Carnegie Mellon University
Software Engineering Institute

A Tool for Satisfying Real-Time Requirements of Software Executing on ARINC 653 with Undocumented Multicore

Bjorn Andersson
Dionisio de Niz
Mark Klein

February 2023

SPECIAL REPORT
CMU/SEI-2023-SR-001

Software Solutions Division

http://www.sei.cmu.edu

Design: REV-03.18.2016.0 | Template: 01.04.2023

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM23-0135

Table of Contents

Abstract iii

1 Introduction 1

2 Background 3

2.1 Single-core ARINC 653 3
2.2 Multicore 4
2.3 Multi-core ARINC 653 5
2.4 The complexity of timing analysis and the need for schedulability test 5
2.4.1 Example 6
2.4.2 Why it is challenging? 6
2.4.3 Schedulability test 6
2.5 Assumptions 7
2.6 System model 8
2.7 Previously-published schedulability test 10

3 A Challenge in implementing Schedulability analysis 11

3.1 Idea how to evaluate the condition (11) 11
3.2 Found interval 13
3.3 Discard interval 13
3.4 Property of not found, not discarded interval 14
3.5 Split interval 14
3.6 Decide if splitting interval is promising 14
3.7 Idea how to evaluate the condition (11) and get smallest t 15
3.8 Algorithm to evaluate the condition (11) 16
3.9 Algorithm for evaluating the condition (10) 16

4 New Tool 17

5 Evaluation 18

6 Conclusions 19

References 20

Appendix A: Detailed Results of Experimental Evaluation 21

List of Figures

Figure 1. ARINC 653 single core 2
Figure 2. ARINC 653 multicore 4
Figure 3. The complexity of timing analysis and the need for schedulability test 5
Figure 4. Definition of reqlpARINC653(τ,Π,PART,i,t) 8
Figure 5. Illustration of a, b, fa, fb, ga, gb. 9
Figure 6. Algorithm for evaluating Condition (11) in schedulability test 12
Figure 7. Algorithm executing the schedulability test in Theorem 1 13
Figure 8. Our new tool that implements the schedulability test in Theorem 1. 17
Figure 9. The time required by our new schedulability test when varying parameters. 18

Abstract

ARINC 653 aims to simplify integration of independently developed (avionics) application software executing on a shared computer platform. A key idea is that the application software is organized as a set of partitions, potentially with different criticality levels, and the underlying operating system attempts to achieve certain isolation properties between the partitions. When using ARINC 653 on multicore, the existence of undocumented hardware is a challenge because it influences timing of software but we do not know exactly how the hardware works. A recent method (developed by us [8]) has addressed this. The main idea is to (i) describe the software system as a set of processes and describe each process with parameters, (ii) introduce an abstraction that describes the execution speed of a process as a function of co-runner processes on other processor cores, (iii) empirically find the numeric values of this abstraction, and (iv) use a formal verification technique (called schedulability test) that takes as input the description of processes and outputs a statement on whether all timing requirements will be satisfied at run-time for all scenarios assumed to be possible. Unfortunately, no software tool that implements this formal verification technique was available. Therefore, in this paper, we present such a software tool. As part of developing this tool, one challenge that we faced (and addressed) is that the schedulability test was formulated as a condition but in order to make this useful, we need to have an algorithm that can evaluate this condition and this turns out to be non-trivial. We also present an evaluation of this tool on randomly-generated tasksets; this evaluation shows that our tool runs reasonable fast when analyzing systems with 12 tasks and 8 processor cores.

1 Introduction

ARINC 653 aims to simplify integration of independently developed (avionics) application software executing on a shared computer platform [1,2]. A key idea is that the application software is organized as a set of partitions, potentially with different criticality levels, and the underlying operating system attempts to achieve certain isolation properties between the partitions. Originally, ARINC 653 was defined for the case that the shared computer platform had a processor chip with a single processor core. But today computers with multicore processors are becoming common; consequently ARINC 653 has been extended for computer platforms with multicore processors. This brings to the fore the question whether, and how well, techniques (run-time isolation and offline verification) originally developed to be used for ARINC 653 for single-core carry-over to multicore.

With single-core processor, satisfying real-time requirements of software on ARINC 653 was typically achieved through four activities. First, a time-triggered schedule (called module schedule) is repeated indefinitely; this provides a predictable supply of processing time to partitions. Second, each partition is assigned two numbers: period and duration. From the perspective of the application software in a partition, the period and duration can be viewed as a guarantee provided by the time-triggered schedule on the supply of processing time. From the perspective of the time-triggered schedule, the period and duration of a partition can be viewed as a requirement on how frequently the partition should receive service and how much. Third, given the period and duration of each partition, one can obtain a function that provides for each possible time interval, a lower bound on the amount of processing time supplied to a partition during this time interval. Fourth, with this lower bound, one can apply techniques that can prove satisfaction of real-time requirements (or disprove it) of a set of concurrent processes given a model of these processes; such techniques are called schedulability tests in the real-time systems research literature. This becomes challenging because of three reasons (i) in a partition, the number of processes can be greater than the number of processor cores so only some of them can execute at a given time, (ii) some processes are event-triggered so that the time when they request to execute are not known in advance, they are rarely requested but when they do, they have urgent need to execute (short deadline), and (iii) hardware resources shared between processor cores (e.g., cache memory, memory bus) can cause the execution speed of one process executing on one processor core to decrease because of execution of a process on another processor core. The last one becomes even more challenging when the multicore processor has undocumented hardware (i.e., the shared resources within the processor chip and within the memory system are not disclosed by the hardware maker).

The research literature has provided methods that can analyze ARINC 653 for single core [3,4,5,6]. The research literature has also provided methods that can analyze multicore systems with undocumented hardware but not for ARINC 653 [7]. Two years ago, there was no method that could analyze ARINC 653 on undocumented multicore. We, however, addressed this in a recently-published paper [8]; specifically, we developed a method that can analyze ARINC 653 on undocumented multicore. The main idea is to (i) describe the software system as a set of

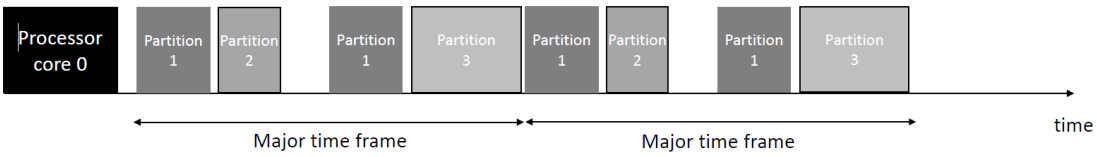


Figure 1. ARINC 653 single core

processes and describe each process with parameters, (ii) introduce an abstraction that describes the execution speed of a process as a function of co-runner processes on other processor cores, (iii) empirically find the numeric values of this abstraction, and (iv) use a formal verification technique (called schedulability test) that takes as input the description of processes and outputs a statement on whether all timing requirements will be satisfied at run-time for all scenarios assumed to be possible. This enables, under certain assumptions, satisfying real-time requirements of multicore software on ARINC 653 even with undocumented hardware. Unfortunately, no software tool that implements this formal verification technique was available [8].

