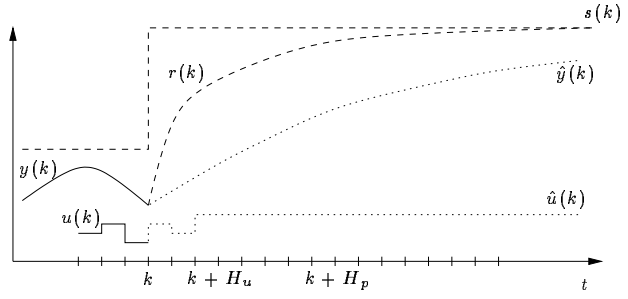


Figure 1 The basic principle of model predictive control.

process is investigated from a real-time perspective. Different scheduling approaches are compared in simulations. Section 5 provides further discussion and possible research directions, while Section 6 gives the conclusions.

2. Model Predictive Control

This section gives a brief overview of model predictive control. In short, MPC is based on three main concepts:

1. Explicit use of a model to predict the output of the controlled process at future discrete time instants, over a *prediction horizon*, H_p.
2. Computation of a sequence of future control actions over a *control horizon*, H_u, by minimizing a given objective function, such that the predicted process output is as close as possible to a desired reference signal.
3. *Receding horizon* strategy, so that only the first control action in the sequence is applied, the horizons are moved towards the future and the optimization is repeated in next sample.

The predicted output values, denoted $\hat{y}(k+i)$ for $i = 1, \ldots, H_p$, depend on the state of the process at the current time k (represented by a collection of past inputs and outputs) and on the future control signals $u(k+i)$ for $i = 0, \ldots, H_u - 1$. If H_u is chosen such that $H_u < H_p$, the control signal is manipulated only within the control horizon and remains constant afterwards, i.e., $u(k+i) = u(k+H_u-1)$ for $i = H_u, \ldots, H_p - 1$, see Figure 1.

The sequence of future control signals $u(k+i)$ for $i = 0, \ldots, H_u - 1$ is computed by optimizing a given cost function. The most often used cost functions are variations of the following quadratic function [Maciejowski, 2002]:

$$V(k) = \sum_{i=1}^{H_p} \alpha_i \left(r(k+i) - \hat{y}(k+i)\right)^2 + \sum_{i=0}^{H_u-1} \beta_i \Delta u(k+i)^2 \quad (1)$$

The first term accounts for minimizing the variance of the process output from the reference, while the second term represents a penalty on the control effort (related for instance to energy). The latter term can also be expressed by using u itself or other filtered forms of u, depending on the problem. The vectors α and β define the weighting of the output error and the control effort with respect to each other and with respect to the prediction step.

A major feature of the MPC algorithm, and the one that has contributed the most to its industrial acceptance, is its ability to handle *constraints*, e.g. saturations of control signals or output signals. These constraints are naturally specified as a part of the optimization problem. At the same time, it is the constraints that make the optimization problem so time-consuming to solve.

The MPC formulation above leads to a quadratic optimization problem with linear inequality constraints. The optimization theory offers strong results for such problems. For example, if a local minimum exists, then it must also be a global minimum. The problem is solved each sample with respect to the control signal increments $\Delta u(k+i)$ for $i = 0, \ldots, H_u - 1$. Only the control signal $u(k) = u(k-1) + \Delta u(k)$ is applied to the process. At the next sampling instant, the process output $y(k+1)$ is available and the optimization and prediction is repeated with the updated values according to the receding horizon principle. The control action $u(k+1)$ computed at time step $k+1$ will be generally different from the one calculated at time step k, since more up-to-date information about the process is available.

3. Feedback Scheduling

Traditional hard real-time scheduling theory assumes that a controller can be described as a periodic task with a fixed period T, a known worst-case computation time C, and a hard deadline D, such that $D = T$. Many controllers however, including hybrid controllers and model predictive controllers, do not fit the traditional task model very well. In particular, these controllers can exhibit very large variations in their execution time. Basing the real-time design on worst-case assumptions can lead to very conservative designs, slow sampling, and, in the end, poor control performance. On the other hand, basing the real-time design on average-case execution times can lead to temporary CPU overloads and starvation of some controllers during runtime. The result can, again, be poor control performance.

One way to handle the large variations in execution time is to introduce feedback in the real-time system. A schematic illustration of a general feedback scheduling system is shown in Figure 2. The idea is to feed back the actual use of critical resources (in our case, CPU time) to the scheduler and to continuously adjust the tasks' demand of resources according to the current situation. In some cases it is also possible to use feedforward, where the tasks can inform the scheduler that they are about to consume more resources.

In the case of control tasks, there are two main ways

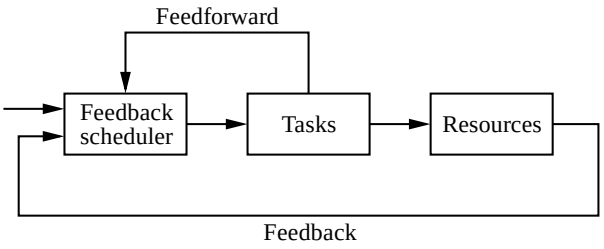


Figure 2 A general feedback scheduling system.

to control the CPU demand. First, one can manipulate the sampling periods of the controllers. This is possible since a controller can typically give satisfactory, although degraded, performance at lower sampling rates than it was designed for. A case study on hybrid controllers where this approach was taken was presented in [Cervin and Eker, 2000]. The feedback in this case consisted of execution-time measurements. The control tasks could also feedforward mode-change information to the scheduler. There was no feedback from the actual control performance.

For model predictive controllers, being milestone tasks, it is more natural to control the CPU demand by manipulating the execution time of the controllers. Also, the standard MPC formulation is based on a sampled-data description with a constant sampling interval, and it would be very difficult (and time-consuming) to allow on-line changes of the sampling period. In the approach suggested in this paper, a scheduling decision is made between each iteration of the optimization algorithm. The feedback information consists of the values of the cost functions of the controllers. No feedforward is needed since the scheduling decisions are so frequent, and since the cost functions automatically include all relevant information, including, for instance, all known future reference changes.

In [Stankovic *et al.*, 1999; Lu *et al.*, 1999], a scheduling algorithm that explicitly uses feedback in combination with EDF scheduling is presented. A PID controller regulates the deadline miss-ratio for a set of soft real-time tasks with varying execution times, by adjusting their requested CPU utilization. It is assumed that tasks can change their CPU consumption by executing different versions of the same algorithm. An admission controller is used to accommodate larger changes in the workload. In [Lu *et al.*, 2000] the same approach is extended. An additional PID controller is added that instead controls the CPU utilization. The two controllers are combined using a min-approach. The resulting hybrid controller scheme, named FC-EDF^2, gives good performance both during steady-state and under transient conditions. Although related to the work presented here, there are important differences. In our approach the tasks that are scheduled are controllers, controlling some physical plants. The performance, or *Quality-of-Control (QoC)* is explicitly used by the feedback scheduler to optimize the overall control performance.

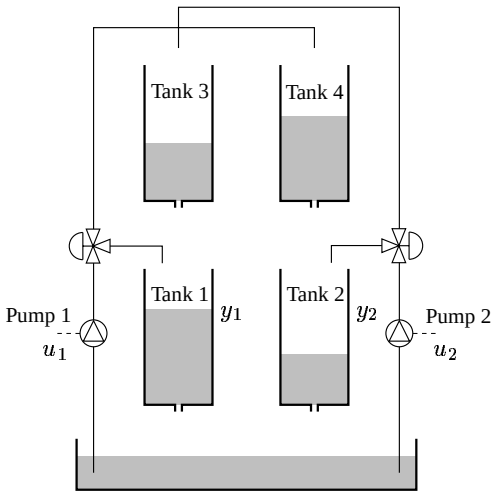


Figure 3 The quadruple-tank laboratory process. From [Johansson, 2000].

4. Case Study

This section contains a case study of a model predictive controller for a quadruple-tank process. The real-time properties of the controller are examined, and scheduling of first one and then two controllers under different strategies is investigated by simulation.

4.1 The Controlled Process

The process used in the case study is the quadruple-tank laboratory process, see Figure 3. This process was first presented in [Johansson, 2000] and is a good example of a multi-variable process. The goal is to control the level of the two lower tanks, y_1 and y_2, using the two pumps, u_1 and u_2. The flow from pump 1 is divided such that a fraction γ_1 enters tank 1 and $1 - \gamma_1$ enters tank 4. Likewise, the flow from pump 2 is divided such that a fraction γ_2 enters tank 2 and $1 - \gamma_2$ enters tank 3. The cross-coupling in the process makes it hard to achieve good performance using standard PID loops.

The process is linearized around a stationary point and is sampled with an interval of one second. This gives a standard discrete-time state-space model of the process to be used as the internal model by the MPC. In the following, y_1, y_2, u_1, and u_2 will denote deviations from the stationary point.

4.2 Real-Time Properties

A 100 seconds simulation scenario, see Figure 4, is studied, where at time t = 30 s, a step reference change is commanded in y_1. Also, a step load disturbance enters in y_2 at the same time. The disturbance is not modeled by the internal model described above, and will therefore make the optimization problem harder. The reference value for y_2 is zero throughout the simulation.

The result of an ideal simulation of the simulation scenario is given by the solid curves in Figure 4. The control perfor-

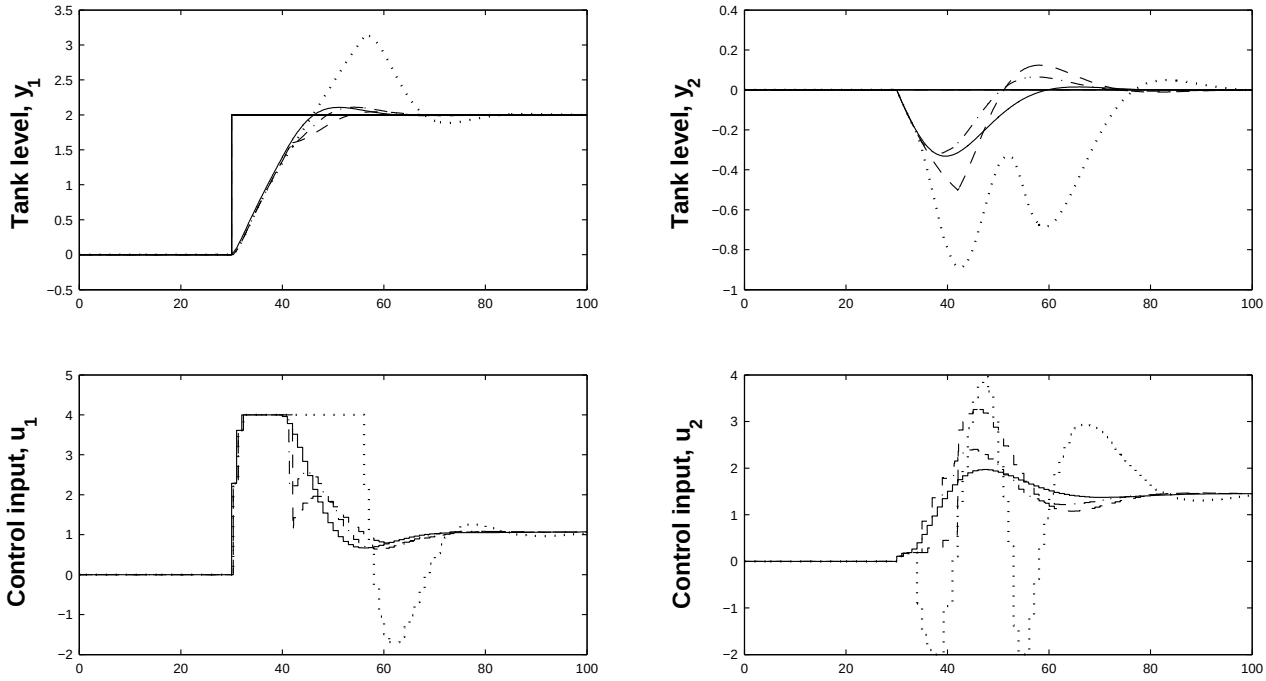


Figure 4 Control performance of a single MPC. The solid curve represents the ideal control performance obtained by not considering the computational delay, and letting the optimization algorithm run to termination in each sample. If the computational time is considered deadlines are missed and the performance degrades (dashed). Enforcing hard deadlines by aborting the optimization at the deadline gives even worse performance (dotted). Best performance is obtained by a dynamic stop criterion (dash-dotted).

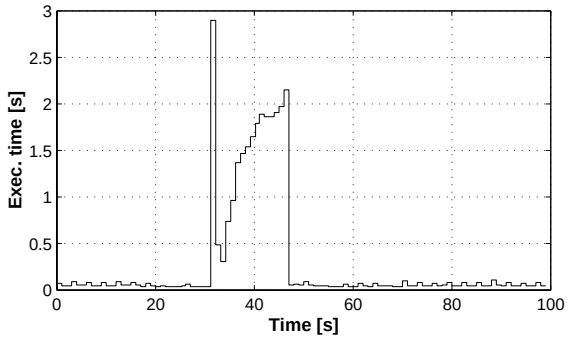


Figure 5 Execution time measurement for the MPC algorithm in the specific simulation scenario given by Figure 4.

mance is optimal and is obtained by letting the optimization algorithm run to termination in each sample and setting the computational delay to zero in the simulation.

A Java implementation of the controller was used to measure the execution time in the described scenario. The result is shown in Figure 5. A considerable difference in execution time can be noticed between different operating conditions. During steady-state operation, the required computation time is negligible, whereas during a step response it may be considerable longer than the period of the control task (one second). It also seen in Figure 4 that the input signal u_1 operates at its constraint (4 V). The active constraint makes the optimization problem harder and contributes to the increase in execution time.

The varying execution times is due to the large difference in the number of iterations required by the optimization algorithm. In the following simulations it is assumed that the execution time only depends on the number of iterations, and the execution time for each iteration is set to 40 ms.

Figure 6 shows a close-up of how the objective function (1) decreases for each iteration during one of the samples when the execution time is long. It can be seen that the cost function decreases monotonically but not smoothly by each iteration, which is due to the specific method used for the optimization. Different choices of optimization algorithms are discussed further in Section 5.3. While the objective function decreases with each new iteration, it is also expected that the true cost will increase because of the latency. This is illustrated schematically by the dashed curve in Figure 6. This fundamental trade-off between input-output latency and optimization will be studied in the examples below.

4.3 Simulation Environment

The MATLAB/Simulink-based simulator TRUETIME presented in [Henriksson *et al.*, 2002] is used to simulate controller task execution in a real-time kernel in parallel with the continuous-time dynamics of the quadruple-tank process. The detailed co-simulation makes it possible to study the effect of different scheduling policies and execution scenarios on the control performance. The real-time simulation environment also allows for system-level communication between the tasks, which will be used to obtain the feedback

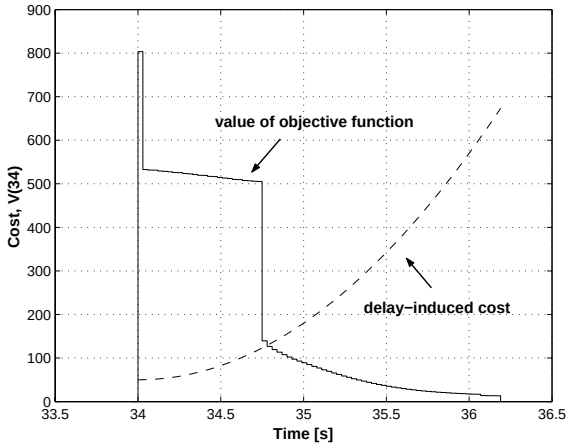


Figure 6 The solid curve shows a close-up of how the objective function (1) decreases for each iteration during a sample when the execution time is long. It is seen that the required execution time for full optimization is well above the period of one second. The dashed curve illustrates the expected loss due to delay.

connection between the controller tasks and the scheduler in the simulation of the feedback scheduling scheme.

4.4 A Single MPC Controller

To get an idea of how the input-output latency affects the control performance, we first investigate the case of a single MPC executing without interference from other tasks. The MPC task is implemented using the standard task model for periodic tasks, i.e., the task is released periodically with the period equal to the sampling interval of the controller, in this case one second. Furthermore the relative deadline of the task is equal to the task period.

In each instance the task will sample the process, perform the optimization, and finally output the computed control signal. The control signal is actuated as soon as the task has completed, i.e., as soon as the optimization has terminated or been aborted. A new instance may not start to execute until the previous instance has completed even if the task executes longer than its period, thus missing its deadline.

If a deadline is missed, which may very well be the case because of the long execution times, the task is immediately released again after completion and the release time is increased by the task period. This will make the task try to ”catch up” to missed deadlines. This situation is studied in a first simulation, where the optimization algorithm is allowed to run to termination in each instance. Since the execution time is sometimes longer than the task period, see Figure 5, there will be considerable sampling jitter and long delays. The result is degraded control performance, as shown by the dashed curves in Figure 4.

An obvious alternative would be to enforce hard deadlines by always terminating the optimization at the end of the period. In this example, however, this turns out to render even worse control performance, which is shown by the dotted curves in Figure 4.

In a third attempt, a dynamic stop criterion is used, where the optimization is terminated after at most two periods, or when the value of the objective function is below a certain threshold. The result is better performance, as shown by the dash-dotted curve in Figure 4. This simulation indicate that dynamic scheduling, where the MPC task is sometimes allowed to miss a deadline, may give better results than terminating the optimization at the deadline or allowing the optimization to run to completion. It is, however, not straightforward to determine when it is best to abort the optimization in relation to the input-output latency. Problems related to this are discussed in Section 5.1.

4.5 Fixed-priority Scheduling

We next consider scheduling of two MPC’s for two identical quadruple-tank processes on the same CPU. The controllers are implemented as synchronous periodic tasks according to the task model described above. The simulation scenario for the first controller is the same as before, whereas the reference change and step load disturbance occur ten seconds later for the second controller. The solid curves in Figure 7 show the result of an ideal simulation, with full optimization, no interference and no delay.

We first consider the situation where the tasks are scheduled in a priority-preemptive real-time kernel using distinct priorities. MPC 2 is given the highest priority, and no termination of the optimization algorithms is performed in this simulation. In addition to the delays caused by the long execution times, MPC 1 will now also experience delay due to preemption from the high-priority controller task. In the time interval 40-55 s, when both processes are in their transient phases, MPC 1 is preempted during significant amounts of time as seen in the computer schedule in Figure 8. The result is poor control performance as shown in Figure 7 (dotted).

4.6 EDF Scheduling

Next, consider scheduling of the two MPC’s using earliest deadline first scheduling. The results are given by the dashed curves in Figure 7, and it is seen that the performance is not considerably improved compared to fixed-priority scheduling. It is a well-known fact that EDF performs bad during overload, where the scheduled system often will experience a *domino effect* in missed deadlines. The degraded performance will in turn influence the execution times, since the actual process output will deviate from the predicted output. This increase in execution time will further worsen the situation. Figure 9 shows a close-up of the computer schedule in the interesting time interval. The required execution time decreases just before 60 seconds, and both tasks are then running frequently to try to ”catch up” to their missed deadlines.

4.7 Feedback Scheduling

The previous simulations indicate that fixed-priority scheduling and earliest deadline first scheduling may be

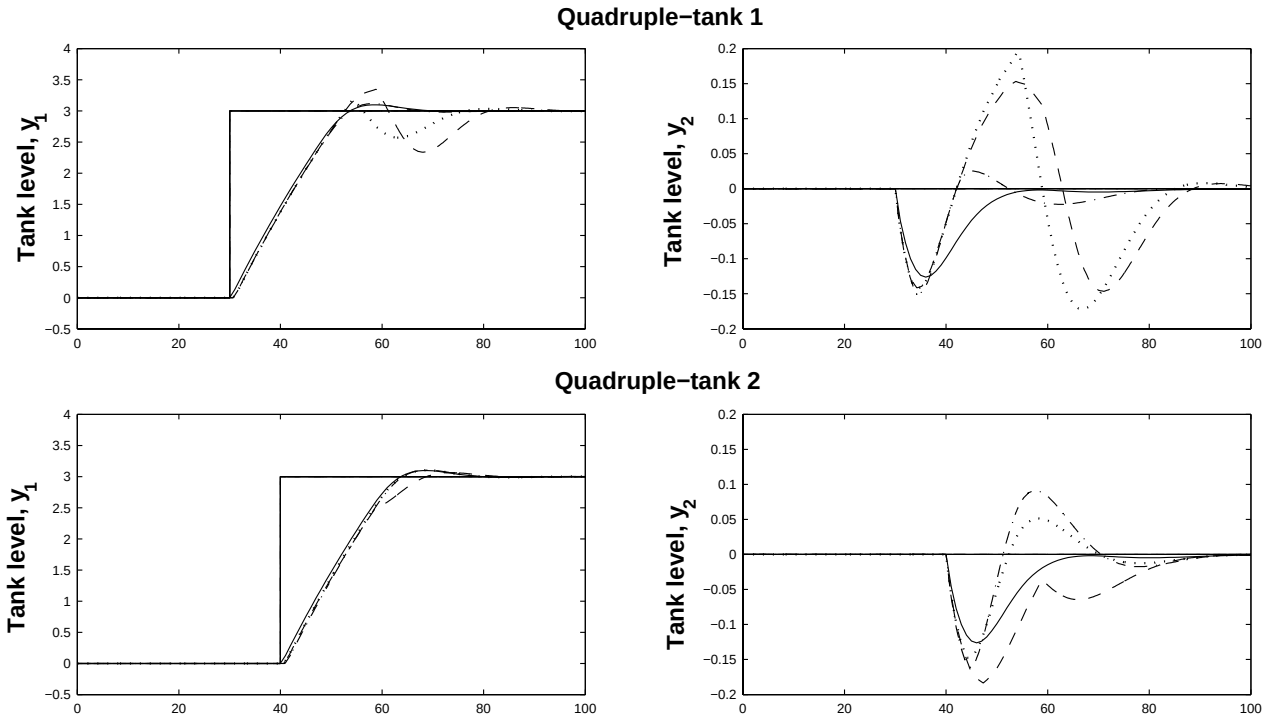


Figure 7 Control performance when scheduling two MPC tasks for two identical quadruple-tank processes. The solid curve shows the optimal performance, where computational delay and interference are neglected. The other curves show a comparison of control performance obtained by fixed-priority scheduling (dotted), EDF scheduling (dashed), and feedback scheduling (dash-dotted). The introduction of the feedback scheduler, using feedback from the cost functions, improves the control performance considerably.

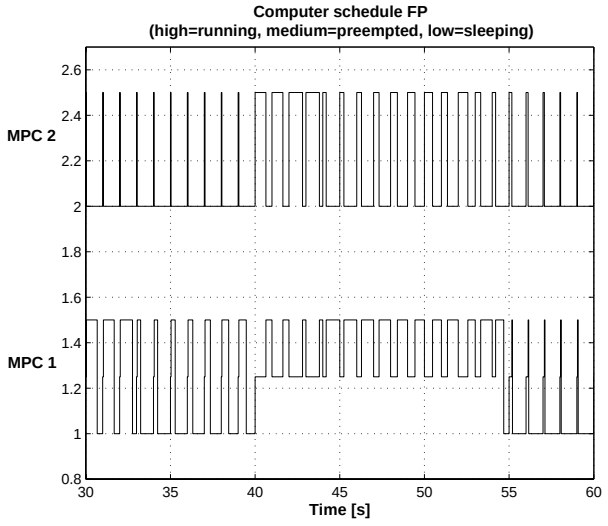


Figure 8 Close-up of the schedule under fixed-priority scheduling. The low-priority controller task (MPC1) is preempted during significant amounts of time with resulting poor control performance.

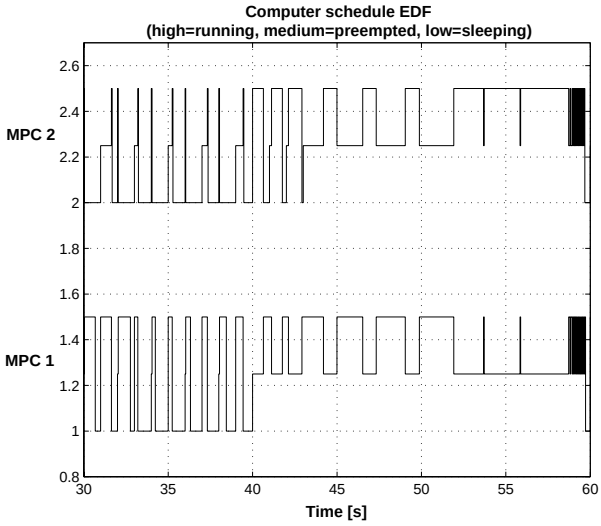


Figure 9 Close-up of the schedule under EDF scheduling. Between 40 and 60 seconds both controllers consume much computing resources and the system becomes heavily overloaded.

inappropriate for scheduling of MPC tasks. The main reason for this is that the relative importance of each task is dynamic and depends on reference changes, disturbances, etc. It is therefore impossible to do relevant off-line priority assignment and dynamic scheduling based on deadlines alone may not be enough. A better criterion to use in a dynamic scheduling scheme would be the value of the objective functions (1) for the respective controller tasks, which offer a nice on-line quality-of-service measure.

