Lia-Anca HÂNGAN

CONTRIBUTIONS TO THE DEVELOPMENT OF PARALLEL AND DISTRIBUTED REAL-TIME SYSTEMS

UTPRESS
Cluj-Napoca, 2020
ISBN 978-606-737-444-5

Lia-Anca HÂNGAN

CONTRIBUTIONS TO THE DEVELOPMENT OF PARALLEL AND DISTRIBUTED REAL-TIME SYSTEMS

UTPRESS
Cluj - Napoca, 2020
ISBN 978-606-737-444-5

Editura U.T.PRESS
Str. Observatorului nr. 34
C.P. 42, O.P. 2, 400775 Cluj-Napoca
Tel.:0264-401.999
e-mail: utpress@biblio.utcluj.ro
http://biblioteca.utcluj.ro/editura

Director: ing. Călin Câmpean

ISBN 978-606-737-444-5
Bun de tipar: 02.04.2020

Preface

Goal

Real-time systems have a wide application domain that includes industrial process control, avionics, military and automotive. For such applications, satisfying timing and performance requirements is critical and sometimes exceeding deadlines can cause catastrophic results. Therefore, the designer must guarantee the fulfillment of time requirements before system start. In many cases, system designers have been using special purpose hardware and networking infrastructures to guarantee a deterministic behavior of real-time systems. However, there has been an increasing interest for using general-purpose platforms instead of dedicated ones and as well for implementing real-time systems on parallel and distributed architectures.

In this context, a set of research problems needed to be analyzed and addressed. This book presents theoretical models and solutions for multiprocessor scheduling of real-time tasks, for the transmission predictability of real-time data on general-purpose networks and for the simulation and performance evaluation of real-time systems.

Multiprocessor scheduling of real-time tasks is not a new subject, but research in this direction recently intensified because of the large-scale use of parallel and distributed systems, including the multicore processors. However, current models do not cope with the complexity introduced by the current real-time applications requirements. On the other hand, the schedulability problem needs more investigation since preliminary theoretical results showed that in some cases multiprocessor systems does not behave better than a uniprocessor system. Moreover, many scheduling solutions have poor performances.

For real-time distributed systems, one of the main concerns is to ensure transmission predictability of real-time data and to find mechanisms that make possible the coexistence of real-time and best-effort data transmission in the same network. There is need for solutions that guarantee a deterministic data transmission using state-of-the-art QoS mechanisms implemented on general-purpose communication infrastructures (e.g. switched Ethernet and IP networks).

Last but not least, the real-time community has recently acknowledged the lack of standard methodologies and tools for the performance evaluation of new research results in the domain of real-time systems. There is need for common metrics and common evaluation methods to be able to make a relevant comparison between similar research results.

This book presents theoretical and pragmatic solutions for multiprocessor real-time systems, extracted from the author's PhD thesis. The contributions presented here spread over several sub-domains that cover the most important problems of parallel and distributed real-time systems: system models, communication issues in distributed systems, scheduling on multiprocessors, performance evaluation of scheduling techniques and of parallel programming languages, tools for simulation and performance evaluation.

Audience

This book is addressed to computer science students that are in their senior year or pursuing a Master's degree. This book assumes that the reader has reasonable knowledge in the areas of parallel and distributed systems, as well as in computer networks.

Outline

This book is organized as follows.

Chapter 1 is a brief introduction to the area of real-time systems that includes an analysis of the status of research and identifies a set of open problems.

Chapter 2 contains a presentation of the most representative theoretical models used in real-time systems research. Starting from these models, directions that need attention from the research point of view are identified. Finally, a general model for parallel and distributed real-time systems is proposed.

In **Chapter 3** describes a model for the availability of open network segments that allow the transmission of real-time traffic at the same time with non-real-time traffic and a method for network bandwidth estimation in the context of a reservation-based communication architecture.

Chapter 4 presents multiprocessor scheduling techniques for real-time transactions.

Chapters 5 and 6 present performance evaluation tools, methods and results in the area of multiprocessor scheduling. First, the design and implementation of a simulation environment that is used as support for the evaluation of multiprocessor real-time systems theoretical models and algorithms is described. Then, the methodology for the analysis of real-time multiprocessor systems based on simulation is presented. Finally, experiments are shown, to validate the simulation-based methodology.

Chapter 1. Real-time systems

1. Introduction

In most cases, it is expected that a computer system will produce a correct result in an acceptable time frame, approximately proportional to the volume of computation required. If you read an online journal, the accepted time to complete loading of the website is less than 20 seconds. But if you want to simulate the biochemical interactions between molecules, a period of up to several days can be accepted to produce results, because the volume of data and computation is very high. In previous cases, the users are satisfied if the system produces a correct result, with no errors and, even if the web page takes longer than 20 seconds to load, only the user's patience is affected. The correctness and the usefulness of the result are not affected.

However, there are situations when it is very important that the system response is generated without errors and within a specified time interval. There are critical cases in which exceeding the time limit causes negative (sometimes catastrophic) unwanted effects, such as most of the industrial control systems, avionics, military systems and other similar systems. In other cases, the response time has to be situated inside a specified interval for the system to be useable. Multimedia streaming or teleconferencing applications have such time constraints. For these applications, the Quality of Service (QoS) the user perceives is influenced by the end-to-end packet delay and by the standard deviation of packet delay (jitter). QoS must be high for these applications to be usable.

A system is *real-time* if its correct functioning depends on meeting certain time constraints. More specifically, a real-time computing system can be defined as a system that when subjected to external stimuli, generates a response within a limited time interval. H. Kopetz gives the following definition in [1]:

"A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced."

Real-time systems usually respond to events that take place in the environment in which they operate and they often have direct effects on it. Therefore, their *response time*, defined as the time between the occurrence of an event and the corresponding response, should be correlated to the timing characteristics of the environment. For example, in the case of an industrial process, if the time between the occurrence of an alarm due to high pressure in a boiler and the time a valve opens resulting in lowering the pressure must be under 10 seconds, the response time of the system that controls this process must be adapted to this timing requirement, otherwise the boiler can explode. Teleconferencing applications are another example. In this case, even if extreme values are admitted, it is very important that average data transmission delays be undetectable to human hearing and vision; moreover, the transmission delay jitter must be low. If these timing requirements are not met, the users will not understand what their interlocutors are saying, and will not be able to see them.

Real-time systems can be classified from different points of view. For example, some classifications can be done by taking into consideration only the characteristics of the applications that are executed on the system. Other classifications can be made by taking into account platform characteristics, resource availability or characteristics of the operational environment. Some of these classifications are presented below.

Hard and soft systems

A very important classification of real-time systems is made considering only the timing constraints of real-time applications. According to this classification, real-time systems are either *hard* or *soft*.

A real-time application is composed of multiple *tasks* (units of work that are scheduled and executed by the system). Each task has a *deadline*, defined as the instant in time by which its execution has to be completed. The characteristics taken into consideration when choosing a category for an application (hard or soft) are linked to the quantification of deadline misses and the effects generated by deadline misses [2]:

- The total number of deadline misses
- The usefulness of the results in case of deadline misses
- The effects of deadline misses on the users and the environment

A real-time application is hard if:

- The total number of deadline misses is zero or close to zero
- The usefulness of the results is zero or close to zero in the case of deadline misses
- Deadline misses have serious, sometimes catastrophic effects on users and the environment (e.g. a boiler explosion; a robot crushes into an obstacle)

For hard real-time applications, deadline misses are not accepted. Moreover, the developer has to guarantee that deadlines are always met by using some proved analytical methods or validation algorithms.

A real-time system is hard if it contains mainly hard real-time applications. To guarantee that in a hard system all timing constraints are satisfied, applications and the platform (hardware and software) on which they execute must have deterministic behaviors. In this way, the validation of the entire system can be done before system start, or, for dynamic systems, during execution.

A real-time application is soft if:

- Occasional deadline misses are accepted. The deadline can even be specified as a probabilistic parameter.
- The results are still useful if the application experiences a small number of deadline misses, but their usefulness decreases with the increase of deadline misses. The relation between the number of deadline misses and the usefulness of the results is specific for each application.

- Deadline misses cause only the degradation of service quality

The avoidance of occasional deadline misses is of little importance for soft real-time applications because they do not cause serious effects. It is more important that these applications have good average response times.

A soft real-time system contains mainly soft real-time applications. These systems do not need such rigorous validation as in the case of hard systems. Even if exact quantification of deadline misses is not required, the validation of the average case timing behavior is desirable.

Event-triggered and time-triggered systems

Another classification is based on the type of mechanism that triggers real-time applications activities, such as the execution of a job or the transmission of a message [1].

If the activities are triggered by the occurrence of an event other than the clock tick, the real-time system is *event-triggered*. An intelligent sensor that reports a significant change in the value of a measured environment parameter (e.g. temperature) is event-triggered. Event-triggered activities are usually modeled as asynchronous or sporadic tasks that require dynamic execution schedulers.

When activities are initiated in predetermined points in time, the real-time system is *time-triggered*. Time-triggered activities are usually modeled as periodic tasks and execution scheduling is predetermined. These systems are more predictable than event-triggered systems, but can overlook the occurrence of some events. Let’s presume that the intelligent sensor mentioned before is part of a time-triggered system. To obtain information about the state of the environment, the system will poll the sensor from time to time. If the state change occurs between two consecutive polls, the sensor has to save the observed state until the system asks for an update, or the observation will be lost.

Resource-adequate and resource-inadequate systems

Real-time systems, in which there are enough computing resources available to handle all presumed scenarios, are *resource-adequate* [1]. Even if it is too expensive to have systems that have enough resources to handle all possible situations, many hard and safety-critical systems have designs based on the resource-adequacy principle.

On the other hand, in many cases, it is hard or even impossible to provide enough resources to cover all possible scenarios. These systems are consequently built on the *resource-inadequacy* principle. Embedded and mobile systems have limited resources that have to be managed in order to provide the best performance. Recently, extensive work has been done in the area of accommodating real-time applications on shared platforms. In this case, applications have to dynamically adapt to the fluctuating resource availability in order to maintain timing constraints.

Single-node and distributed systems

Depending on the characteristics of the underlying platform, a real-time system can be *single-node* or *distributed* [3]. A single-node real-time system consists of one computer and its I/O devices. The single-node system receives an input, performs some processing locally and then, it generates the result.

A distributed real-time system consists of multiple computers (nodes) that communicate through a computer network. Each node of the distributed system may have its own I/O devices and can process inputs received from another node. Applications have distributed components. The validation of timing constraints must take into consideration end-to-end execution, which consist of processing time (measured in the nodes) and data transmission time (measured in the communication network).

Some of the advantages of implementing real-time applications on distributed environments are:

- Reliability
- Resource sharing
- Scalability

Reliability is usually obtained trough replication. If a node fails, other nodes can resume its activity. If nodes have limited computing resources, resource sharing can help overcome this problem. Moreover, a distributed system can be easily extended by adding more processing nodes.

Uniprocessor and multiprocessor systems

Another classification criterion can be the number of CPUs of the underlying platform. Consequently, real-time systems can be *uniprocessor* or *multiprocessor*. Uniprocessor systems contain only one CPU. Multiprocessor systems contain more than one CPU.

Even if many real-time systems contain more than one CPU, a great number of research models and analysis techniques are focused on platforms with only one CPU. However, recent studies tend to address multiprocessor environments and their problems, such as task assignment, task synchronization and the heterogeneity of the CPUs.

Multiprocessor systems and distributed systems resemble in the sense that each system contains more than one CPU [2]. The difference between the two is that multiprocessor systems are tightly coupled and CPUs may have a shared memory, and, in contrast, distributed systems are loosely coupled and each CPU has its own, private memory. As consequence of these characteristics:

- It is much easier to keep global status and workload information up to date on multiprocessor systems.
- Scheduling and synchronization algorithms that are suitable for multiprocessor systems may not work as well for distributed systems.

Many real-time applications, which are developed for uniprocessor systems, are single threaded. If these applications are to be deployed on multiprocessor systems, the easiest solution is to assign the application to only one CPU. But to fully take advantage of multiprocessor systems, we need to find ways to parallelize real-time applications.

Open and closed systems

In [4], the authors classify real-time systems as *closed* and *open* systems based on the characteristics of their operating environment.

Closed real-time systems have the following characteristics:

- The set of real-time applications that coexist in the system is known and does not change in time
- Applications are developed and validated together
- Detailed timing attributes of all real-time applications on each processor are known
- The schedulability of the system in all predicted scenarios is determined beforehand

In contrast with closed systems, *open real-time systems* have the following characteristics:

- The set of real-time applications that coexist in the system may change in time; at run-time, the user can request the start of another application, which was not part of the initial application set
- Applications may be developed and validated independently
- If an application is validated to meet its timing constraints when executing in isolation, the system that accepts its access to the shared platform at runtime, has to guarantee that the application's timing constraints are met on the shared platform
- The system may accept applications with different timing requirements, even non-real-time applications

The design and implementation of open real-time system has lately received much attention from the real-time research community. Recent work has been done in the direction of integrating these systems with complex platforms (many shared resources, not just a single processor) [5].

After analyzing the classifications presented above, the main aspects that differentiate them can be highlighted:

- Classification criteria
- System layer from which the criteria is selected
- The model components influenced by the selected criteria

Table 1 shows these aspects.

Table 1. Comparison between different classifications of real-time systems

Classification	***Layer***	***Criteria***	***Influence***
Closed **Open**	Environment	Environment dynamics	Workload model Resource model Scheduling

Hard **Soft**	Application	Timing constraints (deadline)	Workload model Scheduling
Event-triggered **Time-triggered**	Application	Activity triggering mechanism (implementation)	Workload model
Resource-adequate **Resource-inadequate**	Platform	Resource availability	Resource model
Single-node **Distributed**	Platform	Number of computing nodes	Resource model Scheduling
Uniprocessor **Multiprocessor**	Platform	Number of CPUs	Workload model Resource model Scheduling

2. Status of real-time systems research

Real-time systems have a well defined application domain that includes industrial process control, avionics, military and signal processing applications. The environment in which these systems function is controlled, closed and predictable. For these applications, timing and performance requirements have to be guaranteed before system start. Parameters whose values can't be known or predicted before system start can generate great issues. In this context, to guarantee a deterministic behavior of the real-time system, designers use special purpose and many times expensive hardware, networking infrastructures, protocols and operating systems.

The recent evolution of the real-time domain creates new trends in research and development (see Table 2). These trends are partially generated due to the expansion of the application domain to online multimedia applications (audio and video streaming, teleconferencing, Internet phone), real-time online transactions, real-time web (social networks, real-time search), mobile applications, home automation, sensor networks, and more. On the other hand, the constantly increasing performances and low costs of common use off-the-shelf hardware and networking infrastructures encouraged researchers to find solutions for their use in real-time systems.

Table 2. Current trends in real-time systems research

Level	***Classic approach***	***Current trends***
Hardware and communication platforms	To insure determinism, special purpose real-time hardware, special purpose networks and real-time communication protocols are used.	Use commercial off-the-shelf hardware and networking technologies for real-time systems. E.g. multicore, switched Ethernet, IP networks, etc.

Operating systems and resource management	Real-time operating systems, real-time resource management and execution scheduling.	Manage real-time applications in common use operating systems (E.g. Linux). Develop virtualization technologies for real-time systems.
Applications	Relatively small application domain: industrial process control, avionics, military, signal processing.	Diversification of application domain: multimedia (audio and video streaming, teleconferencing, Internet phone), online real-time transactions, real-time web (real-time search, social networks), mobile applications, home automation, sensor networks, etc.
Environment	The environment is closed and controlled. In restrictive cases, real-time applications do not share resources with other applications. Resources are shared only between real-time applications.	The environment is open and dynamic. Real-time applications can share resources with non-real-time applications.

Fig.1 shows some current points of interest for real-time systems research that emerge as the effect of the two factors we identified:

- Real-time systems application domain diversification
- Evolution of common hardware and communication infrastructures

An important research direction has the objective of implementing real-time systems on multiprocessors (parallel and distributed infrastructures). Multiprocessor scheduling of real-time tasks is not a new research subject [6]. However, because it raises complex problems [7], researchers are still looking for improved solutions. Moreover, research in this direction intensified because of the large-scale production of multicore microprocessors. There are several topics of interest in this direction, such as:

- Multiprocessor scheduling [8][9],
- Multiprocessor/Multicore processor timing analysis [10][11],
- Development of technologies and tools for analyzing the worst case execution behavior on multiprocessors/multicores [12],
- Parallel or multicore-aware programming languages for real-time applications [13]

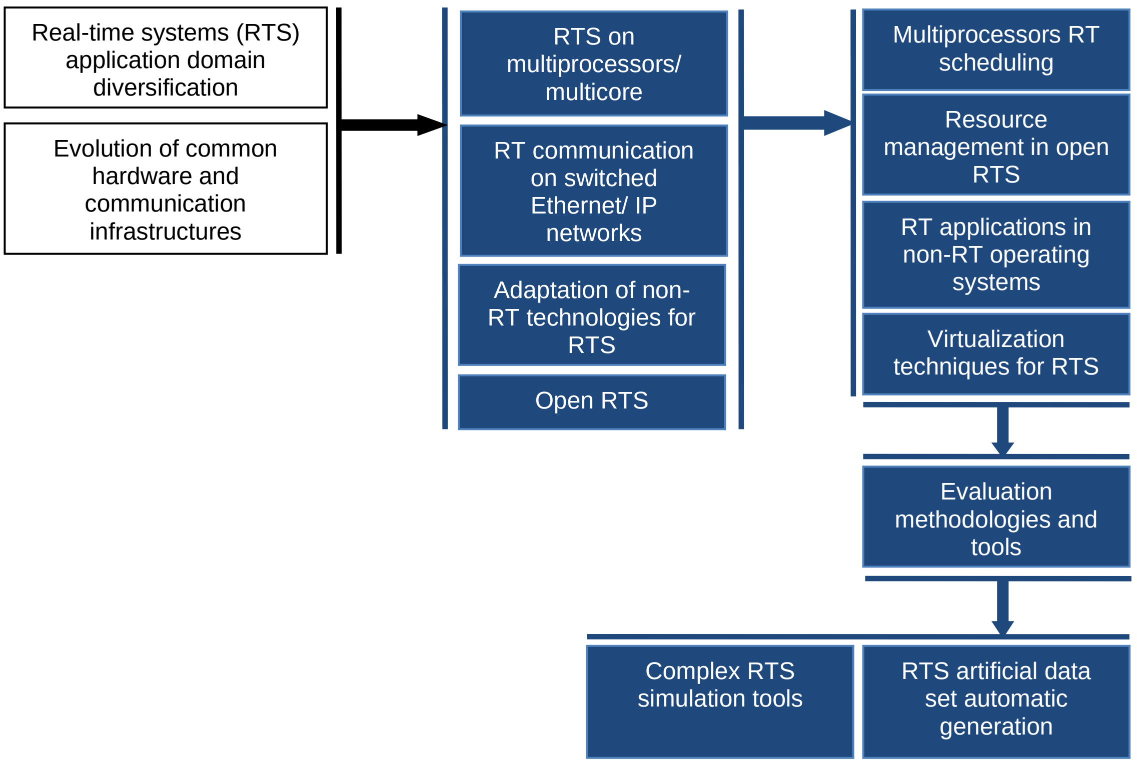

Real-time systems (RTS) application domain diversification
Evolution of common hardware and communication infrastructures
RTS on multiprocessors/ multicore
RT communication on switched Ethernet/ IP networks
Adaptation of non-RT technologies for RTS
Open RTS
Multiprocessors RT scheduling
Resource management in open RTS
RT applications in non-RT operating systems
Virtualization techniques for RTS
Evaluation methodologies and tools
Complex RTS simulation tools
RTS artificial data set automatic generation

Figure 1. Some current points of interest for real-time systems research

Many research groups investigate major problems of multiprocessor real-time systems such as scheduling, timing analysis and worst case execution behavior analysis. Even though there are many results in these directions, there are still important issues that need to be solved. There are several optimal scheduling algorithms for multiprocessors [23][24], but they don't have any practical relevance due to the prohibitively high overheads introduced by migrations and context switches. The schedulability problem (proving that tasks meet their deadlines) needs more investigation since theoretical results show that multiprocessors can in some cases handle task utilizations not much larger than uniprocessors [6][25]. Moreover, schedulability tests are very restrictive [26] and there are many situations when tasks can't be included in the "schedulable" or "not schedulable" groups. A third important problem that, to our knowledge, doesn't have a solution until now is the identification of the worst case execution behavior on multiprocessors [7]. In our opinion, multiprocessor real-time systems research problems need solutions that are more pragmatic and can be used beyond the research lab.

Increased performances of common use communication infrastructures (switched Ethernet and IP networks) and the development of QoS mechanisms were the cause for new research initiatives. Researchers aim to find solutions to use these communication infrastructures and QoS mechanisms in real-time distributed systems. The main concerns of this research direction are:

- To ensure transmission predictability of real-time data [14]
- To find mechanisms that make possible the coexistence of real-time and best-effort data flows in the same network [15]

For many modern real-time systems the computational load and resource availability cannot be determined or predicted a priori, because of the open and dynamic environment in which they operate. Furthermore, resource availability and application resource requests can fluctuate in an unpredictable manner. Embedded and mobile devices, present in many real-time applications, have limited resources that need to be managed efficiently to maximize performance. Many current real-time systems must manage concurrent activities with different restrictions of time and QoS and that require access to shared resources. In these cases, the main challenge is to find models that can deal with the diversity of these activities. The situation becomes even more complex when real-time applications share resources with applications that do not have real-time restrictions. Starting from these issues, the operation of real-time applications in open environments and sharing resources between applications with different time restrictions, some important research topics have emerged:

- Resource management in open real-time systems (which contain applications with different time restrictions) [4] [5] [16]
- Handling real-time tasks in common use operating systems [17]
- Virtualization techniques for real-time systems [18]
- The adaptation of non-real-time technologies such as those used in SOA, Web and Cloud Computing, for real-time systems [19] [20] [21]

Many classic models and solutions, which are used in the development of real-time systems, cannot handle the recent evolution in this domain. A review of classic theories is needed in order to adapt them, if possible, to meet current needs. On the other hand, new models, algorithms and technologies, which deal with the complexity of current real-time systems, are needed.

Performance evaluation of a new real-time model, algorithm or technique implies the comparison with similar existing results according to a certain method and based on a set of metrics. Many research results in the area of real-time systems are validated through complex mathematical analysis by computing the system’s worst case response time. However, with the growing complexity of real-time systems, the worst case behavior is sometimes impossible to identify. Consequently, these methods are becoming out-of-date and there is need for new evaluation methods and tools. The real-time community has recently acknowledged [22] that there are three “major obstacles” in comparing published work in the area of real-time systems:

- The lack of standard methodologies and software tools for performance evaluation
- The lack of public/open software used for performance evaluation
- Research results are not made available as downloadable data files

The real-time systems research community needs a set of open tools and data sets that would be used for research results evaluation. There is need for common metrics and common evaluation methods to be able to make a relevant comparison between similar research results.

References

[1] Hermann Kopetz, Real-Time Systems, Design Principles for Distributed Embedded Applications, Springer, 1997

[2] Jane W.S. Liu, *Real-Time Systems*, Prentice Hall, 2000

[3] Claudiu Ciprian Farcas, “Towards Portable Real-Time Software Components”, PhD Thesis 2006
[4] Z. Deng, J. W.-S. Liu, “Scheduling Real-Time Applications in an Open Environment”, IEEE, 1997
[5] Insik Shin, Arvind Easwaran, Insup Lee, “*Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors*”, Euromicro Conference on Real-Time Systems, 2008
[6] S. K. Dhall, C. L. Liu, *“On a Real-Time Scheduling Problem”*, Operations Research, vol. 26, number 1, pp. 127-140, 1978.
[7] Robert I. Davis, Alan Burns, “*A Survey of Hard Real-Time Scheduling Algorithms and Schedulability Analysis Techniques for Multiprocessor Systems*”, 2009
[8] Nan Guan, Martin Stigge, Wang Yi and Ge Yu, “Fixed-Priority Multiprocessor Scheduling with Liu & Layland’s Utilization Bound”, RTAS10, 2010
[9] Sanjoy Baruah, Joel Goossens, ”Deadline Monotonic Scheduling on Uniform Multiprocessors”, 2008
[10] Nan Guan, Martin Stigge, Wang Yi and Ge Yu, ”*Cache-Aware Scheduling and Analysis for Multicores*”, EMSOFT09, 2009
[11] Mingsong Lv, Nan Guan, Wang Yi, Ge Yu, ” Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software”, RTSS10, 2010
[12] Mingsong Lv, Nan Guan, Wang Yi, and Ge Yu, ”McAiT - Combining Abstract Interpretation and Model Checking for Timing Analysis of Multicore Real-Time Software”, 2011
[13] Jeopard FP7 Project, ”JEOPARD Whitepaper”, 2008
[14] T. Skeie, S. Johannessen, O. Holmeide, ”*Timeliness of real-time IP communication in switched industrial Ethernet networks*”, IEEE Transactions on Industrial Informatics, 2006, 2, pp. 25-39
[15] M. Wijnants, W. Lamotte, “*Managing client bandwidth in the presence of both real-time and non real-time network traffic*”, 3rd International Conference on Communication Systems Software and Middleware and Workshops, 2008, pp. 442-450
[16] A. Mok, X. Feng, D. Chen, “*Resource partition for real-time systems*”, Real-Time Technology and Applications Symposium, 2001, pp. 75–84
[17] Hennadiy Leontyev, James H. Anderson, “*A Hierarchical Multiprocessor Bandwidth Reservation Scheme with Timing Guarantees*”, Euromicro Conference on Real-Time Systems, 2008
[18] Enrico Bini, Marco Bertogna, Sanjoy Baruah, “*Virtual Multiprocessor Platforms: Specification and Use*”, IEEE Real-Time Systems Symposium, 2009
[19] Steffen Pruter, Guido Moritz, Elmar Zeeb, Ralf Salomon, Dirk Timmermann, Frank Golatowski, “*Applicability ofWeb Service Technologies to Reach Real Time Capabilities*”, 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC), 2008
[20] Shuo Liu, Gang Quan, Shangping Ren, “*On-line Scheduling of Real-time Services for Cloud Computing*”, IEEE 6th World Congress on Services, 2010
[21] Marshall Kirkpatrick, “*Explaining the Real-Time Web in 100 Words or Less*”, http://www.readwriteweb.com, 2009
[22] G. Lipari, T. Cucinotta, “Preface JSA Waters 2011”, Journal of Systems Architecture 59 (2013) 296
[23] S.K. Baruah, N. Cohen, G. Plaxton, D. Varvel, “*Proportionate progress: A notion of fairness in resource allocation*”, Algorithmica 15, no. 6, pp. 600–625, 1996
[24] H. Cho, B. Ravindran, E.D. Jensen, *“An Optimal Real-Time Scheduling Algorithm for Multiprocessors”*. In Proceedings of the Real-Time Systems Symposium pp. 1001-110, 2006.
[25] Enrico Bini, Marco Bertogna, Sanjoy Baruah, “*Virtual Multiprocessor Platforms: Specification and Use*”, IEEE Real-Time Systems Symposium, 2009
[26] M. Bertogna, S. Baruah, “*Tests for global EDF schedulability analysis*”, Journal of Systems Architecture, no. 57, pp. 487–497, 2011

CHAPTER 2. REAL-TIME SYSTEM MODELS 1

Chapter 2. Real-time system models

1. Introduction

The main difference between a real-time application and a general-purpose one is that the developer must demonstrate not only the logical correctness of the application but also the fulfillment of a set of predefined time restrictions (e.g. deadlines, periodicity, etc.). In many real-time systems, exceeding the time limits is critical and it may cause accidents (including human injuries) or significant material loses. Therefore, a so-called "best effort" approach typical for non-real-time applications is not applicable for those with time restrictions.

Evaluating, controlling and analytically demonstrating the time-behavior of a system is usually not a trivial task. Any analytical (mathematical) approach requires an abstract model of the real system, which necessarily introduces a number of abstractions and simplifying assumptions. Without such a model, the attempt to demonstrate the timely behavior of a reasonably complex system is impossible [1]. Like in any other science, modeling a physical system is an important step in the process of understanding and controlling its behavior.

In real-time systems, models allow us to predict different time parameters of a system (e.g. execution or response time, delays, transmission times, periodicity, etc.) without the need for experimental measurements and tests. In many cases, experimental validation of a real system is not possible or it is not relevant. For example, if the application is controlling an industrial process, a longer delay in the system's response to a critical event may cause accidents. On the other hand, experiments may not reveal the "worst-case response time".

A *model* is a simplified, abstract representation of a complex reality. The model introduces a set of concepts and relations (e.g. equations) between them that may be used for analytical evaluation of the system's behavior [1]. The model introduces, as well, a set of assumptions that have the role of simplifying its analysis.

The following are some simplifying assumptions that are encountered in real-time systems theory:

- *Discrete time*. Time variables (e.g. execution time, repetition period) are discrete.
- *Predictable execution*. The executable instances of a task will have the same execution time.
- *Preemptivity*. The executable instances of a task are preemptable if they can be interrupted at any time during execution or non-preemptable if not.
- *Uni/Multi processor*. The workload will be executed by a single or by multiple processing resources.
- *Sequential or parallel execution*. The executable instances of a task will be executed by at most one processor at the same time. Otherwise, it will be executed by more than one processor, in parallel.
- *Task independence*. Tasks don't influence each other's execution. Data or precedence dependencies are not considered.

- *Communication time*. Communication time between application components and hence communication-related delays are not considered in the model.
- Other restrictions (e.g. mutual exclusion, synchronization).

As presented in [4], the model of a real-time system may be divided into three sub-models:

- The *workload model* – describes the applications that are executed on the system's platform
- The *resource or platform model* – describes the system resources available to the applications
- A set of *scheduling and resource management algorithms* – describe how the applications use the available resources

In this chapter, we make a presentation of the most representative theoretical models used in real-time systems research. Starting from the presented models, we discuss the status of real-time theoretical analysis and identify some directions that, in our opinion, need attention from the research point of view. Moreover, we present our representation approach, for each model component.

2. Workload models

A real-time application is composed of *periodic* and *non-periodic tasks*. The *real-time task* is a concept used for modeling a software program that has an execution requirement and a deadline. A particular execution instance of a task is called a *job*; a job is in fact program executed on a given processing platform. A periodic task will issue a sequence of jobs that will be released at equal distances in time one after the other. A non-periodic task will issue a single job or a sequence of jobs, but not at equal distances in time. Tasks execution may be correlated or not. If tasks are correlated with a precedence relation, then the set of correlated tasks form a *transaction*. A transaction may have a linear (no branches), tree or graph structure. In our approach, a real-time application may contain a set of transactions or a set of independent tasks.

There are many approaches to real-time workload modeling. The main categories of models we intend to investigate are the following:

- Sequential tasks
- Parallel tasks
- Transactions

Some of the most representative approaches in these categories will be presented below.

2.1. Sequential tasks

Many real-time applications are represented as sets of sequential tasks. A task is *sequential* if each job can execute on exactly one processor at a time. The task models presented as follows assume that job execution is sequential.

2.1.1. Periodic task models

The workload model that, for a long time, received the most attention from researchers is the *periodic task model*, also known as the *Liu and Layland model* [5]. Most of the later workload models are generalizations of this model.

The real-time workload is modeled as a set of periodic tasks. A *periodic task* is a sequence of jobs with identical parameters that occur at constant intervals over time. Periodic tasks have the following important parameters:

- *Period p* – the length of the interval between release times of consecutive jobs. In [5] these intervals are constant.
- *Execution time e* – the maximum execution time of all the jobs in the task.
- *Phase ϕ*– the release time of the first job in the task. In many cases, it is assumed that the phase is equal to zero for all tasks. A task set is called *synchronous* if the phase is equal for all its tasks; otherwise, the task set is *asynchronous*.
- *Deadline D* – the relative deadline.

The most representative parameter of a real-time task is the *deadline*. The *deadline* of a task is defined as the time (relative to job release time) before each job in the task has to finish its execution. There are three main deadline types used in real-time task models:

- *Implicit* – deadline is equal to the task's repetition period.
- *Constrained* – deadline is less than the task's repetition period.
- *Arbitrary* – deadline can have any value, independent of the task's repetition period.

The task deadline type influences the task set feasibility analysis method in terms of complexity (the most complex analysis is for arbitrary deadlines).

The job mostly inherits the parameters of the task; some of its parameters are computed based on the task's parameters. The parameters of a job (*J*) are the following [4]:

- *Release time (r)* – the instant in time at which the job is available for execution. The release time can be fixed, meaning that the exact time instant of the release is known, or can be jittered, meaning that only the time interval $[r_{min}, r_{max}]$, in which the release can occur, is known.
- *Absolute deadline (d)* – the instant in time by which the execution of the job must be completed ($d=r+D$).
- *Relative deadline* (*D)* – if a job is released at *r*, job execution must be completed *D* units of time after *r*. The term *deadline* is usually used when referring to the relative deadline.
- *Execution time (e)* – the amount of time required to complete the execution of the job, when it executes without being interrupted and all resources needed are available. The execution time of a job may be variable, meaning that the exact value can be any, in the interval $[e_{min}, e_{max}]$.

- *Preemptivity* – a job is *preemptable* if its execution can be interrupted at any time, to allow other jobs to be executed. Later, the execution is resumed from the point of suspension. If a job is *nonpreemptable*, its execution can't be interrupted.

The *utilization factor* (u_i) of a periodic task is the fraction of processor time spent in the execution of that task [5] ($u_i=e_i/p_i$). The *total utilization* (U) of a task set is the sum of all task utilizations ($U=\sum u_i$).

In Liu and Layland's model, a periodic task is represented as a tuple $\tau(e, p)$ which contains the worst case (maximum) execution time and the period of the task. It is also assumed that:

- Tasks are preemptable
- Tasks have implicit deadlines
- Tasks are *independent*, meaning that they do not have any precedence constraints or other type of dependencies (e.g. data).

The periodic task model has some strong restrictions:

- Tasks have constant period, even though in real-world cases the inter-release time may be variable.
- It is assumed that tasks have constant execution time, which rarely happens. In fact, tasks usually contain conditional and repetitive sequences of code, which cause variable execution time in consecutive releases.
- It is assumed that tasks are independent. However, it is highly probable that tasks will depend on the availability of data or other operating system on hardware resources that influence the execution behavior of task sets.

Even though these restrictions reduce the model's expressiveness, the less complex analysis methods encouraged many researchers to use it in their work. Therefore, the periodic task model was intensively studied for many years and it is still used for the analysis and validation of uniprocessor and multiprocessor scheduling algorithms.

2.1.2. Sporadic and aperiodic task models

Aperiodic and sporadic tasks are used to model responses of the real-time system to external events that may occur at any time. *Aperiodic tasks* have random interarrival times, and in many cases, their deadlines are either soft, or they do not have deadlines. In the case of *sporadic tasks* [6], job releases will not occur periodically but there is a minimum interval between any two consecutive job releases. Sporadic tasks may have hard deadlines.

The *multiframe task model* is presented by Mok in [7]. This model generalizes the periodic task model, by relaxing the conditions on task execution time and task period. In the multiframe model, it is assumed that:

- Jobs in a task may have variable execution time, but the variation follows a certain recurrent pattern

- Tasks are sporadic, which means that they have a minimum occurrence period
- Task deadline is implicit
- Tasks are independent and preemptable

A multiframe real-time task is represented as a tuple $\tau(E, p)$, where E is a finite list of n $(n \geq 1)$ execution times $(e^0, e^1, \ldots, e^{n-1})$ and p is the minimum separation time between two consecutive frames (jobs). The execution time of the i^{th} frame of the task is $e^{i \bmod n}$. The deadline of each frame is equal to p (implicit deadline). In the particular case when $n=1$ and p is a constant separation time between frames, the multiframe task model reduces to a periodic task model.

The *generalized multiframe task model (GMF)* presented in [8] extends Mok's multiframe model. In the GMF model, deadline and separation time assumptions change:

- Deadlines are explicit, meaning that they differ from the minimum separation time
- Deadlines are not equal for all frames
- Frames have different minimum separation time

A GMF task is represented as a tuple $\tau(E, D, P)$, where E, D and P are finite lists of n $(n \geq 1)$ elements. E contains execution times $(e^0, e^1, \ldots, e^{n-1})$, D contains deadlines $(D^0, D^1, \ldots, D^{n-1})$ and P contains minimum separation times $(p^0, p^1, \ldots, p^{n-1})$. According to this model, for the i^{th} frame of task T, the execution time is equal to $e^{i \bmod n}$, the deadline is $D^{i \bmod n}$ and the next frame will have the release time $r_{i+1} \geq r_i + p^{i \bmod n}$.

The most general model (at this time) which uses multiframe tasks to describe real-time workload is the *non-cyclic GMF model* presented in [9]. The term "non-cyclic" comes from the fact that there is no recurrent pattern for task activations, like in the previous multiframe task models (Mok's multiframe model and GMF model). A non-cyclic GMF task consisting of n frames is defined by a sequence of tuples $((e^0, D^0, p^0), (e^1, D^1, p^1), \ldots, (e^{n-1}, D^{n-1}, p^{n-1}))$. Any frame can be activated at the end of the minimum separation time of the previous frame. The non-cyclic GMF model is used for event-triggered systems, for which the task activation pattern cannot be predicted.

A different approach, of using graphs in real-time workload modeling is presented in [10]. *The recurring real-time task (RRT) model* starts from the idea that, for many event-triggered real-time applications, timing requirements may change at runtime as result of conditional branches that depend on the system state or on environment parameters. In these cases, periodic or multiframe models can be used only to describe the worst case behavior of task systems. But it may be sometimes very difficult to predict the worst case behavior. The RRT model allows the representation of conditional real-time code. A RRT is represented as a *task graph*. A task graph is a directed acyclic graph (DAG) in which *vertices* are subtasks and *edges* are possible flows of control.

A real-time task τ is formally represented by a task graph $G(\tau)$ and a period $P(\tau)$. $G(\tau)$ has a unique *source* vertex and a unique *sink* vertex. Each vertex u represents a subtask and is labeled with a pair $(e(u), d(u))$, where $e(u)$ is the execution time and $d(u)$ is the relative deadline of the subtask. Each time a subtask u is triggered, a job is generated with $(e(u), d(u))$ timing requirements. Each edge (u, v) is labeled with *p(u, v)*, which represents the minimum separation time between the triggering of *u* and v (*v* can be triggered as early as *p(u,v)* time units after *u* was triggered). The source vertex of the task graph can be initially triggered at any time. Then, the source vertex can be triggered only after the sink vertex was triggered and minimum *P(τ)* time units after its previous triggering. Fig. 1 is an example of RRT.

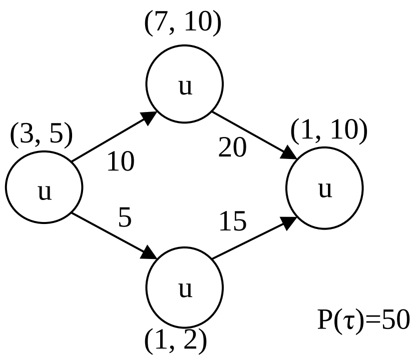


Figure 1. An example of recurring real-time task [10]

An *event* is a tuple *(τ, t, u)* that denotes that a subtask *u* of task *τ* is triggered at time *t*. The sequence of events of task *τ*, *σ*(*τ*)=[$(\tau, t_1, u_1), (\tau, t_2, u_2),...$] may be infinite and is said to be *legal* if the following conditions are satisfied:

- If u_i is not the sink vertex of *G(τ)*, then (u_i, u_{i+1}) is an edge, and $t_{i+1}-t_i \geq p(u_i,u_{i+1})$
- If u_i is the sink vertex of *G(τ)*, then u_{i+1} is the source node of G(τ), and if exists an event (τ, t_j, u_j), j<i in the event sequence for which $u_i=u_j$, then $t_{i+1}-t_i \geq P(\tau)$

A system of recurring real-time tasks Γ consists of independent recurring real-time tasks that are preemptively scheduled on a single processor. A *legal event sequence for a task system* σ(Γ) is obtained by merging one legal event sequence for each task in Γ.

Starting from the example in Fig. 1, σ(Γ)=[$(\tau, 0, u_0), (\tau, 11, u_1), (\tau, 35, u_3), (\tau, 53, u_0), (\tau, 58, u_2), (\tau, 80, u_3), (\tau, 103, u_0), (\tau, 110, u_2)\ (\tau, 130, u_3)$] is an example of a legal sequence of events.

The non-cyclic recurring real-time model presented in [11] generalizes the initial RRT model by removing the task period *P(τ)* and allowing multiple sink vertices, in this way being able to represent non-cyclic behavior. The difference from the RRT model is that:

- A real-time task *τ* is formally represented only by a task graph *G(τ)*
- The task graph has a unique source vertex and one or more sink vertices
- Sink vertices are labeled with a value *p(u, src(τ))*, which denotes the minimum separation time between the triggering of the sink vertex and the triggering of the source vertex. If the sink vertex was triggered at time t_i, than the source vertex will be triggered at $t_{i+1} \geq t_i + p(u, src(\tau))$.

The digraph real-time task model (DRT) presented in [12] is a generalization of the non-cyclic RRT model. The workload is defined by a system of N independent tasks. Each task is represented by a *directed graph* $G(\tau)$. The set of vertices represent the types of jobs that can be released by the task. Each vertex is labeled with a pair $(e(u), d(u))$, where $e(u)$ is the execution time and $d(u)$ is the relative deadline of the job. The edges represent the order in which jobs can be released. Each edge is labeled with $p(u, v)$, the minimum separation time between jobs. In contrast with the RRT models, DRT model allows cycles in $G(\tau)$ and does not require a source vertex (any vertex can be the first to be released).

Multiframe and graph-based models, in their most general form, are able to represent independent sporadic tasks that contain conditional and repetitive code. They are able to model the cases when subsequent executions of the same code have different execution times, with the condition that the execution behavior has to generate a pattern.

The task automata model [13] is a very expressive way of representing sequential non-periodic real-time workload. Task automata can describe complex tasks, which have the following characteristics:

- Non-deterministic generation according to timing constraints
- Interval execution times
- Completion times may influence the releases of other task instances (there may be dependencies between tasks)

A task is defined as a tuple $P(e_{min}, e_{max}, D)$, where P is the name of the task, e_{min} is the best case execution time, e_{max} is the worst case execution time and D is the relative deadline.