Therefore, in this paper, we present a software tool that implements the method in [8]. As part of developing this tool, one challenge that we faced (and addressed) is that the schedulability test was formulated as a condition but in order to make this useful, we need to have an algorithm that can evaluate this condition and this turns out to be non-trivial. We also present an evaluation of this tool on randomly-generated tasksets; this evaluation shows that our tool runs reasonable fast when analyzing systems with 12 tasks and 8 processor cores.

Scope: This paper studies the problem of proving timing correctness of processes executing on ARINC 653 on undocumented multicore where configuration (e.g., generating the module schedule, assigning processes to processor cores) has already been done; how to select configuration is not the scope of this paper.

The remainder of this paper is organized as follows. Section 2 gives background. Section 3 presents a challenge in implementing the schedulability analysis. Section 4 presents our new tool. Section 5 presents performance evaluation of our new tool. Section 6 gives conclusions.

2 Background

2.1 Single-core ARINC 653

ARINC 653 [1,2] is a standard for avionics systems and emphasizes safety. This is a broad scope; in this paper, we focus only on real-time requirements aspects of ARINC 653.

Fig. 1 illustrates ARINC 653 for a single-core system. The software system is organized as a set of partitions. The scheduling of the single core is as follows. There is a schedule, called module schedule, whose length is called a major time frame; this schedule is repeated indefinitely. This module schedule contains partition time windows (ptws); a ptw is a time interval characterized by (i) an offset, (ii) a duration, and (iii) the partition that is served during this ptw. In Figure 1, there are four ptws in a major time frame: two ptws serve Partition 1; one ptw serves Partition 2; and one ptw serves Partition 3. A partition is also characterized by its period and its duration meaning that for time intervals of length equal to the period, it holds that the cumulative amount of time that ptws serve this partition in this time interval is greater than or equal to the duration of the partition. The ARINC 653 standard does not mandate a specific major time frame but it states (page 6 in [2]):

> The major time frame can be defined by a multiple of the least common multiple of all partition periods in the integrated module.

We will not make this assumption; our paper is more general.

A partition may consist of a set of processes. There are two special processes: the main process and the error handling process. The former deals with initialization of a partition and the latter deals with error handling. In this paper, we will ignore them. Instead, we will focus on processes that implement functionality of the application of a partition. Also, in this paper, we focus on the system when it has been initialized (i.e., in case the initialization has real-time requirements, then we ignore that in this paper). In addition, we ignore processing due to error handling in the error process.

In ARINC 653, a process is specified with its period, deadline (called "time capacity" in ARINC 653), whether it is periodic or aperiodic, and whether its deadline is hard or soft. In this paper, we focus only on periodic processes (i.e., we assume that there are no aperiodic processes) and we assume that all processes have hard real-time requirements. We make these assumptions because these are common in the academic literature on real-time scheduling and because these are the type of processes that tend to be safety critical.

To prove that processes meet deadlines, it is necessary to compute a lower bound on the amount of processing time that a module schedule provides to a partition. One way is to compute, for each partition p, this lower bound directly from the module schedule; we call this interpretation 0. Another way is to compute for each partition p, a period and duration for partition p such that this period and duration expresses this lower bound. This requires a careful definition of period and a

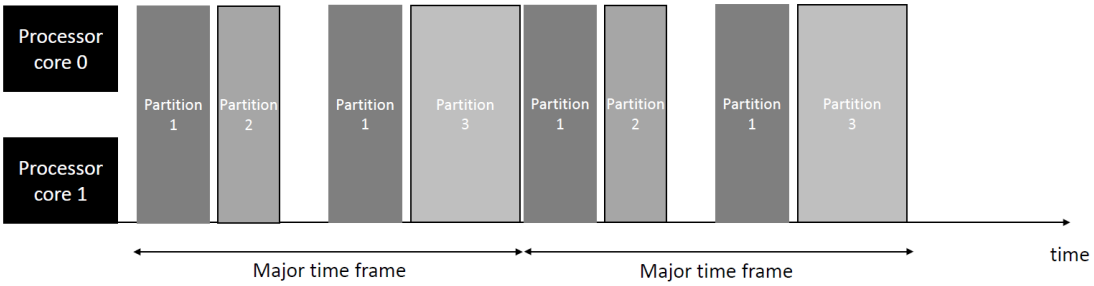


Figure 2. ARINC 653 multicore

duration of a partition and we found an ambiguity in the standard. We let interpretation 1 denote one meaning and let interpretation 2 denote another meaning—for details, see [8].

ARINC 653 uses fixed-priority preemptive scheduling meaning that each process is assigned, before run-time, a priority which is a number and at run-time whenever there is a need to select one process for execution (i.e., there is more than one process eligible for execution) the one with the highest priority among the eligible processes is selected for execution.

Note that in ARINC 653, a process i can only execute at time t if there is a ptw for which the following holds: (i) the ptw is active at time t, (ii) the ptw belongs to the partition of process i. This brings the advantage that ARINC 653 was designed for: isolation between partitions. But it also brings with it the following drawback. Consider a process that has remaining execution and is the highest-priority process at time t (and hence may have a short deadline requiring to execute urgently). This process may be prevented from executing because the ptw that is active at time t belong to another partition than the partition of this process.

2.2 Multicore

A multicore is a computer with many processors implemented on a single chip. Each of these processors is called a processor core. Hence, the computer can execute many processes simultaneously. We are interested in multicores that share memory because these are common in practice and these are the focus of ARINC 653. Typically, a multicore has a shared physical address space and there are various resources in the memory system (cache memories that store frequently accessed data items; write back buffers that store data that a program has written to but where the write has not yet taken effect in the main memory; hardware prefetch units that speculatively fetch data in anticipation of the data that a program needs to access).

An important consequence of these shared resources in the memory system is that they can cause inter-core interference; that is, the execution of one process on one processor core can influence the execution of another process on another processor core (typically, this influence is slowing down the execution of the other process). The amount of slowdown depends on the software, the hardware, and the specific scenario (the memory accesses and their interleaving). But there have been reports that the slowdown can be 103x and 300x [9,10] so we believe this effect deserves attention.