To improve the control performance a feedback scheduler is introduced, which uses feedback from the cost functions to dynamically schedule the two controllers. The feedback scheduler may abort the optimization of the controllers

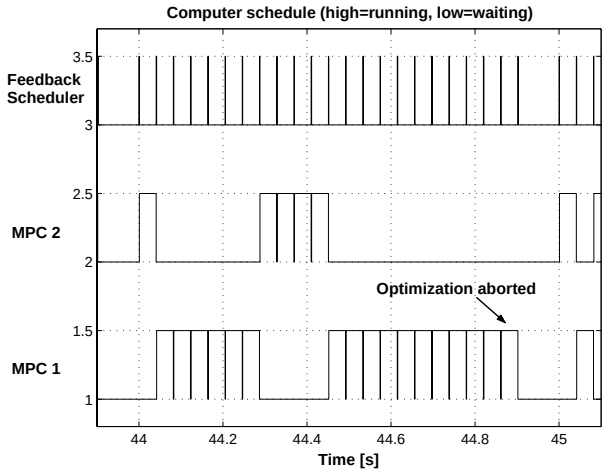


Figure 10 Close-up of the schedule under feedback scheduling. The feedback scheduler (top) distributes the computing on a per-iteration basis based on the values of the cost functions.

to cope with long execution times and to avoid excessive delays. Since both controllers are designed with the same control and prediction horizons, sampling intervals and weighting matrices, it is straightforward to compare the values of the respective cost functions. In other cases, some kind of scaling of the objective functions would have been required. This is discussed in Section 5.2.

The controllers are still implemented as synchronous periodic tasks, but their execution in each instance is governed by the feedback scheduler, which distributes computing resources among the running MPC tasks. The feedback scheduler makes decisions on a per-iteration basis based on the current values of the cost functions of the respective controllers. The MPC with the highest cost will be given the opportunity to perform one iteration. After the iteration the cost function has decreased, and the feedback scheduler will make a new decision as of which MPC to run. For stability reasons the highest priority is to make sure that all MPC's have feasible solutions, i.e., solutions that fulfill the constraints of the optimization problem. Therefore an infeasible solution is associated with an infinite cost.

After each iteration a decision will also be made whether or not to abort the optimization and actuate the plant. This decision depends on the current delay and is made for all active MPC controllers and not only for the one that previously executed. The logic of the feedback scheduler is described in pseudo-code below:

```
LOOP
  for (each active MPC controller) {
    if (delay > limit) {
      abort optimization and actuate plant;
    }
  }
  determine MPC task #i with highest cost;
  let MPC task #i perform one iteration;
END
```

The feedback scheduler and the MPC tasks communicate and synchronize their execution using events. After each invocation of the feedback scheduler it notifies the MPC task that is about to run an iteration. After the iteration the feedback scheduler is notified by another event and repeats the procedure described above.

Simulation results when using the proposed scheme are given by the dash-dotted curves in Figure 7. It can be seen that the control performance has improved, especially so for MPC controller 1. A close-up of the distribution of computing resources in the sample between 44 and 45 seconds is shown in Figure 10. Here it is seen how the execution is divided between the MPC tasks on a per-iteration basis.

5. Discussion

The simulation results indicate that dynamic scheduling based on feedback from cost functions may be a successful way of dealing with the problem of limited computational time when implementing model predictive controllers. However, a number of non-trivial issues have to be addressed to be able to apply the suggested scheme in a more general setting. Some of these are discussed below.

5.1 Computational Delay

In the MPC formulation used in this paper the computational delay was not accounted for. Standard practice is to include a one-sample delay in the process description and then synchronize the writing of the outputs with the reading of the inputs to enforce this. The computational delay, however, could vary from a very small fraction of the sampling interval up to several sampling intervals. In the dynamic schemes presented in the paper, the control signal was actuated as soon as the optimization terminated, not to induce any unnecessary delay that degrades the performance. Ideally, this should be combined with an adjustment of the prediction matrices in the next sample according to the actual delay.

Another issue is the trade-off between delay and cost during optimization. As time goes, the cost function decreases, but there will also be a penalty due to the input-output delay, see Figure 6. If this penalty could be estimated (as a function computed off-line) it could be possible to terminate the optimization when the cost function plus the penalty starts to increase instead of decrease. Or it could perhaps be possible to include an additional term in the cost function that takes the computational delay into account.

5.2 Comparing Cost Functions

When scheduling several MPC tasks, the strategy suggested in this paper was to give priority to the controller with the highest current value of its cost function. However, comparing cost functions directly may not be appropriate when the controllers have different sampling intervals, prediction horizons, magnitude of disturbances, etc. In those cases, it would be necessary to scale the cost functions to obtain a

fair comparison. The scheduling could also use feedback from the derivatives of the cost functions, as well as the relative deadlines of the different controllers.

5.3 Different QP-Solvers

There exist two major families of methods for solving quadratic optimization problems with linear inequality constraints, see for example [Maciejowski, 2002]. The traditionally most used is the *Active Set* method, which was used in the examples in this paper. In this algorithm an active set, the set of active inequality constraints, is introduced. As the algorithm proceeds, constraints are added and removed from the active set until the optimal solution is found. A drawback with this method, as seen in Figure 6, is that the cost function may decrease very irregularly as the number of active constraints changes. This makes it difficult to know how close the optimum is and whether it will pay off to optimize further.

In recent years *Interior Point* methods have won widespread use as an alternative to active set methods. Interior point methods may be more suitable in a dynamic setting, in that the cost typically decreases more smoothly by each iteration. It is then easier to estimate how much it will pay off to optimize further—the scheduler could look at the time derivatives of the cost functions to decide which MPC that should run.

For most optimization problems it is possible to formulate another, often simpler problem, called the *dual problem*. One property relating the original problem and its dual, is that their respective objective functions obtain the same value at the optimum. A particularly attractive feature of certain interior point methods [Wright, 1997] is that the algorithm offers an estimation of the difference between the values of the respective objective functions at each iteration. This is a useful feature, since it gives an indication of how close to the optimal point the solution at hand is, and may be used to decide whether to terminate the algorithm or not. This could be a better indication to the scheduler of what MPC task that needs attention than just looking at the current cost.

As described above, premature termination of an optimization run in one MPC may be justified in order to improve the overall control performance. Given that the algorithm at hand has found a feasible solution (this is considered as a requirement), any of the algorithms may be terminated before the optimum is found. The quality of the solution is then determined by how close to the optimum the solution is. Potentially, this means that an interior-point method is preferable, since it offers an estimation of how far off from the optimal value a solution at a given iteration is.

6. Conclusions

The paper has discussed feedback scheduling of model predictive controllers, and the potential of the suggested approach has been illustrated by simulations. A case study showed that traditional scheduling approaches, such as fixed-priority scheduling and EDF scheduling, may be inappropriate for scheduling of model predictive controllers. The proposed scheduling approach uses feedback from the cost functions that are used explicitly in the controller algorithms. The feedback scheduler may also abort a task to reduce the input-output latency.

References

Årzén, K.-E., B. Bernhardsson, J. Eker, A. Cervin, K. Nilsson, P. Persson, and L. Sha (1999): "Integrated control and scheduling." Technical Report TFRT-7586. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Burns, A., D. Prasad, A. Bondavalli, F. D. Giandomenico, K. Ramamritham, J. Stankovic, and L. Stringini (2000): "The meaning and role of value in scheduling flexible real-time systems." *Journal of Systems Architecture*, **46**, pp. 305–325.

Cervin, A. and J. Eker (2000): "Feedback scheduling of contol tasks." In *Proceedings of the 39th IEEE Conference on Decision and Control*, pp. 4871–4876.

Dunbar, W. B., M. B. William, R. Franz, and R. M. Murray (2002): "Model predictive control of a thrust-vectored flight control experiment." In *Proceedings of the 15th IFAC World Congress on Automatic Control*. Barcelona, Spain.

Henriksson, D., A. Cervin, and K.-E. Årzén (2002): "Truetime: Simulation of control loops under shared computer resources." In *Proceedings of the 15th IFAC World Congress on Automatic Control*. Barcelona, Spain.

Johansson, K. H. (2000): "The quadruple-tank process: A multivariable process with an adjustable zero." *IEEE Transactions on Control Systems Technology*, **8:3**.

Liu, J., K.-J. Lin, W.-K. Shih, A. Yu, J.-Y. Chung, and W. Zhao (1991): "Algorithms for scheduling imprecise computations." *IEEE Transactions on Computers*.

Lu, C., J. Stankovic, T. Abdelzaher, G. Tao, S. Son, and M. Marley (2000): "Performance specifications and metrics for adaptive real-time systems." In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pp. 13–23.

Lu, C., J. Stankovic, G. Tao, and S. H. Son (1999): "Design and evaluation of a feedback control EDF scheduling algorithm." In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pp. 56–67.

Maciejowski, J. M. (2002): *Predictive Control with Constraints*. Prentice-Hall.

Richalet, J. (1993): "Industrial application of model based predictive control." *Automatica*, **29**, pp. 1251–1274.

Stankovic, J. A., C. Lu, S. H. Son, and G. Tao (1999): "The case for feedback control real-time scheduling." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 11–20.

Tia, T.-S., Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W.-S. Liu (1995): "Probabilistic performance guarantee for real-time tasks with varying computation times." In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*.

Wright, S. J. (1997): *Primal-Dual Interior-Point Methods*. SIAM.

A Computational Model for Real-Time Control Tasks

(extended abstract)

Anton Cervin and Johan Eker

Department of Automatic Control
Lund Institute of Technology
Sweden

Summary

We propose a computational model for real-time control tasks which combines ideas from the synchronized I/O model of Giotto [Henzinger *et al.*, 2001] with the CPU resource reservation model of the Constant Bandwidth Server (CBS) [Abeni and Buttazzo, 1998]. The goal of the model is to facilitate co-design of flexible real-time control systems. In particular, the model should provide

- a simple interface between control design and real-time design
- minimal sampling jitter and output jitter
- short input-output latency
- isolation between unrelated tasks (including non-control tasks)
- predictable control and real-time behavior during overruns
- a possibility to combine several tasks (components) into a new task (component) with predictable control and real-time behavior
- a possibility to adapt to changing CPU load to provide optimal control performance

Background

Traditional scheduling models give poor support for the design of multi-threaded control systems. One difficulty lies in the nonlinearity in dynamic scheduling mechanisms such as rate-monotonic or earliest-deadline-first scheduling: a small change in a task parameter, e.g. period, execution time, deadline, priority, etc., may give rise to unpredictable results in terms of input-output latency and jitter. This is crucial, since the performance of a controller depends not only on its sampling period, but also on the input-output latency and the jitter. In the control design, it is straight-forward to account for a constant latency, while it is difficult to address varying or unknown delays. On the other hand, static scheduling may simplify the control design problem (because of the predictable delays) but cannot handle overruns or adaptation to changing CPU load.

Model Overview

Combining the best of two worlds, we propose the use of static (time-triggered) scheduling for I/O (communication with the environment and other tasks), and dynamic scheduling (based on CBS) for all computations in between. We use CBS servers to make each task appear as if it were running at a given fraction of the CPU speed. This means that a task that is given x% of the CPU will appear as if it were running as a lone thread on a machine with only x% of the original capacity. The task is only allowed to communicate with the environment or other tasks at specified *interaction points*. A key property of the model is that, for each task, time only has to progress linearly with respect to its interaction points. Between points, the CBS scheduling algorithm (which is based on EDF) is used to efficiently schedule the pending computations. All I/O is handled by the kernel (which executes at the highest priority) and is hence not prone to jitter.

Task Model

Each task τ_i has a period T_i and a CPU share U_i. To facilitate short latencies (contrary to e.g. Giotto which imposes a one-sample delay) the task is divided into $l_i \geq 1$ segments $S_i^1, \ldots, S_i^{l_i}$. The lengths of the segments are expressed as fractions of the task period, $s_i^1, \ldots, s_i^{l_i}$, $\sum_{j=1}^{l_i} s_i^j = 1$. Within a segment S_i^j, the task is guaranteed (by the CBS scheduling algorithm) to be able to execute for a time $s_i^j U_i T_i$. The beginning of a segment may be declared as a *read point*, where shared variables or physical inputs are read. The end of a segment may be declared as a *write point*, where shared variables or physical outputs are written.

An example of a task with two segments in shown in Figure 1. In the first segment, an input is read, some computations are performed, and an output is written. In the second segment, some more computations are performed. This is a typical model of a controller whose computations can often be split into two parts, Calculate Output and Update State, in order to minimize the input-output latency. In this example, the latency will be equal to $s^1 T$. If the worst-case execution time of the two segments are bounded by $s^1 U T$ and $s^2 U T$ respectively, the controller will have a perfectly predictable run-time behavior, regardless of the execution of the other tasks in the system.

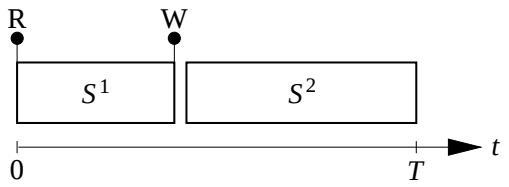


Figure 1 An example of a task divided into two segments. The first segment has two interaction points: a read point (R) in the beginning and a write point (W) at the end.

Task are allowed to communicate with each other through shared variables. To facilitate minimal end-to-end latencies, a task τ_i can be given an optional release offset O_i (which defaults to zero). If the write point of one task should occur at the same time as the read point of another task, the write action is guaranteed to take place first.

An example of two communicating tasks are shown in Figure 2. This could model for instance a cascade controller, where the inner controller (τ_2) executes twice as frequently as the outer controller (τ_1). After executing segment S_1^1, the outer controller writes a control signal to a shared variable which will be read and used as a reference value in the inner controller. To minimize the end-to-end delay, task τ_2 is given an offset $O_2 = s_1^1 T_1$.

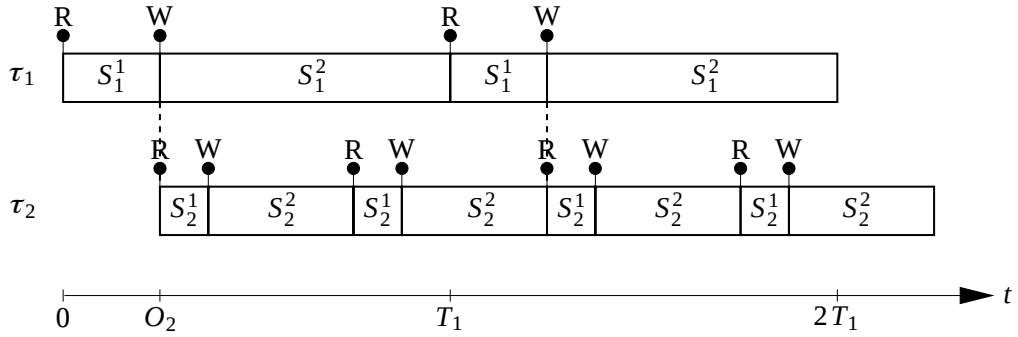


Figure 2 Communicating controller tasks. Task τ_2 is given an offset such that the control signal written by S_1^1 is immediately read by S_2^1.

Notice that, taken together, τ_1 and τ_2 can be viewed as a new task (or component) with the CPU share $U_1 + U_2$.

Control and Scheduling Co-Design

The great advantage of the proposed computational model is that we get a simple control and scheduling co-design problem, which still takes the implementation into account. If we ignore the overhead associated with the I/O operations and the CBS servers, schedulability of the task set is simply a function of the total CPU demand $U = \sum_i U_i$. Just as for plain EDF, the system is schedulable iff $U \leq 1$ [Abeni and Buttazzo, 1998].

The optimal performance of each controller is also just a function of its CPU share U_i[1]. Given a share U_i, it is possible to select a task period T_i and segment sizes $s_i^1 \ldots s_i^{l_i}$ such that the control performance is optimized. In the case of constant execution times $C_i^1 \ldots C_i^{l_i}$ of the segments, the choices that minimize the period and the latency without missing any deadlines are given by $T_i = \frac{1}{U_i} \sum_{j=1}^{l_i} C_i^j$ and $s_i^j = \frac{1}{U_i} C_i^j$. The latency of the controller will be known and can be compensated for in the control design.

For linear controllers, it is possible to compute a quadratic performance index for the controller as a function of the sampling period, the input-output latency, and the jitter [Lincoln and Cervin, 2000]. In our model, however, the jitter is eliminated and the optimal period and latency is determined by the CPU share U.

To summarize, in our model, both schedulability and control performance are functions of the CPU shares U_i only. Each controller can be designed independently of the others, compensating for a known latency.

An Example

As an example, we will look closer at the cascaded controller mentioned above. The principle of the controller is shown in Figure 3. The cascaded

[1]Assuming that shorter period implies better performance and also that shorter latency implies better performance.

controller consists of two components: the outer controller (Ctrl 1) and the inner controller (Ctrl 2). The inner controller handles the fast part of the process dynamics (G_ϕ), and is assumed to have twice the sampling frequency of the outer controller which handles the slower dynamics (G_x).

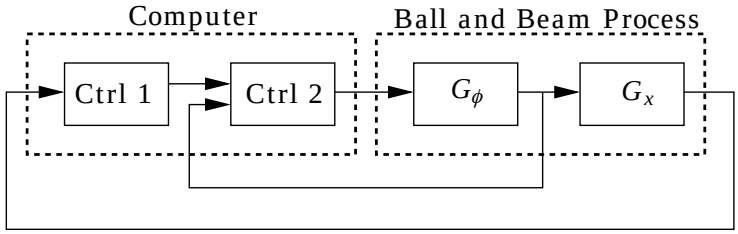


Figure 3 A cascaded controller structure. The inner controller has twice the sampling frequency of the outer controller.

Both controllers execute the same PID control algorithm. Each period, the controller should

1. Read the measurement signal and the reference value
2. Calculate control signal (execution time: 0.4 ms)
3. Write the control signal
4. Update the controller state, log data, etc. (execution time: 1.2 ms)

In the our model, each controller is modeled as a task τ_i with two segments, Calculate Output (S_i^1) and Update State (S_i^2). To use the computing resources optimally, the lengths of the segments are proportional to the execution times: $s_i^1 = 1/4$ and $s_i^2 = 3/4$. There is a read point at the beginning of Calculate Output, where the reference value and the measurement value are read, and there is a write point at the end of Calculate Output where the control signal is written. The control signal of Ctrl 1 acts as the reference signal to Ctrl 2, so this is a shared variable.

To minimize the control latencies, Ctrl 2 should execute its first Calculate Output right after Ctrl 1 has finished its Calculate Output. This is achieved by assigning the offsets $O_1 = 0$ and $O_2 = s_1^1 T_1$.

Now, suppose that 60 % of the total CPU resources are available for the cascade controller, while the rest is needed for other tasks. We would then assign the shares $U_1 = 0.2$ and $U_2 = 0.4$ to the controller tasks. To use the resources fully, we assign the periods $T_1 = 8$ ms and $T_2 = 4$ ms. Note that, should we choose to assign more or less resources to the cascade controller, these numbers are simply rescaled by a factor.

The example has been implemented in the TRUETIME simulator [Henriksson *et al.*, 2002]. A schedule simulation with the two control tasks and a kernel task handling the I/O (now more realistically assumed to take 0.1 ms to execute) is shown in Figure 4. Also shown is the linear CPU time as perceived by the tasks, together with the actual CPU time, and the interaction points.

Note that the remaining 40 % of the CPU can be used by other tasks, without affecting the behavior of the controllers. The actual CBS schedule will change, but the interaction points will remain the same and the segments will finish their computations in time.

Again, note that τ_1 and τ_2 can be viewed as *one component* that consumes 60 % of the CPU. Using JITTERBUG [Lincoln and Cervin, 2000] it

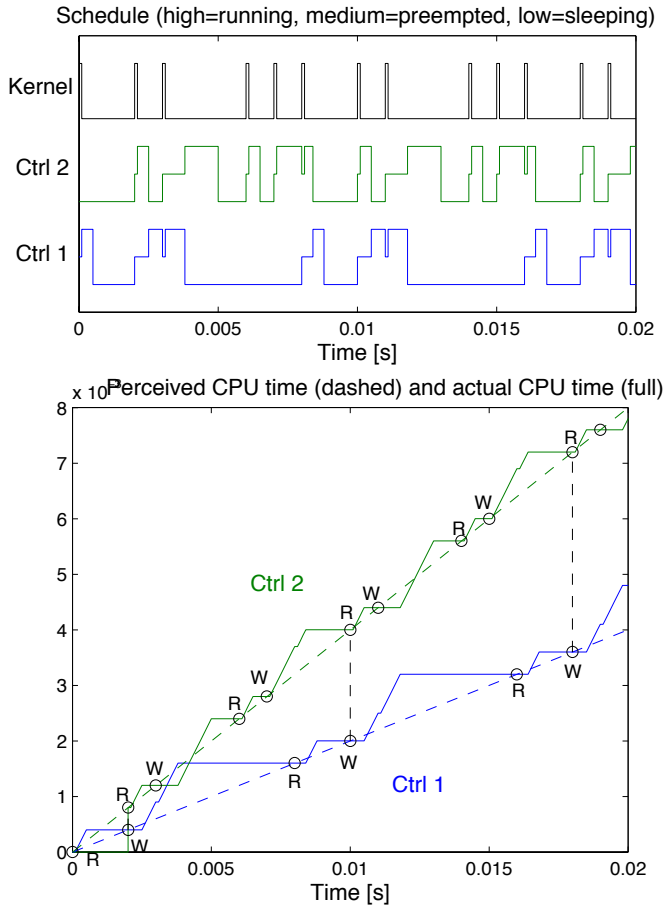


Figure 4 *Above:* The task schedule generated by CBS servers. *Below:* Perceived linear CPU time, actual CPU time, and interaction points for the controller tasks. Notice that the interaction points are always located on the linear timelines.

is possible to compute a quadratic control performance index also for this type of multi-rate controller. As an example, in Figure 5, a cost criterion on the form

$$J(U) = \lim_{T\to\infty} \frac{1}{T} \mathbf{E}_{v,w} \left\{ \int_0^T \left(x^T(t) Q_1 x(t) + u^T(t) Q_2 u(t) \right) dt \right\}$$

has been computed for different values of the total CPU share $U = U_1 + U_2$.

Future Work

Much work remains before the model will be useful. Some things are outlined below.

Overrun Handling If a control task does not finish its computations in time, the CBS server will postpone its deadline to prevent other tasks from suffering. This can mean the controller will have to continue its computations in the next period. There is then the choice between finishing the computation that was started in the last period, or to read a new input and restart the computation. Calculations have shown that the best choice will be different from controller to controller. In each case, the situation must be handled differently in the controller code.

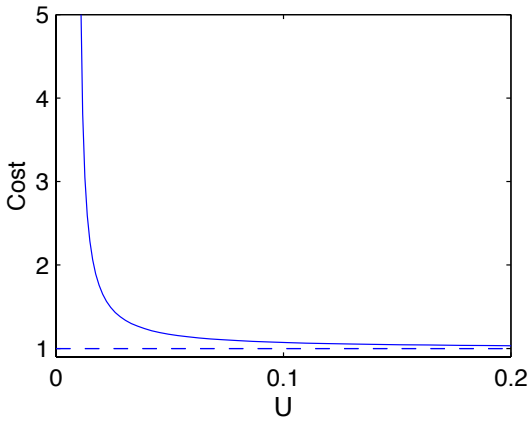


Figure 5 Cost vs total CPU share for the cascaded controller. The dashed line indicates the performance of a continuous-time controller (infinite CPU resources).

Adaptive Resource Distribution When the CPU demand of the tasks in the system change (e.g. when new tasks enter the system or when control tasks switch modes) the CPU resources should be redistributed so that the overall control performance is optimized. Changing the CPU shares of the CBS servers at run-time can lead to transients, however. Perhaps the transients can be tolerated, but this must be investigated further.

Implementation Several issues retaining to the implementation also have to be resolved. As a first step, we will implement the full model in the TRUETIME simulator [Henriksson *et al.*, 2002].

