Work-In-Progress: Models and tools to detect Real-Time Scheduling Anomalies

Blandine Djika+, Frank Singhoff*, Alain Plantec*, and Georges Edouard Kouamou+

+Doctoral Research Unit for Engineering and Applications, University of Yaounde 1, B.P. 337, Cameroon
*Univ. Brest, Lab-STICC, CNRS, UMR 6285, F-29200 Brest, France
medjika@yahoo.fr, frank.singhoff@univ-brest.fr,
alain.plantec@univ-brest.fr, georges.kouamou@gmail.com

Abstract. This paper deals with scheduling anomalies in real-time systems. Scheduling anomalies jeopardize schedulability analysis made prior to execution. In this paper, we propose a model to specify conditions leading to scheduling anomalies. A scheduling anomaly is modeled as a set of constraints on the architecture. We use this model to detect scheduling anomalies by offline and online analysis. To validate our approach, we implemented the approach as an extension to Cheddar, a schedulability tool. We apply our approach to seven scheduling anomalies and we show that most of these anomalies can be successfully detected.

Keywords: Real-Time Scheduling · Scheduling Anomaly.

1 Introduction

This article focuses on scheduling anomalies in real-time systems. As defined in [1], *a scheduling anomaly* refers to a counter-intuitive phenomenon in which increasing the system resources or relaxing the application constraints can make the application unschedulable.

Real-time systems have functions that have to be run before a given deadline. To check that the deadlines will be respected, they are usually validated early, at design time for example. After such validation, a real-time system is said to be schedulable if all deadlines can be met. When a scheduling anomaly occurs at execution time, a deadline can be actually missed. As a consequence, early validation efforts may be jeopardized.

In this article, we propose a set of models to specify seven scheduling anomalies identified by the community. Each model is made of a set of constraints on the architecture and is used both at design time and at execution time to detect scheduling anomalies. To validate the proposed approach, we have implemented tools able to verify the constraints and actually detect scheduling anomalies. These tools are implemented in Cheddar, a real-time scheduling analysis toolset.

The remainder of the article presents background about scheduling anomalies (Section 2), our approach to detect scheduling anomalies (Section 3), the

experiments and preliminary results (Section 4), related works (Section 5) and a conclusion (Section 6).

2 Background

2.1 Scheduling anomalies

In the field of real-time systems, one of the first scheduling anomalies being identified was the Graham's anomaly [10, 1] and was defined as follow:

Definition 1. *If a task set is optimally scheduled on a multiprocessor with some priority assignments, a fixed number of processors, fixed execution times, and precedence constraints, then increasing the number of processors, reducing execution times, or weakening the precedence constraints can increase the schedule length [10].*

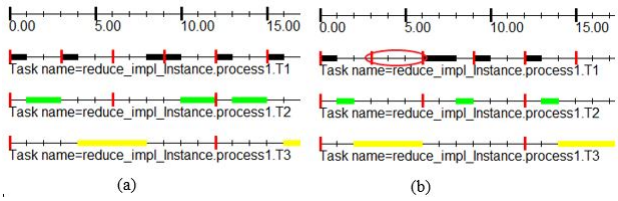

Fig. 1: Anomaly when reducing task execution time

Let us illustrate scheduling anomalies with an example from [15], composed of three periodic tasks scheduled with a non-preemptive fixed-priority scheduler. Each task i is defined by a 3-tuples with its WCET (worst case execution time) C_i, its period T_i and its fixed priority π_i: (C_1=1, T_1=3, π_1=1), (C_2=2, T_2=6, π_2=2) and (C_3=4, T_3=12, π_3=3). Figure 1 presents two schedules of such task set: (a) when the tasks are executed with their WCET, (b) when reducing the execution time of task 2 from 2 to 1, which makes the system unschedulable due to a scheduling anomaly.