A *task automaton* is obtained by extending a *timed automaton* [14] with real-time tasks. It is assumed that $Act=\{a, b, c, \ldots\}$ is a set of actions, $C=\{x_1, x_2, \ldots\}$ is a set of clocks and $B(C)$ is a set of clock constraints. A task automaton over actions Act, clocks C, and tasks P is a tuple $(N, l_0, E, I, M, x_{done})$ where:

- N is a finite set of locations
- $l_0 \in N$ is the initial location
- $E \subseteq N \times B(C) \times Act \times 2^C \times N$ is the set of edges
- I is a function assigning each location with a clock constraint
- M is a partial function that assigns a task to a location
- x_{done} is the clock which is reset every time a task finishes

The *state of a task automaton* is a tuple (l, u, q), where l is the location, u is the clock value and q is the task queue, which contains pairs of remaining computation times and deadlines for all released task instances that did not finish their execution. There are two types of transitions. *Delay transitions* correspond to the execution of the running task. *Discrete transitions* correspond to the release of new task instances.

Task automata model can express precedence dependencies between tasks as well as between task instances (jobs), non-periodic job release and non-deterministic task behavior. However, this model is not used very often in real-time systems analysis due to the complexity of its scheduling analysis methods that, in some cases (e.g. task preemption, dependencies between jobs), do not converge to a result [13].

2.2. Parallel tasks

Most research work that involves real-time multiprocessor systems use sequential task models (e.g. models in which it is assumed that jobs execute on a single processor at a time instant). Recent work [15] [16] [17] introduces models of parallel tasks as a response to the problem of modeling execution parallelism (e.g. multithreading and other parallel programming constructions with finer granularity than threads). A job generated by a parallel task can execute at the same time instant on more than one processor. In [17] the authors classify parallel recurrent (periodic or sporadic) tasks as:

- *Rigid* – the number of processors assigned to the jobs is fixed (does not change throughout its execution) and is specified a priori and externally to the scheduler.
- *Moldable* – the number of processors assigned to the jobs is determined by the scheduler (for each job) and does not change throughout the job's execution.
- *Malleable* – the number of processors assigned to the jobs can be changed by the scheduler during the execution.

A model of *sporadic malleable tasks* is presented in [15]. A task system, which executes on m identical processors, consists of n tasks. A task is represented as a 3-tuple $\tau(e, p, \Gamma)$, where e is the worst case execution time, p is the minimum separation time (the relative deadline is equal to p) and $\Gamma = (\gamma_1, \gamma_2, \dots, \gamma_m)$ is an m-tuple of execution ratios that satisfies $\gamma_1 < \gamma_2 < \cdots < \gamma_m$. A job that executes t time units on j processors completes $\gamma_j * t$ units of execution. The utilization ratio of the task is $U=e/p$. To complete its execution, a task requires $k+1$ processors simultaneously, where:

$$k = 0 \; if \; U < \gamma_1; else, k = max_{k-1}^{m}\{k|U < \gamma_k\}$$

Both models presented in [16] and in [17] define periodic parallel tasks as 4-tuples $\tau(v, e, d, p)$, where v is the number of required processors, e is the worst case execution time, p is the period and d is the relative deadline. It is assumed that $d \leq p$. However, the two groups of authors do not seem to agree on the definitions of rigid and moldable tasks. In [16] the authors consider moldable tasks (even if they do not explain the mechanism the scheduler uses for determining the number of required processors for a task) and in [17] the authors consider rigid tasks.

Recent work presented in [18] and [19] considers the *fork-join model* for parallel real-time tasks. This model assumes that a task is composed of sequence of sequential and parallel execution regions. The task always starts and ends with a sequential execution region. A fork-join task is represented as a tuple $((C_i^1, P_i^2, \dots, P_i^{s-1}, C_i^s), m_i, T_i)$, where C_i^s is the worst case execution time of a sequential execution region, P_i^s is the worst case execution time of each thread in the parallel execution region, m_i is the number of parallel threads in each parallel region and T_i is the repetition period of the task.

2.3. Real-time transactions

Distributed applications are composed of several software components that run on different computers and that interact to achieve a common goal. The main problems in modeling distributed real-time applications are:

- Representing the interaction between tasks (e.g. network communication, execution dependencies)
- The specification of time constraints (e.g. deadlines) in the distributed context

The network communication between two processing tasks is represented as a *message*. The message is a special type of task that has a transmission time and a deadline. The release time of the message depends on the execution of the emitting task. A precedence relation is thus created between the message and the emitting and receiving tasks. The message's repetition period is closely related to that of its predecessor task. In many cases, there are no differences between the representations of tasks and messages in a distributed real-time system's model.

There are different modeling approaches for distributed real-time systems. From among these, the most representative is the real-time transaction model. Many other models are its variants, even if the terminology is different.

Transactions [35] can be described, in their most general form, as directed acyclic graphs in which nodes are tasks or messages and the edges are precedence relations between tasks and messages. In a simplified form, the transaction is a chain of tasks. Each task has at most one predecessor and at most one successor. Chain transactions are frequently used to validate scheduling methods because they contain the smallest number of dependencies between tasks.

Transactions can be periodic or sporadic and have *end-to-end deadlines*, meaning that the transaction's deadline is equal to the deadline of its last task. Intermediate tasks do not have explicit deadlines. However, in some models, intermediate tasks have their own deadlines [36]. A real-time periodic transaction has the following defining elements:

- *Task graph* (G) – a DAG that describes the dependencies between tasks and messages and determines the execution order. Nodes are tasks or messages, while graph edges represent precedence dependencies.
- *Period* (T) – the repetition period of a transaction.
- *Deadline* (D) – the deadline of a transaction, relative to its release time.

In many cases, messages are treated as tasks that are handled by nodes representing network segments, and their transmission time is treated as task execution time.

Another approach similar to chain transactions is the *end-to-end tasks model* [4]. An end-to-end task is a chain of atomic actions that execute in a pipeline fashion, each action on a different processor. If τ is a task that contains n actions, action $i+1$ will be ready to execute only if action i has finished its execution. $V=\{V_1, \ldots, V_n\}$ is the *visit sequence* for task τ, meaning that V_i is the processor on which action i executes. The *end-to-end release time* of a task is the release time of its first action. The *end-to-end* *deadline* of a task is the deadline of its last action. As long as end-to-end timing constraints (deadline) are satisfied, it is not important when other actions finish their execution.

Other approaches for expressing task dependencies are presented in [20] and in [21]. In [20], dependent periodic and sporadic tasks are modeled as *port-based objects*, meaning that a task has a set of input ports and a set of output ports through which it communicates (sends or receives data) with other tasks. A task also has a set of possible behaviors. Different events combined with some input data can trigger different behaviors and some output data. A set of tasks is modeled as a *directed graph*, in which tasks are nodes and the edges represent data flows (links) from output ports to input ports of tasks. There can be *synchronous* and *asynchronous links*. A synchronous message triggers the execution of the receiver task. An asynchronous message is buffered until the receiver task is activated. There are a few constraints on this model:

- Synchronous links can't create cycles
- Each task can have at most one synchronous link

The work in [21] describes a model for dependent periodic tasks. The authors make the distinction between simple and extended precedence. A *simple precedence* is defined as a relation between two tasks $\tau_i \rightarrow \tau_j$, which means that task τ_i, must execute before task τ_j. The graph that represents the precedence constraints of a task set has to be acyclic. An *extended precedence* is defined as a set of precedences between task instances (jobs) $\tau_i \xrightarrow{M_{i,j}} \tau_j$, where $\forall(n,n') \in M_{i,j}, \tau_i[n] \rightarrow \tau_j[n']$.

Few approaches for modeling task dependencies are different from the real-time transaction model. The previously presented task automata model can express precedence dependencies between tasks (e.g. a task can be released only if another task finishes its execution).

2.4. The Sequential-Parallel-Distributed Real-Time workload model

We propose a workload model that allows the representation of a variety of real-time applications that include sequential, parallel and distributed (with network communication). We call it Sequential-Parallel-Distributed Real-Time (SPD-RT) model.

We define a task model that is general enough to represent:

- *Periodic and a-periodic behavior*
- *Parallel execution*

In the most simplified case, our task model is able to represent sequential independent tasks that have a repetition period.

The expressiveness of the proposed task model, in the context of the previously presented task models, is depicted in Fig. 2. Because of the far too complex feasibility analysis [12], we chose not to cover the most complex aspects of sequential execution. However, the strength of our model is that it covers at the same time sequential and parallel execution requirements.

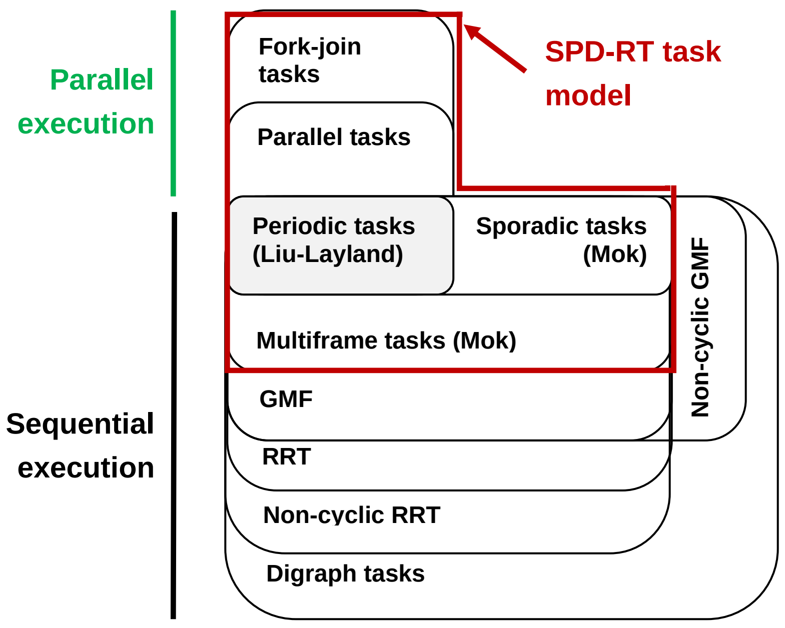


Figure 2. The expresiveness of the proposed task model in the context of the reviewed task models

We consider that tasks may have execution constraints introduced by the execution of other tasks. We represent *precedence dependencies* between tasks through transactions. Our transactions take the form of DAGs, as presented in Section 2.3.

A *task* is represented by the following parameters:

$$\tau = (\phi, E, d, pr, P, C, A)$$

Where:

- *Phase* (ϕ) – the time when the task releases its first job.
- *Execution* (*E*) – a sequence of execution segments, that describe the task's execution requirements:
  $E_\tau = \{(e_i, \pi_i) | i > 0\}$, *e* – segment execution time, π - degree of parallelism
- *Deadline* (*d*) – time limit relative to the release of the job.
- *Priority* (*pr*).
- *Consumed and produced events* (P, C). *Events* are used to model precedence dependencies.
- *CPU affinity* (*A*) – list of processors on which the task can be executed.

A *periodic task* has one more parameter, the *repetition period* (*T*). In the case of periodic tasks, phases can be equal to zero or can be set according to user's requirements. Tasks are *preemptive*.

To represent parallel execution using our model, we describe the task's execution requirements through a sequence of *execution segments*. Applications usually have portions that are inherently sequential, and other portions that can be parallelized. An execution segment may be a sequential portion of a task or it can be executed in parallel on a number of processors. The execution segment has two parameters: the *execution time* and the *degree of parallelism* (number of threads that may run in parallel).

Between two execution segments, there is a *synchronization point*, such that an execution segment will begin only if the previous execution segment is completed. At the beginning of each execution segment, a number of parallel threads equal to the degree of parallelism will be created. Each thread will have the execution time of the segment to which it belongs. The constraint imposed by this model is that all threads generated by an execution segment have the same execution time. If a task contains only one execution segment with a parallelism degree equal to 1, its jobs will be single threaded (sequential).

To be able to represent the *dependencies* between tasks, we propose a mechanism based on the producer-consumer model. A task can produce *events*, which are consumed by other tasks, hence creating an execution dependency between producer and consumer. Events produced by executing jobs are stored in the *global event queue*. Producing an event is a non-blocking action. On the other hand, consumer jobs extract the expected events from the global queue. If the expected event is not in the queue (it has not been produced), the job's execution will be blocked until the expected event is produced.

Tasks that are part of transactions do not have predefined periods and deadlines. However, their period is equal with the transaction's period and the deadline of the last task is equal to the transaction's deadline. The deadlines of intermediate tasks have to be determined using some specific method.

We will use the proposed task and transaction models in the next chapters that deal with problems related to real-time scheduling and real-time systems analysis.

3. Platform models

The platform model describes the system resources available to the applications. Applications require processors in order to execute, networks for communication, and sometimes they need to use other resources during execution, such as storage devices, memory, locks and others.

In [4] two types of system resources are distinguished:

- *Active resources*, which are usually called processors (e.g. CPUs, transmission links, disks)
- *Passive resources* (e.g. shared data objects, buffers, locks)

The model of active resources is of greater importance because it describes the availability of processors, for the execution of the workload. For the rest of this document, the terms *resource* and *processor* will be considered synonyms.

3.1. Classification

3.1.1. Uniprocessor platforms

Platforms that provide only one processor for the execution of applications are called *uniprocessor platforms*. Uniprocessor platforms are intensively studied because the system validation mechanisms are much easier to use than for multiprocessor platforms. Many real world real-time applications are executed on one dedicated uniprocessor (e.g. control applications). In cases in which the system is composed of more concurrent applications, each is executed on a dedicated processor in isolation (e.g. avionics software).

3.1.2. Multiprocessor platforms

Multiprocessor platforms contain more than one processor, $\pi=\{\pi_1,\pi_2,\ldots,\pi_m\}$ where $m>1$. Multiprocessor platforms are further classified by [22] in three categories:

- *Homogeneous*
- *Uniform*
- *Heterogeneous*

In *homogeneous multiprocessor platforms*, the processors are identical, meaning that the execution rate of all tasks is the same on all processors.

A *uniform multiprocessor platform* contains processors characterized by their speed $\pi = \{s_1, s_2, \ldots, s_m\}$. The execution rate of a task depends only on the speed of the processor.

In *heterogeneous multiprocessor platforms*, the processors are different. Task execution rates depend on both the task and the processor. A task execution rate $s_{i,j}\geq 0$ is associated with each pair (T_i,π_j), where T_i is a task.

Distributed platforms can be modeled as a multiprocessor: both nodes and network links between nodes are represented as processors. For simplicity, the multiprocessor can be homogeneous, but the most accurate representation would be the heterogeneous multiprocessor.

3.2. Virtual resource models

Many hard real-time systems require the execution of the real-time application on a dedicated processing resource. This constraint simplifies the representation of the processing resource's availability, since the resource is used by a single real-time application. More complex resource models have been introduced in the context of open real-time systems, in which more applications that have different timing requirements share the same processing resource.

Open real-time systems have received lately an increased attention from the real-time research community. Some of the main issues addressed in research work are:

- Resource sharing between applications that have different timing requirements
- Guaranteeing that applications that are validated in isolation, have the same timing parameters while executing on a platform which is shared with other applications

The concept of *virtual resource* in real-time systems research was first introduced by [23], in the context of open systems. It provided an abstraction for the availability of a single resource shared between different applications. In the same paper, the concept of *supply function* is presented, to measure the amount of time the virtual resource is available.

The virtualization of computing resources is applied to uniprocessors as well as to multiprocessors (Fig.3). There are a number of different approaches in modeling virtual resources. The most representative will be presented as follows.

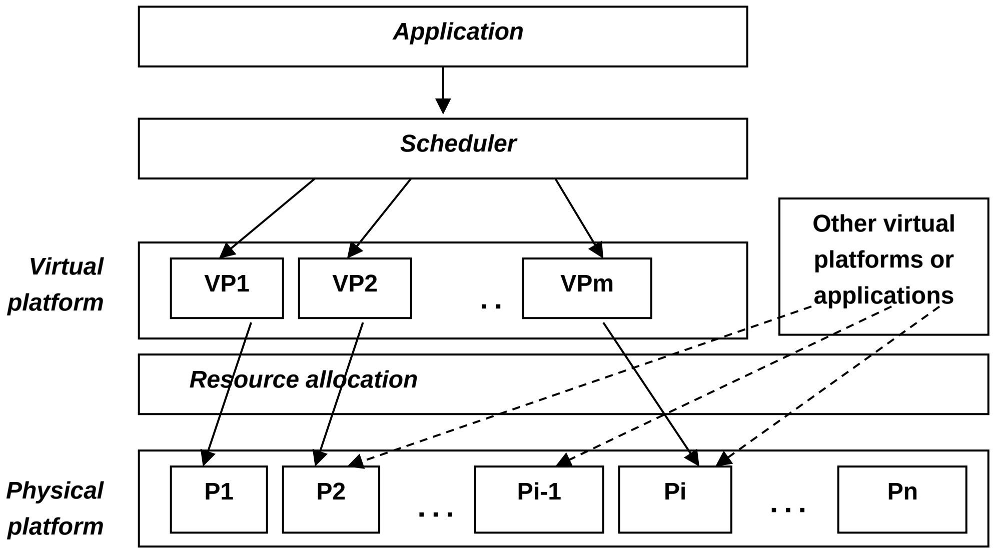


Figure 3. Open real-time system architecture that includes virtual platforms

3.2.1. *Virtual uniprocessor models*

The *resource partition model* was proposed by [23] as a solution for resource sharing in open real-time systems. The main problem addressed was that, in open systems, each task group (application) assumes exclusive access to the physical resource, and hence their scheduling policies may conflict. The authors presented the idea that each task group should have access to a virtual resource that represents a fraction of the shared physical platform. In this way, each individual application can use its specific scheduling policy without conflicts, while a second-level scheduler handles the service requests received from the virtual resources.

A *static resource partition* is defined as a pair Π=(Θ,P), where Θ is an array of *n* time pairs $\{(a_1,b_1), (a_2,b_2),\dots, (a_n,b_n)\}$ and *P* is the partition period. The time pairs represent the time intervals during which the physical resource is available for the partition and have to satisfy the following condition: $0\leq a_1 \leq b_1 \leq \dots \leq a_n \leq b_n \leq P$. The availability factor and the supply function characterize the partition. The *availability factor* of a partition is expressed as:

$$\alpha(\Pi) = \frac{\sum_{i\leq n}(b_i - a_i)}{P}$$

The *supply function S(t)* of a partition is the total amount of available time in the partition starting from time *0* to time *t*.

The static resource partition has a degree of rigidity because it assumes an exact knowledge of the available time intervals on the physical platform. That is why, in the same work [23], the authors provide a more flexible representation of the resource partition, the *bounded delay partition*. The bounded delay partition is described by the *bandwidth* (α) and the *delay* (Δ). The bandwidth represents the amount of resource supplied to the application and the delay represents the worst-case service delay.

In the resource partition approach, each application is independently scheduled to meet real-time requirements on its own partition. Application scheduling depends only on the partition parameters and it is independent of how the partitions are scheduled on the physical resource.

According to the resource reservation paradigm, the capacity of a processor can be partitioned into a set of reservations. Each reservation is equivalent to a virtual processor that provides a fraction of the available computing power.

The Constant Bandwidth Server (CBS) presented in [24] is an example of resource reservation model designed for the integration of soft and hard real-time systems. In this approach, it is assumed that all hard real-time applications run directly on the physical processor and that they are scheduled according to an algorithm which insures that all their deadlines are met. The remaining fraction of processor time will be used for soft real-time applications execution. This fraction of processor time is modeled by the CBS.

A CBS is defined by a budget q_s and a pair (Q_s, P_s), where Q_s is the maximum budget and P_s is the period of the server. The server bandwidth is: $U_s = Q_s/P_s$. The server has a fixed deadline $d_{s,k}$, at each time instant. It is assumed that $d_{s,0}=0$. When a job is served (executed), the budget q_s is decreased with the execution time of that job. When $q_s=0$, the budget is recharged at the maximum value Q_s and $d_{s,k+1}=d_{s,k}+P_s$. If a job arrives and the server is active, the job is put in the queue of pending jobs and served according to a non-preemptive algorithm (e.g. FIFO). If a job arrives and the server is idle, then if $q_s \geq (d_{s,k}-r_{i,j})U_s$, the server generates a new deadline $d_{s,k+1}=r_{i,j}+P_s$ and q_s is recharged at the maximum value, otherwise the job is served using the current budget and deadline.

The CBS model guarantees that if U_s is the fraction of processor time (or bandwidth) assigned to a server, the server's contribution to the total processor utilization factor is at most U_s, even in the presence of overloads. The CBS has the *temporal protection* property, which means that:

- The temporal behavior of a task can't be affected by other tasks' overruns
- The timing parameters of an application allocated to a virtual resource can be guaranteed in isolation and do not depend on other applications that run on the same physical platform.

The *periodic resource model* presented in [25] describes a partitioned resource that each period allocates a part of the total resource capacity, to an application modeled as a periodic task. A periodic resource model $\Pi=(\Theta,P)$ guarantees a resource allocation of Θ time units every P time units. The resource availability is described by two parameters:

- The *resource supply* – the amount of resource allocations that the resource provides during a time interval.
- The *service time* – the amount of time needed by the resource to provide a specified resource supply.

A recent development of the periodic resource model is the *Explicit Deadline Periodic* (EDP) resource model presented in [26]. The initial periodic resource model assumes an implicit deadline for the resource allocation, equal to the resource period. The EDP model introduces an explicit deadline Δ. As a consequence, the resource model $\Pi=(\Theta,P,\Delta)$ provides a resource allocation of Θ time units within Δ time units, every *P* time units. The motivation for using an explicit deadline is that a demand which can't be scheduled on a periodic resource because the length of the interval with no supply is larger than the earliest deadline in demand, could be scheduled on a periodic resource with the same capacity by lowering the resource deadline.

3.2.2. Virtual multiprocessor models

In [2] the authors present a multiprocessor bandwidth reservation scheme in which tasks are grouped in containers (an abstraction that allows isolation between task groups). Containers in a system form a hierarchy. The amount of resource (multiprocessor) which has to be allocated to a container is described by a parameter called *bandwidth*. Given a container *C* and its bandwidth *w(C)*, *C* will receive $\lfloor w(C) \rfloor$ fully available processors and at most one partially available processor. This way, the model enforces a minimum degree of parallelism on the resource supply. The supply function of a processor *k*, over a time interval *t*, is computed as follows:

$$S_k(t) = \max(0, \hat{u}_k(t - \sigma_k))$$

where $\widehat{u_k}$ is the processor bandwidth and σ_k is the maximum interval when the processor does not provide any supply. The resource supply for a container is characterized by a collection of such supply functions. If *m* is the number of processors (and also supply functions), for $1 \leq k \leq m-1$processors the supply functions will be $S_k(t)=t$, as all of them will be fully available for the container.

The *multiprocessor periodic resource model* is presented in [27] in the context of *virtual clustering* of multiprocessors. Clustering is an approach used in multiprocessor scheduling, which assumes that processors in a multiprocessor system are statically grouped into clusters, to reduce task migration during execution. Tasks are assigned to clusters and then they are globally scheduled on the corresponding cluster. In the virtual clustering approach, physical processors are dynamically assigned to clusters, and a processor can be part of two virtual clusters. To express the resource supply of a virtual cluster, the multiprocessor periodic resource (MPR) model is defined as $\Pi=(\Theta,P,m')$. An MPR specifies that an identical multiprocessor platform provides Θ time units in every period *P*, with the maximum degree of concurrency of *m'*. The model is feasible if $\Theta \leq m'P$. For $m'=1$ the MPR model reduces to the periodic resource model presented in [25].

In [28] the authors propose to represent a parallel machine as a virtual platform which contains *m* virtual processors $V = \{v_i\}_{i=1}^{m}$. Each virtual processor provides computing time according to a uniprocessor bounded delay partition, as in [23]. Given this virtual platform model, a *Multi Supply Function* (MSF) is a set of *m* supply functions $\{Z_{v_i}\}_{i=1}^{m}$, one for each virtual processor.

In the context of a time partition *P*, a supply function is the minimum amount of time allocated by the partition in every interval of time of length $t \geq 0$:

$$Z_P(t) = \min_{t_0 \geq 0} \int_{P \cap [t_0, t_0 + t]} 1 dx$$

Given the set of partitions that can be allocated by a virtual processor, legal(*v*), the supply function of the virtual processor is:

$$Z_v(t) = \min_{P \in legal(v)} Z_P(t)$$

A delay bounded model (α,Δ) can be derived for each virtual processor from its supply function,

$Z_v(t)$:

$$\alpha = \lim_{t -> \infty} \frac{Z_v(t)}{t}$$

$$\Delta = \sup_{t \geq 0} \left\{ t - \frac{Z_v(t)}{\alpha} \right\}$$

The approach presented in [28] can be used to implement parallel real-time applications independently of the physical platform. The virtual processors are implemented using reservations, as sequential servers.

The *Parallel Supply Function* (PSF) proposed in [3] is used as the interface of a virtual platform, and expresses the computing capacities of a virtual platform implemented on a multiprocessor. The authors extend the resource partition model presented in [23] to multiprocessors through multi-partitions.

A *multi-partition* is the aggregation of all time partitions defined for each individual processor. The multi-partition is formally defined as a multi-set of time intervals. In a multi-set, individual elements may occur multiple times, as the same time interval may appear on two different processors. The characteristic function of a subset A is defined as:

$$\gamma_A(t) = \begin{cases} 1, t \in A \\ 0, t \notin A \end{cases}$$

The characteristic function of a multi-partition *P* is:

$$\gamma_P(t) = \sum_{[a_i, b_i) \in P} \gamma_{[a_i, b_i)}(t)$$

The *maximum degree of parallelism* of a multi-partition is:

$$M(P) = \max_{t \geq 0} \gamma_P(t)$$

The *level-j supply function* $Y_{j,P}(t)$ expresses the minimum amount of computing capacity provided by a multi-partition every interval of length $t \geq 0$ by at most *j* intervals in parallel:

$$Y_{j,P}(t) = \min_{t_0 \geq 0} \int_{t_0}^{t_0+t} \min\{j, \gamma_P(x)\} dx$$

To define the PSF of a virtual platform Π, implemented on *m* identical processors, the authors first extend the multi-partition level-j supply function to the set of multi-partitions that can be allocated by Π, legal(Π):

$$Y_j(t) = \min_{P \in legal(\Pi)} Y_{j,P}(t)$$

Finally, the PSF of a platform Π is defined as the set of level-j supply functions $\{Y_j\}_{j=1}^{m}$. The PSF is used to derive a delay bounded model (α,Δ) for the virtual platform, as in [28].

3.3. Discussion

For the rest of this book, we will refer to multiprocessor platforms in both their forms:

- Without network communication
- Distributed (with network communication)

We will mainly use the homogeneous multiprocessor model, in which tasks execute at the same speed on all processors (processors are identical).

In Chapter 3, we use the resource reservation approach to model the availability of open network segments that allow the transmission of real-time traffic at the same time with non-real-time traffic.

In Chapter 4, we use a platform availability modeling approach similar to the resource partition approach in [23]. We find this model suitable to express the availability of execution time in a certain discrete time interval that has been partially “occupied” by a number of tasks, to build cyclic schedules for real-time transaction sets.

4. Scheduling models

The time at which each tasks finishes the execution is essential for the correctness of a real-time system. To guarantee that the system response does not exceed the requested deadline, the use of appropriate scheduling techniques is necessary. A *scheduling technique* establishes a set of rules used to impose the execution order of all released jobs. The component of a computing system that applies the rules imposed by the scheduling technique is called *scheduler*. Each hardware platform that has to be shared between a set of jobs needs a scheduler or a set of schedulers that have the role of establishing jobs’ execution order.

Two main approaches are used in real-time scheduling [4]:

- *Clock-driven*
- *Priority-driven*

In the clock-driven approach, scheduling decisions are made at specific time instants. These time instants are chosen before the system is started. The schedule of the jobs is computed off-line and stored. The scheduler uses the pre-computed schedule at run time, at each scheduling decision time, to choose which job will be executed next. The schedule used by a clock-driven scheduler has to contain all scheduling decisions that have to be made since system start to its stop. This is possible only if all parameters of the workload are known a-priori and fixed (do not change over time). The schedule of a workload that contains only periodic tasks, is periodic and it repeats with the task set's hyper-period. The *hyper-period* of a task set is equal to *the least common multiple* (LCM) of the tasks' repetition periods. A periodic static schedule is called a *cyclic schedule*. If a cyclic schedule can be computed before system start, the scheduler will restart the same schedule at the beginning of each hyper-period. The clock-driven approach has the following main advantages:

- The static schedule can be represented by a table of job start times and completion times. The scheduler uses the table at run time. The implementation of such a system is straightforward. From the execution time point of view, there is a minimum scheduling overhead, because the scheduler does not make the decisions at runtime.
- Systems based on clock-driven scheduling approach are relatively easy to test and validate trough simulation.

However, this approach has some important disadvantages:

- It can't handle dynamic systems such as tasks that have variable parameters that can't be predicted in advance.
- It can't handle sporadic and aperiodic tasks.
- They are difficult to modify and maintain.

Clock-driven approach is best suited in the case of periodic task systems that have a bounded hyper-period, and that once built, don't change very often (e.g. small embedded systems).

In the priority-driven approach, each job that is ready for execution is given a priority. At any scheduling decision time, the jobs with the highest priorities are chosen to be executed on the available processors. Scheduling decisions are made at runtime. Compared to the clock-driven approach priority-driven scheduling is more flexible and can handle a large variety of task models. Scheduling decisions are made at runtime, but with a larger scheduling overhead.

It is said that the priority-driven approach is *work-conserving* because it does not allow any processor/resource to be idle when there are jobs ready for execution. This approach is *event-driven*, that is, scheduling decisions are made when events such as job releases or job completions occur. The jobs' priorities are assigned according to a scheduling algorithm. A *scheduling algorithm* receives as input a *task set* and has to establish the *order* in which jobs generated by the task set are executed on the platform, by assigning priorities.

4.1. Classification of scheduling algorithms

One can classify scheduling algorithms based on different criteria. In [37] the authors distinguish three classes based on the priority assignment scheme:

- *Fixed priority* scheduling algorithms assign a priority for each task. The priority is inherited by the task's jobs. Task priority does not change during execution (example: Rate Monotonic algorithm [5]).
- *Job-level dynamic priority* scheduling algorithms assign a priority for each job. Job priority does not change during execution. Jobs belonging to the same task may have different priorities, but the relative priority of two jobs stays the same (example: Earliest Deadline First algorithm [5]).
- *Dynamic priority* scheduling algorithms assign priorities for each job, which may be changed any time during execution. The relative priority of two jobs may change in time (example: Least Slack First algorithm [6]).

In [22] scheduling algorithms are further classified as:

- *Preemptive*, when the algorithms allow jobs with higher priority to interrupt the execution of lower priority jobs.
- *Non-preemptive*, when algorithms do not allow job interruption during execution. A job that starts its execution will occupy the processor until its completion.
- *Co-operative*, when jobs can be preempted at predefined points during their execution.

4.2. Main problems of real-time scheduling

To guarantee the *correctness* of a real-time system, one has to prove that each job will finish its execution before its deadline. Therefore, real time scheduling has been mainly focused on analytically solving the *feasibility* and the *schedulability* problems [37].

Given a system model, a task set is considered *feasible* if there is a schedule so that all tasks are executed without missing any of their deadlines.

A task is *schedulable* according to algorithm *A* if its worst-case response time obtained with this algorithm is less or equal to its deadline. A task set is schedulable if all its tasks are schedulable. Given a system model, a scheduling algorithm is *optimal* if it can schedule all task sets that are feasible on the system.

A *schedulability test* assesses the schedulability of a task set according to a given algorithm. The schedulability test is *sufficient* if all task sets that satisfy the condition are schedulable. The schedulability test is *necessary* if all task sets that do not satisfy the condition are not schedulable. A schedulability test is *exact* if it is at the same time necessary and sufficient. The majority of schedulability tests developed and used in real-time scheduling (especially in the case of multiprocessors) theory and practice are sufficient tests. These tests are based on the system's worst-case behavior and sometimes exclude some of the schedulable task sets. On uniprocessors, the worst case response time of a task occurs at its *critical instant*, when the task is released at the same time with all the tasks that have higher priorities. On multiprocessors, however, the critical instant does not occur under the same conditions as on uniprocessors. It is known [22] that the critical instant has not been identified in the case of multiprocessor scheduling of sporadic tasks. In [32] the authors evaluate different sufficient multiprocessor schedulability tests and show that there is large number of task sets that are in the “un-decidable zone” (see Fig.4), that is, they did not prove unfeasible, but were rejected by the sufficient feasibility tests.

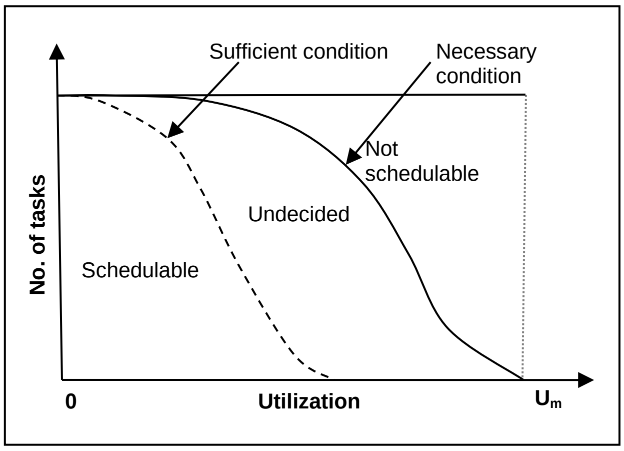


Figure 4. The limits of multiprocessor sufficient and necessary schedulability tests as a function of total task set utilization

4.3. Uniprocessor scheduling

The most representative priority-driven scheduling algorithms for uniprocessors are the Rate Monotonic (RM) and Earliest Deadline First (EDF) algorithms, which were first presented and analyzed by Liu and Layland in [5]. Both RM and EDF algorithms are optimal for sporadic task sets on uniprocessors.

RM is a fixed priority scheduling algorithm that assigns priorities to tasks according to their repetition rate ($f = \frac{1}{p}$). The algorithm will assign priorities that are proportional with the task’s repetition rate. Jobs of the same task will inherit the task’s priority. The RM algorithm can schedule all task sets that have total utilization factor less than $U = m(2^{\frac{1}{m}} - 1)$, where *m* is the task set’s cardinality. The largest utilization factor of tasks sets that can be scheduled is around 70% when *m* is larger than 10, and slightly increases to 78% when *m* decreases.

EDF is a job-level dynamic priority algorithm that assigns priorities to jobs according to their absolute deadline. The job with the closest deadline will be assigned the highest priority. Job priority will not change during execution, but jobs belonging to the same task have different priorities. The EDF algorithm can schedule all task sets that that have total utilization factor less than *U=1*, that is any task set that fully utilizes the processing time will be scheduled.

Another important uniprocessor scheduling algorithm is Least Slack First (LSF). LSF is a dynamic priority scheduling algorithm presented and analyzed by Mok in [6]. It is also known under the name of Least Laxity First algorithm. The *slack* of a job is defined as *“the maximum time the scheduler can delay running the job before it is bound to miss its deadline”*. LSF algorithm assigns priorities to jobs according to their slack computed at the time the scheduling decision is made. The job with the shortest slack is assigned the highest priority. As the slack decreases every time instant if the job does not execute, the job’s priority may change very often. Moreover, there are a lot of preemptions because the relative priority of jobs changes during execution. The overheads introduced by the very frequent priority re-computation and by preemptions discouraged the practical use of this algorithm. The LSF algorithm is optimal for sporadic task sets and can schedule all task sets that that have total utilization factor less than $U=1$(like EDF).

4.4. Multiprocessor scheduling

In multiprocessor systems, including distributed systems, the scheduling problem has two aspects:

- *Allocation*: for each task/job, choose a processor on which it will execute.
- *Priority assignment*: establish in which order jobs will be executed.

From the allocation point of view, multiprocessor scheduling has been classified as [37]:

- *Global*
- *Partitioned*
- *Clustered*

In the global approach, jobs can be allocated to any available processor and can migrate to other processors during execution with no restrictions. Global scheduling assumes the existence of a unique global scheduler and a global job queue. At each time instant, the global scheduler chooses from the job queue n jobs with the highest priority, where n is the number of available processors. The chosen jobs will be executed on the processors. Since job migration is allowed, it is not important for a job that started its execution on a certain processor, once preempted, to continue its execution on the same processor.

Partitioned scheduling assumes that each processor has its own scheduler and job queue. Each task is assigned to a single processor. Jobs can only be assigned to the processor the task was assigned to. Job migration to other processors is not allowed.

In clustered scheduling, job migration is restricted to a subset of the available processors. Processors are grouped into clusters. Each cluster has its own scheduler and job queue. First, tasks are allocated to clusters, and then each cluster scheduler globally schedules jobs inside the cluster. It can easily be observed that the global and partitioned scheduling are instances of clustered scheduling. If the cluster size is equal to the total number of processors, then we have global scheduling. If each cluster contains only one processor (the number of clusters is equal to the number of processors), we have partitioned scheduling.

Partitioned scheduling is done in two steps. First, tasks are allocated on processors by using a partitioning algorithm. Second, a uniprocessor scheduling algorithm such as RM or EDF is applied to schedule the tasks allocated on each processor.

Task set partitioning on a multiprocessor is equivalent to the Bin-Packing problem that requires placing n objects of different dimensions in m boxes. It is known that the Bin-Packing problem is NP-hard. As optimal solutions can only be found through exhaustive search, for practical reasons, suboptimal solutions known as *partitioning heuristics* are used in real-time scheduling. Some of the most common partitioning heuristics are the following:

- First Fit (FF) – allocates the task to the first processor that verifies the schedulability condition after the allocation. The search for an available processor always starts with the first processor in the processor list.
- Next Fit (FF) - allocates the task to the next processor (starting with the current processor) that verifies the schedulability condition after the allocation.
- Best Fit (BF) - allocates the task to the processor that minimizes the remaining processor capacity.
- Worst Fit (WF) - allocates the task to the processor that maximizes the remaining processor capacity.

Usually some task sorting procedure is applied before partitioning the task set. By sorting the tasks, the result of the partitioning heuristic can be improved. The sorting criterion can be task deadline, period, utilization, and others.

For some time, partitioned scheduling received more attention for two reasons:

- The schedulability analysis of a task set partitioned on a multiprocessor reduces to the analysis of m task sets on m uniprocessors.
- Global scheduling suffers from the *"Dhall effect"*. The authors in [38] demonstrated that when there are m tasks with short periods/deadlines and very low utilizations, and one task with utilization close to 1 that have to be scheduled with global EDF on m processors, the heavier task misses its deadline, so the utilization bound of global EDF is $1+\varepsilon$, for arbitrary small ε. Furthermore, authors in [40] showed that for partitioned scheduling of periodic tasks with implicit deadlines, the utilization bound is $(m+1)/2$.

The main drawback to the partitioned approach is the task allocation step, in which each task is allocated to a processor for execution. The allocation problem is known to be NP-hard. Moreover, the partitioned approach is not work-conserving, so ready jobs allocated to one processor may stay in the queue while other processors are idle resulting in the fragmentation of processing capacity.

Compared to partitioned scheduling, global scheduling is work conserving, but with the cost of migration overhead that didn't exist in the partitioned approach. There are some global dynamic priority scheduling algorithms that are optimal for periodic task sets with implicit deadlines such as Proportionate Fair family algorithms [29] and LLREF [39]. However, these optimal algorithms are not used in practice due to very large overheads they introduce through high frequency migrations and preemptions. A large number of research works [34][43][44][45] try to adapt optimal uniprocessor scheduling algorithms such as RM and EDF to global multiprocessor scheduling. It was demonstrated in [34] that the maximum utilization achieved by global EDF is influenced by the value of the heaviest task (in terms of task utilization): $U_{EDF} = m - (m-1)u_{max}$. A number of improved EDF-based algorithms [43][44] have increased the utilization bound, but neither of them is optimal.

Clustered scheduling is similar to partitioned scheduling, as the task set is partitioned on the clusters. Because each cluster contains two or more processors, the scheduling inside each cluster is in fact global scheduling. By reducing the number of processors in a cluster the overheads introduced by global scheduling (global queue size, migrations) are reduced. Capacity fragmentation is less of a problem than in partitioned scheduling.

Semi-partitioned scheduling [41][42] is a hybrid between partitioned and global ones that tries to overcome the disadvantages of both. This approach is applied to task sets that can't be partitioned on the available processors. In the first step, tasks are allocated to processors until no other task can be allocated. These tasks are scheduled according to the partitioned approach. The remaining tasks are scheduled according to the global approach, thus they are allowed to migrate on any processor that is available for execution. There are two main variants of semi-partitioned scheduling, each of them using different methods for reducing the number of migrations for tasks that are scheduled with the global approach. The first method statically allocates portions of tasks to available processors. The task's portions are executed on the allocated processors without supplementary migrations. The second method defines a restricted migration pattern for jobs of tasks that are globally scheduled. In this case, the jobs are not allowed to migrate, but successive jobs of the same task can be released on different processors.

In the case of distributed systems, the scheduling problem is similar to the partitioned scheduling problem if the individual physical processing resources are uniprocessors. Otherwise, if the processing resources are multiprocessors, the scheduling is resolved by a hierarchy of schedulers that may be partitioned, global or clustered. In distributed scheduling, tasks have to be allocated on the individual processing resources by using a partitioning heuristic. Local schedulers solve the task scheduling on each individual processing resource after allocation. The allocation and scheduling problems usually have more constraints, as the tasks have precedence and communication dependencies.

4.5. The Loose Clustered Approach

Priority-based scheduling and in particular Rate Monotonic (RM) [5] and Earliest Deadline First (EDF) [5] algorithms received a great deal of attention from the real-time research community. From among the proven optimal uniprocessor scheduling algorithms, RM and EDF are successfully used in practice.