Conclusions

We have outlined a computational model for real-time control tasks, with the primary goal of simplifying the control and scheduling co-design problem. Using a model where time appears to progress linearly for all components (when viewed in the interaction points only) the low-level scheduling details can be eliminated from the design process.

References

Abeni, L. and G. Buttazzo (1998): "Integrating multimedia applications in hard real-time systems." In *Proceedings of the 19th IEEE Real-Time Systems Symposium*.

Henriksson, D., A. Cervin, and K.-E. Årzén (2002): "Truetime: Simulation of control loops under shared computer resources." In *Proceedings of the 15th IFAC World Congress on Automatic Control*. Barcelona, Spain.

Henzinger, T. A., B. Horowitz, and C. M. Kirsch (2001): "Giotto: A time-triggered language for embedded programming." In *Proceedings of EMSOFT 2001*.

Lincoln, B. and A. Cervin (2000): "Jitterbug: A tool for analysis of real-time control performance." Submitted to the 2002 Conference on Decision and Control.

Graduate Student Conference 2002

QoS Management in Real-Time Data Services

Prof. Sang Hyuk Son
Dept. of Computer Science
University of Virginia
URL: http://www.cs.virginia.edu/~son

Abstract
Many real-time systems are now being used in safety-critical applications, in which human lives or expensive machinery may be at stake. Examples include aerospace and defense systems, industrial automation, traffic control, web-based information services, etc. The unpredictable demands from the operating environments of such systems often pose rigid requirements on their timely performance. These requirements are defined as real-time constraints on their temporal behavior. As real-time systems continue to evolve, their applications become more complex, and often require timely access to temporal data, possibly from sensor devices. This need for advanced data services in real-time applications poses intellectual and engineering challenges to researchers and practitioners. The trade-offs faced by the designers of real-time data services are different from those faced by the designers of general-purpose database systems.

In this talk, we first discuss characteristics of real-time systems and some of the research issues of real-time data services. We then present research work being performed at the University of Virginia, especially on supporting QoS in real-time data services.

Invitation, Schedule

Updated: 14-Mar-2002 14:48 by Roland Grönroos
e-mail: artes@docs.uu.se web: www.artes.uu.se
Location: http://www.artes.uu.se/events/gsconf02/sang.shtml

Swedish Foundation for Strategic Research

Efficient and Simple Implementations of the Wait-Free Queue Classes of the Real-Time Specification for Java*

Philippas Tsigas
Department of Computing Science
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
tsigas@cs.chalmers.se

Yi Zhang
Department of Computing Science
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
yzhang@cs.chalmers.se

ABSTRACT

The Real-Time Specification for Java provides protected, non-blocking, shared access to objects accessed by both regular Java threads (`java.lang.Threads`) and the time-critical `NoHeapRealtimeThreads`. Such access is offered via a set of *wait-free queue* classes. These classes are provided explicitly to enable communication between the real-time `NoHeapRealtimeThreads` and the regular Java threads; they have a unidirectional nature with one side of the queue (read or write) for the real-time threads and the other one (write or read, respectively) for the non-real-time ones. Efficient implementations of these queue classes are presented in this paper. The implementations are designed to have the unidirectional nature of these queues in mind and they are more efficient, with respect to space complexity, compared to previous general bi-directional wait-free implementations, without losing in time complexity.

Yi Zhang is a full time graduate student.
This paper is submitted for regular presentation.

1. INTRODUCTION

Although Java was originally designed by Sun to facilitate the development of embedded system software [12], it was designed more to simplify programming than to enable programmers to write software that complies reliably with real-time constraints. In order to facilitate its major goal of operating system and hardware independence, in areas such as thread behaviour, synchronization, interrupts, memory management, and input/output Java's expressiveness was designed to be weak. However, these are among the critical areas needing explicit management for meeting application timeliness requirements. On the other hand because the simplicity, the object orientation appeal and most significantly the Java platform's independence offer, greater cost-savings potential in the real-time domain than in the desktop and server domains: real-time computing systems use many different processor types and operating systems. The matching between Java and real-time software development led to the formation of the Real-Time for Java Experts Group (RTJEG) that began developing the Real-Time Specification for Java (RTSJ) in March 1999 under the Java Community Process [7]. The goal, was to provide a platform that let programmers correctly reason about the temporal behaviour of executing software. In their effort to do so, they enhanced Java in 7 areas [8]: 1) thread scheduling and dispatching, 2) memory management, 3) synchronization and resource sharing, 4) asynchronous event handling, 5) asynchronous transfer of control, 6) asynchronous thread termination, 7) physical memory access.

In the area of synchronization and resource allocation the RTSJ introduces a set of new synchronisation mechanisms, a set of wait-free queue classes. This wait-free queue classes became obligatory for the support of protected, concurrent access of data by both regular threads (`java.lang.Threads`) and the time-critical `NoHeapRealtimeThreads`[1] that were introduced by the RTSJ to have an implicit execution eligibility logically higher than the garbage collector. `NHRT`s can not access (allocate or even reference) any objects in the heap, which means that they should be able to run while the garbage collector is running. `NHRT`s are introduced for code with a very low tolerance of non-scheduled delays. The RTSJ does not mandate algorithms or specific time constants for such, but requires that the semantics of the implementation are met.

In this work, we present implementations of the wait-free queue classes of the Real-time Specification for Java. These implementations are designed to have the unidirectional nature of these queues in mind and they are more efficient, with respect to space, compared to previous general bi-directional wait-free implementations, without losing in time complexity.

1.1 Wait-Free Synchronization

In the area of synchronization and resource allocation applications often need to share serializable resources. Java also

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

[1]We are going to use the `NHRT` term to denote the `NoHeapRealtimeThread` for the rest of the paper.

provides the ability to introduce concurrency mechanisms into applications. Traditionally, concurrency mechanisms for synchronization and resource allocation use mutual exclusion to protect the consistency of the shared data by allowing only one process at a time to access the method/class. If one declares a method to be `synchronized`, Java will prevent more than one thread from executing that method at any time. The keyword `synchronized` is the only mechanism required by the specification that can enforce mutual exclusion in the traditional sense in RTSJ as in Java. Mutual exclusion i) causes large performance degradation; ii) leads to a complex scheduling analysis since tasks can be delayed, because they were either preempted by other more urgent tasks, or because they are blocked before a critical section by another process that can in turn be preempted by another more urgent task and so on (This is also known as the convoy effect [15]); iii) and most significantly for the real-time systems it leads to priority inversion in which a high priority task can be blocked for an unbounded time by a lower priority task [21]. Several synchronization protocols have been introduced to solve the priority inversion problem for uniprocessor [21] and multiprocessor [18] systems. The solution presented in [21] solves the problem for the uniprocessor case with the cost of limiting the schedulability of task sets and also making the scheduling analysis of real-time systems hard. The situation is even worse in a multiprocessor real-time system, where a task may be blocked by another task running on a different processor [18]. For the RTSJ, it was decided that the least intrusive specification for allowing real-time safe synchronization is to require that implementations of the Java keyword `synchronized` includes one or more algorithms that prevent priority inversion among real-time Java threads that share the serialized resource. But still the use of the `synchronized` keyword implementing the required priority inversion algorithm was not sufficient to both prevent priority inversion and allow the special `NHRT`s, that were introduced in the RTSJ to give the means to time-critical tasks to get an execution eligibility logically higher than the garbage collector, to do so [7].

Wait-free implementation of shared data objects is an alternative approach for the problem of inter-task communication and synchronization. Wait-free mechanisms allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. A wait-free implementation of a shared data object guarantees that every process accessing the object always completes its operation in a bounded number of its own steps, regardless of interleaving (process halts, failures, scheduler behaviour). Wait-free inter-task communication does not allow one task to block another task and thus gives significant advantages over lock-based schemes because:

1. it does not give priority inversion and avoids lock convoys that make scheduling analysis hard and delays longer.
2. it provides high fault tolerance (processor failures will never corrupt shared data objects) and eliminates deadlock scenarios from two or more tasks both waiting for locks held by the other.
3. more significantly, it completely eliminates the interference between process schedule and synchronization; thus, giving a more compositional framework to argue about the 'task' behaviour under the effect of the scheduler and the synchronization mechanism. This gives the ability to a task to keep its execution eligibility during communication and synchronisation, and this was the feature that was incumbent in the RTSJ.

All the above mentioned advantages come from the fact that wait-free solutions are not penalised from the negative effects of blocking.

The decision of the RTJEG to provide wait-free queues to enable communication between the regular Java threads and the real-time `NHRT` follows research in recent years, in which several researchers have investigated the use of wait-free shared-object algorithms as an alternative to lock-based mechanisms in object-based real-time systems [2, 3, 4, 5, 6, 10, 20, 11, 24, 22, 23]. Moreover, research in real-time operating systems [9, 16, 13] has also shown how to incorporate wait-free techniques in real-time kernels.

1.2 Related Work and Our Contribution

Concurrent FIFO queue data structures are fundamental data structures used in many programs and algorithms and, as can be expected, many researchers have proposed implementations for them. But, although there are many non-blocking implementations (see [25] for references), there are only few wait-free. In a non-blocking algorithm, some operations are allowed to perform unbounded number of steps when they are concurrent with other operations; this, of course, is not allowed in a wait-free algorithm. Lamport [17] introduced a wait-free queue that allows only one enqueuer and one dequeuer. All previously mentioned constructions (wait-free or not) were targeted towards asynchronous systems; such constructions require hardware support for strong synchronization primitives such as `Compare-and-Swap` etc. These primitives are not available in the Real-Time Specification for Java. As a matter of fact in the RTSJ only read and write memory operations are supported. The reason is the hardware-independence property that the RTSJ wants to preserve.

Recent research at the University of North Carolina has shown that wait-free algorithms can be simplified considerably in real-time systems by exploiting the way that processes are scheduled for execution in such systems [1, 2, 20]. In particular, if processes are scheduled by priority, then object calls by high-priority processes automatically appear to be atomic to lower-priority processes executing on the same processor. Consequently they show an implementation of the `Compare-and-Swap` from reads and writes in a priority-based uniprocessor system [20]. In a consequent paper [4], a wait-free implementation of a linked-list from compare-and-swap for priority-based systems is presented. These results combined can offer an efficient implementation, with respect to time complexity, that satisfies the specifications of the wait-free queue classes in RTSJ. The space complexity though of this implementation is $O(N*M)$ where N and M is the maximum number of concurrent tasks that the queue supports and the size of the queue respectively; the time complexity of this implementation is $O(N)$.

In this paper implementations of RTSJ queue classes with $O(M+N)$ space complexity and $O(N)$ time complexity are presented. The wait-free queue classes that are provided by RTSJ have been designed to enable communication between the real-time `NoHeapRealtimeThreads` and the regular Java threads; they have a unidirectional nature with one side of the queue (read or write) for the real-time threads and the other one (write or read, respectively) for the non-real-time ones. The implementations presented in this paper are designed having the unidirectional nature of these queues in mind in order to gain efficiency; to the best of our knowledge our implementations are the first unidirectional wait-free queue implementations in the literature.

The remainder of the paper is organized as follows: In section 2 we give a brief introduction to the basic design features of the RTSJ and Section 3 presents our algorithm and its proof. The paper concludes with Section 4.

2. SYNCHRONIZATION AND RESOURCE SHARING IN RTSJ

In this section a short description of the basic design features of RTSJ, that we had to take into account in our implementation, are presented.

The design of the Real-Time Specification for Java aimed at matching the different nature of the Java language and real-time computing. Compatibility between the Real-Time Specification for Java and the Java Language Specification was required, with the extra requirement that the former had also to be cost-effective for real-time computing systems. The RTSJ are designed for uniprocessor systems.

The minimum scheduling semantics that must be supported in all implementations of the RTSJ are fixed-priority preemptive scheduling with at least 28 unique priority levels. With the term "fixed-priority" RTJEG means that the system does not change the threads' priority, with one exception: The system can change thread priorities only as part of executing the priority inversion avoidance algorithm (which preserves priority inheritance). Threads can change their own or another thread's priorities. The application program must see the minimum 28 priorities as unique; for example, it must know that a thread with a lower priority will never execute if a thread with a higher priority is ready. If threads with the same exact priority are eligible to run, they will execute in FIFO order.

The RTSJ defines a `RealtimeThread` class to create threads, which the respective scheduler executes. The default scheduling policy must manage the execution of instances of regular Java threads and `RealtimeThreads`. The `RealtimeThread` class extends the `java.lang.Thread` class and has more precise scheduling semantics. `RealtimeThreads` can access objects on the heap and therefore can experience delays because of garbage collection. To allow critical-time threads to execute in preference to the garbage collector, a subclass of `RealtimeThread` is provided, called `NoHeapRealtimeThread`. `NHRT` have an implicit execution eligibility logically higher than the garbage collector. `NHRT` cannot access (allocate or even reference) any objects on the heap, which means that they should be able to run while the garbage collector is running. `NHRT` are introduced for tasks with a very low tolerance of non-scheduled delays. `RealtimeThreads`, on the other hand, are more suitable for tasks with a higher tolerance for longer delays. Regular Java threads suffice for tasks with no temporal constraints.

The keyword `synchronized`[2] is the only mechanism required by the specification that can enforce mutual exclusion in the traditional sense in RTSJ as in Java. The specification allows the use of `synchronized` by all three thread types: regular Java thread, `RealtimeThread`, and `NHRT`. In this way, when a `NHRT` attempts to synchronize on an object that is locked by a regular Java thread or a `RealtimeThread` although priority inheritance happens as normal, if the garbage collection is in progress, the boosted `NHRT` is suspended until the garbage collection completes. Given that regular Java threads may never have an execution eligibility higher than the garbage collector, no known priority inversion avoidance algorithm can be correctly implemented. This causes the `NHRT` to incur a delay because of the garbage collection, which is diametrical to the design of the `NHRT`: to be eligible to run while the garbage collector is running. The RTSJ needs to provide mechanisms that allow `NHRT` to communicate (share an object) with `RealtimeThreads` and regular Java threads while avoiding garbage collector induced delays in the `NHRT`. The RTJEG realized that a non-blocking, protected access to objects shared between `NHRT` and regular Java threads is the only solution to the problem. The RTSJ provides wait-free queue classes to provide protected, non-blocking, shared access to objects accessed by both regular Java threads and `NHRT`. These classes are provided explicitly to enable communication between the real-time execution of `NHRT` and regular Java threads.

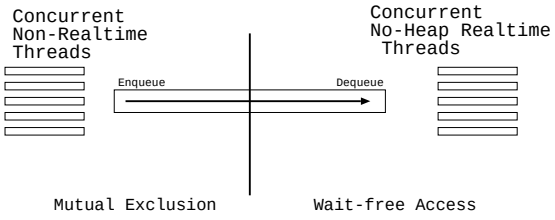


Figure 1: The `WaitFreeReadqueue` class

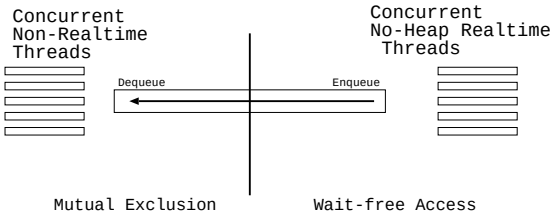


Figure 2: The `WaitFreeWritequeue` class

Basically, there exist two different new queue classes in RTSJ:

1. the `WaitFreeWriteQueue` class
2. and the `WaitFreeReadQueue` class.

[2]If you declare a method to be synchronized, Java will prevent more than one thread from executing that method at any one point in time.

Both these queue classes are unidirectional. The information flow for the `WaitFreeWriteQueue` is from the real-time side to the non-real-time one, as shown in Figure 1. The information flow for the `WaitFreeReadQueue` is from the non-real-time side to the real-time one, as shown in Figure 2. When a `NHRT` wants to send data to a regular Java thread, it uses the `write` (real-time) operation of `WaitFreeWriteQueue` class. Regular threads use the `read` (non-real-time) operation of the same class to read information. The write side is non-blocking, and wait-free, so that `NHRT` will not experience delays from the garbage collection. The `read` operation, on the other hand, is blocking. Since the `write` is wait-free and the arrival dynamics are incompatible, data can be lost. To avoid delays in allocating memory elements, class constructors statically allocate all memory used for queue elements, giving the queue a finite limit. The `WaitFreeReadQueue` class, which is unidirectional from non-real-time to real-time, works in the converse manner.

The third queue class that is described in the RTSJ is the `WaitFreeDeQueue` class and is implemented by putting back-to-back a `WaitFreeWriteQueue` and a `WaitFreeReadQueue`. The formal specification for the `WaitFreeWriteQueue`, `WaitFreeReadQueue` and `WaitFreeDeQueue` classes, can be found in [8].

3. THE ALGORITHM

Algorithmically the implementations of the two wait-free queue classes (`WaitFreeWriteQueue` and `WaitFreeReadQueue`) are similar. In this section, we present the implementation of the `WaitFreeWriteQueue` class to illustrate the idea behind the constructions. The implementation of the `WaitFreeReadQueue` is presented in the full paper [26].

3.1 Informal Description

To simplify the presentation of our algorithm, we start with a simple sequential queue implementation. We will then discuss how to extend this sequential algorithm to a concurrent queue implementation with the specifications that we are looking for. The Java-pseudo-code for this sequential queue is shown in Figure 3.

As it can be seen in Figure 3 we implemented algorithmically the queue using a singly linked list. For efficiency reasons, we choose the front of the queue, where we only delete nodes, to be the head of the list, and the rear of the queue, where we only insert, to be the tail of the list. In this way we only use the operations of the linked list that modify the head and the tail of the list. In order to minimise the interference between the `write (enqueue)` and `read (dequeue)`[3] operations, we introduce a *dumbcell* in the empty list. In this way, when executing a `dequeue` operation, only the *head* needs to be checked in order to see whether the queue is empty or not. If the *next* field of the *head* is *null*, the queue is empty. Therefore, the `dequeue` operation needs only to check the *head* variable and only the *tail* needs to be checked for the

[3]Throughout the paper, the terms queue `write` and `enqueue` are used interchangeably. The same also holds for the terms queue `read` and `dequeue`. To distinguish between the queue `read/write` and normal `read/write` memory operation, we are using typewriter type style for the queue operations and serif type style for the memory ones.

```
public class SeqQueue {
2   RTQueueCell head, tail;
    RTQueueCell dumbcell;
4   void SeqQueue()
    {
6     head = dumbcell;
      tail = dumbcell;
8     dumbcell.data = null;
      dumbcell.next = null;
10  }
    public boolean write(java.lang.Object object)
12  {
      RTQueueCell temp;
14    temp = new RTQueueCell();
      temp.data = object;
16    temp.next = null;
      //tail and tail,next will be shared
18    //read/write in concurrent implementation
      tail.next = temp;
20    tail = temp;
      return TRUE;
22  }
    public java.lang.Object read()
24  {
      RTQueueCell temp;
26    temp = (RTQueueCell)head.next;
      if (temp != null)
28      head = temp;
      return temp;
30  }
}
```

Figure 3: The Sequential implementation of the queue

`enqueue` operation. For the initialization for the simple sequential queue we define a *dumbcell*, with `null` in its next field, and let the *head* and the *tail* of the queue point to it, statements 6 to 9 in Figure 3.

Figure 4 shows the structure of the cells of the linked list that we are using. The class `RTQueueCell` has two public members: one is for the data entry, the other is the next pointer that singly links the elements of the list.

```
public class RTQueueCell {
        public java.lang.Object data = null;
        public java.lang.Object next = null;
        }
```

Figure 4: Definition of the queue cell

To extend the sequential version to a concurrent wait-free enqueue implementation, first we will use a simple announce-and-help scheme for the `enqueue` operations. The announce-and-help scheme uses the priority based scheduler to achieve wait-freedom. This scheme is based on the task priorities to guarantee that an operation will be finished in a bounded number of steps regardless of the status of the other operations, as follows: First, each enqueue-task announces the data (writes a pointer to the memory where the data are) that it wants to enqueue in a special *Announcement* array. The enqueue-task with priority i will use the ith position of the array. After the announcement step, the enqueue-task reads and helps the data that have been announced on the array one by one, starting from the lowest priority up to its own priority. If during this helping phase an enqueue-task A is not going to be preempted by a higher priority task, then all current enqueue operations, with smaller priority than the priority of A, that are announced will be

helped/enqueued by A. If during the helping phase the enqueue task A is preempted by a higher priority task B, then there are two cases:

1. B is not an enqueue-task on the same queue: then the task A will continue its program steps after B finishes, from the same queue-state from which it was pre-empted. Dequeue operations on the same queue are executed by tasks that have lower priority and therefore they can not preempt enqueue operations on the same queue.
2. B performs an enqueue operation on the same queue: in this case B is going to help all lower priority tasks that are announced and then help its own task that has also been announced. Therefore task A will be helped by B. Because the priorities are bounded, there always exists a task which will not be preempted by another queue task. Therefore, all operations that announced their operations will be helped (either by themselves or by higher priority tasks).

The RTSJ have been designed for uniprocessor systems and RTSJ as well as Java support only plain memory synchronization primitives like atomic reads and writes to memory locations as opposed to other advanced synchronization primitives like `Compare-and-Swap`. The weak vocabulary of Java in memory synchronization comes again from the fact that different processors support different memory synchronization primitives and Java was designed to be hardware independent. Although reads and writes are very weak synchronization primitives in the context of general asynchronous systems, by exploiting the fact that the tasks are executed by priority, it has been shown that they are universal primitives for priority-based uniprocessor systems [20]. In RTSJ the application program must see the minimum 28 priorities as unique; for example, it must know that a thread with a lower priority will never execute if a thread with a higher priority is ready. If threads with the same exact priority are eligible to run, they will execute in FIFO order.

During the design phase of any shared data objects, a problem that arises from the use of memory read/write operations is the "enabled late-write" problem [19]. The "enabled late-write" problem arises when a low priority task A is preempted while it is about to write to a memory position, and is preempted by other tasks that access and modify the same memory position. When task A resumes running, it overwrites the previous "fresh" value with an "old" one. Anderson et al. [20] proposed a majority voting scheme to overcome the problem. Their scheme requires $2N-1$ memory words to solve "the enabled late-write" problem for 1 word.

In this paper we propose a new more efficient scheme to face the "enabled late-write". The idea is simple, to try to avoid the problem by:

1. Making sure that, when a task A is preempted before writing to position p, all other tasks that write on to p, (while A is preempted) write the same value that A wanted to write. In order to establish this, we guide the tasks to go through the same computational steps as A when they have to decide about the value that they want to write on the same memory location.
2. When the above is possible, we organise the shared variables that might suffer from the "enabled late-write" problem as arrays that carry information that can be used to determine the correct/new value of the variable algorithmically.

We believe that, the same idea can be used when algorithmically designing other shared objects for the RTSJ.

```
public class WaitFreeWriteQueue {
... ...
private MemoryArea MemPool;
private java.lang.Object[] Announcement;
private RTQueueCell[] tail;
private RTQueueCell head;
// get the minPriority from the scheduler
private int minPriority;
// get the maxPriority from the scheduler
private int maxPriority;
        ... ...
```

Figure 5: Shared private variables for `WaitFreeWriteQueue`

The wait-free part in this class is the part that implements the wait-free `enqueue` operations. The wait-free `write` operations share also the private variable *MemPool* that hold references to a `MemoryArea`[4]. The shared private variables for our `WaitFreeWriteQueue` are as shown in Figure 5. All `RTQueueCells` should be allocated from the `MemoryArea`. The *Announcement* array is used to hold the the different `enqueue` operations. The *tail* and *Announcement* arrays are of equal length, equal to the real-time priority level supported by the scheduler. For the head of the queue we use the simple variable *head*. The *minPriority* and *maxPriority* are the minimum and maximum priorities that real-time threads can be assigned, respectively. This information can be obtained from the scheduler. All shared variables will be initialised when constructing the queue. The initialization is similar to the initialization of the sequential version. Because now we use an array to represent the tail, we need to initialize this array in away that it is easy for the algorithm to find out the correct `tail` (the *dumbcell*), when a task accesses the queue for the first time. But, when a task accesses the *tail* array, it checks from the the cell of the lowest priority task to the highest to find a non-null cell. Henceforth, we initialize the cell corresponding to the lowest priority point to the *dumbcell*. During the initialization part, we also need to initialize the *Announcement* array with the value `null`, that means that there are no announced operations. The pseudo-code for the initialization is described below in Figure 6. The initialization of the local variables is part of the pseudo-code description of the algorithm described in Figure 8.

