Proceedings
Work-in-Progress Session
of the 24th Euromicro Conference on Real-Time Systems (ECRTS 12)

July 10-13. 2012
Pisa, Italy

http://ecrts12.ecrts.org/wip

Organised by the Euromicro Technical Committee on Real-Time Systems

Edited by Julio L. Medina

Preface

Welcome to Pisa and to the Work-in-Progress (WiP) Session of the 24th Euromicro Conference on Real-Time Systems (ECRTS 12). This session is dedicated to present new and on-going promising research into the broad field of real-time systems and applications. The papers included in these proceedings cover a wide range of topics, spanning across many areas of real-time systems, including real-time components and modeling, energy-aware scheduling, multi-core, probabilistic and soft real-time scheduling analysis, controller area networks, and the real-time and energy management capabilities in the new versions of the Linux kernel. These WiP papers will be also published on-line within ACM SIGBED Review (http://sigbed.seas.upenn.edu/)

The primary purpose of this WiP session is to provide researchers with an opportunity to discuss their evolving ideas and gather feedback from the real-time community at large. The creative ideas and approaches in the papers here selected are indeed a prospective view of what the future in our community may bring soon as fresh blood in our research arena.

I would like to thank the members of the WiP session technical program committee for their reviewing efforts, and of course the authors for their good submissions and their confidence in ECRTS as a means to improve and make advance their research. Special thanks also go to the ECRTS 12 organizers, Giorgio Buttazzo, Rob Davis, and Gerhard Fohler, for their confidence, support, and guidance.

I hope you all enjoy this session, participate in the discussions, and have the opportunity to provide your valuable feedback to the authors. Now, if in addition you happen to do so in front of their posters … that would be really great!!

Julio Medina
Work-in-Progress Session Chair
24th Euromicro Conference on Real-Time Systems (ECRTS 12)
July 2012

ECRTS'12 Work-in-Progress Technical Program Committee

Program Committee Members

- Moris Behnam, Mälardalen University, Västerås, Sweden
- Sébastien Gerard, CEA-List, France
- Ricardo Marau, Universidade do Porto, Portugal
- Marco Di Natale, Scuola Superiore Sant'Anna, Pisa, Italy
- Oleg Sokolsky, University of Pennsylvania, USA

Work-in-Progress Session Chair

Julio Medina, Universidad de Cantabria, Spain

Table of Contents

Translating End-to-End Timing Requirements to Timing Analysis Model in Component-Based Distributed Real-Time Systems

Saad Mubeen*, Jukka Mäki-Turja*† and Mikael Sjödin*
* *Mälardalen Real-Time Research Centre (MRTC), Mälardalen University, Västerås, Sweden*
† *Arcticus Systems, Järfälla, Sweden*
{*saad.mubeen, jukka.maki-turja, mikael.sjodin*}*@mdh.se*

***Abstract*—Often, component-based real-time systems are modeled with trigger and data chains. The end-to-end timing requirements on trigger chains are different from those on data chains. For a trigger chain, the interest lies in the calculation of holistic response time and its comparison with end-to-end deadline. Whereas, the schedulability of a data chain requires a comparison between its end-to-end latencies and corresponding deadlines. We discuss the problem of translating end-to-end timing requirements unambiguously from component-based real-time systems into timing analysis models which are required as input by the analysis tools. We also provide preliminary guidelines for such translations in the existing industrial tool suite.**

***Keywords*-Response-time analysis; end-to-end timing analysis; timing model; component-based development.**

I. Introduction

Often, component-based real-time systems are modeled with chains of components that are translated to chains of tasks at run-time. A task chain is a sequence of more than one tasks in which every task (other than the first) receives a trigger, data or both from its predecessor. One way to classify these chains is as trigger chains and data chains. In trigger chains, there is only one triggering source (e.g, event, clock or interrupt) that activates the first task in the chain which, in turn, triggers the next task and so on. On the other hand, data chains have independent source of triggering for every task. Each task (except first) in these chains receives data from its predecessor. In component-based real-time systems, the timing requirements such as end-to-end deadlines on trigger and data chains are specified in the component model.

The safety-critical nature of many real-time systems requires evidence that the actions by the system will be provided in a timely manner, i.e., each action will be taken at a time that is appropriate to the environment of the system. Therefore, it is important to make accurate predictions of the timing behavior of such systems. For this purpose, a priori analysis techniques such as schedulability analysis have been developed by the research community. The analysis tools operate on the timing analysis model which should be extracted from the modeled application. The end-to-end timing requirements should be unambiguously translated to the analysis model from the component model of the real-time application.

In this paper, we discuss the problem of translating the end-to-end timing requirements into analysis model from single-node as well as distributed real-time systems that are developed using component-based approach. We also provide preliminary guidelines for such translations in the industrial tool suite, Rubus-ICE, used for component-based development of distributed real-time systems.

II. Background and Related Work

A. Response Time Analysis (RTA)

RTA [1], [2] is a powerful, mature and well established schedulability analysis technique. It is a method to calculate upper bounds on response times of tasks (or messages) in a real-time system (or a network). RTA applies to systems where tasks are scheduled with respect to their priorities and which is the predominant scheduling technique used in real-time operating systems today [3].

1) RTA of tasks with offsets: Tindell [4] developed the response-time analysis for tasks with offsets for fixed-priority systems. It was extended by Palencia and Harbour [5]. Mäki-Turja and Nolin [6] reduced pessimism from the offset-based RTA. In this work we will consider the tighter version of the offset-based RTA [6] as part of the end-to-end response-time and latency analysis.

2) RTA of Messages in a Network: In this paper, we will focus only on Controller Area Network (CAN) [7] and its high-level protocols. Tindell et al. [8] developed the schedulability analysis for CAN. It was revisited and revised by Davis et al. [9]. In [10], Davis et al. extended the analysis for CAN network with a mix of priority- and FIFO-queued nodes. In [11], [12], Mubeen et al. extended the existing analysis to support RTA of mixed messages in CAN with priority- and FIFO-queued nodes. Later on, Mubeen et al. [13] extended the existing analysis for CAN to support mixed messages that are scheduled with offsets. In this work we will consider all of the above analysis as part of the end-to-end response-time and latency analysis.

3) Holistic RTA (HRTA): It calculates the upper bounds on the response times of event chains that may be distributed over several nodes in a distributed real-time system. It combines the analysis of tasks in nodes and messages in a network. We consider the HRTA that corresponds to the analysis in [14].

B. End-to-End Latency Analysis

Stappert et al. [15] formally described end-to-end timing constrains in automotive domain. In [16], Feiertag et al. presented a framework for the computation of end-to-end latencies for multi-rate automotive embedded systems. They emphasized on the importance of two end-to-end latencies, i.e., "maximum age of data" and "first reaction" in control systems and body electronics domains respectively. A scalable technique for the computation of end-to-end latencies based on model checking is described in [17]. In this work, we will consider the analysis discussed in [16].

C. The Rubus Concept

Rubus is a collection of methods and tools for model- and component-based development of dependable embedded real-time systems. Rubus is developed by Arcticus Systems [18] in close collaboration with several academic

and industrial partners. Rubus is today mainly used for development of control functionality in vehicles. The Rubus concept is based around the Rubus Component Model (RCM) [19] and its development environment Rubus-ICE (Integrated Component development Environment), which includes modeling tools, code generators, analysis tools and run-time infrastructure. The overall goal of Rubus is to be aggressively resource efficient and to provide means for developing predictable and analyzable control functions in resource-constrained embedded systems.

RCM expresses the infrastructure for software functions, i.e., the interaction between software functions in terms of data and control flow separately. The control flow is expressed by triggering objects such as internal periodic clocks, interrupts, internal and external events. In RCM, the basic component is called Software Circuit (SWC). The execution semantics of an SWC is simply: upon triggering, read data on data in-ports; execute the function; write data on data out-ports; and activate the output trigger. Recently, RCM is extended to support the development of distributed real-time systems [20], [21].

1) The Rubus Analysis Framework (RAF).: The Rubus model allows expressing real-time requirements and properties at the architectural level. For example, it is possible to declare real-time requirements from a generated event and an arbitrary output trigger along the trigger chain. For this purpose, the designer has to express real-time properties of SWCs, such as Worst Case Execution Times (WCETs) and stack usage. The scheduler will take these real-time constraints into consideration when producing a schedule. For event-triggered tasks, response-time calculations are performed and compared to the requirements. RAF supports distributed holistic response-time analysis and shared stack analysis.

III. Research Problem

A. Problem Statement

A component-based real-time system can be modeled with trigger chains (see Figure 4), data chains (see Figure 1) or a combination of both. The end-to-end timing requirements on trigger chains are different from those on data chains. If the system is modeled with trigger chains then the interest, from the schedulability point of view, lies in the calculation of their end-to-end or holistic response times. Hence, the end-to-end deadline requirements placed on trigger chains correspond to holistic response times. In order to check the schedulability of such systems, the holistic response times are compared to the corresponding deadlines. If the holistic response times of all trigger chains are less than or equal to the corresponding deadlines, the system is considered schedulable.

On the other hand, merely computing the holistic response times and comparing them with corresponding end-to-end deadlines is not sufficient to predict the complete timing behavior of the real-time system that is modeled with data chains. There may be over and under sampling in a real-time system due to data chains with independent and varying clock periods for individual tasks. This may cause some values in the data buffers to be over written by new values and hence, the effect of the old values may never propagate to the output. Further, it is also possible to have several duplicates of the output. In such systems, the end-to-end timing requirements, especially in automotive domain [16], are placed on the first reaction to input and age of the data at output. Hence, it is also important to compute the end-to-end latencies (or delays) in such systems. The end-to-end latency refers to the time elapsed between the arrival of a signal at the first task and production of actuation signal (in response to the input signal) by the last task in a data chain [17].

In a real-time system that contains only trigger chains, tasks in a chain are not activated by independent events, in fact, there is only one activating event in the chain. Hence, holistic response times and end-to-end latencies will have equal values. On the other hand, these values are not the same for the systems modeled with data chains. Therefore, a complete analysis of a real-time system modeled with data chains requires the calculation of not only holistic response times but also end-to-end latencies.

When real-time systems are modeled with both trigger and data chains then end-to-end timing requirements are specified on both types of chains in the component model. These requirements should be unambiguously translated into the analysis model which is required by the analysis tools (implementing end-to-end timing analysis). In such systems, the translation of modeled timing requirements and corresponding timing information into a analysis model is challenging due to several issues.

The first issue is the identification of each individual chain with respect to its type from the modeled application. This issue becomes more challenging in the case of mixed-type chains, i.e., when some tasks in the chain are activated by a single trigger while others are activated by independent triggers as shown in Figure 6. The end-to-end timing requirements in such task chains can correspond to both end-to-end response times and latencies. Another related issue arises when a task chain mimics as a data chain as well as a trigger chain by means of trigger merges as shown in Figure 7. A similar ambiguity exists in the extraction of distributed transactions that contain mixed messages [11], [12] in the network in distributed real-time systems.

Not only such chains should be unambiguously identified, their end-to-end timing requirements should also be translated to the end-to-end timing analysis model. Another issue is to extract the tracing information in each chain (from initiator to the terminator). This can be challenging in the case of distributed real-time systems because a distributed transaction may comprise of a data chain in one node and trigger chain in another while these chains communicate via network messages. Finally, the network timing and message-related information should also be extracted when data chains are distributed over several nodes.

We proposed a method to trace trigger chains in component-based distributed real-time systems in [21]. A method is needed for the identification, tracing, and extraction of data chains from component-based real-time systems; and unambiguous translations of their end-to-end timing requirements into the timing analysis model.

Now, we discuss what do we mean by end-to-end timing requirements in data and trigger chains.

B. End-to-end timing requirements in data chains

A single-node real-time system modeled with three SWCs in RCM is shown in Figure 1. These SWCs are activated by independent clocks with different periods, i.e., 8ms, 16ms and 4ms respectively. SWC_A reads the input signals from the sensors while SWC_C produces the output signals for the actuators. Assume that each SWC will be allocated to an individual task by the run-time environment generator. Also assume that WCET of each task is one time unit.

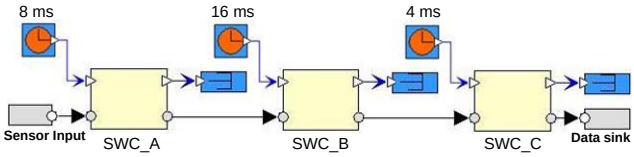


Figure 1. RCM model of a data chain in a single-node real-time system

The time line corresponding to the run-time execution of the three tasks (corresponding to three SWCs) is depicted in Figure 2. It can be seen that there are multiple outputs corresponding to a single input signal. The four end-to-end latency semantics are identified in Figure 2.

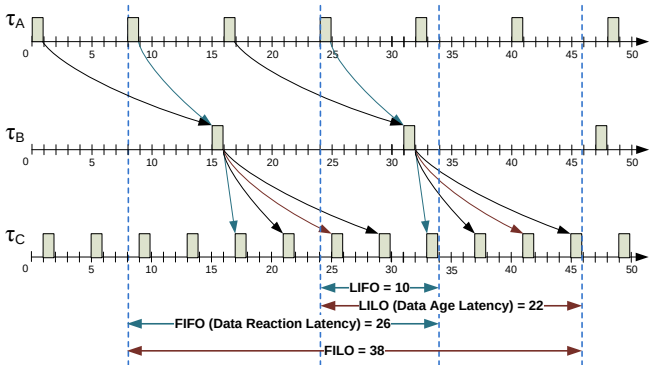


Figure 2. End-to-end latencies of a data chain in a real-time system

1) Last In First Out (LIFO): This latency is equal to the time elapsed between the current non-overwritten release of task τ_A (i.e., input) and corresponding first response of task τ_C (i.e., output).

2) Last In Last Out (LILO): This latency is equal to the time elapsed between the current non-overwritten release of task τ_A and corresponding last response of task τ_C. This latency is identified as "Data Age" in [16]. Data age specifies the longest time data is allowed to age from production by the initiator until the data is delivered to the terminator. This latency finds its importance in control applications where the interest lies in the freshness of the produced data.

3) First In First Out (FIFO): This latency is equal to the time elapsed between the previous non-overwritten release of task τ_A and first response of task τ_C corresponding to the current non-overwritten release of task τ_A. This latency is identified as "Data Reaction" in [16]. Data reaction is the longest allowed reaction time for data produced by the initiator to be delivered to the terminator. This latency finds its importance in the body electronics domain where first reaction to input is important.

4) First In Last Out (FILO): This latency is equal to the time elapsed between the previous non-overwritten release of task τ_A and last response of task τ_C corresponding to the current non-overwritten release of task τ_A.

The data chains may also be distributed over more than one nodes in distributed real-time systems. Consider a model of a two-node distributed real-time system modeled with RCM as shown in Figure 3. The nodes are connected to a CAN network. The internal model of the nodes is also shown in Figure 3. In Node A, SWC_A is triggered by a clock with a period of 8ms. The $OSWC\_A$ component that is responsible for sending a message to the network is triggered by another clock with a period of 16ms. The $ISWC\_C$ is a component that receives a message from the network and is activated by a clock with a period of 4ms. Assume that each component is allocated to a separate task at run-time, i.e., the components SWC_A, $OSWC\_A$ and $ISWC\_C$ are allocated to tasks τ_A, τ_B and τ_C respectively. Since, the system consists of tasks with similar activation patterns and periods as compared to the tasks in the single-node real-time system example discussed above, it can be scheduled in a similar manner as indicated by τ_A, τ_B and τ_C in Figure 2. The end-to-end latencies are also defined in a similar fashion.

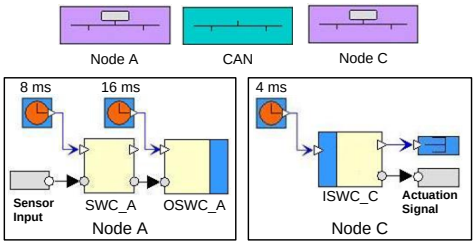


Figure 3. RCM model of a data chain in a distributed real-time system

C. End-to-end timing requirements in trigger chains

An example of a trigger chain that consists of three components is shown in Figure 4. Assume that each components corresponds to a task at run-time. When task τ_{SWC_A} finishes its execution, it triggers τ_{SWC_B}. Similarly, τ_{SWC_C} can only be triggered by τ_{SWC_B} after finishing its execution. There cannot be multiple outputs corresponding to a single input signal. In fact, there will always be one output of the chain corresponding to the input trigger. The focus in a trigger chain is on the calculation of the holistic response-response time only. Hence, the end-to-end timing requirements correspond to the holistic response times. In order to provide a comparison of holistic response time in a trigger chain with the end-to-end latencies in a data chain, assume that the trigger chain shown in Figure 4 is the only chain of tasks in the system. Let the priorities of all tasks be the same while WCET of each task is 1ms. The holistic response time of this trigger chain is equal to the response time of τ_{SWC_C} which is, intuitively, equal to 3ms.

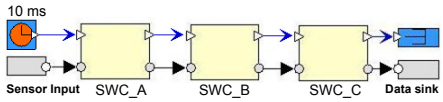


Figure 4. RCM model of trigger chain in a single-node real-time system

Distributed real-time systems can also be modeled with trigger chains. Consider a model of a two-node distributed real-time system modeled with RCM as shown in Figure 5. There is only one triggering ancestor in node A that activates SWC_A. The $ISWC\_C$ in only activated when an interrupt is raised due to the arrival of a CAN message at node C. Once again, the end-to-end timing requirements correspond to end-to-end response times.

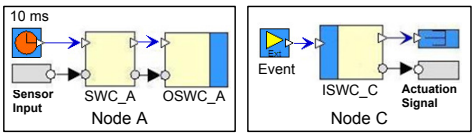


Figure 5. RCM model of trigger chain in a distributed real-time system

IV. Guidelines for the Solution

We provide preliminary guidelines for the development of a method to identify, trace and extract data chains from

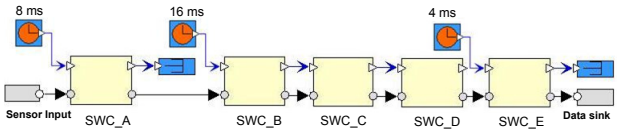


Figure 6. RCM model of a mixed-type chain

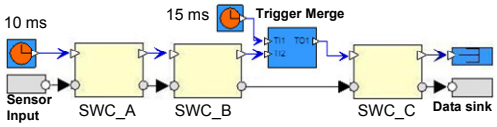


Figure 7. RCM model of a data chain containing trigger merges

component-based real-time systems. The method will also support unambiguous translations of the end-to-end timing requirements specified on data and trigger chains into the analysis model. The new method will be adapted from the existing method in Rubus-ICE [21] to extract the complete tracing information from all data chains.

We will introduce a new object in the component model called trigger map that will extract the triggering information for each task in every chain. Based on this information in the trigger map, an iterative method will determine whether the triggering of every two neighboring tasks in a chain is dependent or independent of each other. If all the triggers are dependent on the initial trigger then the chain will be identified as a trigger chain. If there exists at least one trigger, in the signal map of a chain, that is independent of the rest then the chain will be identified as a data chain. If trigger merges are identified in the trigger map of a chain, it will be regarded as the a data chain. This method will be iterated for all the chains in the system.

The new method will translate the extracted timing information and the trigger map to the analysis model that will input to the analysis tools in XML format. Based on this model, the analysis tools will perform end-to-end response-time and latency analysis. When this method is fully developed, we will implement it in Rubus-ICE as a proof of concept. We believe, this solution will also be applicable to several other component models for real-time systems that use a pipe-and-filter style for component interconnection, e.g., ProCom [22].

V. Summary

We discussed the problem concerning the issues that arise when end-to-end timing requirements are translated into the analysis model from component-based real-time systems that are modeled with data and trigger chains. The end-to-end timing requirements on trigger chains are different from those on data chains. We distinctively identified these requirements in data and trigger chains within single-node and distributed real-time systems. These timing requirements should be unambiguously translated into the analysis model which serves as an input to the analysis tools integrated with the component model. We provided preliminary guidelines for the development of a method to identify, trace and extract data chains and unambiguous translations of their end-to-end timing requirements into a analysis model. Currently, we are developing this method and, in parallel, implementing the end-to-end latency analysis in Rubus-ICE. We plan to provide a proof of concept by conducting an industrial case study using Rubus-ICE.

Acknowledgement

This work is supported by the Swedish Knowledge Foundation (KKS) within the project FEMMVA. The authors would like to thank the industrial partners Arcticus Systems and Volvo Construction Equipment, Sweden.

References

[1] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings, "Fixed priority pre-emptive scheduling:an historic perspective," *Real-Time Systems*, vol. 8, no. 2/3, pp. 173–198, 1995.

[2] L. Sha, T. Abdelzaher, K.-E. A. rzén, A. Cervin, T. P. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. P. Lehoczky, and A. K. Mok, "Real Time Scheduling Theory: A Historical Perspective," *Real-Time Systems*, vol. 28, no. 2/3, pp. 101–155, 2004.

[3] M. Nolin, J. Mäki-Turja, and K. Hänninen, "Achieving Industrial Strength Timing Predictions of Embedded System Behavior," in *ESA*, 2008, pp. 173–178.

[4] K. W. Tindell, "Using offset information to analyse static priority preemptively scheduled task sets," Dept. of Computer Science, University of York, Tech. Rep. YCS 182, 1992.

[5] J. Palencia and M. G. Harbour, "Schedulability Analysis for Tasks with Static and Dynamic Offsets," *Real-Time Systems Symposium, IEEE International*, p. 26, 1998.

[6] J. Mäki-Turja, , and M. Nolin, "Tighter response-times for tasks with offsets," in *Real-time and Embedded Computing Systems and Applications Conference (RTCSA)*. Springer-Verlag, August 2004.

[7] Robert Bosch GmbH, "CAN Specification Version 2.0," postfach 30 02 40, D-70442 Stuttgart, 1991.

[8] K. Tindell, H. Hansson, and A. Wellings, "Analysing real-time communications: controller area network (CAN)," in *Real-Time Systems Symposium (RTSS) 1994*, pp. 259 –263.

[9] R. Davis, A. Burns, R. Bril, and J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, pp. 239–272, 2007.

[10] R. I. Davis, S. Kollmann, V. Pollex, and F. Slomka, "Controller Area Network (CAN) Schedulability Analysis with FIFO queues," in *23rd Euromicro Conference on Real-Time Systems*, July 2011.

[11] S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Extending schedulability analysis of controller area network (CAN) for mixed (periodic/sporadic) messages," in *16th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, sept. 2011.

[12] S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Response-time analysis of mixed messages in controller area network with priority- and FIFO-queued nodes," in *9th IEEE International Workshop on Factory Communication Systems (WFCS)*, may 2012.