2.2 Scheduling anomalies addressed in this article

After Graham's publication, several anomalies that may occur in various real-time systems were identified [13, 11, 12, 8, 3]. Such anomalies may occur in architectures with preemptive or non-preemptive scheduling policies, both uniprocessor and multiprocessor. Scheduling anomalies may occur with well studied scheduling policies such as preemptive uniprocessor EDF [16]. They may also occur in architectures frequently applied by practitioners such as multiprocessor partitioned systems [5].

In this article, we present our contribution to handle the most addressed seven types of scheduling anomalies by the community, namely: 1) Reducing the task execution time [10, 13, 11, 12, 16, 5, 14]; 2) Weakening the precedence constraints between tasks [10, 13]; 3) Increasing the period of the task [8, 3, 2]; 4) Changing priorities of tasks [10]; 5) Delaying the execution of the tasks [5]; 6) Increasing the number of processors of the execution platform [10]; 7) Increasing processor speed [5].

3 Approach

In this section, first we give examples of the constraints we modeled for each scheduling anomaly. Then, we explain how the analysis is conducted with such models.

3.1 Modeling scheduling anomaly conditions by constraints

The conditions leading to scheduling anomalies are described in the literature. Our approach consists in modeling each condition as constraints on the real-time system architecture. We have identified two types of complementary constraints: *static constraints* and *dynamic constraints*.

Static constraints are only related to the architecture specification and can be verified prior to execution. They are related to the task properties and the execution platform properties that are specified at design time. We identified 9 static constraints related to the execution platform and 8 related to the tasks (Tables 1 and 2).

Dynamic constraints depend on the system state during execution time: they are related to particular events that will actually cause scheduling anomalies (Table 3). Thus, checking dynamic constraints can only be done during execution.

These two kinds of constraints are complementary and the combination of a set of static constraints with a set of dynamic constraints defines a scenario that can lead to the detection of an anomaly during execution. In such a scenario, static constraints are necessary conditions to raise a scheduling anomaly but an anomaly only occurs when both static and dynamic conditions hold. Practically, detecting scheduling anomalies implies both offline and online verification.

3.2 Modeling each scheduling anomaly

As each scheduling anomaly can be raised in different scenarios defined by different conditions, detecting a scheduling anomaly requires verifying the constraints for each scenario. We have specified 17 constraints to cover the case of seven

Id	Name	Description
C1	Uniprocessor	Execution platform must be composed of one processor.
C2	Multiprocessor	Execution platform can be multiprocessor, i.e. containing several execution units such as cores.
C3	Partitioned	Tasks cannot migrate. Execution platform is partitioned.
C4	Global	A global multiprocessor scheduling policy is applied.
C5	FP	A fixed priority scheduling policy is applied.
C6	EDF	An EDF scheduling policy is applied.
C7	DM	Priorities are assigned according to Deadline Monotonic.
C8	Preemptive	A preemptive scheduling policy is applied.
C9	Non-Preemptive	A non-preemptive scheduling policy is applied.

Table 1: Static constraints on the execution platform

scheduling anomalies. The seven scheduling anomalies are specified by 19 scenarios of the 17 constraints (Table 4). As an example, to detect the *reducing task execution time* scheduling anomaly, we must verify seven scenarios defined with 15 constraints.

3.3 From the constraint model to the actual detection of anomalies

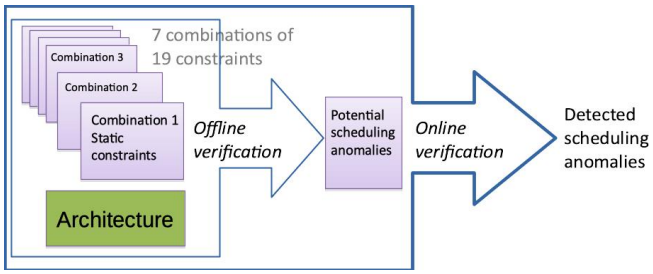

Fig. 2: Verification approach

We perform a two step verification process to detect scheduling anomalies, as shown in Figure 2. First, at design time, static constraints which can be

Id	Name	Description
C10	Synchronous release.	All tasks must have their first release at the same time.
C11	Asynchronous release.	There is no constraint on the first task release.
C12	Periodic tasks.	Tasks must be periodic.
C13	Independent tasks.	No task requires the execution of another to be released.
C14	Suspended tasks.	Tasks may be suspended during a bounded duration.
C15	Precedence constraints.	Task releases may occur on other task completion time.
C16	Shared resources.	Tasks share a resource with a protocol such as PCP.
C17	Aperiodic tasks.	Tasks must be aperiodic.