On the other hand, multiprocessor and distributed scheduling still raise open research problems. Among the open problems related to multiprocessor and distributed scheduling are:

- Task allocation/partitioning in both distributed and multiprocessor systems
- The implementation of optimal multiprocessor scheduling algorithms for real-world applications
- Scheduling real-time parallel applications on multiprocessors
- Scheduling tasks with interdependencies on multiprocessors and distributed systems

Most research efforts were, at first, directed towards adapting the most successful uniprocessor scheduling techniques to multiprocessors. The same well established workload models such as the periodic and sporadic (sequential) task models were chosen for analysis. As multiprocessor platforms introduces new variables, many uniprocessor research results proved inapplicable to multiprocessors. Moreover, the multiprocessor variants of RM and EDF algorithms are not optimal. As result of these observations, two major research directions emerged:

- In the first direction, efforts are made to introduce new multiprocessor scheduling algorithms, which can be implemented in practice.
- The second direction concentrates its efforts on finding the best schedulablity tests for well known algorithms such as RM and EDF.

The work in these directions is far from being finished. Optimal multiprocessor scheduling algorithms such as the Proportionate Fair algorithm [29] and its variants [30][31] are almost impossible to be implemented in practice due to very large overheads introduced by their very frequent scheduling decisions and task preemptions. In the second direction, up until now, mainly necessary or sufficient analytical schedulability tests were found, most of them having very high complexity levels [32]. The existing sufficient schedulability tests introduce very strong constraints that excessively limit their results.

As an example, we can mention the case of the *global EDF* algorithm. In [32] the authors surveyed the most important seven sufficient schedulability tests with different computational complexity levels. Their results show that there is a large interval on the total utilization axis, which is not covered by the schedulability tests. The interval increases with the number of processors. In the 8 processor scenario, this interval starts at a total utilization approximately equal to 4.5 (about 56%). This means that all evaluated schedulability tests introduce strong constraints on the task sets they assess, leaving outside their limits a large number of task sets which may be schedulable.

The results we reviewed show that remains a great deal of research work to be done concerning multiprocessor real-time scheduling.

For the rest of this book we intend to discuss problems related to multiprocessor scheduling, and in particular:

- Real-time communication scheduling in the context of real-time distributed systems timing analysis. We adapt the well known Response Time Analysis technique to real-time communication and we create a method for network bandwidth estimation (detailed in Chapter 3).
- Multiprocessor scheduling of real-time transactions. We propose two different techniques for solving this problem. The first technique creates schedules based on a clock-based approach and is suitable for small embedded systems. The second technique combines a genetic algorithm with simulation-based evaluation of candidate solutions to find feasible system setups (detailed in Chapter 4).
- Simulation-based analysis of real-time multiprocessor systems. We create an environment that allows the simulation of a variety of real-time systems models and as well supports the statistical evaluation of different scheduling techniques (detailed in Chapters 5 and 6).

For our work on multiprocessor scheduling, we addopt a model based on the *clustered* approach. In this way, we can customize the scheduler to be partitioned or global, as they are particular cases of a clustered scheduler. Through parametrization, we can obtain models suited for the representation of multiprocessor and distributed scheduling. Each cluster can apply a different priority-based scheduling algorithm.

We propose an optimization of the classic clustered scheduling, the loose clustered scheduling mechanism. Our approach extends the classic clustered approach with a load balancing mechanism that has the following objectives:

- Equalize cluster utilization;
- Reduce capacity fragmentation due to not optimal task partitioning between clusters.

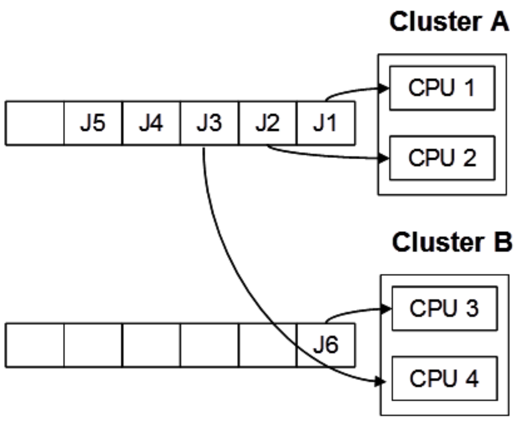


Figure 5. The loose clustered approach

Each time a processor in a cluster is idle, it can execute a job from another cluster's queue if a collaboration relation has been defined between the clusters. A collaboration relation between two clusters can be statically defined before the start of the system. Each of the collaborating clusters can execute a job from the other's queue. The migrated job can be executed on the free CPU with the lowest priority and it will not affect the response time of other jobs. Fig. 5 depicts an example of this mechanism. The collaboration relation is defined between cluster A and cluster B that both contain two CPUs. J3 is scheduled on cluster A, but can execute on cluster B because CPU 4 is idle.

In Chapter 5 we describe the implementation of this model in a simulation environment that we use for real-time systems analysis. In Chapter 6 we evaluate this new approach against the clustered, global and partitioned scheduling.

5. Conclusion

This chapter reviews the most important models and algorithms used in real-time systems current research. The analysis of theoretical models was made by addressing the main components of a real-time system:

- Workload

- Platform
- Scheduling

In the context of multiprocessor real-time systems, recent research work is directed towards finding new models to describe real-time parallel applications and parallel platforms.

Workload models capture real-time application patterns, which are used in real-time systems analysis. Most of the current analysis techniques on multiprocessors use, for simplicity, the Liu and Layland periodic task model or the sporadic task model which were developed in the uniprocessor era of real-time systems. However, latest research efforts try to integrate parallel application models in real-time systems analysis. Another direction, which needs attention, is that of task dependencies and real-time distributed transactions.

Platform models try to describe the computing resource parallelism through different abstractions such as reservations or parallel supply functions.

In the direction of multiprocessor scheduling, efforts are made to introduce new multiprocessor scheduling algorithms, which can be implemented in practice and as well to find the best schedulablity tests for widely used algorithms such as RM and EDF.

Based on the review of the current research results in real-time systems modeling and analysis, we identified some directions, which may accommodate new contributions such as:

- Accommodating real-time communication over general purpose networks, in the context of distributed real-time systems
- Real-time transactions scheduling on multiprocessor platforms
- Real-time systems simulation
- Simulation-based analysis of real-time multiprocessor systems

References

[1] Hermann Kopetz, *Real-Time Systems, Design Principles for Distributed Embedded Applications,* Springer, 1997

[2] Hennadiy Leontyev, James H. Anderson, "*A Hierarchical Multiprocessor Bandwidth Reservation Scheme with Timing Guarantees*", Euromicro Conference on Real-Time Systems, 2008

[3] Enrico Bini, Marco Bertogna, Sanjoy Baruah, "*Virtual Multiprocessor Platforms: Specification and Use*", IEEE Real-Time Systems Symposium, 2009

[4] Jane W.S. Liu, *Real-Time Systems*, Prentice Hall, 2000

[5] C. L. Liu, James W. Layland, "*Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*", Journal of the Association for Computing Machinery, 1973, 20

[6] Aloysius Mok, "*Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*", PhD Thesis, MIT, 1983

[7] Aloysius K. Mok, Deji Chen, "*A Multiframe Model for Real-Time Tasks*", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 1997, 10

[8] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, Aloysius Mok, "*Generalized multiframe tasks*", Real-Time Systems, 1999, 17

[9] Noel Tchidjo Moyo, Eric Nicollet, Frederic Lafaye, Christophe Moy, "*On Schedulability Analysis of Non-Cyclic Generalized Multiframe Tasks*", 22nd Euromicro Conference on Real-Time Systems, 2010

[10] Sanjoy Baruah, "*Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks*", Real-Time Systems, 2003, 24

[11] Sanjoy Baruah , "*The non-cyclic recurring real-time task model*", 31st IEEE Real-Time Systems Symposium, 2010

[12] Martin Stigge, Pontus Ekberg, Nan Guan, Wang Yi, "*The Digraph Real-Time Task Model*", 2011

[13] Elena Fersman, Pavel Krcal, Paul Pettersson and Wang Yi, "*Task Automata: Schedulability, Decidabilityand Undecidability*"

[14] R. Alur, D.L. Dill, "*A Theory of Timed Automata*", Theoretical Computer Science, 1994, 126 (2)

[15] Sébastien Collette, Liliana Cucu, Joël Goossens, "*Integrating job parallelism in real-time scheduling theory*", Information Processing Letters, 2008, 106, 5, pp. 180–187

[16] Shinpei Kato, Yutaka Ishikawa, "*Gang EDF Scheduling of Parallel Task Systems*", 2009 30th IEEE Real-Time Systems Symposium, 2009

[17] Joel Goossens, Vandy Berten, "*Gang FTP scheduling of periodic and parallel rigid real-time tasks*", 2010

[18] Karthik Lakshmanan, Shinpei Kato, Ragunathan (Raj) Rajkumar, "*Scheduling Parallel Real-Time Tasks on Multi-core Processors*", RTSS, 2011

[19] Hoon Sung Chwa, Jinkyu Lee, Hyoungbu Back, Jaebaek Seo, InsikShin, "Schedulability Analysis of Fork-Join ParallelTasks on Multiprocessors", 2011

[20] Jeffrey R. Merrick, Shige Wang, Kang G. Shin, Jing Song, William Milam, "*Priority Refinement for Dependent Tasks in Large Embedded Real-Time Software*", 11th IEEE Real Time and Embedded Technology and Applications Symposium, 2005

[21] Julien Forget, Frederic Boniol, Emmanuel Grolleau, David Lesensy, Claire Pagetti, "*Scheduling Dependent Periodic Tasks Without Synchronization Mechanisms*", RTAS10, 2010

[22] Robert I. Davis, Alan Burns, "*A Survey of Hard Real-Time Scheduling Algorithms and Schedulability Analysis Techniques for Multiprocessor Systems*", 2009

[23] A. Mok, X. Feng, D. Chen, "*Resource partition for real-time systems*", Real-Time Technology and Applications Symposium, 2001, pp. 75–84

[24] Luca Abeni, Giorgio Buttazzo, "*Integrating Multimedia Applications in Hard Real-Time Systems*", Proceedings of IEEE Real-Time System Symposium,Madrid, Spain, 1998

[25] Insik Shin, Insup Lee, "*Periodic Resource Model for Compositional Real-Time Guarantees*", Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS'03), 2003

[26] Arvind Easwaran, Madhukar Anand, Insup Lee, "*Compositional Analysis Framework using EDP Resource Models*", 28th International IEEE Real-Time Systems Symposium, 2007

[27] Insik Shin, Arvind Easwaran, Insup Lee, "*Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors*", Euromicro Conference on Real-Time Systems, 2008

[28] Enrico Bini, Giorgio C. Buttazzo, Marko Bertogna, "*The Multi Supply Function Abstraction for Multiprocessors*", 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2009

[29] S.K. Baruah, N. Cohen, G. Plaxton, D. Varvel, "*Proportionate progress: A notion of fairness in resource allocation*", Algorithmica 15, no. 6, pp. 600–625, 1996

[30] J. Anderson, A. Srinivasan, "*Early-release fair scheduling*". In Proceedings Euromicro Conference on Real-Time Systems, June 2000

[31] J. Anderson, A. Srinivasan, "*Mixed pfair/erfair scheduling of asynchronous periodic tasks*". In Proceedings of the 13th Euromicro Conference on Real-Time Systems, June 2001

[32] M. Bertogna, S. Baruah, "*Tests for global EDF schedulability analysis*", Journal of Systems Architecture, no. 57, pp. 487–497, 2011

[33] L. Cucu, J. Goossens, "*Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors*", ETFA, Prague, September 2006

[34] J. Goossens, S. Funk, S.Baruah, "*Priority-driven scheduling of periodic task systems on multiprocessors*", Real Time Systems 25 (2–3) (2003) 187–205.

[35] J.J.G. Garcia, M.G. Harbour "*Optimized priority assignment for tasks and messages in distributed hard real-time systems*," Parallel and Distributed Real-Time Systems, 1995. Proceedings of the Third Workshop on , vol., no., pp.124-132, 25 Apr 1995

[36] M. Yoo, M. Gen, *"Scheduling algorithm for real-time tasks using multiobjective hybrid genetic algorithm in heterogeneous multiprocessors system",* Computers & Operations Research 34 (2007) 3084 – 3098

[37] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, S.K. Baruah. *"A categorization of real-time multiprocessor scheduling problems and algorithms"* In Handbook of Scheduling: Algorithms, Models, and Performance Analysis, 2004.

[38] S. K. Dhall, C. L. Liu, *"On a Real-Time Scheduling Problem"*, Operations Research, vol. 26, number 1, pp. 127-140, 1978.

[39] H. Cho, B. Ravindran, E.D. Jensen, *"An Optimal Real-Time Scheduling Algorithm for Multiprocessors"*. In Proceedings of the Real-Time Systems Symposium pp. 1001-110, 2006.

[40] B. Andersson, S. Baruah, J. Jonsson, *" Static-priority scheduling on multiprocessors,"* In Proc. 22nd IEEE Real-Time Systems Symposium, pages 193–202, London, UK, Dec. 2001.

[41] L. George, P. Courbin, Y. Sorel, "Job vs. portioned partitioning for the earliest deadline first semi-partitioned scheduling", Journal of Systems Architecture 57 (2011) 518–535

[42] Y. Ishikawa, "Semi-Partitioned Scheduling of Sporadic Task Systems on Multiprocessors", 21st Euromicro Conference on Real-Time Systems, ECRTS, 2009

[43] M. Bertogna, M. Cirinei, G. Lipari, "Improved schedulability analysis of EDF on multiprocessor platforms". In Proceedings of the 17th Euromicro Conference on Real-Time Systems, pp. 209–218, 2005.

[44] T.P. Baker, "An analysis of EDF scheduling on a multiprocessor". IEEE Trans. on Parallel and Distributed Systems, 15(8):760–768, Aug. 2005.

[45] B. Andersson, S. Baruah, J. Jonsson. Static-priority scheduling on multiprocessors. In Proc. 22nd IEEE Real-Time Systems Symposium, pages 193–202, London, UK, Dec. 2001.

CHAPTER 3. REAL-TIME COMMUNICATION OVER GENERAL PURPOSE NETWORKS 1

Chapter 3. Real-time communication over general purpose networks

1. Introduction

Distributed real-time systems such as remote process supervision and control systems require a communication infrastructure that supports reliable and safe real-time data transmission. These requirements are usually solved by using dedicated networks and industrial protocols, as presented in [8] and [9]. As these special purpose protocols are incompatible with general purpose protocols used in local area networks and company intranets, there are interoperability problems between distributed control applications and organizational software. Local computer networks and Internet protocols, on the other hand, fail to satisfy real-time requirements because they apply the best-effort principle in supplying communication services, and it is quite difficult to use them as infrastructure for real-time applications. To satisfy real-time communication requirements on networks that use Internet protocols, solutions that enable a predictable network behavior and that provide end-to-end delivery time guarantees have to be developed.

The recent expansion of the real-time applications domain created a significant research trend towards finding solutions to accommodate both real-time and non-real-time applications on a common environment [1][2][3][4][5][6]. Common use hardware and communication protocols do not offer timing guarantees, which are essential in the case of real-time applications. The main challenge in this case is to find theoretical and pragmatic methods that facilitate the estimation of applications response time, as support for real-time applications. Because of this trend, there is significant research work oriented in three main directions:

- Implement real-time applications on common use microprocessors [4] [5][6]
- Accommodate real-time data transmission on general purpose networks [1][2][7]
- Integrate real-time specific features in common-use operating systems [3]

This chapter presents a solution for using general purpose (non-deterministic) networks for real-time communication. The research work in this direction was mainly encouraged by networking technology developments such as increasing network bandwidth (Gbps) and development of new Quality of Service (QoS) mechanisms. In this context, the (communication-related) issues are generated by the integration of distributed control systems with other information systems that do not require special communication services.

This solution can guarantee the timing requirements for real-time data transmission over general purpose networks. A real-time communication model based on data flow analysis is used as support for a reservation-based communication architecture. Based on the same model, the network bandwidth required by real-time traffic is analytically estimated.

2. Literature review

Many research efforts are directed towards the formalization of models that describe the sharing of resources between applications that have different timing requirements. The main objective in this case is to guarantee the timing requirements of each application, in isolation. The concepts of virtual resources, resource partition [10] and resource reservation [11] have been proposed as support for the analytic determination of application's timing parameters in isolation. Moreover, resource reservation techniques have been combined with feedback techniques to provide response delay and respectively execution time guarantees for tasks that coexists in a shared environment [1][12]. These concepts have been recently extended to multiprocessor platform models [4][5].

Several authors investigated the problem of accommodating real-time traffic on best-effort networks and proposed some pragmatic solutions. In [13] the authors present a method for managing the network bandwidth for multiple client applications. Their communication middleware, the NIProxy, is able to partition available client bandwidth between real-time and non real-time traffic flows by arranging them in a stream hierarchy. This solution gives good results in improving client Quality of Experience, but it does not guarantee any end-to-end timing requirements for real-time traffic.

Another approach is presented in [7], where the authors propose introducing a prioritization mechanism in the Internet protocol suite, a mechanism that complies with the IEEE 802.1D standard. They evaluate the solution through simulation, using the OPNET simulator, by measuring end-to-end latency of real-time packets in the presence of FTP traffic on the same network. Their conclusion is that a large part of the end-to-end message latency occurs at the end nodes, assuming that the network bandwidth is large enough to support all the traffic. In the referred paper the authors do not provide a solution for evaluating network bandwidth requirements, they just assume that the bandwidth is large enough.

In [14], Martinez et al. present Earliest Deadline First (EDF) communication scheduler implementation adapted for high-performance networks. The characteristics of high-performance networks enabled them to simplify the calculus of packet deadline, taking into consideration only the previous packet's deadline, packet size and average bandwidth.

Schantz et al. [2] describe two approaches, priority-based and reservation-based, in developing distributed real-time middleware. For both solutions, the communication infrastructure is an IP network. In the priority-based middleware the standard Differentiated Services (DiffServ) mechanism is implemented for network resource management. In the reservation-based middleware the Integrated Services (IntServ) mechanism is implemented. Their main contribution is in the area of middleware implementation. But there are two quite important issues not addressed by this paper:

- the provision of instruments for evaluating the resource requirements of real-time tasks;
- the differentiation between hard and soft tasks.

IntServ [15][16] provides end-to-end per-flow QoS by means of hop-by-hop resource reservation within the IP network but impose a significant burden on the core routers. To reduce the complexity within each core router, alternative schemes, referred to as Measurement Based Admission Control

Schemes (MBAC) have been proposed [17]. These schemes replace per-flow states with run-time link load estimates performed in each router. However, MBAC solutions still require significant modification of the existing Internet architecture, as core routers must support load estimation algorithms, and still need to be explicitly involved in per flow signaling exchange.

A completely different approach is provided by DiffServ [18]. In DiffServ, core routers are stateless and unaware of any signaling. While DiffServ easily enables resource provisioning performed in a management plane for permanent connections, their widely recognized limit is the lack of support per-flow resource management and admission control, resulting in the lack of strict per flow QoS guarantees. A number of proposals, presented in the literature, have shown that per flow Distributed Admission Control schemes can be deployed over DiffServ architectures [19][20]. Although significantly different in implementation, they share the common idea that accept/reject decisions are taken by the network endpoints and are based on the processing of "probing" packets, injected in the network at setup to verify the network congestion status. A "pure" Extended Admission Control (EAC) scheme, called Phantom Circuit Protocol-Delay Variation (PCP-DV) is proposed in [21]. The scheme determines whether a new connection request can be accepted based on delay variation measurements taken on the probing packet at the edge nodes.

Reinemo et al. [22] propose and evaluate three different admission control schemes for virtual cut-through networks, each one suitable for use in combination with DiffServ based QoS scheme to deliver soft real-time guarantees. Two of the schemes assume pre-knowledge of the network's performance behavior without admission control and are both implemented with bandwidth broker. The third is based on endpoint/egress admission control and relies on measurements to assess the load situation. Due to the way in which the flow control affects latency and the nature of cut-through networks, latency and jitter properties are hard to achieve.

An approach to quantify the impact of end-to-end QoS provisioning through a combination of both intra and inter-autonomous system (AS) traffic engineering (TE) is proposed in [23]. Two offline QoS-aware systems are deployed for this and a direct relationship between intra-AS and inter-AS TE is then established. The interaction between them is analyzed and both the decoupled and integrated approaches are presented.

In [24], several possible algorithms for routing and scheduling that allow coexistence of QoS and best- effort flows are presented. The network algorithm takes into account state imprecision in routers, maxmin bandwidth allocation, and existing link state information.

Most of the contributions presented in the research work we studied are in the area of middleware implementation and do not provide an analytical evaluation of communication resource requirements. Our objective is to propose a solution for accommodating real-time traffic on IP networks and, furthermore, develop a method for the analytical estimation of the network bandwidth required by real-time traffic.

3. Identified issues

Distributed control systems are an integral part of the industrial automation domain. Their functionalities include data acquisition, monitoring and control of industrial processes. While the majority of the control systems are usually located within more confined area (e.g. plant area, company local network) and communications are usually performed using local area network (LAN) technologies that are typically reliable and high-speed, other are geographically distributed (e.g. SCADA systems) and need long-distance communication systems such as the Internet.

We focus on the communication issues of distributed control systems that are deployed in the companies' local IP networks. Usually, these networks are managed by the companies and the nodes (hosts, switches, routers, servers) are configured and administered by a company internal authority. Such a network has to accommodate two broad categories of traffic: non-real-time and real-time. Non-real-time traffic can adjust to changes in delay and throughput and is generated by applications that include common Internet-based applications, such as file transfer, electronic email, remote logon, network management, and Web access. Real-time traffic does not easily adapt, if at all, to changes in delay and throughput and have requirements that include beside delay and throughput, delay variation and packet loss.

In addition, these corporate networks may be connected to strategic partner networks and to the Internet, thus, making more use of Wide Area Networks (WANs) and Internet to transmit their data to remote stations.

Most control applications must satisfy real-time and safety constraints. A very important parameter in real-time environments is the system response time, defined as the time between the occurrence of an event and the corresponding response. In distributed systems, message delivery time, has a large influence on the system's response time. Network protocols must incorporate control mechanisms for message delivery time in order to guarantee maximum delivery time for control messages. These mechanisms assume a deterministic network behavior, which permit a-priori evaluation of maximum message delivery time.

When measuring the performance of a real-time communication system, the following parameters are taken into consideration [25]:

- Deadline miss rate (fraction of all messages that are delivered to late at the destination)
- Delay jitter (the variation of message delays)
- Loss rate (fraction of all messages that are dropped on the route from source to destination)

For hard real-time applications, deadline misses are not acceptable. Moreover, message response time must be guaranteed a-priori. Delay jitter may not cause serious problems as long as deadlines are satisfied. In the case of soft real-time applications, deadline misses are tolerable to some extent, but, under some particular conditions, delay jitter may have negative effects. In the case of distributed control systems traffic, one has to deal with both hard and soft real-time requirements. For these reasons, the goal is to minimize both message response time (end-to-end delay) and delay jitter.

In the case of using an IP network for real-time communication, some important issues may arise. First of all, a maximum response time for packets has to be guaranteed. This can be a serious problem knowing that IP networks function on a best-effort basis.

Another communication issue is message delivery efficiency. Data transmitted through the network in distributed control systems are quite different compared to data transmitted by usual applications that generate traffic in IP networks. Control applications use short, unstructured data (e.g. digital signal values). Process control data is generated, mostly, at well determined periods of time. The majority of supervision and control functions involve data acquisition, processing and storage, visualization of process status and command issuing, which require a short reaction time.

Control applications include different automation and computing devices, which are interconnected. In order to insure interoperability, the communication protocol must allow uniform and transparent access to system's resources and it must be simple enough to allow implementation on devices with limited computing resources.

Last but not least is the issue of real-time and non-real-time traffic coexistence. In the case of remote process control it is quite possible to have both traffic (real-time) generated by the control system and traffic (best-effort) generated by other applications (e.g. office automation) that run in the same network. Network bandwidth has to be managed in order to insure real-time requirements for control traffic and, at the same time, to insure fair treatment for the best-effort traffic.

The objective of this research is to take a new approach in solving some of the following communication issues:

- Guarantee packet delivery time in IP networks in the presence of both real-time and non-real-time traffic
- Ensure predictability of network behavior
- Ensure transmission efficiency of process control data
- Provide device interoperability and uniform access to process control data

To address these problems, a reservation-based communication system architecture for distributed control applications on best-effort networks is described in the next section. As part of the solution, a control traffic model and a method for the estimation of the required network bandwidth are defined.

4. The data-flow model as support for a reservation-based communication architecture

4.1. The reservation-based communication architecture

To provide a comprehensive communication system architecture based on IP infrastructures that meets the challenges of quality of service provisioning for industrial control applications, three major components are integrated:

- Industrial control applications and processes

- A middleware (service manager) between the application and the protocol driver; this middleware closely interacts with the Internet protocol stack
- A network infrastructure based on IPv4 or IPv6 protocol

The first component of the system architecture represents quality of service demanding applications that use a QoS API to send requests to the service manager. These applications generate periodic and aperiodic real-time traffic. The traffic is characterized by packet size, transmitting data rates, priority, and accepted latency.

The middleware bridges the industrial applications and the underlying network systems by dispatching the application requests and returning status and feedback from the underlying system to the application. Examining the application requests and the available network resources, the middleware selects a provisioning service or service level, maps the application QoS to network-specific quality of service, and initiates resource reservation or renegotiates the parameters with the application before the flows' source starts to generate any packets.

The following components are integrated in the middleware:

- Traffic QoS specification
- QoS negotiation
- Traffic and QoS monitoring
- Resource reservation
- Data transfer

The middleware components are depicted in Fig. 1 and will be described as follows.

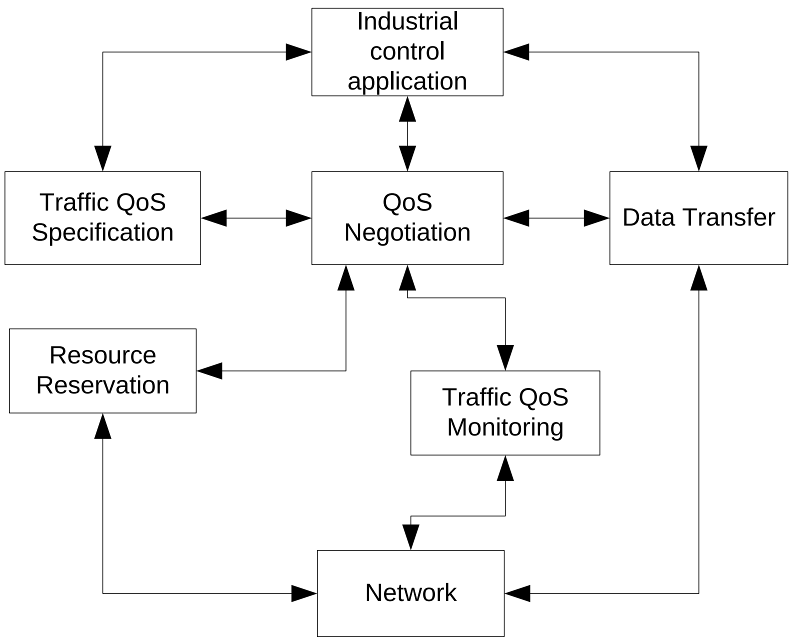


Figure 1. System architecture – components

4.1.1. Traffic QoS Specification and QoS Negotiation modules

An application which wants to set up a connection in order to transmit packets to another application in the network uses the means of the traffic QoS specification to set up a reservation request first. This module is a generic API so that an application demanding quality of service is isolated from the complexity of the provisioning services.

The application defines its generic QoS specification in terms of traffic profile which is composed of parameters that characterize the traffic stream or session (source IP address and port number, destination address, transport protocol) and parameters that define quantitatively the network performance requirements (transmitting rates, message size, transmission deadlines, latency), which can be specified using maximal, average and minimal values.

The traffic QoS specification module contains a set of rules for converting the traffic characteristics to parameters in the underlying message model.

Based on the input from the QoS specification module, the QoS negotiation module is responsible for authorizing the request and check if the network is able to support the new connection interacting with the resource reservation module for resource allocation. The goal of this module is to provide optimal quality of service with respect to critical parameters and previous requests.

Application's requests for quality of service parameters can be solved in two ways: positive, in case the resource reservation module sends a positive acknowledge to the QoS negotiation module that there are enough resources in the network to satisfy the request and the reservation is set along the path, and negative. In case of a negative notification, the application may invoke the QoS negotiation module in order to find what resources and services are available in the network and to adjust the quality of service requirements and start a new negotiating procedure.

4.1.2. Traffic and QoS Monitoring module

In this module components are included for monitoring network resources (available bandwidth, average utilization of a link, delay, jitter) and quality of service related statistics from routers (queue length, number of conforming/exceeding packets in bytes, number of dropped packets, CPU utilization). It also signals significant changes in resource availability.

When an application establishes a network traffic stream, this module starts collecting its performance. It collects data from traffic stream, including quality of service specification, connection times, transmission rates and delays, and communicate the quality of service parameters to the QoS negotiation module in order to determine if there is any quality of service violation. All collected data is stored into a management information base.

4.1.3. Resource Reservation module

The resource reservation module is the ultimate authority for the resource handling in this architecture. Its main building blocks are admission control and reservation setup. Admission control implements request authorization by checking if the network is able to support the flow and the decision algorithm that nodes use to determine whether a new flow can be granted the requested quality of service with/without impacting earlier guarantees. For these tasks it closely interacts with the main entity, the resource reservation protocol.

Resource ReSerVation Protocol (RSVP) [15] is used for resource reservation signaling. It is designed to enable the senders, receivers and routers of communications sessions to communicate with each other to reserve resources for new flows at a given level of QoS. On the other hand, the reservation protocol is responsible for maintaining flow specific state information at the end nodes and at the nodes along the path of the flow.

RSVP requests result in resources being reserved in each node along the data path. Given below are the main attributes of this protocol: it requests resources in only one direction (it treats a sender separately from a receiver, although the same application might be running at both the sender and the receiver); is receiver-oriented (the receiver of a data flow initiates and maintains the resource reservation used for that flow); RSVP itself is not a routing protocol, but it is designed to work with the existing routing protocols; RSVP supports both IPv4 and IPv6.

To make a resource reservation at a node [26], our RSVP daemon uses the admission control mechanism. If check fails, the RSVP returns an error notification to the QoS negotiation module that originated the request. If checks succeed, the RSVP daemon sets the parameters.

4.2. The data flow model

In order to make an analytical evaluation of the traffic generated by the distributed control system, it is required to classify this traffic and then, based on the identified types, to define the traffic model.

4.2.1. Traffic classification

Control traffic is generated by data exchanged between the control applications and industrial devices, such as:

- Values obtained through data acquisition, with a well-defined frequency (e.g. temperature in an oven, liquid level in a tank, engine state – started/stopped, etc.)
- Commands generated at known periods of time
- Operator commands (e.g. start/stop engine, increase oven temperature to 200 degrees, etc.)
- Process events, alarms, alerts, etc.

The previous categories of data generate periodic and aperiodic network traffic, with real-time constraints.

4.2.2. Model definition

Distributed real-time applications (control applications are in the same category) are quite often modeled as chains of task and messages [27][28] that are executed on the distributed system’s active resources (e.g. processors, networks). As example, consider an application that monitors the temperature and humidity in a room. Temperature and humidity variables are measured by a number of sensors with the same periodicity of 30 minutes. The sensors send their data to the process computer. This computer places the values into packets and sends them to the monitoring application. Assuming that there are two temperature and two humidity sensors (we consider two sensors of each type for redundancy), we can model the room monitoring application as four periodic chains of tasks and messages, one for each sensor. A chain will be composed of a task that reads the temperature form the environment and sends it to the process computer through a message. The second task that runs on the process computer; it receives the message from the sensor and it forwards the measured value to the monitoring task in a predetermined format, as a message. The monitoring task runs on a remote site. Fig. 2 shows the model of the room monitoring system with chains of tasks and messages (in black). The message-based model of the Internet communication, in the case of distributed real-time systems, is quite inefficient due to the large overhead introduced by the Internet protocols. As for the feasibility analysis, this model generates overestimated requirements on communication due to the large number of messages that influence each other.

In this context, we propose a model that solves the issues of the message-based model. To model the control traffic, we introduce data flows.

Definition. *A data flow is the sum of all packets sent through the network that have the same source, destination, content and periodicity.*

Traffic between control applications and devices connected to the process is a sum of periodic and aperiodic data flows. In our previous example, the packets containing temperature and humidity values create a data flow. In Fig. 2, the alternative model with data flows is depicted in blue. Another example is shown in Fig. 3, where data flows are established between two control applications connected to remote industrial processes, through an IP network.

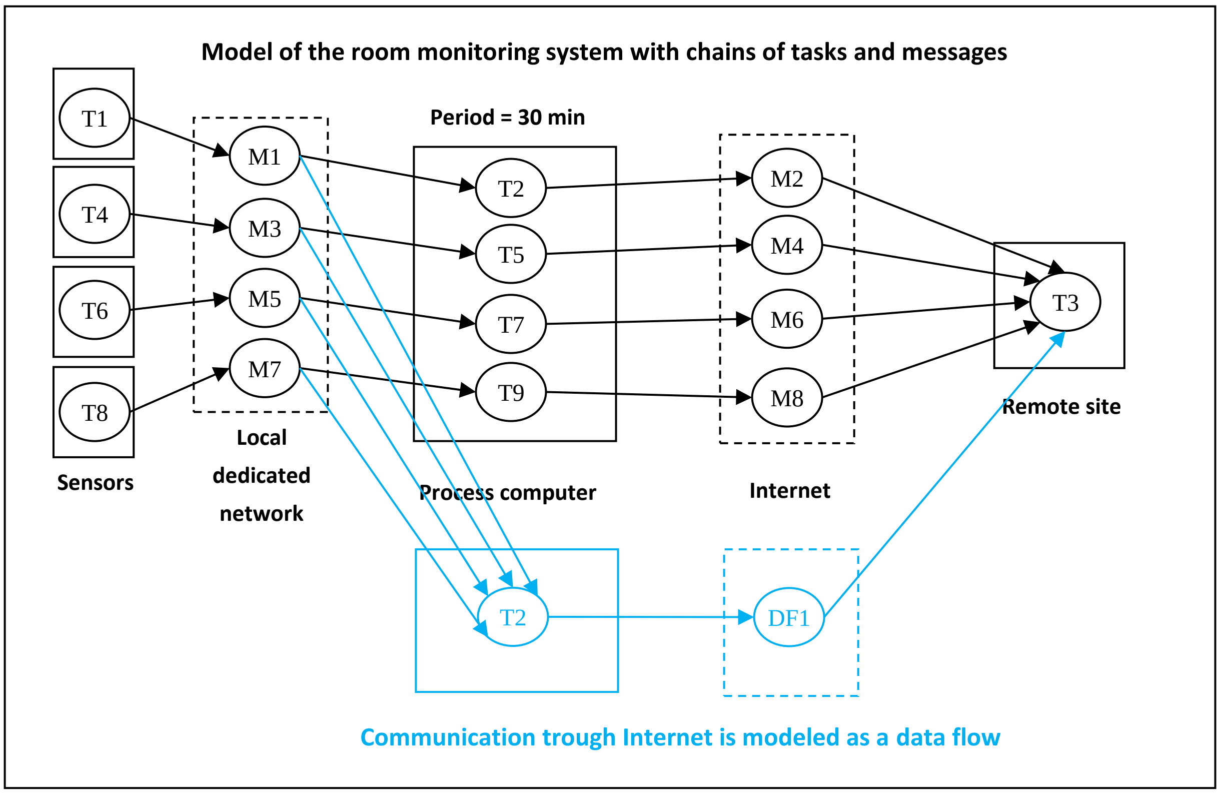

Model of the room monitoring system with chains of tasks and messages
Period = 30 min
T1
T4
T6
T8
M1
M3
M5
M7
T2
T5
T7
T9
M2
M4
M6
M8
T3
Sensors
Local dedicated network
Process computer
Internet
Remote site
T2
DF1
Communication trough Internet is modeled as a data flow

Figure 2. Data flows between control applications

Periodic data flows include values obtained through data acquisition, control commands, which occur at well defined periods. Aperiodic data flows include commands issued by the application operator, high priority alerts and event signals. A number of parameters are identified for each data flow type.

The parameters for aperiodic data flows are:

- Source
- Destination
- Packet size
- Priority (importance)
- Content (process control data included in the flow)
- Required packet delay (or response time)
- Transmission deadline

Periodic data flows have two more parameters with respect to aperiodic data flows, the *inter-release period* and the *phase* (the release time of the first packet in the data flow).

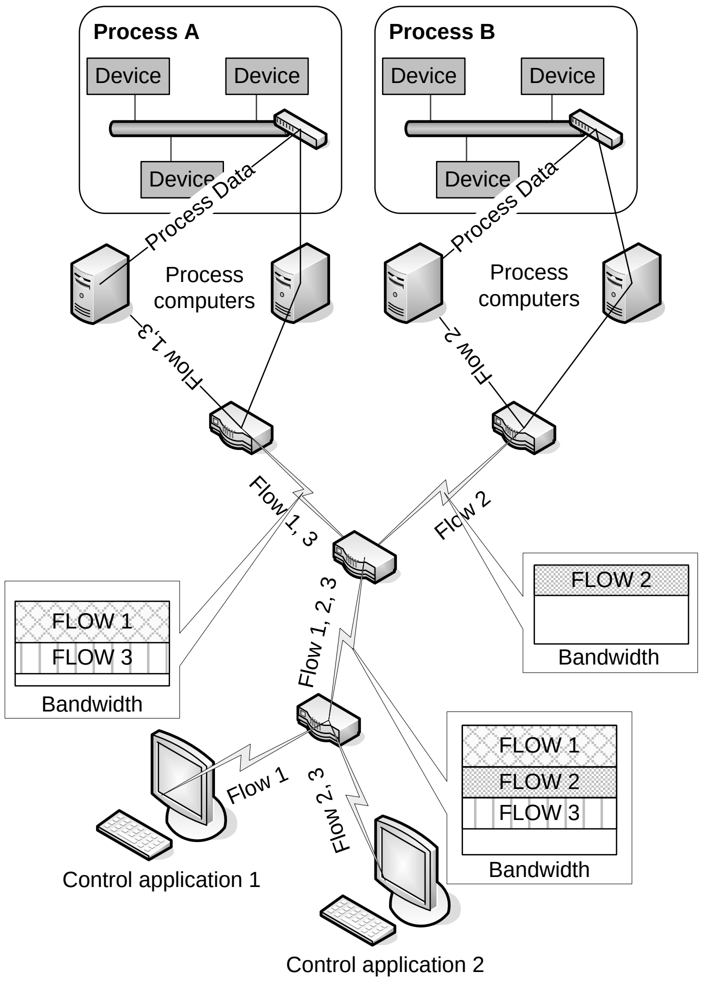


Figure 3. Data flows between control applications

It is a common practice in real-time task modeling to assume that aperiodic tasks have a minimum inter-release period, which is given by process related parameters. Because all tasks are considered periodic, scheduling and feasibility analysis are simplified. For the same reasons, we choose to make the minimum inter-release period assumption for aperiodic data flows.

A periodic data flow is formally defined as an n-tuple, as follows:

$$DF = (\varphi, T, P, r, D, l, Src, Dest, Content) \tag{1}$$

The components of the n-tuple are the data flow parameters. φ is the data flow's phase, *T* is the inter-release period for periodic data flows and the minimum inter-release period for aperiodic data flows. *P* is the priority, *r* is the required packet delay or response time, *D* is the transmission deadline and *l* is the size of a data flow packet. The last three components are the source (*Src*), destination (*Dest*) and content description of the data flow (*Content*).

The Internet protocol suite is optimized to deliver large packets of data in a best-effort manner. Process control messages, on the other hand, are very short (from a few bytes to hundreds of bytes); they have periodic occurrence and real-time constraints. If very short periodic messages are packed and released in an IP network, the protocol overhead is very large compared with the payload data. Because messages with short period of occurrence (e.g. seconds, milliseconds) can generate large amounts of traffic although few effective data is transmitted, one can conclude that the network is inefficiently utilized for data transmission.

To optimize the transmission of control packets in IP networks, we adopt the strategy of aggregating control data from different process devices into the same data flow.

To organize control data into larger data flows, the following parameters must be considered:

- Data acquisition periodicity, assuming that devices which perform data acquisition with the same period read the sensor values virtually at the same time
- Data priority
- Data source and destination

By performing data flow aggregation, the data flow packets contain larger amounts of effective data, hence increasing the efficiency of control data transmission. To achieve a correct data flow aggregation the respective data flows have to be synchronized (their phases are equal). It is not enough to have the same release period. Data has to be produced and transmitted at approximately the same time. Usually, it is not a difficult problem to achieve the synchronization requirement. For this reason, we will assume that aggregated data flows can be easily synchronized.

To aggregate data flows we propose the Aggregate Data Flows (ADF) algorithm. First, an array of data flows (*DF*) is created by defining a data flow for each piece of process data. The parameters are set for each data flow. The array of data flows is then sorted by data flow period. To aggregate data flows that have similar characteristics, each periodic data flow is compared to all other periodic data flows that have the same period. If the data flows have the same source and destination, they are put together in the same data flow. The aggregated data flow (see Procedure 1 – Merge Data Flows) will contain data from both initial data flows, they will have the highest priority and the smallest deadline of the two data flows. The first data flow will be substituted by the aggregated data flow and the second data flow will be deleted.

Algorithm 1: Aggregate Data Flows (ADF)

```
count = 0;
Foreach process_data_value
{
     DF[i] = Create_data_flow ();
     Set_data_flow_parameters ( DF[i] );
     i++;
}
Sort_data_flows_by_period ();
For ( i=0; i < count-1; i++ )
{
     If Periodic (DF[i]) and !Marked_for_delete (DF[i])
     {
          j = i+1;
          While ( DF[i].period == DF[j].period )
          {
               If ( Periodic ( DF[j] ) )
               {
                    If ( DF[i].src == DF[j].src )and( DF[i].dest == DF[j].dest )
                    {

                          Merge ( DF[i], DF[j] );
                          Mark_for_delete ( DF[j] );
                     }
               }
               j++;
          }
     }
}
Foreach DF
{
     If (Marked_for_delete ( DF[i] ))
     {
     Delete ( DF[i] );
     }
}
```

Procedure 1: Merge Data Flows (MDF)

Input: DF[i], DF[j]

Output: DF[i]

```
DF[i].content = DF[i].content + DF[j].content;
DF[i].size = DF[i].size + DF[j].size;
DF[i].priority =  max (DF[i].priority, DF[j].priority);
DF[i].deadline = min (DF[i]. deadline, DF[j]. deadline);
```

In this way, we obtain the specification of periodic data flows for a control system. For aperiodic data flows, which cannot merge with other flows, transmission efficiency is reduced. A solution for these flows, if their frequency of occurrence is high, is to reserve space for aperiodic data in periodic flows. This space (e.g. a few bytes) is used only if the aperiodic event takes place right before a periodic data flow packet is sent.