[13] S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Worst-case response-time analysis for mixed messages with offsets in controller area network," in *17th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, sept. 2012.

[14] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocess. Microprogram.*, vol. 40, pp. 117–134, April 1994.

[15] F. Stappert, J. Jonsson, J. Mottok, and R. Johansson, "A Design Framework for End-To-End Timing Constrained Automotive Applications," in *Embedded Real-Time Software and Systems (ERTS), 2010*.

[16] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, "A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics," in *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, dec. 2008.

[17] A. C. Rajeev, S. Mohalik, M. G. Dixit, D. B. Chokshi, and S. Ramesh, "Schedulability and end-to-end latency in distributed ecu networks: formal modeling and precise estimation," in *Proceedings of the tenth ACM international conference on Embedded software*, ser. EMSOFT '10. ACM, 2010, pp. 129–138.

[18] "Arcticus Systems," http://www.arcticus-systems.com.

[19] K. Hänninen et.al., "The Rubus Component Model for Resource Constrained Real-Time Systems," in *3rd IEEE International Symposium on Industrial Embedded Systems*, June 2008.

[20] S. Mubeen, J. Mäki-Turja, M. Sjödin, and J. Carlson, "Analyzable modeling of legacy communication in component-based distributed embedded systems," in *37th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Sep. 2011, pp. 229–238.

[21] S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Extraction of end-to-end timing model from component-based distributed real-time embedded systems," in *Time Analysis and Model-Based Design, from Functional Models to Distributed Deployments (TiMoBD) workshop*. Springer, October 2011, pp. 1–6.

[22] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic, "A Component Model for Control-Intensive Distributed Embedded Systems," in *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008)*.

An Energy-aware Scheduling for Real-time Task Synchronization Using DVS and Leakage-aware Methods

Da-Ren Chen
Department of Information Management, National Taichung University of Science and Technology
Taichung, Taiwan, R.O.C.
danny@nutc.edu.tw

You-Shyang Chen
Department of Information Management,Hwa Hsia Institute of Technology
Taipei, Taiwan, R.O.C.
ys_chen@cc.hwh.edu.tw

***Abstract*-Due to the importance of resource allocation and energy efficiency, this paper considers minimizing priority inversion and energy consumptions in the embedded real-time systems. While dynamic voltage scaling (DVS) is known to reduce dynamic power consumption, it also causes increased blocking time of lower priority tasks and leakage energy consumption due to increased execution. We proposed a concept of latency locking to prevent priority inversion using sleeping mode and define a block-free interval in which both DVS and leakage-aware methods can be applied. In order to compute the optimal sleeping time and its duration and to meet the timing constraints, we also propose a weighted directed graph (WDG) to obtain additional task information. By traversing WDG, task information can be updated online and the scheduling decisions could be done in linear time complexity.**

Keywords*-real-time scheduling; priority ceiling protocol; priority inversion.*

I. INTRODUCTION

The power-aware real-time scheduling problem has been well studied, relatively few work address energy-efficient real-time task synchronization. Most embedded real-time applications have shared resources in the system and mutually exclusive access to shared resources. On such a system, real-time tasks can lead to priority inversion if a task is blocked by a lower priority task due to non-preemptive resource sharing. The recent related work is an extension of Priority Ceiling Protocol (PCP) [7] in frequency inheritance. Zang and Chanson [9] proposed a dual-speed (DS) algorithm: One is for the execution of a task when it is not blocked, and the other is adopted to execute the task in the critical section when it is blocked. Jejurikar and Gupta [5] computes two slowdown factor, which can be classified into static slowdown, computed offline based on task properties, and dynamic slowdown, computed using online task execution information. Chen et al. [2] proposed a DVS method using frequency locking concept which can be used to render energy-efficient to the existing real-time task synchronization protocols. The work which is designed with DVS capability to slowdown or speedup the blocked or blocking tasks in the critical section. These methods may receive additional priority inversion, and thus increase the difficulties of schedulability. Our goal is to propose a energy-aware task synchronization protocol, which can minimize the priority inversion and reduce the energy-consumption of the processor. The basic idea is to postpone the intention to the locking on resources invoked by lower-priority tasks, and to construct a blocking-free interval in which the tasks' speed can be reduced using DVS. Additionally, the extended execution due to DVS does not increase the priority inversion.

II. TASK MODEL

This paper studies periodic real-time tasks that are independent during the runtime. Let $\mathcal{T}$ be the set of input periodic tasks, and n denotes the number of tasks. Each task τ_i is an infinite sequence of task instances, referred to as jobs and indexed in order of decreasing priorities. A three-tuple $\tau_i = \{T_i, D_i, C_i\}$ represents each task, where T_i is the period of the task, D_i is the relative deadline with $D_i = T_i$, and C_i denotes the worst-case execution time (WCET). The length of T_i is unique in order to have each task a unique priority index in the rate-monotonic (RM) scheduling. The j^{th} invocation of task τ_i is denoted as $J_{i,j}$ whose actual start and finish times are denoted as $S(J_{i,j})$ and $F(J_{i,j})$, respectively. Notation $s_{i,j}$ denotes the available static slack time for job $J_{i,j}$. Job $J_{i,j}$ could be completed early at time $EC(i, j)$ during its WCET.

All tasks are scheduled on a single processor which supports two modes: dormant mode and active mode. When the processor is switched to the dormant mode, the power-consumption of the processor is assumed S_{dorm}=0 by scaling the static power consumption [1], while the system clock and chipset retain necessary functions to support motoring and waking up processor at right time. To execute jobs, the processor has to be in the active mode with speed S_{active}. The time and power overhead required to switch the processor to the dormant mode can be neglected by treating them as a part of the overhead to turn on the processor. Let E_{sw} and t_{sw} denote the energy and time overhead, respectively, for switching from dormant mode to active mode. When the processor is idle in the active mode, the processor executes NOP instruction at processor speed S_{idle} for low-power consumption. Additionally, when the idle interval is longer than break-even time $\frac{E_{sw}}{P(S_{idle})}$, turning it to the dormant mode is worthwhile. The DVS model is similar to those in lpWDA [6] and can be abridged here.

We assume that semaphores are used for task synchronization. All tasks are assumed to be preemptive, while the access to the shared resources must be serialized. Therefore, task can be *blocked* by lower priority tasks. When a task has been granted access to a shared resource, it is said to be executing in its critical section [8]. The k^{th} critical section of task τ_i is denoted as $z_{i,k}$ which is properly

nested. Each task specifies the access to the shared resource types and the required WCETs. With the given information in [4], we can compute the maximum blocking time for a task. In the different resource synchronization protocol, such as PCP [7], each job might suffer from a different amount of blocking time from lower-priority task, due to access conflict. The goal of this paper is to propose an energy-aware real-time scheduling with task synchronization based on PCP, which can minimize the priority inversion while reducing the energy-consumption of the processor. We propose a data structure called weighted directed graph (WDG) which expresses possible priority inversion online. By traversing WDG, we can not only postpone the intention of locking on the resources invoked by lower-priority tasks but also construct a blocking-free interval to slowdown the tasks speed using DVS methods.

III. MOTIVATING EXAMPLE

Suppose that we have three jobs J_1 , J_2 and J_3 , and two shared data structures protected by the binary semaphores z_1 and z_2 in the system. In accordance with PCP, the sequence of events is depicted in Fig. 1(a). A line at a lower level indicates that the corresponding job is in blocked or preempted by a higher-priority job while the processor mode is active. A line raised to a higher level denotes that the job is executing, and the absence of a line denotes that the job has not yet been initiated or has completed. A bold line at low level denotes that the processor has been switched in dormant mode. Suppose that

- At time t_0, J_3 is initiated and it then locks semaphore z_1.
- At time t'_1, J_2 is initiated and preempts J_3.
- At time t_2, J_2 cannot lock z_1, and J_3 inherits the priority of job J_2 and resumes execution.
- At time t_3, J_1 preempts J_3 in the critical section of z_1 and executes its noncritical section code.
- At time t_4, J_1 attempts to enter its critical section z_1 and is blocked by J_3 due to priority ceiling, and J_3 inherits the priority of job J_1.
- At time t_5, J_3 exits its critical section z_1 and returns to its original priority. J_1 is awakened and locks z_1.

The priority inversions are $[t_2, t_3]$ and $[t_4, t_5]$.

This research work is motivated by the significant priority inversion and power consumption due to unused slack time and context switches. The objective is to minimize the priority inversion while reducing energy-consumption. When the available static slack time (unused time in the WCET schedule) or dynamic slack (occurred in the early-completed task) is larger than break-even time, the lower-priority task intent to lock a semaphore can be postponed until the start time of a higher-priority task. A practical approach is to postpone the task execution by switching processor to dormant mode. During the sleeping time, system still has awareness of the arrival of other jobs, and awakes processor at proper time. The example in Fig.1(b) postpones the request of lower-priority task intent to a lock semaphore. At time t_1, J_3 has available slack in interval $[t_9, t_{10}]$ with length longer than break-even time. When a system is conscious that J_3 has intent to lock z_1, it computes the upcoming start time of higher priority tasks that might be blocked by J_3 according to PCP. In the example, J_1 and J_2 could be blocked by J_3 due to z_1, and the lengths of interval $[t_1, t_1']$ are less than the available slack. Therefore, processor switches to dormant mode at time t_1 until the start time of J_2. At time t_1', processor becomes *active* and J_2 preempts J_3 such that J_3 is still unable to lock z_1, and thus J_2 could lock z_2 at time t_2. However, J_1 could be blocked when it intends to lock z_1 if J_2 successfully lock z_2 at t_2. To further reduce priority inversion, job J_2's intent to lock z_2 should be postponed after t_3. Therefore, comparing to the result of Fig. 1(a), the idea eliminates all priority inversions in intervals $[t_3, t_7]$.

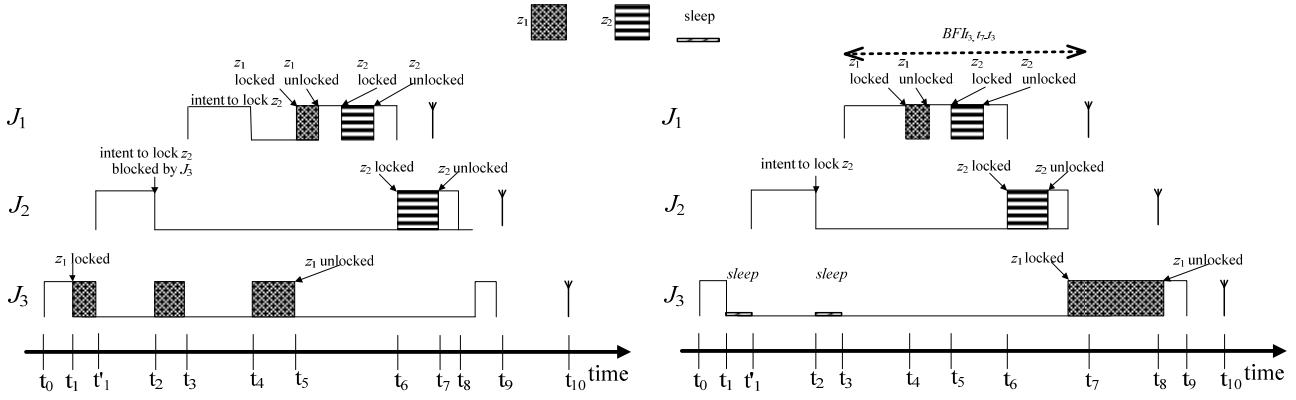


Fig.1. The task synchronization of (a) primitive PCP and (b) latency locking method

IV. LATENCY LOCKING PCP

In this research work, PCP is extended with the concept of latency locking, referred to as LL-PCP. The idea is to do pre-analysis of possible priority inversion and available slack time in the schedule. The objective is to derive the best timing and duration for switching processor to dormant mode, and thus minimize priority inversion. To understand and control the sequence of intent to lock resources, tasks are organized as a WDG reduced from the resource allocation bipartite graph in [4]. Let $G=(U, V, E)$ denote a bipartite graph whose partition of vertices has two subsets U and V. E denotes the set of edges of G, and U denote a task set $\mathfrak{T}$. Let WDG($\mathfrak{T}$, A) denote a weighted directed graph whose vertices in $\mathfrak{T} \subseteq U$ are arranged according to their task indices. For each edge $e_{u,v} \in E$, $\tau_u \in U$ and $z_v \in V$, the set of arcs A in WDG are generated as follows

Step1. For any pair of vertices $\tau_x, \tau_y \in U$ and $x>y$, a solid arc $a(x, y) \in A$ is directed from τ_x to τ_y if there exists two or more edges $e_{x,v}$ and $e_{y,v}$ in G where $z_v \in V$.

Step2. For any pair of vertices $\tau_x, \tau_w \in U$ and $x>w$, a dotted arc $a(x, w) \in A$ is directed from τ_x to τ_w if there exists a vertex $\tau_y \in U$, $w>y$, and τ_x and τ_y satisfy Step1.

Step3. In WDG, for any pair of vertices with multiple arcs, eliminate the dotted arcs having the same blocking time as that of one of their solid arcs.

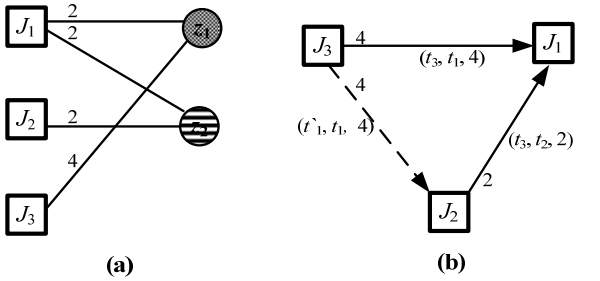


Fig.2 Graph reduction from (a)bipartite graph to (b)WDG

Fig.(2) illustrates the graph reduction from bipartite to WDG. In the bipartite graph, each indirect edge is labeled with the time required to access the resources. Different from the bipartite graph, WDG has a vertex for each task but resources. A task τ_L directly blocks a higher-priority task τ_H is represented by an solid arc $a(\tau_L, \tau_H)$ from task vertex τ_L to τ_H, while an indirect block is represented by an dotted arc. In Fig. 2(a), the bipartite graph is derived from Fig.1(a) and can be reduced to the WDG in Fig.2(b). The maximum priority inversion of J_2 is an indirect blocking incurred by J_3. We may label each arc by a 3-tuple, the first element of each 3-tuple give the actual starting time of higher-priority tasks and defined as $S(\tau_H)$, while the second element gives the locking time of τ_L on semaphore z and denoted as $L(\tau_L, z)$. The last element specifies the duration of the maximum priority inversion and denoted as $I(\tau_L, \tau_H)$. The first two elements are updated during runtime while the third element is derived directly from the arcs in WDG.

The example of the 3-tuple labels is illustrated in Figure 2(b), we have the following definitions.

Algorithm LL-PCP

Input: a set of task $\mathfrak{T}$, a set of resources $\mathfrak{R}$

(Offline part)

1. Reduce bipartite graph to WDG($\mathfrak{T}$, A);
2. Compute the value of $I(\tau_L, \tau_H)$ with respect to each arc $a(\tau_L, \tau_H)$ in WDG;

(Online part)

On arrival of a job J_i

3. Identify a set of tasks $\mathfrak{T}_H$ containing higher priority task τ_H than that of J_i;
4. Compute the value of $REW(\tau_i, \tau_H)$ for each arc $a(\tau_i, \tau_H)$ in WDG and $\tau_H \in \mathfrak{T}_H$;
5. Construct a set A_i' of outgoing arcs of τ_i, $A_i' = \{ a(\tau_i, \tau_H) | \alpha_{i,H} \geq \frac{E_{sw}}{P(S_{idle})} \}$
6. Compute the static available slack s_H for each job in $\mathfrak{T}_H$;
7. Compare each s_H to the corresponding α value of the arcs in A_i';
8. Construct an arc set $A_i'' \subset A_i'$ where $A_i'' = \{a(\tau_i, \tau_x) | s_H \geq \alpha_{x,i}, a(\tau_i, \tau_x) \in A_i' \text{ and } \tau_x \in \mathfrak{T}_H\}$;
9. Search for an arc $a(\tau_i, \tau_x)$ in A_i'' with the maximum value of $REW(\tau_i, \tau_x)$ where $\tau_x \in \mathfrak{T}_H$;

On beginning of one of the intervals in $ESI_{L,x}$

11. Switching the processor to S_{dorm} until the end of the interval;

On turning the processor to the active mode at time t

12. *Schedule the highest priority job in the ready queue*; On early-completion of a job at time t;
13. Compute dynamic slack time due to early completion;

On completing or beginning a job J_i at time t

14. **IF** completes early **THEN**
    obtain dynamic slack s_i^d from $F(J_i)$–$EC(J_i)$;
15. Set BFI=[S(J_x), $F(J_x)$] according to the recently carried out $ESI_{L,x}$;
16. Slowdown the speed of tasks whose deadlines are earlier than $F(J_x)$ using lpWDA[6];

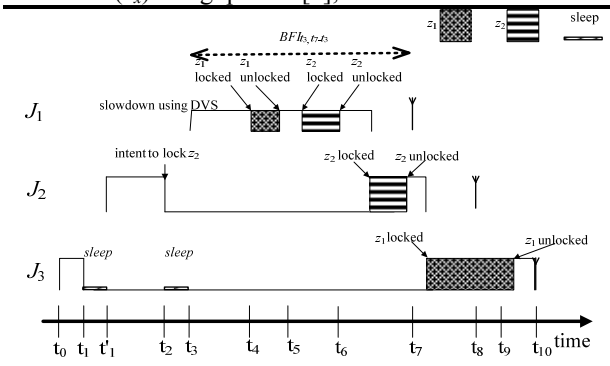


Fig.3 An example using DVS scheduling

Definition 1. In order to prevent job J_L from blocking job J_H, the expected sleep interval (ESI) for J_L is defined as

$$ESI_{L,H} = [L(J_L, z), S(J_H)) - \bigcup_{\forall \tau_\lambda \in \mathfrak{T}, \lambda < L} NC_\lambda^{[L,H]}. \quad (1)$$

where $NC_{\lambda}^{[L,H]}$ denotes the set of noncritical-section intervals of job J_λ in interval [$L(J_L, z)$, $S(J_H)$). The length of $ESI_{L,H}$ denotes as

$$\alpha_{L,H}=S(J_H) - L(J_L, z). - \sum_{\forall \tau\_\lambda \in I, \lambda<L} \left| NC_{\lambda}^{[L,H]} \right| \quad (2)$$

Definition 2. defines the expected reduction of priority inversion (*RPI*) due to the processor sleeping in the $ESI_{L,H}$. The value of *RPI* is derived from

$$\beta_{L,H}= I(\tau_L, \tau_H) - \alpha_{L,H} . \quad (3)$$

According to equations (1), (2) and (3), we define a reward function for each arc in WDG.

Definition 3. A reward function for each arc in WDG is

$$REW(\tau_L, \tau_H)=\frac{\beta_{L,H}}{\alpha_{L,H}}. \quad (4)$$

The *reward* for an arc is referred to the reduction of priority inversion time if the processor is switched to sleep during the interval $ESI_{L,H}$. Whenever a new job J_i arrives, the value of $REW(\tau_L, \tau_i)$ with respect to each arc is refreshed. The larger the value of *REW*, the longer the priority inversion will be avoided. For example, in Fig.2(b), the values of $\alpha_{3,1}$ and $\alpha_{3,2}$ are set respectively $x= t_3 - t_1 - (t_2 - t'_1)$ and $y= t'_1 - t_1$, and the values of $\beta_{3,1}$ and $\beta_{3,2}$ are respectively $4 - x$ and $4 - y$. In accordance with equation 4, the values of $REW(3,1)$ and $REW(3,2)$ are $\frac{4-x}{x}$ and $\frac{4-y}{y}$, and obviously $REW(3,1) < REW(3,2)$. Assuming that available slack for τ_3 is larger than the values of x and y, the proposed algorithm switchs the processor to sleep in the duration of [t_1, t'_1], and traverses from vertex J_3 to J_2. In the vertex J_2, we can traverse from J_2 to J_1 by switching the processor to sleep mode in interval [t_2, t_3].

Definition 4. (Blocking-free interval, *BFI*)

A time interval $BFI_{t,\ell}$ is said to be blocking free in the real-time task synchronization if the interval [t, $t+\ell$] does not have any priority inversion.

Lemma 1. When the time at which J_L has intent to lock z is later than the $S(J_H)$, they do not give rise to priority inversion during interval [$S(J_H)$, $F(J_H)$].

***Proof sketch*:** In accordance with WDG, J_H has higher priority than J_L, as soon as J_L begins after $S(J_H)$, J_L cannot lock z until J_H completes. Therefore, J_H is not preempted by J_L until the completion time of J_H.

Obviously, jobs J_H and J_L do not give rise to the priority inversion in the interval [$F(J_H)$, D_H] when J_H completes before F(JH). Therefore, when the processor sleeps in interval $ESI_{L,H}$, the value of *BFI* can be derived from

$$BFI_{t,\ell}=[S(J_H), D_H] \quad (5)$$

for all jobs J_H are successors of J_L in WDG.

The purpose of BFI is to identify the jobs for saving more energy using DVS. The jobs whose deadlines are in the BFI interval can compute their available slack time to decrease their speed and satisfy their timing constraints. The slack computation such as lpWDA [6] can be applied without modification to our method. An example is presented in Fig.3. By updating the information of arcs in WDG during runtime, we can traverse the WDG by following the current job and make decisions on switching the processor to active or dormant mode.

V. CONCLUSIONS

This work-in-progress continuously improves energy-efficiency of real-time task synchronization with speed switching overhead consideration. By using DVS and leakage-aware techniques, we decrease not only the priority inversion but also energy consumption in the real-time systems. The objective is to minimize the priority inversion and reduce both dynamic and leakage energy, provided that the schedulability of tasks is guaranteed. By traversing the vertex of WDG, the scheduling decisions can be done efficiently during the runtime. Another characteristic is that, in the proposed concept, DVS does not worsen the situation of inevitable priority inversion.

For further study, we shall explore an evaluation function that provides suggestions on how to use DVS or leakage-aware technique during runtime. Future research and experiments in these areas may benefit several mobile system designs.

VI. REFERENCES

[1] Butts, J. Adam, and Sohi, G. S. 2000. A Static Power Model for Architects. In *Proceedings. of the 33rd Annual International Symposium on Microarchitecture* (In Monterey, California from Dec. 10 - 13). MICRO-33, 191-201.

[2] Chen, J.-J., and Kuo, T.-W. 2006. Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Ottawa, June 14-16, 2006). *LCTES* 06. ACM, 153–162.