Now, in order to extend algorithmically the sequential version that we presented at the beginning of this section to the concurrent one that we are aiming for, we first need to make sure that the shared read/write operations to the *tail* and the *tail.next* variables (the shared variables of our implementation where overwriting might take place) do not suffer from the "enabled late-write" problem. The wait-free `enqueue` operation is presented in the `write` function

[4]The RTSJ introduces the `memory area` concept, which is a region of memory outside the garbage-collected heap that you can use to allocate objects. The RTSJ uses the abstract class `MemoryArea` for this.

```
RTQueueCell dumbcell = new RTQueueCell();
... ...

Announcement = new
java.lang.Object[maxPriority + 1];
tail = new RTQueueCell[maxPriority + 1];

for (i=minPriority;i<=maxPriority;i++)
{
Announcement[i] = null;
tail[i] = null;
}
tail[minPriority] = dumbcell;
head = dumbcell;
dumbcell.data = null;
dumbcell.next = null;
```

Figure 6: Initialization for `WaitFreeWriteQueue`

below. The announce-and-help scheme, that is used in our implementation, uses the priority based scheduler to achieve wait-freedom. Each priority is mapped to the respective entry of the array *Announcement*. An `enqueue` operation first gets the priority of its thread, then it allocates a free cell from the memory area assigned to the queue. The memory area is where the queue and its internal elements are allocated. After writing the data in the free cell, the task announces this cell in the *Announcement* array at the index that is associated to its priority. This constitutes the last part of the announcement phase. This is, as we are going to see, the "linearizability point" of the enqueue operation at the linearizability history. After it announces in the *Announcement* the object that it wants to enqueue, a task will enter the helping phase that was described at the beginning of this subsection. The helping phase is described in relation to the implementation pseudo-code next. During the helping phase, an `enqueue` operation with priority i helps the tasks with priority $j \leq i$ that have been announced in the array *Announcement*, one at a time, starting from the operation with the smallest priority that it can find (statement 28 in the implementation). For each such operation, it finds the tail of the queue (statements 35-47 on the protocol); then puts the data announced at the end of the *tail* of the queue; then changes the *tail* variable to point to the new position; and finally cleans the *Announcement*$[j]$.

3.2 Other Methods Supported by the Class

Here we describe the implementation of the other methods supported by the `WaitFreeWriteQueue` class. These implementations are described in Figure 7.

The functions `isFull` and `size` need information from the `MemoryArea` and the size of the RTQueueCell. In the Java specification and the RTSJ, there exist no function `sizeof`. One way to overcome this is to measure the size of the RTQueueCell off-line. The best way though, would have been if the RTJEG had added this function in the RTSJ. In the use of non-realtime Java, the virtual machine will handle all memory accesses and the programmers have no need to find the size of some primitives or objects in order to allocate a space of memory for them. On the other hand for the NHRT, programmers need to manage the memory themselves, such a function is necessary and will be appreciate. The functions `clear` is described in Figure 7. The `force` function that we have implemented does not follow

```
public void clear()
{
int i;
for (i=minPriority;i<=maxPriority;i++)
{
Announcement[i] = null;
tail[i] = null;
}
tail[minPriority] = DumbNull;
head = DumbNull;
DumbNull.next = null;
}
public boolean isEmpty()
{
if (head.next == null)
return true;
else
return false;
}
public boolean force()
{
return false;
}
public boolean isFull()
{
if (MemPool.memoryRemaining() <
sizeof(RTQueueCell));
return true;
else
return false;
}
public int size()
{
int i;
i = MemPool.memoryRemaining()/
  sizeof(RTQueueCell);
return (i);
}
```

Figure 7: Other functions for the `WaitFreeWriteQueue`

the specifications of the RTSJ. As we implement the queue as a link-list, a cell of the link-list will not be freed when it is removed from the link-list. The cell will be freed only after the reader consumes it. Therefore, the situation described in the specification for the `force` function to return `false` can never happen in any non-trivial extension of our implementation. On the other hand, we would like to mention that the specification n the RTSJ of this method are not clear in the presence of concurrency. When several concurrent NHRTs invoke the `force`, only one value will remain/forced in . There is no correct definition for this case in the RTSJ. We decided not to support `force`, our `force` function always returns `false`.

3.3 Correctness Proof

In the helping phase two sets of variables are used, the *tail* array (tasks help to enqueue data at the tail of the queue) and the *Announcement* array; all of them are shared variables and can be read and written by different tasks. In the implementation, the value of a variable *Announcement*$[i]$ changes from *null* to a non-null value, when a task with priority i announces its `enqueue` operation. The value of the same variable changes back to *null* when the item that the `enqueue` operation wanted to enqueue was enqueued by the same operation or by another higher priority enqueue operation. If there are e enabled writes that are ready to write to *Announcement*$[i]$ then at least $e-1$ of them are helping operations and have priority higher than i and want to change the value of the *Announcement*$[i]$ from *non-null* to *null*. The one "enabled late-write", that might exist, is the

```
1 public boolean write(java.lang.Object object)
2 {
3 boolean find = false;
4 int i,j, mypriority;
5 RTQueueCell tempcell,
6       temptail=null;
7 java.lang.Object tempAnnounce;
8 java.lang.Thread currentone;
9 //Find your priority
10 currentone =
11 java.lang.Thread.currentThread();
12 mypriority =
13 currentone.getPriority();
14 //Allocate a cell in the MemoryArea
15 try
16 {
17 tempcell =
18 MemPool.newInstance(RTQueueCell);
19 tempcell.data = object;
20 tempcell.next = null;
21 } catch(OutofMemoryError x) {
22 return false;
23 }
24 //Announce your operation
25 Announcement[mypriority]=tempcell;
26 //Enter helping phase and help
27 //lower priorities and yourself
28 for(i=minPriority;i<=mypriority;i++)
29 {
30 tempAnnounce=Announcement[i];
31 if (tempAnnounce == null)
32 continue;
33 //Try to find the actual tail
34 find = false;
35 for(j=minPriority;j<=maxPriority;j++)
36 {
37 if (tail[j]!=null)
38  if (tail[j].next == null)
39  {
40    find = true;
41    break;
42  }
43 }
44 if (find)
45 //No preemption detected, the
46 //actual tail has been found.
47 temptail =  tail[j];
48 else
49 //Preemption detected!
50 //there are 2 possibilities
51 if(Announcement[i]!=null)
52 {
53  //Preempted, helped but not completely.
54  //Help the task with priority $i$ to update
55  //tail and Announcement array
56  tail[i]=(RTQueueCell)tempAnnounce;
57  Announcement[i] = null;
58  continue;
59 }
60 else
61 //Preempted and helped by a higher
62 //priority task that helped the task that
63 //you were helping and you. Return!
64  return true;
65
66 //Did you preempt a lower priority task
67 //when it was on statements 80 and 81 ?
68 if(temptail==tempAnnounce)
69 {
70 //Help it to update the Announcement
71 Announcement[i] = null;
72 continue;
73 }
74 //Preempted by higher priority task
75 //that helped you completely?
76 if(Announcement[i]==null)
77 return true;
78 //Enqueue the announcement
79 temptail.next=tempAnnounce;
80 tail[i]=(RTQueueCell)tempAnnounce;
81 Announcement[i] = null;
82 }
83 return true;
84 }
```

Figure 8: Wait-free enqueue operation for the `WaitFreeWriteQueue`

write with priority i that wants to announce a new enqueue operation. This write will not be scheduled before the other pending writes, with higher priority, take place, and thus, its write will not be overwritten by them. The above proves the following lemma:

LEMMA 1. *$\forall i$, $minPriority \leq i \leq maxPriority$, $Announcement[i]$ will not suffer from the "enabled late-write" problem.*

LEMMA 2. *When a task A is preempted just before it writes to the tail array, a higher priority task will write the same content to the same position in the tail array.*

PROOF. The decision of what to write on the tail is based on the contents of the *Announcement* and *tail* arrays. If a higher priority task preempted task A just before A was to write the *tail* array, then, since, nothing changed on the *Announcement* and *tail* of the object from the time that A read them, the higher priority task, that preempted A, will compute the same value to write to the *tail* array. □

The "enabled late-write" problem could have happened in the *tail* variable, if we would have used a simple *tail* variable for the queue, as it is used in the sequential implementation of the queue. To solve the problem, we organise the *tail* of the queue as an array. Each location in the array corresponds to the respective priority. All tasks with the same priority will be executed in a FIFO order and will use the same location in the array. Each enqueued item from a task with priority i will become the tail of the queue once, and the ith index of the *tail* will point to it. In our construction, all tasks that try to help a task with priority i that has been announced, are going to write to the *tail* array at the index that corresponds to priority i. If a task A, which is helping task C, is preempted, with an enabled write on the *tail* array, by another high priority task B, the new task B will help the same task C and will go through the same computational steps and at the end it will update the same entry of the *tail* array with the same value as the preempted enabled write of task A. This is guaranteed from Lemma 2. In this way, the "enabled late-write" problem can not take place in any $tail[i]$ variable. This sketches a proof of the following lemma.

LEMMA 3. *$\forall i, minPriority \leq i \leq maxPriority$, $tail[i]$ will not suffer from the "enabled late-write" problem.*

Now, we need to give a way to the tasks to read the *tail* array and compute the real tail of the queue. Each item in the *tail* array has been the real tail of the queue at some point in time but only one of them can be the current tail of the queue. In our implementation, there is at most one *tail* entry that has the value *null* on its next field. As we are

designing a concurrent queue, an enqueue operation can be preempted anywhere; a task A can be preempted between statement 79 and 80 by a task B. The *tail* array then will have no element with the value *null* on its next field. The actual tail in this case should point to the object enqueued by task C, which is being helped by task A (A executes statement 79 and 80 i only when it is helping another task). When task B tries to find the tail of the queue in order to add in the queue the task that it is helping, it uses the local *temptail* variable to store it. In the pseudo-code, when task B executes statements 35-43, it goes through the *tail* array from the lowest priority to the highest priority and tries to find the one index in the array with *null* in the next field, if it is there. If there is no overlapping with other enqueue operations, task B will find the index with *null* in the next field. As a next step, it will store the value in *temptail* (statement 47).

LEMMA 4. *temptail.next will not suffer from the "enabled late-write" problem.*

PROOF. When a task A with priority j helps a task with priority i that has been announced, where $i < j$, all items in the announcement array from *minPriority* to $i-1$ should have the value *null* because task A starts its helping phase from *minPriority*, and, tasks with priority less than j can not preempt task A and make changes in the announcement array. Before task A updates the next field of the tail of the queue (statement 79), nothing changes in the tail array and the announcement array. If a task B with priority k, where $k > j$, preempted task A, task B will add its own announcement in position k and nothing between *minPriority* to j in the announcement array will change, so task B will help the task with priority i announced and it will find the same tail and make the same decision with task A and finally put the task with priority i that was announced on the next field of the tail. □

If task B overlapped with other `enqueue` tasks, then task B might not find an index on the array with *null* in the next field. If the later happens, the task B has already enough information as we will see below to find the actual tail of the queue and help the preempted task to finish updating the tail of the queue. To see this, let us look at the possible ways that the above could have happened; there are two cases:

- Task B preempts the lower priority task A, when A was between statements 79 and 80; e.g. A had just finished enqueueing the data before updating the tail of the queue. The actual tail of the queue at this point is the task which is being helped by both task A and task B. Task B will help task C to update the tail when B runs statements 56-57.
- Task B is preempted by a higher priority task D and D updates the *tail* array in such a way that task B misses the actual tail of the queue when B is scheduled back. In this case, D will help all lower priority tasks. So, task B just needs to stop its helping phase and return. B will detect that it has been helped and return in statement 64.

The above sketches a proof that items are going to be put on the singly link-list one after the other.

```
1 public synchronized java.lang.Object read() {
2 RTQueueCell tempcell;
3 tempcell = head.next;
4 if (tempcell != null)
5 head = (RTQueueCell) tempcell;
6 return tempcell;
7 }
```

Figure 9: Lock-based dequeue operation for `WaitFreeWriteQueue`

Since different tasks are going to try to help the same task, we need to show that an item is not going to be enqueued more than one time. That is the reason that statement 68 is used from task B to detect that it has preempted a task A when A was between statements 80 and 81 of its pseudo-code. When the preemption happens, the announcement has not been cleaned but the announcement has been added to the queue as a tail, which has been read by task B in *temptail*. If such a preemption is detected, the task B will help task A to clean the announcement array, when task B executes statement 71. As both of them want to write *null* at the same position, no "enabled late-write" problem exist. Statement 76-77 is used from task B to detect that it had been preempted by a higher priority tasks D and to conclude that task D has helped the task that B was helping when preempted.

The following lemma also proves that it is necessary and sufficient for a task to help other tasks with priority up to its own priority.

LEMMA 5. *When a task A with priority i announces an enqueue data in the Announcement array, all elements of the array from $i+1$ to maxPriority have the value null.*

PROOF. Assume towards a contradiction that *Announcement*$[j]$ is not *null*, where $j > i$. Then there must exist a task B with priority j thus announced its enqueue object in *Announcement* array and the announcement by task B hasn't been "cleaned". The *Announcement*$[j]$ is cleaned as the last step of the enqueue operation. The task A must preempt task B to announce its enqueue object in *Announcement*, in order to preempt task B, $i > j$ must be hold, contradiction,

As the contents of the *Announcement* array from index $i+1$ to index *maxPriority* are *null* when task A announce its operation, there is no need to help them. It is sufficient to help tasks with priority up to i. As task A can preempt any lower priority task after announcing, it is necessary to help them. □

Figure 9 shows the lock-based `read` operation of the `WaitFreeWriteQueue` class. It's a straight forward implementation that uses mutual exclusion to serialise concurrent dequeue operations.

The access of the queue is modelled by a history h. A history h is a finite (or not) sequence of operation invocation and response events. Any response event is preceded by the corresponding invocation event. For our case there are two different operations that can be invoked, a `write` operation or a `read` operation. An operation is called complete if there is a response event in the same history h; otherwise, it is said

to be pending. A history is called complete if all its operations are complete. In a global time model each operation q "occupies" a time interval $[s_q, f_q]$ on one linear time axis ($s_q < f_q$); we can think of s_q and f_q as the starting and finishing time instants of q. During this time interval the operation is said to be *pending*. There exists a precedence relation on operations in history denoted by $<_h$, which is a strict partial order: $q_1 <_h q_2$ means that q_1 ends before q_2 starts; Operations incomparable under $<_h$ are called *overlapping*. A complete history h is linearisable if the partial order $<_h$ on its operations can be extended to a total order $\rightarrow_h$ that respects the specification of the object [14].

In the rest of this section, we prove that our implementation is a concurrent linearizable queue implementation. In order to do so, we will show that any possible history ($<_h$), produced by our implementation, can be extended to a total order ($\rightarrow_h$) by using a "linearization point" for each operation. The "linearization point" of an operation is an atomic point on its execution, during which the operation takes effect.

LEMMA 6. *The* write *to the announcement array is the "linearization point" for the* `write` *operations.*

PROOF. By Lemma 5, when a task A with priority i executes statement 25, all items of the announcement array from $i+1$ to $maxPriority$ have the value $null$. Task A will help all operations announced in *Announcement* from the lowest to its own priority. Enqueue operations with lower priority than i that have been announced by executing statement 25, will be enqueued before A's announcement on the *anouncement* array. If the current task A is preempted by a higher priority task after executing statement 25, the announcement will be enqueued before the announcement of the task with higher priority. So, the execution order of statement 25 in the `write` operation extends the precedence partial order to a total order that respects the FIFO specifications of the `WaitFreeWriteQueue` class. □

LEMMA 7. *The* read *of the next field of the head of the queue is the "linearization point" for the* `read`*s of the queue.*

PROOF. Since, mutual exclusion is used between `read` operations on the queue, the order in which they get access to the critical section totally orders them. But as the `read` operations of the queue have lower priority than all the `write` operations of the queue, they can be preempted and run concurrently with `write` operations. As all high priority tasks will appear atomic to a low priority task, a `write` operation will only be observed if it starts executing before statement 3 of the `read` operation. By selecting then the execution of the statement 3 of a `read` operation as its "linearizability point", all operations are totally ordered with a relation that extends the precedence relation and respects the specification of the `WaitFreeWriteQueue` class. □

The lemma below proves that our queue implementation is a FIFO one and no element enqueued gets lost. For simplicity we introduce in the history `write`($empty$) operations when a queue is empty.

LEMMA 8. *In a complete history such that* `write`$(x) \rightarrow_h$ `write`(y), *then* `read`$(x) \rightarrow_h$ `read`(y).

PROOF. From the assumption, we have that `write`$(x) \rightarrow_h$ `write`(y) which means x is announced before y. If there is no overlapping, x will be put in the list before y as in the sequential version. If overlapping exists, by lemma 5, the task A who announces y has higher priority than the task who announces x. As task A will help from the task with lowest priority to itself, it will put x in the list before y. As `read` uses mutual exclusion, only one `read` operation processes the list from the head to the tail. So `read` operations will find x first. □

The lemma below proves that dequeue operations dequeue items that have really been enqueued.

LEMMA 9. *In a complete history, if x is* `read`*, then it has been* `write`*d, and* `write`$(x) \rightarrow_h$ `read`(x)

PROOF. The linearizability point of the `read`(x) is the point where the `read` operation reads the *next* field of the *head*. Because a `write` operation announces its operation and the announcement takes place before the helping phase, and in the helping phase the announcement will be put in the next field of a $tail[i]$. If x is read, then some task must write in the field during its helping phase. Helping an announcement can only happen after it has been announced by some task in the announcement array. So, the x read by a task must be written by a task before the `read` operation. □

From the lemmas of this section, we have that our implementation is a linearizable FIFO queue implementation:

THEOREM 1. *Our algorithm for the* `WaitFreeWriteQueue` *is a linearizable FIFO concurrent queue without the "enabled late-write" problem.*

4. CONCLUSION

Efficient implementations of the RTSJ queue classes are presented in this paper. The implementations are designed to have the unidirectional nature of these queues in mind and they are more efficient, with respect to space, compared to previous general "bi-directional" wait-free implementations without losing in time complexity.

There are several ways that future research in wait-free synchronization can contribute to real-time Java. A very promising, we believe, is the investigation of practical wait-free implementations for garbage collection in the RTSJ model. The garbage collector is a central component of the Java environment, its wait-free implementation will ease the programmers effort to correctly reason about the temporal behaviour of their Java programs.

5. REFERENCES

[1] J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 346–355, Dec 1998.

[2] J. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 111–122. IEEE, Dec. 1997.

[3] J. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96*, pages 94–105. IEEE, Dec. 1996.

[4] J. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects on priority-based systems. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC '97)*, pages 229–238. ACM, Aug. 1997.

[5] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(2):134–165, May 1997.

[6] J. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay. *Lock-Free Transactions for Real-Time Systems, Real-Time Database Systems: Issues and Applications*. Kluwer Academic Publishers, Amsterdam, 1997.

[7] G. Bollella and J. Gosling. The real-time specification for java. *IEEE Computer*, 33(6):47–54, June 2000.

[8] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000. URL: `www.javaseries.com/rtj.pdf`.

[9] G. C. Buttazzo. HARTIK: A real-time kernel for robotics applications. In *Proceedings of the Real-Time Systems Symposium*, pages 201–205. IEEE Computer Society Press, Dec. 1993.

[10] J. Chen and A. Burns. Asynchronous data sharing in multiprocessor real-time systems using process consensus. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 2–9, 1998.

[11] A. Ermedahl, H. Hansson, M. Papatriantafilou, and P. Tsigas. Wait-free snapshots in real-time systems: Algorithms and their performance. In *Proceedings of the fifth International Conference on Real-Time Computing Systems and Applications (RTCSA '98)*, pages 257–266, dec 1998.

[12] J. Gosling and H. McGilton. *The Java Language Environment: A White Paper*. SUN Microsystems, Inc., 1995.

[13] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, 1999.

[14] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[15] H. Kopetz and J. Reisinge. The non-blocking write protocol NBW: A solution to a real-time synchronisation problem. In *Proceedings of the Real-Time Systems Symposium*, pages 131–137. IEEE Computer Society Press, Dec. 1993.

[16] G. Lamastra, G. Lipari, G. C. Buttazzo, A. Casile, and F. Conticelli. Hartik 3.0: A portable system for developing real-time applications. In *Proceedings of the 4th IEEE International Workshop on Real-Time Computing Systems and Applications (RTCSA)*, pages 43–50, October 1997.

[17] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, Apr. 1983.

[18] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *10th International Conference on Distributed Computing Systems*, pages 116–123. IEEE, IEEE Computer Society Press, May–June 1990.

[19] S. Ramamurthy. *A Lock-Free Approach to Object Sharing in Real-Time Systems*. PhD thesis, University of North Carolina at Chapel Hill, 1997.

[20] S. Ramamurthy, M. Moir, and J. H. Anderson. Real-time object sharing with minimal system support. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 233–242. ACM, May 1996.

[21] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.

[22] H. Sundell and P. Tsigas. Space efficient wait-free buffer sharing in multiprocessor real-time systems base on timing information. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA '00)*, pages 433–440, 2000.

[23] H. Sundell, P. Tsigas, and Y. Zhang. Simple and fast wait-free snapshots for real-time systems. In *Proceedings of the 4th International Conference On Principles Of DIstributed Systems (OPODIS 2000)*, pages 91–106, December 2000.

[24] P. Tsigas and Y. Zhang. Non-blocking data sharing in multiprocessor real-time systems. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications(RTCSA '99)*, pages 247–254, dec 1999.

[25] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '01)*, pages 134–143. ACM, July 2001.

[26] P. Tsigas and Y. Zhang. Efficient and simple implementations of the wait-free queue classes of the real-time specification for java. Technical Report TR2002-01, Department of Computing Science, Chalmers University of Technology, 2002. http://www.cs.chalmers.se/~tsigas/papers/tr2002-01.pdf

Towards a Framework for Automated Testing of Transaction-Based Real-Time Systems*

Robert Nilsson Sten F. Andler Jonas Mellin

Department of Compter Science
University of Skövde
Box 408, 541 28 Skövde, Sweden
phone: +46(0)500-448370, fax: +46(0)500-448399
email: {robert,sten,jonas}@ida.his.se

Abstract

In real-time systems, 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 of real-time applications 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. We propose a framework for testing of transaction-based real-time systems using automatic test-case generation and execution techniques. An overview of previously proposed methods for generating test cases for testing of temporal properties of real-time systems is also presented.

Keywords: Testing, Event-Triggered, Real-time systems, Test-case generation, Timeliness

1 Introduction

Modern real-time systems tend to be increasingly complex. Moreover, real-time systems generally must be dependable as they often operate in safety or business critical environments. These characteristics imply that there is a need for rigorous verification methods to detect errors that arise from temporal faults [1]. 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 ad-hoc. *Testing* is a method to dynamically verify software by execution in order to detect errors and failures. *Errors* are the internal states in application software which may lead to externally observable *failures* [1]. *Test-case generation* is the process of selectively generating test cases that exercise system behaviors likely to reveal such errors. Test-case generation is based on application knowledge such as specification models, code structure, and system design. This information is used to selectively generate sets of test cases, sometimes referred to as *test suites*, that have high probability of revealing specific classes of errors or deviations from the expected behavior [1]. An *execution environment*, in this context, is the architecture of the system in which the tested applications run. This includes real-time operating system services, communication primitives, scheduling algorithms and properties of the underlying hardware. We suggest that properties of the execution environment be considered in the test-case generation process. 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, executed and observed.

This paper focuses on generation and execution of test cases for event-triggered real-time systems. Real-time systems are often concurrent; this complicates generation and selection of suitable test cases as system behavior is dependent on the non-deterministic order in which tasks execute, e.g. due to varying execution times. The event-triggered paradigm of real-time systems design greatly complicates testing since the behavior of the controlled environment continuously influence the execution order trough sporadic requests [2]. We demonstrate how existing test-case generation methods take these factors into consideration for testing of *timeliness*, which is the ability of the system to meet its time constraints. Further, we present work in progress on a testing framework for

*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.

This article was published at RTCSA 2002, part of the material was also presented at last ARTES graduate student conference. New contributions regarding test-case specifications and execution is found in section 3

testing of event-triggered real-time applications implemented as transaction processing systems.

2 Target system model

The classical approach for achieving temporal correctness in real-time systems is off-line scheduling analysis with full a priori knowledge of resource requirements and time constraints. Scheduling analysis is typically performed in systems with static, or fixed priority, scheduling schemes to guarantee that deadlines are met for worst-case behaviors. Because of variations in release times and varying execution times in such system, there is still a need for rigorous testing (see for example Thane [3]). This kind of systems will be referred to as *time triggered* [4] in this article. The penalty of having time-triggered systems is that utilization, flexibility and extendibility suffer. For example extendibility is degraded in the sense that when some system parameters change (for example the execution time of some task) the whole system needs to be reanalyzed, re-scheduled and re-tested.