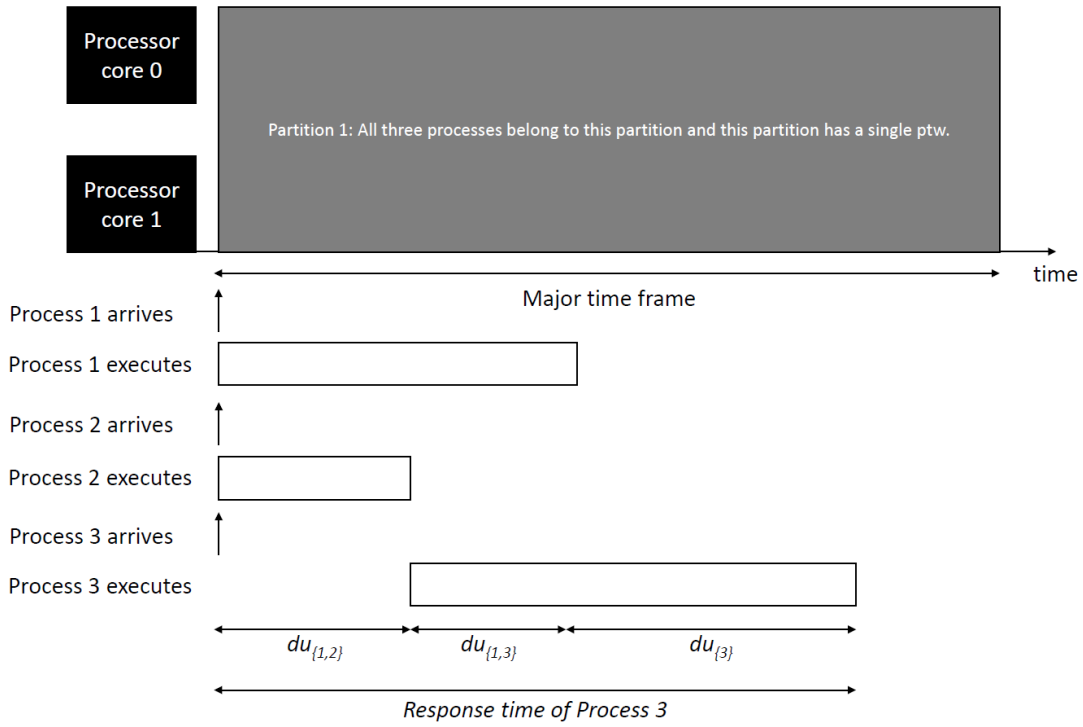


Figure 3. The complexity of timing analysis and the need for schedulability test

2.3 Multi-core ARINC 653

In recent years, ARINC 653 has been extended for multicores. The basic idea is that: (i) a partition is assigned a set of virtual processor cores, (ii) a process in the partition is assigned to a virtual core local to the partition, (iii) the partition also provides a mapping from the local virtual processor cores of the partition to physical processor cores, and (iv) at each instant, there is at most one partition active. We will assume this behavior in the rest of the paper; other ideas, however, have been considered for standardization (e.g., allowing processes to migrate between processor cores and allowing two or more partitions to active simultaneously) but we will not consider them in this paper.

Fig. 2 illustrates multicore ARINC 653.

2.4 The complexity of timing analysis and the need for schedulability test

ARINC 653 allows many processes in a single partition; this brings complexity to timing analysis and this complexity is even greater for multicore and undocumented multicore. To illustrate this, and to give a pre-view of our schedulability test, we show a simple example below.

2.4.1 Example

Consider a computer system with two processor cores and a single ARINC 653 partition and this partition has a single ptw whose duration is equal to the major time frame (that is, it is active all the time). Figure 3 shows it. Assume that there are three processes: Process 1, Process 2, and Process 3 so that (i) Process 1 is assigned to processor core 0 and Process 2 and 3 are assigned to processor core 1 and (ii) Process 3 has lower priority than Process 2. Suppose that we are interested in the response time of Process 3; that is, the time from when it arrives until it finished. The response time of Process 3 depends on (i) the time it has to wait until Process 2 (which has higher priority and is on the same processor) finished and (ii) the time that Process 1 executes in parallel with Process 3. In addition, the former depends on how much slowdown that Process 1 causes on Process 2. To simplify our discussion, assume that Process 1 does not experience inter-core interference and assume that Processor 2 does not experience inter-core interference. But assume that Process 3 executes with half speed at those times when it executes simultaneously with Processor 1.

2.4.2 Why it is challenging?

The example above shows one particular scenario: all processes arriving simultaneously and with given execution times of processes and this yields a response time for this particular execution of Process 3. But changing to another scenario (e.g., moving the arrival time of Process 3 later) leads to a different response time of Processor 3. Typically, execution times are bounded by worst-case execution time (WCET) estimates that are known but there is still a very large number of scenarios. So the problem that we are facing is: given certain bounds on the scenario (e.g., WCET of processes, description of slowdown of processes, bounds on when processes can arrive), find an upper bound on the response time of each process and compare it to the deadline of the process. Computing upper bounds on response times of processes is one form of schedulability test.

2.4.3 Schedulability test

Let us create a schedulabiity test for the aforementioned example and assume that we want to find the response time of Process 3. Let $du_{\{1,2\}}$ denote the cumulative duration of time that Process 1 and Process 2 executes simultaneously. Let $du_{\{1,3\}}$ denote the cumulative duration of time that Process 1 and Process 3 executes simultaneously. Let $du_{\{1\}}$ denote the cumulative duration of time that Process 1 executes while there is no execution on Processor core 1. Analogous for $du_{\{2\}}$ and $du_{\{3\}}$. Let C_1 denote the WCET of Process 1 for the case that it executes when the other processor core (process core 1) is idle. Assuming the execution speeds as mentioned in the example and using these notations, we obtain the following:

The response time of Process 3 is:

$$du_{\{1,2\}} + du_{\{2\}} + du_{\{1,3\}} + du_{\{3\}} \quad (1)$$

Since (from our assumption), Process 1 does not experience inter-core interference, it holds that:

$$du_{\{1\}} + du_{\{1,2\}} + du_{\{1,3\}} \leq C_1 \quad (2)$$

$$du_{\{1\}} + du_{\{1,2\}} + du_{\{1,3\}} \leq C_1 \tag{2}$$

For similar reason:

$$du_{\{2\}} + du_{\{1,2\}} \leq C_2 \tag{3}$$

Since (from our assumption), Process 3 executes with half speed at those times when it executes simultaneously with Process 1, it holds that:

$$du_{\{3\}} + 0.5 * du_{\{1,3\}} \leq C_3 \tag{4}$$

Clearly, durations must be non-negative. Hence:

$$0 \leq du_{\{1,2\}} \quad 0 \leq du_{\{1,3\}} \quad 0 \leq du_{\{1\}} \quad 0 \leq du_{\{2\}} \quad 0 \leq du_{\{3\}} \tag{5}$$

We can find an upper bound on the response time of Process 3 by solving the following optimization problem: Maximize (1) subject to the constraints (2),(3),(4),(5).