[3] Irani, S., Shukla, S., and Gupta, R. 2003. Algorithms for power savings. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (On the Inner Harbour Baltimore, MD, USA Jan. 12-14, 2003). DA 03. ACM, New York, NY, 37–46.

[4] Jane W. S. Liu. Real-time systems. Prentice Hall PTR Upper Saddle River, NJ, USA, (2000), ISBN:0130996513.

[5] Jejurikar, R., and Gupta, R. K. 2005. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *Proceedings of the 42nd Design Automation Conference* (San Diego, CA, USA, June 13-17). *DAC* 05. ACM, New York, NY, 111–116.

[6] Kim, W., Kim,J., and Min, S. L. 2003. Dynamic voltage scaling algorithm for fixed-priority real-time systems using work-demand analysis. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design* (ISPLED'03), ACM Press, New York, NY, 2003, pp. 396–401.

[7] Sha, L., Rajkumar, R., and Lehoczky,J. P. 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. on Computers*, 39 (Sept. 1990), no.9, 1175-1185.

[8] Silberschatz, A. P., Galvin, B., and G. Gagne. 2011. Operating System Concepts. John Willey and Sons, Inc., (2011).

[9] Zhang, F., and Chanson, S. T. 2002. Processor voltage scheduling for real-time tasks with non-preemptible sections. In *23rd Proceedings IEEE Real-Time Systems Symp.*, (Austin, TX, Dec. 2002). RTSS 02. IEEE, 235–245.

Dynamic Scheduling Algorithm for Parallel Real-time Graph Tasks

Manar Qamhieh, Serge Midonnet
Université Paris-Est, France
{manar.qamhieh,serge.midonnet}@univ-paris-est.fr

Laurent George
ECE-Paris, France
lgeorge@ece.fr

***Abstract*—In this paper, we propose a dynamic global scheduling algorithm for a previously-presented specific model of real-time tasks called "Parallel Graphs" [1], based on the Least Laxity First priority assignment policy "LLF", we apply LLF policy on each subtask in the graphs individually, taking in consideration their precedence constraints. This model of tasks is a combination of graphs and parallelism, in which each subtask in the graph can execute sequentially or parallel according to its number of processors defined by the model. So we study parallelism possibilities in order to find the best structure of the tasks according to the practical specifications of the system.**

I. Introduction

Physical constraints such as chip size and continuous heating forced processors' manufacturers to produce multi-processor systems, and as they are constantly growing, software parallelism has been widely studied and applied in practice.

However, parallelism in real-time embedded systems is still a rising challenge with many open questions to be studied. There are parallel task models exist in practice, such as the fork-join model used in OpenMP [2] and has been studied in [3], and a more general model of parallel tasks has been proposed recently in [4] which overcomes the restrictions of the fork-join model. Those models consider the tasks as a sequence of parallel and sequential tasks.

The graph model of real-time tasks is a general representation of the models described previously. In our previous work [1], we proposed a new model of real-time tasks called "Parallel Graphs", which is a combination of graphs and parallel tasks. By this we added an inner parallelism level to the graph as will be described in II. In this paper, we will extend our previous work by proposing parallelizing options a dynamic scheduling algorithm for the parallel graphs.

In this paper, firstly we will describe our task model in section II. In section III we will discuss the various parallelizing possibilities for parallel graphs. Section IV will present our dynamic scheduling algorithm. Finally in section V, we will finish the paper by concluding and giving perspectives.

II. Task Model

In [1] we presented previously a new model of real-time tasks called "parallel graphs". In this model, Each parallel real-time task is represented by a directed acyclic graph (DAG), which is a collection of subtasks and directed edges representing the execution flow of the subtasks and the precedence constraints between them.

Each parallel graph task τ_i consists of a set of q_i subtasks and it is denoted as:
$\tau_i = (\{\tau_{i,1}, \tau_{i,2}, ..., \tau_{i,q_i}\}, D_i, P_i)$, where D_i is the deadline of the graph and P_i is its period. Each subtask $\tau_{i,k}$ is represented as the following:
$\tau_{i,k} = \{c_{i,k}, m_{i,k}\}$, where $c_{i,k}$ is the total worst execution time of the subtask, and $m_{i,k}$ is the maximum degree of parallelism of $\tau_{i,k}$, which means that $\tau_{i,k}$ can be scheduled on $m_{i,k}$ parallel processors at the most.

Figure 1 shows an example of parallel graph task.

Precedence constraint means that each subtask can start its execution when all of its predecessors have finished theirs. If there is an edge from subtask $\tau_{i,u}$ to $\tau_{i,v}$, then we can say that $\tau_{i,u}$ is a predecessor of $\tau_{i,v}$, and $\tau_{i,v}$ has to wait for $\tau_{i,u}$ to finish its execution before it can start its own. Each subtask in the graph may have multiple predecessors, and multiple successors as well, but each graph should have a single source and a single sink vertex.

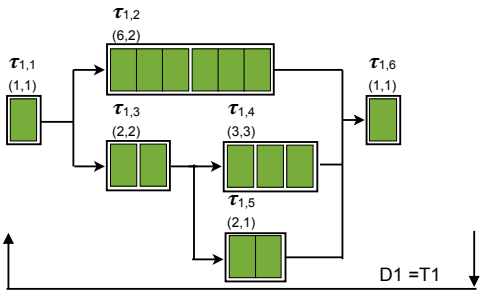


Figure 1. Example of the parallel graph model.

In this work, we study the global scheduling of n synchronous periodic parallel real-time graphs with implicit deadlines on m identical processor system. A task set is denoted as $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, where each graph has period equals to its deadline. The schedulability is studied on the hyper period of each taskset.

A. Notation

Definition 1: Critical path [1] of a graph τ_i is the longest path in the graph through its subtasks when respecting their dependencies.

$$CP_i = \sum_{j \in \text{critical subtasks}} c_{i,j}$$

The critical path of the graph τ_1 from Figure 1 is $(\tau_{1,1}, \tau_{1,2}, \tau_{1,6})$ and $CP_1 = 8$.

Definition 2: The worst case execution time of a graph C_i is the total execution time of all the subtasks in the

graph τ_i when executed sequentially.

$$C_i = \sum_{j=1}^{q_i} c_{i,j}$$

Definition 3: Laxity of a parallel graph L_i is the difference between its deadline and its critical path execution time.

$$L_i = D_i - CP_i$$

III. Different parallelizing algorithms

In our previous paper [1], we described a parallelizing algorithm which finds the best structure of the parallel graph according to the response time of the graph, by maximizing the number of parallelized subtasks in the graph. This algorithm does not consider the number of processors in the system, so there might be better structures of the parallel graphs when considering the actual specifications of the system like the number of processors, the scheduling policy, etc.

In this section we will discuss 2 possible parallelizing algorithms, a maximizing and minimizing algorithms according to the level of parallelism.

A. Maximizing parallelism

The parallelizing algorithm proposed in [1] is an iterative algorithm whose aim is to parallelize the maximum number of necessary subtasks in the graph up to their maximum level of parallelism, on the basis of the following algorithm:

- Find the critical path of the graph using the depth-first search algorithm.
- Parallelize all the subtasks in the critical path up to their maximum level of parallelism.
- Repeat the 2 previous steps until there is a critical path with no parallelizable critical subtasks.

B. Minimizing parallelism

Another approach can be considered for this type of graph which is the reverse of the one in III-A. We can find the best structure of the graph according to the number of processors in the system, by trying to stretch the graph as long as possible in order to execute on a minimum number of processors without missing their deadline, and by filling the laxity of the graph with a maximum number of subtasks.

According to the following equation, a parallel graph τ_i can execute sequentially on 1 processor if:

$$\frac{D_i}{C_i} \geq 1 \tag{1}$$

Where D_i is the deadline of the graph task, and C_i is defined in II-A.

If this test fails, then we have to reduce the sequential execution time of the graph by parallelizing some of its subtasks using the following algorithm:

- Apply equation 1 on the parallel graph τ_i.
- If the test succeeds, then τ_i can execute on a minimum number of processors without missing its deadline.
- If the test fails, we calculate the critical path of τ_i, parallelize the critical subtasks in order to reduce its sequential worst case execution time C_i.
- Repeat the first step on the newly parallelized graph τ_i until the test succeeds.

C. Other possibilities

Choosing the best structure of the parallel graph is not an easy process. It is controlled by a lot of restrictions and limitations of the system, for example, using the maximizing parallelism algorithm will reduce the response time of the graph if executed on a system with a large number of processors, and theoretically, it will increase the number of preemptions and job migrations while scheduling (depends on the used priority assignment algorithms).

The minimizing algorithm will increase the response time of the graph while reducing the number of processors needed, which will decrease the energy consumption as a result.

There is a large number of parallelizing structures for the parallel graph task, which affects the schedulability of the tasks, the migration and preemption costs. In the future we aim to study those various possibilities and their effects by comparing them using a real-time simulation tool, and taking into account the different characteristics of embedded systems, such as the number of processors, energy consumption, etc.

IV. Scheduling Parallel Graphs

In this section, we propose a global preemptive scheduling algorithm on an implicit-deadline parallel graph task set Γ_i of n graphs, on the hyper period of the taskset:

$$hyper(\Gamma_i) = LCM(\tau_j), \forall j : 1 \leq j \leq q_i$$[1]

Since the active subtasks of each graph share the same period and deadline, we decided to use a dynamic priority assignment policy based on the least laxity first technique (LLF), which gives higher priority to tasks with lower laxity (slack time). Scheduling algorithms based on this priority assignment are optimal on mono-processor systems but not on multiprocessor systems, unless laxity priorities are verified all the time during the scheduling process to make sure delayed tasks gain higher priorities in time.

A. Scheduling algorithm

After applying a parallelizing algorithm on each graph τ_i in Γ_i, the generated graph task contains both sequential and parallel subtasks. A sequential subtask needs one processor to execute on, while parallel subtasks execute on multiple processors at the same time. Here we should say that we are interested in input-data parallelism, in which the same code is repeated on multiple processors while only the input data are changed.

As described in Section II, each graph task τ_i consists of q_i real-time subtask each has a WCET of $c_{i,j}$, and due to the precedence constraint on the subtasks of the

[1]LCM is the Least Common Multiple

same graph, a subtask $\tau_{i,j}$ cannot be activated unless all of its predecessor subtasks finish their execution. Because of that not all of the subtasks in the graph are activated at the same time or at the very instant of activating the graph.

There are 2 types of laxity in τ_i, a general laxity of the graph as whole, denoted as L_i and described in II-A, and a subtask laxity for each subtask $\tau_{i,j} \in \tau_i$.

As explained in our previous work in [1], when a parallelizing algorithm is applied on a graph task τ_i, we can also calculate the laxity of each subtask in τ_i, by calculating the earliest and the latest finishing time of the execution time of each of them, the difference between those 2 time values is the laxity of the subtask. A subtask with laxity equaling to 0 is a critical subtask in the graph.

In order to organize the scheduling process of the graph set, we will consider 3 types of subtasks: active jobs, executing jobs and completed jobs. The active list contains the jobs that are activated either by the activation of the original graph (jobs of the starting subtask in the graph), or when the predecessors of a subtask complete their execution. When a job starts its execution, it will be moved to the executing list until its execution is over when it will be moved to the completed list. If an executing job is interrupted by a higher priority job, it will be moved back to the active list.

For each instant in time t where

$$0 \leq t \leq hyper(\Gamma_i),$$

we calculate the dynamic priority $Pr_{i,j}(t)$ for each active job of subtask $\tau_{i,j}$ of each graph task in Γ_i, where:

$$Pr_{i,j}(t) = L_i + L_{i,j} - (t - A_{i,j}) \tag{2}$$

where $A_{i,j}$ is the activation time of $\tau_{i,j}$, subtasks with lower $Pr(t)$ values have higher priorities.

The scheduling algorithm:

- At t = 0:
  Each graph task $\tau_i \in \Gamma$ is activated (since we consider synchronous activation), which means all the starting subtasks $\tau_{i,1}$ are activated as well and added to the active list. Then we calculate $Pr(0)$ for all of the subtasks in the active list using Equation 2.
  According to the number of processors in the system, we start executing the subtasks with the highest priority which are moved to the executing list at the same time.
  If an executing subtask $\tau_{i,j}$ is a parallel subtask according to the parallelizing algorithm, then it will need $m_{i,j}$ available processors in order to enable all of its parallel parts to execute concurrently.
- $\forall t$ where $0 < t \leq hyper(\Gamma_i)$:
  If an executing job finishes its execution at this instant of time, it will be moved to the completed list. And if all the predecessors of a subtask are in the completed list, then its job will be activated at this instant of time and added to the active list.
  Since we use dynamic priority assignment, at each instant of time we re-calculate $Pr_{i,j}(t)$ for all subtasks $\tau_{i,j}$ in the active list. By this, the priority of delayed active subtasks will be increased in time since their laxity decreased.
  According to the newly assigned priorities of the active subtasks, we fill the processors available in the system, and if there are active subtasks with higher priority than the ones already executing, they are allowed to interrupt their execution.
- The scheduling algorithm of the graph set Γ will fail if -at any instance of time t- a job in the active list reaches a $Pr(t) = 0$ without having an available processor to execute on, since at $t + 1$ this job will have a negative laxity and miss its deadline.

In the case of equal priorities between 2 active jobs, we choose the executing one randomly, but if an active job and an executing job have the same priority, we give the priority to the executing job to continue its execution without allowing the active one to interrupt the execution of the other. In that way we reduce the number of unnecessary cost of preemptions and migration.

Using this dynamic global scheduling algorithm, we scheduled each subtask in the graphs individually, by assigning priorities as shown in the previous algorithm. Precedence constraints between the subtasks due to the structure of the graph were visible only in the activation process. However, this scheduling algorithm was different from the previous scheduling techniques applied on the real-time tasks of the graph model seen in the literature. For example in [4], the authors propose to use a decomposition algorithm in order to assign local deadlines to the subtasks in the task, and to schedule each segment of tasks as independent tasks on multi-processor systems.

B. Scheduling example

In this section, we will apply the above-mentioned scheduling algorithm described in IV-A on a graph task set consisting of 2 graphs on a 2-processor system.

graph task set $\Gamma = \{\tau_1, \tau_2\}$
$\tau_1 = (\{\tau_{1,1}(1,1), \tau_{1,2}(3,1), \tau_{1,3}(2,2), \tau_{1,4}(1,1)\}, 10, 10)$
$\tau_1 = (\{\tau_{2,1}(1,1), \tau_{2,2}(1,1), \tau_{2,3}(1,1), \tau_{2,4}(1,1)\}, 5, 5)$

The graphs in the task set have implicit deadlines, and all subtasks of the same graph share the same period and deadline. The scheduling algorithm is studied on the hyper period of the task set:

$$hyper(\Gamma) = LCM(10, 5) = 10$$

Figure 2 shows the graphs of the task set with the precedence constraints.

Graph τ_1 has a parallel subtask $\tau_{1,3}$ which needs 2 processors available at the same time in order fo it to execute, or its execution will be delayed otherwise. All the other subtasks in this example are sequential.

By applying the critical path calculations described previously in [1], we find that the laxity of the graph τ_1 equals to 5, and 2 for τ_2, and all the subtasks of both

Table I
SCHEDULING TABLE OF 2 GRAPH TASKS

	Active subtasks in order of priority			
Time	Highest			Lowest
t=0	$Pr_{2,1} = 2$	$Pr_{1,1} = 5$		
t=1	$Pr_{2,2} = 2$	$Pr_{2,3} = 2$	$Pr_{1,2} = 5$	$Pr_{1,3} = 6$
t=2	$Pr_{2,4} = 2$	$Pr_{1,2} = 4$	$Pr_{1,3} = 5$	
t=3	$Pr_{1,2} = 4$	$Pr_{1,3} = 4$		
t=4	$Pr_{1,3} = 3$	$Pr_{1,2} = 4$		
t=5	$Pr_{2,1} = 2$	$Pr_{1,2} = 3$	$Pr_{1,3} = 3$	
t=6	$Pr_{2,2} = 2$	$Pr_{2,3} = 2$	$Pr_{1,3} = 2$	
t=7	$Pr_{1,3} = 1$	$Pr_{2,4} = 2$		
t=8	$Pr_{2,4} = 1$	$Pr_{1,4} = 1$		
t=9				
t=10				

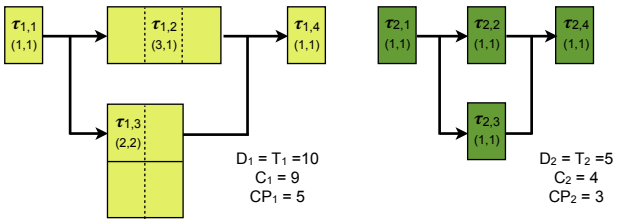


Figure 2. Graph taskset example.

graphs don't have local laxities (they are critical subtasks), except for subtask $\tau_{1,3}$ which has a laxity $L_{1,3} = 1$.

At $t = 0$, the first subtasks of the each graph are activated ($\tau_{1,1}$& $\tau_{2,1}$), according to the mentioned-above equation 2, we can calculate the priority of the subtasks as the following:

$$Pr_{1,1} = 5 + 0 - (0 - 0) = 5$$
$$Pr_{2,1} = 2 + 0 - (0 - 0) = 2$$

According to the results, $\tau_{2,1}$ has higher priority than $\tau_{1,1}$, but since we have 2 processors available, both subtasks will be scheduled. Those calculations will be repeated at each instant of time in the hyper period of the task set, unless a deadline miss occurs before the end of the period. Table IV-B shows the priorities of the active subtasks of both graphs over the hyper period of the task set, while Figure 3 shows the final scheduling of the subtasks on the 2 processors of the system. We can notice that our proposed global preemptive scheduling algorithm based on LLF has succeeded in scheduling the taskset without any subtask misses its deadline.

V. PERSPECTIVE AND CONCLUSION

In this paper, we have introduced a dynamic global scheduling algorithm on multi-processor systems, for a specific real-time task set of parallel graph models, based on the Least Laxity First "LLF" job priority assignment in order to schedule each subtask in the graphs according to their laxity while considering the global deadline and period of the original graph with no need to assign a local deadline for them, we have also shown by an example the schedulability of this algorithm on a task set of the graph model.

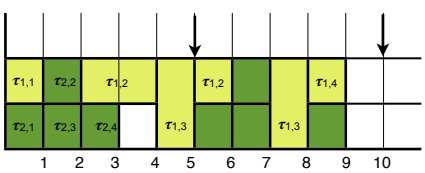


Figure 3. Graph taskset scheduled using LLF.

The graph model of tasks has been studied recently in the literature, but adding the parallelism constraint to this model has raised a schedulability challenge and made it more complicated. That is why we presented 2 parallelizing algorithms in this paper, both algorithms depending on the constraints of the embedded systems which we aim to study in more details in the future.

In order to provide valid results to show the performance of our proposed scheduling algorithm, we started implementing it on a simulation tool called "YARTISS" [5], developed by a real-time team in the research laboratory of Universit Paris-Est. By using this simulator we will be able to compare the performance of our own scheduling algorithm with other techniques and algorithms used in the literature, which will allow us to enhance its performance with respect to the practical issues of real embedded systems such as a limited number of processors, optimizing the schedulability in order to reduce the energy consumption by reducing the number of migrations and preemptions.

In parallel, we would like to study real-time scheduling anomalies and provide real-time feasibility tests for the proposed algorithm, in order to support the simulation results.

Finally, we hope to apply our model of tasks on real embedded systems, and propose adjustable techniques in order to enhance their performance such as schedulability and energy consumption.

REFERENCES

[1] M. Qamhieh, S. Midonnet, and L. George, "A Parallelizing Algorithm for Real-Time Tasks of Directed Acyclic Graphs Model," in *RTAS Work-In-Progress Session*, 2012.

[2] "Openmp." [Online]. Available: http://www.openmp.org

[3] K. Lakshmanan, S. Kato, and R. (Raj) Rajkumar, "Scheduling Parallel Real-Time Tasks on Multi-core Processors," in *Proceedings of the 31st IEEE Real-Time Systems Symposium*, 2010.

[4] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core Real-time Scheduling for Generalized Parallel Task Models," in *The 32nd IEEE Real-Time Systems Symposium*, 2011.

[5] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, and M. Qamhieh, "YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-time Scheduling Algorithms," in *WATERS*, 2012.

A New Technique for Analyzing Soft Real-Time Self-Suspending Task Systems *

Cong Liu and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

We consider the problem of globally scheduling soft real-time sporadic self-suspending task systems on multiprocessors. Existing analysis methods are pessimistic, yielding $O(n)$ utilization loss where n is the number of tasks in the system. Unless the number of tasks is small and suspension delays are short, such methods entail significant capacity loss. We identify the fundamental sources that cause pessimism in existing methods, and propose a new analysis technique that entails only $O(m)$ suspension-related utilization loss, where m is the number of processors.

1 Introduction

In many real-time systems, suspension delays may occur when tasks interact with external devices such as I/O devices. Unfortunately, schedulability in real-time systems is negatively impacted by such delays if deadline misses cannot be tolerated [4]. In this paper, we consider whether, on multiprocessor platforms, such negative impacts can be ameliorated if task deadlines are soft. Our focus on multiprocessors is motivated by the advent of multicore platforms. There is currently great interest in providing operating-system support to enable real-time workloads to be hosted on such platforms. Many such workloads can be expected to include self-suspending tasks. Moreover, in many settings, such workloads can be expected to have soft timing constraints. The soft timing constraint considered in this paper pertains to implicit-deadline sporadic task systems and requires that deadline tardiness be bounded.

Two analysis approaches can be applied to analyze soft real-time (SRT) sporadic self-suspending (SSS) task systems on multiprocessors. Perhaps the most commonly used is the *suspension-oblivious* approach [3], which simply integrates suspensions into per-task worst-case-execution-time requirements. However, this approach clearly yields $O(n)$ utilization loss where n is the number of self-suspending tasks in the system. The alternative is to explicitly consider suspensions in the task model and the corresponding schedulability analysis; this is known as *suspension-aware* analysis. Although suspension-aware analysis can improve schedulability in many cases compared to the suspension-oblivious approach, existing suspension-aware analysis [2] still entails significant utilization loss.

To analyze SSS task systems more efficiently, we propose a new suspension-aware analysis technique that yields $O(m)$ suspension-related utilization loss, where m is the number of processors. Our technique is derived by identifying and then eliminating the fundamental sources that cause pessimism (i.e., $O(n)$ utilization loss) in previous analysis. Specifically, we derive a schedulability test under the proposed technique showing that any given SSS task system can be supported under global-earliest-deadline-first (GEDF) with bounded tardiness if $U_{sum} + \sum_{j=1}^{m} v^j \leq m$ holds, where U_{sum} is the total system utilization and v^j is the j^{th} maximum ratio of a task's suspension time over its period among tasks in the system.