Table 2: Static constraints on the tasks

computed offline are verified. When static constraints are true it means that scheduling anomalies could occur. Otherwise, we can state that the scheduling anomalies resulting of the 19 scenarios cannot occur. In case potential scheduling anomalies could occur at runtime, we run the second step in which an online verification is applied. During runtime, we verify the dynamic constraints to actually detect scheduling anomalies.

4 Experiments

We explain now how we expect to validate the proposed method to detect scheduling anomalies. The proposed method requires the implementation of 2 different tools: (1) a first tool is needed to verify static constraints prior execution; while (2) a second tool must verify dynamic constraints during execution. In the following, first we describe a prototype of each tool. Then, preliminary evaluation results are given.

4.1 Implementation of the static constraints verification tool

To implement the first tool, we have extended Cheddar [17]. Cheddar is implemented in Ada. It provides a modelling language called CheddarADL and an analysis framework. CheddarADL allow the modelling of real-time systems by a set of concepts such as tasks, processors, or shared resources [7]. To implement the static constraint verification tool, no extension was needed to CheddarADL but we implemented into the Cheddar framework a new library of 17 functions. Each function verifies a static constraint of Tables 1 or 2. Figure 3 depicts the Ada implementation of constraint C10 and how it is used for the reducing execution time anomaly.

Id	Anomaly	Description
D1	Reducing execution time	Tasks may see their execution time decreasing during runtime.
D2	Increasing period	Task periods may increase during runtime.
D3	Increasing number of processors	Number of processors may increase during runtime.
D4	Changing priority	Task priorities may change during runtime.
D5	Weakening precedence constraint	Precedence constraints may be relaxed during runtime.
D6	Increasing processor speed	Processor speed may be increased during runtime.
D7	Delaying task execution	Tasks may suffer unexpected latency before being released.

Table 3: Dynamic constraints

4.2 A monitoring service to verify the dynamic constraints

The second step is the verification at runtime of the dynamic constraints of Table 3. By example, with the reducing execution time anomaly, we have to check that a task actually run quicker than expected, leading to a scheduling anomaly. For such a purpose, we need a monitoring service implemented in the operating system.

Figure 4 presents a possible interface of such monitoring service for an operating system compliant with POSIX. The main entry point of such interface is *scheduling_anomaly_handler* which is called by the operating system when an event which may lead to an anomaly occurs (see type *event_type*). This handler maintains a dynamic and a static view of the monitored system and raises an alarm (by the *output_param* parameter) when a scheduling anomaly is detected (see type *anomaly_type*).

4.3 Preliminary evaluation

Before implementing the monitoring service of Figure 4 in a real operating system, we prototyped it into the scheduling simulator of Cheddar. This scheduling simulator is able to simulate the behaviour of many task models and scheduling policies in the field of real-time scheduling. Figure 5 shows a sample of this implementation in Ada. We need 1600 lines of Ada code to implement in Cheddar the 17 static constraints with their 19 scenarios and the monitoring service which handles events generated by the Cheddar simulator to verify the dynamic constraints. The Ada implementation of the static and the dynamic constraints shares many existing Cheddar software units which reduces significantly the implementation effort.