It can be seen here that (i) the response time of a process depends on its arrival and other processes, (ii) since a process can in general arrive many times, we are typically interested in an upper bound on the response time over all those requests, and (iii) to compute an upper bound on the response time of a process, we need to make assumptions. Therefore, in the next section, we will state assumptions that we make and the system model that we use. Then, we present the schedulability test from previous work [8].

2.5 Assumptions

We have already stated some assumptions that we make in this paper. In addition, we make the following assumptions:

A1. We assume that processes do not self-suspend. There are primitives in ARINC 653 that are potentially blocking (operations on counting semaphores, mutexes, and message passing). We assume that these are not used.

A2. We assume that processes do not request to execute non-preemptively. There is a primitive (called lock_preempt) in ARINC 653 and we assume that this is not used.

A3. We assume that the operating system uses some means to make sure that the speed of execution of one partition does not depend on execution of another partition. One way to achieve this (which is done in some operating systems) is to flush the cache and TLB after each ptw.

A4. We allow for the possibility that a process in one partition executing on one processor core can experience smaller execution speed (due to inter-core interference) from another process on another processor core if the other process is in the same partition. Indeed, our schedulability test takes this effect into account.

A5. We assume that each process is assigned to a processor core; i.e., it does not migrate between processor cores.

Maximize

$$\sum_{i\prime \in hep(\tau,\Pi,PART,i)} \sum_{v_{i\prime}^{k\prime} \in V_{i\prime}} \sum_{s \in S(\tau,\Pi,PART,i\prime,k\prime)} du_s \tag{6}$$

Subject to

$$\forall i' \in hep(\tau,\Pi,PART,i)\ \forall v_{i'}^{k'} \in V_{i'} \quad \sum_{s \in S(\tau,\Pi,PART,i\prime,k\prime)} pw_{i',s\backslash\langle i',k'\rangle}^{k\prime} * du_s \leq \left\lceil \frac{t}{T_{i\prime}} \right\rceil * C_{i\prime}^{k\prime} \tag{7}$$

$$\forall i' \in op(\tau,\Pi,PART,i)\ \forall v_{i'}^{k'} \in V_{i'} \quad \sum_{\substack{s \in S(\tau,\Pi,PART,i',k')\wedge \\ \left(\begin{matrix}\exists\langle i'',k''\rangle\,(i'' \in hep(\tau,\Pi,PART,i)) \\ \wedge\left(v_{i''}^{k''} \in V_{i''}\right)\wedge\left(v_{j''}^{k''} \in s\right)\end{matrix}\right)}} pw_{i',s\backslash\langle i',k'\rangle}^{k'} * du_s \leq xUB(\tau,\Pi,PART,i',k',t) \tag{8}$$

$$\forall s\ s.t.\left(\exists\langle i'',k''\rangle\ i'' \in hep(\tau,\Pi,PART,i)\ v_{j''}^{k''} \in s\right) \quad dur_s \geq 0 \tag{9}$$

Figure 4. Definition of reqIpARINC653(τ,Π,PART,i,t)

2.6 System model

We consider a set of processes τ and a computer platform Π. Let PART denote the set of partitions. A process $\tau_i \in \tau$ is characterized by the parameters T_i, D_i, $prio_i$, and $part_i$ with the interpretation that (i) T_i is the period of process τ_i; its meaning is the same as the meaning in ARINC 653 and (ii) D_i is the deadline of process τ_i; in ARINC 653, it is called "time capacity", (iii) $prio_i$ is the priority of process τ_i; its meaning is the same as the one in ARINC 653; and (iv) $part_i$ is the partition to which process τ_i is assigned. Let P_l denote the period of partition l and let Q_l denote the duration of partition l.

At run-time a process arrives periodically. Each time a process arrives, it clearly requests to execute and we call this requested execution a job. Hence a process generates a sequence of jobs. Each job has an arrival time and a deadline. The deadline of a job of process τ_i is Di time units after the arrival time of this job. If a job finishes by its deadline, then we say that the job meets its deadline. The arrival time of the kth job of process τ_i is T_i time units after the arrival time of the k-1^{th} job of process τ_i. We assume that for each process τ_i it holds that $D_i \leq T_i$ (this is called constrained-deadline periodic processes).

We use a model where each process is described by a set of segments. The reason for using this description is that it allows describing different parts of a program having different sensitivities with respect to how much slowdown they experience from other programs. Let V_i denote the set of segments of process τ_i. The interpretation is that the segments in V_i are ordered so that the k^{th} segments in V_i is called v_i^k. Note that, a process is permitted to have just a single segment; if so $|V_i|=1$ and v_i^1 is this single segment. A job of process τ_i executes its segments in order; that is, it executes segment v_i^1 first; then if there is a 2nd segment, it executes segment v_i^2; and so on.

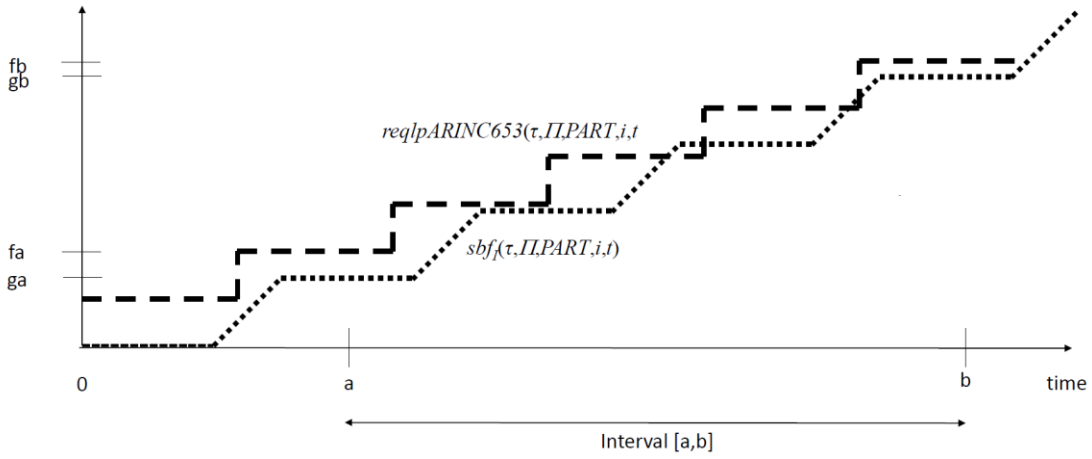


Figure 5. Illustration of a, b, fa, fb, ga, gb.