The rest of this paper is organized as follows. In Sec. 2, we present the SSS task model. Then, in Sec. 3, we identify the fundamental sources causing pessimism in prior analysis and present our new analysis technique and a resulting schedulability test. We conclude in Sec. 4.

2 System Model

We consider the problem of scheduling a set $\tau = \{\tau_1, ..., \tau_n\}$ of n independent SSS tasks on $m \geq 1$ identical processors $\{M_1, M_2, ..., M_m\}$. We assume $n > m$; otherwise we can simply assign each task to one processor. Each task is released repeatedly, with each such invocation called a *job*. Jobs alternate between computation and suspension phases. We assume that each job of τ_l executes for at most e_l time units (across all of its execution phases) and suspends for at most s_l time units (across all of its suspension phases). We place no restrictions on how these phases interleave (a job can even begin or end with a suspension phase). The j^{th} job of τ_l, denoted $\tau_{l,j}$, is released at time $r_{l,j}$ and has a deadline at time $d_{l,j}$. Associated with each task τ_l is a period p_l, which specifies both the minimum time between two consecutive job releases of τ_l and the relative deadline of each such job, i.e., $d_{l,j} = r_{l,j} + p_l$. The utilization of a task τ_l is defined as $u_l = e_l/p_l$, and the utilization of the task system τ as $U_{sum} = \sum_{\tau_i \in \tau} u_i$. An SSS task system τ is said to be an *implicit-deadline* system if $d_i = p_i$ holds for each τ_i.[1] In this

*Work supported by NSF grants CNS 0834270, CNS 0834132, and CNS 1016954; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

[1] τ is said to be a *constrained-deadline* system if, for each task $\tau_i \in \tau$, $d_i \leq p_i$, and an *arbitrary-deadline* system if, for each τ_i, the relation

paper, we consider implicit-deadline SSS task systems.

Successive jobs of the same task are required to execute in sequence. If a job $\tau_{i,j}$ completes at time t, then its *tardiness* is $max(0, t-d_{i,j})$. A task's tardiness is the maximum tardiness of any of its jobs. Note that, when a job of a task misses its deadline, the release time of the next job of that task is not altered. We require that $e_i + s_i \leq p_i$ and $u_i \leq 1$ hold for any task $\tau_i \in \tau$, and that $U_{sum} \leq m$ holds for τ; otherwise, tardiness can grow unboundedly.

Under GEDF, released jobs are prioritized by their absolute deadlines. We assume that ties are broken by task ID (lower IDs are favored).

3 An $O(m)$ Analysis Technique

In this section, we present our proposed $O(m)$ analysis technique and a resulting schedulability test for SRT SSS task systems. We first provide brief summaries of existing analysis approaches and highlight sources of pessimism in them. Efforts to overcome such pessimism will drive the design of the proposed new technique.

There are two existing approaches for dealing with globally-scheduled SRT multiprocessor SSS task systems: the suspension-oblivious approach, denoted SC, which converts all suspensions to computation [3], and a suspension-aware analysis approach presented by Liu and Anderson in [2], denoted LA.

Overview of the SC approach. The SC approach converts all suspensions into computation. After transforming all SSS tasks into ordinary sporadic tasks with only computation, prior SRT schedulability analysis [1] can be applied, resulting a utilization constraint of $U_{sum} + \sum_{i=1}^{n} \frac{s_i}{p_i} \leq m$.

Overview of the LA approach. The LA test is built around the following general strategy, first introduced by Devi and Anderson [1]. First, let $\tau_{l,j}$ be a job of a task τ_l in τ, and S be a GEDF schedule for τ with the following property: the tardiness of every job $\tau_{i,k}$ with priority greater than $\tau_{l,j}$ is at most $x + e_i + s_i$, where $x \geq 0$. Then, determine the smallest x such that the tardiness of $\tau_{l,j}$ is at most $x + e_l + s_l$. This by induction implies a tardiness of at most $x + e_i + s_i$ for all jobs of every task τ_i in τ. The smallest x is determined by computing an upper and a lower bound on the pending work at $d_{l,j}$ for tasks in τ that can compete with $\tau_{l,j}$ after its deadline $d_{l,j}$. Next, we briefly summarize the process of obtaining such upper and lower bounds, and then identify sources causing pessimism in them.

The upper and lower bounds are obtained by comparing the allocations to jobs with priority at least that of $\tau_{l,j}$ in S and a processor share (PS) schedule, both on m processors, and quantifying the difference between the two. The PS schedule is an ideal schedule where each job of each task in τ completes exactly at its deadline. In the PS schedule, each task τ_i executes with a rate equal to u_i in any job execution window $[r_{i,j}, d_{i,j})$, which ensures that each job $\tau_{i,j}$ completes exactly at its deadline. (*Note that suspensions are not considered in the PS schedule*.) A valid PS schedule exists for τ if $U_{sum} \leq m$ holds.

The upper bound on the pending work at $d_{l,j}$ can be obtained by bounding the pending work at time t_n, where t_n is defined to be the end of the latest non-busy interval (i.e., at least one processor is idle at any instant within this interval). This is because the amount of pending work (in comparison to the PS schedule) cannot increase throughout a busy interval (as all processors are busy at any instant within this interval). To bound the pending work at t_n, we have to bound the number of tasks that have enabled tardy jobs at t_n.[2] For ordinary task systems with no self-suspensions, the number of such tasks can be upper bounded by $m-1$, for otherwise t_n would be busy. For SSS task systems, however, all n tasks can have enabled tardy jobs suspending at t_n and t_n can still be non-busy. Since such a worst-case scenario may happen and thus must be considered in the analysis, significant pessimism is incurred in the obtained upper bound.

In lower-bounding the pending work at $d_{l,j}$, we need to bound the least amount of the pending work that executes within $[d_{l,j}, f_{l,j})$, where $f_{l,j}$ is the completion time of our analyzed job $\tau_{l,j}$. For ordinary task systems with no self-suspensions, such a bound is straightforward to obtain because within $[d_{l,j}, f_{l,j})$[3] a non-busy time instant could exist if and only if there are fewer than m tasks that have enabled jobs waiting for execution after $d_{l,j}$. For SSS task systems, unfortunately, idle intervals could exist within $[d_{l,j}, f_{l,j})$ due to suspensions even if at least m tasks have enabled tardy jobs. Thus, to lower bound the pending work, we need to upper bound the idleness that could possibly exist within $[t_{l,j}, f_{l,j})$. The worst-case scenario as mentioned above, where all tasks have multiple tardy jobs suspending simultaneously within $[t_{l,j}, f_{l,j})$, is the main source causing pessimism in the obtained lower bound.

The fundamental cause of pessimism in prior analysis. By the above discussion, we can identify the fundamental source causing pessimism in prior analysis, which is the following worst-case scenario: *all n self-suspending tasks have tardy jobs that suspend at some time t simultaneously, thus causing t to be non-busy; this creates idleness that results in pessimism in the analysis*.

Key observation that motivates this research. Interestingly, the suspension-oblivious approach eliminates the worst-case scenario just discussed, albeit at the expense of pessimism elsewhere in the analysis. That is, by converting all n tasks' suspensions into computation, the worst-case scenario is avoided because then at most $m-1$ tasks can have enabled tardy jobs at any non-busy time instant. *However, converting all n tasks' suspensions to computation is clearly*

between d_i and p_i is not constrained (e.g., $d_i > p_i$ is possible).

[2] Job $\tau_{i,v}$ is *enabled* at t if $r_{i,v} \leq t$, $\tau_{i,v}$ has not completed by t, and its predecessor job $\tau_{i,v-1}$ (if any) has completed by t. Job $\tau_{i,v}$ is *tardy* at t if $d_{i,v} < t$.

[3] Note that all jobs considered here have deadlines no later than $d_{l,j}$ since their priorities are at least that of $\tau_{l,j}$.

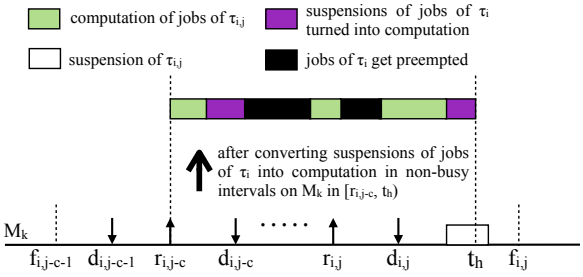


Figure 1: The transformation method.

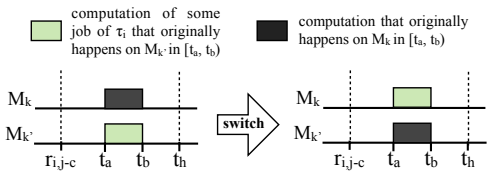


Figure 2: Switching the computation of τ_i originally executed on $M_{k'}$ to M_k.

overkill when attempting to avoid the worst-case scenario; rather, converting at most m tasks' suspensions to computation should suffice. This observation motivates the new analysis technique we propose, which yields a much improved schedulability test with only $O(m)$ suspension-related utilization loss.

New Analysis Technique. We now sketch the new $O(m)$ analysis technique. Motivated by the above discussion, the key idea behind our new technique is the following: *At any non-busy time t, if k processors ($1 \leq k \leq m$) are idle at t while at least k suspending tasks have enabled tardy jobs that suspend simultaneously at t, then, by converting suspensions of k jobs of k such tasks into computation at t, t becomes busy. Converting the suspensions of all such tasks into computation is clearly unnecessary and pessimistic.*

Similar to [2], our analysis draws inspiration from the seminal work of Devi and Anderson [1], and follows the same general framework (which has been described earlier). Due to space constraints, we cannot provide every detail concerning the derivations of the upper and lower bounds. Instead, we focus on explaining how the proposed analysis technique eliminates the worst-case scenario and thus leads to a schedulability test with only $O(m)$ suspension-related utilization loss.

As described earlier, we apply the same proof setup to define our analyzed job $\tau_{l,j}$ and the GEDF schedule S. The part of the schedule S that needs to be analyzed is $[0, f_{l,j})$. We transform this part of the schedule from right to left (i.e., from time $f_{l,j}$ to time 0) to obtain a new schedule $\overline{S}$ as described next. The goal of this transformation is to convert certain tardy jobs' suspensions into computation in non-busy time intervals to eliminate idleness as discussed above. For any job $\tau_{i,k}$, if its suspensions are converted into computation in a time interval $[t_1, t_2)$, then $\tau_{i,k}$ is considered to execute in $[t_1, t_2)$. We transform S to $\overline{S}$ by applying the following

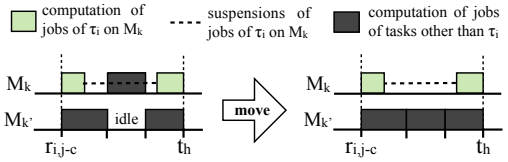


Figure 3: Moving the computation of tasks other than τ_i from M_k to some idle processor $M_{k'}$.

transformation method to each processor in turn (ordered by processor ID). In the following, M_k denotes the current considered processor (initially M_1). For simplicity, we use "S" to denote the updated schedule after each intermediate transformation step (the final transformed schedule $\overline{S}$ is obtained after the whole transformation process completes).

Transformation method. Moving from $f_{l,j}$ to the left in S on M_k, let t_h denote the first encountered non-busy time instant on M_k where at least one task τ_i has an enabled job $\tau_{i,j}$ suspending at t_h where

$$d_{i,j} < t_h. \tag{1}$$

Let $j-c$ ($0 \leq c \leq j-1$) denote the minimum job index of τ_i such that all jobs $\{\tau_{i,j-c}, \tau_{i,j-c+1}, ..., \tau_{i,j}\}$ are tardy, as illustrated in Fig. 1. We assume that all the computation and suspensions of jobs of τ_i occurring within $[r_{i,j-c}, t_h)$ happen on M_k. This can be achieved by switching any computation of τ_i in some interval $[t_a, t_b) \in [r_{i,j-c}, t_h)$ originally executed on some processor $M_{k'}$ other than M_k with the computation (if any) occurring in $[t_a, t_b)$ on M_k, as illustrated in Fig. 2, which is valid from an analysis point of view. Then for all intervals in $[r_{i,j-c}, t_h)$ on M_k where jobs not belonging to τ_i execute while some job of τ_i suspends, if any of such intervals is non-busy (at least one processor is idle in this interval), then we also move the computation occurring within this interval on M_k to some processor $M_{k'}$ that is idle in the same interval, as illustrated in Fig. 3. This step guarantees that all intervals in $[r_{i,j-c}, t_h)$ on M_k where jobs not belonging to τ_i execute are busy on all processors. (Note that after performing the above switching and moving steps, the start and the completion times of jobs remain unchanged.) Due to the fact that all jobs of τ_i enabled in $[r_{i,j-c}, t_h)$ are tardy, interval $[r_{i,j-c}, t_h)$ on M_k consists of three types of subintervals: (*i*) those in which jobs of τ_i are executing, (*ii*) those in which jobs of τ_i are suspending (note that jobs of tasks other than τ_i may also execute on M_k in such subintervals; if this is the case, then note that any such subinterval is busy on M_k), and (*iii*) those in which jobs of τ_i are preempted. Thus, within any non-busy interval on M_k in $[r_{i,j-c}, t_h)$, jobs of τ_i must be suspending (for otherwise this interval would be busy on M_k). Therefore, within all non-busy time intervals on M_k in $[r_{i,j-c}, t_k)$, we convert the suspensions of all jobs of τ_i that are enabled within $[r_{i,j}, t_h)$ (i.e., $\{\tau_{i,j-c}, \tau_{j-c+1}, ..., \tau_{i,j}\}$) into computation, as illustrated in Fig. 1. This transformation guarantees that M_k is busy within $[r_{i,j-c}, t_k)$. Note that when applying the rule on the next processor M_{k+1} (if any),

τ_i clearly cannot be chosen again for the same transformation process (i.e., converting suspensions into computation in idle intervals) within $[r_{i,j-c}, t_h)$. Moreover, since all intervals within $[r_{i,j-c}, t_h)$ on M_k where jobs not belonging to τ_i execute are busy on all processors, any later switch or move does not change the fact that $[r_{i,j-c}, t_h)$ is busy on M_k in the final transformed schedule $\overline{S}$. Next, further moving from $r_{l,j-c}$ to the left in S on M_k, find the next t_h, $\tau_{i,j}$, and $\tau_{i,j-c}$ following the same definitions given above, transform the schedule in the newly defined interval $[r_{i,j-c}, t_h)$ on M_k using the same approach. This process is repeatedly performed on M_k until $t_h = 0$. As mentioned earlier, this process will be applied on each processor in turn, after which we obtain the transformed schedule $\overline{S}$.

Analysis. By transforming S into $\overline{S}$ according to the above rule, we are able to eliminate the worst-case scenario where at least m tasks have enabled tardy jobs suspending at the same non-busy instant, as formerly presented by the following claim.

Claim 1. *At any non-busy time instant $t \in [0, f_{l,j})$ in the transformed schedule $\overline{S}$, at most $m-1$ tasks can have enabled jobs with deadlines before t.*

Proof. For any non-busy time instant $t_a \in [0, f_{l,j})$ in $\overline{S}$, there is at least one processor that is idle at t. Let M_k denote such a processor. Assume more than $m-1$ tasks have enabled jobs at t_a with deadlines before t_a. If m such jobs execute at t_a, then t_a would be busy. Thus, at most $m-1$ such jobs execute and at least one such job is suspending at t_a. Since t_a is non-busy on M_k, by our transformation method, one of the tasks that has an enabled job suspending at t_a with a deadline before t_a would be chosen at t_a such that the suspension of the enabled job of this task at t_a is converted to computation at t_a,[4] which makes t_a busy on M_k, a contradiction. □

Our analysis proceeds by comparing the allocations to jobs with priority at least that of $\tau_{l,j}$ in $\overline{S}$ and the corresponding PS schedule $\overline{PS}$ after the transformation.[5] A crucial step for our analysis to be valid is to show that such a $\overline{PS}$ exists. That is, we need to guarantee that at any time t in $\overline{PS}$, the total utilization of τ is at most m. The following claim provides a necessary condition that can provide such a guarantee.

Claim 2. *If $U_{sum} + \sum_{j=1}^{m} v^j \leq m$, then after the transformation, a PS schedule $\overline{PS}$ exists.*

Proof. Consider any processor M_k. Moving from right to left in $[0, f_{l,j})$ on M_k, whenever we first choose a task τ_i for the transformation process, we use tardy jobs of this same task until $r_{i,j-c}$ (which is the earliest job release time among all tardy jobs of τ_i used in one transformation step). Clearly all these jobs have non-overlapping periods since they belong to the same task. Next, moving further left from $r_{i,j-c}$ in the schedule on M_k to a new time t_h (as defined in the transformation method), if we use some task τ_j other than τ_i for the next transformation step on M_k, then the enabled job of τ_j at t_h must satisfy (1). Since this new t_h occurs before $r_{i,j-c}$, the enabled job of τ_j at t_h must have a deadline before $r_{i,j-c}$. This implies that jobs of τ_j whose suspensions are converted into computation have non-overlapping periods with those jobs of τ_i used for the transformation with respect to M_k. By applying the same reasoning to the rest of the schedule $\overline{S}$ on M_k, it follows that all jobs whose suspensions are converted into computation on M_k do not have overlapping periods. This same reasoning can be applied to all other processors. Since there are m processors and the jobs used for the transformation on each processor do not have overlapping periods, at any time t in $\overline{PS}$, there exist at most m jobs with overlapping periods whose suspensions are converted into computation, which can increase total utilization by at most $\sum_{j=1}^{m} v^j$. This implies that at any time t in $\overline{PS}$, the total utilization of τ is at most $U_{sum} + \sum_{j=1}^{m} v^j$, which is at most m according to the claim statement. □

In order to correctly apply the transformation technique as described above, the expense we have to pay is the potential utilization loss of at most $\sum_{j=1}^{m} v^j$ due to the conversion of any m tasks' suspensions into computation. Thus, any SRT SSS task system that can accommodate this expense can be proved to be GEDF-schedulable, which is formerly described by the following schedulability test.

Theorem 1. *Any SRT SSS task system τ is GEDF-schedulable with bound tardiness on m processors if $U_{sum} + \sum_{i=1}^{m} v^j \leq m$.*

4 Summary

In this paper, we presented a new multiprocessor schedulability analysis technique for globally-scheduled SRT SSS task systems. By identifying and eliminating the sources causing pessimism in prior analysis, our proposed analysis technique achieves a much improved schedulability test with only $O(m)$ suspension-related utilization loss. Given that m is often small in practice (typically two, four, or eight cores per chip), this technique is significant and is applicable to real systems.

References

[1] U. Devi. Soft real-time scheduling on multiprocessors. In *Ph.D. Dissertation, UNC Chapel Hill*, 2006.

[2] C. Liu and J. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Proc. of the 30th RTSS*, pp. 425-436, 2009.

[3] J. Liu. *Real-time systems*. Prentice Hall, 2000.

[4] F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Proc. of the 25th RTSS*, pp. 47-56, 2004.

[4] If there are h processors that idle at t_a, then at least h such tasks have enabled jobs suspending at t_a, and hence each processor is guaranteed to have one task available at t_a for the transformation

[5] Note that in $\overline{PS}$, any job of any task still completes exactly at its deadline. To ensure this, each task τ_i executes with a rate between u_i and $u_i + \overline{s}_{i,j}/p_i$ in any job execution window $[r_{i,j}, d_{i,j})$, where $\overline{s}_{i,j} \leq s_i$ is the amount of suspension time of job $\tau_{i,j}$ that is converted into computation in $\overline{S}$.

Towards an analysis framework for tasks with probabilistic execution times and probabilistic inter-arrival times

Dorin Maxim
INRIA Nancy Grand Est
615 rue du Jardin Botanique
54600 Villers les Nancy
dorin.maxim@inria.fr

Liliana Cucu-Grosjean
INRIA Nancy Grand Est
615 rue du Jardin Botanique
54600 Villers les Nancy
liliana.cucu@inria.fr

***Abstract*—In this paper we investigate the problem of calculating the response time distribution for real-time tasks with probabilistic worst-case execution times, probabilistic inter-arrival times and probabilistic deadlines. We propose a definition for the probabilistic deadlines and a first discussion on the response time calculation.**

***Index Terms*—probabilistic real time, probabilistic execution time, probabilistic inter-arrival times, probabilistic deadlines**

I. Introduction

In embedded real-time systems there is a strong demand for new functionality that can only be met by using advanced high performance microprocessors. Building real-time systems with reliable timing behavior on such platforms represents a considerable challenge. Deterministic analysis for these platforms may lead to significant over-provision in the system architecture, effectively placing an unnecessary low limit on the amount of new functionality that can be included in a given system. An alternative approach is to use probabilistic analysis. Probabilistic analysis techniques rather than attempting to provide an absolute guarantee of meeting the deadlines, provide the probability of meeting the deadlines.

In this paper we investigate the problem of calculating the response time distribution for real-time tasks with probabilistic worst-case execution times, probabilistic inter-arrival times and probabilistic deadlines. The scheduling policy is a preemptive fixed-priority one and it is considered as given. The tasks are scheduled on one processor.

In this paper we propose a definition for the probabilistic deadlines and a first discussion on the response time calculation for a task. As future work we leave the proposition of a general formulation for n tasks and the associated proof.

II. Model and notations

A. Model

In this paper, we consider a task set of n synchronous tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i is characterized by three parameters $(\mathcal{C}_i, \mathcal{T}_i, \mathcal{D}_i)$ where $\mathcal{T}_i$ is the minimal inter-arrival time (commonly known as period), $\mathcal{D}_i$ the relative deadline (to be defined in Section II-B), and $\mathcal{C}_i$ the worst-case execution time. The parameters are described by random variables[1].

A random variable $\mathcal{X}_i$ describing a parameter of τ_i is assumed to have a known probability function (PF) $f_{\mathcal{X}_i}(\cdot)$ with $f_{\mathcal{X}_i}(x) = P(\mathcal{X}_i = x)$ giving the probability that the respective parameter of τ_i is equal to x. The values of $\mathcal{X}_i$ are assumed to belong to the interval $[x_i^{\min}, x_i^{\max}]$.

For instance the worst-case execution time $\mathcal{C}_i$ can be written as follows:

$$\mathcal{C}_i = \begin{pmatrix} C_i^0 = C_i^{\min} & C_i^1 & \cdots & C_i^{k_i} = C_i^{\max} \\ f_{\mathcal{C}_i}(C_i^{\min}) & f_{\mathcal{C}_i}(C_i^1) & \cdots & f_{\mathcal{C}_i}(C_i^{\max}) \end{pmatrix}, \quad (1)$$

where $\sum_{j=0}^{k_i} f_{\mathcal{C}_i}(C_i^j) = 1$.