For applications where these penalties cannot be accepted the alternative is to have systems that are dynamically scheduled (using EDF or similar deadline scheduling approaches [5]) and whose behavior is determined by the current state of the system and the incoming requests. These systems are referred to as *event triggered* [4]. Event-triggered systems do not suffer from the penalties described above; instead, they impose problems to perform a complete off-line scheduling analysis, and hence, testing must be used to build confidence in their temporal correctness. Unfortunately, testing of event-triggered systems has been shown to be much harder than testing of their time-triggered counterparts [2]. A reason for the difference is that static schedules repeat themselves) whereas dynamic scheduling schemes often do not. A related reason is that tasks in time-triggered systems are released at well-defined points in time, whereas tasks in event-triggered systems are triggered by the sporadic arrival of events. Typical for an event-triggered application is some core functionality with associated hard time constraints and tasks that are executed to increase quality of service. For example, consider an air-traffic control system or a monitoring system of a chemical plant.

3 Testing framework for event-triggered real-time systems

The approach presented here aims at testing event-triggered systems. For increased testability we propose a framework where event-triggered applications are implemented on top of a transaction processing system, which provide resource locking, ECA-rules and event filtering, e.g., a real-time database. There are several benefits of using such a framework apart from the testability issue, e.g., synchronization and data management are supported. The testability in this framework is increased by the properties associated with transactions, e.g. isolation and atomicity properties imply that concurrently executing transactions will interact in well-defined ways. When testing an event-triggered application we would like to concentrate our test efforts on guaranteeing that the critical transactions meet their deadlines. A problem when testing real-time systems are the repeatability issues related to the non-determinism caused by varying execution orders and execution times of transactions. This leads to problems in stating that a particular behavior has been sufficiently tested. Most test-case generation methods for testing real-time systems aim at generating *event-sequences* i.e. sequences of events with associated relative time-stamps. However, the system-wide state in which these inputs occur is seldom considered. This implies that the same event-sequences have to be run multiple times to gain confidence on the system behavior (hoping that some particular execution order will occur). Consider the simple example when two transactions are executed on the same processor and in 70% of all cases the transactions are scheduled in one particular order. To detect an error when the reversed order occurs - the test case must be executed at least 13 times to have 99% probability of revealing the error. The alternative is to have some means of controlling the execution environment so that a particular execution order occurs. However, when performing timeliness testing (and time-dependent testing) probes cannot be inserted in the execution environment during test execution and then removed, as they will change the temporal behavior of the system. An approach to increase the value of test-case executions is to include information about the system-wide state in each test case, and hence, to achieve a more deterministic test execution. By using such test cases, the problem of finding timeliness violations is transformed into the problem of selecting (and executing) the right test cases. Adding system state-information to test cases has the consequence of dramatically increasing the total number of test cases possible to run in the system. This can be compensated for by adding constraints on the system behavior as described by Mellin et al. [6] [7]. In particular, the inclusion of designated preemption points reduce the number of possible behaviors of the system significantly and simplifies test execution. Test-case gen-

eration and selection can be automated, given structured or formal specification of the system and its execution environment. Test-case execution can also be automated, given a uniform test-framework and well-specified test cases. In the framework presented here, a non-intrusive event monitoring facility is provided by the underlying transaction processing system, and since the monitor will remain in the system during normal operation the temporal behavior is preserved. Further, all instrumentation to setup a test case is done prior to the actual test execution.

Test-case specification

Test cases are defined to contain an initial state description, an event sequence and an expected outcome. In a concurrent real-time system, the state is defined by the state of all currently active tasks (including their point in execution and used data objects), and hence, a sequence of input events is not sufficient to deterministically bring the system into a specified state. Here, we aim to bring the system to a specified initial state without explicitly defining a sequence of events that may lead to it. A test case for a real-time system should at least include an event sequence with associated time-stamps (or a script which implement such sequence). In simple control applications this may be enough, but in a typical event-triggered application, the events arrives with parameters that are fed to the processing task. The content of these parameters influence the execution behavior of the triggered task and need to be included in the test case. In this work, we assume that event-parameters are passed with the events and argue that their values should be part of the generated event-sequences (this is not assumed in the related work section). The expected outcome of a test case is a definition of the correct system behavior acquired by applying an event sequence on a system executing from the specified initial state. In a real-time system a time constraint is associated with each outcome. In this work, the expected results are limited to the time constraints, parameters of output events and changes to data objects in the underlying database.

Test-case execution scheme

The first step in executing a test case is to force the system into the specified initial state. Under the strict pessimistic concurrency control policy, the transactions that are active have also acquired locks on all required resources. Hence, serial execution up to the required transaction states are possible while no time dependencies are reflected in the data objects (in Figure 1; transactions T1, T2, and T3 are executed sequentially to specified preemption points). Obviously, the state of the resources will influence the execution time of the individual transactions; hence, classes of relevant initial states of resources should be selected. Once each active task is executed to the specified preemption point, they are combined to a concurrent initial state. Pending transaction tasks that have not been allowed to execute yet, but is part of the initial state specification, are added to the initial state. The dynamic scheduler is assumed to update its schedule as a direct response of incoming events and hence, one must allow it to build an initial schedule before the test execution begins. At this point the specified initial state has been enforced and can be saved so that multiple test cases with different event-parameters can be tested from this state (increasing performance of the testing). One possibility of saving the state is to have some means of replication of the database-node. In the second phase the system is executed from the initial state and inputs are externally injected during the execution according to the event-sequence part of the test-case specification. The behavior of the system is continuously monitored and logged using the event monitoring facilities present in the database architecture. The event logs are used after test case execution to verify that the correct behavior was observed.

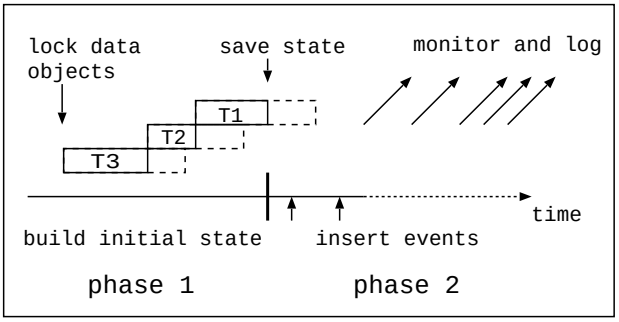


Figure 1: Test execution phases

4 Related work

Clarke and Lee [8] introduce a framework for testing time constraints of real-time systems. Time constraints are specified in a constraint graph, and the specification model is process algebra. This method is interesting for 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 time constraints on the input to the system. No method for generation of test cases for testing of time constraints on the output sig-

nals is presented.

There are a number of methods for testing time constraints based on timed automata models. In general, these methods supply interesting event sequences for testing time constraints. However, they do not consider the variations in behavior imposed by the execution environment. One such method is presented by Petitjean and Fochal [9] who propose a test architecture for testing of timed systems. A timed system is described as a timed automaton where time constraints can be expressed as a region graph. The specification model is the timed- I/O automaton theory proposed by Alur and Dill [10]. Another approach was presented by En-Nouaary et al [11]. Their approach exploits a sampling algorithm using grid-automata and non-deterministic finite-state machines as intermediate representations in the test generation process to reduce the test-effort and maintain test-coverage. However, the complete-testing assumption used in their work does not hold for the type of implementations we consider. Nielsen and Skou [12] use a subclass of timed automata called ERA (event recording automata model) for specifying time dependent applications. The main contribution with their method is a coarser equivalence partitioning of temporal behaviors over the time constraints expressed in the specification. However, here it is inconclusive if the degree of test coverage can be maintained. Laurencot and Castanet [13] recapitulate formal modeling languages which can be used to test real-time systems. The different specifications they compare are the automata of Alur and Dill [10], Timed Transition Models (e.g., [14]), and Extended Time Input Output State Machines. The method decomposes a timed formal model into sub-models for each timer and generates tests by traversing the sub-models. Cardell-Oliver and Glover [14] also starts the test generation process from a formal methods perspective. This requires that the system under test can be viewed as a deterministic finite state automaton. In our target model it is hard to construct such an abstraction that captures all aspects of the temporal behavior for the entire system (due to e.g. data-dependencies and race conditions).

Mandrioli et al. [15] suggest a method for testing of real-time systems based on specifications of system behavior in the temporal logic language TRIO. The elements of 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. For system level testing the number and complexity of logic formulas will increase rapidly. In a more recent work, the authors [16] expand their previous results to incorporate a high-level, structured specifications that can be combined with the low-level specifications proposed earlier for testing modular software applications. This refinement is valuable, but the result is only event sequences, and still no consideration is taken to internal states when events occur.

Morasca and Pezze [17] propose a method for testing concurrent and real-time systems that uses high-level Petri-nets for specification and implementation. 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 are not Petri-nets. Braberman et al. [18] also introduce a method for generating test cases for real-time systems based on timed Petri-net designs. The method uses the design notation SA/SD-RT for specifying the behavior of concurrent real-time systems. The design specification is translated to a timed Petri-net notation from which a timed reachability tree can be derived. Listed future work is to investigate how architectural information, such as scheduling policies, can be taken into account during test-case generation. We believe that the method by Braberman et al. is one of the most promising methods for our purpose since it has the potential to generate both initial state specifications and event-sequences.

Raymond et al. [19] present a method for generating event sequences for reactive systems. Their approach models environmental constraints and test requirements as external observers. However, the method does neither explicitly consider test case generation for testing temporal constraints nor the internal execution behavior of the tested system.

There exist several methods for finding input data that cause the maximum execution time of a task, e.g., static analysis of code and measurements. Wegener et al.[20] propose a method that uses genetic algorithms to generate test data for testing temporal properties of real-time systems. For a thorough analysis of the related work see [21]

5 Conclusions

In this article we briefly outlined a target system domain and stressed the necessity of verifying temporal correctness in that domain. We suggested a platform (and paradigm) for implementing event-triggered applications that increase testability. Problems with current testing methods for event-triggered real-time systems has been outlined and a framework for executing test cases that contain information about the initial system-wide state has been introduced. By incorporating state information in automatically gener-

ated test cases, we avoid the need of running test cases multiple times to find a faulty temporal behavior, and thus, we potentially decrease the test-effort while increasing confidence in temporal correctness.

We argued that by using a transaction-based system as a platform for implementing event-triggered applications, it is possible to exploit the isolation property and enforce system states by running each transaction to a specified designated preemption point. This can in turn be used for gaining control over the test execution and fully automate testing of timeliness.

We have presented an overview of related work, namely frameworks and methods for testing temporal properties of real-time applications, and highlighted their applicability for testing of event-triggered systems. Work is currently in progress to evaluate some of these methods and their associated tools and specification models for automatic test-case generation purposes in the presented framework. Ongoing research also includes further refinement and evaluation of the test-case execution scheme.

References

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

[2] W. Schütz, "Fundamental issues in testing distributed real-time systems," *Real-Time Systems*, vol. 7, no. 2, pp. 129–157, September 1994.

[3] H. Thane, *Monitoring, Testing and Debugging of Distributed Real-Time Systems*, Ph.D. thesis, Royal Institute of Technology. KTH, Stockholm, Sweden, 2000.

[4] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schütz, "An engineering approach to hard real-time system design," in *Proceedings of the Third European Software Engineering Conference*, Milano, Italy, 1991, pp. 166–188.

[5] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, *Deadline scheduling for real-time systems*, Kluwer academic publishers, 1998.

[6] J. Mellin, "Supporting system level testing of applications by active real-time databases," in *In Proceedings 2nd International Workshop on Active, Real-Time, and Temporal Databases, ARTDB-97, number 1553 in LNCS. Springer-Verlag.*, 1998.

[7] 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)*, Hong-Kong, December 1999, pp. 212–215, IEEE Computer Society Press.

[8] 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.

[9] 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.

[10] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[11] R. En-Nouaary, Khendek F. Dssouli, and A. Elqortobi, "Timed test cases generation generation based on state characterization technique," in *Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98)*, Madrid, Spain, December 1998.

[12] B Nielsen and A. Skou, "Automated test generation from timed automata," in *Proceedings of the 21st IEEE Real-Time Systems Symposium*, Walt Disney World, Orlando, Florida, 2000, IEEE.

[13] P. Laurencot and R. Castanet, "Integration of time in canonical testers for real-time systems," in *Proceedings of Int. Workshop on Object-Oriented Real-Time Dependable Systems*, California, 1997, IEEE Computer Society Press.

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

[15] D. Mandrioli, S. Morasca, and A. Morzenti, "Generating test cases for real-time systems from logic specifications," *ACM Transactions on Computer Systems*, vol. 4, no. 13, pp. 365–398, Nov. 1995.

[16] P. SanPietro, A. Morzenti, and S. Morasca, "Generation of execution sequences for modular time critical systems," *IEEE Transactions on Software Engineering*, vol. 26, no. 2, pp. 128–149, feb 2000.

[17] S. Morasca and M. Pezze, "Using high level Petri-nets for testing concurrent and real-time systems," *Real-Time Systems: Theory and Applications*, pp. 119–131, 1990, Amsterdam North-Holland.

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

[19] Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber, "Automatic testing of reactive systems," in *Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98)*, 1998.

[20] J. Wegener, H. H. StHammer, B. F. Jones, and D. E. Eyres, "Testing real-time systems using genetic algorithms," *Software Quality Journal*, vol. 6, no. 2, pp. 127–135, 1997.

[21] R. Nilsson, "Automated selective test case generation methods for real-time systems," M.S. thesis, University of Skövde, September 2000.

Exact Best-Case Response Time Analysis of Fixed Priority Scheduled Tasks

Ola Redell and Martin Sanfridson
Mechatronics Laboratory, Dept. of Machine Design, KTH
{ola, mis}@md.kth.se

Abstract

We present a solution for exact calculation of the best-case response times of a periodic task set with fixed priorities. The solution is based on the identification of the best-case phasing of a low priority task compared to the higher priority tasks. This phasing occurs when the low priority task is released such that it finishes simultaneously with the releases of all the higher priority tasks, when these have experienced their maximum release jitter. A recurrence equation is applied to find the best-case response time. The dualism between worst-case and best-case response time calculation is characterized. The most important application of the solution is in the analysis of response jitter.

1. Introduction

In this paper we present the solution to the previously unsolved problem of finding the minimum response time of tasks in a preemptive fixed priority task set. The solution is based on the identification of the *best-case phasing* of a low priority task compared to its higher priority tasks. We show that a task has its best-case phasing when it finishes its execution at an instant when all higher priority tasks are released, after having experienced their maximum release jitter. The response time of a task that has executed in its best-case phasing is the shortest possible. Hence, the suggested method finds the exact best-case response time and not a lower bound, assuming that tasks can be phased arbitrarily.

Real-time system analysis has traditionally been focused on the analysis of worst-case behaviour in order to provide guarantees for tasks meeting their deadlines. There are however situations when the best-case behaviour of a task or a system is important as well. One such situation is in the calculation of response jitter, the maximum variation in the response time of a task or a sequence of tasks. Typically, response jitter is an important parameter in control systems. Control theory is dependant on the assumption that sampling and actuation is performed with precise periodicity and hence large variations in these periods can make an otherwise stable control system become unstable [8]. In order to guarantee control performance and stability it is therefore important to find tight bounds to the magnitudes of such variations. A good best-case analysis is a necessity for such tight bounds.

Analysis of fixed priority preemptive tasks known as RMA (Rate Monotonic Analysis), has been well developed since the paper by Liu and Layland in 1973 [5], from which it all originated. The set of analysis methods now handles a wide range of systems in which the original restricting assumptions of RMA have been successively removed. Systems that can be analysed include systems with task synchronisation (blocking), a mixture of periodic and aperiodic tasks, tasks with arbitrary deadlines etc. [1][3][4].

Even though RMA was initially formulated as a theory for analysis of tasks executing on a single processor, some extensions cover scheduling analysis in distributed systems. A method proposed by Tindell and Clark [7] produces worst-case response times of precedence related tasks in distributed systems. The method is based on local analysis of each node with the response jitter of one task or message being the release jitter of the next task in the sequence. Pessimistic bounds of response jitter are in this method a source for pessimism in the bounds of subsequent task response times. Hence, too pessimistic jitter bounds may make a schedulable system being considered unschedulable. Since response jitter is the difference between the worst- and best-case response times, finding tight bounds on these response times is important. While good methods exist for determining worst-case response times of tasks, the best-case has not received the same attention.

The method by Tindell and Clark was further developed by Palencia et al. [6] with an estimate of best-case response times. Palencia et al. find a lower bound on the response time of a task by assuming (optimistically) that all higher priority tasks finish at the instant the analysed lower priority task is released (after having experienced their best-case response times). This results in a correct lower bound for the best-case response times of tasks, but it is not exact. In a recent article, Henderson et al. [2] tried to further develop the ideas in [6] and find the exact best-case response times. However, their solution leads to a numerically intractable search through all possible orderings of higher priority task executions, prior to the release of the analysed low priority task.

This paper is organized as follows: In section 2 we define the task model and the problem to be solved. In section 3 we present an exact solution to the best-case analysis of task response times when the task deadlines are shorter than the periods. Analysis of task sets with longer deadlines is discussed in section section 4. Section 5 discusses the applicability of the method and gives suggestions for further work. Section 6 offers our conclusions.

2. Task model and problem formulation

A uniprocessor executes a set of independent tasks τ_k with $k = 1, 2,..., m$. Each task is periodic and characterized by the following parameters:

- a period time, T_k
- a fix and unique priority
- an execution time interval, $[C_k^{min}, C_k^{max}]$, limited by the task's minimum and maximum execution times. Each task instance may assume any execution time within the interval.
- a deadline, D_k, such that $D_k \leq T_k$
- a maximum release jitter, J_k.

A task τ_k *arrives* periodically with period T_k. On arrival, the task is either *released* (put in the ready queue) immediately, or the release is delayed by an amount of time called *release jitter*. The magnitude of the jitter can change from one instance to the next, but is bounded by $[0, J_k]$. The arrival time of a task τ_k is denoted a_k, while the release time is r_k. After release, a task starts its execution (*start time*) as soon as it is the highest priority ready task in the system. The *finishing time* of a task is denoted f_k and the response time R_k is defined to be the time between arrival and finish: $R_k=f_k-a_k$. In the following, tasks are labelled by their numbers. The analysed task is denoted task i while task j denotes a task in $hp(i)$. $hp(i)$ is the set of tasks with higher priorities than task i.

There are no exclusion or precedence constraints that can make a lower priority task block the execution of a higher priority task. Hence, the only sources of variation in the response time of a task is the execution time, release jitter and interference from higher priority tasks. The problem is to calculate the best-case response times R^b of the tasks in the task set.

As a reference for the solution described in this paper, we state the equivalent result for worst-case calculations here [1]. The worst-case response time R^w of a task i is expressed by:

$$R_i^w = w_i + J_i \tag{1}$$

where

$$w_i = C_i^{max} + \sum_{j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil \cdot C_j^{max} \tag{2}$$

Since equation (2) cannot be solved analytically, it has to be solved by iteration starting with e.g. $w_i = C_i^{max}$. When the iteration has converged, equation (1) is used to calculate the worst-case response of task i.

We will henceforth assume that any task set that is subject to best-case analysis has already been verified to be schedulable. Such a verification could be done by comparing the worst-case response times computed by (1) to the task deadlines.

The assumption that the task deadlines are smaller than the periods is a necessary requirement for the analysis derived in this paper to be exact. In section 4 we discuss how task sets that do not satisfy this requirement can be handled.

3. Best-case Analysis

The response time of a task is the duration between its arrival time and its corresponding finishing time. Because of interference, a low priority task can not always start executing immediately when it is released. We use Lemma 1 to state that a task that experiences its best-case response time must have arrival, release and starting times that coincide.

Lemma 1. A task that achieves its minimum response time must arrive and be released simultaneously, at an instant when no higher priority task is executing or is released.

Proof. Since any release jitter increases the response time, it is clear that the best-case response time corresponds to zero release jitter. Hence the arrival and release times must coincide.

Assume that a low priority task *i* arrives when a higher priority task *j* occupies the processor. The response time of *i* will always be shortened by delaying the arrival time to the instant when the higher priority task has stopped executing. This comes from the fact that the delay of *i*'s arrival time will not affect its start or finishing times and the response time of *i* is counted from its arrival to its finish.

Q.E.D.

In the following we will assume that the analysed task satisfies Lemma 1 and hence has arrival, release and start times that coincide.

To calculate the best-case response time, we first need to know when the low priority task *i* should be released. Obviously, the worst release time is the critical instant of that task. It is a very reasonable assumption that task *i* should be released before this instant such that it finishes exactly when all higher priority tasks *j* arrive simultaneously. For example, consider Figure 1 which shows three priority-ordered tasks with no release jitter. If the releases of the instances of task 2 are *advanced* in time (moved to the left) the release (and arrival) instant r_3 needs to be *delayed* (moved to the right) to keep the finishing time at f_3, rendering a shorter response time of task 3.

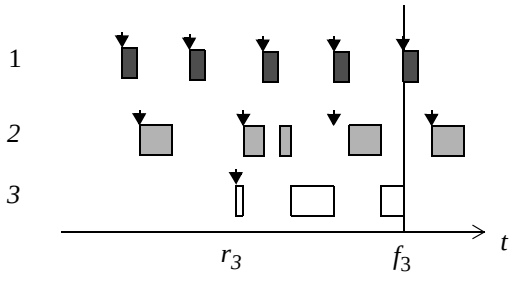


Figure 1. Example of phasing two higher priority tasks relative to task 3 which is released at r_3 and finishes at f_3. Vertical arrows indicate release instants. No release jitter is assumed in this example.

In fact, Theorem 1 states what phase every high priority task should have relative to the finishing time of the low priority task *i* in order for task *i* to experience its best-case response time. There may be other choices of phases that will give an equally short response time, but none will be shorter. Note that contrary to the case of critical instant, it is the finishing time of *i* that is considered, not the release time.

Theorem 1. Best-case phasing.
The best-case phasing of a task *i* occurs whenever it finishes simultaneously with the release of all its higher priority tasks and these have experienced their maximum release jitter. All instances of tasks in *hp*(*i*) released prior to the finish of *i* should have zero release jitter.

Proof. Let τ_j, $j = 1, 2, 3,..., i$ denote a set of priority-ordered tasks with task *i* being the task with the lowest priority. The task set is successfully scheduled on a uniprocessor. Consider a particular execution of task *i* which finishes at time f_i. Instances of a task *j*, $j < i$, arrive periodically at times $\{..., t_j-2T_j, t_j-T_j, t_j, ...\}$ where the instance of *j* that arrives at t_j is the last instance of task *j* to execute before f_i.

We assume, without loss of generality, that task *i* will always finish at f_i. Hence, task *j* is not allowed to be phased such that any of its instances executes at f_i. In the following, we will use the fact that the release time of task *i* is given implicitly by f_i, the phasing of higher priority tasks, their release jitter and Lemma 1.

The release time of *i* is either unchanged or delayed by reducing the release jitter of an instance of task *j* that arrives at t_j or earlier. Increasing the release jitter of one of these instances will not delay the release time of *i*. Hence the instances of task *j* up to and including the one that arrives at t_j should have zero release jitter for task *i* to have its best-case phasing.

Delaying the arrival time t_j will not delay the release time of *i*. Advancing the arrival time t_j will not advance the release time of *i*. The release time of *i* is either unchanged or delayed by an advance of t_j. As a consequence, the release of *i* is the latest when t_j is advanced as much as possible – up to a point when a succeeding instance of *j* gets released at f_i. The succeeding instance arrives at $t_j + T_j$. Release jitter will delay its release enabling a further advance of t_j. Thus, the instance of *j* that is released at f_i should experience its maximum release jitter, J_j. Repeating the argument successively for all τ_j, $j = 1, 2, 3,..., i-1$ proves the theorem.

Q.E.D.

Remark. The instant f_i associated with the finishing time of task *i* is identical to the *critical instant* of task *i*-1 as defined in [5]. We call the finishing time f_i in the situation of best-case phasing the *favourable instant* to underline the dualism between worst-case and best-case analysis.

As a direct result of Theorem 1, we formulate the following corollary which will be useful in the discussion about task sets with deadlines longer than the periods in section 4.

Corollary 1. Given a schedule of tasks 1,...,*i*-1, there cannot exist an interval with arbitrary length *L*, in that schedule, that includes more idle time than the interval $[f_i\text{-}L, f_i]$. Where f_i is a favourable instant of some lower priority task *i* as defined in Theorem 1.