Eventually, when the last segment of the job has finished, the job finishes. Segment v_i^k is described with C_i^k; the meaning is that the execution requirement of v_i^k is in $[0, C_i^k]$; that is, C_i^k represents the worst-case execution requirement. C_i^k influences how long it takes for segment v_i^k to execute but the time it takes depends on its co-runners which can be different at different times.

We introduce a speed of execution of a segment v_i^k at a time given a schedule. If a schedule is given, one can identify which other segments executes at this time when v_i^k executes; we call these other segments co-runners. The execution speed of segment v_i^k given co-runners is not fixed so we describe it with a lower bound and an upper bound.

Let the lower bound on the execution speed of segment v_i^k given co-runners co be denoted $pw_{i,co}^k$ and let the upper bound on the execution speed of segment v_i^k given co-runner co be 1. The intuition behind this notation is that pw can be thought of as progress worst case and this can be thought of as meaning lower bound on the speed of execution. In order to represent the lower bound with parameters, we introduce CO_i^k as a set of tuples where each tuple $<cs_{i,h}^k, pcs_{i,h}^k>$ has the meaning that if v_i^k has the co-runner set $cs_{i,h}^k$ then $pw_{i,co}^k$ is $pc_{si,h}^k$. The segment v_i^k is also characterized by pd_i^k, meaning progress default; this provides a lower bound on the execution speed of segment v_i^k if such a bound is not provided by CO_i^k for a certain co.

A segment finishes when the number of units of execution it has completed is equal to its execution requirement. The completed number of units of execution of a segment in a time interval is the integral of the speed of execution of the segment over the time interval.

Note that we have not specified how many jobs a process generates; and we have not specified the arrival times of jobs (only that they are periodic but we did not specify the time when the first job of a process arrives); we did not specify the execution requirement of a segment (only specified an upper bound on it); we did not specify the speed of execution of a segment (only a lower and upper bound on it). Therefore, there are many possible schedules that the system can generate.

We say that a system (described by a set of processes τ, a computer platform Π, and partitions PART) is schedulable if for all schedules that it can generate, for all processes, for all jobs in this schedule, the job meets its deadline.

hep means higher-than-or-equal-priorities. Formally:

$$hep(\tau, \Pi, PART, i) = \left\{ i' | (part_{i'} = part_i) \bigwedge (proc_{i'} = proc_i) \bigwedge (prio_{i'} \geq prio_i) \right\}$$

We let op means other-processors. Formally,

$$op(\tau, \Pi, PART, i) = \left\{ i' | (part_{i'} = part_i) \bigwedge (proc_{i'} \neq proc_i) \right\}$$

We say that a segmentset s exactly-executes at time t if both of the following are true (i) each segment in s executes at time t and (ii) for each segment that executes at time t, the segment is in s. We say that a segment is assigned to the processor core to which the task that the segment belongs to is assigned.

Note that in order for it to be possible for a set of segments to exactly-execute, it must be that the segments in this set are assigned to different processor cores. Hence, there are some sets of segments that cannot exactly-execute.

Let $S(\tau,\Pi,PART,i,k)$ denote the set of segmentsets such that for each segmentset s in $S(\tau,\Pi,PART,i,k)$ it holds that (i) s can exactly-execute and (ii) v_i^k is in s.

When we analyze a process τ_i, we will consider the amount of execution of processes on other processor cores than the one that process τ_i executes on. Then, we will find xUB useful. Let $xUB(\tau,\Pi,PART,i',k',t)$ be an upper bound on number of units of execution performed by segment $v_{i'}^{k'}$ in a time interval of duration t assuming that no deadline miss occurs before the end of the time interval; formally:

$$xUB(\tau, \Pi, PART, i', k', t) = \left\lceil \frac{t + D_{i'}}{T_{i'}} \right\rceil * C_{i'}^{k'}$$

2.7 Previously-published schedulability test

Let $sbf_I(\tau,\Pi,PART,i,t)$ (meaning supply bound function) be a function that yields a lower bound on the cumulative duration of time that the partition to which task τ_i belongs to is served among all time intervals of duration t. Previous work provides expressions for this—see [8]—so it is not repeated here. With these defininitions and notations, we can express a schedulability test as follows:

Theorem 1 (from [8]): If $\forall \tau_i \in \tau \; \exists t \in [0,D_i]$ $reqlpARINC653(\tau,\Pi,PART,i,t) \leq sbf_I(\tau,\Pi,PART,i,t)$, then the system is schedulable.

Proof: See [8].

3 A Challenge in implementing Schedulability analysis

To actually use the schedulability analysis of Theorem 1, and to build a tool based on it, we need to evaluate the condition:

$$\forall \tau_i \in \tau \ \exists t \in [0, D_i] \ \text{reqlpARINC653}(\tau, \Pi, \text{PART}, i, t) \leq \text{sbf}_I(\tau, \Pi, \text{PART}, i, t) \quad (10)$$

Since this condition has two quantifiers (one universal and one existential) and since it involves real numbers and functions that are expressed on a form that are not closed-form expressions, it is not trivial to evaluate this condition. We will discuss how to do this now.

We can observe that:

O1. $\forall t \geq 0 \quad \sum_{\tau_i \in \tau_i} C_i^* \leq \text{reqlpARINC653}(\tau, \Pi, \text{PART}, i, t)$

This follows from the definition of reqlpARINC653.

O2. $\forall t \geq 0 \ \ \text{sbf}_I(\tau, \Pi, \text{PART}, i, t) \leq t$

This follows from the definition of $\text{sbf}_I(\tau, \Pi, \text{PART}, i, t)$—see ref [8].

O3. The universal quantifier in (10) is over a finite set.

O4. $\text{reqlpARINC653}(\tau, \Pi, \text{PART}, i, t)$ is non-decreasing with increasing t.
This can be seen from the definition of reqlpARINC653.

O5. $\text{sbf}_I(\tau, \Pi, \text{PART}, i, t)$ is non-decreasing with increasing t.

This can be seen in [8].

From O1 and O2, it follows that the range $[0, D_i]$ can be replaced by $[\sum_{\tau_i \in \tau_i} C_i^*, D_i]$ without changing the evaluated result of (10).

From O3, it follows that we can iterate over all tasks and for each task, evaluate the condition:

$$\forall \tau_i \in \tau \ \exists t \in [\sum_{\tau_i \in \tau_i} C_i^*, D_i] \ \text{reqlpARINC653}(\tau, \Pi, \text{PART}, i, t) \leq \text{sbf}_I(\tau, \Pi, \text{PART}, i, t) \quad (11)$$

If for each task τ_i it holds that (11) is true, then (10) is true and hence the taskset is schedulable. So we will now discuss how to evaluate (11).