For example for a task τ_i we might have a worst-case execution time $\mathcal{C}_i = \begin{pmatrix} 2 & 3 & 25 \\ 0.5 & 0.45 & 0.05 \end{pmatrix}$; thus $f_{C_i}(2) = 0.5$, $f_{C_i}(3) = 0.45$ and $f_{C_i}(25) = 0.05$.

Each task τ_i generates an infinite number of successive jobs $\tau_{i,j}$, with $j = 1, \ldots, \infty$. All jobs are assumed to be independent of other jobs of the same task and those of other tasks, hence the execution time of a job does not depend on, and is not correlated with, the execution time of any previous job.

The set of tasks is scheduled according to a preemptive fixed-priority policy, i.e., all jobs of the same task have the same priority.

B. Deadline

In this section we provide an answer to the question "how do we define the deadline of a task with probabilistic periods?".

Given a task set with one task, τ, with probabilistic period described by the random variable $\mathcal{T} = \begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix}$, we show that its probabilistic deadline should have the same distribution as the period of the same task.

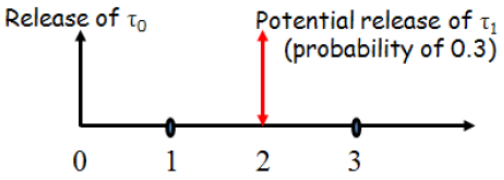
We analyze the possible scenarios and extract the corresponding probabilities.

[1]In this paper we will use a calligraphic typeface to denote random variables.

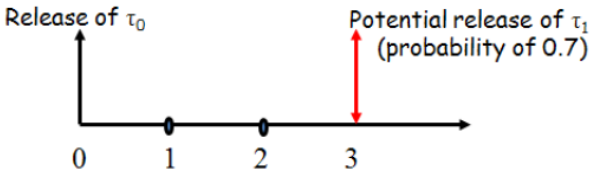
For the first job τ_0, released at $t = 0$, we have two possible scenarios:

Scenario 1: a new job τ_1 will be released at $t = 2$. This moment becomes the deadline of τ_0. The probability associated to this scenario is 0.3. This scenario is depicted in Figure 1a.

Scenario 2: if τ_1 does not arrive at $t = 2$ but it arrives at $t = 3$, then this is considered to be the deadline of τ_0. The probability associated to this scenario is 0.7. This scenario is depicted in Figure 1b.



(a) Scenario 1



(b) Scenario 2

Fig. 1: The two possible release scenarios of τ_1

Combining the two scenarios we obtain a distribution of the deadline of the first job equal to $\mathcal{D}_0 = \begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix} = \mathcal{T}$, which was expected and somewhat obvious.

Let us now analyze what happens in the case of the second job of τ.

For the second job there are four release scenarios, two for each scenario of the previous job.

Scenario 1, i.e. τ_1 arrived at $t = 2$, has two possible continuations: τ_2 can arrive either at $t = 4$ or at $t = 5$, i.e. 2, respectively 3 units of time after the release of τ_1. Subtracting the two already passed units of time we obtain a relative deadline of $\begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix}$. *

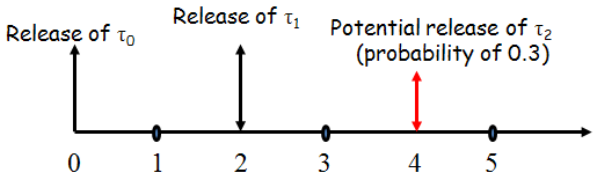
These two possibilities are depicted in Figure 2.

Scenario 2, i.e. τ_1 arrived at $t = 3$, has two possible continuations: τ_2 can arrive either at $t = 5$ or at $t = 6$, i.e. 2, respectively 3 units of after the release of τ_1. Subtracting the three already passed units of time we obtain a relative deadline of $\begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix}$. **

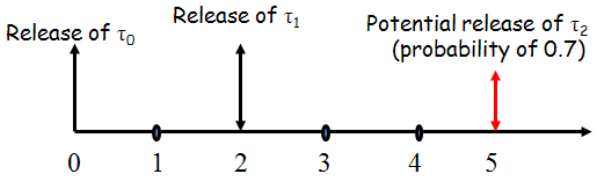
These two possibilities are depicted in Figure 3.

From * and ** we conclude that the relative deadline of τ_1 is $\mathcal{D}_1 = \begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix} = \mathcal{T}$.

Continuing this reasoning, we obtain that the relative deadlines of all jobs of a task with probabilistic period have the same probability distribution as the period.

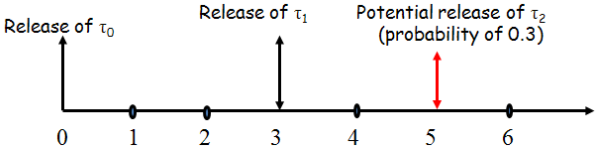


(a) τ_2 has a probability of 0.3 to be released at $t = 4$

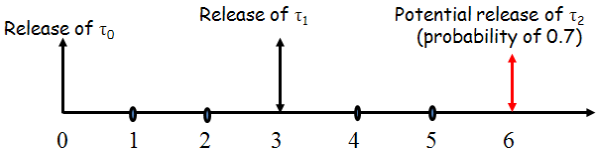


(b) τ_2 has a probability of 0.7 to be released at $t = 5$

Fig. 2: Scenario 1 continued with its two sub-scenarios



(a) τ_2 has a probability of 0.3 to be released at $t = 5$



(b) τ_2 has a probability of 0.7 to be released at $t = 6$

Fig. 3: Scenario 2 continued with its two sub-scenarios

III. Solution for a single task

A. The case of a single task - the step by step approach

Given a task $\tau = \left(\begin{pmatrix} 2 & 3 \\ 0.8 & 0.2 \end{pmatrix}, \begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix} \right)$, where $\begin{pmatrix} 2 & 3 \\ 0.8 & 0.2 \end{pmatrix}$ is the distribution of its probabilistic execution time and $\begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix}$ is the distribution of its probabilistic period, one has to compute the response time distributions of its jobs.

We also know about the task that its first job τ_1 is released at $t = 0$ and that its deadline is implicit, i.e., the release of a job determines the deadline of the previous job.

1) Response time and deadline miss probabilities: **For the first job** τ_0, which is released at $t = 0$, its response time has a distribution equal to the distribution of the execution time.

In order to obtain the probability of τ_0 missing its deadline, we analyse the two scenarios given by the deadline.

In the first scenario, when τ_1 is released at $t = 2$ we have two possibilities:

If its execution time is $C = 2$, then τ_0 reaches its deadline. The probability of this happening is $0.3 \times 0.8 = 0.24$, i.e.,

the probability of the job having an execution time of 2 and a deadline of 2.

If its execution time is $C = 3$, then τ_0 misses its deadline. The probability of this happening is $0.3 \times 0.2 = 0.06$, i.e., the probability of the job having an execution time of 3 and a deadline of 2.

In the second scenario, when the deadline is equal to 3, τ_0 reaches its deadline regardless if it has an execution time of 2 or of 3. This two sub-scenarios summed have a probability of $0.7 \times 0.8 + 0.7 \times 0.2 = 0.7$.

Combining the two scenarios, we obtain that τ_0 has a 0.06 probability of missing its deadline and a 0.94 probability of finishing before the next release, i.e. reaching its deadline.

For the second job there are the following possible scenarios:

Scenario 1: τ_1 arrives at $t = 2$, has a probability of 0.3 of happening. There are two possibilities:

a) τ_0 finishes execution at $t = 2$. This has a 0.8 probability of happening. In this case there is no backlog and there are once more two possible outcomes: τ_4 arrives at $t = 4$, in which case τ_1 reaches its deadline if it has an execution time of 2 or **it misses its deadline** if it has an execution time of 3. The probability of having an execution time of 3 is 0.2 which gives a probability of τ_1 missing its deadline equal to $0.3 \times 0.8 \times 0.3 \times 0.2 = 0.0144$, i.e., the multiplication of, respectively, the probability that τ_1 arrives at $t = 2$, the probability that τ_0 has an execution time of 2, the probability that τ_2 arrives at $t = 4$ and the probability that τ_1 has an execution time of 3.

b) τ_0 finishes execution at $t = 3$. There are here multiple possibilities:

If τ_1 has an execution time of 2 (0.8 probability) and τ_2 arrives at $t = 5$ (probability 0.7) then τ_1 finishes its execution before its deadline. The probability of this happening is $0.3 \times 0.8 \times 0.2 \times 0.7 = 0.0294$.

If τ_1 has an execution time of 2 (0.8 probability) and τ_2 arrives at $t = 4$ (probability 0.3) then τ_1 **misses its deadline**. The probability of this happening is $0.3 \times 0.8 \times 0.2 \times 0.3 = 0.0144$.

If τ_1 has an execution time of 3, then it misses its deadline no mater if τ_2 arrives at $t = 4$ or at $t = 5$. This has a probability of happening of $0.06 \times 0.2 = 0.012$

Summing up all the deadline miss probabilities obtained for Scenario 1, we get a partial probability of $0.0144 + 0.0144 + 0.012 = 0.0408$ that τ_1 misses its deadline.

Scenario 2: τ_1 arrives at $t = 3$. In this scenario there is no backlog, which means that everything happens as for the first release (at $t = 0$) just that the probabilities are multiplied by 0.7.

We get that τ_1 has a $0.06 \times 0.7 = 0.042$ probability of missing its deadline and a $0.7 \times 0.94 = 0.658$ probability of reaching it.

Combining the two scenarios we obtain that τ_1 has a total probability of $0.0408 + 0.042 = 0.0828$ of missing its deadline.

2) Discussion: The value obtained for the deadline miss probability of a job is an upper bound and not and exact value, since it is the summation of two probabilities, the ones resulted in the two scenarios.

One could argue towards not summing the two scenarios since this gives pessimistic results. Another option would be to do the average of the resulting scenarios. this is not a valid option though, since it can produce optimistic results. For example, by doing the average of the two scenarios obtained in the previous example, we would get a value of $0,0414$ which is less than the probability computed for the first scenario (0.0444). One could argue that this second value is the real and the only value one should take into account.

Keeping the probabilities for each possible scenario separated has its own drawbacks. The first and obvious one is the fact that the number of scenarios could be very big, and not to mention storing them, it can be very complex to work with so many values.

Another drawback is, one could argue, that each scenario in itself can be optimistic. For example, for the second job (τ_1), second scenario, one could argue that the respective probabilities are exactly the ones obtained for the first job (τ_0), and not multiplied with 0.7 as we did in computing the probabilities of the scenario. This would mean that its deadline miss probability in the scenario would be 0.06, which is greater than 0.042. Fortunately, the upper bound (summation of all the scenarios) covers the value 0.06, so it is a safe (pessimistic) bound.

B. The case of a single task - The analytical approach

We recall here the task system that is under analysis. This task system has only one task characterized by:
$\tau = \left(\begin{pmatrix} 2 & 3 \\ 0.8 & 0.2 \end{pmatrix}, \begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix} \right)$,
where $\begin{pmatrix} 2 & 3 \\ 0.8 & 0.2 \end{pmatrix}$ is the distribution of its probabilistic execution time and $\begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix}$ is the distribution of its probabilistic period.

We also know about this task that its first release occurs at $t = 0$, i.e., $r_0 = 0$. This, in turn, implies that there is no backlog at the moment of its arrival, which we denote with $\mathcal{B}_0 = 0$, backlog at the arrival of job τ_0.

The release distributions of subsequent jobs can be computed using the formula $r_i = r_{i-1} \otimes \mathcal{T}$. We would have $r_1 = \begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix} = r_0 \otimes \mathcal{T}$ and $r_2 = \begin{pmatrix} 6 & 5 & 4 \\ 0.39 & 0.42 & 0.09 \end{pmatrix} = r_1 \otimes \mathcal{T}$.

The response time of the first job is $\mathcal{R}_0 = \begin{pmatrix} 2 & 3 \\ 0.8 & 0.2 \end{pmatrix}$.

Knowing that the deadline of τ_0 is $\mathcal{D} = \begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix}$, we can compute the backlog at the release of the next instance of τ and also the deadline miss probability of τ_0, by using an operation similar to the convolution, which, instead of doing addition of the values, does subtraction. The probabilities are multiplied the same as in a convolution. We call this operation a "subtracting convolution".

For example, the backlog at the release of τ_2 can be computed as:

$\mathcal{B}_1 = \mathcal{D} \ominus \mathcal{R}_0 = \begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix} \ominus \begin{pmatrix} 2 & 3 \\ 0.8 & 0.2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0.56 & 0.24 & 0.14 & 0.06 \end{pmatrix}$.

By gathering all the non-negative values in zero, we obtain $\mathcal{B}_1 = \begin{pmatrix} 0 & -1 \\ 0.94 & 0.06 \end{pmatrix}$. This random variable implies that there is a 94% chance that there will be 0 backlog at the arrival of τ_1 and a 6% chance that 1 more unit of time is necessary for τ_0 to finish its execution.

In the distribution of the backlog, the deadline miss probability of τ_0 is the (summed) probability corresponding to the negative values of the distribution, in this case 0.06, the probability corresponding to -1.

By using the backlog and the execution time distributions, the response time distribution of the next instance of the task can be computed using the formula
$\mathcal{R}_i = |\mathcal{B}_i| \otimes \mathcal{C}$.

The response time of τ_0 is, indeed, $\mathcal{R}_0 = |\mathcal{B}_0| \otimes \mathcal{C} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 2 & 3 \\ 0.8 & 0.2 \end{pmatrix} = \begin{pmatrix} 2 & 3 \\ 0.8 & 0.2 \end{pmatrix}$.

For τ_1 we have that $\mathcal{R}_1 = |\mathcal{B}_1| \otimes \mathcal{C} = |\begin{pmatrix} 0 & -1 \\ 0.94 & 0.06 \end{pmatrix}| \otimes \begin{pmatrix} 2 & 3 \\ 0.8 & 0.2 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 4 \\ 0.752 & 0.236 & 0.012 \end{pmatrix}$.

The response time distribution can be compared with the relative deadline, $\mathcal{D} = \begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix}$, in order to see which are the combinations that would lead to a deadline miss, i.e., the situations when the response time is (strictly) larger than the deadline. Those situations are given by the following combinations:

- when the response time is 3 and the deadline is 2, which has a probability of $0.3 \times 0.236 = 0.0708$;
- when the response time is 4 and the deadline is 2, which has a probability of $0.7 \times 0.012 = 0.0084$;
- and also when the response time is 4 and the deadline is 3, which has a probability of $0.012 \times 0.3 = 0.0036$.

Summing up the above probabilities, we obtain that τ_1 has a probability to miss its deadline equal to $0.0708 + 0.0084 + 0.0036 = 0.0828$, which is exactly what we got from the step-by-step verification in the previous section.

By computing the backlog with the formula $\mathcal{B}_i = \mathcal{D} \ominus \mathcal{R}_{i-1}$ we obtain that $\mathcal{B}_2 = \mathcal{D} \ominus \mathcal{R}_1 = \begin{pmatrix} 3 & 2 \\ 0.7 & 0.3 \end{pmatrix} \ominus \begin{pmatrix} 2 & 3 & 4 \\ 0.752 & 0.236 & 0.012 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & -1 & -1 & -2 \\ 0.5264 & 0.2256 & 0.1652 & 0.0708 & 0.0084 & 0.0036 \end{pmatrix}$

After gathering all the non-negative values in zero, we obtain that the backlog at the release of τ_2 is $\mathcal{B}_2 = \begin{pmatrix} 0 & -1 & -2 \\ 0.9172 & 0.0792 & 0.0036 \end{pmatrix}$, where the probabilities of the negative values give the deadline miss probability DMP of τ_1, i.e., $DMP_1 = 0.0792 + 0.0036 = 0.0828$, which is, once more, exactly the value obtained trough the step-by-step verification in the previous section.

In conclusion, we have that the backlog at the release of a job can be computed using the formula

$\mathcal{B}_i = \mathcal{D} \ominus \mathcal{R}_{i-1}$, with $\mathcal{B}_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$,

and the response time can be computed using the formula

$\mathcal{R}_i = |\mathcal{B}_i| \otimes \mathcal{C}$, where $|\mathcal{B}_i|$ is the modulo of the backlog, meaning that each value of the backlog is taken in the positive.

This way we may compute for example that the response time of τ_2 is
$\mathcal{R}_2 = |\mathcal{B}_2| \otimes \mathcal{C} = \begin{pmatrix} 2 & 3 & 4 & 5 \\ 0.73376 & 0.2468 & 0.01872 & 0.00072 \end{pmatrix}$
We can compute also the backlog at the release of τ_3, which is
$\mathcal{B}_3 = \mathcal{D} \ominus \mathcal{R}_2 = \begin{pmatrix} 0 & -1 & -2 & -3 \\ 0.90652 & 0.087144 & 0.00612 & 0.000216 \end{pmatrix}$,
which means that τ_2 has a probability of missing its deadline of $0.087144 + 0.00612 + 0.000216 = 0.09348$ and a probability of 0.90652 to finish execution in time.

IV. Related work

There has been a significant work devoted to probabilistic real-time analysis the last years. However, and to our best knowledge, there are no comparable results so far. In fact, there are a few related works which consider special scheduling models providing isolation between tasks [1], or assuming a known (a priori) maximum number of arrivals, thus introducing an unnecessary level of pessimism in the analysis [2], [3].

For the problem where the worst-case execution times are probabilistic the approach in [4] is the most general. However, to our best knowledge their approach is not extended to the case of random inter-arrival times.

The closest work to our contribution is described in [5], but the results are only valid for particular cases of random variables.

V. Conclusion

In this paper we propose a definition for the probabilistic deadlines and a first discussion on the response time calculation for a task. As future work we leave the proposition of a general formulation for n tasks and the associated proof.

References

[1] L. Abeni and Buttazzo, "QoS guarantee using probabilistic deadlines," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS99)*, 1999.

[2] A. Burns, G. Bernat, and I. Broster, "A probabilistic framework for schedulability analysis," in *Third International Embedded Software Conference (EMSOFT03)*, 2003, pp. 1–15.

[3] I. Broster and A. Burns, "Applying random arrival models to fixed priority analysis," in *Proceedings of the Work-In-Progress of the 25th of the IEEE Real-Time Systems Symposium (RTSS04)*, 2004.

[4] J. Díaz, D. Garcia, K. Kim, C. Lee, L. Bello, L. J.M., and O. Mirabella, "Stochastic analysis of periodic real-time systems," in *23rd of the IEEE Real-Time Systems Symposium (RTSS02)*, 2002, pp. 289–300.

[5] J. Lehoczky, "Real-time queueing theory," in *10th of the IEEE Real-Time Systems Symposium (RTSS96)*, 1996, pp. 186–195.

Traffic Shaping to Reduce Jitter in Controller Area Network (CAN)

Robert I. Davis
Real-Time Systems Research Group, Department of Computer Science,
University of York, YO10 5DD, York, UK
rob.davis@cs.york.ac.uk

Nicolas Navet
INRIA / RTaW
615, rue du Jardin Botanique
54600 Villers-lès-Nancy (France)
nicolas.navet@inria.fr

Abstract— **When a message is transferred from one CAN bus to another via a gateway, variability in the response time of the message on the source network typically translates into queuing jitter on the destination network. This jitter inheritance accumulates across each gateway and can significantly impact the schedulability of lower priority messages. In this paper, we show that the real-time performance of the network can be enhanced by a simple method of traffic shaping that eliminates this inherited queuing jitter. This method does not require access to global time, nor does it require precise time-stamping of when messages are received at the gateway or blocking read calls. It can also be extended to account for clock drifts between networks.**

Keywords-Controller Area Network (CAN); traffic shaping; jitter; response time analysis; scheduling.

I. INTRODUCTION

In automotive applications, Controller Area Network (CAN) [1], [4] is typically used to provide high speed networks (500Kbits/s) connecting chassis and power-train Electronic Control Units (ECUs), for example engine management and transmission control. It is also used for low speed networks connecting body and comfort electronics. Data required by nodes on different networks is transferred between different CAN buses by a gateway connected to both.

Schedulability analysis for CAN [3] computes upper bounds on the worst-case response times of messages on a single network, and thus can be used to provide guarantees that messages will meet their deadlines during normal operation. When messages are transferred from one network to another by a gateway, then holistic analysis [5] can be used to determine the overall end-to-end response time of each message, and thus determine if end-to-end deadlines will be met.

If the gateway forwards messages for transmission on the destination network as soon as they have been received from the source network then this can result in those messages exhibiting significant queuing jitter on the destination network. Effectively all of the variability in the message's response time on the source network can manifest itself as queuing jitter with respect to transmission of the message on the destination network. In the worst-case, multiple instances of the same message that were originally queued periodically with only a small amount of jitter on the source network, may end up being transmitted back-to-back on a destination network, causing increased delays to lower priority messages.

Holistic analysis [5] assumes that messages inherit all of the response time up to their reception at a gateway plus the maximum delay in being processed by the gateway as queuing jitter on the destination network. This analysis can be improved by considering the time at which messages are queued on the destination network as a dynamic offset [8] which can vary between some minimum and maximum values. This model eliminates jitter equivalent to the best-case response time. Techniques for reducing variability in the length of messages caused by stuff-bits can also eliminate some jitter [12], [13]; however, the difference between best-case and worst-case response times can still be large and so significant jitter remains. The use of offset release times [11] between messages sent by the same node can reduce worst-case response times on the source network, again reducing but not eliminating queuing jitter on the destination network.

The problem is that gatewayed messages with large queuing jitter can have a significant impact on the schedulability of lower priority messages on the destination network. With two networks, the interference from gatewayed messages can easily be doubled, with two instances of a gatewayed message being sent during the response time of a lower priority message, rather than just one. Response time upper bounds for non-pre-emptive scheduling highlight this effect – see equation (33) in [9].

In this short paper, we introduce a simple traffic shaping technique that can be used in gateways to eliminate all of the jitter due to variability in message response times up to the point at which they are received by the gateway. This technique builds upon the No Global Time (NGT) method [2]. Thus it does not require the use of global time, and the source and destination networks are assumed to be unsynchronised. However, unlike NGT, it does not require information about when each message was received from the source network, nor does it assume the use of a blocking read call, necessitating the use of a separate task per gatewayed message. Instead, this technique assumes only that there is a free-running timer that may be read by a single periodic communications task in the gateway. The technique proposed is related to the *leaky bucket* / *token bucket* method [6], [10], [15] of traffic shaping. In particular, it is a greedy method of traffic shaping [14] which comes for free, in the sense that it does not introduce any additional end-to-end delays. It also has some similarities to the use of servers to shape flows on Ethernet described in [17].

