Proceedings of the

10th Junior Researcher Workshop on Real-Time Computing

JRWRTC 2016

http://jrwrtc2016.gforge.inria.fr/

Brest, France
October 19-21, 2016
Message from the Workshop Chairs

Welcome to the 10th Junior Researcher Workshop on Real-Time Computing, held in conjunction with the 24th International Conference on Real-Time and Network Systems (RTNS) in Brest, October 2016. The workshop provides an informal environment for junior researchers, where they can present their ongoing work in a relaxed forum and engage in enriching discussions with other members of the real-time systems community.

Organizing this workshop would not have been possible without the help of many people. First, we would like to thank Alain Plantec and Frank Singhoff, General Chairs of RTNS 2016 for their guidance. We would also like to thank the local organizing committee, Mickaël Kerboeuf, Laurent Lemarchand, Steven Costiou, Stephane Rubini, Jalil Boukhobza, Nam Tran Hai, Arezki Laga, Hamza Ouarnoughi, Mourad Dridi and Rahma Bouaziz for having put their time in ensuring that all the details were smooth. We would also like to thank Sébastien Faucou and Luis Miguel Pinho, Program Chairs of RTNS 2016, for their scientific work. We finally acknowledge Benjamin Lesage (University of York) for his precious advices in the organisation of the event.

Behind the organization and the realization of the scientific program there is the careful work of reviewers, who provided their time free of charge and with dedication and devotion. We would like to express them our gratitude. Their work allowed us to put together a high-quality scientific program. Finally, we would like to thank all the authors for their submissions to the workshop. We wish you success in your scientific careers and we hope that the workshop will help you develop your ideas further.

On behalf of the Program Committee, we wish you a pleasant workshop. May the environment be stimulating, with fruitful discussions and the presentation be enjoyable and entertaining.

Antoine Bertout (Inria Paris, France) and Martina Maggio (University of Lund, Sweden)
JRWRTC 2016 Workshop Chairs
# Program Committee

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pontus Ekberg</td>
<td>Uppsala University, Sweden</td>
</tr>
<tr>
<td>Fabrice Guet</td>
<td>ONERA, France</td>
</tr>
<tr>
<td>Tomasz Kloda</td>
<td>Inria Paris, France</td>
</tr>
<tr>
<td>Angeliki Kritikakou</td>
<td>University of Rennes 1, IRISA/INRIA, France</td>
</tr>
<tr>
<td>Meng Liu</td>
<td>Mälardalen University, Sweden</td>
</tr>
<tr>
<td>Cong Liu</td>
<td>University of Texas at Dallas, USA</td>
</tr>
<tr>
<td>Renato Mancuso</td>
<td>University of Illinois at Urbana-Champaign, USA</td>
</tr>
<tr>
<td>Alessandra Melani</td>
<td>Scuola Superiore Sant’Anna, Italia</td>
</tr>
<tr>
<td>Geoffrey Nelissen</td>
<td>CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal</td>
</tr>
<tr>
<td>Sophie Quinton</td>
<td>Inria Grenoble, France</td>
</tr>
<tr>
<td>Hamza Rihani</td>
<td>University Grenoble Alpes, France</td>
</tr>
<tr>
<td>Youcheng Sun</td>
<td>University of Oxford, United Kingdom</td>
</tr>
<tr>
<td>Houssam-Eddine Zahaf</td>
<td>University of Lille, France / University of Oran1, Algeria</td>
</tr>
</tbody>
</table>
# Table of Contents