5. Response time analysis for real-time communication

In real-time systems, it is essential to guarantee response times. The system architecture proposed in this chapter uses an IP network as a communication infrastructure. The main challenge in this case is to guarantee real-time constraints on a best-effort communication infrastructure. For solving this problem, we use a network bandwidth reservation mechanism.

The bandwidth reservation mechanism requires the estimation of network bandwidth for the real-time traffic generated in the system. For this purpose, we adopt a method commonly used in the case of real-time tasks, the response time analysis using a priority-based scheduling algorithm. This method allows the calculation of the worst case response time of real-time tasks [28][29] as the solution of a recursive equation. We adapted the classic method of response time analysis to our data flow model. This allows the computation of response time for each data flow.

The most important time restriction in our case is that the response time of any data flow should not exceed its deadline. To obtain an estimation of the network bandwidth we have to find the minimum bandwidth value for which all data flows meet their deadlines, in the worst case.

In our adaptation of response time analysis, we assume that each data flow corresponds to a “task” and the network corresponds to the “processor”, which has to be shared between all “tasks” in the system. Priorities are assigned to data flows in a rate-monotonic fashion [30], that is, the priority of a data flow is given by its period parameter. This way, a data flow that has a lower period will have a higher priority. Data packets will inherit the priority of the data flow to which they belong. The packet with the highest priority will be transmitted first.

Periodic and aperiodic data flows are taken into consideration for scheduling. Aperiodic data flows are considered to have a minimum inter-release period. The next step is to compute the response time for each data flow. The data flow system is feasible if, for each data flow, the response time is less than its transmission deadline ($r < D$). In this work we consider that transmission deadline for a data flow is equal to its inter-release period ($T = D$). By obtaining the appropriate values for the response time of all data flows, in the worst case, the value for the required network bandwidth can be derived. The bandwidth value obtained is used to make resource reservations. In this way, it can be guaranteed that actual response time for each data flow will be less or equal than the computed response times.

It is considered that the worst case response time for a data flow happens when a packet has to wait for the transmission of packets that belong to all data flows with higher priority and for one packet with lower priority, but with the largest transmission time. It is also assumed that all data flows start at the same time.

In order to compute the data flow response time (r_i) the following variables are taken into consideration:

- The delay caused by the devices found on the network path (t_{delay})
- The transmission time of packets that belong to the data flow (C_i)
- The data flow's inter-release period (T_i)
- The data flow's priority (P_i)
- The transmission time of packets that belong to data flows with higher priority
- Maximum transmission time of packets that belong to data flows with lower priority
- The number of hops from source to destination (nHops)

Response time for each data flow is computed using the following equations:

$$\begin{cases} r_i^0 = t_{delay} + nHops * (C_i + \sum_{P_j > P_i} C_j) \\ r_i^{t+1} = t_{delay} + nHops * (C_i + \sum_{P_j > P_i} \left\lceil \frac{r_i^t - C_i}{T_j} \right\rceil * C_j + Max\{C_k \mid P_k < P_i\}) \end{cases} \quad (2)$$

The response time is obtained through iteration, until the computed response time is approximately equal to the response time computed in the previous step ($\left\lceil r_i^t - r_i^{t+1} \right\rceil < \varepsilon, \varepsilon \to 0$). Two important conditions that have to be imposed:

- All response times have to be less than the corresponding deadlines ($r_i<D_i$)
- Network utilization has to be less than 100% ($\sum \frac{C_i}{T_i} < 1$)

The bandwidth value is at first equal to $B_0=\sum \frac{C_i}{l_i}$ and it is gradually increased from with 10% while the response time and utilization are computed, until these requirements are met. The bandwidth value used for the reservation (B_r) is the minimum bandwidth value for which the requirements are met.

In the first iteration, the response time of a data flow is computed by summing up the transmission time, the delay caused by the network devices such as switches and routers, and the transmission time of all other data flows that have higher priority. In subsequent iterations, the response time equation has two new components: the maximum transmission time of packets that have lower priority and the sum of transmission time of all packets of higher priority that are likely to be released while the packet is being transmitted through the network. The number of packets with higher priority that influence the response time of the data flow equals the number of packets that are released in the response time of a packet from the data flow. To decrease the number of iterations, the time interval when the actual packet is being transmitted (C_i) is subtracted from the response time, as in that time interval packets from other data flows cannot be transmitted.

The transmission time for packets that are included in a data flow is computed as follows:

$$C = \frac{Packet\_length}{Bandwidth} \tag{3}$$

The delay caused by network devices can be approximated by using a mean round-trip time of a probe packet sent on the same route on which the bandwidth reservation will be made.

As the response time is computed recursively, the computation time could be a problem. In our case, because bandwidth requirements are assessed and reservations are made before starting the control system, a larger computation time is not an issue.

The main disadvantage of the response time analysis method is that sometimes it does not converge to a solution. In this case, we have to increase ε and give an approximate solution.

6. An experimental reservation-based control system

To validate the system architecture, the data flow model and the method of estimating network bandwidth described earlier, a prototype of a distributed control system, was developed. The prototype includes the communication middleware, the industrial application and the industrial device simulator (for the provision of industrial process data).

The communication middleware runs on a network infrastructure based on Internet protocol suite, with IPv6 as a network protocol. The adopted solution is to use the IPv6 Traffic Class and Flow Label fields. The Traffic Class field enables a source to identify desired traffic-handling characteristics of each packet relative to other packet from the same source. The intent is to support various forms of services. In case of IPv6 standard [31], a flow is defined as a sequence of packets sent from a particular source to a particular destination for which the source desires some special handling by the intervening routers. From the source's point of view, a flow is just a sequence of packets that are generated from a single application instance at that source and have the same transfer service requirements. From the router's point of view, a flow is a sequence of packets that share attributes that affect how these packets are handled by the router. In principle, all of a user's requirements for a particular flow could be defined in an extension header and included in each packet, but for our implementation, we leave the concept of flow open to include a wide variety of requirements and adopt the flow label, in which the flow requirements are defined before traffic generation and a unique flow label is assigned to the flow.

The RSVP module is designed as a state machine. The objects defined in this module represent:

- RSVP sessions
- State information extracted from PATH message and information from RESV message
- Reservations installed in an outgoing interface
- Information about a previous hop in a session, i.e. the last reservation that has been sent to this hop

For each RSVP session, all relevant information is bundled and the destination address and port is saved. From each PATH message all relevant information is held, i.e., the sender's address and traffic specification, routing information. For each reservation requested from a next hop, reservation specification is held, i.e., the FlowSpec, which determines the amount of resources that are requested, depending on the service class.

The industrial control application uses all the facilities offered by the communication middleware and implements the following functionalities:

- Remote process control and visualization
- Input and output data flow definitions for devices participating in the industrial process
- Control data flow through commands sent to devices connected to processes
- Specification and negotiation of resources needed for communication with other devices
- Receive and process data flows from industrial devices
- Register data flow delay time

The operator can visually create the diagram of the industrial process, by dragging the symbols of different types of devices on the control board. Next, the operator has to specify input data flows (data received from devices connected to the process) and output data flows (commands sent to devices) in order to establish communication parameters.

After the definition of data flows, the negotiation process for resources starts. Input and output data flows are analyzed and, as a result, bandwidth needed to satisfy real-time communication constraints is computed. The application sends a query asking for the available bandwidth and round-trip time to destination process. The response time can be guaranteed only for the data flows having the period less than the delay caused by the network devices (e.g. switches, routers). If the available bandwidth is insufficient, data flows having the smallest period are deleted, data is recomputed and application begins the resource reservation process. After the negotiation and reservation process, the application can start to send and receive data flows.

```
<Flow>
      <ID> data_flow_ID </ID>
      <SrcIP> source_IP </SrcIP>
      <DestIP> destination_IP </DestIP>
      <Per> data_flow_period </Per>
      <Pri> data_flow_priority </Pri>
      <Name> data_flow_symbolic_name </Name>
     <Content>
      XML_content_specification
     </Content>
 </Flow>
```

Figure 4. Data flow specification in XML

The industrial device simulator sends periodical data flows (requested by the control application) containing process values randomly generated from a predetermined range and receives periodical data flows representing commands from the control application for devices connected to the process. Devices cannot negotiate resource reservations for generated data flows nor to specify quality of service parameters. The control application connected to these devices is responsible for the negotiation and bandwidth reservation.

An important issue encountered during the implementation of both the industrial control application and the device simulator is the specification of data flows. In order to assure the device and application interoperability, data flow parameters and content are specified using XML. In this way, messages between applications and devices are interpreted easier and the access to process and control data is uniform. Fig. 4 shows an example of a periodic data flow specification in XML.

7. Experimental evaluation

Two sets of experiments are conducted. First, a simulator is used to validate the proposed method for network bandwidth estimation. Second, tests are performed using the implemented control system prototype, which was deployed on the experimental infrastructure.

7.1. Estimation of network bandwidth

For the first set of experiments, we measure the response time, jitter and packet loss for multiple periodic real-time data flows, which are released in a simulated IP network. The transport protocol is UDP, because it is more appropriate to real-time traffic. The main objective of these experiments is to check if the computed network bandwidth value guarantees the required response time and low jitter for all data flows.

The simulation study was performed on Network Simulator (NS-2) [32], version 2.33. The simulation results were evaluated for different scenarios using the topology depicted in Fig. 5.

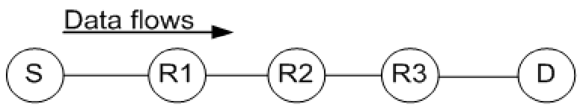


Figure 5. The topology used for simulations

The topology consists of 5 nodes. These nodes are connected with full-duplex bidirectional links. All links have the same available bandwidth and propagation delay. It is assumed that per link delay is negligible. Constant-Bit-Rate (CBR) agents were attached to the source node (S) and used to generate periodic, fixed size packet traffic in the network. User Datagram Protocol (UDP) was used as transport layer protocol to minimize the overhead of establishing a connection. Five periodic data flows were defined, having the same source (S) and destination (D). The parameter settings are summarized in Table 1.

Table 1. Parameter settings for periodic real-time data flows

Flow	**Period (ms)**	**Packet size (B)**
F1	*10*	*300*
F2	*120*	*300*
F3	*50*	*300*
F4	*75*	*300*

F5	*520*	*300*

Two scenarios were simulated. Measurements were made to compare data flows' response times with the corresponding deadlines and to observe to what extent the jitter affects the response time of packets.

In the first scenario the network bandwidth was set to the minimum value which can accommodate all defined data flows (379 Kbps). Results analysis revealed that the average measured response times were acceptable in the case of data flows which had larger periods, but for the other data flows response times were very often greater than the corresponding deadlines. Jitter measurements showed that even if the average value was quite small, the maximum value was very large, approximately equal to the measured minimum response time. For this scenario no packets were lost.

For the second scenario, equations (2) were used to derive the maximum bandwidth needed by the set of data flows in order to satisfy the deadlines. The computed maximum bandwidth was 2106 Kbps. As expected, a considerable difference can be observed between the measured maximum response time and the maximum computed response time. This difference is due to the fact that the worst-case scenario does not occur during simulation time, thus the resulting network utilization is low. All the deadlines were satisfied and the average delay jitter is very small for the flow with the largest period. There was no packet loss.

For both scenarios, measured values can be found in Tables 2-5 and the comparison between data flows in terms of response time and jitter are shown in Fig. 6-9.

Table 2. Response times (1st scenario)

Flow	**Response time (ms) - measured**		
	Maximum	**Average**	**Minimum**
F1	50.66	28.87	25.33
F2	50.66	33.01	25.33
F3	50.66	32.98	25.33
F4	50.66	30.48	25.33
F5	50.66	33.03	25.33

Table 3. Response times jitter (1st scenario)

Flow	**Response time jitter (ms) - measured**		
	Maximum	***Average***	***Minimum***
F1	25.33	3.22	0
F2	18.68	3.58	0
F3	19	4.33	0
F4	24	4.41	0
F5	25.33	3.59	0

Table 4. Response times (2nd scenario)

Flow	**Response time (ms)**

	Measured			*Computed*
	Maximum	***Average***	***Minimum***	
F1	6.078	3.103	3.039	9.12
F2	6.078	3.839	3.039	68.4
F3	6.078	3.870	3.039	27.36
F4	6.078	3.447	3.039	41.04
F5	6.078	3.724	3.039	86.64

Table 5. Response times jitter (2nd scenario)

Flow	**Response time jitter (ms) - measured**		
	Maximum	***Average***	***Minimum***
F1	3.039	0.114	0
F2	2.279	0.547	0
F3	2.279	0.484	0
F4	3.039	0.815	0
F5	3.039	0.002	0

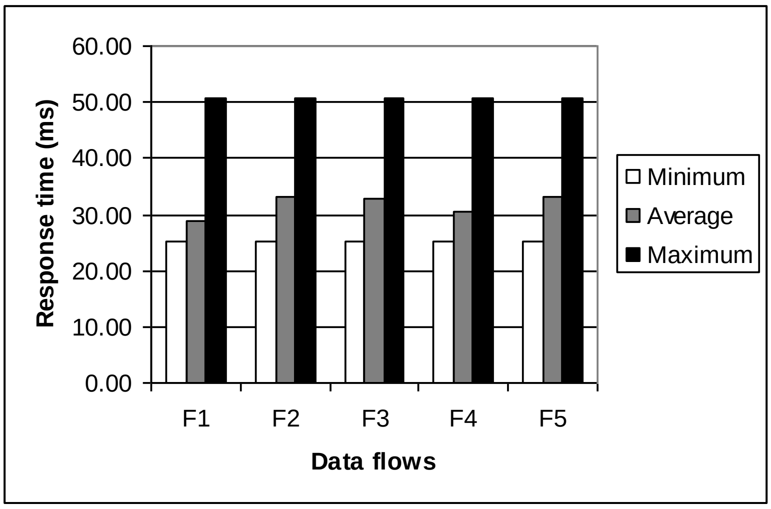


Figure 6. Response times (1st scenario)

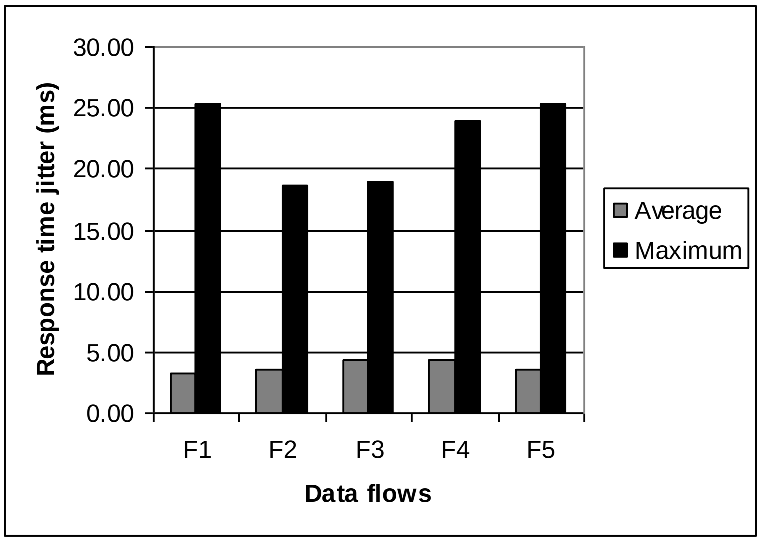


Figure 7. Response times jitter (1st scenario)

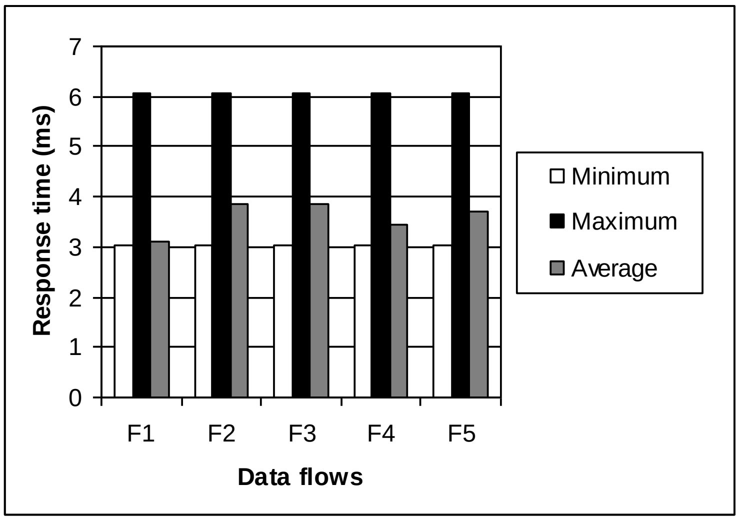


Figure 8. Response times (2nd scenario)

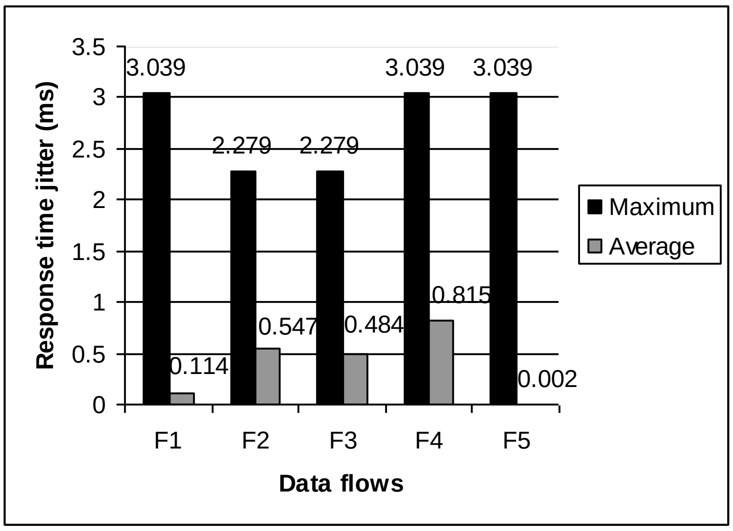


Figure 9. Response times jitter (2nd scenario)

7.2. Tests performed using the experimental control system

To test the distributed control system prototype, two PCs connected in a local network were configured as traffic source and destination. A static route consisting of another two PCs which played the role of routers was established between these nodes. The network infrastructure was based on IPv6 protocol. The communication middleware, the control application and the device simulators were deployed on the test infrastructure.

A process schema containing monitoring elements connected to two data flows was specified in the control application. The first data flow (Flow 1) has a 2 seconds period and 270 byte packet size. The second data flow (Flow 2) has a 0.5 second period and the same packet size. After starting the remote control application and the device simulators, response time for all packets was measured.

Measurements were made in two cases. In the first case, the communication middleware was used to make network bandwidth reservations before starting the traffic. In the second case, no reservations were made for the real-time traffic. For both data flows, response time measured during tests was less than the maximum allowed response time, in the case of reservations, presented in Table 6.

Table 6. Measured response time for experimental data flows with reservations

	Flow 1	***Flow 2***	***Flow 1***	***Flow 2***
Computed bandwidth	9 kbps			
Available bandwidth	100 kbps		64 Mbps	
Measured t_{delay}	1.5 ms		0.65 ms	
Computed maximum response time	0.745 s	0.497 s	0.744 s	0.496 s
Measured response time	0.685 s	0.372 s	0.677 s	0.367 s

The measurements showed that the proposed system architecture, traffic model and method of data flow scheduling are able to satisfy the control system's requirements and guarantee a maximum delivery time. They also showed that the analytical evaluation of the response time is an upper limit to the measured time parameters.

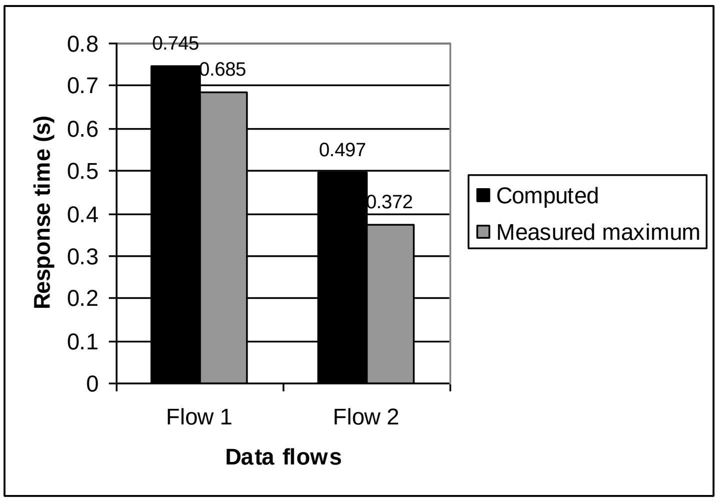
If no reservations were made, for both data flows, measured response time fluctuated between a minimum of 0.367 seconds and a maximum of 0.617 seconds, as can be observed in Table 7. Packets of Flow 1 have the same priority on the network as packets of Flow 2, even though Flow 2 requires a better response time. Real-time requirements were not satisfied, because for Flow 2 the maximum measured response time was greater than the computed maximum response time.

Table 7. Measured response time for experimental data flows without reservations

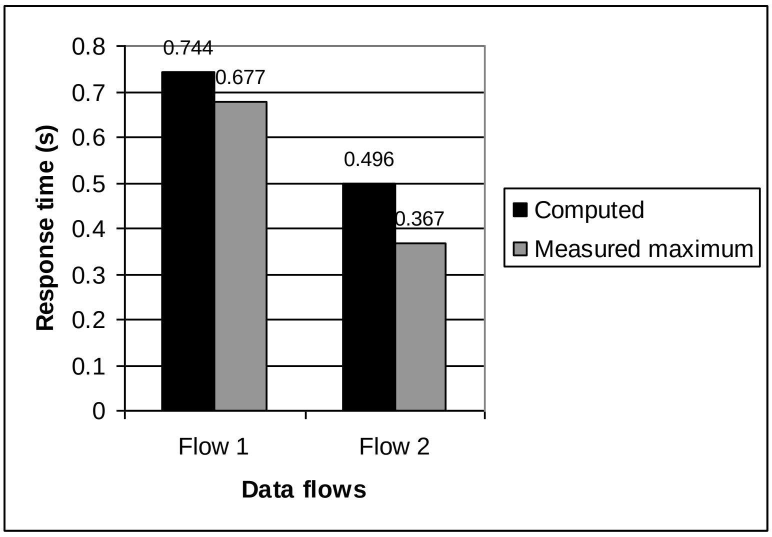
	Flow 1	***Flow2***
Computed bandwidth	9 kbps	
Available bandwidth	100 Mbps	
Measured t_{delay}	0.4 ms	0.4 ms
Computed maximum response time	0.744 s	0.496 s

Maximum measured response time	0.617 s	0.617 s
Minimum measured response time	0.367 s	0.367 s

Fig. 10 and Fig. 11 show charts that compare the computed response time for the two data flows with the measured response time, on both test scenarios.



(a) Available bandwidth = 100kbps



(b) Available bandwidth = 64Mbps

Figure 10. Response time measurements using reservations

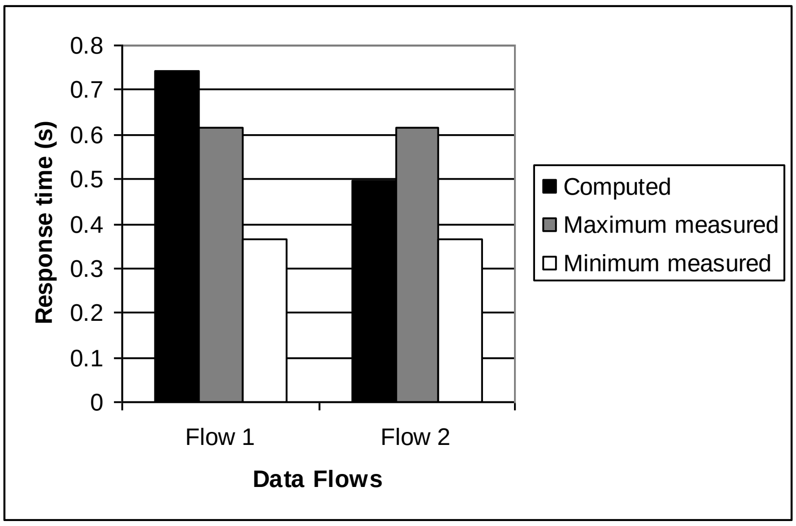


Figure 11. Response time measurements without using reservations

8. Conclusions

This chapter presents an approach in solving the control of network delivery time and data delivery efficiency in distributed control systems, on IP infrastructures. The data flow traffic model provides the basis for communication scheduling, and a reservation-based communication system architecture, and is the support for the rate-monotonic response-time analysis method used for computing the required network bandwidth. The solution uses Integrated Services/RSVP and the facilities of IPv6 protocol as support for real-time communication.

The experimental results show that the implemented prototype satisfies the real-time constraints. This proves the validity of the proposed communication model and the method for computing the required network bandwidth. Moreover, the analytical evaluation of response time demonstrated to be an upper limit for measured delivery time.

References

[1] Lu, Y. Lu, T. Abdelzaher, J. Stankovic, S. Hyuk Son, "Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers", IEEE Transactions on Parallel and Distributed Systems, vol. 17, no. 9, Sep. 2006, pp. 1014-1027

[2] R.E. Schantz, J.P. Loyall, C. Rodriguez, D.C. Schmidt, Y. Krishnamurthy, I. Pyarali, "Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware", Proceedings of Middleware 2003, ACM/IFIP/USENIX international middleware conference, Rio de Janeiro , Brazil, 2003, pp. 374-393

[3] A. Mancina, D. Faggioli, G. Lipari, J. N. Herder, B. Gras, A. S. Tanenbaum, "Enhancing a dependable multiserver operating system with temporal protection via resource reservations", Real-Time Systems, 43(2), 177-210, 2009

[4] Insik Shin, Arvind Easwaran, Insup Lee, "*Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors*", Euromicro Conference on Real-Time Systems, 2008

[5] Enrico Bini, Giorgio C. Buttazzo, Marko Bertogna, "*The Multi Supply Function Abstraction for Multiprocessors*", 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2009

[6] B. Ward, J. Herman, C. Kenna, J. Anderson, "Making Shared Caches More Predictable on Multicore Platforms", Euromicro Conference on Real-Time Systems, 2013

[7] T. Skeie, S. Johannessen, O. Holmeide, "Timeliness of real-time IP communication in switched industrial Ethernet networks", IEEE Transactions on Industrial Informatics, Volume 2, Issue 1, Feb. 2006, pp. 25 – 39

[8] L. B. Fredriksson, “Controller area networks and the protocol CAN for machine control systems”, Mechatronics, vol. 4, no. 2, pp. 159-192, 1994

[9] S. Koubias and G. Papadoupoulous, “Modern fieldbus communication architectures for real-time industrial applications”, Computers in Industry, vol. 26, no.3, pp. 243-252, Aug. 1995

[10] A. Mok, X. Feng, D. Chen, “*Resource partition for real-time systems*”, Real-Time Technology and Applications Symposium, 2001, pp. 75–84

[11] Luca Abeni, Giorgio Buttazzo, “*Integrating Multimedia Applications in Hard Real-Time Systems*”, Proceedings of IEEE Real-Time System Symposium,Madrid, Spain, 1998

[12] L. Abeni, L. Palopoli, G. Lipari, J. Walpole, “Analysis of a Reservation-Based Feedback Scheduler”, IEEE Real-Time System Symposium (RTSS), Austin, Texas, 2002, pp. 71

[13] M. Wijnants, W. Lamotte, “Managing client bandwidth in the presence of both real-time and non real-time network traffic”, 3rd International Conference on Communication Systems Software and Middleware and Workshops, (COMSWARE), Bangalore, Jan. 2008, pp. 442-450

[14] Martínez Vicente, G. Apostolopoulos, F. J. Alfaro, J. L. Sánchez, J. Duato, "Efficient Deadline-Based QoS Algorithms for High-Performance Networks," IEEE Transactions on Computers, vol. 57, no. 7, Jul., 2008, pp. 928-939

[15] R. Braden, L. Zhang, S. Berson, S. Herzog, S. Jamin, “Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification”, RFC2205, September 1997

[16] J. Wroclawski, “The use of RSVP with IETF Integrated Services”, RFC2210, Sept. 1997

[17] S. Georgoula.; P. Trimintzios, G. Pavlou, K.H. Ho, “Heterogeneous real-time traffic admission control in differentiated services domains”, IEEE Global Telecommunication Conference, (GLOBECOM), Volume 1, 28 Nov.-2 Dec. 2005

[18] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss, “An Architecture for Differentiated Services”, RFC2475, Dec. 1998

[19] F. Kelly, R. Key, S. Zachary, “Distributed Admission Control”, IEEE Journal on Selected Areas in Communications, Vol. 18, Dec. 2000, pp. 2617-2628

[20] L. Breslau, E. Knightly, S. Shenker, I. Stoica, H. Zhang, “Endpoint Admission Control: Architectural Issues and Performance”, in Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication 2000, Stockholm Sweden, pp. 57-69

[21] G. Bianchi, F. Borgonova, A. Capone, L. Fratta, C. Petrioli, “Endpoint admission control with delay variation measurements for QoS in IP networks”, ACM SIGCOMM Computer Communication Review, vol. 32, no. 2, April 2002, pp. 61 - 69

[22] S.-A. Reinemo, F. O. Sem-Jacobsen, T. Skeie, O. Lysne, “Admission Control for diffServ Based Quality of Service in Cut-through Networks”, 10th International Conference on High Performance Computing (HiPC 2003), ed. by Timothy Mark Pinkston and Viktor K. Prasanna, pp. 118-129, Heidelberg, Springer. Lecture Notes in Computer Science

[23] K.-H Ho, M. Howarth, N. Wang, G. Pavlou, S. Georgoulas, “Two Approaches to Internet Traffic Engineering for End-to-End Quality of Service Provisioning”, 1st EuroNGI Conference on Next Generation Internet Networks - Traffic Engineering, Rome, Italy, 18-20 April 2005, pp. 135 – 142

[24] K. Nahrstedt, S. Chen, “Coexistence of QoS and Best Effort Flows - Routing and Scheduling”, Proceedings of 10th Tyrrhenian International Workshop on Digital Communications: Multimedia Communications, Ischia, Italy, Sept. 1998

[25] Liu, J. W.S., Real-Time Systems, Prentice Hall, 2000

[26] S. Shenker and L. Breslau, “Two Issues in Reservation Establishment”, ACM SIGCOMM '95 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Cambridge, 1995, pp. 14-26

[27] J.J.G Garcia, M.G Harbour, "*Optimized priority assignment for tasks and messages in distributed hard real-time systems*," Parallel and Distributed Real-Time Systems, 1995. Proceedings of the Third Workshop on , vol., no., pp.124-132, 25 Apr 1995

[28] K. Tindell, J. Clark, "Holistic schedulability analysis for distributed hard real-time systems." Microprocessing and microprogramming 40.2 (1994): 117-134.

[29] N. Audsley, A. Burns, M. Richardson, K. Tindell, A.J. Wellings, “Applying new scheduling theory to static priority pre-emptive scheduling”, Software Engineering Journal, pp.284-292, Sept. 1991

[30] C.L. Liu, J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”, Journal of the ACM, Vol. 20, No. 1, January 1973, pp.46-61

[31] R. Banerjee, The Internet Protocol version 6 (IPv6): issues and challenges, Technical Report, Computing Science Laboratory, Oxford University, Feb. 2002

[32] The Network Simulator - ns-2, http://www.isi.edu/nsnam/ns/

CHAPTER 4. MULTIPROCESSOR REAL-TIME SCHEDULING 1

Chapter 4. Multiprocessor real-time scheduling

1. Introduction

A scheduling strategy imposes a set of rules that are used to establish the execution order of all tasks that are ready for execution. One of the main tasks of a real-time system's designer is to configure the application on a given platform and find a proper scheduling strategy so that all tasks satisfy their time requirements (e.g. deadlines). In the case of real-time uniprocessor systems, several widely studied optimal algorithms find a feasible schedule for a given setup, such as RM, EDF [1] and LSF [2].

On the other hand, for multiprocessor real-time systems, the search space for a feasible scheduling solution is multi-dimensional. There are far more restrictions and therefore finding a feasible solution is much more complex [3]. Moreover, real world applications, are usually of high complexity, and can't always be modeled as independent task sets. In the case of distributed real-time systems, for example, applications consist of tasks with resource usage constraints and with data dependencies among them. Consequently, there are many more parameters which have to be considered before making a scheduling decision.

Most multiprocessor scheduling algorithms offer real-time guarantees only at very low resource utilization rates [4] compared to their uniprocessor equivalents, or they are very difficult to implement in real-world cases [5]. However, as multicore and distributed systems are becoming the typical computing platforms for a wide range of real-time applications, recent research efforts are mostly directed towards finding pragmatic solutions for multiprocessor systems.

In the case of multiprocessor platforms, in order to reduce the complexity of the real-time scheduling problem, one often used approach is to divide it into several sub-problems, such as:

- Allocate tasks to available execution nodes (e.g. processors, networks).
- Set priorities/deadlines for tasks contained in distributed transactions.
- Apply local scheduling techniques for tasks' execution. If tasks are not allowed to migrate (partitioned scheduling), these local scheduling techniques are uniprocessor.

In this chapter, we present two scheduling techniques for real-time transactions on multiprocessors.

The first scheduling technique is best suited for distributed embedded systems that have limited processing and communication resources. In this case, it is important to have the lowest scheduling overhead possible, so that applications that run on the platform will not be delayed.

The second scheduling technique is best suited for larger systems for which a traditional scheduler will fail at lower system loads. Our technique combines a genetic algorithm with simulation-based evaluation of candidate solutions to find feasible system setups. When using the term "system setup", we refer to a task to processor mapping and an intermediate task deadline assignment for which the EDF scheduling algorithm will produce a feasible schedule.

We implement tools that based on these scheduling techniques and starting from abstract real-time transactions and platform representations can produce static schedules and feasible system setups. The tools allow us to validate our research and to make the performance evaluation of the proposed techniques.

2. Literature review

Real-time scheduling on multiprocessor systems, as we already noted in Chapter 2, includes two important sub-problems:

- The allocation or mapping of tasks to processors
- The assignment of task priorities, which consequently establishes the order of execution

In transactional systems, that consider precedence restrictions between tasks, an extra problem is to establish priorities not only for the end of a transaction but also for the tasks contained in it. In systems that use EDF schedulers, the priority is given by the task's deadline. Usually, the intermediate task deadlines are not determined by the nature of the real-time application, but they may have an important impact on the schedulability of the system. Intermediate task deadlines are necessary to the local scheduling of tasks (on each individual processor).

An important number of research works investigate the two scheduling sub-problems separately, or as a composite solution.

Task allocation in distributed real-time systems is known to be an NP-complete problem [6], so an algorithm that generates an optimal solution in polynomial time, does not exist. To generate sub-optimal solutions for task allocation in polynomial time, one can use heuristics such as First Fit, Best Fit, Worst Fit or Next Fit [6]. If the system workload is heavy, a feasible solution may not be found even if such a solution exists.

Solutions that address only the tasks order of execution consider that task allocation to processors is already resolved. The most complex issues appear in the case of distributed applications, which are modeled as transactions or sequences of tasks with end-to-end deadlines. In this situation, intermediate tasks do not have predefined deadlines, so the only imposed restriction is that the last task of the sequence finishes its execution before the end-to-end deadline. To obtain a schedule, researchers proposed different algorithms [7] and heuristics [8][9] for assigning deadlines to intermediate tasks. In [7] the authors investigate a deadline assignment that reduces resource utilization. They consider a component-based approach, analyzing each transaction individually, and use separate windows of execution for the tasks in a transaction. In [8] and [9] the authors propose two similar heuristics for intermediate deadline assignment, which distribute the end-to-end deadline evenly or proportionally between all tasks. In [11] the authors use an iterative optimization algorithm called HOPA to assign deadlines to the tasks inside a transaction set. The authors in [12] use a genetic algorithm to optimize the deadlines assigned by HOPA.

In the case of composite solutions, which solve both task allocation and priority assignment, optimization techniques such as simulated annealing [10] or genetic algorithms [13][14][15][16][17] can be used. Other techniques use constraint programming [18] or combine search heuristics with Response Time Analysis [19].

It is rather difficult to make a comparison between the existing scheduling techniques, as they use different system models, schedulability analysis and different metrics for performance evaluation. For example, [6] evaluates task allocation heuristics using a periodic task model, with independent tasks, under EDF and Fixed Task Priority (FTP) scheduling. In [10] and in [18] the authors use periodic task models with inter-task communication and FTP scheduling, while in [13], [14] and [19] the authors use linear transaction models with end-to-end deadlines under FTP scheduling. In [15][16][17] the authors consider transactions described by directed acyclic graphs that contain non-preemptive tasks and communication costs, under FTP scheduling. For performance evaluation, authors use metrics such as:

- *Success rate*: ratio between the number of data sets for which a feasible scheduling solution was found, and the total number of data sets. [12][13]
- *Scheduling index*: the worst distance between the transaction's worst case response time and its deadline.[11]
- *Execution time* of the algorithm.[11][12][14]

Another important aspect for performance evaluation is the test data set. It has to be general enough to allow a relevant interpretation of the experimental results. However, there are papers [12][14] that use test data sets that, in our opinion, are not general enough.

When doing a comparison between two algorithms, it is a good idea to run the same test data sets with both algorithms. This approach is very hard to achieve or even impossible if researchers do not make public their tools and test data sets. At this time, very few research groups make available their tools and data sets.

3. Identified issues

The objective of our research is to schedule a set of real-time transactions on multiprocessor (distributed) platforms and to guarantee that all their constraints are respected. Our work addresses two important aspects of multiprocessor scheduling: task allocation to processors and the order for task execution on each processor. We intend to give a composite solution to these problems.

We consider the following types of constraints for the explored real-time systems:

- Precedence: the release and execution of a task may be conditioned by the execution of other tasks or reception of some messages, and thus, precedence relations between tasks are created. We represent applications as transactions.
- Real-time: Transactions have to finish their execution before their deadlines. If any deadline misses are detected, the application is considered not schedulable.
- Resource: each task can be deployed on a sub-set of the available platform nodes, those that provide all requested capabilities.

We propose two solutions to this problem:

- An algorithm that builds static cyclic schedules for each processor or network segment that belongs to the platform.
- A genetic algorithm that finds feasible system setups. Tasks are allocated to processors and are assigned deadlines such that, under local EDF scheduling, all transactions will meet their deadlines.

The detailed solutions are presented as follows.

4. The cyclic executive-based scheduling technique

The cyclic executive-based scheduling technique is best suited for multiprocessor (distributed) embedded systems that have limited processing and communication resources. In this case, the processing overhead introduced by the scheduler must be as low as possible such that the applications will not be delayed. The lowest scheduling overhead is obtained when using pre-computed schedules. Then, the scheduler does not need to make any scheduling decisions at runtime. It only chooses the job indicated by the schedule. The cyclic executive scheduling approach uses these pre-computed schedules.

The cyclic executive scheduling approach was chosen after weighing both its advantages and disadvantages [20].

The advantages are:

- The execution schedule is predetermined, and it guarantees that all deadlines are met. Future system execution is determined based on the schedule computed for a cycle.
- There are fewer context switches.
- Overheads introduced by context switches are low.
- The processor level scheduler is very simple. It only has to dispatch jobs according to the predefined schedule. There is no need for complex scheduling policies that can introduce large computational overheads.

The disadvantages are:

- The cyclic executive is best suited for harmonic task systems.
- The maximum allowed worst case execution time is constrained by the value of the minor cycle.
- Changes in the system can’t be made at runtime.
- Job overruns can’t be handled.

Some of the mentioned disadvantages can however be solved through system design decisions, such as enforcing harmonic task periods or splitting large tasks into smaller ones.

The design of the algorithm that generates cyclic schedules for real-time transactions on multiprocessors is presented as follows.

4.1. System model

To be able to build the schedule, one has to formally describe the workload and the underlying platform. In the next subsections, the workload and the platform models are presented.

4.1.1. The Workload Model

Applications are represented as preemptive *sets of tasks* (T). Each task ($t \in T$) is characterized by its worst case execution time (e) and by a set of execution constraints. Task execution is constrained by resource and data availability. Each task can only be executed on a subset of the available processors and can start its execution only if all its data dependencies have been solved. Tasks can produce data, which is considered "available" only at the end of their execution. Data is then ready to be used by other tasks, which, at their turn, can produce data. Data dependencies create precedence relations between tasks. Due to these precedence relations, tasks form directed acyclic graphs (DAGs) called *transactions*.

Real-time transactions have the following parameters:

- *Task graph* (G) – a DAG which describes the execution flow dictated by data dependencies between tasks
- *Period* (p) – the repetition period of the transaction
- *Phase* (ϕ) – the release time of the first instance of the transaction
- *Deadline* (D) – the relative deadline of the transaction; we assume that transactions have hard deadlines (deadline misses are not accepted)

The task graph consists of a set of nodes (N). A node $n_i \in N$ contains a task (t_i) and its *"children"*, a set of tasks which use data produced by t_i. Between two nodes n_i and n_j there is a directed edge (n_i, n_j) only if $t_j \in children(t_i)$. In the context of a task graph G, we will use notations $children(t_i)$ and $parents(t_i)$ to refer to the set of tasks which use the data produced by t_i and the set of tasks that produce the data which is used by t_i, respectively.

At the start of each period, an instance of the transaction is ready to be executed. A transaction instance contains task instances (jobs). The execution of a transaction starts with the execution of jobs that don't have data dependencies (jobs don't have to wait for data to be available). Gradually, all jobs in the transaction instance are executed, as soon as their data dependencies are resolved.