Other methods of reducing jitter include synchronising networks and then using offsets to determine message release times on the destination network [16]. The work on FTT-CAN [7] is an example of this approach; however,

requiring synchronisation and effectively using TDMA on top of CAN's priority based arbitration has the disadvantage of adding complexity and overheads. De-coupling source and destination networks via periodic message activation on the destination network is another way of reducing jitter [19]; however, unlike the approach taken in this paper, this reduction in jitter comes at a cost of significantly increased end-to-end latencies.

Due to space constraints, we do not describe the CAN protocol or its schedulability analysis in this paper. The interested reader is referred instead to [3].

II. GATEWAYS AND TRAFFIC SHAPING

In automotive applications, there is often a communications task that is responsible for the forwarding of gatewayed messages. Let Δ be the maximum delay between an event occurring (e.g. a message instance being received) and the communications task recognising it. Typically, Δ corresponds to the period (T_{COM}) plus worst-case response time (R_{COM}) of the communications task; assuming that it can execute as early as possible in one period and then as late as possible in the next.

With a simple immediate forwarding policy, each time the communications task runs, it checks for any received messages from the source network and queues the corresponding message on the destination network. Here, as described in [5], the message m on the destination network inherits queuing jitter from both the message on the source network, and the communications task:

$$J_m^{DEST} = J_m^{SRC} + R_m^{SRC} + \Delta \quad (1)$$

Where J_m^{SRC} and J_m^{DEST} are respectively the queuing jitter on the source and destination networks, and R_m^{SRC} is the worst-case response time of the message on the source network. (Note, here we assume that R_m^{SRC} is made up of the queuing delay and transmission time of message m on the source network, but not its queuing jitter J_m^{SRC} which is included separately). The inherited jitter J_m^{DEST} impacts negatively schedulability on the destination network. It is therefore interesting to consider ways in which this inherited jitter can be eliminated. We note that in practice not all of the response time on the source network contributes to jitter, only the variability between the worst-case and the best-case response time; however, we use the simpler model here.

A. Jitter reduction policies

The No Global Time (NGT) policy introduced in [2] removes jitter, without the need for global time or clock synchronisation between different nodes.

With the NGT policy (see Algorithm 1), it is assumed that the `read` is a blocking call that waits until the next instance of the message is available from the source network, if it has not already been received. The time t is the local time at which the message instance was received, `period` is the message period, and `next` is the earliest time at which the next message may be queued onto the destination network.

The effect of the NGT policy is to enforce a minimum delay of `period` between the queuing of instances of the message on the destination network, thus eliminating jitter due to variability in the message response time on the source network.

```
1 next = 0
2 read msg from source network
3 returning the time t at which it was
4 received
5 loop
6     queue output to destination network
7     next = max(next, t) + period
8     delay_until next
9     read msg from source network
10    returning t
11 end loop
```

Algorithm 1: NGT Policy for message m

One of the drawbacks of the NGT policy is that it requires the time at which each message instance was received to be available. This is not necessarily possible with all CAN controller hardware. A second drawback is that the `read` call is assumed to be blocking, and so requires a separate task per forwarded message.

We now adapt the NGT policy making it more suited to gateway nodes connecting networks using CAN. We refer to the new policy as NJR for Non-blocking Jitter Reduction. With the NJR policy, instead of a blocking read call, we assume a single communications task that executes periodically and has a maximum delay of Δ as explained earlier. We do not require precise recording of the times at which message instances are received from the source network, instead this will be done approximately by the communications task. We assume that instances of gatewayed message m are placed in a FIFO buffer by the CAN controller or interrupt handler, ready for processing by the communications task.

In the following, we assume that the event initiating the queuing of message m on the source network occurs sporadically with a minimum inter-arrival time of T_m referred to as the message period. (We assume that the maximum delay Δ of the communications task is less than the message period T_m).

```
1 if the message m FIFO buffer is empty
2     return
3 get local time t
4 if t < X_m // too early to process message m
5     return
6 get instance of message m from buffer
7 queue instance on destination network
8 X_m = max( X_m , t – Δ ) + T_m
```

Algorithm 2: NJR Policy for message m

Algorithm 2 illustrates the policy implemented in the body of the periodic communications task for instances of message m. X_m represents the next time at which it is permissible to queue an instance of message m on the destination network. X_m is assumed to be initialised to zero, and the local incrementing free-running timer t started at zero, before the task first runs. (Note that the same communications task can deal with forwarding multiple messages).

The NJR policy operates as follows. When the first instance of message m arrives, it can be forwarded immediately, and is queued on the destination network by the communications task as soon as the task executes. As the task has a maximum delay of Δ in detecting that there is a message instance to process, the policy assumes that all of this delay may have happened, and that it is therefore permissible to queue the next instance of the message on the destination network at a time $X_m = t - \Delta + T_m$ or later. If one or more subsequent instances of message m arrive before this time, then they will wait in the FIFO buffer for later processing, with queuing of the kth subsequent instance of the message now only permitted at or after time $X_m + kT_m$.

The remaining behaviour of the NJR policy, and the fact that reading and using the local time obtained on line 3 is sufficient to correctly determine the next permissible queuing time, can be seen by considering what happens when the communications task runs and *first* finds an instance j of message m at the head of the FIFO buffer. There are two cases to consider:

Case 1: The local time $t < X_m$. In this case, we can be certain that the time at which instance j was received is not relevant for setting the earliest permitted queuing time for the next instance (j+1). Instance j will be processed by a subsequent invocation of the communications task that obtains a local time t: $X_m \le t \le X_m + \Delta$. On that invocation, line 8 will set $X_m = X_m + T_m$, without needing a value of t that reflects when instance j was actually received.

Case 2: The local time $t \ge X_m$. In this case, instance j cannot have been in the FIFO buffer when the communications task previously ran; otherwise we would have Case 1. (Trivially, it cannot have been at the head of the buffer. Further, it cannot have been behind any previous instances in the buffer, because as $\Delta < T_m$, X_m advances by more than Δ for each instance of message m processed, and so any instance that is received while a previous instance is in the buffer must belong to Case 1). Instance j must therefore have been received at some time between $t - \Delta$ and t, and so t is a valid time to use in computing the earliest permissible queuing time for instance j+1, via line 8.

The effect of the NJR policy is to ensure that the period or minimum inter-arrival time of message m on the destination network is T_m and its queuing jitter is $J_m^{DEST} = \Delta + R_{COM}$. Note, the additional R_{COM} term arises because there could be a delay of at most R_{COM} between the timer being read (i.e. time t on line 3 of Algorithm 2) and the message being queued (line 7), yet the next instance of the message may be queued at time $t + T_m - \Delta$. The longest possible delay between the event triggering the sending of an instance of message m on the source network and its processing by the communications task being permitted is $J_k^{SRC} + R_k^{SRC}$ (i.e. the same as the maximum delay between the initiating event and the reception of the corresponding message instance at the gateway). This is the case because processing of an instance of message m can only ever be delayed if it arrives less than kT_m after the kth previous instance. Hence processing of message instances with the maximum delay $J_k^{SRC} + R_k^{SRC}$ is unconstrained by the NJR policy, and processing of instances which are received after a delay of $J_k^{SRC} + R_k^{SRC} - y$ is only constrained (not permitted) for at most an interval of time y. The NJR policy therefore adds nothing to the worst-case delay with which gatewayed messages are queued onto the destination network. Recall that with immediate forwarding, such messages need to be considered as having jitter of $J_m^{DEST} = J_m^{SRC} + R_m^{SRC} + \Delta$. Instead with the NJR Policy, they can, in the worst-case, be considered as having been subject to a fixed delay of $J_m^{SRC} + R_m^{SRC} - R_{COM}$, before being queued with a period of T_m and a queuing jitter of $J_m^{DEST} = \Delta + R_{COM}$. We note that while there may be little if any difference in the overall worst-case end-to-end response time for a single gatewayed message, compared to an immediate forwarding policy, there can be significant differences in the response times of lower priority messages, including other gatewayed messages. This is because the elimination of the majority of the queuing jitter reduces the number of instances of messages that can be queued onto the destination network in a short interval of time.

B. Jitter reduction policy experiments

We conducted some simple experiments to demonstrate the jitter reduction that occurs with the NJR policy.

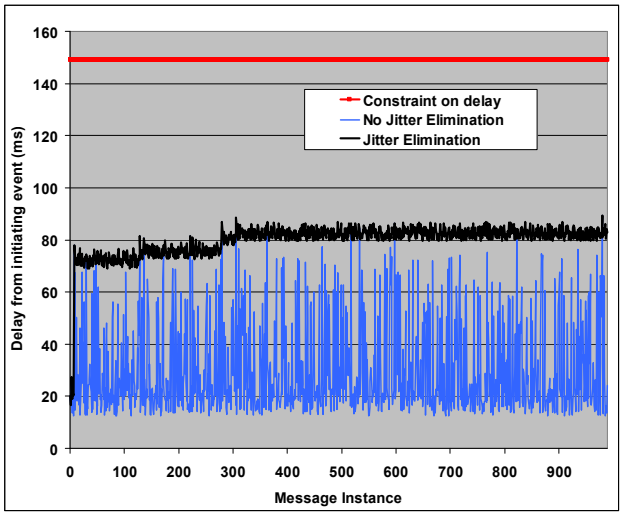


Figure 1: Overall delay in queuing messages onto the destination network

The results of one of these experiments is shown in Figure 1. This experiment is based on a typical 125Kbit/s body electronics network configuration generated by NETCARBENCH [18] with 145 messages. We selected the lowest priority message with a period of 200ms, as an example of a gatewayed message. This message has a worst-case response time of just under 140ms. The period (T_{COM}) of the communications task was set to 6ms and its worst-case response time (R_{COM}) to 3ms, giving a value for Δ of 9ms.

The bottom line in Figure 1 shows the delay from the initiating event for each message instance to the time at which it was queued on the destination network, assuming

an immediate forwarding policy. The middle line shows the same delay using the NJR policy. The top line shows the bound on this time (147.5ms in this case). It is notable that the delay is far more consistent with the NJR policy.

It is noticeable in Figure 1 that there is a transient behaviour for the first 300 or so message instances, this happens as the observed delay since the initiating event increases up to a maximum value. The small overshoot evident at around 300 is due to using $t + T_m - \Delta$ as the next permitted queuing time when the communications task has not actually exhibited a delay as long as Δ.

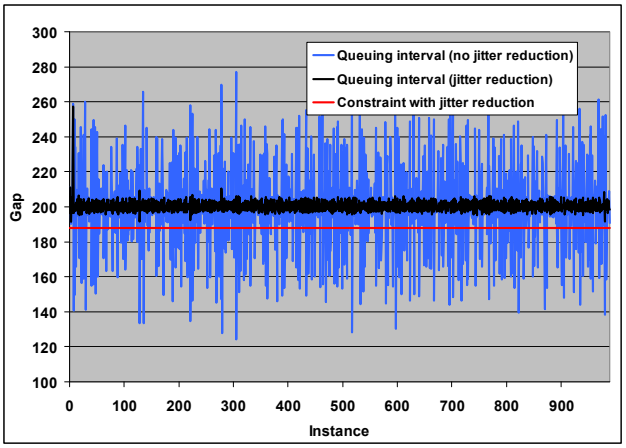


Figure 2: Time between queuing message instances (with / without jitter reduction)

Figure 2 shows the time interval between queuing two instances of the same message on the destination network with and without jitter reduction. Figure 2 illustrates how the NJR policy ensures that the queuing of $k \geq 2$ instances of message m can only take place in an interval whose length is at least $T_m - \Delta - R_{COM} + (k-2)T_m$ (i.e. respecting the period T_m and queuing jitter $J_m^{DEST} = \Delta + R_{COM}$ of the message on the destination network) This constraint is shown as a horizontal line on the graph.

III. Summary and Conclusions

The simple traffic shaping policy *Non-blocking Jitter Reduction* (NJR) introduced in this paper significantly reduces the amount of queuing jitter on gatewayed messages forwarded onto a destination network. The policy is particularly suited to CAN for the following reasons:

- It does not require the use of global time. The source and destination networks can be unsynchronised.
- It does not require the precise time-stamping of when messages are received. Instead only access to a local free-running timer is needed.
- The gatewaying of messages can be performed by a single, simple periodic task that does not block, and can therefore be implemented on a single stack operating system (e.g. OSEK BCC1).

The approach can easily be adapted to cater for clock drifts by simply assuming a slightly smaller period on the destination network, thus ensuring that the long term rate at which messages are gatewayed onto that network does not fall below the rate at which they arrive from the source network. We note that the NJR policy could also be applied to messages forwarded onto other types of network.

Acknowledgements

This work was partially funded by the UK EPSRC funded Tempo project (EP/G055548/1), the EU funded ArtistDesign Network of Excellence. The authors would like to thank Ralph Eastwood for his work on simulating the traffic shaping policy.

References

[1] Bosch. “CAN Specification version 2.0”. Robert Bosch GmbH, Postfach 30 02 40, D-70442 Stuttgart, 1991.

[2] A. Burns, Y. Chen, “Implementing Transactions in a Distributed Real-Time System without Global Time”. In proceedings Work-in-Progress session, RTSS 2009.

[3] R.I. Davis, A. Burns, R.J. Bril, J.J. Lukkien. “Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised”. Real-Time Systems, Volume 35, Number 3, pp. 239-272, April 2007.

[4] ISO 11898-1. “Road Vehicles – interchange of digital information – controller area network (CAN) for high-speed communication”, *ISO Standard-11898*, International Standards Organisation (ISO), Nov. 1993.

[5] K.W. Tindell, J.A. Clark, Holistic schedulability analysis for distributed hard real-time systems, Microprocessing and Microprogramming, Vol. 40, Issues 2-3, pp 117-134, April 1994.

[6] J.S. Turner. New directions in communications (or which way to the information age?). IEEE Communications Magazine, 24(10):8{15, October 1986.

[7] P. Pedreiras, L. Almeida, "Message routing in multi-segment FTT networks: The Isochronous Approach" In proceedings WPDRTS 2004.

[8] J.C. Palencia, M. González Harbour, “Schedulability Analysis for Tasks with Static and Dynamic Offsets”. In proceedings RTSS 1998.

[9] R.I. Davis, A. Burns. "Response Time Upper Bounds for Fixed Priority Real-Time Systems" . In proceedings RTSS 2008

[10] J. Löser and H. Härtig. Low-Latency Hard Real-Time Communication over Switched Ethernet. In proceedings ECRTS, pp 13–22, 2004.

[11] M. Grenier, L. Havet, and N. Navet. Pushing the limits of CAN - scheduling frames with offsets provides a major performance boost. In proceedings ERTS 2008.

[12] T. Nolte, H. Hansson, C. Norstrom, S. Punnekkat. Using Bit-Stuffing Distributions in CAN Analysis. In proceedings RTES 2001.

[13] T. Nolte, H. Hansson, C. Norstrom. “Minimizing CAN response-time analysis jitter by message manipulation”. In Proceedings RTAS, pp 197-206, 2002.

[14] E. Wandeler , A. Maxiaguine , L. Thiele, “Performance analysis of greedy shapers in real-time systems”, In proceedings DATE, 2006.

[15] S.-K.Kweon, K.G. Shin, “Achieving real-time communication over Ethernet with adaptive traffic smoothing”. In proceedings RTAS 2000.

[16] P. Pedreiras, L, Almeida, “Minimizing the end-to-end latency in multi-segment time triggered networks”, INCOM 2004.

[17] R. Santos, P. Pedreiras, M. Behnam, T. Nolte, L. Almeida. “Multi-level Hierarchical Scheduling in Ethernet Switches”. In proceedings EMSOFT 2011.

[18] C. Braun, L. Havet, and N. Navet, “NETCARBENCH: a benchmark for techniques and tools used in the design of automotive communication systems,” in proceedings FET 2007. Available at http://www.netcarbench.org.

[19] M. Di Natale, W. Zheng, C. Pinello, P. Giusto, A.S. Vincentelli, “Optimizing end-to-end latencies by adaptation of the activation events in distributed automotive systems”, pp. 293-302 RTAS 2007

Implementing and Evaluating Communication-Strategies in the ProCom Component Technology

Rafia Inam, Mikael Sjödin
Mälardalen Real-Time Research Centre,
Västerås, Sweden
Email: {rafia.inam, mikael.sjodin}@mdh.se

***Abstract*—This paper presents two strategies to support communication between real-time executable Runnable Virtual Node (RVN) components in the ProCom component technology. We describe the currently implemented server-based communication strategy which uses a dedicated server for communication. We compare the server-based technique with a direct (RVN-to-RVN) communication strategy. The paper also describes how these strategies could be evaluated for real-time performance, and the real-time analysis technologies needed to perform such an evaluation.**

***Index Terms*—real-time software components; hierarchical design**

I. Introduction

ProCom is a component technology for development of hard real-time embedded control-systems [1]. We have previously presented the ProCom-concept of a *runnable virtual node* that allows a two-step deployment process [2]. The first-step is an initial virtual deployment to a virtual node and in the last-phase step virtual nodes are deployed on physical nodes. In this paper we focus on the communication strategies among virtual nodes. We explain how ProCom realizes the communication today, and outline how an alternative strategy for communication could be designed. We also propose an evaluation metric to allow comparison of the real-time performance of two different communication strategies.

In ProCom the realization of a piece of functionality can follow a flow through many software components. Data may originate at one component (e.g. a sensor) and passes through various other computational components, before terminating at the final component (e.g. an actuator). Hence, the data follows a chain of components $(C1, C2, \ldots, Cn)$, each potentially having its own periodicity and timing properties. For an embedded system with real-time constraints, the end-to-end timing behavior is not only dependent on the timing properties of its constituent components but also on the message-chains among those components. ProCom provides a hierarchy of component types, where the top-level component-type is called a *virtual node*. The virtual node (typically) contains a set of other ProCom components and can be synthesized to become an independent executing unit called a *runnable virtual node (RVN)* [3].

In ProCom today, communication *inside* an RVN is implemented with shared buffers and semaphore-locks. In this paper we focus on the communication *between* RVNs, which is implemented using a *communication server*. The communication server provides temporal isolation between RVNs and buffers all communication, making the execution of each RVN very predictable. However, the buffering incurs delays which could adversely affect the fulfillment of real-time requirements. Hence, an interesting approach would be to use a more direct communication strategy, where RVNs communicate with each other directly without an intermediate node. In this paper we describe how a comparison study of the server-based and direct approaches could be performed.

The main contributions of this paper are:

- We present the current server-based implementation for predictable communication between RVNs in ProCom.
- We present an alternative communication strategy using a direct communication method.
- We propose how these two strategies could be evaluated and compared for their real-time performance.

Outline: In Section II, we describe the RVN, how it embeds hierarchical scheduling using the two-level deployment process, and how it is used within the ProCom technology. Section III presents the inter-RVN communication strategies and their performance evaluation criteria are provided in Section IV. Finally, Section V concludes the article and describes future work.

II. The Runnable Virtual Node (RVN)

A runnable virtual node is an execution platform concept that preserves functional as well as temporal properties of the software executed within it [3]. The idea is to encapsulate the real-time properties into model-driven reusable component-based systems to achieve predictable integrations and reusability of those components along with maintainability, testability, and extendibility.

A. An RVN-server

An RVN is implemented as a server within a two-level *Hierarchical Scheduling Framework (HFS)* (an RVN-server), and includes a set of tasks, a resource allocation ($\langle budgetQ, periodP \rangle$ of server), and a real-time scheduler as shown in Figure 1. The scheduler is local-level, and schedules the task set according to allocated resources using a scheduling policy (currently fixed-priority preemptive scheduling FPPS). The final executables that can be downloaded and executed on the physical node consists of a set of RVNs and a top-level real-time scheduler linked together. The top-level scheduler

in the HSF is responsible for dispatching the RVN-servers according to their bandwidth reservations. Thus, once a server has been configured for the RVN, its non-functional (timing) properties are preserved along with its functional properties when the RVN is integrated with other RVNs on a physical node, or when it is reused in another context [3]. Further it reduces the efforts related to testing, and validation.

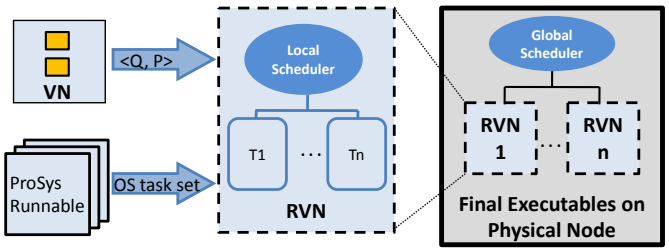


Fig. 1. An RVN encapsulates and preserves functional and non-functional (timing) properties.

In our implementation, the RVN is executed by a server in the HSF running on-top of FreeRTOS [4]. Using HSF, the functionality of different servers can be isolated from each other for, e.g., fault containment, compositional verification, validation and certification, and unit testing [5]. The official release of FreeRTOS only supports a single level fixed-priority scheduling. We have, however, previously presented an implementation of two-level HSF for FreeRTOS [6] with associated primitives for hard real-time sharing of resources both within and between servers [7]. The HSF implementation supports two kinds of servers, idling periodic [8] and deferrable servers [9]. The implementation uses FPPS for both global and local-level scheduling. For local resource-sharing (within a server) the Stack Resource Policy (SRP) [10] is used, and for global resource-sharing (between servers) the Hierarchical Stack Resource Policy (HSRP) [11] is implemented. The HSF supports CPU resource reservations by associating a tuple $\langle Q, P \rangle$ to each server, where P is the server period and the server budget Q ($0 < Q \leq P$) is the allocated portion of CPU resources every P. Given Q, P, and information on resource holding times, the schedulability of a server and/or a whole system can be calculated with the methods presented in [7].

B. The RVN Concept in The ProCom Component Model

The ProCom component technology targets control-intensive embedded systems like software used in trains, airplanes, cars, industrial robots, etc. The ProCom component model [12] is specifically developed to address the reuse of design artefacts (e.g., extra-functional properties, analysis results, and behavioral models) as well as predictable integration and reuse of the executable components [3]. The PROGRESS Integrated Development Environment (PRIDE) tool [13] supports modeling and automatic-synthesis of components at different levels [1].

The RVN concept uses a two-level deployment process rather than a single big deployment: i.e. deploying functional entities to RVNs in *a first-step* (during which, e.g., the timing properties of RVNs are validated), and then, deploying RVNs to the physical nodes in *a second-step* (integrating RVNs along with their preserved timing properties) [3]. The two-level process gives development benefits with respect to composability, system integration, testing, validation, certification, and reuse.