## Contents

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message from the Workshop Chairs</td>
<td>iii</td>
</tr>
<tr>
<td>Program Committee</td>
<td>iv</td>
</tr>
<tr>
<td>Model-Driven Development of Real-Time Applications based on MARTOP and XML</td>
<td>1</td>
</tr>
<tr>
<td>Andreas Stahlhofen and Dieter Zöbel</td>
<td></td>
</tr>
<tr>
<td>Model Checking of Cache for WCET Analysis Refinement</td>
<td>5</td>
</tr>
<tr>
<td>Valentin Touzeau, Claire Maïza and David Monniaux</td>
<td></td>
</tr>
<tr>
<td>On Scheduling Sporadic Non-Preemptive Tasks with Arbitrary Deadline using Non-Work-Conserving Scheduling</td>
<td>9</td>
</tr>
<tr>
<td>Homa Izadi and Mitra Nasri</td>
<td></td>
</tr>
<tr>
<td>Tester Partitioning and Synchronization Algorithm For Testing Real-Time Distributed Systems</td>
<td>13</td>
</tr>
<tr>
<td>Deepak Pal and Juri Vain</td>
<td></td>
</tr>
<tr>
<td>Framework to Generate and Validate Embedded Decision Trees with Missing Data</td>
<td>17</td>
</tr>
<tr>
<td>Arwa Khannoussi, Catherine Dezan and Patrick Meyer</td>
<td></td>
</tr>
<tr>
<td>Quantifying the Flexibility of Real-Time Systems</td>
<td>21</td>
</tr>
<tr>
<td>Rafik Henia, Alain Girault, Christophe Prérot, Sophie Quinton and Laurent Rioux</td>
<td></td>
</tr>
<tr>
<td>Towards schedulability analysis of real-time systems with precedence constraints and different periods</td>
<td>25</td>
</tr>
<tr>
<td>Slim Ben-Amor, Liliana Cucu-Grosjean and Dorin Maxim</td>
<td></td>
</tr>
</tbody>
</table>
Model-Driven Development of Real-Time Applications based on MARTOP and XML

Andreas Stahlhofen
Research Group for Real-Time Systems
Institute for Computer Science
University of Koblenz-Landau
Koblenz, Germany
astahlhofen@uni-koblenz.de

Dieter Zöbel
Research Group for Real-Time Systems
Institute for Computer Science
University of Koblenz-Landau
Koblenz, Germany
zoebel@uni-koblenz.de

ABSTRACT
Nowadays, there exists still a growing disaffection by the real-time community with the perceived gap between real-time scheduling methods on the one hand and the lack of their availability when implementing real-time applications on the other hand. This paper is one of those efforts trying to overcome this annoying gap. Following a model-driven approach, this paper presents the implementation of a XML schema which aggregates the major features of a certain real-time application at the descriptive abstraction of a specification. This schema is model-based in the sense that it reflects all notational aspects which are common to the real-time community. At the same time the XML schema allows for the generation of verified code fragments which control the timeliness of the tasks building up the real-time application. For reasons of portability these code fragments are POSIX-compliant.

Keywords
real-time system development; real-time system modeling; scheduling; model-driven development

1. INTRODUCTION

1.1 Motivation
At the present time real-time systems are pervasive and play a significant role in many applications. Examples can be found in the field of avionics, robotics, factory automation, medical, or in everyday life such as in the automotive sector or the field of mobile communications and application development of smartphones. Allowing for this fact, one would expect that the development process for systems of this nature is well-specified and straightforward. Taking a glance at the current practice, this expectation, however, can not be confirmed. Developers with the intention to implement a real-time application are in most cases at a complete loss and often start to reinvent the wheel by beginning with their implementation from scratch.

1.2 Related Work
In general, the established approach of model-driven development is well-studied and used in many areas of software development and engineering [13]. One of the most prominent modeling language in this field is the Unified Modeling Language (UML) [11]. There exists a mass of modeling tools including features like model checking and automatic code generation especially in the Java and C/C++ domain. Pure UML, however, does not support a notation for an appropriate specification of timing constraints. Although there are extensions available which integrate this missing feature, e.g. [5], in our opinion UML as a language is too com-
plex and does not reflect the models and notations from the real-time theory. Available studies also show that UML is hardly used in many software companies [10, 1]. A more intuitive way for specifying particularly timing constraints and scheduling policy of a real-time application is necessary.