With this implementation, we are able to verify that all scheduling anomalies are correctly detected by their static constraints. Dynamic constraints of five

```
function check_c10_synchronous_release
  (a : in system) return Boolean is
 iterator1    : tasks_iterator;
 task1, task0 : generic_task_ptr;
begin
 reset_iterator(a.tasks,iterator1);
 current_element(a.tasks,task0,iterator1);
  loop
   current_element(a.tasks,task1,iterator1);
    if (task1.start_time/=task0.start_time)
      then return False; ...

function check_static_reducing_execution_time
   (a : in system) return Boolean is
begin
 return
   (check_c1_mono_processor_system  (a) and
    check_c3_partitionned_scheduling(a) and
    check_c10_synchronous_release (a) and ...
```

Fig. 3: Specification of static constraints in Ada

```
enum  anomaly_type {
   task_execution_time_decrease,
   task_period_increase, ...

enum event_type {
   thread_context_switch,
   semaphore_or_mutex_lock, ...

typedef struct input_param {
   enum event_type an_event;
   struct timespec current_time; ...

typedef struct output_param {
   enum anomaly_type detected_anomaly;  ...

typedef struct cheddar_system { ...

int scheduling_anomaly_register(
   struct cheddar_system m);

int scheduling_anomaly_handler(
   struct input_param in,
   struct output_param* out); ...
```

Fig. 4: Monitoring API specification for dynamic constraints verification

Anomaly	**Dynamic Constraints**	**Static Constraints**	**Ref**
Reducing execution time	D1	C1,C3,C5,C10,C12,C15	[13]
		C1,C3,C5,C9,C10,C12,C13	[14]
		C1,C3,C7,C8,C10,C12,C16	[11]
		C1,C3,C6,C8,C11,C12,C13,C14	[16]
		C2,C4,C5,C8,C11,C12,C13,C17	[12]
		C2,C3,C5,C9,C10,C12,C16	[5]
		C2,C4,C5,C9,C10,C12,C15	[10]
Increasing period	D2	C2,C4,C7,C10,C12,C13	[8]
		C2,C4,C5,C8,C10,C12,C13	[2]
		C2,C4,C6,C8,C10,C12,C13	[3]
		C2,C3,C5,C8,C10,C12,C13	[3]
Increasing number of processor	D3	C2,C4,C5,C9,C10,C12,C15	[10]
Changing priority	D4	C1,C3,C5,C10,C12,C15	*
		C2,C4,C5,C9,C10,C12,C15	[10]
Weakening precedence constraint	D5	C1,C3,C5,C10,C12,C15	[13]
		C2,C4,C5,C9,C10,C12,C15	[10]
Increasing processor speed	D6	C1,C3,C5,C8,C11,C12,C16	[5]
		C1,C3,C5,C9,C11,C12,C13	[5]
Delaying execution time	D7	C1,C3,C5,C8,C10,C12,C16	[5]

Table 4: Modeling each scheduling anomaly. * *This scenario for D4 was identified by the authors during this work.*

anomalies were also verified. The verification of two dynamic constraints (D3 and D7) stays an ongoing work due to the number of updates needed in the Cheddar simulator. This evaluation was made by scheduling simulation of several task sets over their feasibility interval [9], which exhibits a schedulability proof for the tested task sets. Finally, a new scheduling anomaly scenario for D4 described in Table 4 has been identified during the experimentations.

5 Related work

Previous research on scheduling anomalies has generally focused on identifying and presenting different types of scheduling anomalies in both uniprocessor and multiprocessor systems [10, 14], but without mentioning how to prevent them practically.

As far as we know, only [6] and [2] have investigated scheduling anomalies prevention.

```
type handler_output_parameter is record ...
type handler_input_parameter is record ...

procedure scheduling_anomaly_register(
   m : in system);

procedure scheduling_anomaly_handler(
   inp   : in handler_input_parameter;
   sched : in generic_scheduler'class;
   outp  : out handler_output_parameter);
```

Fig. 5: Monitoring service implementation in Cheddar

Chen [6] looks for scheduling anomaly prevention when software components have to be ported from uniprocessor to multiprocessor environments. Chen introduces the notion of scheduler stability. This concept allows designers to prevent scheduling anomaly when tasks synchronize themselves during I/O operations.