3.1 Idea how to evaluate the condition (11)

In the area of real-time system, it is common to express a schedulaility analysis as: $\exists t \ f(t)=t$ and this can (if f is non-decreasing with increasing t) be solved using fixed-point iteration. But the expression (11) does not have this structure—its right-hand side is not t. Hence, we will seek another way.

```
function  compute_t_for_given_task(enable_find_smallest_t,i,τ,Π,PART)  returns  <Boolean,Real>
    a= ∑_{τ_k∈Γ_i} C_k^k b=D_i;tolerance=(b-a)/10000.0;found_sat=False;
     found_sought = False; val = -1
    fa,ga =  =  compute_f_and_g(i,a,τ,Π,PART)
    if fa<=ga then return True, a
    fb,gb =  compute_f_and_g(solver_to_use,i,b,τ,Π,PART)
    my_fg_queue = fg_queue()
    my_fg_queue.enqueue(found_sat,fg_queue_element(a,b,fa,fb,ga,gb))
    if fb<=gb then
      found_sat = True
      val = b
      my_fg_queue.sorting_entire_Q(found_sat)
      my_fg_queue.prune_large_a(val)
      if enable_find_smallest_t then
        if my_fg_queue.is_queue_non_empty() then
           found_sought = ((val-my_fg_queue.get_smallest_a_in_Q())<=tolerance)
        else                                        found_sought = True
      else
        found_sought = True
    if (not found_sought) then
      my_fg_queue = fg_queue()
      my_fg_queue.enqueue(found_sat,fg_queue_element(a,b,fa,fb,ga,gb))
      while (my_fg_queue.is_queue_non_empty()) and (not found_sought) do
        e = my_fg_queue.dequeue()
        if e.fa<=e.gb then
          m = (e.a+e.b)/2.0
          fm,gm =  compute_f_and_g(solver_to_use,i,m,  τ,Π,PART)
          my_fg_queue.enqueue(found_sat,fg_queue_element(e.a,m,  e.fa,fm,  e.ga,gm  ))
          my_fg_queue.enqueue(found_sat,fg_queue_element(  m,e.b,  fm,e.fb,gm,  e.gb))
          if fm<=gm then
            if found_sat then
              val = min(val,m)
            else
              found_sat = True
              val = m
              my_fg_queue.sorting_entire_Q(found_sat)
            my_fg_queue.prune_large_a(val)
            if enable_find_smallest_t then
              if my_fg_queue.is_queue_non_empty() then
                 found_sought = ((val-my_fg_queue.get_smallest_a_in_Q())<=tolerance)
              else                                        found_sought = True
            else
              found_sought = True
    return found_sought, val
```

Figure 6. Algorithm for evaluating Condition (11) in schedulability test

Our idea on how to evaluate (11) is as follows: We seek a value of $t \in [\sum_{\tau_k \in \Gamma_i} C_k^k, D_i]$ such that reqlpARINC653($\tau,\Pi,$PART,i,t) $\leq$ sbf$_l$($\tau,\Pi,$PART,i,t). We will do this through branch-and-bound on the interval $[\sum_{\tau_k \in \Gamma_i} C_k^k, D_i]$ that is, we will divide the interval $[\sum_{\tau_k \in \Gamma_i} C_k^k, D_i]$ into subinterval and explore the subintervals.

We will define rules for this exploration. These rules pertain to an interval [a,b] and tells us what we can do with it; here we assume that [a,b] is a subset of [,Di] because we will perform the exploration for only such intervals. One rule tells us that we have found an interval [a,b] such that in this interval, there is a t that satisfies (11). Another rule tells us that an interval [a,b] cannot contain a value of t that we seek. Another rule tells us how to split an interval [a,b]. Yet another rule tells us whether an interval [a,b] is promising for further search (i.e., it is promising to split this into subintervals for further exploration).

When searching through these intervals, the question we ask is whether:

$$\exists t \in [a,b] \text{ reqlpARINC653}(\tau,\Pi,\text{PART},i,t) \leq \text{sbf}_l(\tau,\Pi,\text{PART},i,t) \qquad (12)$$

```
function     do_sched_test(terminate_on_first_unschedulable_task,enable_find_smallest_t,τ,Π,PART)
return <Boolean,array of Real>
  success = True;  i = 1;  terminate_schedulabilitytesting = False
  while ((i<=ntasks) and (not terminate_schedulabilitytesting)) do
    flag, tᵢ = compute_t_for_given_task(solver_to_use,enable_find_smallest_t,i,,τ,Π,PART)
    success = success and flag
    terminate_schedulabilitytesting = (terminate_on_first_unschedulable_task and (not success))
    i = i +1
  if (success) then return True, t
  else              return False, []
```

Figure 7. Algorithm executing the schedulability test in Theorem 1

because if this is true, then (11) is true as well (follows from the fact that [a,b] is within [$\sum_{i \in \forall_i} c_i^t$,D_i]).

Evaluation of reqlpARINC653(τ,Π,PART,i,t) and sbf_I(τ,Π,PART,i,t) for a given t can be expensive so later on, we will store these for specific values of t. Let fa denote evaluation of reqlpARINC653(τ,Π,PART,i,t) for t=a and let ga denote evaluation of sbf_I(τ,Π,PART,i,t) for t=b. Analogous for fb and gb. Figure 5 illustrates this.

Clearly:

$a \leq b$

O4 and O5 (which express monotonicity), yields:

$\forall t \in [a,b]$ fa $\leq$ reqlpARINC653(τ,Π,PART,i,t) $\leq$ fb (13)

$\forall t \in [a,b]$ ga $\leq sbf_I(\tau,\Pi,PART,i,t) \leq$ gb (14)

Combining (13) and (14) yields:

$\forall t \in [a,b]$ fa-gb $\leq$ reqlpARINC653(τ,Π,PART,i,t) - sbf_I(τ,Π,PART,i,t) (15)

From the definition of fa, fb, ga, gb, we obtain:

$\exists t \in [a,b]$ reqlpARINC653(τ,Π,PART,i,t) - sbf_I(τ,Π,PART,i,t) $\leq$ min(fa-ga, fb-gb) (16)

Found interval

3.2 Found interval

From the definition of fa, fb, ga, gb, (11), (12), and the fact that satisfying (12) implies satisfying (11), we obtain:

$fa \leq ga \Rightarrow$ (12) is true $\Rightarrow$ (11) is true (17)

$fb \leq gb \Rightarrow$ (12) is true $\Rightarrow$ (11) is true (18)

Hence, these can be used to declare that we have found an interval that we seek.

3.3 Discard interval

Suppose that fa>gb. Then, applying (13) on its left-hand side and applying (14) on its right-hand side yields:

$\text{reqlpARINC653}(\tau,\Pi,\text{PART},i,t) > \text{sbf}_I(\tau,\Pi,\text{PART},i,t)$

We can repeat this for any t and this yields that:

$fa>gb \Rightarrow \forall t\in[a,b]\ \text{reqlpARINC653}(\tau,\Pi,\text{PART},i,t) > \text{sbf}_I(\tau,\Pi,\text{PART},i,t)$

Hence:

$fa>gb \Rightarrow$ (12) is false (19)

Note that fa>gb does not imply that (11) is false.