Task description does not contain any specific real-time parameters such as phase (ϕ), period or deadline. These parameters are present in the transaction description. These real-time parameters are inherited by the tasks in the task graph, from the transaction. The phase of the first task in the transaction is equal to the transaction phase. The deadline of the last task in the transaction is equal to the transaction deadline. All tasks that execute after the first task and before the last task have variable phases and deadlines. Their phases are computed as a function of their "parents" phase and execution time:

$$\phi(t_i) = \max(\phi(t_k) + e(t_k), t_k \in parents(t_i)) \qquad (1)$$

A real-time workload model $W=(T, TR)$ contains a set of tasks and a set of transactions. Fig. 1 shows an example of such a workload model. T is a set of five tasks and TR a set of three transactions. For each task, the worst case execution time and the set of processors (on which the execution is possible) are known. Transactions describe the flow of execution created by precedence relations between tasks. For example, in transaction tr_3, the jobs of tasks t_2 and t_3 can begin their execution only if the job of task t_4

has finished its execution. In the same transaction, the job of t_5 will begin its execution if the jobs of t_2 and t_3 have finished.

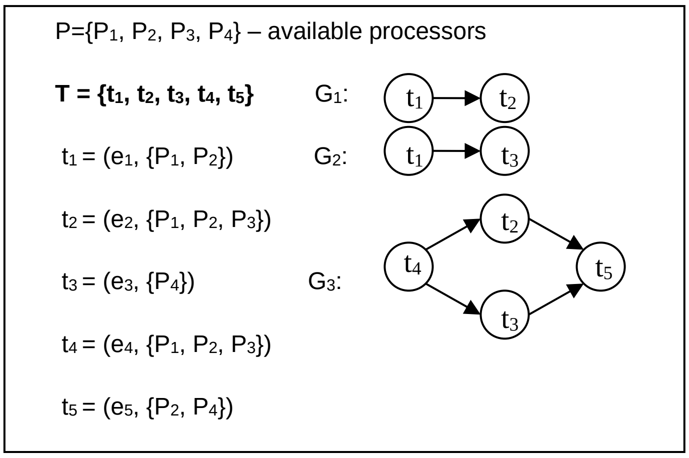


Figure 1. A workload model with five tasks and three transactions that run on 4 processors.

4.1.2. The Platform Model

These real-time applications execute on a distributed platform, which consists of multiple nodes that communicate through a network. Each node has a single processor and a buffer, where data generated by tasks is memorized until use. The execution rate of all tasks is the same on all processors. Nodes have different capabilities (e.g. not all nodes can measure the temperature of the environment), and therefore, a task can execute only on the platform nodes which provide all requested capabilities. As previously presented, each task has an associated set of processors that are able to execute it.

As communication scheduling is out of the scope of this research, it is assumed that the communication network insures real-time transmission of data packets between tasks.

Data transmission time is included in the execution time of the task that produced the data. Even if more than one tasks use the produced data, it is considered that the transmission is done only once. Data reception time is included in the execution time of the task that uses the data. Data transmission and data reception times are the same if the transmitter and receiver tasks execute on the same node, or on different nodes. Data transmission is asynchronous. This means that a task will finish its execution as soon as data transmission is finished. Data is then buffered until the receiver tasks are released. The transmitter task will not block until the release of the receivers.

Fig. 2 shows an example of communication between tasks. Tasks are placed on time axes. Tasks t1 and t2 execute on the same processor, t3 executes on another processor. Task t1 produces data that is later used by t2 and t3. The gray part of task execution depicts the portion of task execution time that is used for data transmission or data reception.

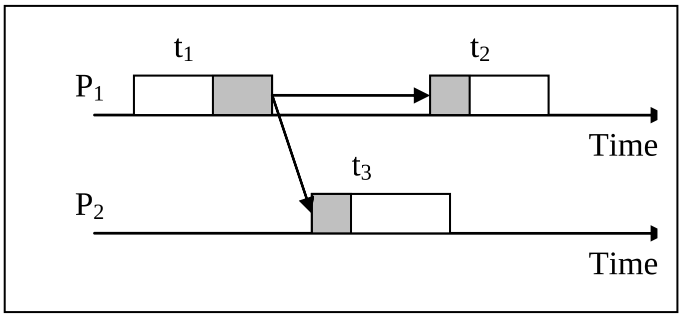


Figure 2. Communication between tasks.

The processing resources of the platform are modeled as a homogeneous multiprocessor P={P_1, P_2, … , P_m} composed of *m* processors. Similar to the resource partition model presented in [21], each processor is modeled as $P_i = (\Delta_i, \Pi_i)$, where Δ_i represents the processor available time during the period Π_i. Δ_i is expressed as a union of non-overlapping time intervals:

$$\Delta_i = \bigcup_k [a_k, b_k], \text{ where } a_k < b_k < a_{k+1} \text{ and } b_k \le \Pi \qquad (2)$$

The semantic of this representation is that processor P_i has the same availability in any $[k * \Pi_i, (k+1)\Pi_i]$ time interval. This representation will help us compute a periodic schedule for each processor.

4.2. Building the cyclic schedule

Under the assumptions of the previously described workload and platform models, our scheduling problem reduces to multiprocessor scheduling. This solution is an adaptation of the uniprocessor cyclic executive model described in [22], to multiprocessors and transactions.

The cyclic executive is a simple scheduler that uses a cyclic schedule that establishes the job execution order. The cyclic schedule is computed offline, before system start. This schedule covers job execution over a time period. At the start of each new period, the cyclic schedule is restarted.

The cyclic executive model is best suited to schedule harmonic task systems, because in other cases the length of the schedule can be too large to be of any use in real-world systems. A periodic task system is harmonic if all task periods are multiples of the smallest task period.

In this case, tasks inherit the periodicity of transactions. In consequence, many tasks will have the same period. So, the condition to have a harmonic transaction set is less restrictive on application system design.

The choice of the cyclic executive was made because:

- The static cyclic schedule used by the executive guarantees that all transaction deadlines are met. It does not need further analysis to demonstrate its feasibility.
- By computing the schedule for a cycle, the entire future system execution is predicted.

- The method does not involve complex Response Time Analysis or other schedulability analysis techniques.
- The computational overhead introduced by the local processor scheduler is minimized. No additional scheduling decisions have to be made at runtime.
- The method reduces the number of context switches.

Other scheduling-related assumptions are:

- Jobs can be preempted.
- Jobs are not allowed to migrate, but different jobs of the same task can be started on different processors.

The objective is to allocate jobs to processors, such as job constraints are respected (deadline, resource constraints and data dependencies). For each processor, a cyclic schedule will gradually be built, as jobs are allocated processor time.

A cyclic schedule is defined as a pair $S = (\sigma, \Pi)$, where Π is the schedule repetition period. σ is an ordered sequence of executables $\varepsilon_i = (j_k, r_i, e_i)$, where j_k is the executed job, r_i is the start time of the executable and e_i is its execution time.

$$\forall\ \varepsilon_i,\ \varepsilon_j \text{ in } \sigma \text{, if } \varepsilon_i < \varepsilon_j \text{, then } r_j \geq r_i + e_i \qquad (3)$$

Jobs can be preempted, so the execution of a job can be composed of more than one executables. Each executable has a repetition period equal to the schedule repetition period.

To obtain the set of cyclic schedules for the multiprocessor, a cyclic schedule for each processor $P_i = (\Delta_i, \Pi_i)$ must be computed. It is assumed that processors have equal periods. Each schedule's repetition period must be equal to the processor period and equal to the least common multiple (LCM) of the transaction set periods:

$$\Pi = LCM(\{p(tr_k) \mid tr_k \in TR\}) \qquad (4)$$

Each executable in the schedule occupies processor time, each period. A new executable can be added to the cyclic schedule only if the requested time interval is available on the processor.

To create the cyclic schedules:

- First, the list of jobs that will have to be executed during Π is computed.
- Then, jobs are allocated while gradually building intermediate cyclic schedules for each processor.
- Each taken allocation decision is final.

4.2.1. The List of Jobs

For each transaction $tr \in TR$, there will be $\frac{\Pi}{p(tr)}$ transaction instances. Each transaction instance contains jobs that have to be executed according to its task graph. For each job (*j*) the following information is needed:

- Release time (*r*) – time when the job starts its execution. The release time of a job is computed with equation (5).
$$r = \phi(t) + (k-1) * D(tr) \tag{5}$$
- Deadline (*D*) – equal to transaction instance deadline.
$$D = k * D(tr) \tag{6}$$
- Completion time (*c*) - time when the job completes its execution.
- Remaining transaction execution time (*E*) – transaction execution time starting with this job, to transaction end.
- Priority (*Pr*) – The job with the earliest release time is scheduled first. If there are jobs with the same release time, the one that is closest to deadline is first.
$$\mathrm{Pr} = r + \frac{D-(r+E)}{\Pi} \tag{7}$$
- Scheduled – “true” if the job was scheduled.
- Ready – “true” if the job is ready to be scheduled. A job is ready to be scheduled only if its “parents” have already been scheduled.
- The transaction to which it belongs (*tr*).
- The task graph node that generated it – needed to compute data dependencies (*n*).
- The task that generated it (*t*).
- The instance sequence number (*k*, $1 \le k \le \frac{\Pi}{p(tr)}$)

4.2.2. Job Allocation

To allocate jobs to processors, the following steps are repeated until all jobs in the list of jobs are scheduled:

1. The ***first step*** is to choose the job that has the lowest priority value, from the job list. Only jobs which are “ready” (do not have any unresolved data dependencies) can be selected.
2. The ***second step*** is to select a processor for the job. From the subset of processors on which the job can execute (obtained from the corresponding task), the processor on which the job would complete earliest is chosen. If there are two processors that can provide the same completion tine, the one least loaded is selected.
3. In the ***third step***, the job is locally scheduled on the selected processor.
4. At the end, job’s “children” are marked as “ready”, and the release time in recomputed for all unscheduled jobs belonging to the transaction instance. The release times have to be recomputed, because:

- It is not guaranteed that each job receives a time interval that starts exactly at its estimated release time.
- Jobs can be split over many execution intervals. Job's children can only have release times greater or equal to its completion time.

After re-computing the release times, there may be cases in which unscheduled jobs would certainly miss their deadlines, and then, the transaction set would be considered not schedulable.

The pseudocode for the job allocation strategy is listed below.

Algorithm 1: Job Allocation Strategy

Input: J - the list of jobs; P – the list of processors
Output: $\sum$ – the set of cyclic schedules

```
Begin

while(NotScheduled(J))
{
     j = SelectJobToSchedule(J);
     p = SelectProcessor(j,P);
     if (Processor selection failed)
     {
          Transactions could not be scheduled;
          break;
     }
     Schedule(j,p);
     MarkReadyJobs(Children(j));
     RecomputeReleaseTimes(j,J);
     if (Jobs miss deadlines)
     {
          Transactions could not be scheduled;
          break;
     }
}
Σ = GetSchedules(P);

end
```

4.2.3. Schedule generation

For each processor, a scheduling algorithm (scheduler) receives the job as input, and produces a list of executables, which are then added to the processor schedule. The scheduler searches for available execution time intervals on the processor, which match the job execution request. The needed execution time intervals are then occupied and the corresponding executables are created.

To optimize the search for available execution time intervals, the processor available time Δ_i is partitioned in $l = \frac{\Pi}{GCD}$ sub-intervals, where GCD is the greatest common divisor of the transaction set periods. Each sub-interval can have at most *l* available units of execution time. When an execution time interval is occupied, it is removed from Δ_i . Δ_i is expressed as:

$$\Delta_i = (\delta_1, \delta_2, ..., \delta_l); \delta_j = \bigcup_k [a_k, b_k], \text{ where } a_k < b_k < a_{k+1} \text{ and } b_k \le l \tag{8}$$

The job *j* which has to be scheduled, initially requests *[r(j), r(j)+e(j)]* execution time interval. In most cases, this interval is not available. The scheduler searches the best match, which can be an ordered list of *n* intervals $[a_k, b_k] \in \Delta_i$, where $r(j) \le a_1, b_n \le r(j) + e(j)$ and $a_k < b_k < a_{k+1}$. For each interval, the corresponding executable is created $\varepsilon_k = (j, a_k, b_k - a_k)$ and added to the cyclic schedule. The pseudocode for the generation of a cyclic schedule is listed below.

Algorithm 2: Generation of a cyclic schedule

Input: j - the list of job; r(j) – release time of j; e(j) – execution time of j; Δ – processor available time; l – number of sub-intervals of Δ
Output: EL – the list of executables

```
Begin

remainingExecTime = e(j);
searchKey = r(j)/l;
startTime = r(j)%l;
while(remainingExecTime>0)
{
    Foreach(Interval [a,b] in [searchKey])
    {
        if (startTime < b)
        {
            if (a > startTime)   {  startTime = a; }
            if (b >= startTime + remainingExecTime)
            {
              s= searchKey * l + startTime;
              EL.Add(j, s, s + remainingExecTime);
              Δ.Remove(δ[searchKey], startTime, startTime+ remainingExecTime);
              return EL;
            }
            else
              {
                    s= searchKey * l + startTime;
                    EL.Add(j, s, s + b);
                    Δ.Remove(δ[searchKey], startTime, b);
                    remainingExecTime -= b - startTime;
                    startTime = b;
              }
        }
    }
  searchKey ++;
  startTime = 0;
}
```

```
    return EL;

    end
```

To obtain a multiprocessor schedule, cyclic schedules have to be generated for each processor. The proposed scheduling algorithm splits the job's execution over the available processor time. Unlike the classic cyclic executive technique, our approach does not impose that the job's execution time is less than the minor cycle (GCD of the transactions' periods). Therefore, our method is more general.

4.3. Experimental evaluation

To validate the proposed scheduling method, we implemented a tool that receives as input a text file which contains the workload model as previously described, and the number of processors on which the workload should be scheduled. The tool generates a text file that contains a cyclic schedule for each processor, by using the proposed multiprocessor cyclic executive method.

$P=\{P_1, P_2, P_3\}$ – available processors

$T = \{t_1, t_2, \dots , t_{20}\}$

$t_1 = (3, \{P_1, P_2\})$	$t_8 = (1, \{P_1, P_2, P_3\})$	$t_{15} = (2, \{P_1, P_2, P_3\})$
$t_2 = (3, \{P_1, P_2, P_3\})$	$t_9 = (2, \{P_1, P_2, P_3\})$	$t_{16} = (2, \{P_1, P_2, P_3\})$
$t_3 = (3, \{P_1, P_2, P_3\})$	$t_{10} = (2, \{P_1, P_2, P_3\})$	$t_{17} = (2, \{P_1, P_2, P_3\})$
$t_4 = (2, \{P_1, P_2, P_3\})$	$t_{11} = (3, \{P_1, P_2, P_3\})$	$t_{18} = (2, \{P_1, P_2, P_3\})$
$t_5 = (2, \{P_1, P_2, P_3\})$	$t_{12} = (1, \{P_2, P_3\})$	$t_{19} = (2, \{P_1, P_2, P_3\})$
$t_6 = (4, \{P_1, P_2, P_3\})$	$t_{13} = (3, \{P_1, P_2, P_3\})$	$t_{20} = (2, \{P_1, P_2, P_3\})$
$t_7 = (2, \{P_2, P_3\})$	$t_{14} = (3, \{P_1, P_2\})$	

$TR = \{tr_1, tr_2, tr_3, tr_4, tr_5, tr_6\}$

$tr_1 = (G_1, 10)$; $G_1 = \{(t_1, \{t_2\}), (t_2, \{\})\}$

$tr_2 = (G_2, 10)$; $G_2 = \{(t_3, \{t_4\}), (t_4, \{\})\}$

$tr_3 = (G_3, 15)$; $G_3 = \{(t_5, \{t_6\}), (t_6, \{t_7\}), (t_7, \{\})\}$

$tr_4 = (G_4, 30)$; $G_3 = \{(t_8, \{t_9, t_{10}\}), (t_9, \{t_{11}\}), (t_{10}, \{t_{11}\}), (t_{11}, \{t_{12}, t_{13}\}), (t_{12}, \{t_{14}\}),$
$(t_{13}, \{t_{14}\}), (t_{14}, \{\})\}\}$

$tr_5 = (G_5, 15)$; $G_5 = \{(t_{15}, \{t_{16}, t_{17}\}), (t_{16}, \{t_{18}\}), (t_{17}, \{t_{18}\}), (t_{18}, \{\})\}$

$tr_6 = (G_6, 30)$; $G_6 = \{(t_{19}, \{t_{20}\}), (t_{20}, \{\})\}$

Figure 3. A workload model with five tasks and three transactions that run on 4 processors.

The test configuration has 3 processors, 20 tasks and 6 transactions. The workload model is described in Fig. 3. It is assumed that each transaction starts at time $\phi=0$, and that transaction period is equal to the deadline. The total utilization of the transaction set is 2.8, which would generate an average load of 0.933 on each processor.

Our tool generated the following schedules with Π =30:

- S(P1)={(t_1;0;3), (t_8;3;1), (t_6;4;4), (t_{20};8;2), (t_{10};10;2), (t_{18};12;2), (t_{11};14;3), (t_{15};17;2), (t_{16};19;2), (t_1;21;3), (t_{14};24;3), (t_{18};27;2)}
- S(P2)={(t_3;0;3), (t_{19};3;2), (t_4;5;2), (t_{16};7;2), (t_7;9;2), (t_3;11;3), (t_2;14;3), (t_{12};17;1), (t_{13};18;3), (t_3;21;3), (t_7;24;2), (t_4;26;2)}
- S(P3)={(t_5;0;2), (t_{15};2;2), (t_2;4;3), (t_9;7;2), (t_{17};9;2), (t_1;11;3), (t_4;14;2), (t_5;16;2), (t_6;18;4), (t_{17};22;2), (t_2;24;3)}

Processor loads obtained after scheduling are: 0.9 on P1, 0.94 on P2 and 0.96 on P3.

In another experiment, the configuration is 7 tasks and 3 transactions to be scheduled on 2 processors. The transaction set generated an average load of 0.84 on the processors. One task had the execution time equal to the most frequent transaction period. This condition makes scheduling more difficult in the case of cyclic executives. Our tool generated schedules for this configuration, too.

To make a statistical evaluation of the proposed scheduling technique, transaction sets were generated with the workload generation tool presented in Chapter 5. The imposed constraints on the generated data sets are:

- Chain transactions
- Transactions in a set have harmonic periods and deadlines
- The system is composed of 6 transactions on 4 processors, and of 10 transactions on 8 processors
- A transaction contains at most 10 tasks

Transaction sets were generated with increasing average system utilization starting at 40% and up to 99%. For this evaluation, 390 transaction sets with these characteristics were used.

The following metrics were used:

- *Success rate*: ratio between the number of data sets for which a feasible scheduling solution was found, and the total number of data sets.
- *Execution time* of the algorithm.

The evaluation of the proposed cyclic executive scheduling technique (CYEX) was made by comparison to a global EDF scheduler (LAX-EDF). The global EDF scheduler chooses the jobs that have the closest deadlines at runtime. These jobs are allocated to any available processor; jobs can migrate on different processors during their execution. For the EDF scheduler to work, intermediate tasks inside the transaction must be allocated deadlines. In this case, the deadlines were computed with equation (9).

$$l = D - \sum_{Tasks} C_k \tag{8}$$

$$d = C + l * \frac{C}{\sum_{Tasks} C_k} \tag{9}$$

Where *D*, *C* are the transaction's deadline and execution time, *l* is the transaction's laxity and *d*, *c* are the deadline and the execution time of a task. To compute intermediate tasks' deadlines, the transaction's laxity time is proportionally divided between tasks.

LAX-EDF scheduling technique was implemented using the RTMultiSim simulation tool that is presented in Chapter 5. As it is a runtime scheduler, its functioning was simulated, and the performance comparison was based on the simulation results.

It is expected that LAX-EDF has a better success rate than CYEX, because it is a runtime scheduler that allows job migrations. All test transaction sets were scheduled with both techniques.

Experimental results are presented in Fig 4, 5 and 6. It was observed that CYEX has better success rate than LAX-EDF for scheduling 6 transactions on 4 processors. For the case of 10 transactions on 8 processors, CYEX has 20% less success rate than LAX-EDF. During the experiments, it was observed that CYEX has some problems in scheduling heavy (low laxity) transactions.

In terms of execution time, CYEX is better than LAX-EDF (see Fig. 6). CYEX finds the schedules earlier that LAX_EDF is able to generate them in RTMultiSim.

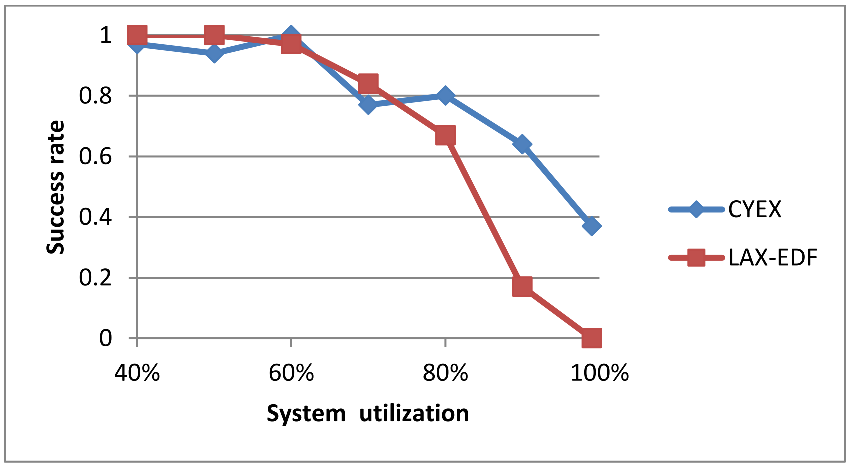


Figure 4. Success rate for scheduling workloads with 6 transactions on 4 processors.

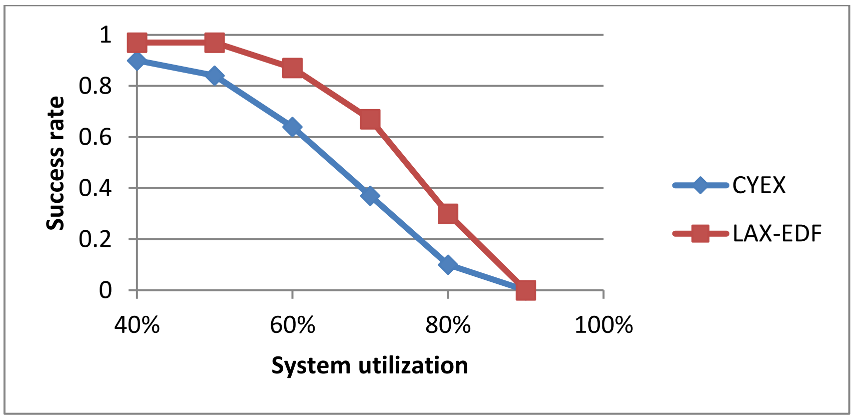


Figure 5. Success rate for scheduling workloads with 10 transactions on 8 processors.

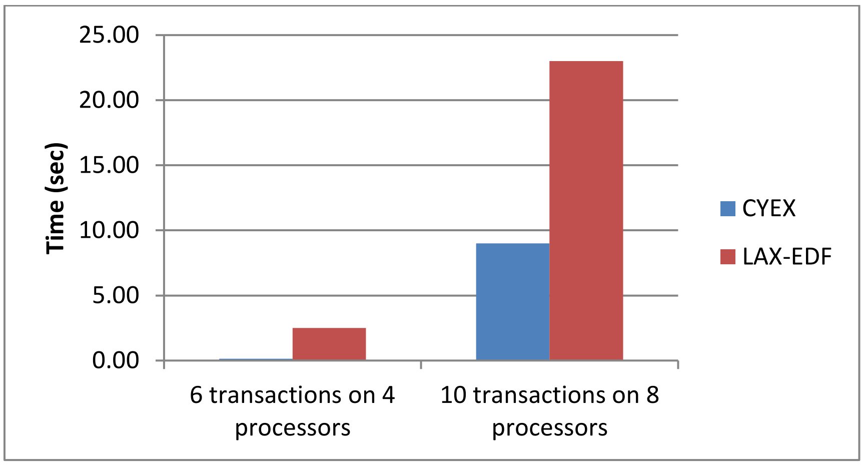


Figure 6. Execution time of CYEX compared to LAX-EDF.

Concerning related work, the cyclic executive model didn't receive much attention from the research community, even if it is used in safety-critical systems and in real-time network protocols such as WorldFIP [23]. A uniprocessor cyclic executive model implemented in Ada was formally described in [22]. In [24], an adaptation of the cyclic executive is used for scheduling high-criticality tasks in a mixed-criticality system. The cyclic executive is one of the several schedulers they used in their hierarchical scheduling model.

More recently, an implementation of a multiprocessor cyclic executive in safety-critical Java was presented in [25]. They compute cyclic schedules using the UPPAAL [26] model checker. As they do not make a statistical evaluation of their approach, we are not able to compare it with our approach in terms of success rate or a similar metric. However, they report that their schedule generation process, done with UPPAAL, takes in the case of 16 tasks and multiple resource constraints as much as 30 minutes, which is far more than the execution time of CYEX.

We conclude that our approach, CYEX, has better success rate compared to a global EDF approach, but only in the case of small systems. Moreover, CYEX is faster than the global EDF approach and even than the UPPAAL approach presented in [25].

5. A genetic approach for multiprocessor real-time scheduling

In this section, an optimization-based technique that enhances the scheduling of real-time transactional multiprocessor systems is described. The technique addresses two important aspects: task allocation and task deadline assignment. In order to satisfy real-time restrictions genetic search and simulation are combined to find feasible system setups. The genetic engine looks for a feasible task-to-processor mapping (allocation) and deadline setting that meets the given real-time and dependency restrictions. The simulator is used to evaluate the behavior and consequently the quality of different candidates. Through an iterative process, we obtain a feasible scheduling solution by choosing the best candidate result.

This contribution addresses the adaptation of a genetic algorithm to the multiprocessor scheduling problem and specifically the definition of a multi-criteria fitness function that describes the quality of a schedule related to the imposed time restrictions.

5.1. System model

5.1.1. Workload model

The real-time transactions model described in Chapter 2 is used for the genetic approach. A transaction is a sequence of tasks executed periodically, which has an end-to-end deadline. This means that the transaction must finish its execution before that deadline. We choose to represent the precedence dependency between tasks in a transaction through a list.

A real-time transaction has the following defining elements:

- Task list (L) – a list that describes the dependencies between tasks and determines the execution order restrictions.
- Period (T) – the repetition period of a transaction.
- Deadline (D) – the time limit for a transaction, relative to its release time.

A task has the following parameters:

- Execution time (C).
- Deadline (d) – time limit relative to the task's release time.
- CPU affinity – list of processors on which the task can be executed.

Transactions are released periodically in accordance with some external or functional requirements. A transaction instance contains task instances called generically jobs. The execution of a transaction starts with the execution of jobs that do not have precedence dependencies. A job is considered for scheduling only if its dependences are solved (jobs that precede it are executed).

In case of a real application, only transaction deadlines are specified. Intermediate task deadlines are not specified. However, the scheduling algorithm, in our case EDF, requires such deadlines in order to establish the execution priorities. One of the main goals of our research is to determine these intermediate deadlines in a way that all real-time, precedence and resource restrictions are satisfied.

Our workload model is general, in the sense that it may be configured to represent independent sets of tasks or distributed applications, including network communication tasks (messages). The message, in our case, can be represented as a task, which is handled only by a network segment (represented as a processor), configured in the CPU affinity parameter.

5.1.2. Platform and Scheduling Models

The processing resources of a platform are modeled as a multiprocessor system P={P1, P2, … , Pm} composed of processors (CPUs) and possibly network segments. From an abstract point of view, the network segments may be assimilated with processors that can handle messages. Messages are scheduled for transmission on a network segment in a similar way as tasks on a processor. This model can cover a wide range of system configurations that span from parallel systems (without messages and network segments) to distributed ones.

It is assumed that each processing resource has its own scheduler. The scheduler chooses the job with the highest priority to be executed, at a certain point in time, on the processing resource. The priority is computed by the scheduling algorithm implemented in the scheduler. The experiments use the EDF algorithm that assigns priorities to jobs according to their deadlines, so the job that has the closest deadline will have the highest priority. It is known that the EDF algorithm is optimal for single processor systems [1] and it can handle task set utilizations of up to 100% in the case of independent tasks.

The workload model must be mapped on the platform, so that all transactions meet their deadlines. To accomplish this goal, each task must be allocated to a suitable processing resource. On each processing resource, each task must have a deadline, to be able to compute its jobs priorities. Both, allocation and deadline assignment are solved with the proposed approach.

5.2. Scheduling with a genetic approach

This solution addresses two important aspects of multiprocessor scheduling: task allocation to processors and intermediate task deadline assignment.

The task allocation and deadline assignment problems generate a multidimensional solution space, which increases with the number of processors and the number of tasks in the system. As mentioned before, finding an optimal solution is an NP-complete problem. However, sub-optimal solutions are acceptable in our case if transactions' end-to-end deadlines are not exceeded, even if some intermediate task deadlines are missed. Our objective is to find this type of sub-optimal scheduling solution.

The genetic algorithm is used as a search and optimization method, because it is well suited for problems with a large search space and multiple optimization objectives. Genetic algorithms are inspired from biological evolution. A genetic algorithm starts with an initial set of possible solutions called population. At each iteration step, it generates a new population by means of natural selection, crossover and mutation. Each solution is evaluated with a fitness function. The fittest solutions will propagate their characteristics to later populations, generating improved new solutions.

The continuous genetic algorithm was adapted to the problem domain. Scheduling variables that must be optimized are task allocations to processors and task deadlines. The parameters that need to be minimized are transaction response times, task response times and the processor utilization factor (defined later by equation 16).

Initial solutions are obtained by applying known heuristics for both task allocation to processors and intermediate task deadline assignment. New populations are generated mostly through crossover, but also by keeping the best individuals from the previous population. After obtaining a new population, mutation is applied. The variables of the genetic algorithm are population size, crossover and mutation probability factors.

The fitness of a solution is evaluated through simulation. The simulator receives a workload model obtained from the individual representation created in the genetic algorithm, and creates the execution schedule using the platform and the scheduling predefined models. This approach is based on the results in [27], where the authors showed that for deterministic fixed priority scheduling algorithms (e.g. EDF) the schedulability analysis must be performed during a period equal to twice the hyper-period of the task set (hyper-period = least common multiple of tasks' repetition periods). Therefore, each configuration setup is simulated for two hyper-periods of the transaction set, and the timing results obtained through simulation are used for the computation of the fitness function.

A conceptual schema of the proposed technique is depicted in Fig.7. The optimization-based technique comprises three steps. In the first step, the genetic algorithm generates a solution population. The solutions are evaluated through simulation in the second step. In the third step, based on the evaluation, the genetic algorithm generates a new solution population, which replaces the previous. These steps are iterated until a feasible solution is reached.

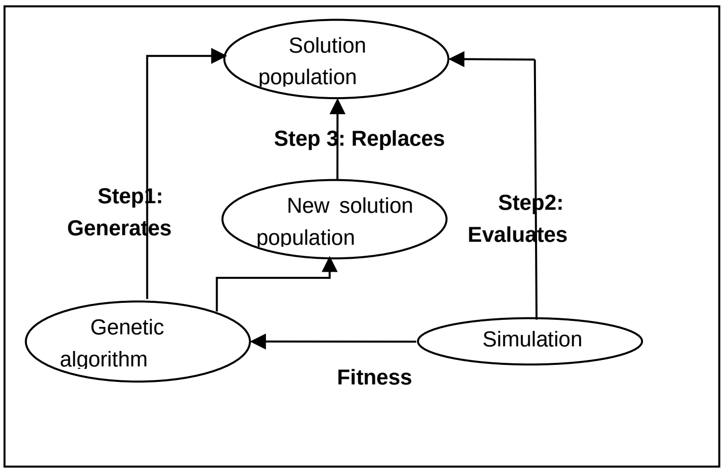


Figure. 7. Scheduling using a genetic approach

Algorithm 1 shows the pseudo-code for the genetic algorithm. New populations are mostly generated through crossover, but also by keeping the best individuals from the previous population. After obtaining a new population, mutation is applied. The configurable variables of the genetic algorithm are population size, crossover and mutation probability factors. If at any iteration step, it is observed that there is not enough genetic diversity mutation is applied with a higher rate on the population, before the creation of a new population.

Algorithm 1: Genetic scheduling

```
Begin

Generate initial population;
For each individual in population compute fitness;
While (generations < max_generations)
{
   If (not enough genetic diversity)
   {
        Apply mutation on population with higher rate;
   }
   Create new individuals with crossover;
   Add new individuals to new population;
   Choose best individuals from population;
   Add best individuals to new population;
   Replace population with new population;
   Apply mutation on population;
   For each individual in population compute fitness;
```

```
    }
End
```

5.2.1. Genetic representation of a scheduling solution

Solutions to the processor allocation and deadline assignment problems are represented as individuals in a population. Each individual (chromosome) is composed of a sequence of genes. A gene is an integer value that may represent a processor number (identifier) on which the task is allocated, or a task deadline. Each gene has its own domain of values (Δ). For processor allocation, the domain is the task's CPU affinity list.

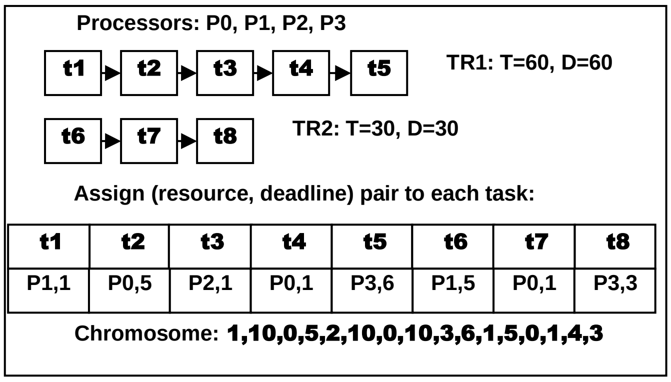
$$\Delta_P^{Task_j} = \{P_i | P_i \in CPUAffinityList_{Task_j}\} \tag{10}$$

In the case of task deadlines, the value domain is continuous between the execution time of the task (minimum) and the largest possible deadline (maximum) assuming that all subsequent tasks in the transaction can execute without interruption, without exceeding the transaction deadline. Note that a task's deadline is relative to its release time. A task is not released until all its predecessors finished their execution, so the assigned deadlines do not determine or enforce the precedence between tasks.

$$\begin{aligned} &\Delta_d^{Task_j} = \left[d_j^{min}, d_j^{max}\right], \\ &d_j^{min} = C_j, \\ &d_j^{max} = D_k - \sum C_l \,, Task_l \in Transaction_k \end{aligned} \tag{11}$$

Even though the transactions have hard deadlines (meaning that if a transaction misses its deadline, then the scheduling is considered not feasible) the tasks inside transactions have soft deadlines. This happens because even if several intermediate tasks miss their deadlines, it is still possible for the last task (and consequently the transaction) to meet its deadline. For this reason, we allow a less constrained value domain for genes that represent task deadlines. A less constrained deadline domain can increase the chances of finding a good scheduling solution.

The chromosome contains two adjacent genes for each task in the workload. The first gene corresponds to the processor to which the task is allocated and the second is the task's deadline. The order of the genes is given by transactions. Fig. 8 gives an example of how the chromosome is constructed based on the system model. The example is composed of four processors and two transactions (with the corresponding tasks). Tasks can be allocated to any of the four processors. For instance, the first gene shows that task t1 is allocated to processor P1 and its intermediate deadline is 10 time units. The next gene is for task t2 allocated on P0 and with deadline of 5 time units.


Processors: P0, P1, P2, P3
t1
t2
t3
t4
t5
TR1: T=60, D=60
t6
t7
t8
TR2: T=30, D=30
Assign (resource, deadline) pair to each task:
Chromosome: 1,10,0,5,2,10,0,10,3,6,1,5,0,1,4,3

t1	t2	t3	t4	t5	t6	t7	t8
P1,1	P0,5	P2,1	P0,1	P3,6	P1,5	P0,1	P3,3

Figure 8. Chromosome construction based on the transaction model

For an efficient search, the initial population must have a large genetic variety. We generated the initial population in two steps. First, we created a number of individuals using task allocation heuristics such as First Fit or Round Robin. We also assigned the deadlines for tasks and messages choosing the smallest possible value as deadline (d^{min}). In the second step, we choose a random individual from the population, copy it and apply several mutations. The obtained individual is then added to the population only if there are no duplicates. The second step is repeated until the specified population's size is reached.

5.2.2. *Genetic operators*

Two types of genetic operators are used: crossover and mutation. These operators are applied on the current population with predefined probability rates, to create new generations with better fitness.

Crossover combines genes from two individuals to create two offspring, which will have mixed characteristics from the parents. For our scheduling problem, we use a two-point crossover operator. The parents are selected by tournament. From two randomly selected individuals, the fittest is chosen to be one of the parents. This way we give a chance to individuals that do not have the best fitness but may have good partial solutions to propagate their genes to later generations.

Crossover is allowed only between different individuals. The crossover points are selected at random. Cases are eliminated when the points coincide or when they are at the two ends of the chromosome. A certain task can inherit the allocation gene from a parent and the deadline gene from the other parent, so it is not necessary that the genes between the crossover points include both allocation and deadline genes for a task.

Mutation randomly changes the value of a gene in order to find a better solution and consequently to maintain a certain degree of genetic variety. Mutation is applied with a given probability on all individuals each time a new population is obtained.

Two different mutation operators are applied. In the majority of cases, we applied the classic mutation operator that selects a random gene from the chromosome and changes its value, with another random value from the gene's value domain. We also implemented a mutation operator, which chooses the new value from a sub-interval around the current value of the gene. The limited interval is set as a percentage of the maximum allowed interval.

Experimental results improved when the interval was limited to 50% of the initial domain and later to 25%. We applied this type of mutation only on populations with good average fitness, because we suppose that their genes are close to their best values and we do not want to spoil good possible solutions through mutations that radically change the gene’s value.

5.2.3. The fitness function

The fitness function evaluates the quality of a given scheduling solution related with some chosen optimization objectives. A scheduling solution is considered the best if all transactions finish before their deadlines, all tasks finish their execution before their designated deadlines and if the tasks are uniformly allocated to the available processors. But, as mentioned before, we only look for sub-optimal solutions, in which at least all transactions finish before their deadlines. For this purpose, we establish three optimization objectives, with different weights.

The most important optimization parameter is the transaction’s completion time, which has to be less than the transaction’s deadline. A solution is considered feasible if the transaction’s completion time is less or equal to the transaction’s absolute deadline. Another optimization objective is to have a rather uniform allocation of tasks over the existing resources. This will increase the system’s robustness and it can increase the effectiveness of the search. The third optimization objective is the task response time, which has to be less than the task’s deadline. We tried to differentiate between satisfying transactions deadlines and intermediate task deadlines in the sense that transaction deadlines are much more important (because they derive from real application constrains) and task deadlines are artificially introduced for the scheduling algorithm.

In order to express all the above optimization goals, we defined a fitness function as a weighted combination (sum) of four fitness expressions. A smaller value means a better solution.

The first component f_{TR} measures the quality related to the transaction’s completion time (equation 3). It is computed as a sum of terms, each term representing a transaction that does not meet its deadline. A term is an exponential function of the difference between the maximum response time R_i and the deadline D_i of a transaction. The goal is to have all the differences equal to zero, which means that all transactions meet their deadlines.

$$f_{TR} = \sum_{(R_i - D_i)>0} 2^{max(R_i - D_i)} \qquad (12)$$

The second component measures if the intermediate tasks meet their deadline (equation 4). This component has a smaller weight.

$$f_t = \sum_{tasks} 2^{\max(r_j - d_j)} \qquad (13)$$

The next component measures the degree of uniform allocation of tasks on processors (equation 14). It is the sum of the differences between the actual processor’s utilization factor U_p and an average value. The goal is to obtain an allocation as close as possible to the average value. It is obvious that this component will have lower values if the transactions are composed of tasks with lower individual utilization that can be evenly spread on the available processing resources.

$$f_{alloc} = \sum_{p \in P} \left| U_p - \overline{U} \right| \quad (14)$$

where:

$$U_p = \sum_{tasks \in p} \frac{C_k}{d_k} \quad (15)$$

$$\overline{U} = \frac{\sum_{tasks} \frac{C_k}{d_k}}{card(P)} \quad (16)$$

C_k and d_k are the execution time and deadline of task k. Card(P) is the number of processors.

Below is the complete expression of the fitness function.

$$F = f_{TR} * w_1 + f_{alloc} * w_2 + f_t * w_3 \quad (17)$$

Based on experiments, we found that a good combination of weights is: w_1 = 1000, w_2=100 and w_3=1. A significant observation is that the most important factor should receive the largest weight.

5.3. A technique that reduces the search space

As the task allocation and deadline assignment problems generate very large solution spaces that increase with the number of processors and the number of tasks, the execution time of the genetic algorithm can be very large (e.g. hours). Moreover, as the solution space increases, the algorithm's success rate reduces because it doesn't always converge to an acceptable solution in an acceptable number of iterations, or it deadlocks (finds a local minimum).

We intend to investigate if a smaller search space has a good impact on the genetic algorithms success rate, its speed and the fitness of the best solutions. This technique combines the optimization-based approach for finding a task to processor allocation with a non-iterative approach for intermediate task deadline assignment.

The steps for finding a scheduling solution are:

- In the first step, the chromosomes are created.
- In the second step, a non-iterative algorithm or heuristic is applied to assign deadlines to all tasks.
- In the third step, the fitness is computed.
- In the fourth step, the genetic algorithm chooses the best individuals and applies the genetic operators to obtain a new population.

Second, third and fourth steps are iterated until an acceptable solution is found. The genetic algorithm will create the chromosomes as described in section IV, but the genes that represent task deadlines will have constant values (determined in the second step) and will not be modified (mutated) during genetic iterations.

We implemented two variants of the proposed technique, by using two distinct solutions for task deadline assignment. In the first variant, we used the algorithm proposed in [7], where the authors find a deadline allocation by constructing an ordered list of tasks. The obtained deadline assignment depends on the distribution of computation times among the tasks and on task to processor allocation.

In the second variant, we propose a deadline assignment heuristic similar to the ones presented in [8] and [9]. We computed the transaction laxity time (l) as in equation 9 and we divided it proportionately between the transaction's intermediate tasks. The deadline is computed as the execution time to which is added the portion of the transaction's laxity time (see equation 10).

This deadline allocation heuristic is influenced by the transactions time parameters (deadline and execution time) and not by the task to processor allocation.