Proof. Assume that an interval $[a, a+L]$ in the schedule exists that does include more idle time than $[f_i\text{-}L, f_i]$. Let the total amount of idle time in $[a, a+L]$ be *l*. Assume that a low priority task *i* with $C_i^{min} = l$ and $T_i = \text{LCM}(T_1, \ldots, T_{i-1})$, is added to the task set. This task would achieve its minimum response time if it was executed in $[a, a+L]$ and not in $[f_i\text{-}L, f_i]$. This contradicts Theorem 1 and hence such an interval $[a, a+L]$ cannot exist.

Q.E.D.

We next continue by deriving an expression for the response time of a task *i*, when it is phased as described in Theorem 1 and when it satisfies Lemma 1. This is the best-case response time of task *i*.

Theorem 2. Best-case response time.
Given a schedulable set of independent fixed priority scheduled tasks with deadlines equal to or smaller than their periods, the best-case response time R_i^b of a task *i* in the set satisfies the following equality:

$$R_i^b = C_i^{min} + \sum_{j \in hp(i)} \left\lceil \frac{R_i^b - J_j - T_j}{T_j} \right\rceil_0 \cdot C_j^{min} \qquad (3)$$

where $\lceil x \rceil_0 = \max(0, \lceil x \rceil)$.

Proof. Assume a best-case phasing of task *i* according to Theorem 1 and that the arrival, release and start times of task *i* coincide as stated in Lemma 1. We define the phase from the release of task *i* to the first following release of a higher priority task *j* to be ϕ_j. Figure 2 shows the best-case phasing of a low priority task 3 when executed together with the two higher priority tasks 1 and 2. Tasks 1 and 2 are released at $\{..., f_3\text{-}2T_1\text{-}J_1, f_3\text{-}T_1\text{-}J_1, f_3, ...\}$ and $\{..., f_3\text{-}2T_2\text{-}J_2, f_3\text{-}T_2\text{-}J_2, f_3, ...\}$ respectively. Task 3 arrives at at times $\{..., r_3\text{-}T_3, r_3, ...\}$. Task instances released at and after f_3, do not need to be considered since they

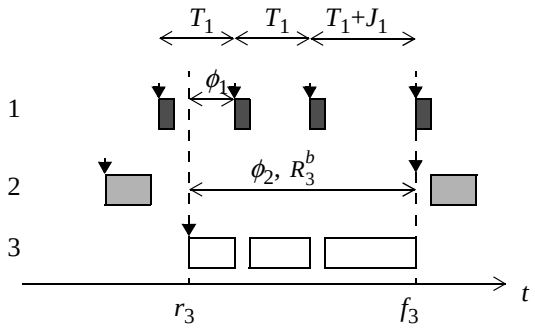


Figure 2. The best-case phasing of task 3 with release time r_3 and finishing time f_3.

Task *i*'s response time will be minimal only if its execution time is minimal. Hence, the minimum response time of task *i* can be expressed as the sum of the minimum execution time and interference by all higher priority tasks:

$$R_i^b = C_i^{min} + \sum_{j \in hp(i)} I_i^b(j) \qquad (4)$$

Here $I_i^b(j)$ is the interference of task *i* by task *j* when the tasks are phased according to the best-case phasing for task *i*. Hence, in order to find the best-case response time of task *i*, we need to find the amount of interference due to all higher priority tasks. By studying Figure 2 we find that there are two different cases to be considered for the interference of a higher priority task *j* to the execution of task *i*:

Case 1: $R_i^b > T_j + J_j$

and

Case 2: $R_i^b < T_j + J_j$

where $R_i^b = f_i - r_i$. Since R_i^b is the best-case response time of *i* it cannot be equal to $T_j + J_j$ as a consequence of Lemma 1. A more general inequality, based on Lemma 1, that holds for all higher priority tasks *j* is:

$$R_i^b \neq k \cdot T_j + J_j \text{ for } k = 1, 2, \ldots \qquad (5)$$

Case 2 is exemplified by task 2 which has T_2+J_2 large enough not to interfere with task 3's execution at all. This case is obviously simple to handle. Case 1 is exemplified by task 1, which does interfere with task 3. It follows from Figure 2 that R_3^b can be expressed as

$$R_3^b = f_3 - r_3 = \phi_1 + 2 \cdot T_1 + J_1$$

while a general relation between the best case response time of a low priority task i and any $j \in hp(i)$ that belongs to case 1 is

$$R_i^b = f_i - r_i = \phi_j + n_j \cdot T_j + J_j \tag{6}$$

where $n_j \in \{1, 2, \ldots\}$ is the number of instances of task *j* that interfere with task *i*. As a result of Lemma 1 we have

$$0 < \phi_j < T_j \tag{7}$$

and by rewriting (6) as

$$\phi_j = R_i^b - n_j \cdot T_j - J_j \tag{8}$$

we get

$$0 < R_i^b - n_j \cdot T_j - J_j < T_j \tag{9}$$

Rewriting this expression results in

$$\frac{R_i^b - J_j}{T_j} > n_j > \frac{R_i^b - J_j}{T_j} - 1 \tag{10}$$

Now, since $(R_i^b - J_j)/T_j$ cannot be an integer (by equation 5), there are two equally valid solutions for n_j in (10):

$$n_j = \left\lfloor \frac{R_i^b - J_j}{T_j} \right\rfloor \tag{11}$$

and

$$n_j = \left\lceil \frac{R_i^b - J_j}{T_j} \right\rceil - 1 = \left\lceil \frac{R_i^b - J_j - T_j}{T_j} \right\rceil \tag{12}$$

Any of the two expressions (11) and (12) can be used to express the number of instances of task *j* that interfere with *i* during its best-case execution. We will henceforth use (12) since it has characteristics better suited for finding the best-case response time of task *i*. This will be explained later.

By combining the results for the two cases of higher priority task interference, we find that the number of interfering instances of task *j* can be expressed as:

$$n_j = \begin{cases} 0 & \text{for } R_i^b < T_j + J_j \\ \left\lceil \dfrac{R_i^b - J_j - T_j}{T_j} \right\rceil & \text{for } R_i^b > T_j + J_j \end{cases} \tag{13}$$

which is equivalent to

$$n_j = \left\lceil \frac{R_i^b - J_j - T_j}{T_j} \right\rceil_0 \tag{14}$$

It is now possible to derive an expression for the interference of a task *i* by a higher priority task *j*, when task *i* is phased according to the best-case phasing defined in Theorem 1. By taking into account that, in the best case for task *i*, all interfering instances of task *j* execute with their minimum execution times, we find the interference to be

$$I_i^b(j) = \left\lceil \frac{R_i^b - J_j - T_j}{T_j} \right\rceil_0 \cdot C_j^{min} \tag{15}$$

We now add up the interference of all tasks in $hp(i)$ and the minimum execution time of task *i* itself, as given by (4). This results in the expression for the best-case response time of *i* as shown in (3).

Q.E.D.

At this point we have an expression, (3), that the best-case response time of *i* must satisfy. Just like in worst-case response time calculations however, this equation cannot be solved analytically. Therefore, the minimum response time has to be found through iteration. We now need to show that this is possible. We restate equation (3) as a recurrence equation,

$$R^{n+1} = C_i^{min} + \sum_{j \in hp(i)} \left\lceil \frac{R^n - J_j - T_j}{T_j} \right\rceil_0 \cdot C_j^{min} \tag{16}$$

and note that the right hand side corresponds to the

complete high priority workload released in an open interval $(f_i\text{-}R^n, f_i)$ plus the workload of task i itself. Figure 3 shows how R^n should be interpreted.

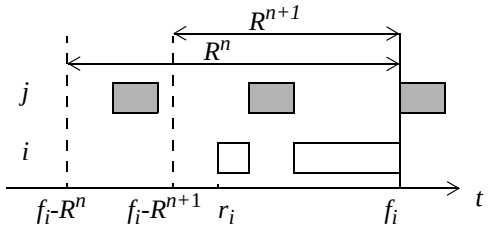


Figure 3. Interpretation of R^n during recursion.

We first need to show that recursion over (16) will converge to a stable value. Then we need to show that this value is the correct minimum response time and not one of the other possible solutions to (16) (there is usually more than one, see e.g. Figure 4).

We note that there must exist a finite R^0 such that the interval $(f_i\text{-}R^0, f_i)$ includes some interval during which the processor is either idle, executes tasks with a priority lower than i, or executes a previous instance of task i (whose workload is not included in (16)). If the iteration is started with such an R^0 we know that $R^1<R^0$. We will call an R^0 of this type a *valid* R^0.

Lemma 2. Iteration over R^n in (16) starting with a valid R^0, will converge to the largest possible solution to (16).

Proof. By observing that the right hand side of (16) is monotonically decreasing with decreasing R, and since the number of possible values produced by (16) is finite, the iteration will converge. It is obvious that this solution is the largest solution to (16).

Q.E.D.

We will henceforth use R^* to denote the largest solution of (16). We also note that any R^0 larger than or equal to R^* will make the iteration converge to R^*. It now remains to be shown that R^* is in fact the best-case response time of task i. The two following lemmas take care of that:

Lemma 3. The instant $f_i\text{-}R^*$ must correspond to the release (and start) of task i.

Proof. We show this by making two statements about R^* that can easily be verified through observation of (16).

First, R^* corresponds to an interval $[f_i\text{-}R^*, f_i]$ during which the processor is completely occupied executing task i or higher priority tasks. Furthermore all these tasks are released within the interval $[f_i\text{-}R^*, f_i)$.

Second, R^* cannot correspond to an instant $f_i\text{-}R^*$ at which a task in $hp(i)$ is released. This is easiest understood by noting that the workload due to tasks in $hp(i)$ that is included in (16) must be released within the open interval $(f_i\text{-}R^*, f_i)$. This behaviour is a result of choosing expression (12) instead of (11) for the number of interfering instances of a higher priority task.

By combining the two observations, we find that task i must be released and start its execution at $f_i\text{-}R^*$.

Q.E.D.

Lemma 4. The best-case response time of a task i is equal to R^*.

Proof. Assume that R is a solution to (16) such that $R^*>R$. Then there must be a release of at least one task j in $hp(i)$ within the interval $(f_i\text{-}R^*, f_i\text{-}R]$ (interval open to the left by Lemma 3) that has a total execution time of $R^*\text{-}R$ and therefore interferes with the execution of i. Hence, R cannot be the best-case response time of i. This argument can be applied to any R that is smaller than R^*, and therefore R^* must be the best-case response time of i.

Q.E.D.

Figure 4 shows the difference between the two solutions R and R^* of (16), for a task set with two tasks.

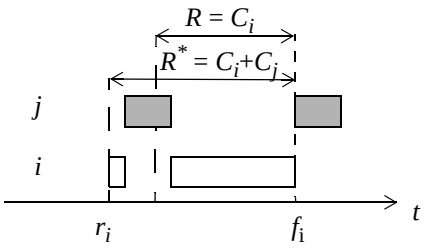


Figure 4. Two solutions to (16): R and R^*. R cannot be the best-case response time of i.

Theorem 3. Best-case response time calculation. The best-case response time of a task i, R_i^b, can be found by iteration over (16) starting with a value $R^0 = R_i^w$ as derived from (1). When the iteration has converged, the corresponding R^n is equal to the best-case response time of task i.

Proof. The worst-case response time of i, R_i^w, is

always larger than or equal to R_i^b and hence R^*. Therefore, by Theorem 2 and Lemmas 3 and 4, the iterations will converge to R_i^b.

Q.E.D.

Remark. Another possible initial value R^0 that is always larger than or equal to R_i^b is T_i.

4. Analysis of Task Sets with D > T

In the above analysis, one important assumption has been that a task's deadline has to be smaller than or equal to its period. In this section we discuss how the analysis is affected if this assumption is removed. It will be evident that the analysis derived in section 3 can be used to compute correct lower bounds on the best-case response times of tasks with deadlines longer than the periods.

We first note that even though deadlines may be larger than the periods, the interval (of arbitrary length) with the most amount of idle time is still correctly specified by Corollary 1. This result does not depend on the deadline of the imaginary lower priority task. Hence, if an analysed task *i* can be phased to arrive, be released, and start its execution simultaneously and then finish at the favourable instant as defined in Theorem 1, the analysis derived in section 3 would still be exact. Unfortunately however, when tasks are allowed to have deadlines larger than their periods, the exact analysis falls. This is due to the fact that earlier instances of task *i* may interfere with the execution of the analysed instance. Therefore the implication of Lemma 1, that a task that experiences its best case response time must start its execution at the instant it arrives, does not hold.

To illustrate this, consider the following simple example involving a task set with two tasks. Task *j* which has the highest priority has a constant execution time of $C_j^{min} = C_j^{max} = 4$ and a period $T_j = 8$. Task *i* has a period $T_i = 5$ and an execution time $C_i^{min} = C_i^{max} = 2.5$. Both tasks have zero release jitter ($J_i = J_j = 0$) and their deadlines are $D_j = 8$ and $D_i = 10$. The total utilisation of the task set is 1 and the tasks are feasibly schedulable on a uniprocessor (this can be verified by, for example, a simple schedule simulation). Figure 5 shows the execution of two instances of task *j* together with the execution of two instances of task *i*. The instances of task *i*, numbered 1 and 2, are released at 0.5 and 5.5 respectively. The release times have been chosen such that instance 2 would finish at a favourable instant ($t = 8$) unless instance 1 would interfere with its execution. However, due to interference from instance 1, instance

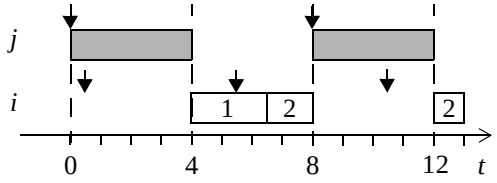


Figure 5. Instance 2 of task *i* cannot start executing when it is released and hence fails to finish at 8 due to interference from an earlier instance. Vertical arrows indicate release times.

2 finishes at 13. It is obvious that, if instance 2 is to finish at 8, task *i* has to be phased to be released earlier. Hence in this case, task *i* cannot be released such that it starts executing immediately, if it is to finish in a favourable instant as defined by Theorem 1.

As a direct consequence of the above discussion, we find that allowing deadlines larger than task periods can not make the best case response times get smaller. Hence, if we assume (optimistically) that the analysed task *i* will not experience any interference by preceding instances, the analysis method described in section 3 can be used to compute a lower bound to the best case response time of task *i*. However, the value produced will not be exact, as is the case when $D_i \le T_i$.

5. Discussion

The best-case response time analysis derived in section 3 is similar to the corresponding worst-case analysis. One major difference is that in best-case analysis, the finishing time of the analysis task is fixed to the release of all the higher priority tasks, and the algorithm searches the latest possible release time of the task. Furthermore, the iteration performed to find the best-case response time must be started with a value larger than or equal to the final solution, which is the largest value satisfying the recurrence equation. If a task model with no release jitter is used, equation (3) becomes:

$$R_i^b = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^b - T_j}{T_j} \right\rceil \cdot C_j \qquad (17)$$

Hence, the maximisation over zero can be skipped since R_i^b can never be smaller than or equal to zero. Obviously equation (17) is very similar to the corresponding equation for worst-case analysis.

The inclusion of release jitter in the task model used in the derivation of (3) makes it applicable to methods for response time analysis in distributed

systems with precedence related tasks, as discussed in section 1. However, when using any best-case analysis in such methods, one must be aware that the precedence relations between tasks may make the best-case analysis incorrect. This is a result of the fact that the tasks in such a system are not independent and low priority tasks in one task sequence may block higher priority tasks in another.

The exact best-case analysis is based on the assumption that tasks can be arbitrarily phased. This means that the periodic arrival times of different tasks can assume any relative phasing to each other. If a task model with offsets is used instead, fixing the phasing between task arrivals, the analysis described here can be used to produce lower bounds on the best-case response times. This is similar to the upper bounds on worst-case response times found by corresponding worst-case analysis methods when phases are not arbitrary.

An obvious target for future work will be to find exact best-case response times for tasks with deadlines larger than their periods.

6. Conclusions

We have derived an algorithm for the calculation of *best-case* response times for independent tasks with fixed priorities. The method produces exact best-case response times for all tasks with $D_i \leq T_i$, and lower bounds for tasks with deadlines longer than the periods. If offsets are used in the task model such that the tasks are not arbitrarily phased, the analysis produces lower bounds on the best-case response times.

The presented algorithm is similar to the one frequently used in analysis of *worst-case* response times, and the dualism between best-case and worst-case is quite clear. A difference in the recurrence equation defined here is that the iteration starts from an initially high value and iterates down to the best-case response time. The solution is found at the release time of the low priority task; whereas in worst-case response time analysis, the solution is found at the finishing time.

We have also identified the best time for a low priority task to finish, rendering the shortest possible response time. The *favourable instant* for a low priority task to finish execution is the *critical instant* of the next higher priority task.

7. Acknowledgements

The authors wish to acknowledge Christer Norström, Martin Törngren and Jan Wikander for valuable comments to early versions of this paper.

This work was partially supported by SSF, the Swedish strategic research initiative, through ARTES project 0005-18: PICADOR, and partially supported by VINNOVA, the Swedish agency for innovation systems, through the DICOSMOS project.

References

[1] N. Audsley, A. Burns, K. Tindell, M. Richardson and A. Wellings, “Applying New Scheduling Theory To Static Priority Pre-emptive Scheduling”, Software Engineering Journal, 8(5):284-292, September 1993.

[2] W. Henderson, D. Kendall and A. Robson, “Improving the Accuracy of Scheduling Analysis Applied to Distributed Systems”, Intl. Journal of Real-Time Systems, Kluwer, 20(1):5-25, January 2001.

[3] M. Klein, T. Ralya, B. Pollack, R. Obenza and M. González Harbour, “A Practitioners Handbook for Real-Time Systems Analysis”, Kluwer Academic Publishers, 1993.

[4] J.P. Lehoczky, “Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines”, Proc. of the 11th IEEE Real-Time Systems Symposium, pp. 201-209, December 1990.

[5] C.L. Liu and J.W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”, Journal of the ACM, 20(1):46-61, 1973.

[6] J.C. Palencia Gutiérrez, J.J. Gutiérrez García and M. González Harbour, “Best-Case Analysis for Improving the Worst-Case Schedulability Test for Distributed Hard Real-Time Systems”, Proc. of tenth Euromicro Workshop on Real-Time Systems, Berlin, pp. 35-44, June 1998.

[7] K. Tindell and J. Clark, “Holistic Schedulability Analysis for Distributed Hard Real-Time Systems”, Microprocessing & Microprogramming, 50(2-3):117-134, April 1994.

[8] M. Törngren, “Fundamentals of implementing Real-time Control applications in Distributed Computer Systems”, J. of Real-time Systems, Kluwer, 14:219-250, 1998.

ACTA UNIVERSITATIS UPSALIENSIS
Uppsala Dissertations from the Faculty of Science and Technology 36
Distributor: Uppsala University Library, Box 510, SE-751 20 Uppsala, Sweden

Jakob Engblom

Processor Pipelines and Static Worst-Case Execution Time Analysis

Dissertation in Computer Systems to be publicly examined in Polhemssalen, Ångströmlab, Uppsala University, on Friday, April 19, 2002 at 01:15 PM for the Degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract

Engblom, J. 2002. Processor Pipelines and Static Worst-Case Execution Time Analysis. *Uppsala Dissertations from the Faculty of Science and Technology* 36. 130 pp. Uppsala. ISBN 91-554-5228-0.

Worst-Case Execution Time (WCET) estimates for programs are necessary when building real-time systems. They are used to ensure timely responses from interrupts, to guarantee the throughput of cyclic tasks, as input to scheduling and schedule analysis algorithms, and in many other circumstances. Traditionally, such estimates have been obtained either by measurements or labor-intensive manual analysis, which is both time consuming and error-prone. Static worst-case execution time analysis is a family of techniques that promise to quickly provide safe execution time estimates for real-time programs, simultaneously increasing system quality and decreasing the development cost. This thesis presents several contributions to the state-of-the-art in WCET analysis.

We present an overall architecture for WCET analysis tools that provides a framework for implementing modules. Within the stable interfaces provided, modules can be independently replaced, making it easy to customize a tool for a particular target and perform performance-precision trade-offs.

We have developed concrete techniques for analyzing and representing the timing behavior of programs running on pipelined processors. The representation and analysis is more powerful than previous approaches in that pipeline timing effects across more than pairs of instructions can be handled, and in that no assumptions are made about the program structure. The analysis algorithm relies on a trace-driven processor simulator instead of a special-purpose processor model. This allows us to use existing simulators to adapt the analysis to a new target platform, reducing the retargeting effort.

We have defined a formal mathematical model of processor pipelines, which we use to investigate the properties of pipelines and WCET analysis. We prove several interesting properties of processors with in-order issue, such as the freedom from timing anomalies and the fundamental safety of WCET analysis for certain classes of pipelines. We have also constructed a number of examples that demonstrate that tight and safe WCET analysis for pipelined processors might not be as easy as once believed.

Considering the link between the analysis methods and the real world, we discuss how to build accurate software models of processor hardware, and the conditions under which accuracy is achievable.

Jakob Engblom, Department of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden.

The Potential of Resource Budgets in Complex Real-Time System Design

Mattias Weckstén and Jonas Vasell
Department of Computing and Communication
Halmstad University
[Mattias.Wecksten, Jonas.Vasell]@ide.hh.se

April 2 , 2002

Abstract

To make the complex system design task clearer this paper suggests a tool that will generate, evaluate, and annotate budget proposals. The generation of budgets is based on a task graph representation of the system together with an architectural description. Some optimization criteria (e.g. timing, energy, and cost) will be evaluated for each budget and compared with earlier results in order to get the "best". The best proposals will then be annotated for better understanding of how the separate parts of the system depends on each other. To do exhaustive optimization of this search space is not possible for larger problems why a number of heuristics are introduced.

1 Design of Complex Real-Time Systems

When designing complex real-time systems it is desirable to find a practical method to evaluate the proposed solution as early as possible in the design cycle, improving the possibility of meeting the non-functional demands. To make the design task iterations easier it is also desirable to make relations between different parts of the system clearer. A typical design process could look like Figure 1, where the work starts with functionality analysis and requirement specification and ends with an implementation.

1.1 Approach

The suggested approach to accomplish this is to identify functional units in the system and then assign a resource budget for each unit, in such way that the possibility of implementing the function to fulfill the non-functional demands is maximized. To make the system more stable to corrections and alterations, the dependencies between budgets should be annotated. The term *resource* include, among other things

- Execution time
- Processor element
- Memory

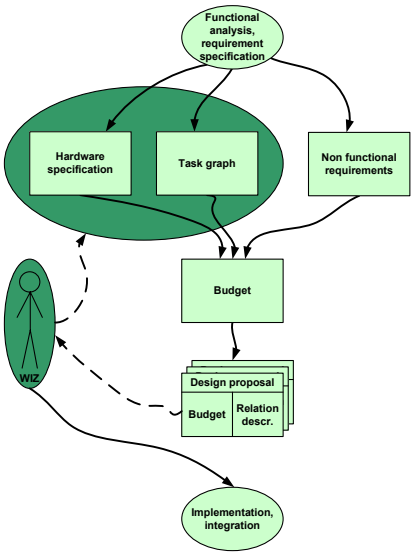


Figure 1: A typical design process where functional and non functional demands are used to generate a budget proposal. The proposal could then be feed back for more iterations or implemented.

- Energy

Note that resource budget assignments should be based on relative estimates of the actual resource need since we are at early design stages.

1.2 Work Focus

In this paper we will assume that input data is present in the form of an acyclic task graph describing the precedence constraints, where each task has an execution time estimate and an estimate of the accuracy. The architecture will be modelled as homogenous resources connected by a single bus, which gives us the maximum number of resources and the bus speed as parameters. When resource budgets are assigned we will only address the assignment of allowed execution time and static processor to reside on. To do this we will utilize each tasks estimated execution time in conjunction with the end-to-end requirements for the system, in our case deadlines and release-times. The result we look for is the task graph with the minimum tightness, where "tightness" is equal to the path through the task graph with the most work to execute in comparison to the length of the path. This will all sum up to a number of practical methods oriented towards giving a system designer support in finding and evaluating design alternatives. These methods does not necessary strive for optimality (in the case that it would be possible to define), but rather a set of good estimates that will point us in the right direction.

1.3 Related Work

One solution for part of this problem was presented in [Zha99], where the author assumes we have an allocated system in the form of a task graph with end-to-end timing constraints and the only thing left is to try different execution orders for the tasks on the different resources in the system. The measure of how good a specific "order" is inversely proportional to the tightness. Based on this measure we can allocate more, or if needed - less, time for each task in the system so the idle time is kept at a minimum, but still budgeted in a "fair" fashion. A similar method that scales well with problem size is presented in [AH98] that has the drawback that it sometime "misses" feasible assignments.