3.4 Property of not found, not discarded interval

For an interval [a,b], for which we have not found what we sought ((17) and (18)), it holds that:

$fa>ga$ (20)

$fb>gb$ (21)

For an interval [a,b] which has not been discarded (19), it holds that:

$fa\leq gb$ (22)

Combining them yields that for an interval [a,b] for which we have not found what we sought and for which has not been discarded, it holds that:

$ga<fa\leq gb<fb$ (23)

Note that this ensures that for these intervals, there is variation of $\text{reqlpARINC653}(\tau,\Pi,\text{PART},i,t)$ within the interval and there is variation of $\text{sbf}_I(\tau,\Pi,\text{PART},i,t)$ within the interval. For this reason, there is no need to check whether $\text{reqlpARINC653}(\tau,\Pi,\text{PART},i,t)$ remains constant in the interval or check whether $\text{sbf}_I(\tau,\Pi,\text{PART},i,t)$ remains constant in the interval.

3.5 Split interval

For an interval [a,b], we can choose an intermediate point and split the interval. We choose the intermediate point as m=(a+b)/2 and then split the subinterval [a,b] into [a,m] and [m,b]. Then, compute reqlpARINC653 and sbf_I for these subintervals; this yields f and g for endpoints of these subintervals. Note that we already know f and g for the endpoints of [a,b]. Then evaluating f and g for m, yields f and g for the endpoints of [a,m] and [m,b]. Hence, splitting a subinterval requires just one evaluation of f and one evaluation of g.

3.6 Decide if splitting interval is promising

Let h(t) denote $\text{reqlpARINC653}(\tau,\Pi,\text{PART},i,t) - \text{sbf}_I(\tau,\Pi,\text{PART},i,t)$. Applying this on (15):

$\forall t\in[a,b]\ fa-gb \leq h(t)$ (24)

Applying this on (16):

$\exists t \in [a,b]\ h(t) \leq \min(fa-ga, fb-gb)$ (25)

When considering an interval [a,b] with knowledge of fa, fb, ga, gb, and asking whether there is a $t \in [a,b]$ such that $reqlpARINC653(\tau, \Pi, PART, i, t) \leq sbf_I(\tau, \Pi, PART, i, t)$ we view our goal as that of finding t such that $h(t) \leq 0$. The right-hand side of (25) gives us the smallest value of h(t) that we are aware of so far if we have not divided [a,b] into subintervals yet. The left-hand side of (24) gives us a lower bound on how low we can possibly get h(t) after dividing [a,b] into subintervals. Subtracting one of these from the other, yields:

$fa-gb-\min(fa-ga, fb-gb)$ (26)

Applying reasoning with (24) and (25) on (26) yields:

$fa-gb-\min(fa-ga, fb-gb) \leq 0$ (27)

If the expression (26) is equal to zero, then it is impossible to find a t that yields a smaller h(t) in [a,b] than what we have already found. So there is no need to divide [a,b] into subintervals. If the expression (26) is strictly negative, then the smaller (26) is the more potential we might have to find a t with smaller h(t). Therefore, we call (26) the feasibility-key for [a,b]. When doing the exploration of subintervals, we maintain a queue of subintervals that we have not yet explored; we implement this queue as a list and these subintervals are sorted in ascending order of their feasibility key.

Based on the definition of $reqlpARINC653(\tau, \Pi, PART, i, t)$ and $sbfI(\tau, \Pi, PART, i, t)$, it holds that these are piece-wide linear functions and there is a finite number of points where the slope changes. However, we would like our algorithm for evaluating (11) to be robust so that it can handle other functions (to allow new schedulability analysis that has lower pessimism in the future) and therefore, we would like to guarantee termination of the computations in our branch-and-bound procedure even if there is not a finite number of points where slope changes. Therefore, we introduce a rule so that if the interval [a,b] is very small, then we do not continue searching. For this reason, let tolerance be a number such that when given an interval [a,b], if b-a<tolerance, then we deem the interval to be so small that we do not search further. This means that there may exist tasksets such that if we continued searching, we might have been able to determine that the taskset is schedulable; so this introduce a small amount of pessimism. Note, however, that it is always safe in the sense that if our procedure determines that (11) is true, then (11) is indeed true. We use $tolerance=(D_i - \sum_{\tau_k \in \tau_i} C_i^k)/10000$.

3.7 Idea how to evaluate the condition (11) and get smallest t

In the previous section, we have seen how to search efficiently for a t such that (11) is true (if such a t exists). This is enough for such evaluation of (11) to be used to evaluate (10) and this makes it possible to evaluate the condition used as schedulability test. But the meaning of t is actually an upper bound on the response time of a task (for the value of i that we consider). In some cases, we may want to know that and report to the user. In this case, when a t exists, we want to not just report a t but actually report the smallest t which satisfies (11). Hence, we would like to change the aforementioned procedure so that it produces the smallest t that satisfies (11) rather

than just a t. We can do this as follows. When we have found a value t such that (11) is true, then do the following:

1. If [a*,b*] is the subinterval for which this t is found, then remove all subintervals with larger starting; that is remove all subintervals [a,b] such that a*<a.

2. For the subintervals, that remain, change the sorting of the list of subintervals so that instead of having key of an interval [a,b] being the feasibility key, it is the beginning of the interval (that is a).

3. Continue the exploration (but recall that now the key used is not the feasibility key anymore).

4. With this continuing exploration, if we find an interval [a,b] with t that satisfies (11), we can stop the search because then we know that among the t that satisfies (11), this is the t that is the smallest.

3.8 Algorithm to evaluate the condition (11)

Using these ideas, we obtain an algorithm—shown in Figure 6—which evaluates the condition (11). It can be configured so that it evaluates condition (11) without being concerned about getting the smallest t. But it can also be configured to find the smallest t. The variable enable_find_smallest_t determines this

3.9 Algorithm for evaluating the condition (10)

We can now simply iterate over all tasks and for each task, call the algorithm that evaluates the condition (11). Figure 7 shows this algorithm. In some cases, we only want to determine if a taskset is schedulable. In other cases, we also want to obtain t for as many tasks as possible even if the taskset is unschedulable. Therefore, we have a variable (that is Boolean) terminate_on_first_unschedulable_task that indicates this; if terminate_on_first_unschedulable_task=True, then it means that if for a task τ_i, we find that there is no t that satisfies (11), then we terminate the schedulability test and report unschedulable.