Another alternative to model the timing constraints of a real-time application is based on timed automata and was first introduced by Alur and Dill in [2]. The main goal of this approach is the verification of the system behaviour according to a given specification and based on model-checking algorithm from the field of formal methods. A common tool with graphical user interface is UPPAAL [8] which can be used for modeling and verifying real-time application specifications based on timed automata[4]. But the modeling process is a complex task and requires a great knowledge in the field of formal methods and reachability analysis. Thus, this approach yields no benefit for our purpose because we want to provide a familiar notation in accordance to real-time theory.

1.3 Outline of this paper

The remainder of this paper is organized as follows. In section 2.1 we describe the underlying task model and the necessary notations which are important for understanding the rest of the paper. Chapter 2.2 introduces an example application which is used to describe our model-driven approach. The XML schema to model real-time applications is described in chapter 2.3. The features of scheduling analysis and code generation are introduced in chapters 2.4 and 2.5. Finally, the results of our work are discussed in chapter 3 and an outlook to planned future work is given.

2. MODEL-DRIVEN APPROACH

The overall process of the presented model-driven approach is illustrated in Figure 1 and is divided into an intellectual and a semi-automated process. The intellectual part is the one where the developer has to lend a hand whereas the semi-automated process is tool-supported. The core component is the XML specification according to a respective XML schema.

2.1 Task model and notations

The following assumes a set \( T \) of independent periodic preemptive tasks which are scheduled on a single core CPU. A given task \( T_i \) is specified by a tuple of the form \((C_i, T_i, A_i, D_i)\) where \( C_i \) is the worst case execution time, \( T_i \) the rate, \( A_i \) the relative arrival time and \( D_i \) the relative deadline of the task.

A schedule for a given set of tasks \( T \) is a function \( N \rightarrow T \) which assigns for each time unit an active task. The schedule repeats after a duration \( L_T \in N \) according to all scheduled tasks \( T \) with

\[
L_T = \text{lcm}(T_1, ..., T_n). \tag{1}
\]

Figure 1: Conceptual illustration of the model-based development of real-time systems using the presented XML schema.

The schedule is feasible if for the duration of \( L_T \) no task \( T_i \in T \) invalidates its relative deadline \( D_i \) for its current execution. Furthermore, it is possible to assign a priority \( p_i \in N \) to a task \( T_i \in T \). At each clock cycle of a schedule the task with the highest priority is executed. The priority value assigned to a task depends on the used scheduling policy.

2.2 Example application: Ball on a plate system

As an example application we use a ball on a plate system. The goal of such a system is balancing a ball in the center of a plate which is controlled in a two-dimensional manner using a stepping motor for each axis. A webcam is mounted vertically above the plate to capture the ball position. Overall, we can divide the system into three main tasks:

1. Capturing the ball position using the webcam.
2. A calculation component which is divided in the following subtasks:
   (a) Image processing algorithm to calculate the exact ball position.
   (b) Calculating the manipulated value for the stepping motors using a PID controller.
   (c) Sending the value to the stepping motors.
3. Sending diagnostic information via UDP port.

Although the calculation component comprises three subtasks, they are executed as one task due to their strong sequential nature. The three main tasks are scheduled using EDF as scheduling policy. This ensures that a new image can be captured while the previous one is processed. Diagnostic informations, e.g. the current ball position, are sent asynchronously via UDP to a specified remote computer. The timing constraints including the wcet and the rate are illustrated in table 1. The wcet of a single task was determined empirical by executing a set of test runs.
Table 1: Timing constraints of the three main tasks of the ball on a plate system.

<table>
<thead>
<tr>
<th>Task</th>
<th>Constraint 1</th>
<th>Constraint 2</th>
<th>Constraint 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image capturing</td>
<td>16500 µs</td>
<td>33500 µs</td>
<td>4300 µs</td>
</tr>
<tr>
<td>Calculation component</td>
<td>70000 µs</td>
<td>35000 µs</td>
<td>40000 µs</td>
</tr>
<tr>
<td>UDP component</td>
<td>33000 µs</td>
<td>40000 µs</td>
<td>40000 µs</td>
</tr>
</tbody>
</table>