The implementation variants allow us to explore two distinct strategies for deadline assignment, combined with the optimization-based technique, and to observe which of them is the most effective.

5.4. Experimental evaluation

For the experimental part, we developed a tool, which receives as input the system model and generates schedules according to the proposed optimization-based technique. The scheduling tool is composed of a genetic engine linked to a real-time system simulator. The genetic engine receives as input the application model composed of tasks and transactions. The transactions are translated into chromosomes. Afterwards, the genetic engine generates populations in search for better individuals (system configurations). At the end of the genetic iteration, it writes the results (chromosomes and corresponding fitness values) in a file.

We use RTMultiSim for simulation. This tool is described in Chapter5. RTMultiSim is a discrete time-stepped simulator for real-time multiprocessor task scheduling and execution. It executes a given system model from an initial moment to a preset maximum simulation time. The simulation results are stored in a database. After an iteration of the genetic algorithm, all individuals in the population are passed to the simulation tool. The individual is translated into a simulation model. The simulation is executed for a time equal to twice the transaction set's hyper-period. During simulation, the maximum response times of tasks and transactions are determined. Based on these parameters, the simulation tool computes the fitness value of the individual. Fitness values are supplied to the genetic algorithm, which continues the search with a new generation.

To explore the behavior of the proposed techniques in different scenarios, we use tools for automatic task set generation and transaction set generation that are described in Chapter 5. These tools can generate task/transaction sets with predefined cardinality (number of tasks/transactions) and total utilization factor, ensuring a random distribution of individual task utilization factors.

The optimization-based technique can be used in different situations and scenarios. The most general problem is to find a feasible schedule for distributed transactions. By simplifying the general problem, our technique can find solutions for the allocation of independent tasks on a given multiprocessor platform, or the assignment of time parameters to tasks. This is possible due to the flexibility of the system model and of the simulation tool used for solution evaluation.

We chose to evaluate the optimization-based scheduling technique in the case of distributed transaction scheduling, which is the most complex of the scenarios mentioned before. To solve this problem, the genetic algorithm must find a good task deadline distribution besides a good task to processor allocation.

Experiments allowed us to adjust the parameters of the genetic algorithm for a faster generation of a good solution. In case of mutation probability the best values was 1% for shorter chromosomes (20-30 genes) and 0.6% for longer chromosomes (more than 100 genes), and for the crossover probability 70%. In some cases, a variable mutation ratio with a tendency of decreasing the probability in every generation had better results. The initial population was set to 60 individuals, as a compromise between variability and algorithm execution time; higher number of chromosomes require more simulation time. We let the population evolve to maximum 1000 generations. We also observed that using intuitive heuristics for the initial population generation, instead of random generation, is a better choice for a faster search.

For the experiments, we generated two types of system models with the workload generation tool presented in Chapter 5. In the first case, we generated system models composed of 6 transactions on 4 processors. In the second case, we generated models composed of 10 transactions on 8 processors. The generated models have partitioned scheduling (a task is assigned to only one processor, the task does not migrate) and don't have any resource restrictions (all tasks are able to be execute on any processor). A transaction contains at most 10 tasks. Transaction execution times and task execution times are generated at random. The number of tasks in a transaction is random between 1 and the maximum number of tasks. The transaction sets have random periods of repetition, while their hyper-periods are less than 1500 discrete time units. The method used for transaction set generation is described in detail in Chapter 5. Our method insures a certain degree of generality for the data sets used for the evaluation. We generated transaction sets with average system utilization of 60%, 70%, 75%, 80%, 85%, 90%, 94%, 97% and 99% (10 models for each utilization configuration).

The obtained results from 3 sets of experiments (on the same data sets):

- With the proposed optimization-based technique applied on both task allocation and deadline assignment sub-problems (OPT);
- With the optimization-based technique composed with the deadline assignment algorithm in [9] (ORDER-OPT);
- With the optimization-based technique composed with the deadline assignment heuristic that divides the laxity time between tasks (equation 10), which is similar to [8] and [9] (LAX-OPT).

A feasible solution has the fitness less than 1000.

For the evaluation, the following metrics were used:

- Success rate: ratio between the number of transaction sets for which a feasible scheduling solution was found, and the total number of transaction sets.

- Average number of iterations: average number of iterations until a feasible scheduling solution is found. The runs that don't find a feasible solution are not considered. This measure shows how fast a feasible solution is found.
- Average fitness: average fitness of feasible scheduling solutions.

Fig. 9 shows the success rate of the three proposed optimization-based approaches for 6 transactions on 4 processors as a function of system utilization (a description of system load). We use LAX-EDF scheduler, which we described in the previous section, as reference. All three algorithms have better success rate by at least 30%. LAX-OPT has slightly better results than OPT, while ORDER-OPT has the worst performance. OPT and LAX-OPT find feasible solutions for all transaction sets with utilizations up to 90%. Between 90% and 99%, the success rate drops from 1 to 0. Fig. 10 shows the success rate for each approach for 10 transactions on 8 processors. OPT and LAX-OPT find feasible solutions for all transactions sets in the test set for utilizations of up to 70%. There is an average of 30% improvement compared to LAX-EDF.

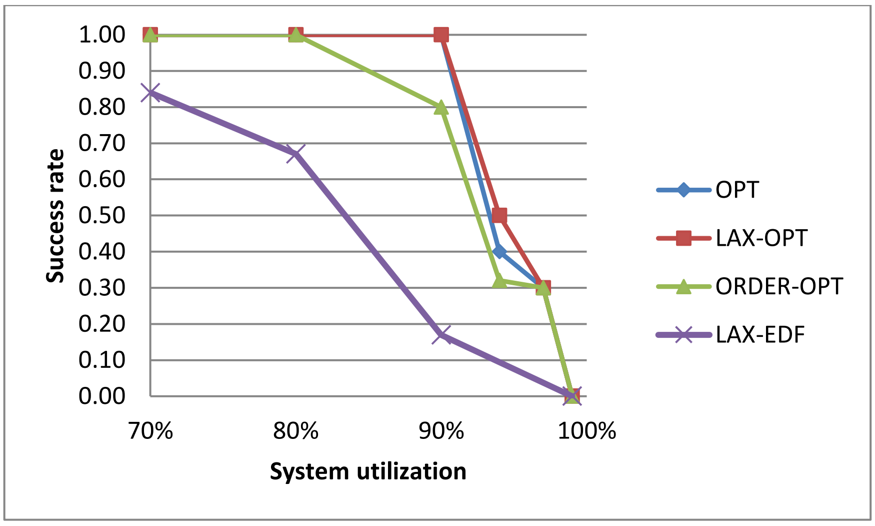


Figure 9. The success rates of the proposed techniques, for 6 transaction on 4 processors

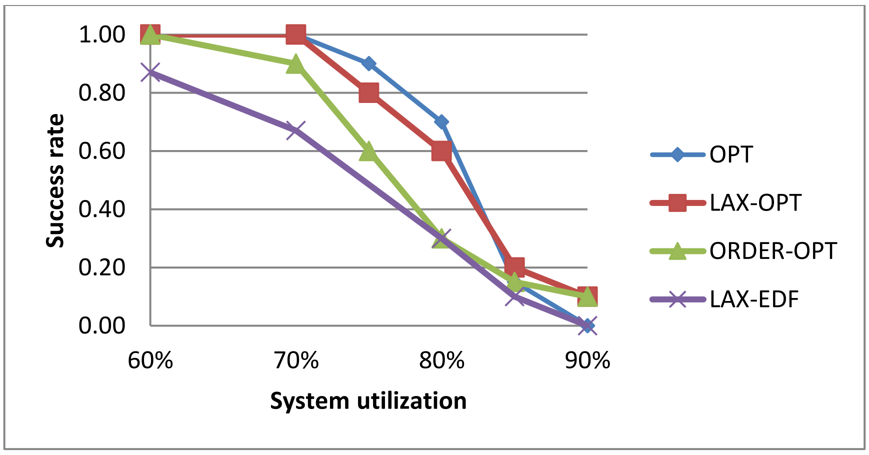


Figure 10. The success rates of the proposed techniques, for 10 transaction on 8 processors

Fig. 11 and 12 show the average number of steps executed by the genetic algorithm until a feasible solution is found, as a function of system utilization. It may be observed that for 6 transactions OPT has the best results, but note that for utilizations greater than 90%, it finds less feasible solutions than LAX-OPT. For the case with 10 transactions OPT and LAX-OPT find feasible solutions in fewer steps.

Fig. 13 and 14 show the average fitness of feasible solutions found with our approach, as a function of system utilization. In the case of 6 transactions on 4 processors, the solutions obtained by OPT have the best fitness. The solutions obtained by LAX-OPT have the worst fitness between utilizations of 80% and 97%, but, on the same interval, LAX-OPT has the highest success rate. In the case of 10 transactions on 8 processors ORDER-OPT has the best average fitness. However, considering that ORDER-OPT has the worst success rate we consider that the best overall results are obtained by LAX-OPT.

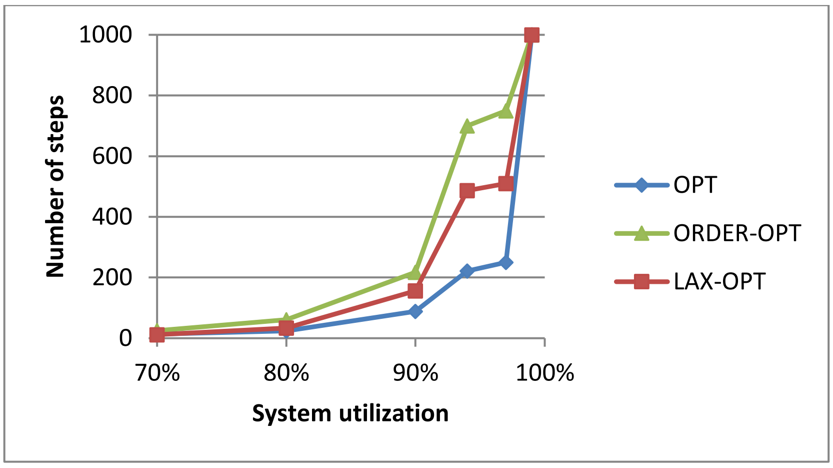


Figure 11. Average number of steps executed until a feasible solution is found, for 6 transaction on 4 processors

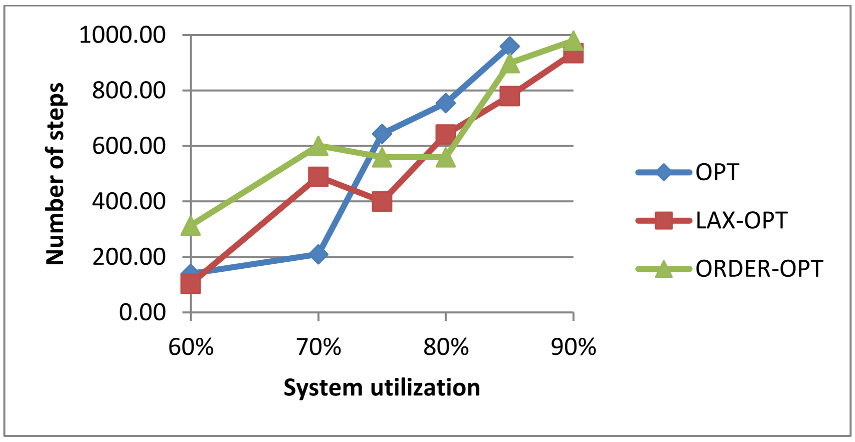


Figure 12. Average number of steps executed until a feasible solution is found, for 10 transaction on 8 processors

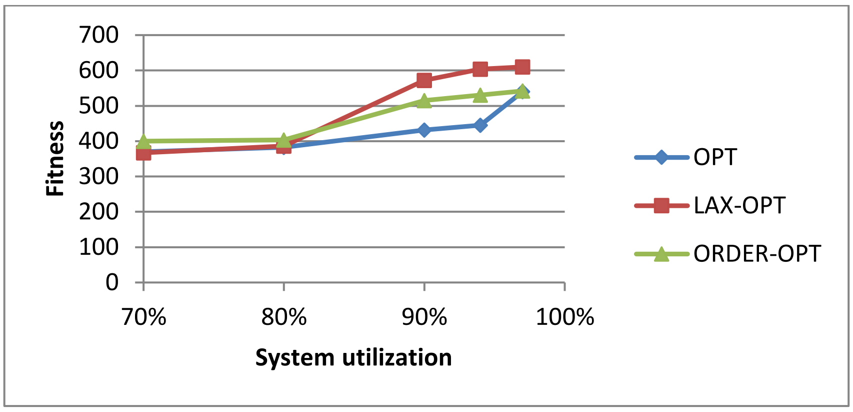


Figure 13. Average fitness of feasible scheduling solutions found, for 6 transaction on 4 processors

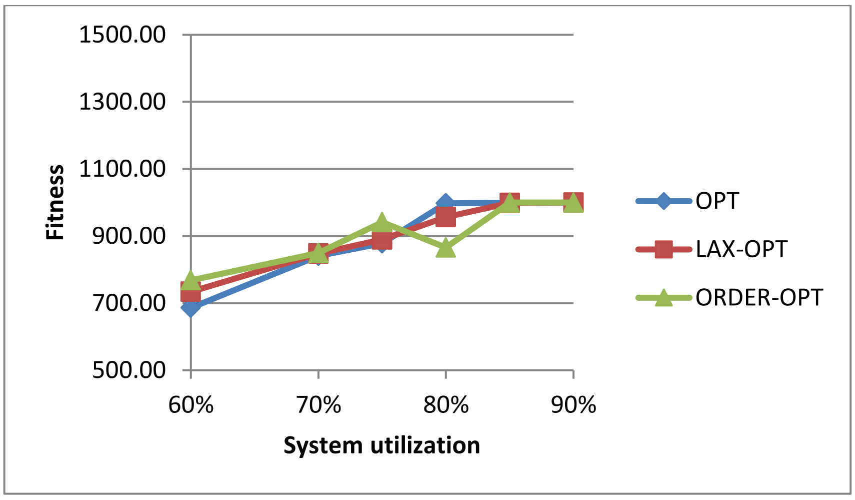


Figure 14. Average fitness of feasible scheduling solutions found, for 10 transaction on 8 processors

At this point, we can conclude that the optimization approach is efficient as it improves the results to as much as 80% for some data sets, compared to an algorithm that does not use optimization techniques, like LAX-EDF. Overall, the best average performance is obtained by OPT, but in some cases, LAX-OPT finds more (5-10%) feasible solutions than OPT. During our experiments, we observed that LAX-OPT and ORDER-OPT perform better for large problems, while for small problems they fail much more often that OPT.

We further evaluate the performance of our optimization-based approach by making a performance comparison with related work. Azketa et. al. solve in [12], [13] and [14] a similar problem to ours by using a genetic algorithm. In [12] they use the genetic approach to optimize the scheduling solution found by an iterative algorithm that assigns static priorities to tasks in a transaction set. In [13] and [14] they extend their work and solve the task to processor allocation at the same time with priority assignment. We highlight the differences between our approach and the one presented in [14] in Table 1.

Table 1. OPT vs GA-Azketa

Characteristics	**OPT**	**GA-Azketa**
Workload model	Chain transactions	Chain transactions
Platform model	**Identical multiprocessor**	Heterogeneous multiprocessor
Resource restrictions	Yes	Yes
Scheduler	**EDF**	FTP
Chromosome	**Genes for processor allocation and task deadline**	Gene for processor allocation, priority is given by the gene's position.
Operators	**Mutation, Crossover**	Mutation, Crossover, Clustering
Fitness function	**Minimize transaction response time and task response time; Uniform processors' utilization**	Minimize transaction response time and resource utilization
Fitness evaluation	**Fitness computed from simulation results**	Response time computed by holistic schedulability analysis

The authors evaluate their work in [14] with only one transaction set for which they gradually increase the execution time of inner tasks to generate greater load, while keeping the other characteristics (repetition period, deadline, number of tasks) unchanged. In our opinion, our evaluation method is more general. However, because their implementation and test data set are not available, we replicated their tests using the description in the article and we applied OPT. We found that our approach starts failing at 67% load on this particular data set. They report that their genetic algorithm starts failing at 67% load, too. We can conclude that our approach has similar performance in terms of success rate, on their evaluation data set.

In [13] the authors make a statistical evaluation of their approach by randomly generating 2 types of transaction sets, small (6 transactions) and large (12 transactions). They gradually increase the tasks' execution time to obtain increasing system loads. Their best success rates are for small systems with 6 transactions on 4 processors. In those cases, their algorithm started failing at system loads of around 70%. For large systems, their algorithm starts failing at loads of around 60%. Our approach seems to be better in terms of success rate, since it starts failing at over 70% systems loads, for larger systems (10 transactions on 8 processors).

6. Conclusions

This chapter presents two solutions for multiprocessor scheduling of real-time transactions.

The first solution is best suited for small embedded systems, as it minimizes the computation overhead introduced by a scheduler. Performance evaluation shows that the cyclic executive-based approach is slightly better than a global EDF runtime scheduler, but only for small systems (e.g. 6 transactions on 4 processors). For larger systems, the runtime EDF scheduler is better. In terms of execution time, we conclude that our schedule generation technique is faster than the runtime scheduler, and faster than other cyclic schedules generators know of.

The second solution uses a genetic algorithm combined with a simulation tool to find feasible system setups. By system setups, we mean:

- Task to processor allocation
- Intermediate tasks deadline assignment

The genetic algorithm searches through the possible solutions, while the simulation tool evaluates the candidate solutions by means of the fitness function. We demonstrated through experiments that our approach improves non-iterative scheduling techniques by at least 30%. Compared to other similar optimization-based approaches, we estimate that our technique is at least 10% better in terms of scheduling success rate.

References

[1] C.L. Liu, J.W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, Journal of the ACM, Vol. 20, No. 1, pp.46-61. , 1973

[2] Aloysius Mok, "*Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*", PhD Thesis, MIT, 1983

[3] R.I. Davis, A. Burns, "A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems." techreport YCS-2009-443, University of York, Department of Computer Science, 2009.

[4] M. Bertogna, S. Baruah, Tests for global EDF schedulability analysis, Journal of Systems Architecture, no. 57, pp. 487–497, 2011.

[5] S. Baruah, N. Cohen, G. Plaxton, D. Varvel, Proportionate progress: A notion of fairness in resource allocation., Algorithmica 15, 6, pp600–625, 1996.

[6] I. Lupu, P. Courbin, L. George, and J. Goossens, "Multi-Criteria Evaluation of Partitioning Schemes for Real-Time Systems", 15th International conference on Emerging Technologies and Factory Automation, ETFA'2010, Bilbao, Spain.

[7] N. Serreli, G. Lipari, E. Bini, "Deadline assignment for component-based analysis of real-time transactions", 2nd Workshop on Compositional Real-Time Systems, Washington, DC, USA, 2009.

[8] M. Di Natale, J.A. Stankovic, "Dynamic end-to-end goarantees in distributed real-time systems", Proceeding os the 15th IEEE Real-Time Systems Symposium, pp.215-227, 1994.

[9] B. Kao, H. Garcia-Molina, Deadline assignment in a soft real-time system, IEEE Transactions on Parallel and Distributed Sysetms, vol.8, no.12,pp.1268-1274, 1997.

[10] K. Tindell, A. Burns, and A. Wellings, Allocating Hard Real-TimeTasks: An NP-Hard Problem Made Easy, Real-Time Systems, vol. 4, no. 2, pp. 145-165, 1992.

[11] J. J. Gutiérrez García, and M. González Harbour, "Optimized Priority Assignment for Tasks and Messages in Distributed Hard Real-Time Systems", in Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems. IEEE Computer Society, 1995, p. 124.

[12] E. Azketa, J. Uribe, M. Marcos, L. Almeida, and J. Javier Gutierrez, "Permutational genetic algorithm for the optimized assignment of priorities to tasks and messages in distributed real-time systems", 2011 International Joint Conference of IEEE TrustCom-11/IEEE ICESS-11/FCST-11

[13] E. Azketa, J. Uribe, M. Marcos, L. Almeida, and J. Javier Gutierrez, "Permutational genetic algorithm for fixed priority scheduling of distributed real-time systems aided by network segmentation," in Proceedings of the 1st Workshop on Synthesis and Optimization Methods for Real-time Embedded Systems (SOMRES), 2011.

[14] E. Azketa, J. Javier Gutierrez, M. Marcos, L. Almeida, "Permutational genetic algorithm for the optimized mapping and scheduling of tasks and messages in distributed real-time systems," Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, TRUSTCOM 2011.

[15] Y. Monnier, J.P. Beauvais, A.M. Deplanche, "A Genetic Algorithm for Scheduling Tasks in a Real-Time Distributed System", Euromicro Conference, 1998, pp.708-714

[16] J. Oh, C. Wu, “Genetic-algorithm-based real-time task scheduling with multiple goals”, Journal of Systems and Software, Volume 71, Issue 3, May 2004, Pages 245-258

[17] M. Yoo, M. Gen, “Scheduling algorithm for real-time tasks using multiobjective hybrid genetic algorithm in heterogeneous multiprocessors system”, Computers & Operations Research, Volume 34, Issue 10, October 2007, Pages 3084-3098

[18] P.E. Hladik, H. Cambazard, A.M. Deplanche, and N. Jussien, Solving a Real-Time Allocation Problem with Constraint Programming,J. Systems and Software, vol. 81, no. 1, pp. 132-149, 2008.

[19] J.M. Rivas, J.J. Gutierrez, J.C. Palencia, M.G. Harbour, “Schedulability Analysis and Optimization of Heterogeneous EDF and FP Distributed Real-Time Systems,” 23rd Euromicro Conference on Real-Time Systems, ECRTS, 2011.

[20] C. Douglas Locke. “Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives.” Real-Time Systems, 4(1):37–53, 1992.

[21] A. Mok, X. Feng, and D. Chen. Resource partition for realtime systems. In Real-Time Technology and Applications Symposium, pages 75–84, 2001.

[22] Theodore P. Baker and Alan C. Shaw. The cyclic executive model and Ada. Real-Time Systems, 1(1):7–25, 1989.

[23] Almeida, L.; Tovar, E.; Fonseca, J.A.G.; Vasques, F.; , "Schedulability analysis of real-time traffic in WorldFIP networks: an integrated approach," *Industrial Electronics, IEEE Transactions on* , vol.49, no.5, pp. 1165- 1174, Oct 2002

[24] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. “Mixed-Criticality Real-Time Scheduling for Multicore Systems.” In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology* (CIT '10). IEEE Computer Society, Washington, DC, USA, 1864-1871.

[25] A. P. Ravn, M. Schoeberl, “Cyclic Executive for Safety-Critical Java on Chip-Multiprocessors”, The 8th International Workshop on Java Technologies for Real-time and Embedded Systems JTRES’10 August 19–21, 2010 Prague, Czech Republic

[26] UPPAAL tool, http://www.uppaal.org/

[27] L. Cucu-Grosjean, J. Goossens, Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms, Journal of Systems Architecture, 2011.

Chapter 5. Real-time systems simulation

1. Introduction

Multiprocessor systems are becoming the execution environment for most of today's real-time applications. In consequence, developers need theoretical methods and experimental tools for evaluating the feasibility and time behavior of such systems. As showed in many recent papers [1][2], real-time analysis on multiprocessor systems is not a trivial task, and in the general case when different restrictions (synchronization, casual dependencies, data race conditions, etc.) are considered, beside the time conditions, the problem has a non-polynomial complexity.

In this context, empirical methods like simulation have become a more practical alternative to complex mathematical schedulability analysis. Even though simulation is mainly used to assess systems' defects, it can be used to generate system schedules during a preset time interval. The authors of [3] and [4] showed that fixed priority algorithms (including RM and EDF) generate periodic schedules in the case of feasible periodic task systems. Therefore, an exact schedulability test would be to check the task set's schedulability during this period. This type of test can be implemented in practice through a simulation tool.

On the other hand, empirical methods such as simulation are used for the evaluation of new scheduling techniques. As we already mentioned in Chapter 1, the lack of standard methodologies and tools is an issue in the area of real-time systems. When we first started to evaluate our work in the area of real-time multiprocessor scheduling, we noticed that there were just a few multiprocessor simulation tools. We were not able to use any of those simulators, since some were very hard to extend with new features and new algorithms, while others were not even available for public use. Therefore, we started to develop our own simulation tool.

The goal was to develop a real-time simulator that can cover a multitude of cases from parallel architectures to distributed ones and from independent task sets to transactions or fork-join parallel tasks. We also included aspects of network communication.

The resulting tool called *RTMultiSim* is a discrete time simulator that can be used to measure the time parameters of such systems and in predefined scenarios to demonstrate the feasibility of a real-time scheduling policy. It can be useful in evaluating the statistical influence of different parameters (e.g. CPU utilization, parallelism degree) over the real-time behavior of multiprocessor systems. In addition, we developed a synthetic workload generation tool, which provides input for the simulation tool.

The simulation tool is useful in system modeling and design phases in order to establish the number of required processors, the maximum utilization/load factor, the worst-case response time of critical tasks, or to demonstrate the feasibility of a given setup.

2. Literature review

In the field of real-time system's analysis and simulation, there are a number of solutions and tools, offered as open source software (e.g. STORM [5], MAST [6]) or as commercial products (e.g Simulink

[12], SymTA/S [13]). The first ones are more generic and are dealing with the real-time behavior of systems at higher abstraction levels. The commercial ones are more related to some pragmatic solutions or to specific platforms.

The differences between these tools are regarding the following aspects:

- The workload model used in simulation – types of executing units (tasks, threads), periodicity of jobs, processor affinity, clustering, etc.
- The execution environment model – uniprocessor, parallel or distributed systems, uniform or heterogeneous execution, with or without communication (networking).
- The scheduling model – fixed or dynamic priorities, time-triggered or event-triggered.
- The real-time parameters and restrictions model – discrete time, global or local time.
- Non-real-time conditions accepted in the model – task synchronization, causal dependencies, and concurrent access to common resources.

In our approach, we tried to cover as many scenarios as possible, allowing seamless variation between different models. The above-mentioned model types may be obtained as particular cases through the tool's parametric configuration. This is not the case for a number of existing simulation tools, specialized for a given workload and system model.

There are several simulation or analysis tools for real-time systems, but many do not have proper documentation, they are not extendable (e.g. add new scheduling policies), nor open source. Moreover, there isn't any tool imposed as standard in this area. That is why many researchers choose to develop their own performance evaluation tools. As result of this context, there are a lot of very specialized tools that work only for a specific workload model or test only a specific problem.

We will mention four simulation tools that have documentation, seem to be extendable, some of them are open source and provide automatic task set generation. These tools are STORM [5], MAST [6], FORTAS [7] and YARTISS [8].

The closest tool to *RTMultiSim* is STORM (Simulation tool for real-time multiprocessor scheduling) [5]. STORM can handle multiprocessor architectures and data exchange between periodic or aperiodic tasks. The simulated system's description is specified through an XML file that contains simulation parameters, tasks set, data exchanged by tasks, CPU and scheduler specification. Compared to our tool, this simulator does not cover intra task parallelism (e.g. multi-threading or fork-join model). It uses only global scheduling strategies and it does not allow partitioned and clustered scheduling approaches. STORM isn't reported to have a task set generation tool. STORM is written in Java, it is not open source but one can easily write their own scheduler as long as they don't need a different task or platform model.

Another similar tool is YARTISS [8], an event-based real-time systems simulator obtained as result of the extension and redesign of RTSS simulator [9]. This tool is able to handle real-time periodic tasks (the Liu & Layland model) and transactions (graph) executed on multiprocessor systems. The task model is augmented with energy related parameters. The scheduling policies implemented are uniprocessor and global multiprocessor variants of RM, DM, EDF and LLF. Moreover, this tool provides a task set generator based on the UUniFast-Discard [11] algorithm and simulation results visualization facilities. YARTISS is written in Java and is extendable as it is open-source.

MAST (Modeling and analysis suite for real-time applications) [6] allows the analysis of uniprocessor and complex distributed systems. It uses an event-triggered execution model with the possibility to handle complex task dependencies. As scheduling strategies, it supports fixed priority and EDF. The main features of MAST are worst-case response time schedulability analysis, sensitivity analysis and optimized priority assignment. The file that contains the model received by MAST as input is quite complex. Moreover, they do not provide automatic model generation. Due to the complexity of the input model specification and the lack of an automatic model generator, some will argue that MAST is difficult to use for performance evaluation. Because it is written in Ada, many will consider MAST hard to extend.

FORTAS (Framework for real-time analysis and simulation) [7] is a real-time system analyzer and simulator. It offers functionalities for feasibility testing of various multiprocessor scheduling algorithms as well as for viewing task schedules with different uniprocessor and global multiprocessor scheduling algorithms (RM, DM, EDF, LLF and PF). It also provides a task set generation tool that uses UUniFast algorithm. The task generator has as parameters the interval for task priorities (and deadlines) and the distribution for task utilizations. FORTAS is written in Java, but it is not open source and consequently, not extendable.

Compared to MAST and FORTAS, which are focused on analytical scheduling evaluation, our approach is mainly concentrated on simulation-based evaluation offering step-by-step information about the evolution of the system under test. Our tool implements a clustered scheduling approach and it allows intra-task parallelism. Moreover, in the same simulated system, different scheduling strategies can be defined for each particular executing element (processor, or network).

Another research problem closely related to real-time systems performance evaluation and simulation is the automatic generation of task sets. Researchers use various methods to generate tasks sets that are used to demonstrate the performance of scheduling algorithms. Some of these methods can produce biased experimental results, as shown in [10]. This happens in the context in which there aren’t any reference task sets to be used as benchmarks, or any standard methods for conducting the performance evaluation of real-time systems.

The authors in [10] identify the requirements of automatic task set generation. In their opinion, the tool has to be efficient, independent and unbiased. The efficiency refers to the number of generated task sets that has to be large in order to achieve statistically significant results. Independence refers to the possibility of generating task sets by varying certain parameters such as number of tasks or task set utilization independent of other (constant) parameters. Finally, the distribution of the generated task sets should be equivalent to randomly choosing a task set from all possible ones, and discarding those that do not match the predefined parameters.

For task sets that are used to validate uniprocessor systems (total utilization equal to 1), there is a very good method based on the UUniFast algorithm presented in [14]. This method was extended to multiprocessor task sets (total utilization can be greater than 1) by [11]. But the UUniFast-Discard algorithm proposed in [11] proved inefficient when the total utilization is equal to half the task set cardinality (nearly all task sets are discarded). The authors in [10] argue that Randfixedsum algorithm is appropriate to be used for multiprocessor task sets generation and provide a python implementation of the generator. Randfixedsum is able to efficiently generate a predefined number of task utilization values that add up to a predefined total utilization. We will demonstrate later in this Chapter that the algorithm we propose for task sets automatic generation is comparable to Randfixedsum, while easier to implement. Moreover, we propose an algorithm that generates chain transactions.

In the following sections, we describe the simulation environment that we developed.

3. The simulation environment

The *RTMultiSim* simulation environment provides a set of tools that aim to assist researchers or real-time systems designers in:

- Assessing new scheduling methods,
- Comparing existing scheduling methods,
- Studying the behavior of real-time applications in different execution scenarios
- Finding settings (e.g. timing parameters assignation, processor allocation) that would improve the real-time applications' schedulability

As seen in Fig.1, the *RTMultiSim Simulation Tool* is the main component. It receives as input specifications of the workload and platform models, executes the simulation and stores the results in a database. The simulation results (e.g. statistics, execution traces) are later manually interpreted, or viewed using the *Visualization Module*. The workload and platform models and other simulation settings have to be specified in XML files that have predefined structures. Alternatively, the user can configure the platform model and the simulation settings configured through the graphical interface.

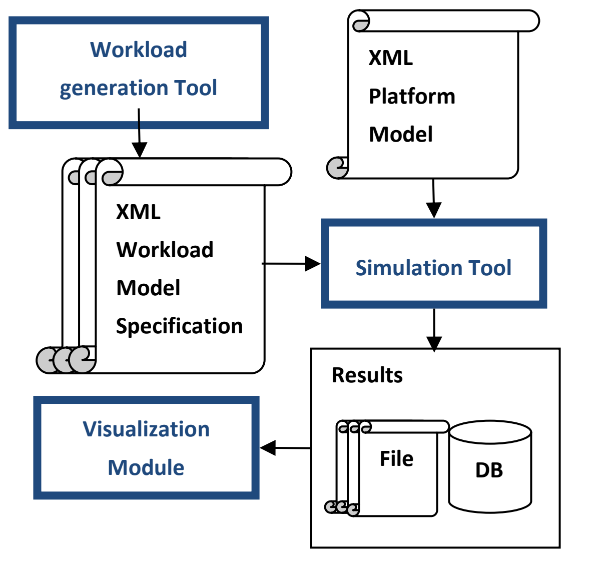


Figure 1. The RTMultiSim simulation environment

The *Workload Generation Tool* automatically generates synthetic task or transaction sets that can be used for statistical analysis of real-time systems, according to some predefined parameters (e.g. total utilization, hyperperiod, cardinality). The generated workload models are produced in the XML format required by the Simulation Tool.

The *Visualization Module* provides a graphic representation of the simulation results. The user can view the execution trace of a task set and its corresponding jobs on processors and general information about the simulation (e.g. number of migrations, number of deadline misses, processors utilizations).

The design and main features of the *RTMultiSim* simulation environment will be detailed as follows.

4. The simulation tool

The workload and platform models specifications as well as the simulation settings received as inputs are transformed into components of the Simulation Tool, as shown in Fig. 2. The simulator executes the task set on the platform's available CPUs according to the selected scheduling strategy. During the simulation process, the simulator records relevant time parameters related to the behavior of the system that are later stored as results.

The characteristics of the workload and platform models implemented in the simulator, as well as the simulation engine will be described next.

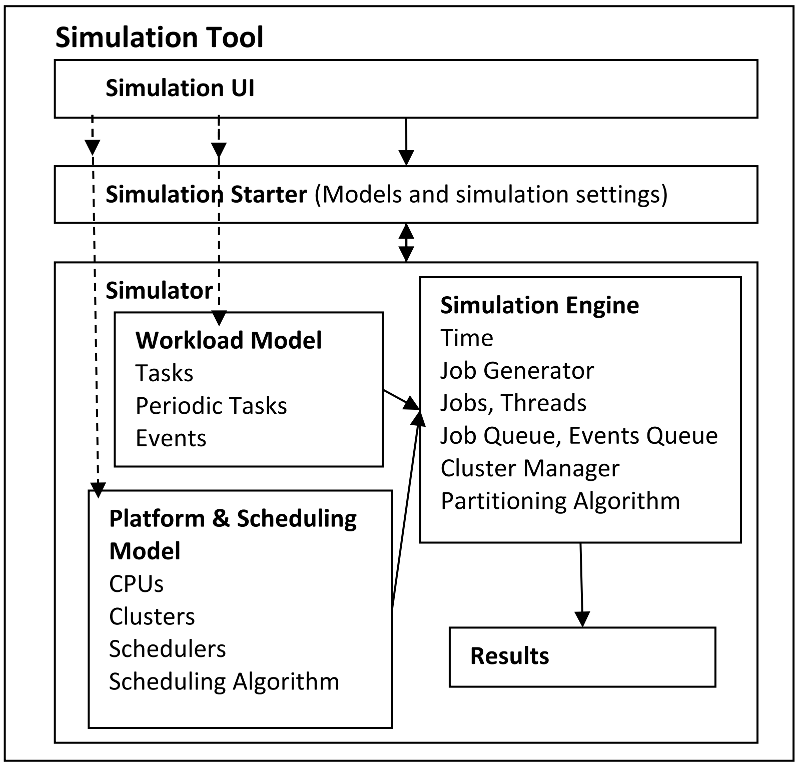


Figure 2. The RTMultiSim simulation tool components

4.1. The Workload Model

The workload model used in *RTMultiSim* allows the representation of a variety of real-time applications that fall in the following general categories: sequential (single threaded), parallel (multi threaded) and distributed (with network communication).

We propose a task model that is general enough to represent precedence dependencies and parallel execution and that, in the most simplified case, is able to represent sequential independent tasks that have a repetition period. We have already described the workload model in Chapter 2, so we will use that description as reference.

The *RTMultiSim* workload model is implemented as a set of tasks with dependencies like depicted in Fig. 3. The model contains periodic and aperiodic tasks that produce and consume *events*. We implement the dependencies between tasks with events. A task can produce events, which are consumed by other tasks, creating an execution dependency between producer and consumer. With this model, we can represent chain or graph transactions.

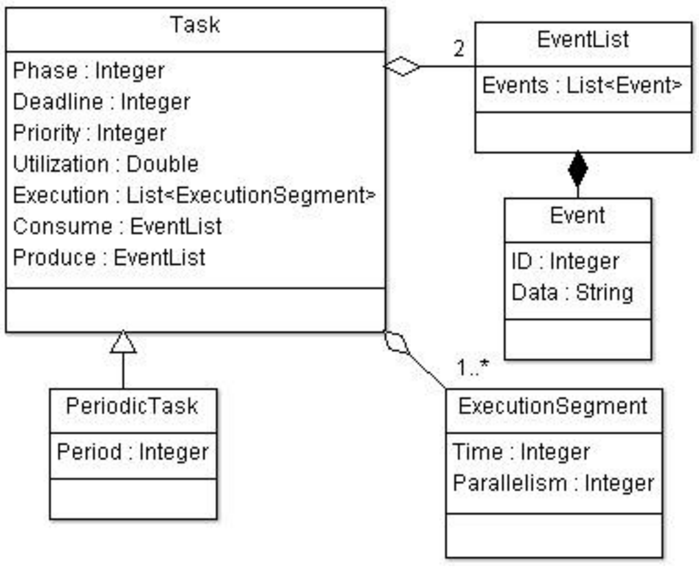


Figure 3. Workload model implementation (class diagram)

Fig.4 shows five tasks that create a graph transaction. Each task has P (produce) and C (consume) event lists. The execution precedence between tasks is created by the fact that a tasks that has to consume an event will be blocked (not scheduled for execution) until that event is produced and put in the *global event queue*.

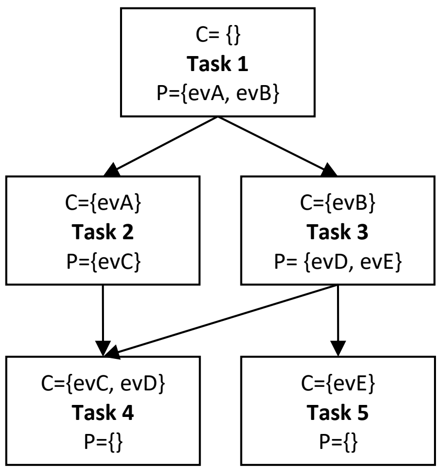


Figure 4. Tasks that create a transaction through precedence dependencies

The task's execution requirements are represented as a sequence of *execution segments*. An execution segment may be a sequential portion of a task and generate a single thread, or it can generate multiple threads that could be executed in parallel on a number of processors (if available). Each thread will have the execution time of the segment to which it belongs. Depending on the execution model, the task can be fork-join, or multiframe. In the case of a fork-join task, the execution segments represent the task's sequential and parallel portions. These portions are executed in the same order each period. In the case of multiframe tasks, the execution segment represents the execution behavior during a period. Each task instance will have the execution requirements of a single execution segment, in the order given by the execution segments list. Fig. 5 shows the difference between the fork-join execution model and the multiframe execution model. Note that our task model supports both execution models, but in our experiments, we used only the fork-join model.

4.2. The Platform and Scheduling Models

The platform is modeled as a sum of identical processing units (CPUs). The execution speed of all tasks is the same on all processors. Job context switch time and job migration time can be taken into consideration as constant values. This type of platform model can represent parallel systems as well as distributed systems. In the case of distributed systems, the networks are represented as processing units.

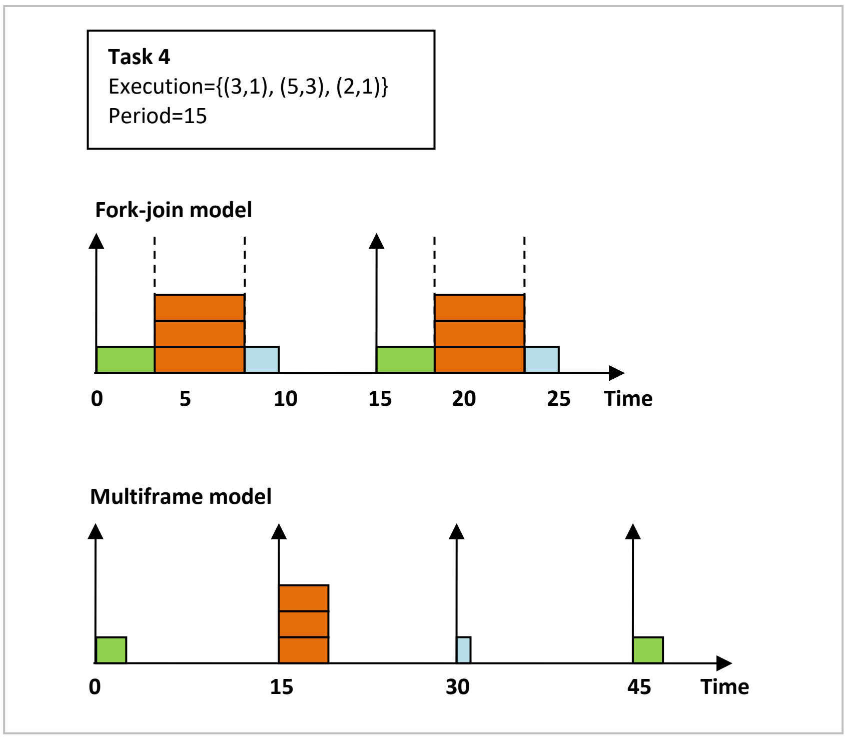


Figure 5. Two uses of execution segments: fork-join parallel tasks and multiframe tasks

From the allocation point of view, there are three main approaches for multiprocessor scheduling:

- Global
- Partitioned
- Clustered

In the global approach, jobs can be allocated to any available processor and can migrate to other processors during execution with no restrictions. Partitioned scheduling assumes that each processor has its own scheduler and job queue. A task is allocated to a single processor. In clustered scheduling, job migration is restricted to a subset of the available processors. Processors are grouped into clusters. Each cluster has its own scheduler and job queue. First, tasks are allocated to clusters, and then each cluster scheduler globally schedules jobs inside the cluster.