4 New Tool

We have developed a tool based on these aforementioned ideas. The tool has a graphical user-interface and it written in Python 3 and is available at [11]. Figure 8 shows it.

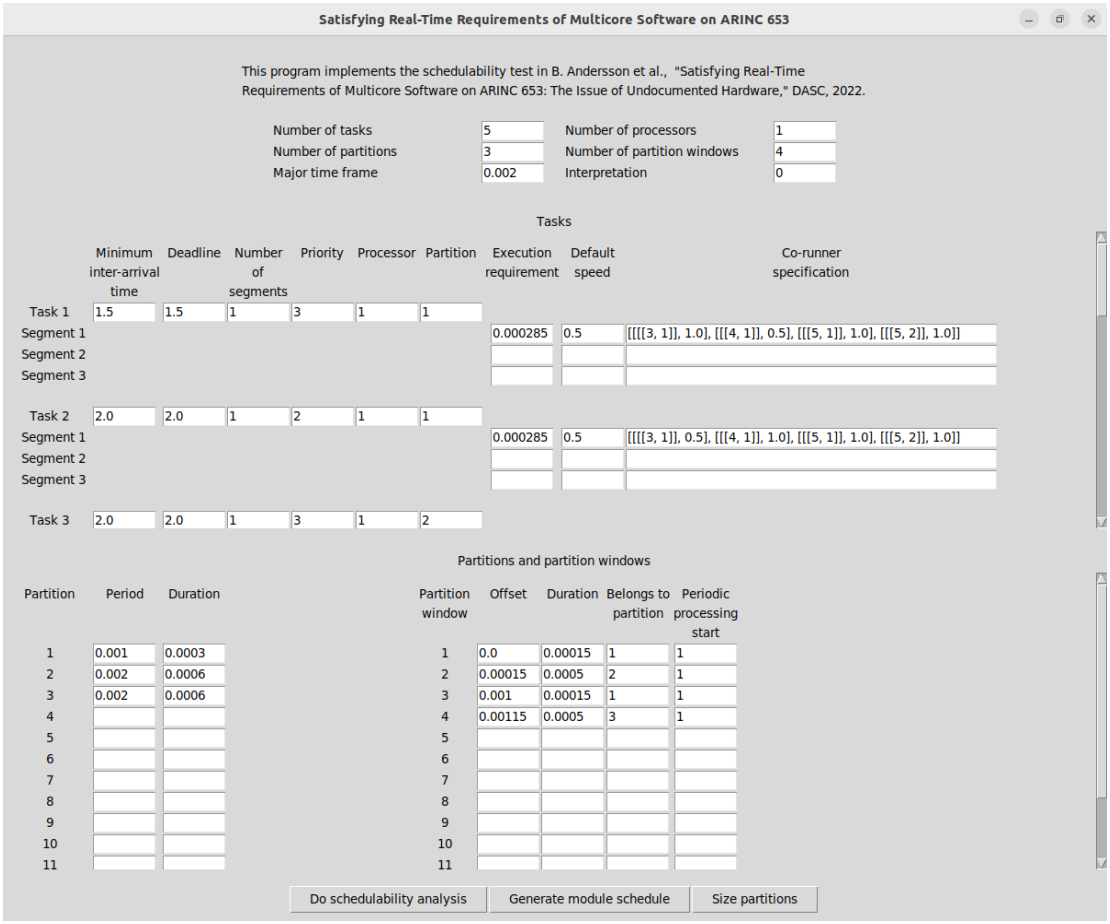
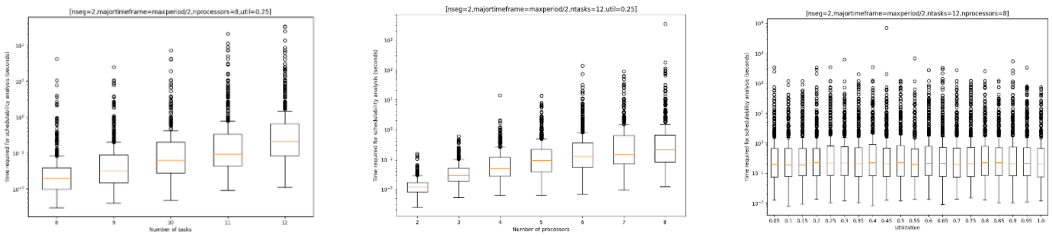


Figure 8. Our new tool that implements the schedulability test in Theorem 1.

5 Evaluation

We have conducted an extensive evaluation of our new tool on randomly-generated tasksets. Full details are available in Appendix A; here we only provide an overview.

Figure 9 shows part of the experimental results. Figure 9(a) shows the time it takes to run our schedulability analysis when varying number of tasks; Figure 9(b) for the case of varying number of processors; and Figure 9(c) for the case of varying utilization. It can be seen that time required grows rapidly with the number of processors but it does not grow rapidly when varying number of tasks and utilization; this is expected given that the cardinality of the set S grows exponentially with the number of processors.



(a) *(b)* *(c)*

Figure 9. The time required by our new schedulability test when varying parameters.

6 Conclusions

We have presented a new schedulability test for ARINC 653 on undocumented multicore. Our schedulability test is sufficient but not exact. We left open the question how to decrease pessimism.

References

[1] AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE, PART 0 VERVIEW OF ARINC 653, 2015.

[2] AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE, PART 1 REQUIRED SERVICES, 2015.

[3] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal, "A Compositional Scheduling Framework for Digital Avionics Systems," RTCSA, 2009..

[4] Y.-H. Lee, D. Kim, M. Younis, and J. Zhou, "Scheduling tool and algorithm for integrated modular avionics systems," DASC, 2000.

[5] A. Mok, D.-C. Tsou, and R. de Rooij, "The MSP.RTL real-time scheduler synthesis tool," RTSS, 1996.

[6] N. C. Audsley and A. J. Wellings, "Analysing APEX applications," RTSS, 1996.

[7] B. Andersson, H. Kim, D. de Niz, M. Klein, R. Rajkumar, and J. Lehoczky, "Schedulability Analysis of Tasks with Co-Runner-Dependent Execution Times," TECS, 2018.

[8] B. Andersson, D. de Niz, and M. Klein, "Satisfying Real-Time Requirements of Multicore Software on ARINC 653: The Issue of Undocumented Hardware," DASC 2022.

[9] H. Yun and P. K. Valsan, "Evaluating the Isolation Effect of Cache Partitioning on COTS Multicore Processors," OSPERT, 2015.

[10] M. G. Bechtel and H. Yun, "Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention," RTCSA 2019.

[11] https://www.andrew.cmu.edu/user/banderss/software/pyschedanalysiscorunnerarinc653/pyschedanalysiscorunnerarinc653.py

Appendix A: Detailed Results of Experimental Evaluation

Extensive experimental results are shown below.