The RVN is an integrated concept in ProCom. From the modeling perspective, a virtual node (VN) is equivalent to a set of ProSys components with an added resource reservation. In the executable form, the RVN is constructed by mapping the set of tasks (synthesized from ProSys-runnable components) to a server and assigning scheduling parameters (assignment of task-priorities) during the first-step of deployment. Internal validity of the timing-constraints of the RVN can then be assessed using, e.g., simulation, testing or a *local scheduling-analysis* provided in [7]. In this manner, after configuration the RVN-server preserves its timing properties within it. It also adds an implementation of message channels used to send messages among virtual nodes. The final binaries are generated for a hardware node during the second-step of deployment by connecting different RVNs together with a global scheduler, assigning server-priorities, and using a *middleware API* for *inter-RVN communications*.

III. The Inter-RVN Communication

The RVN provides main benefits of predictable integration and increased reuse of executable real-time components. It leads up to the necessity to make the communication among RVNs not only fast and predictable but the communication should also support the reuse of the executable components. To achieve these benefits, the inter-component or inter-RVN communication is implemented independently from the underlying platform as a *middleware API* by moving the information about system and communication outside the component code. Later the middleware interface functions are integrated into the layered ProCom model.

A. Middleware API

The inter-RVN communication is a combination of data and trigger ports and is based on messages. The middleware API is implemented a-synchronously via message passing and the cyclic shared buffers, where channels are used to distribute the messages to other RVNs using a defined set of connections. The communication is independent of the underlying operating system (HSF implementation in our case). It includes the support for transparent communication within RVNs mapped on the same hardware node, called *local-RVN communication*, and among RVNs mapped on different hardware nodes through a communication media or channel (e.g. CAN bus), called *distributed-RVN communication*. Both types of communication can be synthesized before generating the final binaries for the target platform.

The middleware API is well-integrated with the layered deployment process of the ProCom model. The main communication code (including data structures) is initialized and two periodic tasks *sender* and *message-port updater* are created at every physical node during the first-step of deployment. These

tasks are responsible to send and receive messages among RVNs respectively.

The inter-RVN communication could be realized in two different ways: either integrating the middleware API directly into the communicating RVNs, called *Direct Communication*, or using a server to embed the middleware tasks in it, called a *Server-based Communication*. Both strategies are explained here.

B. Server-based Communication Strategy

Since RVNs are implemented as servers within a two-level HSF, it makes sense to embed the middleware API within a server. A *Communication (also called a system) server* along with its timing properties is automatically generated for inter-RVN communications (if needed), at the second-step of deployment. Both communication tasks, sender and message-port updater, are automatically assigned to the server as shown in Figure 2. Additionally there can be a hardware-driver task in it if needed. The communication server has the highest priority among all the servers in the system, with a very small budget and its only functionality is to copy the messages from the sender port of one component to the receiver port of another component, by executing the middleware API tasks within it.

This method provides benefits of (1) increased reuse of RVN by keeping the communication separated from the RVN code and (2) predictability by executing the communication API in a server within the HSF implementation. However, it has some overhead of server execution. Moreover, it also increases (3) maintainability and flexibility to change the communication code without affecting the timing properties of RVNS.

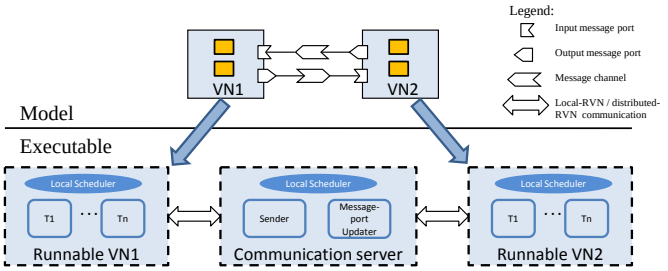


Fig. 2. Server-based inter-RVN communication at modeling and executable levels.

The final executables are generated by resolving the local-RVN communication by mapping it to middleware API, and synthesizing the distributed-RVN communication among hardware nodes (if needed). All synthesis is done by generating C-code, so the final step is compiling the generated code, and linking all code with the operating system and middleware binaries.

The local-RVN communication is provided by the middleware where the output and input message ports write and read the data respectively. One-step shared cyclic buffers that can be accessed by multiple tasks, are added to these ports for reliable message delivery and efficient memory usage. The only additional requirement is a communication channel to be generated for the distributed-RVN communication. (At this stage the PRIDE tool provides the distributed-RVN communication in the form of a TCP/IP connection over Ethernet. A real-time distributed-RVN communication is not automatically generated by the tool and has to be provided manually).

Since inter-RVN communication is implemented as a server within a two-level HSF, simple semaphores cannot be used to protect shared buffers. Thus some synchronization protocols for hierarchical schedulers are required to access the buffers in a safe manner. SRP and HSRP protocols are implemented in the HSF implementation [7]. The shared buffers among the tasks of same RVN and among tasks of different RVNs are arbitrated using SRP and HSRP respectively. HSF leverages the communication among components with the advantages of short and predictable global blocking, and predictable and well-defined communication.

C. Direct Communication Strategy

Inter-RVN communication can also be done directly among RVNs (i.e. RVN-to-RVN) without using the communication server. Shared message queues which are accessed via SRP and HSRP APIs [7] can be used for this purpose. The RVN can encapsulate the middleware API within it to send and receive the data and/or messages (see Figure 3) at the first-step of deployment. It requires a separate configuration of RVN for each communication. The final binaries are generated from RVNs along with the code for the communication mechanism used (local- or distributed-RVN) at the final-step of deployment.

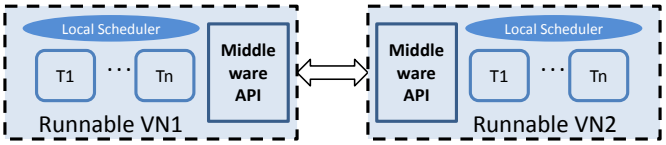


Fig. 3. Direct inter-RVN communication at executable level.

The direct communication is fast as compared to the server-based communication. However, it reduces the reusability of executable RVNs since RVNs need to be configured for each system separately, depending on that system's communication requirements. Further, any change in the middleware communication code will not only require a code-change in all RVNs but will also affect the timing properties of all RVNs involved in that communication.

IV. Performance Evaluation

We plan to evaluate and compare the costs of the server-based strategy with that of the direct communication strategy. To perform the evaluation, we plan to implement a simulator, generating random subsystems and tasks in subsystems, both types of communication, and measuring the costs of server-based and direct communication methods among RVNs. For evaluation, the end-to-end latencies of inter-RVN communication should be measured for both strategies and then compared.

A. Evaluation Criteria

We propose the use of a metric *Worst Case Reaction Time (WCRT)* as the total time taken by a message-chain from message-originator until the message reaches its message-terminator component. For hard real-time systems, of course, the worst case is a more interesting metric than any statistical metric. Also, metrics like worst-case response-time for tasks do not help us to evaluate end-to-end delays.

B. Evaluating the Server-based Strategy

When using the communication server, there is no direct control-flow between components. Instead, data flows through buffers inside the communication server.

The originator, the communication server, and the terminator are running asynchronous, each with its own period. This makes the traditional scheduling analysis technique very difficult to use to obtain the WCRT. Instead, data-path analysis, like the technique proposed by Feiertag *et al.* [14], could be used. A prerequisite to use their technique is that the worst-case response-times of each task in the RVNs are known. The response-times can be calculated using the analysis techniques proposed in [7].

C. Evaluating the Direct Communication Strategy

When using direct communication, control-flow is direct between the RVNs and we could use existing scheduling-analysis techniques to calculate WCRTs. However, different analysis techniques would likely give different results – and there are no analysis techniques that provide a perfect fit for our strategy, rather a complicated task-model.

Real-time calculus could be used to summarize delays from the different components [15]. This could take into account the potential difference in periodicity between an RVN and the components executing inside the RVN. However, it would be difficult to model the explicit control-flow between RVNs in real-time calculus. To improve the precision it would be desirable to use more exact analysis techniques based on the *holistic scheduling analysis* by Tindell [16] with explicit modeling of precedence constraints [17]. These techniques cannot be used right out of the box though, instead they would have to be incorporated into the existing scheduling analysis applicable to the RVNs [7].

V. Conclusions and Future Work

We have presented two different strategies to support communication among real-time executable components RVNs, and plan to evaluate and compare them. We have implemented the server-based technique as a means to encapsulate the middleware API to provide a predictable communication mechanism to preserve the RVN's timing properties at run-time and for better RVN reuse.

For future work, we plan to compare the cost of the server-based method with the cost of the direct communication method (among components/subsystems). We plan to present the simulated results of our evaluation for both methods and their comparison. The direct communication can be evaluated by incorporating the scheduling analysis of [16], [17] into the analysis techniques provided in [7], while the server-based method will be evaluated using the techniques provided in [14].

At the moment the message passing mechanism is implemented using periodic tasks for receiving and sending messages. This may cause delays in the system and inefficient use due to unnecessary idle time. It would be interesting to see how much performance can be gained by making the message passing mechanism event-triggered. For example, we expect a better response time, less context switching.

References

[1] Etienne Borde and Jan Carlson. Towards verified synthesis of procom, a component model for real-time embedded systems. In *14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE)*. ACM, June 2011.

[2] R. Inam, J. Mäki-Turja, J. Carlson, and M. Sjödin. Virtual Node – To Achieve Temporal Isolation and Predictable Integration of Real-Time Components. *International Journal on Computing (JoC)*, 1(4), 2012.

[3] Rafia Inam. *Towards a Predictable Component-Based Run-Time System*. Number 145. Licentiate thesis, January 2012.

[4] FreeRTOS web-site. http://www.freertos.org/.

[5] Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *IEEE Real-Time Systems Symposium(RTSS'97)*, pages 308–319, 1997.

[6] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, S. M. H. Ashjaei, and Sara Afshar. Support for Hierarchical Scheduling in FreeRTOS. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11)*.

[7] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, and Moris Behnam. Hard Real-time Support for Hierarchical Scheduling in FreeRTOS. In *7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11)*, pages 51–60, Porto, Portugal, 2011.

[8] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some Practical problems in Prioritised Preemptive Scheduling. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 181–191, 1986.

[9] J.K. Strosnider, J.P. Lehoczky, and L. Sha. The deferrable server algorithm for Enhanced Aperiodic Responsiveness in Hard Real-time Environments. *IEEE Transactions on Computers*, 44(1), 1995.

[10] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[11] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *IEEE Real-Time Systems Symposium (RTSS'06)*, pages 389–398, 2006.

[12] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković. A Component Model for Control-Intensive Distributed Embedded Systems. In *11th International Symposium on Component Based Software Engineering*, pages 310–317, October 2008.

[13] PRIDE Team. PRIDE: the PROGRESS Integrated Development Environment, 2010. "http://www.idt.mdh.se/pride/?id=documentation".

[14] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics. In *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'08)*, November 2008.

[15] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *IEEE International Symposium on Circuits and Systems*, pages 101–104 vol. 4, Geneva, May 2000.

[16] K. Tindell and J. Clark. Holistic Schedulability Analysis For Distributed Hard Real-Time Systems. Technical Report YCS197, Real-Time Systems Research Group, Department of Computer Science, University of York, November 1994. URL ftp://ftp.cs.york.ac.uk/pub/realtime/papers/-YCS197.ps.Z.

[17] J.C. Palencia Gutierrez and M. Gonzalez Harbour. Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems. In *Proc. 20^{th} IEEE Real-Time Systems Symposium (RTSS)*, December 1999.

Enhancing the Real-time Capabilities of the Linux Kernel

Paulo Baltarejo Sousa, Nuno Pereira, and Eduardo Tovar
CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto
4200-072 Porto, Portugal
{*pbs,nap,emt*}*@isep.ipp.pt*

***Abstract*—The mainline Linux Kernel is not designed for hard real-time systems; it only fits the requirements of soft real-time systems. In recent years, a kernel developer community has been working on the PREEMPT-RT patch. This patch (that aims to get a fully preemptible kernel) adds some real-time capabilities to the Linux kernel. However, in terms of scheduling policies, the real-time scheduling class of Linux is limited to the First-In-First-Out (`SCHED_FIFO`) and Round-Robin (`SCHED_RR`) scheduling policies. These scheduling policies are however quite limited in terms of real-time performance. Therefore, in this paper, we report one important contribution for adding more advanced real-time capabilities to the Linux Kernel. Specifically, we describe modifications to the (PREEMPT-RT patched) Linux kernel to support real-time slot-based task-splitting scheduling algorithms. Our preliminary evaluation shows that our implementation exhibits a real-time performance that is superior to the scheduling policies provided by the current version of PREMPT-RT. This is a significant add-on to a widely adopted operating system.**

I. Introduction

Multiprocessors implemented on a single chip, called *multicores*, are a mainstream computing technology. Multicores with 8 cores are common on desktops today and it is already possible to find commercial chips with up to 100 generic processing cores [1]. With chip manufacturers focused on improving performance by increasing the number of cores, it is expected that the number of cores per chip will continue to increase.

Due to its wide adoption, Linux is well positioned to take an important role leveraging the processing power of large multicores and its wide adoption is also driving developments towards enabling real-time computing by using the Linux kernel. The main objective of such efforts is reducing the unpredictability sources that exist in the mainline Linux kernel, as these can cause arbitrary delays to the real-time tasks running on the system.

There are many sources of unpredictability in the Linux kernel: (i) interrupts are generated by the hardware often in an unpredictable manner and when an interrupt arrives, the processor execution switches to handle it; (ii) multiple kernel threads running on different processors in parallel can simultaneously operate on shared kernel data structures, requiring serialization of the access to such data; (iii) disabling and enabling preemption features used in many parts of the kernel code can postpone some scheduling decisions.

Currently, the PREEMPT-RT patch[1], is the foremost development effort towards supporting the execution of real-time tasks using the Linux kernel. The PREEMPT-RT patch addresses these sources of unpredictability by making most of the Linux kernel preemptible, by implementing priority inheritance (to avoid priority inversion phenomena), and by converting interrupt handlers into preemptible kernel threads. These are important properties to enable real-time computing. However, appropriate real-time scheduling policies are also needed.

The real-time scheduling class implemented in the PREEMPT-RT patch supports the same scheduling policies of the mainline Linux kernel: the First-In-First-Out (`SCHED_FIFO`) and Round-Robin (`SCHED_RR`) scheduling policies. While these scheduling policies are appropriate for unicore processors, they are not adequate for multicore (or multiprocessor) systems because (i) their performance is poor on multiprocessors - there exist task sets where a system with a load a little over 50% will fail to meet deadlines and (ii) they adopt an active push-pull approach for balancing tasks across processors - since the Linux kernel uses a per-processor runqueue (a runqueue stores ready tasks), such push-pull operations require locking multiple processor runqueues, which are an additional source of unpredictability.

This paper describes the modifications of the (PREEMPT-RT patched) Linux 3.2.11-rt20 kernel to support real-time task-splitting scheduling algorithms where the time is divided into timeslots (called slot-based task-splitting). Slot-based task-splitting scheduling algorithms [2], [3] assign most tasks (called *non-split tasks*) to just one processor and a few (called *split tasks*) to only two processors and have a utilization bound of 65%, configurable up to arbitrarily close to 100% at the cost of more preemptions and migrations.

Among existing slot-based task-splitting scheduling algorithms, the Notional Processor Scheduling – Fractional capacity (NPS-F) [3] is notable for its high utilization bound (configurable from 75% up to arbitrarily close to 100%) and is the focus of the implementation reported in this paper. NPS-F is a semi-partitioned multiprocessor scheduling algorithm: tasks are partitioned to servers (termed *notional processors*), in turn mapped onto the (physical)

[1]Available online at http://www.kernel.org/pub/linux/kernel/projects/rt/

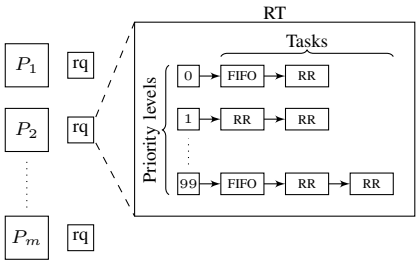


Figure 1. RT scheduling class Runqueue

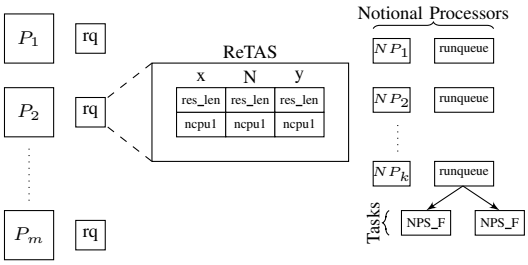


Figure 2. ReTAS Runqueue

processors. Some notional processors use just one physical processor; others use two processors and migrate between them in a controlled manner.

This work is an evolution of [4], which implements the same scheduling algorithms by modifying the mainline Linux Kernel. In that previous work, a scheduling policy module – ReTAS (Real-time TAsk-Splitting) – was added on top of the native Linux module hierarchy, making ReTAS the highest priority module. However, such approach cannot be employed in conjunction with the PREEMPT-RT patch because important functionalities, such as timer interrupt handlers, needed by the ReTAS scheduler, are implemented within the real-time scheduling class and thus ReTAS cannot have a higher priority than the real-time scheduling class implemented in the PREEMPT-RT patch.

II. Background on Real-Time Scheduling in the Linux Kernel and PREEMPT-RT

The linux kernel scheduler consists of a scheduler core (or dispatcher) and various modules, where each module implements a scheduling class encapsulating a scheduling policy. These scheduler modules are hierarchically organized by priority and the dispatcher looks for a runnable task of each module in a decreasing order of priorities. Currently, the Linux kernel implements three native scheduler modules: RT (Real-Time), CFS (Completely Fair Scheduling) and Idle. The dispatcher first inquires the RT module for a runnable task and, if this module does not have any ready task, the dispatcher then inquires the CFS module. The Idle module is used for the idle task, executed when there is no other runnable task.

As depicted in Figure 1, tasks in the RT scheduling class are organized by priority level. Inside each priority level, the RT module implements two scheduling heuristics: `SCHED_FIFO` and `SCHED_RR`. `SCHED_FIFO` is based on the first-in-first-out heuristic: whenever a `SCHED_FIFO` task is executing, it continues until preempted (by a higher-priority task) or blocked (e.g., by an I/O operation). `SCHED_RR` implements the round-robin heuristic: a `SCHED_RR` task executes (if it is not preempted or blocked) until it exhausts its timeslice.

The mainline Linux defines one runqueue (an instance of `struct rq`, where all ready tasks are stored) per-physical processor and, at any time instant, the processor is executing one task stored in its runqueue. This may result in unbalanced workloads across processors. In order to balance the workload across processors, the RT module adopts an active push-pull strategy as follows: whenever the dispatcher inquires the RT module, it first tries to pull the non-executing highest-priority task from the other runqueue (if it is not in its runqueue) and, after selecting the next running task, it checks if it can push the (freshly) preempted task to another processor which is executing a task with lower priority than the preempted task. Observe that, moving tasks between two runqueues requires locking both runqueues and this may introduce considerable overheads.

The PREEMPT-RT patch reduces the kernel latencies by reducing its non-preemptible sections. This is done by replacing most kernel spinlocks by mutexes, which support priority inheritance, and by transforming all interrupts handlers into preemptive kernel threads, scheduled by the RT scheduling class. These kernel threads have assigned a priority level (50 by default) and, therefore, they can be preempted by other RT tasks with higher-priority. As mentioned before, the RT scheduling class does not implement any scheduling algorithm suitable for multiprocessor systems and the PREEMPT-RT patch does not add any scheduling algorithms suitable for multiprocessor systems.

III. Implementing slot-based task-splitting

Achieving an effective implementation of NPS-F in the Linux Kernel is a challenging task, as it requires efficient mechanisms to: (i) handle migrations; (ii) manage ready tasks; (iii) handle reserves and (iv) mapping of notional processors to physical processors. The following section discusses these challenges and Section III-B describes how the ReTAS scheduler module was integrated into the RT scheduling class.

A. Issues in implementing NPS-F

As explained before, the mainline Linux kernel may incur in overheads due to the way task migrations are implemented. In NPS-F, migrations involve the entire notional processor. As this would typically imply moving multiple tasks, adopting a similar strategy for the implementation of NPS-F would be inefficient. Indeed, our implementation

employs a different arrangement that largely solves these issues. Namely, we opt for one runqueue per *notional* (not per physical) processor (see Figure 2 - ReTAS is used to denote the implementation of slot-based task-splitting scheduling algorithms). Under this approach, all ready tasks assigned to a notional processor are always stored on (i.e. inserted to/dequeued from) the same respective (per-notional-processor) runqueue. Then, when a notional processor migrates (i.e. with all its tasks) from processor P_p to processor P_{p+1}, we simply change the runqueues used by P_p and P_{p+1}.

To implement NPS-F, each physical processor needs to be configured with its timeslot composition. For this purpose, we introduce the following set of variables that store information about the processor reserves. The variable `begin_curr_timeslot` stores (as suggested by its name) the beginning of the current timeslot and it is incremented by S (the timeslot length). Observe that no synchronization mechanism is required for updates to this variable. The timeslot composition is defined by an array of 2-tuples <`res_len`, `ncpu1`> (see Figure 2). Each element of this array maps a reserve of length (`res_len`) to the notional processor (`ncpu1`). A timer is used to trigger scheduling decisions at the beginning of each reserve.

If one observes two consecutive timeslots, whenever a split notional processor consumes its reserve on processor P_p, whichever task was executing at the time has to "immediately" resume execution on another reserve on processor P_{p+1}. However, due to many sources of unpredictability, common in a real operating system, arbitrary levels of time precision are not possible. Consequently, the dispatcher of processor P_{p+1} can be prevented from selecting the task in consideration from execution because processor P_p has not yet relinquished (the runqueue associated with) that task.

One solution could be for processor P_{p+1} to send an inter-processor interrupt (IPI) to P_p to relinquish (the runqueue associated with) that split task. Another could be for P_{p+1} to set up a timer x time units in the future to force the invocation of its dispatcher. We chose the latter option for two reasons: (i) we know that if a dispatcher has not yet relinquished the split task it was because something is preventing it from doing so (e.g. the execution of an interrupt service routine (ISR)); (ii) using an IPI solution introduces a dependency between processors that can compromise the scalability of the dispatcher.

B. Adding ReTAS to the RT Scheduling Class

Apart from the required code to manipulate notional processors (enqueue and dequeue of ReTAS tasks as well as getting the task with earliest absolute deadline) and the timeslot infrastructure, incorporating ReTAS into the RT scheduling class implies a set of modifications in functions implemented in the `sched_rt.c` file. Those functions are `enqueue_task_rt`, `dequeue_task_rt`, `check_preempt_curr_rt`, and `pick_next_task_rt`.