In *RTMultiSim*, we implemented the *loose clustered* approach, which we described in Chapter 2. We can configure the scheduling mechanism to be partitioned, global, clustered or loose clustered.

The *RTMultiSim* implementation of the scheduling model (Fig. 6) contains a cluster manager and a set of clusters. The cluster manager applies a partitioning heuristic to allocate all tasks to the existing clusters. If there is only one cluster (global scheduling), all tasks are allocated to that cluster.

A *cluster* contains the allocated task set, the list of CPUs and two schedulers (the cluster's own scheduler and the alternative scheduler). The alternative scheduler can be excluded if the user does not want to use the loose clustered mechanism. The local scheduling algorithm decides the execution order of jobs. In our model, each cluster can use a different scheduling algorithm.

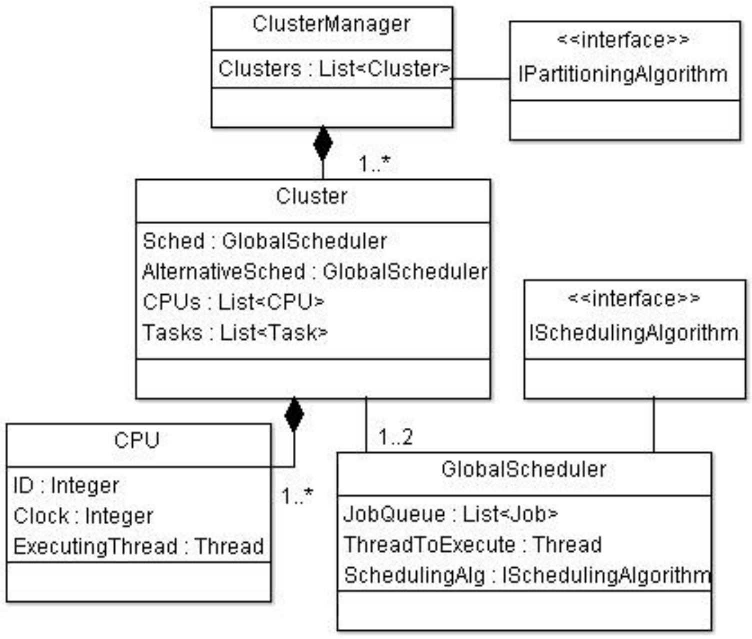


Figure 6. The RTMultiSim scheduling implementation (class diagram)

The partitioning heuristics available in *RTMultiSim* are: *Next Fit* and *Affinity*. In *Next Fit*, tasks are sorted in a decreasing order according to their utilization factor (execution time divided with period). Each time, the first task is allocated to the next available cluster. In *Affinity*, tasks with largest utilization factor and the shortest CPU Affinity list are allocated first. Each task is allocated to the first available cluster, which contains a CPU from its CPU Affinity list. New partitioning heuristics can be added by writing classes, which implement a predefined abstract interface (*IPartitioningAlgorithm*).

In *RTMultiSim*, we implemented priority-based scheduling algorithms. A scheduling algorithm of this category chooses the jobs with the *n* highest priorities to be executed on *n* available processors, after it has computed the priority for each job. The available scheduling algorithms are: *Rate Monotonic* (RM), *Earliest Deadline First* (EDF), *Least Laxity First* (LLF) and *First In First Out* (FIFO). *RTMultiSim* supports preemptive (e.g. RM, EDF, LLF) and non-preemptive (e.g. FIFO) scheduling algorithms. The simulator may be extended with other user-defined scheduling algorithms by creating classes, which implement a predefined abstract interface (*ISchedulingAlgorithm*).

4.3. The Simulation Engine

The simulation is performed for a fixed time interval or until the feasibility interval is covered. For instance, according to [3] and [4] the feasibility interval for fixed priority algorithms on multiprocessor systems is a multiple of the task set hyper-period. The simulation engine receives as input the system model in order to perform the simulation. The most important components of the simulation engine are:

- System model (tasks, CPUs, schedulers)
- Clock
- Job generator

- Jobs and threads

Simulation time and job generation. The simulation time is modeled as a *global clock*. All CPUs are synchronized to this clock. The global clock advances every time with one simulation time unit (STU). At each clock tick (transition), the state of the system is recomputed. There can be new job releases and, depending on the scheduling strategy, there can be schedule updates that generate preemptions. A job's execution time is equal to an integer number of STUs. In a time step, each CPU executes exactly one time unit from the total execution time of a running job.

The *job generator* creates new jobs according to the workload model. After each global clock transition, this component gets from the workload model all the tasks that have to release new jobs at the current simulation time. Based on those tasks, it creates new jobs, which are copied in the *global job queue*.

Jobs and threads. In *RTMultiSim*, jobs inherit all task parameters. The job release time is set to its creation time. Job execution is performed according to the task's sequence of execution segments.

A job contains a list of *threads*, which can be started and executed in parallel. Threads are created at the start of an execution segment. When all the threads in an execution segment are completed, the threads for the next segment are created. A thread can pass through several states during its existence, from creation to completion. The possible thread states are:

- *Ready*
- *Scheduled*
- *Running*
- *Blocked*
- *Completed*

In "ready" state, a thread is prepared to be scheduled. If it was chosen by the scheduler and assigned to a CPU, the thread is in "scheduled" state. During execution, the thread is in "running" state. In the thread is preempted, it returns in "ready" state. The thread is "blocked" if it waits for an event to be produced, in order to start or resume its execution.

Scheduling. The cluster manager takes the jobs from the global job queue and places them in the clusters' job queues, according to task partitioning. Then, each cluster's scheduler chooses the jobs that will be executed on each of its CPUs, according to the scheduling algorithm. Threads belonging to the same job have the same priority. Ready threads will not interrupt running threads that belong to the same job. A multithreaded job can be allocated to more than one CPU at the same time. For real-time transactions, deadlines have to be assigned to intermediate tasks before these will be scheduled. The simulator has some heuristic algorithms that assign deadlines to tasks. It also has an extension that finds optimized deadlines by using a genetic algorithm.

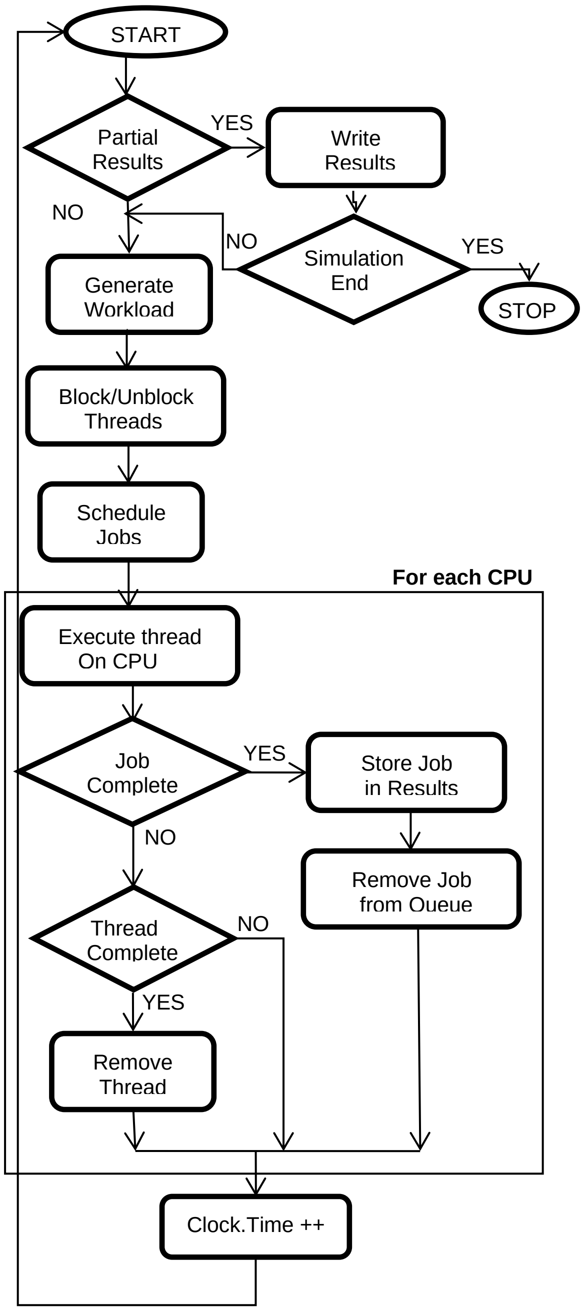


Figure 7. Simulation execution flowchart

Execution. A simulation execution step, as shown in Fig. 7, starts with job generation. Generated threads that do not meet the requirements to enter ready state (e.g. the thread can start only if a certain event is produced) are blocked. Blocked threads that meet the requirements to enter the ready state are unblocked.

New jobs, if any, are placed in the clusters' job queues. Each cluster scheduler allocates jobs (threads) to its CPUs. Next, the threads are executed on CPUs. During thread execution, events can be consumed or produced, execution time is increased and execution statistics are recorded. If the thread is completed, it is removed from the jobs' current threads list. If the job is completed, it is removed from the cluster's queue and placed in the results.

Finally, the simulation time is increased, and the simulation execution moves to the next time step.

Results. Simulation results are periodically written in a database, until simulation completion. Results for each simulation are recorded separately. For each released job, the simulator records the response times and deadline. For each thread, release time, start time, completed time, execution time, number of migrations and CPU visitation sequence are recorded. Based on the recorded raw data, a number of statistical parameters may be computed such as: the number of successfully finished tasks, number of deadline misses, number of migrations, effective utilization of CPUs, etc.

4.4. Simulation examples

RTMultiSim can simulate a variety of systems, defined through configuration parameters, without any changes in the code. From the platform point of view, these systems can span from multiprocessor (parallel) to distributed architectures. The scheduling strategies may be global, partitioned, clustered or loose clustered. The workload can be represented as independent parallel or sequential tasks, periodic or aperiodic tasks and distributed transactions.

The workload and other simulation parameters such as number of processors, cluster configuration, scheduling and partitioning strategies are specified in XML or text files.

An independent task set may have the following specification:

```
<Tasks>
  <Task type="periodic" id="1" C="10" T="15" D="12" />
  <Task type="parallel" id="2" C="5,4,6" P="1,3,1" T="15" D="15" />
  …
</Tasks>
```

Where *id* is the task identifier, *C* is the execution time or list of execution times (for each execution segment), *T* is the period, *D* is the deadline, and *P* is a list of parallelism values, one for each execution segment.

A transaction is specified as follows:

```
;60,60
t1 child m1
m1 child t2
t2 child
```

Where the first two values are the period and the deadline *t1*, *t2* are tasks and *m1* is a message. For each task or message, the user has to specify the execution time and the processors (networks) on which they can execute like:

```
t1, 23, 1, 2
t2, 11, 2
m1, 10, 0
```

Task t1, for example, has execution time equal to 23 and can be executed by processors with id's 1 and 2.

A DAG transaction is specified as follows:

```
;30,40
t1 child t2, t3
t2 child t4
t3 child t4
t4 child
```

Task t1 has two children, t2 and t3 that have a common child, t4.

The platform and other simulation parameters can be specified in an XML file or configured through the simulation tool's GUI. An example of platform XML is as follows:

```
<Platform>
<CPU No="4" Cluster="{0 1 2 3}" Sched="EDF" Part="Affinity" LinkClusters="false" />
<Settings WriteResults="true" SimTime="5000"/>
</Platform>
```

"Cluster" attribute describes the scheduling approach. In the previous example, it is the global approach. For partitioned approach, each CPU ID is put in separate brackets like "{0}{1}{2}{3}". For clustered approach, the CPU IDs are grouped (one pair of brackets for each cluster) like "{0 1}{2 3}". "LinkClusters" is "true" if the user chooses the loose clustered approach and "false" if it chooses traditional clustered approach. "Sched" and "Part" are the scheduling and the partitioning algorithms.

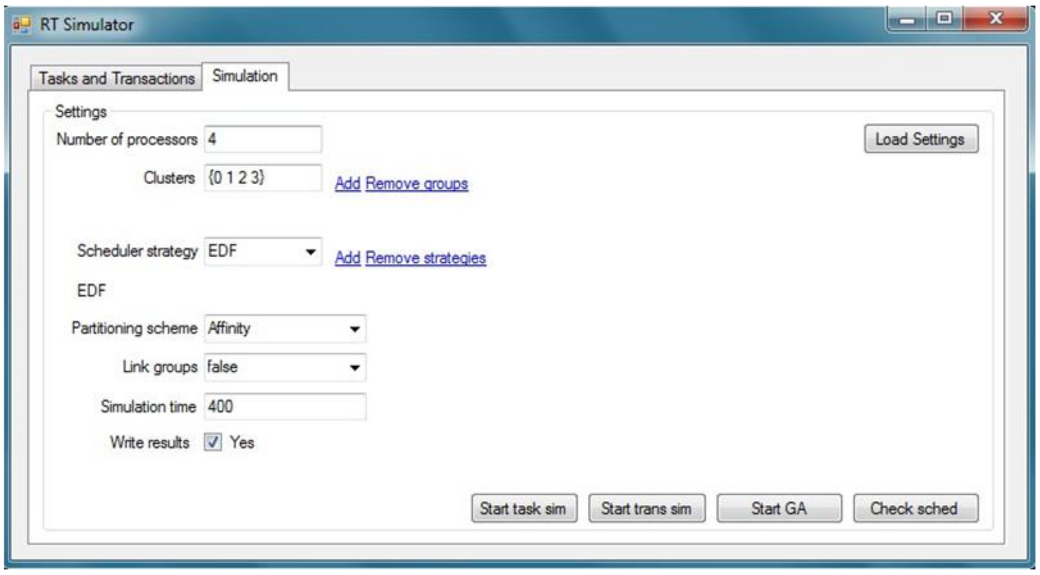

Settings
Number of processors: 4
Clusters: {0 1 2 3}
Scheduler strategy: EDF
EDF
Partitioning scheme: Affinity
Link groups: false
Simulation time: 400
Write results: [✓ Yes]

Figure 8. The RTMultiSim user interface.

The RTMultiSim GUI allows choosing the workload and platform model files and the manual input of some configuration parameters. Moreover, there are controls for starting the simulation or other types of evaluation. Fig. 8 shows the RTMultiSim GUI, the Simulation tab.

5. Synthetic workload generation tool

In order to generate different simulation scenarios that fit statistically to some globally defined parameters (e.g. processor utilization) we developed a tool for automatic workload generation. The goal is to establish for every simulation scenario some parameters that statistically cover the most relevant cases. Our tool can generate periodic independent task sets or periodic transaction (chain of tasks) sets.

5.1. Task Set Generation

Task sets used for evaluations are generated automatically in the majority of cases. The method used to generate tasks is essential, as some task set characteristics like the task set cardinality, the distribution of task periods or the distribution of individual task utilizations may influence the evaluation. For instance, for the same task set utilization factor, we obtained different schedulability results when we used uniform and exponential task utilization distribution. On 6 processors, with utilization factor of 4.8 and global EDF scheduling, task sets generated with a uniform distribution were better scheduled then those obtained with an exponential task utilization distribution.

We needed a tool that generates task sets that do not produce misleading simulation results. The parameters that can be configured for the task set are:

- Total utilization
- Number of tasks
- Task set maximum hyperperiod (LCM of repetition periods)

The main problem to be solved is to generate *n* individual utilization values of which the sum is equal to *U*. Even though there are two important results, which address this problem, the UUnifast-Discard algorithm [11] and the Randfixedsum algorithm [10], we decided to use a new approach. We made this decision because, in the UUniFast-Discard case, the algorithm fails to generate task sets for particular values of *n* and *U* [10], and because Randfixedsum is very complex and difficult to understand and implement.

To obtain a task set with *n* independent periodic tasks with implicit deadlines and utilization equal to *U*, we developed the following methodology:

1. Randomly choose *n* task periods (T_i) uniformly distributed in the interval [T_{min}, T_{max}], having the least common multiple (*LCM)* less than a given LCM_{max}.
2. Generate *n* random task utilization values (u_i) for which the sum is equal to a given *U*. Each u_i has to be equal or greater than $1/T_i$ and less than 1 (because the computed execution time C_i has to be greater than 0).
3. For each pair (T_i, u_i), compute the execution time $C_i=u_i*T_i$ of task *i*.
4. Verify if the task set satisfies the requested parameters. If not, the task set is discarded.

For step 2, we propose an algorithm that generates *n* task utilization values with the sum equal to *U*. The algorithm starts with assigning each u_i the mean value (*U/n*). To obtain a random distribution of the utilization values inside the task set and keep the total utilization equal to *U*, at each iteration step we randomly choose two u_i values which will be modified by adding and subtracting a random value from the interval [*0,U/n*]. After a large number of iterations, we obtain task utilization values, which are well spread in the interval [$1/T_i$, *1*]. The proposed algorithm's pseudocode is listed below.

Algorithm: generate n utilization values of sum equal to U.

Input: n, U, $T_1,T_2,\ldots,T_n$

Output: $u_1, u_2, \ldots,u_n$

```
Begin

For(i=1; i<=n){ $u_i$=U/n;}
Repeat $n^4$ times
{
  d = random(0,U/n);
  x = random(1,n);
  y = random(1,n);
  if(x!=y)&&(1/$T_x$<=$u_x$-d<=1)&&(1/$T_y$<=$u_y$-d<=1)
  {
    $u_x$ = $u_x$+d;
    $u_y$ = $u_y$-d;
  }
}

End
```

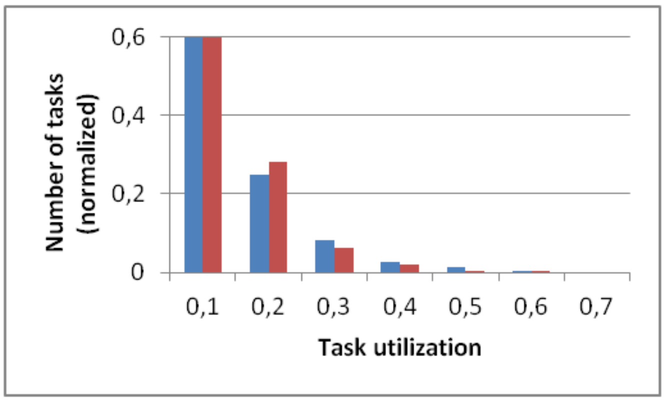


Figure 9. Comparison between Randfixedsum (blue) and our algorithm (red)

We compared our results with the results of Randfixedsum [10] and we concluded that the two approaches are equivalent because they generate similar distributions of individual utilization values. We generated task sets with 40 tasks per set and with total utilization of 4 with both algorithms (Randfixedsum and ours). Fig. 9 shows the distribution of individual task utilizations we obtained by using the two algorithms. In terms of execution time, our algorithm is slightly less efficient, but it produces results in an acceptable time for less than 100 tasks per task set.

Moreover, our algorithm does not have the problem of UUnifast-Discard [11], being able to generate without any problem task sets having the total utilization equal to half the task set cardinality.

5.2. Transaction Set Generation

We generate periodic transaction sets, using a similar methodology. The transactions are in fact chains of tasks, meaning that each task has at most one predecessor and at most one successor. Processing tasks may alternate with communication tasks.

The parameters that can be configured for the transaction set are:

- Total utilization
- Number of transactions
- Maximum number tasks per transaction
- Transaction set maximum hyperperiod (LCM of repetition periods)

The methodology for automatically generating a transaction set based on the previous preset parameters is the following:

1. Randomly choose *n* periods (T_i) uniformly distributed in the interval [T_{min}, T_{max}], having the least common multiple (*LCM)* less than a given LCM_{max}.
2. Generate *n* random transaction utilization values (u_i) for which the sum is equal to a given *U*.
3. Based on the generated periods and utilizations, create *n* transactions.
4. For each transaction, create a task set using the methodology presented in section IV.A. All tasks will have the same period, equal to the transaction's period. The precedence relations between them will be based on the order in which they are generated.

This version of the *Workload Models Generation Tool* does not generate platform related constraints for tasks. The user can add this type of constraints (e.g. task to CPU affinity) manually to the task or transaction set specification files.

6. The Visualization Module

After the simulation has ended, the Visualization Module can generate statistics and graphic representations of the simulation results. The user has to choose a simulation for which the module will show the results.

The user can view statistical data such as:

- Number of task migrations
- Number of job migrations
- Number of deadline misses
- Actual utilization per processor

The user can also view execution traces of tasks and jobs on processors, or the execution trace for all processors in parallel in the same window. Fig. 10 and 11 are graphic representations generated by the module.

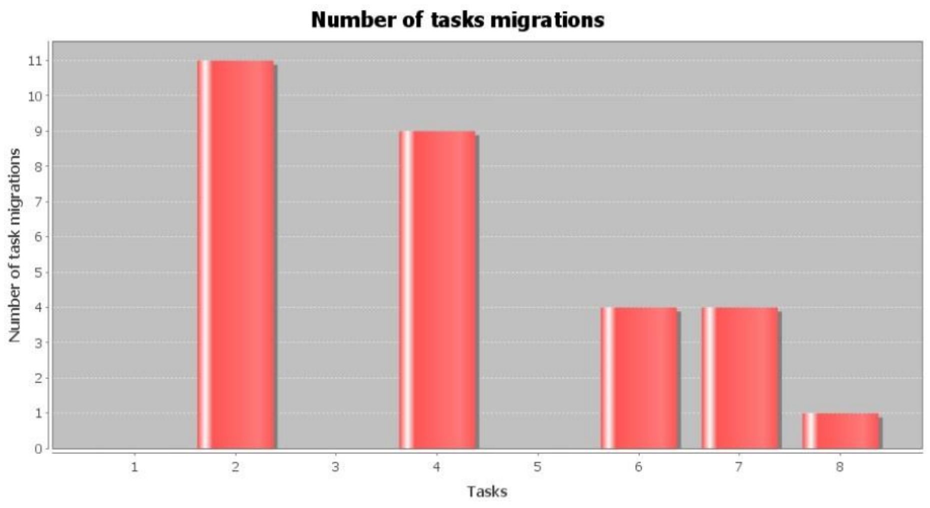


Figure 10. Task migrations in the Visualization module

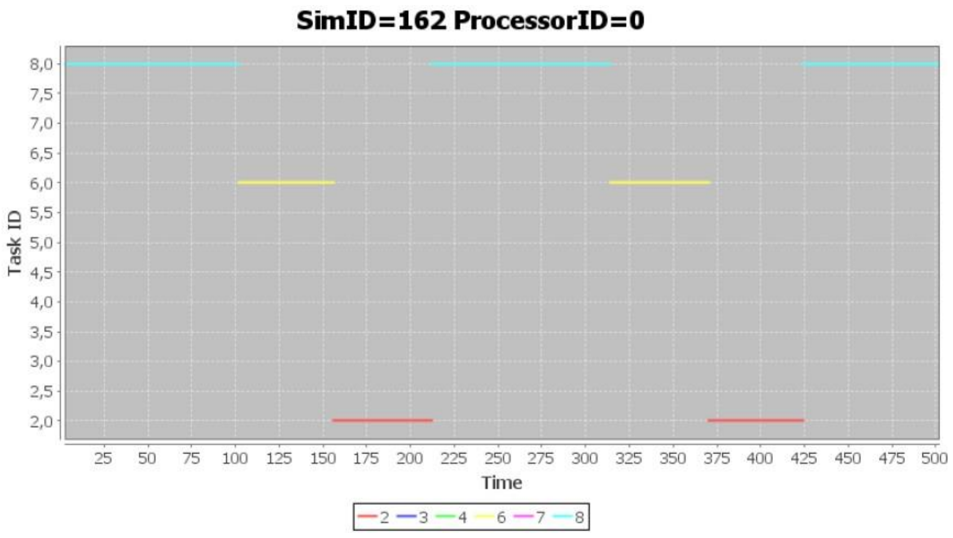


Figure 11. Task execution trace on one processor in the Visualization module

7. Comparison with other simulation tools

To highlight the main characteristics of RTMultiSim, we compare it with similar tools. We selected three of the most representative and recent real-time systems simulation tools: STORM, YARTISS, MAST. Table 1 shows the comparison. We highlighted the most general approach for each characteristic.

Table 1. Comparison between RTMultiSim and other real-time systems simulation tools

Characteristics	**RTMultiSim**	**STORM**	**YARTISS**	**MAST**
Simulation	Yes	Yes	Yes	No
Schedulability analysis	Yes	No	No	**Yes**
Set and optimize scheduling parameters (processor allocation,	Yes	No	No	Yes

deadline assignation)				
Workload model	**Parallel tasks, periodic tasks, non-periodic tasks, transactions, tasks with precedence dependencies**	Sequential periodic tasks, data messages between tasks	Sequential periodic tasks (energy aware), graph transactions	Chain transactions
Platform model	Homogeneous multiprocessor	Homogeneous multiprocessor	Homogeneous multiprocessor	**Heterogeneous multiprocessor**
Scheduling model	**Clustered, Loose Clustered, Global, Partitioned**	Global	Global	Partitioned
Scheduling algorithms	RM, EDF, LLF, FIFO	FP, EDF, FIFO	FP, EDF, LLF, FIFO	FP, EDF
Partitioning heuristics	**Next fit, Affinity**	No	No	Partitioning is done manually
Energy aware	No	No	**Yes**	No
Task set generation	**Yes**	No	Yes	No
Transaction set generation	**Yes**	No	No	No
Language	C#	Java	Java	Ada
OS	Windows	Windows, Linux	Windows, Linux	Windows, Linux
Open source	Yes	No	Yes	Yes
Visualization	Yes	Yes	**Yes**	Yes

We conclude that, compared to these other tools, RTMultiSim's has the following advantages:

- The most general workload model
- The most general scheduling model
- It implements partitioning heuristics
- Has task set and a transaction set generation tools
- The task and transaction set generation tools rely on a more efficient algorithm than UUnifast-Discard, while others rely on UUnifast-Discard.

As for the disadvantages:

- It is not available on Linux; however, with some minimum effort it can be compiled with Mono [15], to obtain its Linux version.
- The Visualization module is weak compared to others; however, by using some external, more complex tools we can achieve good graphical results analysis.
- It is not energy-aware; but only YARTISS has this characteristic.

8. Conclusions

This chapter presented *RTMultiSim*, a versatile simulation tool that covers most of the typical cases of multitasking and multiprocessor real-time systems. The workload model used by the simulator allows representation of various types of real-time tasks such as independent periodic tasks, dependent tasks, parallel tasks, fork-join tasks, and transactions. The scheduling strategy may be global, partitioned, or clustered. *RTMultiSim* provides multiple partitioning and scheduling algorithms and an easy method to integrate new algorithms. The tool includes a feature for automatic task and chain transaction set generation that can be used to create different simulation scenarios that fit some predefined parameters.

The RTMultiSim tools (source code and executables) and documentation are available at http://users.utcluj.ro/~ancapop/research.html.

References

[1] R. I. Davis and A. Burns. "A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems." techreport YCS-2009-443, University of York, Department of Computer Science, 2009.
[2] M. Bertogna, S. Baruah, "Tests for global EDF schedulability analysis", Journal of Systems Architecture, no. 57, pp. 487–497, 2011.
[3] L. Cucu, J. Goossens, "Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors", ETFA, Prague, September 2006.
[4] L. Cucu-Grosjean, J. Goossens, "Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms", Journal of Systems Architecture, 2011.
[5] R. Urunuela, A. Depaposlanche, and Y. Trinquet, "Storm a simulation tool for real-time multiprocessor scheduling evaluation," in Emerging Technologies & Factory Automation (ETFA), 2010 IEEE Conf., pp. 1-8, Sep 2010.
[6] M. G. Harbour, J. J. G. García, J. C. P. Gutiérrez, and J. M. D. Moyano, "Mast: Modeling and analysis suite for real time applications," in 13th Euromicro Conference on Real-Time Systems, 2001, p. 125.
[7] P. Courbin, L. George,"FORTAS : Framework for real-time analysis and simulation", 2nd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems, WATERS 2011.
[8] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet and M. Qamhieh, "YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms", 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems, WATERS 2012.
[9] D. Masson, "RTSS v1 and v2", https://svnigm.univ-mlv.fr/projects/rtsimulator/.
[10] P. Emberson, R. Stafford and R.I Davis, "Techniques for the synthesis of multiprocessor tasksets", 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-timeSystems, WATERS 2010.
[11] R. I. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009), December 2009, pp. 398–409.
[12] Simulink, http://www.mathworks.com/products/simulink/
[13] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System Level Performance Analysis–the SymTA/S Approach," IEEProceedings-Computers and Digital Techniques, vol. 152, no. 2, 2005.
[14] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," Real-Time Systems, vol. 30, no. 1-2, pp. 129–154, May 2005.
[15] Mono Project, http://www.mono-project.com/

CHAPTER 6. PERFORMANCE EVALUATION OF REAL-TIME MULTIPROCESSOR SYSTEMS 1

Chapter 6. Performance evaluation of real-time multiprocessor systems

1. Introduction

Rea-time scheduling strategies need schedulability tests for validation. This chapter presents a simulation-based schedulability test methodology and an evaluation method that are used to assess the performance of multiprocessor real-time systems. These evaluation methods aim to reduce the complexity of analytical test procedures.

Moreover, in the last section introduces a set of benchmark tests that assess the main features of a parallel real-time programming language. These tests measure the performance of individual operations executed on parallel threads. This chapter includes descriptions for the conducted experiments and results analysis.

2. A simulation-based schedulability test methodology for multiprocessor real-time systems

In the last years, important research efforts were made in the direction of finding feasible scheduling strategies for parallel architectures, but the results are far from being final or stable. One conclusion of these efforts is that the simple extension of uniprocessor theory and experience to parallel systems is not a successful approach. There are many examples when researchers based their demonstration on uniprocessor system results and proved to be wrong on parallel systems. For instance, it was proved that in case of a parallel architecture the critical time interval is not the period that follows after all the tasks are released at the same time, as in the case of a single CPU. In addition, the optimality of RM and EDF scheduling algorithms on parallel systems is not true, at least in some cases.

In the direction of feasibility analysis most of the results concentrate on finding necessary or sufficient conditions for feasibility. Nevertheless, these conditions are in most cases too complex [6] and hard to implement in practice or the gap between the necessary and sufficient condition leave too many cases undecided. The existing sufficient schedulability tests introduce very strong constraints, which excessively limit the utilization factor of many systems. Through simulation, we will show that these feasibility limits are too high, and we can find schedulable tasks sets that are not verifying the sufficient conditions.

Another research direction is based on the computation of demand and supply functions for multiprocessor real-time systems. In [7] and [8] the authors derive sufficient conditions for the feasibility of task systems using these functions. The complexity of the obtained relations, as well as the pessimistic results, limits the use of this method.

Recent research on priority-based multiprocessor scheduling showed that fixed priority algorithms (including RM and EDF) generate periodic schedules in the case of feasible periodic task systems [9][10]. This result will be used later in this section to demonstrate the feasibility of a task set using the simulation approach.

2.1. System model

The simulation-based schedulability test works for applications that can be modeled as *independent periodic tasks* with no restrictions other than real-time ones. It should be noted that the system model is a particularization of the one presented in Chapter 2.

A task set (τ) comprises n tasks and each task (τ_i) is described by its execution time (C_i), repetition period (T_i), and relative deadline (D_i). Task deadlines are *implicit*. The total utilization of a task set is computed as $U=\Sigma u_i$, where $u_i=C_i/T_i$ is the task's utilization factor. Another important task parameter that we consider here is the *phase*. If the phase is equal for all tasks, the task set is called *synchronous*. If the phases are different, the task set is *asynchronous*. In this case, task sets are synchronous. It is assumed that all jobs are sequential, meaning that a job has no multiple threads and it can be executed on at most one processor at the same time.

The platform is modeled as a *homogeneous multiprocessor* $P=\{P_1, P_2, \ldots , P_m\}$ composed of m processors with identical characteristics. The platform uses the *global scheduling* model that assumes the existence of a unique global scheduler and a global job queue. At each time instant, the global scheduler chooses from the job queue, m_a jobs with the highest priority, where m_a is the number of available processors. The chosen jobs will be executed on the processors. The priority of each task is computed by using a specific algorithm.

2.2. Theoretical results on global EDF schedulability analysis

We analyze the case of synchronous task sets executed on a homogenous multiprocessor system and scheduled with global EDF algorithm.

Most of the theoretical results on global EDF algorithms establish necessary or sufficient conditions for feasibility. In [6] the authors surveyed the most important seven sufficient schedulability tests with different computational complexity levels. One sufficient feasibility test [8], given below, states that a task set is schedulable with global EDF if:

$$U \leq m\ (1 - u_{max}) + u_{max} \tag{1}$$

This rather simple condition takes into account only the number of processors m and the usability factor u_{max} of the most demanding task from the set (the task with the maximum utilization factor). It can be seen that if u_{max} is close to 1 the usability of m processors is reduced to the usability of a system with a single processor. We used equation (1) as a reference for the simulation results.

A more powerful global EDF schedulability test (in terms of the detected number of schedulable task sets) [6] is based on an iterative method and has a pseudo-polynomial computational complexity. This test computes the minimum slack S_k (distance between the deadline and the completion time of a job) for each task. If the slack is negative for a task, the task set fails the schedulability test. The equation is:

$$S_k = D_k - C_k - \left\lfloor \frac{1}{m} \sum_{i \neq k} \min(I_k^i, D_k - C_k + 1) \right\rfloor \tag{2}$$

$I_k^i = \left\lfloor \frac{D_k}{T_i} \right\rfloor C_i + \min(C_i, (D_k \bmod T_i - S_i)_0)$, where indicates the interference caused by other tasks *i* on task *k*.

As we already commented in Chapter 2, according to the latest studies [6], there is a large region between the sufficient and necessary conditions, where task sets' schedulability is undecided. The "undecided" region increases with the number of processors. For instance, in the 8 processor scenario, this interval starts at a total utilization of about 56% of the maximum utilization. Even if some schedulability tests find more schedulable task sets than others, the conclusion of [6] is that all evaluated schedulability tests introduce strong constraints on the task sets they assess, leaving outside their limits a large number of task sets which may be schedulable.

We want to show, through simulation, that a large number of task sets from the undecided region are, in fact, schedulable.

Our method is based on the results in [9] and [10], where the authors show that deterministic fixed priority algorithms generate periodic schedules in the case of feasible periodic task systems. In [9] the authors prove that *a feasible schedule obtained using deterministic global EDF of a synchronous constrained deadline system on m identical processors is periodic with a period P that begins at instant 0*. The schedule repetition period *P* is equal to the task set's hyper-period. The authors extend their results in [10], where they give some guidelines for the development of an exact schedulability test for global EDF on multiprocessors. To decide if a task set is schedulable, the authors suggest to build a schedule and check if the periodic part of the schedule is reached or not. The periodic part is reached when the execution states at the two ends of the interval coincide (this is true because it is assumed that the algorithm is deterministic). If there are no missed deadlines in the first period, then the task set is schedulable. In the case of asynchronous task systems, the periodic part of a feasible schedule may begin after more than one hyper-period.

Following these results, we focused our research on combining simulation with theoretical analysis results to assess the schedulability of synchronous periodic task systems. The main objectives of our research are to:

- Develop a method for assessing the multiprocessor schedulability of periodic task systems through simulation.
- Experiment the behavior of multiprocessor systems with global EDF using randomly generated periodic task sets, in order to determine the features that influence the schedulability.

The outcome of our work will help real-time system designers to choose a practical strategy that does not involve complex and restrictive mathematical analysis and which is well suited to the characteristics of their application.

2.3. The schedulability test and evaluation method

The simulation-based exact schedulability test was developed following the guidelines presented in [10]. The test uses RTMultiSim, the real-time systems simulator presented in Chapter 5 to build the periodic schedule for independent, synchronous, periodic task sets scheduled with global EDF. Theory says that the schedule period is equal to the task set's hyper-period.

To check if the schedule's period has been reached, we simulate the system during two hyper-periods and we verify if the states at the end of each hyper-period coincide. The state of the schedule, at a certain time is given by the jobs that are executing at that time. A feasible schedule is one that, at the end of the task set's hyper-period has the following properties:

- All jobs meet their deadlines
- All jobs that started during the hyper-period, have finished their execution during the same hyper-period

We will use this tool to assess the schedulability of task sets with global EDF in different scenarios.

As there are not available any standard benchmark models or model sets inspired from real systems, to evaluate a scheduling technique or algorithm, we propose to use synthetic system models. This method is widely used in the performance evaluation of real-time scheduling techniques. However, the evaluation results depend on certain model parameters such as:

- The number of processors. An important factor is how the performance degrades as a function of the number of processors in the system.
- The task set cardinality. The number of tasks in a set influences the individual task utilization distribution. If there are few tasks in a set, those tasks will have larger utilizations compared to larger task sets that have the same total utilization.
- The individual task utilization distribution in a task set. Task utilization distribution is known to influence the schedulability rate in evaluation tests [11].
- The task set's hyper-period. In our case, the schedulability test can be performed only if the task set's hyper-period is less than 10^6. If the hyper-period is larger, the simulation will take too much time to execute.

We intend to investigate how these parameters influence the results of the schedulability test. After the results analysis, we will be able to draw some conclusions and give some advice on how to choose and vary the parameters of task sets that are used for real-time systems performance evaluation.

For our experiments, we will use the task set generation tool that we presented in Chapter 5. Our tool usually generates task sets with exponential distribution of individual task utilizations. To obtain task sets with other distribution, we temporarily altered the tools generation process.

2.4. Experiments and results analysis

We conducted experiments with the global EDF algorithm for different systems and determined through simulation which of the generated tasks sets are schedulable. We use synchronous periodic tasks with implicit deadlines, so the schedule period will start at time 0 and will be equal to the task set's hyper-period. We use the schedulability test we developed with RTMultiSim that was previously presented.

The total number of task sets generated and simulated was approximately 10,000. We investigate how many schedulable task sets are found by the simulation-based test and how the parameters of the task sets influence their schedulability.

First, we investigated how the individual task utilization distribution influences the schedulability rate. We generated tasks sets with different total system utilizations that will be executed on a platform with 8 processors. We generated three cases:

- In the first case, the distribution is uniform, between 0.1 and 0.5.
- In the second case, the distribution is uniform, between 0 and 0.7.
- In the third case, the distribution is exponential.

Fig. 1shows (left) the distribution of individual task utilizations in three cases, and (right) the schedulability test results.

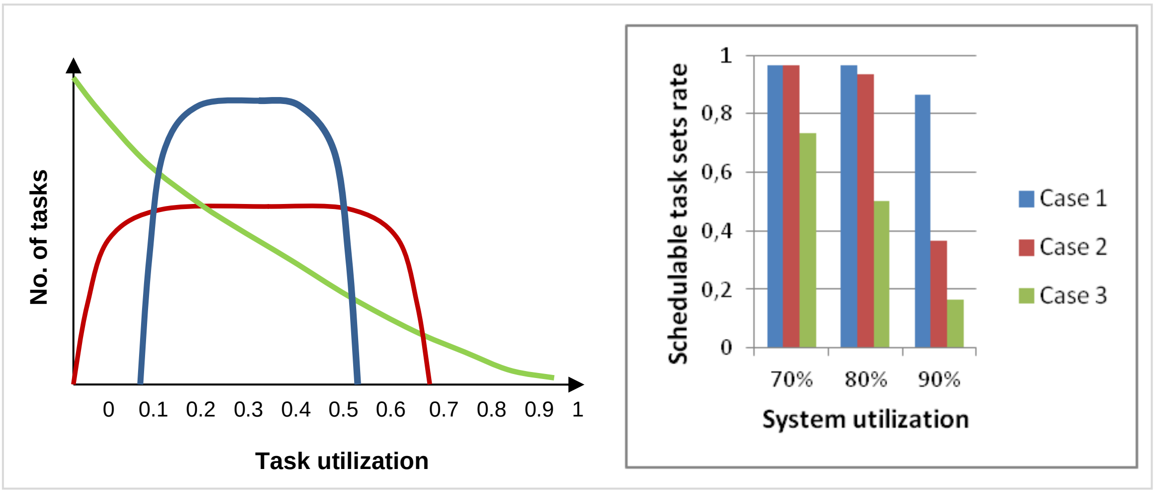


Figure 1. Experiments with different task utilization distribution. Distribution of individual task utilization in 3 cases and results obtained through simulation on 8 processors for the 3 cases.

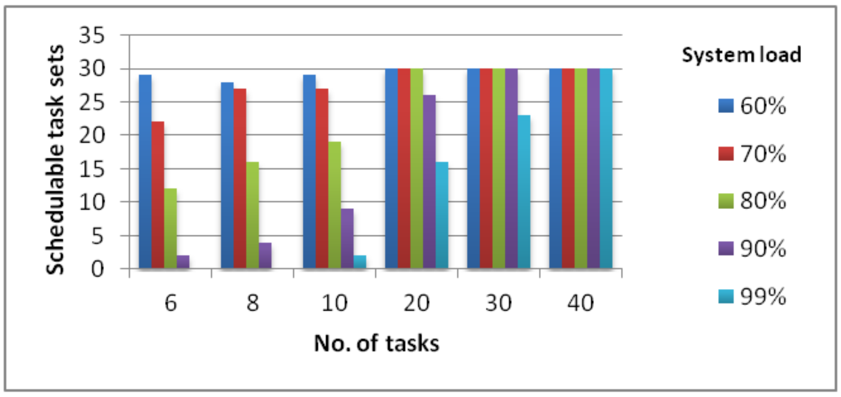
After this experiment, we are able to draw the following conclusions:

- For the same number of processors and total utilization, the schedulability depends on the distribution of individual task utilizations. The results in Fig. 1 show that the uniform distribution causes higher schedulability rates. The schedulability rate is even higher when the individual task utilization values are in a smaller interval (case 1).
- The schedulability of a task set is more critical when there are a few tasks with utilization close to 1. From Fig. 1 case 3 we observe that if we have task utilizations in the interval close to 1 the schedulability decreases. This is somehow in accordance with equation (1) that states that the largest possible schedulable total utilization is a function of the largest individual task utilization.