2.3 Textual modeling using XML

The metamodel is formulated as an XML Schema so that the properties of a real-time application can be expressed as an XML document. The mass of established XML validation parsers makes this choice the mass of established tools available. Furthermore, the readability of XML is a benefit. The following example models the scheduler which schedules the tasks of our example application using EDF as scheduling policy. The CPU is identified by its index. A utilization test is applied for proving that the given set of tasks is feasible according to EDF as scheduling policy. Consequently, the tasks in the given set of tasks are modeled.

```xml
<rt:app name="BallOnAPlateSystem">
  ...
</rt:app>
```

According to the task model, each task of a real-time application contains a name as child element of the task element. The name attribute specifies the name of the task. We also offer the possibility to define an identifier for the scheduler and the tasks and optionally a priority attribute. The timing constraints of a tasks are modeled as child elements and optionally a priority attribute. The timing constraints can also modeled in nanoseconds (ns), milliseconds (ms) or seconds (s). In the following listing, the image capturing task of the example application is modeled.

```xml
<rt:task name="ImageCapturingTask">
  <rt:rate value="40000" unit="us"/>
  <rt:deadline value="40000" unit="us"/>
  <rt:arrival time="0" unit="us"/>
  <rt:wcet value="17500" unit="us"/>
</rt:task>
```

It is also possible to model the CPU on which the scheduler and the tasks are executed using the respective attributes. This is especially important if the target platform on which the application is executed uses a multicore architecture. The CPUs are identified by indexes starting from 0 up to the number of available CPUs. The following example models the scheduler which schedules the tasks of the example application. The scheduling policy is the EDF policy.

```xml
<rt:scheduler policy="edf"/>
<rt:scheduler-cpu cpu="0"/>
<rt:scheduler-cpu cpu="1"/>
```

The set of tasks of the example application is modeled as child elements of the task element. Each task is associated with the scheduler and the CPU on which it is scheduled. The application starts with the root element named `rt:app`. The following example models the calculation component of the example application.

```xml
<rt:task name="CalculationComponent">
  <rt:rate value="40000" unit="us"/>
  <rt:deadline value="40000" unit="us"/>
  <rt:arrival time="0" unit="us"/>
  <rt:wcet value="35000" unit="us"/>
</rt:task>
```

2.5 Code generation

If the chosen scheduling policy is EDF, the well-defined utilization test is applied for proving that the given set of tasks including their timing constraints.

```
U_{\text{ExampleApp}} = \frac{\sum_{i=1}^{n} \text{rate}[T_i]}{\text{period}[T_i]} \leq 1.
```

By parsing the given model using a XML parser we get the specified tasks including their timing constraints. The code is based on MARTOP and offers the developers a full functional template as entry point for implementing an abstract class including the concept of a custom scheduling policy. The class is generated for each task described in the model. The concept of a custom scheduling policy is associated with the app model.

### Code Generation

- **MARTOP**
- **edaRT**
3. CONCLUSION AND FUTURE WORK

In this paper we presented an approach to model the software part of a real-time application using XML. The structure of the XML document is described via a XML schema and can be appropriately validated. The resulting XML structure is based on the task model from the real-time theory and offers developers an intuitive possibility for describing scheduling policy and timing constraints of a real-time application. With the help of an example application the features of scheduling analysis and code generation are presented.

The aim of our work is to support real-time application developers and make their life easier. In our opinion, a step in the right direction has been made with the approach presented in this paper. But there are still lacking features. First, we assume a single-core processor but multi-core processors are already state of the art. Accordingly, a future development in the direction of multi-core scheduling is urgent and necessary. Second, the modeling capabilities are kept simple and the full advantage of this approach is not taken. On the one hand, this fact makes it easy for developers to learn how to model a real-time application using our approach. On the other hand, it is only possible to model applications with low complexity that makes our approach for most real-world applications insufficient at this time. Our major request is to offer our software and tools to other developers and support them in developing real-time applications, so these future works will make our approach applicable for the practice.