The author of [Axe97] discusses resource budgets and their importance and calls attention to that one great benefit of the resource budgets is that they localize the global constraints. He also point out that [Mey88] introduced resource budgets under the notation "programming by contract".

In [Eke01] the author describes how constraint programming can be applied to the problem of scheduling and allocation and shows how this can be used for modelling as well as for implementation.

In [GiKHS95] the authors present a method to break down end-to-end requirements to task specific demands and, if this is not possible, to identify bottlenecks in the system. This information is then used as input in a design tool where the system could be restructured for another iteration step.

In [NS94] we find a similar method called slicing that does not address budgets but focus on how to increase online schedulability in a distributed system with off-line calculations. A discussion about the same method but under more relaxed forms (i.e. at early design stages) is found in [JS97].

2 Solution Overview

As mentioned earlier, the assumption is that in-data is available as an acyclic task graph with estimations of the execution time in each task and also end-to-end timing constraints for each possible trace. Besides the actually estimate of the workload we need to describe the actual architecture, which for now is modelled as homogenous resources connected with one single bus. The parameters for the architecture are thus maximum number of resources and bus speed. With this raw data the budgeting tool is supposed to try out all possible (or promising) solutions, or in other words generate budgets for all possible allocations and execution orders in the graph. The budgets with the lowest tightness are kept in a repository and finally returned to the user. The actual process chain consists of the following stages:

Ordering Decides in which order the tasks in the task graph should be executed if they were allocated on one single resource.

Allocation Decides on which resource a task should reside. After the actual allocation communication overhead is added.

Budgeting Distributes the "slack" in the task graph among all nodes to make implementation easier.

2.1 Budgeting

The budgeting aims to distribute as much of the idle time in the processors as possible in some "fair" fashion. The proposed scheme starts with an ordered and allocated graph and distributes the slack in proportion to the estimated execution times in the tasks. This will always render a valid solution if it is done with the path with least slack first and then in ascending order. To do this we need to find the "tightest" path through the graph. To do so, we find all possible paths through the graph, where a path is a number of nodes that can be accessed in the given order (see Section 2.3), following the edges in the graph. For each possible path we assign a "tightness" that is the total work on the path, or the sum of all estimates on the path, divided with the length of the actual path, or the deadline for the last node minus the release time for the first (see Equations 1 – 4).

$$P = [t_1, t_2, \ldots, t_N] \tag{1}$$

$$W_p = \sum_{i=1}^{N} e_i \tag{2}$$

$$L_p = D_p - R_p \tag{3}$$

$$T_p = \frac{W_p}{L_p} \tag{4}$$

The tightest path is the path with the largest T_p-value. When found, time is distributed over this tightest path proportional to the execution time estimate in each node. To make the new W_p fill all slack between R_p and D_p, each estimate is multiplied with the inverse of the tightness (see Equation 5).

$$\sum_{i=1}^{N} \frac{e_i}{T_p} = \frac{\sum_{i=1}^{N} e_i}{W_p} \cdot L_p = L_p \tag{5}$$

To make a distinction between the original estimate and the new proposal, the new value is named "Allowed Execution Time" or for short AET. The nodes on the tightest path is now removed and replaced with deadlines or release times where so is needed. The procedure is now repeated until no more nodes are left in the graph.

Unfortunately this method will render a lot of possible paths to search through for each budget and it seems to not be especially scalable for many nodes. What is even worse is that this algorithm is the kernel in the system and will be executed millions and millions of times for each run. We have to come up with something better than the naive "search-all" method.

Budgeting Based on Uncertainty

One interesting variation of the standard budgeting algorithm, where execution time estimates is base for the budget allocation, is to assign a relative uncertainty value to each of the tasks to describe how sure you are of your execution time estimate. The new budgeting algorithm would divide the available slack in proportion to the execution estimate and the uncertainty. In other words, the task which we know the least about will get the largest extra allocation.

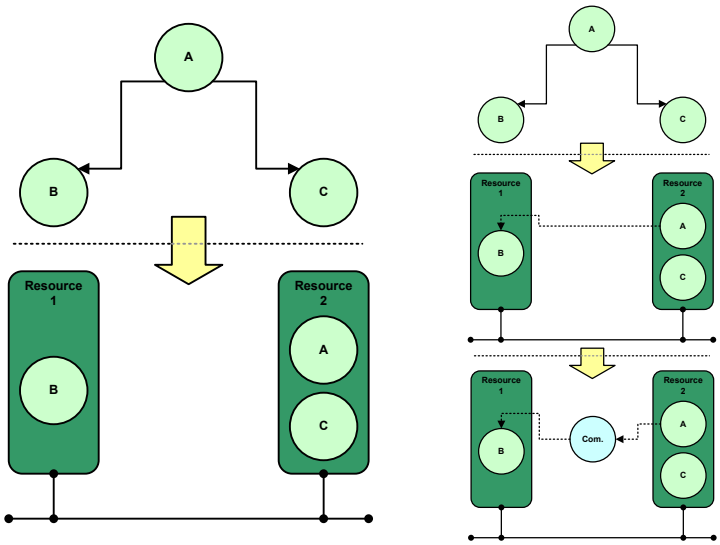


Figure 2: Allocation of three ordered tasks on two resources. After allocation we add communication penalties for tasks on different resources and have to use the bus.

2.2 Resource Allocation

The aim of the allocator is to test all possible allocations for a given order of the tasks in our system. The only limit we have for this is a maximum number of processors allowed in the system. Since it is a rather big (and unnecessary) task to test *all* possible allocations it is useful to introduce some expert knowledge. One simple rule we could apply would be that if there is a precedence constraint between two nodes (e.g. due to data flow) these two nodes should be placed in same resource. The reason for this is that it is impossible for these tasks to run in parallel, and to distribute them will only result in increased communication (i.e. not considering load balancing). If we consider a simple graph of three nodes where one node is the parent for the other two, this could be allocated on two resources in a 1-2 or a 2-1 configuration. If we name the parent A and the children B and C, we will see that the interesting allocations will be $\{\{A, B\}, \{C\}\}$ or $\{\{A, C\}, \{B\}\}$ (see Figure 2). The case with three processors is not interesting at this stage since this only will generate overhead communication as well as for the allocation $\{\{A\}, \{B, C\}\}$.

Communication

After the system is allocated we need to add communication penalties for tasks communicating between resources. For each data flow between two tasks, the number of bytes is specified. This translates into a communication delay and is finally represented as task locked on the bus resource (see Figure 2). To determine a schedule (or an order) for the bus traffic, the communication tasks inherits the order of their "parent". Since all tasks in the system first of all is ordered in a global order this will result in an unambiguous bus schedule (see Section 2.3).

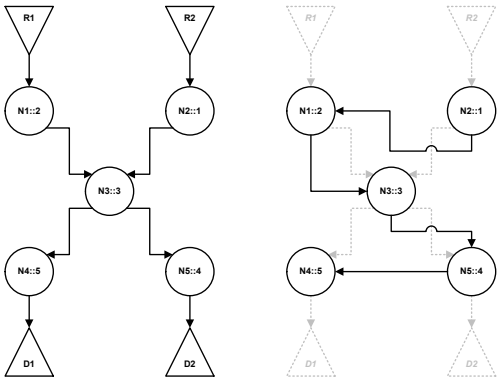


Figure 3: Based on the precedence constraints (e.g. data dependencies) an execution "path" is decided through the graph. This path shows the execution order of the tasks if they were to be executed on a single processor.

2.3 Ordering

To decide in which order to execute two parallel tasks if they end up in (or sharing) same resource due to allocation (see Section 3) we have to order all tasks. Our suggestion is to order all tasks (see Figure 3) in the same way as they would have been ordered if merely one processing element had been used (and therefore no communication is added at this stage). The example in the Figure 3 shows how the graph is put in the order N_2, N_1, N_3, N_5, N_4. To find all possible orders for a large system is also a huge task so we need some narrowing criteria. One suggested criteria is that the tasks with "latest possible finishing time" earlier than other tasks "earliest possible start time" should be put in the mentioned order. The reason for this is that if we try to do it in the other way and these two tasks end up on the same resource we will not be able to find a solution (see Section 3.1).

3 Handling Complexity

Elementary analysis shows that the suggested algorithm scales very badly why we need to isolate the reasons for this and try to find ways to reduce complexity.

Ordering

The complexity in the system originates from different sources. When we address the ordering of the task graph the number of nodes is not the main issue; but how they are connected. The more independent tasks, the higher complexity (see Figure 4). Complexity will probably increase with the number of tasks anyway, since the general case is loosely coupled. This is unfortunately the case for many regulator systems. The "fix" for this was suggested to make graphs more connected by inserting edges (or pre ordering so to say) where time limits show that other solutions would be pointless.

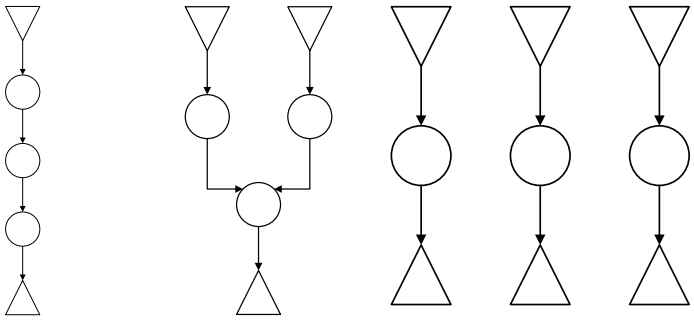


Figure 4: All three graphs have three tasks, but the first one has just one order, the second has two orders, and finally the third has six possible orders.

Allocation

When it comes to the complexity of the allocation algorithm the main factor here is the maximum number of resources allowed. For the moment, only homogenous systems are allowed, but heterogeneous systems will be supported, and this will increase the complexity of the search.

Budgeting

For the moment, budgeting is done by finding all possible paths through the system. The complexity for this search is dependent on the number of possible paths through the graph. The number of paths through the ordered allocated graph is based on the number of edges, which will increase in the step between allocation and budgeting. This is therefore important to keep the number of edges at a minimum.

3.1 Reducing Complexity

During the construction of the development tool it showed clearly that the runtime would scale very badly with the problem size (the problem is just a special case of multiprocessor scheduling). Due to this fact a number of expert methods were developed to shrink the search space and get reasonable computation loads for searches.

Virtual Edges

To describe a precedence constraint, between two tasks, not present because of a data dependence, we insert a virtual edge. In other words, virtual edges is to be inserted in the graph for tasks that clearly have a natural order due to their timing constraints (e.g. $LFT_a < EST_b \Rightarrow N_a \curvearrowright N_b$). This means that two nodes, where one node has to finish before the second one is allowed to start, are pre-ordered in that way. The pre-ordering will result in less possible total orders, without imposing any restrictions on the feasible part of the search space.

In the example setup "SimpleBaseStation" a run without pre-ordering results in 840 generated orders that are budgeted in approximately 230 seconds. If pre-ordering is turned on 504 orders were generated and budgeted in less than 160 seconds.

Clustering

One way to reduce calculation complexity is to make the graph data less complex. This could be done by clustering nodes that probably would end up in the same resource anyway. A typical case for this is a straight sequence of tasks where we do not gain anything by executing them on separate resources. As an example we clustered the "BaseStation" example and got the "SimpleBaseStation" example. In the clustering the number of tasks reduces from 17 to 8 nodes and this in turn reduces the number of orders to search through from 12 612 600 down to 540. The drawbacks of this process is that we do not get as good schedules as possible with the original graph; the optimal tightness in the "BaseStation" example is 0.4514 meanwhile the "SimpleBaseStation" has 0.5013, a deviation of approximately 10 percent. Despite this, clustering is especially suitable since it follows the typical project development cycle, where you start with large functions that are derived into finer grained modules during the iterations.

Expert Knowledge of Resource Allocation

Some of the tasks in the system have a natural place to reside, for example inputs or outputs have physical layout demands, and some tasks may have demand for large memory space or specific demands on instruction setup. To be able to make these restrictions the following operators are added:

SET Describes the set of allowed processors for this node to reside on.

GROUP Describes the set of tasks that we are allowed to share resource with.

UNGROUP Describe the set of tasks that we are not allowed sharing resources with.

Early Stopping

An interesting notation made during the tests is that we will find a solution as good as the best one among the first 10-20 thousand budgets for examples of size close to "BaseStation" (i.e. around 17 nodes, medium connected). If we assume that the tightness suddenly will make a "jump" close to the optimum (see Figure 5) we can introduce an early stopping criterion to get quicker results or indications. Examples of stopping criteria could be a predefined level of the tightness that is sufficient for our designers or a maximum numbers of iterations until we need the tightness estimate. Since iterations are quite correlated to time the second criteria could be expressed in a number of seconds we are willing to wait for the answer.

4 Preliminary Conclusion

The article presents an implemented method that fulfills the goal to evaluate proposed solutions in a number of aspects. Complexity is still an important problem to tackle; optimality is impossible to assign but how close will we be? Indications show that despite the complexity of finding optimal solutions it is still possible to achieve a practically and useful design support using the various heuristics.

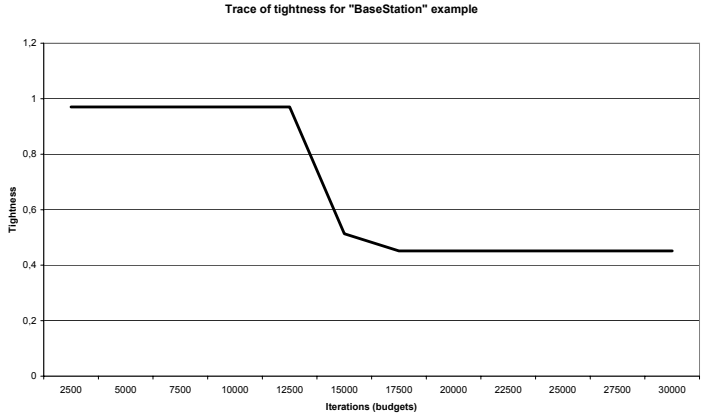


Figure 5: It would be suitable to choose an early stopping level by indicating a criterion for the tightness itself or the run time in seconds or iterations.

5 Future Work

For future evaluation we see more heuristics coming up to reduce the work load of searching. Related to this is to define new optimization criteria such as utility and cost. One very important part of the problem, barely touched, is the analysis of budget dependencies which is of importance for the design iterations (i.e. when we actually want to alter things). Since only basic homogenous architectures are supported we need to add support for heterogeneous architectures. When it comes to the restricted task graph representations we need to allow periodic tasks in the task graph, allow periods smaller than the deadline and find a way to do general task graph transformations.

References

[AH98] Peter Altenbernd and Hans Hansson. The slack method: A new method for static allocation of hard real-time tasks. *Real-Time Systems*, 15(2):103–130, 1998.

[Axe97] Jacob Axelsson. *Analysis and synthesis of hetrogenous real-time systems*. PhD thesis, Linköping University, 1997.

[Eke01] Cecilia Ekelin. Scheduling of embedded real-time systems: A constraint programming approach. Licentiate thesis, Computer Science and Engineering, december 2001.

[GiKHS95] Richard Gerber, Dong in Kang, Seongsoo Hong, and Manas Saksena. End-to-end design of real-time systems. Technical Report CS-TR-3476, 1995.

[JS97] Jan Jonsson* and Kang G. Shin**. Deadline assignment in distributed hard real-time systems with relaxed locality constraints. Technical report, 1997.

[Mey88] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[NS94] Marco Di Natale and John A. Stankovic. Dynamic end-to-end guarantees in distributed real time systems. In *IEEE Real-Time Systems Symposium*, pages 216–227. IEEE, 1994.

[Zha99] Yufeng Zhao. Derivation of local timing constraints in early design stages. Master's thesis, Chalmers University of Technology, April 1999.

Resource-based Web Server Overload Protection

Thiemo Voigt
Swedish Institute of Computer Science*
thiemo@sics.se

Per Gunningberg
Information Technology
Uppsala University
Per.Gunningberg@it.uu.se

Abstract

The main web server resources are the network interface, CPU and disk. Depending on the workload, some server resources can be overutilized while the demand on other resources is low because certain types of requests utilize one resource more than others. In this paper, we present an architecture that performs admission control based on the current server resource utilization combined with knowledge about resource consumption of requests. Our experiments demonstrate more than 40% higher throughput during overload compared to a standard server and several magnitudes lower response times.

1 Introduction

The increasing number of Internet users and innovative new services such as e-commerce are placing new demands on web servers. It is becoming essential for web servers to be highly available, have fast response times, and provide continuous service during overload.

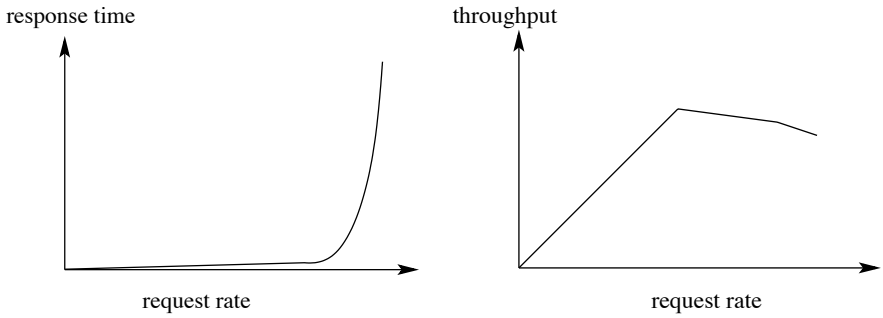


Figure 1: Impact of server overload on response time and throughput

Web servers become overloaded when one or several server resources become overutilized. As shown in Figure 1, server overload affects both the server throughput and the response time experienced by the clients. The left part of Figure 1 demonstrates how the response time increases with the server load and the right part depicts the decrease in server throughput. The response time is low as long as no server resource is overutilized. However, when the server resource bottleneck becomes overutilized, i.e., the service rate of the bottleneck cannot keep up with the arrival rate of jobs/requests, the queue length to the resource bottleneck and thus the response time theoretically increases to infinity. This is depicted by the sudden increase of the response time. One of our goals is to avoid this increase of the response time during overload.

The server throughput increases with the request rate until the request rate exceeds the capacity of the web server. At this point, the throughput decreases due to the additional time the CPU spends on processing

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

incoming connection requests that are dropped when the listen queue is full. Lower server throughput leads to loss of revenue, while long delays cause user frustration and decrease task success and efficiency.

The main server resources are the network interface, CPU and disk [10]. Any of these may become the server's bottleneck, depending on the kind of workload the server is experiencing [12]. For example, the majority of CPU load is caused by a few CGI requests [4]. The network interface typically becomes overutilized when the server concurrently transmits several large files.

While most of the proposed architectures for server overload protection throttle requests when one resource is overutilized, the main contribution of this work is an architecture that avoids server overload by preventing overutilization of specific server resources. This is achieved using adaptive inbound controls. We utilize the information found in the HTTP header of incoming requests with knowledge about the resource consumption of requests to avoid resource overutilization and server overload. We call our approach resource-based admission control.

Resource-based admission control deploys a kernel-based mechanism for overload protection and service differentiation called *HTTP header-based connection control* [22]. HTTP header-based connection control performs admission control based on application-level information such as URL, sets of URLs (identified by, for example, a common prefix), type of request (static or dynamic) and cookies. HTTP header-based control deploys token bucket policers for admission control. HTTP header-based connection control is used in conjunction with filter rules that specify application-level attributes and the parameters for the associated control mechanism, i.e. the rate and bucket size of the policer.

When the request rate reaches above a certain level, resource-based admission control alone cannot prevent overload, for example during flash crowds or Denial-of-Service (DoS) attacks. When such situations arise, we use *TCP SYN policing* [22]. This mechanism is efficient in terms of resource consumption of rejected requests because it provides "early discard". The admission of connection requests is based on network-level attributes such as IP addresses and a token bucket policer.

Another contribution of this work are mechanisms that dynamically set the rate of the token bucket policers based on the utilization of the critical resources. Since the web server workload frequently changes, for example when the popularity of documents or services changes, assigning static rates that work under these changing conditions may either lead to underutilization of the system when the rates are too low or there is a risk for overload when the rates are too high. The adaptation of the rates is done using feedback control loops. Techniques from control theory have been used successfully in server systems before [2, 17, 13, 9].

We have implemented this admission control architecture in the Linux OS and conducted experiments in a controlled network using an unmodified Apache web server. Experiments demonstrate that overload protection and adaptation of the rates works as expected. Our results show that our architecture keeps the response times lower and the throughput higher during overload compared to a standard Apache on Linux configuration.

The remainder of the paper is structured as follows: In the next section we present related work. Section 3 presents the system architecture including the controllers. Section 4 presents experiments that evaluate various aspects of our system. Section 5 discusses architectural extensions and further insights. Finally, Section 6 concludes the paper.

2 Related Work

Several research efforts have focused on overload control and service differentiation in web servers [3, 8, 15, 11]. *WebQoS* [8] is a middleware layer that provides service differentiation and admission control. Since it is deployed in middleware, it is less efficient compared to kernel-based mechanisms. Cherkasova et al. [11] present an enhanced web server that provides session-based admission control to ensure that longer sessions are completed. Their scheme is not adaptive and rejects new requests when the CPU utilization of the server exceeds a certain threshold. The focus of cluster reserves [5] is to provide performance isolation in cluster-based web servers by managing resources, in their work CPU. Their resource management and distribution strategies do not consider multiple resources.

Also other have used approaches from control theory for server systems. Abdelzaher and Lu [2] use a control loop to avoid server overload and meet individual deadlines for all served requests. They express

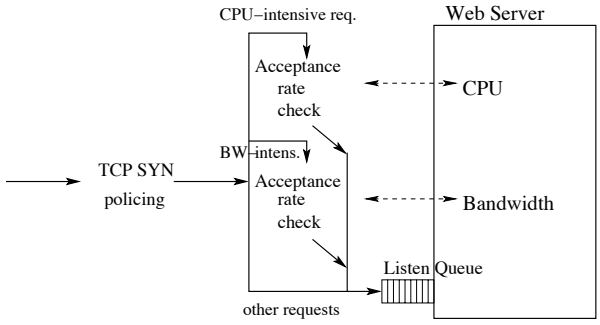


Figure 2: Admission control architecture

server utilization as a function of the served rate and the delivered bandwidth [1]. Their control task is to keep the server utilization at $ln2$ in order to guarantee that all deadlines can be met. In our approach we aim for higher utilization and throughput. Furthermore, our approach also handles dynamic requests. In another paper, Lu et al. [17] use a feedback control approach for guaranteeing relative delays in web servers. In their work, large delays are caused by HTTP 1.1 that keeps persistent connections open to await further requests in combination with the small number of concurrent connections Apache servers can handle. Parekh et al. [13] use a control-theoretic approach to regulate the maximum number of users accessing a Lotus Notes server. While a focus of these papers is to use control theory to avoid the absence of oscillations, Bhoj et al. [9] in a similar way as we, use a simple controller to ensure that the occupancy of the priority queue of a web server stays at or below a pre-specified target value. Reumann et al. [16] use a mechanism similar to TCP SYN policing to avoid server overload.

3 Architecture

The Admission Control Architecture

Figure 2 illustrates our admission control architecture. In the right part of the figure we see the web server and some of its critical resources. We use *HTTP header-based connection control* to avoid overutilization of critical resources. HTTP header-based connection control is activated when the HTTP header is received. Using this mechanism an informed control is possible which provides the ability to, for example, specify lower access rates for CGI requests than other requests that are less CPU-intensive. This is done using filter rules, e.g. checking URL, name and type. With each of the critical resources, a filter rule and a token bucket policer is associated. Token buckets have a token rate, which denotes the average number of requests accepted per second and a bucket size which denotes the maximum number of requests accepted at one time. For example, a filter rule `/cgi-bin` and an associated token bucket policer restrict the acceptance of CPU-intensive requests. On receipt of a request, the HTTP header is parsed and matched against the filter rules. If there is a match, the corresponding token bucket is checked for compliance. Compliant requests are inserted into the listen queue. We call this part of our admission control architecture resource-based admission control.