Another approach was investigated by Andersson [2]. Andersson designed a partitionned multiprocessor scheduling approach that prevents scheduling anomaly providing that processor utilization stays under 41%.

Contrary to our work, none of them proposed and implemented tools to detect various scheduling anomalies.

6 Conclusion

Our work focuses on the modeling and the detection of scheduling anomalies in real-time systems. To detect a scheduling anomaly, we need to define the context in which the anomaly may occur. We then propose to specify the context of seven of the most known scheduling anomalies as a set of models composed of constraints. Each model is a set of constraints on the architecture allowing us to detect if an anomaly occurs at runtime. These constraints and the related verification tools were prototyped in Cheddar, a real-time schedulability tool. Through simulations on the feasibility interval, we have shown that our toolset is able to detect most of investigated anomalies. For future work, we will further explore scheduling anomalies analysis by the implementation of our proposal in an real operating system such as RTEMS [4].

7 Artefact

All programs and models described in this article are available at http://beru.univ-brest.fr/svn/CHEDDAR/trunk/artefacts/rtss21

References

1. L. Almeida, P. Pedreiras, and R. Marau. Traffic scheduling anomalies in temporal partitions. *LSE - IEETA / DET*, 2006.
2. B. Andersson. Static-priority scheduling on multiprocessors. *Chalmers University of Technology*, 2003.
3. B. Andersson and J. Jonson. Preemptive multiprocessor scheduling anomalies. *ARTES*, 2002.
4. G. Bloom and J. Sherill. Scheduling and thread management with rtems. In *ACM SIGBED Review*, 2014.
5. G.C. Buttazzo. *Hard Real-Time Computing Systems; Predictable Scheduling Algorithms and Applications*. Springer, 2011.
6. Y. Chen, L. Chang, T. Kuo, and A.K. Mok. An anomaly prevention approach for real-time task scheduling. *The Journal of Systems and Software*, 2009.
7. C. Fotsing, F. Singhoff, A. Plantec, V. Gaudel, S. Rubini, S. Li, H.N. Tran, L. Lemarchand, P. Dissaux, and J. Legrand. Cheddar architectute description language. Technical report, Lab-STICC technical report, 2014.
8. J. Goossens. Introduction à l'ordonnancement temps réel multiprocesseur. *Université Libre de Bruxelles*, 2007.
9. Joël Goossens, Emmanuel Grolleau, and Liliana Cucu-Grosjean. Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms. *Journal of Real-Time Systems*, 2016.
10. R. Graham. Bounds on the performance of scheduling algorithms. *Computer and job shop scheduling theory*, 1976.
11. S. Pailler. *Analyse Hors Ligne d'Ordonnançabilité d'Applications Temps Réel comportant des Tâches Conditionnelles et Sporadiques*. PhD thesis, Ecole Nationale Supérieure de Mécanique et d'Aérotechnique, 2006.
12. H. Rhan and W.S. Jane. Validating timing constraints in multiprocessor and distributed real time systems. *Proceedings of IEEE 14th International Conference on Distributed Computing Systems*, 1994.
13. M. Richard, P. Richard, E. Grolleau, and F. Cottet. Contraintes de précédences et ordonnancement mono-processeur. *Real-time and embedded systems (RTS'02)*, 2002.
14. P. Richard. On the complexity of scheduling real-time tasks with self-suspensions on one processor. 2003.
15. P. Richard, G. Phavorin, J. Goossens, T. Chapeaux, and C. Maiza. Scheduling with preemption delays anomalies and issues. *RTNS*, 2015.
16. F. Ridouard, P. Richard, F. Cottet, and K. Traoré. Some results on scheduling tasks with self-suspensions. *Journal of Embedded Computing*, 2006.
17. Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. Cheddar: A flexible real time scheduling framework. *ACM SIGAda Conference, Atlanta, USA*, 2004.