4. REFERENCES

ABSTRACT

On real-time systems running under timing constraints, scheduling can be performed when one is aware of the worst case execution time (WCET) of tasks. Usually, the WCET of a task is unknown and schedulers make use of safe over-approximations given by static WCET analysis. To reduce the over-approximation, WCET analysis has to have information about the underlying hardware behavior, such as pipelines and caches. In this paper, we focus on the cache analysis, which classifies memory accesses as hits/misses according to the set of possible cache states. We propose to refine the results of classical cache analysis using a model checker, introducing a new cache model for the least recently used (LRU) policy.

Keywords

Worst Case Execution Time, Cache Analysis, Model Checking, Least Recently Used Cache

1. INTRODUCTION

On critical systems, one should be able to guarantee that each task will meet its deadline. This strong constraint can be satisfied when the scheduler has bounds on every tasks’ execution time. The aim of a WCET analysis is to compute such safe bounds statically. In order to provide a satisfiable bound, the WCET analysis needs to model the execution of instructions at the hardware level. However, to avoid the huge latency of main memory access, one can copy parts of the main memory into small but fast memories called caches. In order to retrieve precise bounds on the execution time of instructions, it is thus mandatory to know which instructions are in the cache and which are not. In this paper we focus on instruction caches, i.e. we aim at knowing whether instructions of the program are in the cache when they are executed.

For efficiency reasons, the main memory is partitioned into fixed size blocks. To avoid repeated accesses to the same block, they are temporary copied into the cache when requested by the CPU. This way, they can be retrieved faster on the next access, without requesting the main memory again. Moreover, to speed up the retrieval of blocks from the cache, caches are usually partitioned into sets of equal sizes. A memory block can only be stored in one set that is uniquely determined by the address of the block. Thus, when searching a block in the cache, it is looked for in only one set. Since the main memory is bigger than the cache, it may happen that a set is already full when trying to store a new block in it. In this case, one block of the set has to be evicted in order to store the new one. This choice does not depend on the content of the other sets and is done according to the cache replacement policy. In our case, we focus on the Least Recently Used (LRU) policy (for other policies, refer to [5]). A cache set using the LRU policy can be represented as a queue containing blocks ordered from the most recently accessed (or used) to the least recently accessed. When the CPU requests a block that is not in the cache (cache miss), it is stored at the beginning of the queue (it becomes the most recently used block) and the last block (the least recently used) is evicted. On the other hand, when the requested block is already in the cache, it is moved to the beginning of the queue, delaying its eviction. The position of a block in the queue is called the age of the block: the youngest block is the most recently used and the oldest is the least recently used.

The aim of a cache analysis is to provide a classification of memory accesses as "cache hit", "cache miss" or "unknown" (not always a hit, and not always a miss) to be used as part of the WCET analysis. This classification is usually established by abstract interpretations called "May Analysis" and "Must Analysis" that respectively compute a lower and upper bound of every block's age. For more details about these analysis, refer to [3]. Must analysis is used to predict the cache hits (if a block must be in the cache when accessing it, access is a hit), whereas may analysis is used to predict the misses (if a block may not be in the cache when accessing it, access is a miss). However, if a block is in the may cache but not in the must cache, it is classified as unknown. This can happen because this access is sometimes a miss and sometimes a hit, or because the abstract interpretation is too coarse. An example is given on Figure 1. Program states (basic blocks) are on the left, whereas abstract cache states (may and must) at the exit of basic blocks are on the right. For simplicity, every basic block is stored in exactly one memory block. For example, at the exit of block 5, the minimum age of blocks a, b, c and d computed by the may analysis are respectively 1, 0, 2 and 1. At block 6, a is accessed and is in the cache (because there are only 4 different blocks, and they all fit together in the cache), thus it should be classified as a hit. However, it is classified as "unknown" by the may and must analysis because of an over-approximation performed by the must analysis. Indeed, at entry of block 5, the most cache is [⊥,⊥,⊥,a] because a is the only block that must be accessed, and b, c and d may be accessed since the last access to a. An other possibility to classify memory access...
as hit or miss is to use a model checker. Both the program and the cache are encoded as a transition system. Then, the question of the existence in the cache of a given block at a given program point is encoded as a logical formula. Both the formula to check and the transition system are provided to the model checker, that classifies the block as “in the cache”, “out of the cache” or “unknown”. Since the model deals with every reachable program/cache states separately, model checking is usually more precise than abstract interpretation. However, it is also much slower and often requires more memory during the analysis.