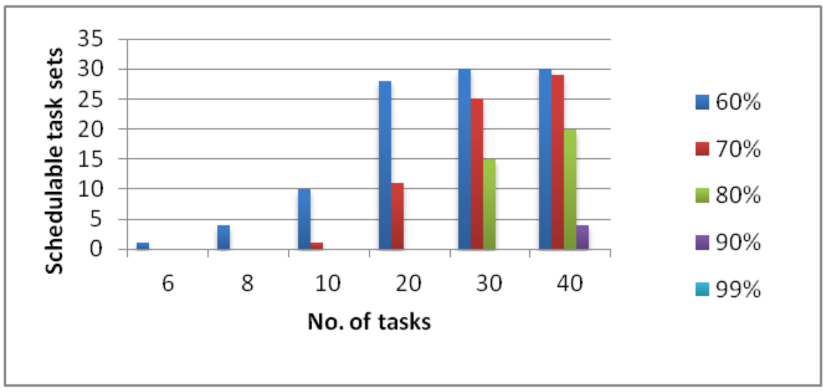
Moreover, we made simulations on 4 different platforms that contain 2, 4, 6 and 8 processors. For each platform we generated synchronous periodic implicit deadline task sets with increasing cardinalities between *m+2* and *m*10*, and increasing total utilizations between *0.6*m* and *0.99*m*, where *m* is the number of processors. We created 30 task sets for each individual scenario. For each task set we selected task periods from the interval [10, 250], having the *LCM* less than 100,000.

On each of the generated task sets, we also applied the global EDF schedulability test described by equation (1), in order to compare the results with those obtained with the simulation-based method.

Below, we include the graphical representation of the results we obtained for the platforms with 4 (Fig. 2) and 8 processors (Fig. 3). The results for 2 and 6 processors are similar and follow the same pattern, but were not included as graphical representations.

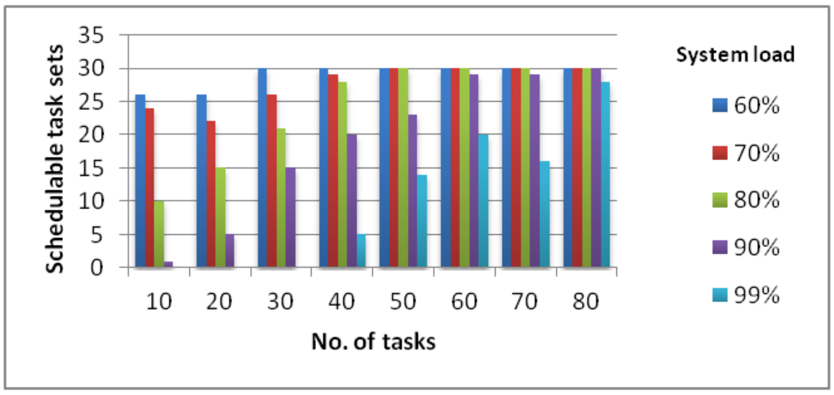


a. Results obtained through simulation

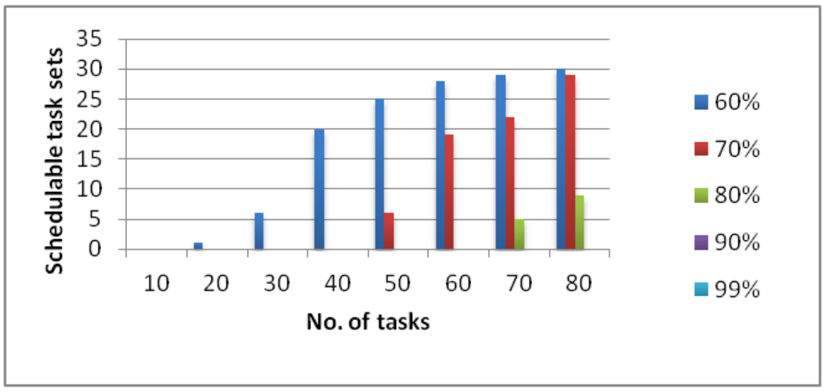


b. Results obtained with equation *(*1)

Figure 2. Experiment results for 4 processors



a. Results obtained through simulation



b. Results obtained with equation *(*1)

Figure 3. Experiment results for 8 processors

After the evaluation of the results, we made the following observations:

- The schedulability rate of all task sets decreases while the number of processor increases, for the same total (normalized) utilization. This is true for the simulation-based test, as well as for the test done with equation (1).
- As expected, the schedulability of a task set decreases with the increase of its total utilization. This can be seen in Fig. 2a and Fig. 3a, where for 4 and 8 processors the number of not schedulable task sets is represented related with the total utilization factor.
- For a given total utilization, the most critical scenarios are when the number of tasks in a set is slightly higher than the number of processors; as the number of tasks increases the number of not schedulable tasks decreases, becoming equal to 0 when the number of tasks is more than 10 times the number of processors.
- The results obtained through simulation are more optimistic than those obtained for the same scenarios using the analytical sufficient schedulability conditions. Fig. 2 and 3 compare the two situations; in Fig. 2a and 3a are results obtained through simulation and in Fig. 2b and 3b are results obtained with equation (1). This observation motivates the use of our simulation-based methodology.

The conclusion of these experiments is twofold. For a given total utilization, an increased (greater) number of tasks increases the feasibility chance. The presence of tasks with utilization close to 1 drastically reduces the chances for the whole set to be schedulable. The results obtained through simulation proved to be more optimistic than the theoretical ones.

The experiments and their results offer some guidelines in choosing the parameters of synthetic system models used for performance evaluation. Moreover, the simulation-based exact schedulability test can be a more pragmatic alternative to analytic tests that introduce too many restrictions and rule out many systems models that may be schedulable.

3. Simulation-based evaluation of global, partitioned and clustered scheduling approaches

We continue our investigation regarding multiprocessor scheduling of real-time systems with more simulation-based tests that focus on evaluating and comparing the three main multiprocessor scheduling approaches and the new approach proposed in Chapter 2, the loose clustered approach. We use the experience gained during the previous section.

The criteria used to measure the performance of a scheduling strategy are often chosen in accordance with the application domain. For hard real-time systems, where deadline misses are not allowed, the performance of a scheduling strategy is mostly measured by its ability to find a feasible schedule [12]. A schedule is feasible if, given a set of tasks all tasks complete their execution before their deadline.

In the case of soft real-time applications, performance measures can include the average and maximum response time or deadline miss rate. The evaluation of the performance can be done by formally proving the limits of the performance measures. Sometimes, finding these limits is a very complex problem. In these cases, simulation can be a good method for scheduling strategy performance evaluation.

In the uniprocessor world, the performance evaluation of scheduling strategies is mostly done through analytical evaluation. In 1973 Liu and Layland [2] proved the schedulability conditions of RM and EDF strategies (see equations (3) for RM and (4) for EDF) for sets of periodic independent tasks.

$$\sum U_i \leq n\left(2^{1/n} - 1\right) \tag{3}$$

$$\sum U_i \leq 1 \tag{4}$$

Later, Response Time Analysis (RTA) was proposed [13] as an iterative method for the exact evaluation of task worst case response time for fixed priority schedulers. This method relies on the analysis of the worst case arrival sequence (when all jobs are ready to execute at the same time, t=0).

The transition to multiprocessors proved to be challenging from the theoretical point of view. Not all uniprocessor theory can be generalized to multiprocessors. An important example is uniprocessor RTA, which cannot be applied to multiprocessors because the uniprocessor worst case arrival sequence is not valid under these new conditions [14]. To our knowledge, the worst case arrival sequence remains unknown for multiprocessors. The majority of the schedulability tests for global multiprocessor schedulers introduce a large computation overhead [14]. But for hard real-time systems, tests that guarantee timing requirements are necessary. On the other hand, non-critical real-time systems don't need such exact proofing mechanisms. An approximate performance evaluation of the scheduling strategy can be accepted.

The simulation of abstract representations or models is an approximate evaluation method. The approximation is better if the model used by the simulation is closer to the real-world and there are enough measurements done. To make a relevant evaluation of a scheduling strategy, the input for the simulator and the performance criteria have to be chosen according to the profile of the system that will use the strategy.

As already discussed in Chapter 2, there are advantages and disadvantages for each of the multiprocessor scheduling approaches.

Partitioned approach is preferred because after the step at which tasks have been assigned to processors, scheduling is reduced to uniprocessor scheduling on multiple processors. The main disadvantages are that this approach is not work-conserving and that task partitioning is done with non-optimal heuristics. The schedulability rate of these systems is closely related to the efficiency of the task partitioning step.

Global approach is work-conserving but suffers from *"Dhall's effect"* [15]. In the presence of heavy tasks, the schedulability of task sets is reduced almost to the maximum utilization of a uniprocessor ($1+\varepsilon$). Another problem of global scheduling is caused by the frequent migrations. The cost of migration is not quantified in many theoretical models, but in real systems, this cost is very high. Many developers of real-time systems prefer strategies that minimize the number of migrations, of context switches and the scheduling overhead introduced by a unique, large process queue.

Clustered scheduling emerged because of the observation that the cost of migration decreases in the case of shared cache memories. Migration is confined to processors that have shared caches. Moreover, the number of migrations and the scheduling overhead are reduced. As in the case of partitioned scheduling, the clustered approach is influenced by the partitioning heuristic, as well.

The loose clustered approach described in Chapter 2, was proposed as an optimization of the clustered approach and can be applied for the partitioned one, as well. Our approach extends the clustered with a load balancing mechanism that:

- Equalizes cluster utilization;
- Reduces capacity fragmentation due to not optimal task partitioning between clusters.

We estimate that the loose clustered approach will schedule more tasks sets than the clustered one, while maintaining a lower migration rate than the global approach.

To evaluate and compare the above scheduling approaches, we use the following metrics:

- The *number of schedulable task sets*, normalized with the total number of evaluated task sets. This metric shows how many tasks sets the scheduler is able to schedule without deadline misses.
- The *migration rate* defined as the total number of migrations divided by the number of schedulable task sets. This is because we do not record all steps for tasks sets that have deadline misses. The migration rate is important when the systems developer wants to quantify the migration-related overhead introduced by a scheduler.
- The *average slack* of task sets. The slack or laxity is defined as the time remaining from the job's completion time until the actual deadline. The slack is a metric for the maximum additional load a system would be able to accept, and still be schedulable. The scheduler that obtains larger slacks is more efficient in using the processors' available time.

3.1. Experimental setup

A similar system model with the one used in the previous section is used:

- Independent periodic task sets with implicit deadlines
- Homogenous multiprocessor

The scheduling model proposed in Chapter 2 and implemented in RTMultisim is used. The scheduling model can be configured to be:

- Global
- Partitioned
- Clustered
- Loose clustered

In the cases when a partitioning scheme is needed, Next Fit heuristic is used. The scheduling algorithm is EDF.

RTMultiSim and its task set generator are used to conduct the simulations and to produce synthetic workload models. The previously presented simulation-based schedulability test is used to assess the schedulability of a task set. The migration rate and average slack are computed from the simulation results.

The evaluation is done as follows:

- We do all our tests on a platform with 4 processors.
- We generate 450 task sets with total loads of 50% to 99%, with 6 to 20 tasks.
- We simulate the scheduling in case of all four scheduling models, using the same task sets.

3.2. Experimental results

The experimental results are presented in Fig. 4, 5 and 6. The partitioned approach can find the most feasible schedules. The greatest advantages of this approach is that jobs do not migrate and that it has one distinct scheduler for each processor. The greatest disadvantage is that if the partitioning heuristic is weak, there will be less feasible schedules. Moreover, the average slack is the smallest.

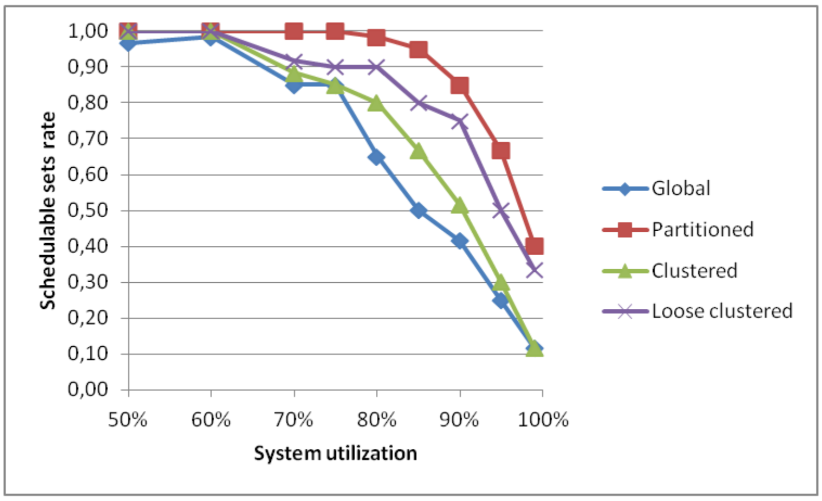


Figure 4. Schedulable task sets as a function of system utilization

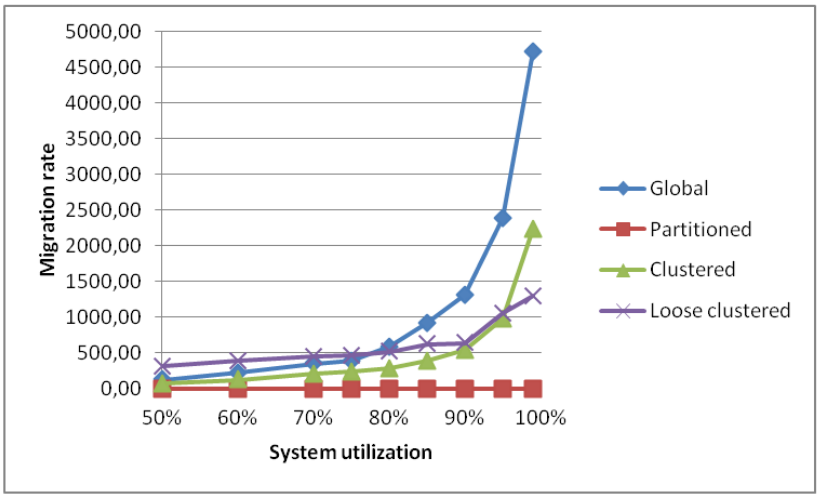


Figure 5. Migration rate as a function of system utilization

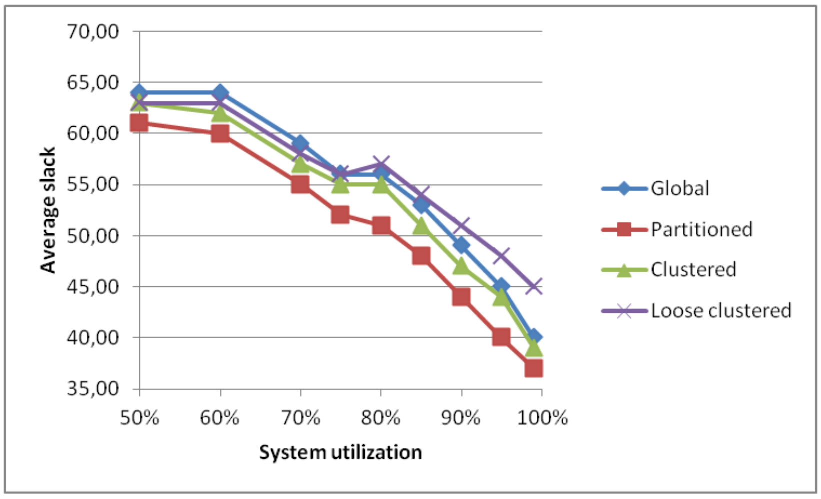


Figure 6. Average slack as a function of system utilization

The global approach finds the fewest feasible schedules. This is mostly because o the Dhall's effect. The migration rate is very high for task sets with system utilization larger than 80%. However, the average slack is one of the largest; only the loose clustered approach has a larger slack for task sets with system utilization larger than 80%.

The clustered approach finds slightly more feasible schedules than the global approach, but only for task sets with system utilization larger than 70%. However, the migration rate is half the one measured for the global approach. The average slack is less than the one measured for the global.

Compared to the clustered approach, the loose clustered one finds more schedulable task sets, but with the cost of larger migration rates. But, for system utilizations larger than 80% it os only a little more higher. The average slack is one of the highest.

Due to these observations, we can conclude that the loose clustered approach has good results mainly for task sets with large system utilization, in these cases improving the schedulability rate with the cost of a very small increase of the migration rate.

4. Tests for assessing the features of parallel real-time Java implementations

Real-time applications mostly use special processors in order to be predictable and to show a deterministic behavior. General purpose processors, on the other hand, are not widely used in real-time systems because of the non-determinism introduced by various features that are used to improve their performance such as the processor cache hierarchy, speculative execution of instructions or hardware parallelism (Hyper-Threading). However, the technology advances and mainly the improvement of multi-core processors made general purpose processors more interesting to the real-time community. The use of multicore processors and of parallel programming generated new areas of interest in real-time systems. As noted in Chapter 1, the interest in multicore platforms is twofold. There is a special interest in the timing analysis of multicores and that of operations that involve a cache hierarchy. On the other hand, there is a special interest in parallel or multicore-aware programming languages for real-time applications.

The most popular programming languages for real-time applications were until recently C or Ada. In the past years, in an effort for introducing Java and its advantages (e.g. object-oriented programming, platform independence) to the real-time community, the Real-Time Specification for Java (RTSJ) [16] standardized a set of constraints on the Java language and runtime environment. These constraints are related to several features like its garbage-collector, dynamical memory allocation, threading and synchronization methods, that make Java unsuitable for real-time programs. Currently, there are several available RTSJ implementations [17][18][19][20]. Due to the development of multicore processors and the real-time community's interest in it, the group that defined the RTSJ showed their intention to create a specification for a parallel/ multi-threaded version of RTSJ[21].

In this context, our research targets the development of benchmark tests that will assess the features of parallel real-time Java implementations.

Currently, there are just a few real-time Java benchmarks, but they are either not multiprocessor-aware [22] or not complete real-time benchmarks [23] (the latter doesn't support *NoHeapRealtimeThreads* and *AsyncEventHandlers* [16]). The micro-benchmark presented in [24] deals with memory allocation, communication buffering and assessing timer accuracy for real-time Java. Even if it does not discuss the consequences of using multi-core processors, uses test scenarios that rely on multi-threading and a multi-core system for testing. However, the accent falls on real-time issues and not on parallelism (the two threads are started concurrently, but not necessarily in parallel) or low-level multi-core issues. In the embedded systems world, there are a few C benchmarks that are parallel/multi-core aware, such as MiBench [25] and EEMBC [26]. MiBench is open-source and has a collection of 35 C applications that are used to characterize embedded workloads and determine the performance of embedded processors. EEMBC provides also a Java application benchmark suite for the evaluation of multi-core processors but is only commercially available.

4.1. Tests overview

We developed a set of benchmark tests (RTSJMcBench) that assess important features of parallel Java implementations such as:

- Memory operations: allocation, copy, shared cache read.
- Asynchronous event handling.
- Locking.

A test starts a collection of parallel threads that execute a given task. Depending on the test purposes, a particular primitive operation is timed by using the processor timestamp counter (e.g., in the case of Intel processors, the *rdtsc* instruction [27]). Such operations, for instance, can be the acquiring of a lock or an array copy. The timing operation is done in a loop, in order to take several measurements that allow for a statistical interpretation of the results. Each thread reports the time values in a separate file. For timing and reporting, we use the jTools [28] package. A separate tool merges the individual result files into a CSV file where each column represents the values of a single thread. Practically, this is the final output of the benchmark. We developed an additional tool for processing of this result files and interpreting the results.

Running a RTSJMcBench test can be also complicated by the number of arguments the program needs. To configure the run-time arguments, the application that starts the tests reads these parameters from an XML file. By default, the name of the arguments file is *args.xml* and it must reside in the same directory with the test application. Below is an example of such an arguments file used by a memory allocation test for Linear Time Scoped Memory [16]:

```
<testapp name="rtsj.mcperf.mem.alloc.memalloc">
    <arg name="no_of_threads" value="10"> Number of realtime threads </arg>
    <arg name="type_of_threads" value="rt"> Type of threads: RealtimeThread (rt), NoHeapRealtimeThread
    (nh) </arg>
    <arg name="scheduling_policy" value="rr"> POSIX realtime scheduling policy: SCHED_RR (rr), SCHED_FIFO
    (fifo)</arg>
    <arg name="count" value="10000"> Number of iterations </arg>
    <arg name="priority" value="30"> Priority of the real-time threads </arg>
    <arg name="datadir" value="./datadir"> Directory storing the results of the run </arg>
    <arg name="instr_load" value="1000"> Number of instructions to simulate activity </arg>
    <arg name="memtype" value="LT"> Memory type: scoped (LT or VT), immortal </arg>
    <arg name="memsize" value="32M"> Memory size: bytes, Kilobytes (K), Megabytes (M) </arg>
    <arg name="allocsize" value="8"> Size of the allocated memory chunks: bytes, Kilobytes (K), Megabytes
    (M) </arg>
</testapp>
```

The arguments describing the collection of parallel real-time threads of a benchmark test are: the number of threads, their type (RealtimeThread [16] in the above case), their scheduling policy (SCHED_RR [29] in the above example) and real-time priority, the number of iterations performed in order to time a given operation, the directory where the results of the run will be stored and a so called "instruction load" that is used to run some fake computation that allows simulating thread activities besides those represented by the operations that are timed. The rest of the arguments in the above example are those particular to the memory allocation test, namely the type of memory to be allocated (Linear Time Scoped Memory), the total amount of Scoped Memory allocated for the test and the size of each of the allocation operations performed by the test.

The following sections will describe the individual tests and their experimental results obtained for Jamaica VM, a multicore-aware implementation of RTSJ. The Linux system we used for testing is a Slackware distribution running on a dual quad-core Intel Xeon E5405 running at 2 GHz with 3 GB of RAM. This quad-core processor is a uniform memory access symmetric multiprocessor that packages two dual-core dies into the same chip. There is no Hyper-Threading available. The cache sharing of the processors, which can be detected by means of the CPUID instruction, is: cores 0 and 2 share their own L2 cache and so do cores 4 and 6, 1 and 3 as well as 5 and 7.

4.2. Memory operations

Memory allocation. This test aims to evaluate the performance of allocating either regular Java memory (heap memory) or the various types of RTSJ memory: Scoped Memory (either Linear Time or Variable Time) and Immortal Memory [16]. The test measures the time taken by the new instruction when allocating an array of given size for various types of memory. The user can set the size of the memory area to be used and the size of the array allocated by new. The allocation is performed a number of times specified by the iteration count value taken from the arguments file. An automated procedure detects whether the choice of the various parameters doesn't lead to memory overflow (i.e., the number of iterations times the allocation size is smaller than the available memory).

The main thread creates the test threads and calls the start method of the benchmark framework. Because the constructor of the thread test class takes a reference to a Memory Area object (either Scoped or Immortal memory), the start method will create the corresponding real-time threads using the specified type of memory. If no RTSJ MemoryArea object is specified, normal Java heap memory is used instead. Each test thread sets its processor affinity, attempts to pass a barrier waiting for all the threads of the collection to be ready to run and then enters a loop. The processor affinity is set in a Round Robin manner across the set of available processors based on the logical thread ID. Within the loop, the thread iterates a fixed number of times (the iteration count is taken from the args file) over a sequence of operations that allocates an array of bytes and simulates some work. The allocation instruction is timed using the start/stop methods of HighResTimer [28] object for that thread.

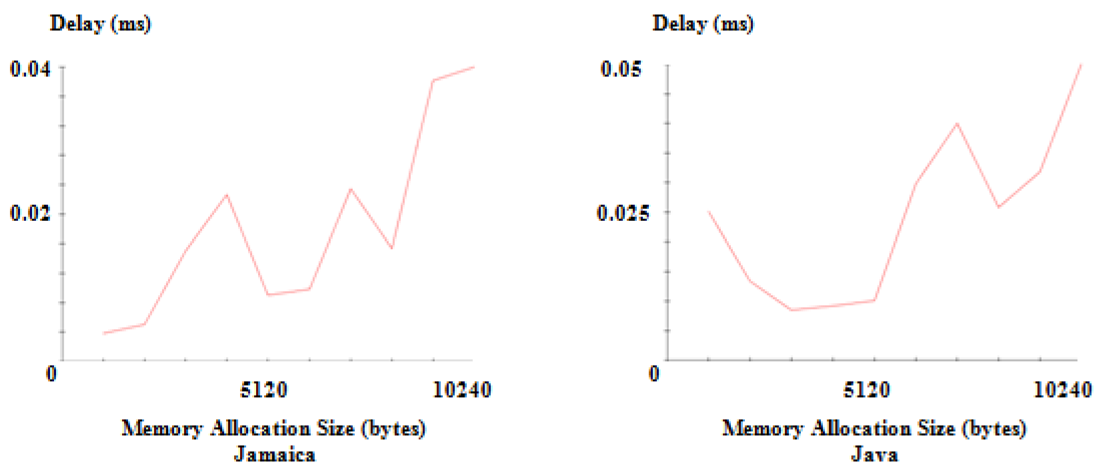


Figure 7. Results of the memory allocation test

Fig. 7 describes the results obtained by running an allocation test with 16 real-time threads for heap memory, both with JamaicaVM [17] (parallel version) and plain Java running on Linux. The threads used a SCHED_FIFO [29] policy (priority based, equivalent to PriorityScheduler [16]) and real-time priorities of 30. The size of the allocation unit varied from 1 KB to 10 KB with a 1KB increment. On the y-axis is reported the average latency of allocations.

Memory copy. This test is almost the same with the previous one, with one difference. This time, we measure the time is takes to copy an array by using System.arraycopy. Both the source and destination arrays are local to the test thread and are allocated before entering the main loop of the run method of the test thread.

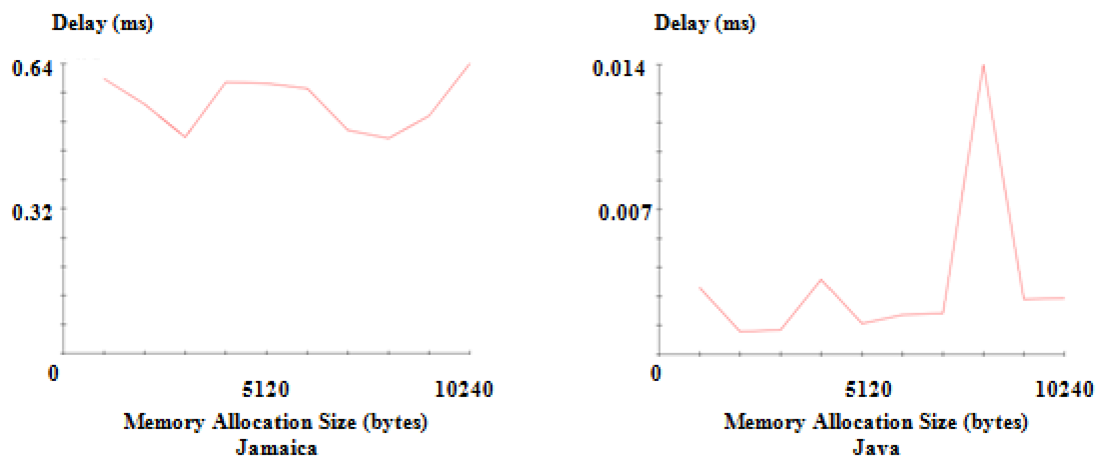


Figure 8. Average memory copy latencies

Fig. 8 describes the average latencies of copying heap memory arrays of various sizes, with Jamaica VM (parallel version) and plain Java with 16 real-time threads using SCHED_FIFO and priorities of 30. All the processors of the system described at the beginning have been used. The size of the arrays to be copied ranged from 1 to 10 KB with an increment of 1 KB. It can be noticed that the copy performance of Jamaica is worse than that of plain Java, but this is mainly due to a particular implementation of the arrays in the Jamaica VM.

Shared cache read performance. This test aims to assess the performance of using the CPU caches when copying byte arrays. The source of the array copy is shared by all the threads, while the destination of the copy is a local array for each of the test threads. The source array is allocated to fit in the L2 cache. The source array size is taken from the arguments file and is expressed as a fraction of the L2 cache size. Threads' CPU affinity is set so that they all share a L2 cache. The assignment is perfectly balanced according to a Round Robin scheme based on the logical thread ID.

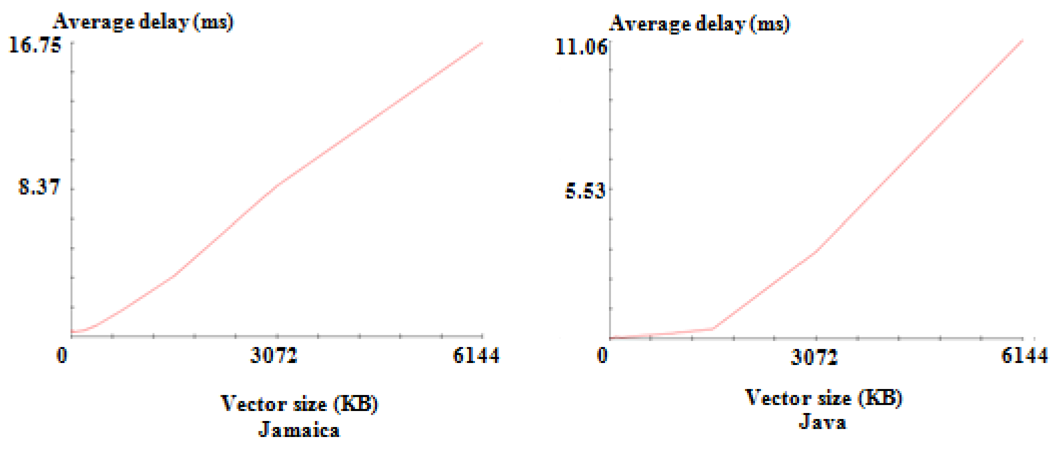


Figure 9. Shared cache copy performance

Fig. 9 shows the test results for 16 real-time threads running on two CPUs that share an L2 cache. The L2 cache size for Intel 5405 is 6 MB. Therefore, the size of the array to be copied varied from 6 MB to 6 KB, using a fraction factor growing exponentially according to the powers of 2 from 1 to 1024. The graphs report the average latency of copying a shared array allocated in heap memory. Naturally, as its size increases to the L2 cache size, the performance degrades. It can also be noticed that the Jamaica VM performance is somehow worse than that of Java.

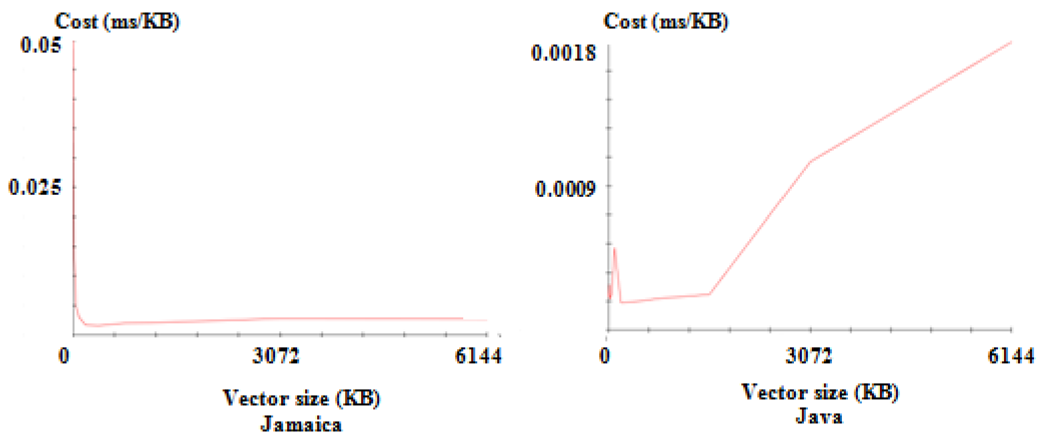


Figure 10. Average cost per byte for array copy

Fig. 10 describes an average cost per byte as the size of the array increases. It can be noticed that the cost per byte of Jamaica VM is insensitive to the increase of the array size.

4.3. Asynchronous event handling

This test evaluates the dispatch delay of an RTSJ event handling (i.e., the time between firing an event and the call of the associated handler). Each test thread defines a local event and associates a handler with it (a BoundAsyncEventHandler [16] object; therefore, according to RTSJ, each handler executes within a separate thread). Each handler has its own CPU affinity, which is the same with that of the thread associating the handler with the event. The CPU affinity is set Round Robin across the available processors, both for the test threads and their corresponding handlers. Within the main loop, each thread starts the local timer, fires the local event and waits for an event handler notification. The event handler records the current time by stopping the timer. Before finishing, the event handler notifies its completion to the firing thread. The thread unblocks, simulates some activity and starts a new loop iteration.

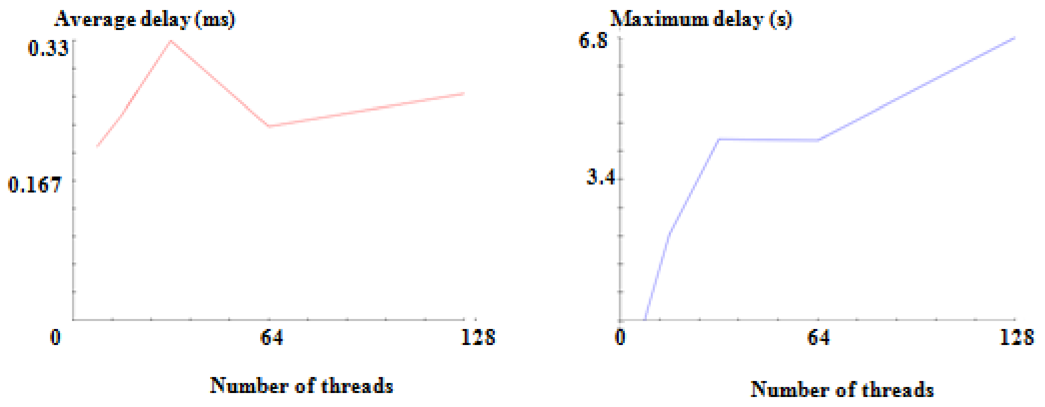


Figure 11. One-on-one average event handling dispatch latency and maximum event handling dispatch latencies

Fig. 11 shows the average dispatch latency and its maximum value for real-time threads on Jamaica VM and Linux. The tests used all the processors and have been run with 8, 16, 32, 64 and 128 real-time threads respectively. Note that the average dispatch time stays somehow constant with the number of threads. However, the maximum dispatch delay grows naturally with the number of threads. It is also worth noting that the maximum delays are on the order of seconds, while the average delays are worth hundreds of microseconds. One possible explanation is that the dispatch of some of the bound event handlers (Linux threads) might have been delayed by the activity of system threads.

4.4. Locking

This test evaluates the performance of storing spin-locks in shared CPU caches. When two threads that synchronize using spin-locks run on cores that don't share any processor cache, grabbing and releasing the spin-lock are operations that fire the cache coherence protocol. The test measures the impact of the cache coherence protocol on the locking performance. Note that even if two cores are on the same chip, they may not share an L2 cache. The test chooses a random CPU from the processor set and then finds CPUs that share an L2 cache with it. Half of the threads perform cache-aware locking, while the other half does not. Instead, they run on a processor subset not sharing an L2 cache. The threads set their CPU affinities Round Robin across their corresponding CPU subsets. Each of the halves uses its own shared spin-lock. We have run simultaneously the two thread sets to make sure that the measurements are taken in the same conditions. After setting the appropriate CPU affinity, each thread enters the main loop where it attempts to grab the corresponding shared spin-lock. The local thread timer measures the acquire operation. After performing the critical section, the lock is released. Fig. 13 shows the average spin-lock acquire latency when using 4 real-time threads. Two threads perform cache-aware locking (Threads 1 & 2), while two other (Threads 3 & 4) perform non-cached locking. The graphs show three values per thread, one for each processor set that has been used. The first bar (the red one) shows the performance of running cache-aware locking on CPUs {0,2} and non-cached locking on CPUs {1,5}. The second bar, the blue one, shows the performance on another experiment using CPUs {0,2} and {3,7}. The third bar (the green one) shows the performance of using the set of CPUs {1,3} and {0,4}.

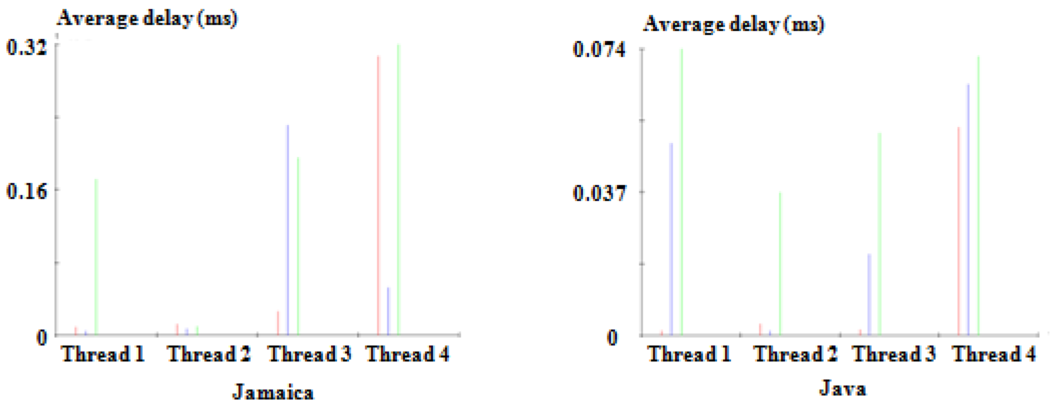


Figure 13. Average cache-aware latencies

Note that for non-cached locking at least one of the threads 3 and 4 is constantly hit be synchronizing penalties. In only one case, the green bar on the Java graph, both cache-aware and non cache-aware locking perform comparably poor. For the rest of the cases, cache-aware locking outperforms non-cached locking (at least one of the threads 1 and 2 takes advantage of the shared cache). In terms of absolute values, Jamaica VM performs one order of magnitude poorer than Java.

5. Conclusions

This chapter presented solutions for two directions in the area of multiprocessor real-time systems performance evaluation, namely the evaluation of scheduling strategies and execution time measurements of individual operations in real-time programs.

The validation of real-time multiprocessor scheduling strategies through analytical schedulability tests is often very difficult. In this chapter, an exact schedulability test that is based on simulation is presented. This test is based on the theoretical result which states that if a feasible schedule with the period equal to the task set’s hyper-period is found, then the task set is feasible. The performance evaluation methodology developed with this test is then used to evaluate several scheduling strategies. This schedulability test proved to find more schedulable task sets than other analytical tests.

To conclude the discussion on real-time systems performance evaluation, the last section of this chapter presents a set of micro-benchmark tests made for assessing the timing of individual memory-related, asynchronous event handling and locking operations implemented in parallel real-time Java.

References

[1] R. I. Davis and A. Burns. "A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. "techreport YCS-2009-443, University of York, Department of Computer Science, 2009

[2] C.L. Liu, J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", Journal of the ACM, Vol. 20, No. 1, January 1973, pp.46-61

[3] S.K. Baruah, N. Cohen, G. Plaxton, D. Varvel, "Proportionate progress: A notion of fairness in resource allocation", Algorithmica 15, no. 6, pp. 600–625, 1996

[4] J. Anderson, A. Srinivasan, "Early-release fair scheduling". In Proceedings Euromicro Conference on Real-Time Systems, June 2000

[5] J. Anderson, A. Srinivasan, "Mixed pfair/erfair scheduling of asynchronous periodic tasks". In Proceedings of the 13th Euromicro Conference on Real-Time Systems, June 2001

[6] M. Bertogna, S. Baruah, "Tests for global EDF schedulability analysis", Journal of Systems Architecture, no. 57, pp. 487–497, 2011

[7] Bini, E.; Bertogna, M.; Baruah, S.; , "Virtual Multiprocessor Platforms: Specification and Use," Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE , vol., no., pp.437-446, 1-4 Dec. 2009

[8] Bini, E.; Buttazzo, G.; Bertogna, M.; , "The Multi Supply Function Abstraction for Multiprocessors," *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on* , vol., no., pp.294-302, 24-26 Aug. 2009

[9] L. Cucu, J. Goossens, "Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors", ETFA, Prague, September 2006

[10] L. Cucu-Grosjean, J. Goossens, "Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms", Journal of Systems Architecture, 2011

[11] P. Emberson, R. Stafford and R.I Davis, "Techniques for the synthesis of multiprocessor tasksets", 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-timeSystems, WATERS 2010

[12] J. W.S. Liu, Real-Time Systems, Prentice Hall, 2000

[13] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy Wellings, "Applying new scheduling theory to static priority preemptive scheduling", Software Engineering Journal 8 (1993), no. 5, 285–292.

[14] M. Bertogna, Real-Time Sceduling Analysis for Multiprocessor Platforms, PhD Thesis, 2007

[15] S. K. Dhall, C. L. Liu, *"On a Real-Time Scheduling Problem"*, Operations Research, vol. 26, number 1, pp. 127-140, 1978.

[16] Greg Bollella and James Gosling. *The Real-Time Specification for Java*, 2000. Addison-Wesley Longman Publishing Co., Inc.

[17] Aicas Gmbh. *The Jamaica Virtual Machine*, http://www.aicas.com/jamaica/doc/rtsj_api

[18] A. Corsaro and D. Schmidt. The Design and Performance of the jRate Real-Time Java Implementation. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, October 2002.

[19] Sun Microsystems. *The Sun Java Real-Time System*. http://java.sun.com/javase/technologies/realtime/index.jsp

[20] TimeSys Corporation. RTSJ Reference Implementation and Technology Compatibility Kit. http://www.timesys.com/java/

[21] Jeopard FP7 Project, "JEOPARD Whitepaper", 2008

[22] A. Corsaro and D. Schmidt. Evaluating Real-Time Java Features and Performance for Real-Time Embedded Systems. In Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), September 2002.

[23] Brian P. Doherty. A Real-time Benchmark for Java. In Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '07), September 2007.

[24] J. F. Schommer, D. Franke, C. Weise, and S. Kowalewski. "Evaluation of the Real-Time Java Runtime Environment for Deployment in Time-Critical Systems". In Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, September 2009.

[25] MiBench Version 1.0, http://www.eecs.umich.edu/mibench/

[26] EEMBC, The Embedded Microprocessor Benchmark Consortium. http://www.eembc.org/home.php

[27] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B: Instruction Set Reference, N-Z, September 2008

[28] Angelo Corsaro. *jTools*, http://www.cs.wustl.edu/~corsaro/jRate/Download.html

[29] Linux sched_setscheduler manual page