For each of the critical resources, we use a feedback control loop that adapts the token rate at which we accept requests in order to avoid overutilization of the request. Note that we do not perform resource-based admission control on all requests. Requests such as those for small static files do not put significant load on one resource. However, if requested at a sufficiently high rate, these requests can still cause server overload. Hence, admission control for these requests is needed. It would have been possible to insert a default rule and use another token bucket for these requests. Instead, we have decided to use *TCP SYN policing* and police all incoming requests. TCP SYN policing limits acceptance of new SYN packets based on compliance with a token bucket policer. TCP SYN policing enables service differentiation based on information in the TCP and IP headers of incoming connection requests (i.e, the source and destination addresses and port numbers). The main reason for deploying TCP SYN policing is its early discard capability. Using SYN policing, less resources are wasted for requests that are eventually discarded. Also for TCP SYN policing,

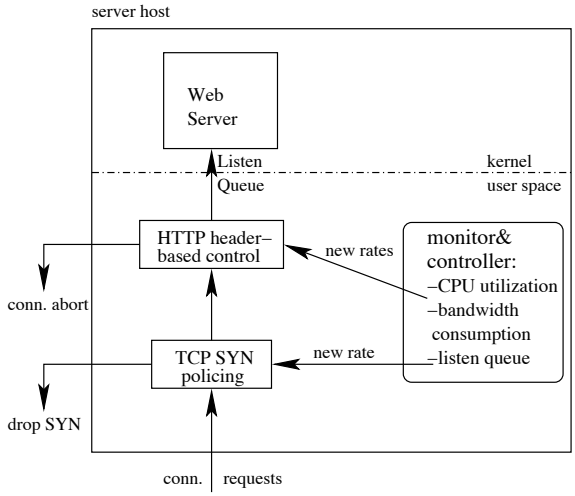


Figure 3: The control architecture

we adapt the token rate while keeping the bucket size fixed.

Both HTTP header-based connections control and TCP SYN policing are located in the kernel which avoids the context switch to user space for rejected requests. The deployment of mechanisms in the kernel has proven to be much more efficient and scalable than in the web server [22]. Note that usually connections are enqueued into the server's listen queue before the HTTP header is received. In our architecture we delay this enqueuing until the HTTP header is received, parsed and HTTP header-based connection control has been performed.

One of our major design goals for the adaptation mechanisms is to keep TCP SYN policing inactive while resource-based admission control can protect resources from being overutilized. Resource-based admission control cannot only check for URLs but also for other application-level information, such as cookies. This gives us the ability to identify ongoing sessions or premium customers while SYN policing cannot access such information.

The Control Architecture

Figure 3 shows the control architecture. The monitor measures the utilization of each critical resource and passes the values to the controller. Based on these values, the controllers adapt the rates for the admission control mechanisms. We deploy one controller for the CPU utilization and one for the bandwidth on the outgoing network interface. We call the former CPU controller and the latter bandwidth controller. Both high CPU utilization and dropped packets on the networking interface can lead to long delays and low throughput. Other resources that could be controlled are disk I/O bandwidth and memory. In addition, we use a third controller that is not responsible for a specific resource but performs admission control on all requests, including those that are not associated with a specific resource. The latter controller, called SYN controller, controls the rate of the TCP SYN policer.

Adaptation of the Rates

Different resources have different properties. Therefore, we do not use the same type of controller for each resource. Doing some initial experiments, we found that the simplest resource to control is the CPU since the CPU utilization changes directly with the rate of CPU-intensive requests the server is exposed to. This allows us to use a simple controller, namely a proportional (P) controller. The equation that computes the new rates is called the *control law*. The control law for our P-controller is:

$$rate_{\text{cgi}}(t+1) = rate_{\text{cgi}}(t) + K_{\text{P\_CPU}} * e(t) \quad (1)$$

where $e(t) = CPU\_util_{\text{ref}} - CPU\_util(t)$, i.e. the difference between the *reference* or desired CPU utilization and the current, measured CPU utilization. $rate_{\text{cgi}}(t)$ is the acceptance rate for CGI-scripts at time t. $K_{\text{P\_CPU}}$ is called the *proportional gain*. It determines how fast the system will adapt to changes in the workload. For larger $K_{\text{P\_CPU}}$ the adaptation is faster but the system is less stable and may experience oscillations [14].

The other two controllers base their control laws on the length of queues: The bandwidth controller on the length of the queue to the network interface and the SYN controller on the length of the listen queue. The significant aspect here is actually the change in the queue length since this derivative reacts faster than a proportional factor. The fast reaction is more crucial for these controllers, because the delay between the acceptance decision and the actual occurrence of high resource utilization is higher than when controlling CPU utilization. For example, the delay between accepting too many large requests and overflow of the queue to the network interface is non-negligible. One reason for this is that it takes several round-trip times until the TCP congestion window is sufficiently large to contribute to overflow of the queue to the network interface.

Therefore, we decided to use a proportional derivative (PD) controller for these two controllers. The derivative is approximated by the difference between the current queue length and the previous one, divided by the number of samples. The control law for our PD-controllers is:

$$rate(t+1) = rate(t) + K_{\text{P\_Q}} * e(t) + K_{\text{D\_Q}} * (qlen(t) - qlen(t-1)) \tag{2}$$

where $e(t) = qlen_{\text{ref}} - qlen(t)$. The division is embedded in $K_{\text{D\_Q}}$. $K_{\text{D\_Q}}$ is the *derivative gain*.

Since naive application of Equation 1 results in an increase in the acceptance rate when the measured value is below the reference value, i.e. the resource could be utilized more, we imposed some conditions on the equations above. It is not meaningful to increase the acceptance rate, when the filter rule has less hits than the specified token rate. For example, if we allow 50 CGI requests/sec, the current CPU utilization is 60%, the reference value for CPU utilization is 90% and the server has received 30 CGI requests during the last second, it does not make sense to increase the rate to more than 50 CGI requests/sec. On the contrary, if we increase the rate in such a situation, we would end up with a very high acceptance rate after a period of low server load. Hence, when the measured CPU utilization is lower than the reference value, we have decided to update the acceptance rate only when the number of hits was at least 90% of the acceptance rate during the previous sampling period.

A similar condition is imposed on the SYN and the bandwidth controller. Since both the listen queue and the queue to the network interface very often have a length of zero, their measured length is below the reference value. For the same reason as in the discussion above, if the queue length is below the reference value, we update the acceptance rate only when the length of the queue has changed.

When performing resource-based admission control, we do not police all requests, even if all requests consume resources at least some CPU. Hence, if the CPU utilization is already high, i.e. it is above the reference value, we do not want to increase the amount of work that enters the system since this might cause server overload. Thus, we increase the TCP SYN policing rate only when the CPU utilization is below the reference value.

Another important parameter is the sampling rate. For ease of implementation, we started with a sampling rate of one second. Since even slow web servers can process several hundred requests per second, a sampling rate of one second might be considered long. The question is if we do not miss important events such as the listen queue filling up. This would be the case given that the requests entering the server during one second was not limited. However, TCP SYN policing limits the number of requests entering the system. This bounds the system state changes between sampling points and allows us to use a sampling rate of one second.

4 Experimental Evaluation

The testbed for our experiments comprises a server and two traffic generators connected via a 100 Mb/sec Ethernet switch. The server host is a 600 MHz Athlon with 128 MBytes of memory running Linux 2.4. The traffic generators run on a 450 MHz Athlon and a 200 MHz Pentium Pro. The server is an unmodified Apache web server, v.1.3.9., with the default configuration.

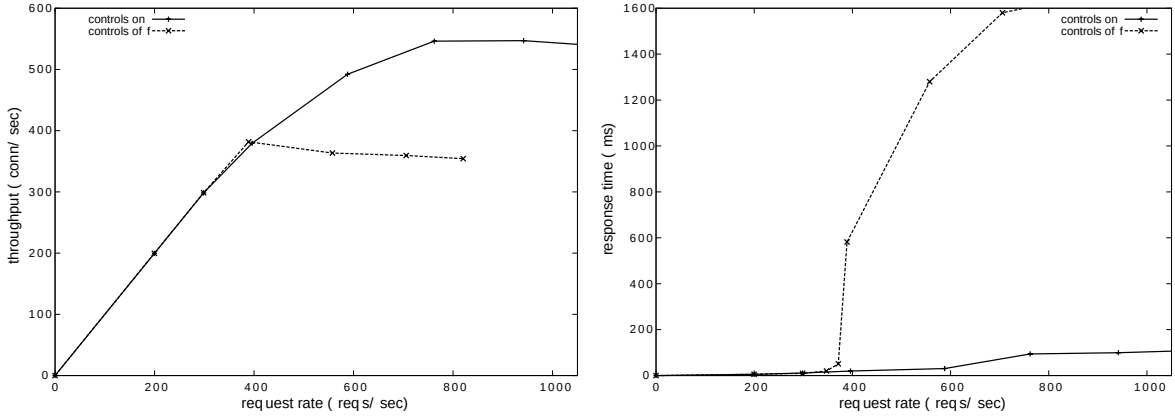


Figure 4: Comparison standard system and system with adaptive overload control

For the control algorithms we have used the following values: The reference values for the queues are set to 100 for the listen queue and 35 for the queue to the network interface. The latter has a length of 100. These values are chosen arbitrarily but they can be chosen lower without significant impact on the stability since the queue lengths are mostly zero. The reference value for CPU utilization is 90%. We chose this value since it allows us to be quite close to the maximum utilization while higher values would more often lead to 100% CPU utilization during one sampling period. The proportional gain for the CPU-controller is set to 1/5. The proportional gain for the SYN and bandwidth controller is set to 1/16, the derivative gain to 1/4. These values were obtained by experimentation. We consider these values to be specific to the server machine we are using. However, we expect them to hold for all kinds of web workloads for this server since we use a realistic workload as described in the next section. The bucket size of the token bucket used for TCP SYN policing is set to 20 unless explicitly mentioned. The token buckets for HTTP header-based controls have a bucket size of five.

Workload

For workload generation we use the sclient traffic generator [6]. Sclient in its unmodified version requests a single file. For most of our experiments we have modified sclient to request files according to a workload that is derived from the surge traffic generator [7]. The workload has three important properties: The size of the files stored on the server follows a heavy tailed distribution. The request size distribution is heavy tailed and the distribution of popularity of files follows Zipf's Law. Zipf's Law states that if the files are ordered from most popular to least popular, then the number of references to a file tends to be inversely proportional to its rank.

Determining the total number of requests for each file on the server is also done using surge. The dynamic files used in our experiments are minor modifications of standard Webstone [19] CGI-scripts. We separated the static files in two directories on the server. The files larger than 50 KBytes were put into one directory (`/islarge`), the smaller files into another directory. For the acceptance rate of both CGI-scripts and large files, minimum rates can be specified. The reason for this is that the processing of CGI-scripts or large files should not be completely prevented even under heavy load. This minimum rate is set to 10 reqs/sec in our experiments.

4.1 Supervising the CPU Utilization and the Listen Queue Length

We use two controllers in this experiment: the CPU controller that adapts the acceptance rate of CGI-scripts and the SYN controller. In the experiment, about 20% of the requests are for dynamic files. We vary the request rate across runs. The goals of the experiment are the following: First, showing that the control algorithms and in particular resource-based admission control prevent overload. Second, showing that TCP SYN policing becomes active when resource-based admission control alone cannot prevent server

req rate large workload	metric	large workload		surge workload	
		no controls	controls	no controls	controls
50 reqs/s	tput (reqs/sec)	46.8	41.5	270.7	289.2
50 reqs/s	response time (ms)	2144	80.5	1394.8	26.9
80 reqs/s	tput (reqs/sec)	55.5	45.8	205.2	285.1
80 reqs/s	response time (ms)	5400	94	3454.5	29.3

Table 1: Outgoing bandwidth

overload. Third, demonstrating that the system achieves high throughput and low response times over a broad range of request rates.

When the request rates are low, we expect that no requests should be discarded. When the request rate increases, we expect that the CPU becomes overutilized mostly due to the CPU-intensive CGI-scripts. Hence, for some medium request rates, policing of CGI-scripts is sufficient and TCP SYN policing will not be active. However, when the offered load increases beyond a certain level, the processing capacity of the server will not be able keep up with the request rate even when discarding most of the CPU-intensive requests. At that point, the listen queue will build up and, therefore, TCP SYN policing will become active.

Figure 4 illustrates the throughput and response times for different request rates. When the request rate is about 375 reqs/sec, the average response time increases and the throughput decreases when no controls are applied. Since our workload contains CPU-intensive CGI-scripts, the CPU becomes overutilized and cannot process requests with the same rate as they arrive. Hence, the listen queue builds up which contributes additionally to the increase of the response time.

Using resource-based admission control, the acceptance rate of CGI-scripts is decreased which prevents the CPU from becoming a bottleneck and hence keeps the response time low. Decreasing the acceptance rate of CGI-scripts is sufficient until the request rate is about 675 reqs/sec. At this point the CGI acceptance rate reaches the predefined minimum and cannot be decreased anymore despite the CPU utilization being greater than the reference value. As the server's processing rate is lower than the request rate, the listen queue starts building up. Due to the increase of the listen queue, the controller computes a lower TCP SYN policing rate which limits the number of accepted requests. This is shown in the left-hand graph where the throughput does not increase anymore for request rates higher than 800 reqs/sec. The right-hand graph shows that the average response time increases slightly when TCP SYN policing is active. This increase is partly caused by the additional waiting time in the listen queue.

To summarize this experiment, for low request rates, we prevent server overload using resource-based admission control that avoids over-utilization of the resource bottleneck, in this case CPU. For high request rates, when resource-based admission control is not sufficient, TCP SYN policing reduces the overall acceptance rate which keeps the response times low and the throughput high.

4.2 Supervising the Queue Length to the Network Interface

Although the workload used in the previous section contains some very large files, we noticed few packet drops on the network interface. In the experiments in this section we make the bandwidth of the outgoing interface a bottleneck by requesting a large static file of size 142 KBytes from another host. The original host still requests the surge-like workload at a rate of 300 reqs/sec. From Figure 4 we know that the server can cope with the workload from this particular host requested at this rate. The request of the large static file will cause overutilization of the interface and a proportional drop of packets to the original host.

Without admission control, we expect that packet drops on the outgoing interface will cause lower throughput and in particular higher average response times by causing TCP to back off due to the dropped packets. Therefore, we insert a rule that controls the rate at which large files are accepted. Large files are identified by a common prefix (`/islarge`). The aim of the experiment is to show that by adapting the rate with that requests for large files are accepted, we can avoid packets drops on the outgoing interface.

Requests to the large file are generated with a rate of 50 and 80 reqs/sec. The results are depicted in Table 1. As expected the response times for both workloads become very high when no controls are applied. In our experiments, we observed that the length of the queue to the interface was always around the maximum value which indicates a lot of packet drops. By discarding a fraction of the requests for

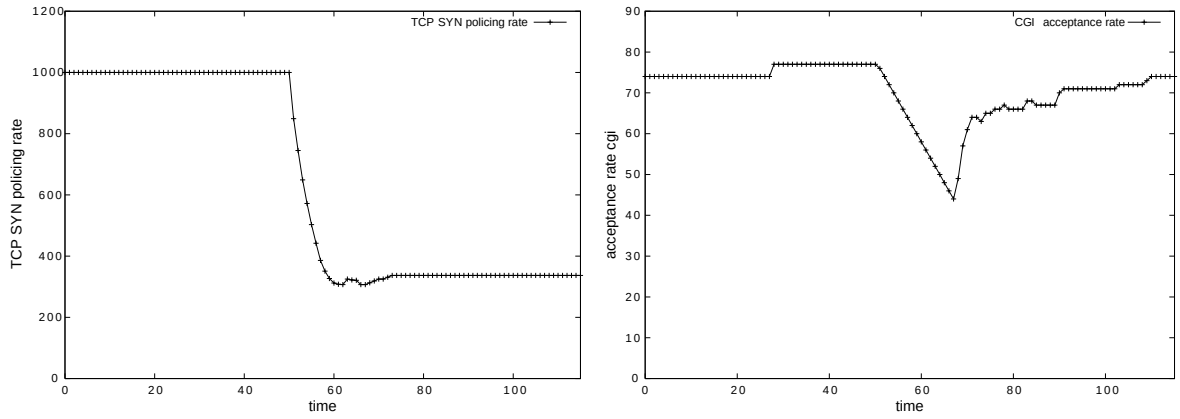


Figure 5: Adaptation under high load (left SYN policing rate, right CGI acceptance rate)

large files our controls keep the response time low by avoiding drops in the queue to the network interface. Although the throughput for the large workload is higher when no controls are applied, the sum of the throughput for both workloads is higher using the controls. When controls are applied the sum of the throughput for both workloads is the same for both request rates.

4.3 Sudden High Load

The objective of this experiment is to expose the server to a sudden high load and study the behaviour of the control algorithms. Such a load exposure could occur during a flash crowd or a DoS attack. We start with a relatively low request rate of 300 reqs/sec. After 50 seconds we increase the offered load to 850 reqs/sec and sustain this high request rate for 20 seconds before we decrease it to 300 reqs/sec again. We set the initial TCP SYN policing rate to 1000 reqs/sec.

From Figure 5, we see that the TCP SYN policing rate decreases very quickly when the request rate is increased at time 50. This rapid decrease is caused by both parts of the control algorithms in Equation 2. Since the length of the listen queue increases quickly, the contribution of the derivative part is high. Since the absolute length of the listen queue at that time is higher than the reference value, the contribution of the proportional part of Equation 2 is high as well. The TCP SYN policing rate does not increase to 1000 again after the period of high load. However, we can see that around time 70, the policing rate increases to around 340 which is sufficiently high so that no requests need to be discarded by the SYN policer when the request rate is 300 reqs/sec. The SYN policing rate settles at higher rates for higher request rates after the period of high load.

As expected, the CGI acceptance rate does not decrease as fast. Since $K_{\text{P\_CPU}}$ has a value of 1/5, the CGI acceptance rate is decreased at most with two per sampling point during the period of high load. Figure 5 also shows that the CGI-acceptance rate is restored fast after the period of high load. At a request rate of 300 reqs/sec, the CPU utilization is between 70 and 80%. Thus, the absolute difference to the reference value is larger than during the period of high load which enables faster increase than decrease of the CGI acceptance rate. At time 30 in the right-hand graph, we can see the CGI acceptance rate jump from 74 to 77. The reason for this jump is that during the last sampling period, the number of hits for the corresponding filter rule was above 90%, while it was otherwise below 90% until time 50.

5 Discussion

The sclient program generates web server requests at a constant rate. The resulting requests also arrive at a constant rate to the web server. We have modified the sclient program to generate requests following a Poisson distribution with a given mean. In another experiment, we have shown that our adaptation mechanisms should be able to cope with bursty traffic provided we make a sensible choice of the bucket size [21].

Although our architecture is implemented as a kernel module, it could also be deployed in user space or in a middleware layer. Since our basic architecture is implemented as a kernel module, we have decided to put the control loops in the kernel module as well. An advantage of having the control mechanisms in the kernel is that they are actually executed at the correct sampling rate.

The proposed solution of grouping the objects according to resource demand in the web server's directory tree, is not intuitive and awkward for the system administrator. We assume that this process can be automated using scripts.

The interaction between the different control loops might cause oscillations. Fortunately, requests to large static files do not consume much CPU while CPU-intensive requests usually do not consume much network bandwidth. Hence, we assume that the control loops for these resources will not experience any significant interaction effects. The adaptation the TCP SYN policing rate affects the number of accepted CPU-intensive and bandwidth-intensive requests, which may cause interactions between the control loops. By increasing rates quite conservatively we hope to avoid this effect. Furthermore, in none of our experiments we have seen an indication that such an interaction might actually occur. The main reason for this is that TCP SYN policing becomes active when the acceptance rate of CPU-intensive requests is very low and most of the CPU-intensive requests are discarded.

Extensions of the Architecture

In our adaptive architecture accepted requests can be processed quickly since server resources are not overutilized and the listen queue is short. This leads to low response times. Therefore, the major goal of service differentiation in our architecture is to provide high throughput to premium customers. This can be done by splitting the token buckets used for admission control into logical partitions [18]. Each logical partition corresponds to one service class with larger partitions for more important service classes.

Our current implementation is targeted towards single node servers or the back-end servers in a web server cluster. We believe that the architecture can easily be extended to LAN-based web server clusters and enhance sophisticated request distribution schemes such as Harvard Array of Clustered Computers HACC [23] and Locality-aware Request Distribution (LARD) [20]. In both LARD and HACC, the front-end distributes requests based on locality of reference to improve cache hit rates and thus increase performance. In the extended architecture, the front-end performs resource-based admission control. The back-end servers monitor the utilization of each critical resource and propagate the values to the front-end. Based on these values, the front-end updates the rates for the token bucket based policers using the algorithms. After the original distribution scheme has selected the node that is to handle the request, compliance with the corresponding token bucket ensures that critical resources on the back-ends are not overutilized. This way, we consider the utilization of individual resources as a distribution criteria which neither HACC nor LARD do. HACC explicitly combines these performance metrics into a single load indicator. There are two potential problems: First, the need to propagate the values from the back-ends to the front-end causes some additional delay. If the evaluation of the system shows that this is indeed a problem, we should be able to overcome it by setting more conservative reference values or by increasing the sampling rate. Second, there is a potential scalability problem caused by the need for $n * c$ token buckets on the front-end, where n is the number of back-ends and c the number of critical resources. However, we believe that this is not a significant problem since a token bucket can be implemented by reading the clock (which in kernel space is equal to reading a global variable) and performing some arithmetical operations.

6 Conclusions

In this paper, we have presented an adaptive server overload protection architecture for web servers. Using the application-level information in the HTTP header of the requests combined with knowledge about resource consumption of resource-intensive requests, the system adapts the rates at which requests are accepted. The architecture combines the use of such resource-based admission control with TCP SYN policing. TCP SYN policing first comes into play when the load on the server is very high since it wastes less resources when rejecting requests. Our experiments have shown that the acceptance rates are adapted as expected. Our system sustains high throughput and low response times even during overload.

Acknowledgements

This work builds on the architecture that has been developed at IBM TJ Watson together with Renu Tewari, Ashish Mehra and Douglas Freimuth [22]. Without them, this work would not have been possible. Thanks to Andy Bavier, Ian Marsh, Arnold Pears and Bengt Ahlgren for valuable comments on earlier drafts of this paper. The authors also want to thank Martin Sanfridson and Jakob Carlström for discussions on the control algorithms.

References

[1] T. Abdelzaher and N. Bhatti. Web server qos management by adaptive content delivery. In *Int. Workshop on Quality of Service*, June 1999.

[2] T. Abdelzaher and C. Lu. Modeling and performance control of internet servers. In *IEEE Conference on Decision and Control*, December 2000.

[3] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated levels of service in web content hosting. In *Proc. of Internet Server Performance Workshop*, March 1999.

[4] M. Arlitt and C. Williamson. Web server workload characterization: The search for invariants. In *Proc. of ACM Sigmetrics*, April 1996.

[5] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Proc. of ACM SIGMETRICS*, June 2000.

[6] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Proc. of USITS*, December 1997.

[7] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proc. of SIGMETRICS*, 1998.

[8] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, September 1999.

[9] P. Bhoj, S. Ramanathan, and S. Singhal. Web2k: Bringing qos to web servers. Technical report, HP, May 2000.

[10] E. Casalicchio and M. Colajanni. A client-aware dispatching algorithm for web clusters providing multiple services. In *Proc. of 10th Int'l World Wide Web Conference*, May 2001.

[11] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for improving the performance of an overloaded web server. Technical report, HP, 1999.

[12] L. Eggert and J. Heidemann. Application-level differentiated services for web servers. *World Wide Web Journal*, 3(2):133–142, September 1999.

[13] S. Parekh *et al.* Using control theory to achieve service level objectives in performance management. In *IFIP/IEEE International Symposium on Integrated Network Management*, May 2001.

[14] T. Glad and L. Ljung. *Reglerteknik: Grundläggande teori (in Swedish).* Studentlitteratur, 1989.

[15] R. Iyer, V. Tewari, and K. Kant. Overload control mechansims for web servers. In *Performance and QoS of Next Generation Networks*, November 2000.

[16] H. Jamjoom and J. Reumann. Qguard: Protecting internet servers from overload. Technical report, University of Michigan, 2000.

[17] C. Lu, T. Abdelzaher, J. Stankovic, and S. Son. A feedback control approach for guaranteeing relative delays in web servers. In *Real-Time Technology and Application Symposium*, June 2001.

[18] A. Mehra, R. Tewari, and D. Kandlur. Design considerations for rate control of aggregated tcp connections. In *Proc. of NOSSDAV*, June 1999.

[19] Mindcraft. Webstone. http://www.mindcraft.com.

[20] V. Pai, M. Aron, G. Banga, M. Svendsen, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, USA, October 1998.

[21] T. Voigt and P. Gunningberg. Handling multiple bottlenecks in web servers using adaptive inbound controls. In *Int. Workshop on Protocols for High-Speed Networks*, Berlin, Germany, April 2002.

[22] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proc. of Usenix Annual Technical Conference*, June 2001.

[23] X. Zhang, M. Barrientos, J. Chen, and M. Seltzer. HACC: An architecture for cluster-based web servers. In *Third Usenix Windows NT Symposium*, pages 155–164, Seattle, WA, July 1999.