The `enqueue_task_rt` is called whenever a RT task enters into a runnable state. If the runnable task is a ReTAS task (ReTAS tasks are also RT tasks, with a priority level, but are scheduled according to the `SCHED_NPS_F` scheduling policiy), then it is enqueued into the respective notional processor runqueue. When a RT task is no longer runnable, then the `dequeue_task_rt` function is called to remove the task from the respective notional processor runqueue.

As the name suggests, the `check_preempt_curr_rt` function (Listing 1) checks whether the currently running task must be preempted or not (e.g. when a RT task wakes up). It receives two pointers, one for the processor runqueue that is running this code (`rq`) and another to the woken up task (`p`). If the priority of the woken up task is higher or equal than (lower `prio` values mean higher priority) the currently executing task (pointed by `rq->curr`), it checks if `p` is a ReTAS task and if the current reserve is mapped to task `p` notional processor. If that is the case, then the currently running task is marked for preemption.

```
static void
check_preempt_curr_rt(struct rq *rq, struct task_struct *
    p, int flags)
{
 if (p->prio <= rq->curr->prio)
   if(retas_policy(p->policy))
     if(_check_preempt_curr_retas(rq)){
       resched_task(rq->curr);
       return;
     }
 ...
}
```

Listing 1. Changes on the `check_preempt_curr_rt` function.

The `pick_next_task_rt` function also needs a small modification. This function selects the task to be executed by the current processor and is called by the dispatcher whenever the currently executing task is marked to be preempted or finishes its execution. It first gets the highest-priority RT task, then it gets the highest-priority ReTAS task. After, it selects the highest-priority task between them, if there is some, otherwise it returns `NULL`, which forces the dispatcher to inquiry the CFS scheduling module.

IV. Evaluation

In order to compare the performance of our implementation[2] of the NPS-F scheduling policy with the Linux native real-time scheduling policies, we have conducted a range of experiments with a 4-core platform (Intel(R) Core(TM) i7 CPU @ 2.67GHz). A set of implicit-deadline task sets was generated as follows. We have defined four types of tasks: "normal", 'heavy", "medium", and "light", where normal tasks have a u_i (the individual task utilization) in the range 0.05 to 0.95. Heavy tasks have a u_i in the range 0.65 to

[2]The implementation is available at http://webpages.cister.isep.ipp.pt/~pbsousa/retas/

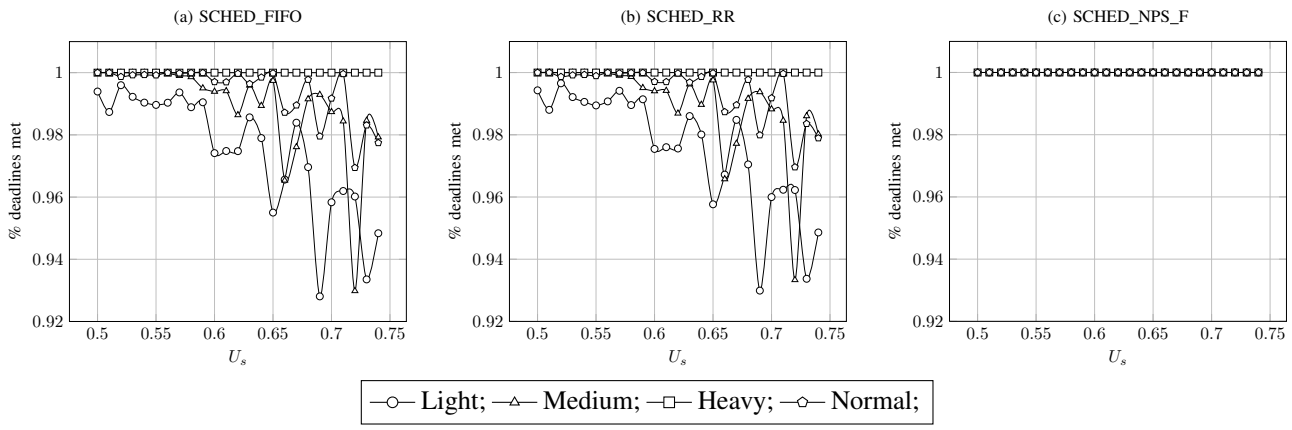


Figure 3. Success (deadlines met) Ratio of Tasks using SCHED_FIFO, SCHED_RR and SCHED_NPS_F

0.95, while medium tasks have a u_i in the range 0.35 to 0.65. Finally, light tasks have u_i in the range 0.05 to 0.35.

For each of the four task types, we generated 5 task sets and repeated this for 25 different U_s ($U_s = \sum_{\tau_i \in \tau} \frac{C_i}{T_i}$) values, varying from 0.50 to 0.75 (this value is the utilization bound of the NPS-F scheduling algorithm) with an increment of 0.01. The periodicity of all tasks was uniformly generated in the range 5 ms to 50 ms. The characteristics of these tasksets are not particularly suited for NPS-F. They were generated with the purpose of testing high-utilization periodic workloads with different characteristics. There is a relevant setup phase in NPS-F, prior to runtime, were tasks are assigned to processors. This phase is only part of NPS-F, and thus other algorithms have a slightly more simplified setup.

We ran each task set using the `SCHED_FIFO`, `SCHED_RR`, and `SCHED_NPS_F` for a total of over 39 hours. Figure 3 plots the percentage of all task instances executed during the experiment which met its deadline.

As it can be seen by inspecting Figure 3, that `SCHED_NPS_F` was the only scheduling policy that met all deadlines, as predicted by the theory. Both `SCHED_FIFO` and `SCHED_RR` fail to meet all deadlines. Observe that longer experiments could reveal a different percentage of deadlines met, however, the theory tells us that all deadlines will be met in a correct NPS-F scheduler. Interestingly, with heavy tasks, none of the schedulers fails deadlines. This is expected as, for this case, the task set generation method produces a number of tasks which is smaller or equal to m, thus each processor is only assigned one task.

V. Conclusions

This paper addressed the relevant problem of providing adequate real-time scheduling policies for multicore systems using the Linux kernel. We have overviewed the implementation challenges posed by this implementation and overviewed the structure and the main modifications introduced. We also presented an evaluation showing that our implementation is able to meet all deadlines and that its real-time performance is superior to that of the other real-time scheduling policies available in the Linux kernel.

Our contribution is completely compatible with the PREEMPT-RT patch and was implemented with minor modifications. In our opinion, adding adequate scheduling algorithm to the Linux kernel (compatible with PREEMPT-RT patch) is an important concern to make this widely adopted operating system more suitable for real-time systems.

Acknowledgment

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project Ref. FCOMP-01-0124-FEDER-022701 and also the REHEAT project, ref. FCOMP-01-0124-FEDER-010045.

References

[1] Tilera, "TILE-Gx processor family overview," http://www.tilera.com/products/processors/TILE-Gx_Family.

[2] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemption," in *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Application (RTCSA 06)*, Sydney, Australia, 2006, pp. 322–334.

[3] K. Bletsas and B. Andersson, "Notional processors: an approach for multiprocessor scheduling," in *15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09)*, San Francisco, CA, USA, 2009, pp. 3–12.

[4] P. B. Sousa, K. Bletsas, E. Tovar, and B. Andersson, "On the implementation of real-time slot-based task-splitting scheduling algorithms for multiprocessor systems," in *Proc. of the 13th Real-Time Linux Workshop (RTLWS'13)*, Prague, Czech Republic, 2011, pp. 207–218.

Cleaning Up Linux's CPU Hotplug For Real Time and Energy Management

Thomas Gleixner
Linutronix

Paul E. McKenney
IBM LTC assigned to Linaro

Vincent Guittot
ST-Ericsson assigned to Linaro

Abstract

Linux's CPU-Hotplug facility was originally designed to allow failing hardware to be removed from a running system. Hardware fails quite infrequently, so CPU-hotplug performance (much less real-time response) was not a major consideration. However, CPU hotplug is now used for energy management and (believe it or not!) real-time response, both of which have unsurprisingly exposed some shortcomings in CPU hotplug. This document reviews a number of these shortcomings, and then proposes an alternative CPU-hotplug approach that we believe will address these shortcomings.

1 Introduction

The Linux kernel's CPU-hotplug facility allows CPUs to be added to or removed from a running kernel. CPU hotplug has historically been used to isolate failing CPUs or to simplify running scalability benchmarks [5]. The roots of the Linux kernel's CPU-hotplug facility go back almost ten years [4], but during that time, it has gained a number of additional uses, including adjusting the sizes of guest OSes in virtualized environments, clearing current and future work from a given CPU, and improving energy efficiency. These new uses place considerable stress on the Linux kernel's implementation, which was not designed with them in mind.

Linux's CPU-hotplug implementation is based on *notifiers*, which are callbacks into the subsystems that need to be aware of CPUs coming and going. These notifiers are invoked repeatedly in multiple phases, so that when a CPU is coming online, they are invoked with `CPU_UP_PREPARE` (which runs on some other CPU), then `CPU_STARTING` (which runs with interrupts disabled on the CPU coming online), and finally `CPU_ONLINE` (which might run on any CPU, but after the CPU has come online). CPUs going offline have four notification phases: `CPU_DOWN_PREPARE` (which might run on any CPU), `CPU_DYING` (which runs with interrupts disabled on the offlining CPU while all other CPUs are spinning waiting), `CPU_DEAD` (which runs on some other CPU after the CPU has gone offline), and `CPU_POST_DEAD` (which runs on some other CPU after some of the CPU-hotplug locks have been dropped). The `CPU_UP_PREPARE` and `CPU_DOWN_PREPARE` notifiers are permitted to "fail", in other words, to refuse to allow the hotplug operation to proceed.

Section 2 reviews shortcomings of the current implementation, Section 3 overviews our work in progress, Section 4 lists alternative proposals, and Section 5 lists potential issues with our approach.

2 CPU-Hotplug Shortcomings

The shortcomings of CPU hotplug are well known, but worth discussion. The most obvious from a real-time-computing perspective is OS jitter, discussed in Section 2.1. From a Linux-kernel implementation viewpoint, the lack of a well-defined CPU model during hotplug operations is most vexing, as described in Section 2.2. A few unlucky portions of the Linux kernel must correctly handle offline (or "zombie") CPUs, which is covered in Section 2.3. Finally, CPU hotplug notifies kernel subsystems of hotplug operations, but a number of these notifiers run in extremely constrained software contexts, as documented in Section 2.4.

2.1 Overhead and OS Jitter

In a perfect world, a given CPU could come online or go offline quickly and without disturbing the rest of the system. Unfortunately, in this world, handling the CPU's per-CPU kthreads takes a long time (hundreds of milliseconds or even seconds) [1]. This limits CPU hotplug's use as an energy efficiency measure because the CPU must stay powered off for quite some time to make

up for the CPU-hotplug overhead [3][1]. Furthermore, CPU hotplug uses __stop_machine(), which halts application execution on all online CPUs for an extended period of time. This rules out use of CPU hotplug for real-time workloads—and makes its use difficult on battery-powered systems because the CPU hotplug operation might consume more energy than is saved by powering off the CPU for a short time.

The traditional rule "don't use CPU hotplug on real-time systems" is now starting to fail due to real-time guests running on real-time hypervisors. In this case, the hypervisor needs to offline a failing CPU without causing all the real-time guest OSes to miss their deadlines. Furthermore, a guest might need to add or remove CPUs without disrupting its real-time application.

In addition, offlining and immediately onlining a CPU has the useful side-effect of forcing all current and future work off of that CPU, providing better response to its real-time application. Unfortunately, such offlining and onlining will cause any pre-existing real-time application running on that same system to miss its deadlines.

These use cases are specific examples of the trend towards increasing general-purpose functionality on real-time systems [2]. Now that CPU hotplug has moved away from its original intended use of removal of failing CPUs, excessive overhead and OS jitter from CPU-hotplug operations is no longer acceptable.

2.2 Ill-Defined Model of CPU

Suppose that a given CPU is going offline, and that half of its notifiers have completed. What state is the CPU in?

The answer to this question is unclear. The CPU is marked as online in the cpu_online_mask, but some of its functionality really has been disabled. Worse yet, the default is for the notifiers to execute in the same order as they were registered at boot time. This is a problem because the boot process adds capabilities to the CPU in a good order that respects dependencies among those capabilities, which means that removing them in the same order can be problematic. For example, the scheduler uses both RCU and IPIs, and during boot, RCUs and IPIs are initialized before the scheduler. So the boot process builds up the CPU's capabilities in a tree-like fashion, and then the CPU-hotplug system attempts to remove the tree starting at the trunk, in this case removing RCU and IPIs before removing the scheduler.

Although notifier priorities are used to handle this specific case, this requires painstaking manual intervention. The Linux kernel deserves better.

[1] Five milliseconds is a good upper bound on CPU-hotplug latency.

2.3 Zombie CPUs

The __stop_machine() primitive forces all CPUs on the system to switch to special kernel threads (kthreads). The outgoing CPU then executes CPU_DYING class of notifiers in the context of its kthread, while the other CPUs spin in the context of their kthreads. Therefore, a newly offlined CPU passes through the scheduler when switching from its __stop_machine() kthread to the idle loop, where it is powered off.

This in turn means that both the scheduler and RCU must handle zombie offline CPUs for a short period after they have marked themselves offline. RCU handles this by assuming that a given CPU will not remain a zombie for more than one jiffy, which does currently work, but will eventually lead to baffling failures. Again, the Linux kernel deserves better.

2.4 Inconvenient Software Contexts

The CPU_DYING and CPU_STARTING classes of notifiers execute with interrupts disabled, which prevents them from blocking, which in turn prevents them from starting or stopping kthreads, which in turn can be problematic.

For example, when preemptible RCU is configured with priority boosting, it uses a set of per-CPU kthreads to boost callback-execution priority. RCU must interact with these threads in the CPU_UP_PREPARE, CPU_ONLINE, CPU_DOWN_PREPARE, and CPU_DEAD notifiers, which means that RCU must deal with either a CPU that doesn't have an RCU kthread or an RCU kthread that doesn't have a CPU, both of which are fragile and bug-prone. Once again, the Linux kernel deserves better.

3 Approach

A successful approach to new-age CPU hotplug must provide the following:

1. Robust design for CPUs that are partially online.
2. Simple and fast handling of per-CPU kthreads.
3. Explicit specification of notifier dependencies.
4. Parallel CPU-hotplug notification.
5. Full software capabilities in all CPU-hotplug notifiers.
6. Elimination of OS jitter.

To provide all this, our approach provides a generic facility to create and park per-CPU hotplug kthreads (see Section 3.1), executes execute in per-CPU kthread context (see Section 3.2, runs notifiers in reverse order for offline (see Section 3.3), and restricts hotplug-time execution to per-CPU hotplug kthreads (see Section 3.4.

```
 1 static int my_cpu_hotplug_kthread(void *arg)
 2 {
 3   int cpu = (int)(long)arg;
 4
 5   /*
 6    * Code here from CPU_STARTING notifier.
 7    */
 8
 9   cpu_hotplug_kthread_started();
10
11   while (!kthread_should_park()) {
12
13     /* Do actual work here. */
14
15   }
16
17   /*
18    * Code here from CPU_DYING notifier.
19    */
20
21 }
```

Figure 1: CPU-Hotplug Per-CPU kthread Structure

3.1 Generic Per-CPU Hotplug kthreads

Vincent established that creation and deletion of kthreads can add multiple *seconds* to CPU-hotplug latencies [1]. One best to avoid this is simply to leave those kthreads in place while the corresponding CPU is offline. A key observation (due to Thomas) is that the Linux kernel already has some threads that remain runnable and bound to offline CPU, namely the idle threads. A generic facility will allow the per-CPU hotplug kthreads to have this same capability, so that they remain quiescent while the corresponding CPU is offline.

3.2 Notify From Per-CPU kthreads

Given special kthreads managed by the CPU-hotplug facility, it makes sense to run the code currently in CPU-hotplug notifiers from within these kthreads. This allows the `CPU_DYING` and `CPU_STARTING` notifiers to use the full capabilities of the scheduler, allowing more of the notifier code to execute at `CPU_DYING` and `CPU_STARTING` time. This in turn simplifies the CPU-hotplug per-CPU kthreads, as shown by the `my_cpu_hotplug_kthread()` function in Figure 1. When a CPU boots or comes online, this function is invoked. When it finishes initialization, it invokes `cpu_hotplug_kthread_started()` as shown on line 11, signalling that the next notifier or kthread may now be started.

The loop spanning lines 11-15 terminates when `kthread_should_park()` returns true, indicating that the corresponding CPU is going offline. The function then executes offline-time cleanups as indicated by the comment on lines 17-19.

As noted earlier, the `CPU_UP_PREPARE` and `CPU_DOWN_PREPARE` phases can block CPU hotplug. Those that actually do (for example, `smp_core99_cpu_notify()`) must remain notifiers. That said, most do not, and can therefore can run in kthread context.

However, a great many of the Linux kernel's notifiers do not involve a kthread. Creating an additional per-CPU kthread for each of these notifiers would be overkill, so these notifiers should remain notifiers. They nevertheless should run in kthread context, for example, in the context of the kthread coordinating CPU-hotplug operation.

3.3 Reverse Notifier Order For Offline

One of the reasons for notifier priorities and for the current multi-phase CPU-hotplug operation is dependencies among different subsystems. These dependencies must be handled manually in a distributed fashion, and is a major source of pain and of bugs.

A better approach is to note that CPU hotplug is not atomic, and that CPUs are booted up in an orderly manner, with later function depending on earlier function. For example, the scheduler uses IPIs and RCU, so a CPU initializes its IPI and RCU handling before it starts scheduling processes. Given that the scheduler relies on IPIs and RCU, it makes no sense whatsoever for the CPU-hotplug offlining path to shut down IPIs and RCU before the scheduler. However, that is exactly what the current CPU-hotplug notifier operation encourages by invoking offline-time notifiers in the same order that it invokes online-time notifiers.

The only reasonable approach is to run the offline-time notifiers in the opposite order from online-/boot-time notifiers. With this approach, IPIs and RCU are notified before the scheduler when a CPU comes online, and the scheduler is notified before IPIs and RCU when a CPU goes offline. This means that the scheduler can count on IPIs and RCU being operational at all times.

In addition, this approach permits a CPU to be partially torn down to a well-defined checkpoint, for example, a CPU might be torn down to the point that all it can do is run in the idle loop, possibly permitting more aggressive power-efficiency measures to be brought to bear while providing improved CPU-online latency. Another example is partially tearing the CPU down to the point that it still handles interrupts, but does not run normal tasks, providing a form of CPU isolation.

There will of course be the occasional necessary evil of layering violations, but with this scheme such violations should be the exception rather than the rule.

3.4 Only Per-CPU Hotplug kthreads During CPU Hotplug

One feature of the current CPU-hotplug approach that has caused RCU much grief is the fact that interrupts are

disabled during the CPU_STARTING and CPU_DYING notifiers. This means that RCU cannot create or destroy kthreads at the logical time to do so, but must instead do so in one of the other notifiers, and handle either semi-crippled or semi-safe operations betweentimes.

On the other hand, it would be far worse to continue running arbitrary tasks during this time because those tasks would use facilities that had already been torn down. This is bad for the kernel's actuarial statistics.[2]

One solution to this problem is for the scheduler to run only special CPU-hotplug per-CPU tasks during CPU-hotplug processing. This allows the notifiers to make full use of the scheduler facilities when handling their kthreads, while preventing unwary normal tasks from straying onto a CPU that is only half working.

4 Alternative Approaches Considered

We considered several alternatives. The first alternative, continuing with existing CPU hotplug, was dispensed with in Section 2.

The second alternative was continuing with existing CPU-hotplug, but running the offline-time notifiers in reverse order. While this would be an improvement, it would do nothing to solve the kthread-parking problem.

The final alternative was dispensing with CPU hotplug completely in favor of things like cpusets and interrupt affinity. However, the most troublesome aspects of CPU hotplug are inherent in clearing all current and future work from a given CPU [3], so little is gained. In addition, CPU hotplug is still required for failing hardware.

5 Issues

Old-Style Interrupt Controllers Some interrupt controllers cannot be directed away from a given not-yet-offline CPU. If such an interrupt controller absolutely must be used, we propose interrupt trampolining as a workaround.

Scheduler Once RCU has marked a CPU as offline, it cannot safely do a context switch because the scheduler relies on RCU. This can be handled by splitting the RCU notifiers into an earlier-in-boot notifier that handles marking the CPU online or offline, and a later-in-boot notifier that manages RCU's kthreads.

Early-Boot kthreads RCU initializes itself and registers its notifiers during early boot, long before kthreads may be created. The implementations of RCU that require kthreads manually defer kthread creation until after the scheduler is running.

x86 MTRRs Updates to x86 memory type range registers (MTRRs) require that all hardware threads in a given core be quiesced. Although this might slow down hotplug for hyperthreaded x86 kernels it should be quite a bit faster than a full stop_machine().

Scanning Online CPUs One of the advantages of stop_machine() is that the final CPU-offline process appears atomic to any in-kernel code that is not hotplug-aware. Removing this atomicity means that each of the several hundred occurrances of for_each_online_cpu() (which iterates over all online CPUs) will need to be inspected and perhaps modified.

6 Summary and Conclusions

We expect that the new approach to CPU hotplug will reduce hotplug latencies to 5ms, making Linux more useful in both the real-time and battery-powered-embedded arenas.

Acknowledgements and Legal Statement

We thank Amit Kucheria, Peter Zijlstra, Srivatsa Bhat, and Steven Rostedt for many valuable and illuminating discussions. We are indebted to Dave Rusling and Jim Wasko for their support of this effort.

References

[1] GUITTOT, V. Cpu hotplug. Available: https://wiki.linaro.org/WorkingGroups/PowerManagement/Doc/Hotplug [Viewed April 20, 2012], February 2012.

[2] MCKENNEY, P. E. SMP and embedded real time. *Linux Journal*, 153 (January 2007), 52–57. Available: http://www.linuxjournal.com/article/9361 [Viewed May 31, 2007].

[3] MCKENNEY, P. E. The linaro connect scheduler minisummit. Report from the Q2 2012 Linaro Connect scheduler mini-summit on ARM's big.LITTLE architecture., February 2012.

[4] RUSSELL, R. Hotplug cpu toy for i386. Available: http://lwn.net/Articles/76667/ [Viewed April 25, 2012], March 2004.

[5] SEQUENT COMPUTER SYSTEMS, INC. tmp_ctl - multi-processor-control. http://oss.sgi.com/projects/numa/download/dynix, March 2001.

[2] That said, RCU currently must handle offline CPUs running through the scheduler on their way from their CPU_DYING notifiers to the idle loop. RCU's method of handling this issue is at best inelegant.