To avoid the precision loss of abstract interpretation without performing a heavy analysis using a model checker, we propose to mix both analysis. We first perform the classical may/must analysis by abstract interpretation, and then refine the classification using a model checker. Thus, we only use the model checker to classify blocks that were classified as "unknown" by the abstract interpretation. Moreover, we introduce a new abstract model that can be used by the model checker to efficiently represent LRU cache states.

2. WCET ANALYSIS

This section gives some basic notions about WCET analysis and explain the link with cache analysis.

Usually, WCET analysis are performed by following steps:
First, a control flow graph is retrieved from the binary code under analysis by grouping instructions into basic blocks (sequences of instructions with only one entry point and one exit point). Then, the WCET of the program is computed by bounding the execution time of every basic block and finding the “longest” path inside the CFG.

The computation of basic blocks execution time is done by micro-architectural analysis that models pipelines and caches. Independently, several other analysis like value analysis and loop bound analysis are performed to provide information about the semantics to the WCET analyzer.

At hardware level, the uncertainty about execution times of instructions comes from pipelines (which can start executing an instruction before the previous one is finished) and caches (which avoid costly main memory accesses). The aim of a cache analysis is to classify memory accesses as cache hit or cache miss. Since an access to the main memory can be 100 times slower than an access to a cache, it is mandatory to classify memory access as hit/miss in order to provide accurate bounds on WCET estimations. Moreover, when a memory access is not classified as a hit or a miss, the WCET analysis must treat the both cases [2]. Thus, it can make the analysis slower and increase the pessimism.

To avoid this precision loss, we aim at refining the classification of unknown block using a model checker.

3. RELATED WORK

Some previous works use a model checker for performing WCET analysis of programs. The approach of Lv, Yi, Guan and Yu [6] is the following. Using abstract interpretation, they classify memory access as "cache misses", "cache hits" or "unknown". Every memory access is translated in a timed automaton that describe the access to the memory bus by introducing some delay. Finally, these automata are connected together according to the CFG and the model checker explores the transition system, computing a WCET estimation. This approach allows them to perform WCET analysis of multicore systems. Although they are using a model checker in a WCET analysis, our approach is different and complementary. Indeed, they are using the model checker to refine the timing analysis itself, and not the classification of memory accesses as we do. Therefore, our refinement of the cache content can be added to the first step of their analysis to retrieve better bounds. The work of Chattopadhyay and Roychoudhury [1] is closer to our approach. They use the model checker to analyze behavior of caches shared by several cores. Moreover, by instrumenting the program at source code level they can take the program semantics into account and do not treat infeasible path, make the analysis more precise. Since they are performing a may/must analysis as a first step, we believe our analysis can be used to refine this first step, before taking shared caches into account.

4. OUR ABSTRACT MODEL

To perform the cache analysis using a model checker, we have to provide models both for the program and for the cache. We use these models to answer the following questions: “At a given program point, is a given block in the cache whatever the path to reach this point is?” (classify as hit) and “Is the given block never in the cache at the given program point, whatever the path used to reach this point is?” (classify as miss).

To model the cache, we use an abstraction of the real cache state to avoid the state space explosion problem one can meet when using a model checker. In the following formal description of our model, we adopt the notations from Jan Reineke’s PhD [7]:

**Definition 1. Cache size**

\[ k \in \mathbb{N} \text{ is the size of the cache set (in blocks)} \]

**Definition 2. Set of memory blocks**

\[ M \text{ represents the set of memory blocks.} \]

\[ M_{\bot} = M \cup \{\bot\} \text{ includes the empty line.} \]
Definition 3. Set of Cache States

\[ C^{LRU} \subseteq M^k, \]

symbolizes the set of reachable cache states

\[ [b_1, b_2, ..., b_k] \in C^{LRU} \text{ represents a reachable state} \]

\[ b_i \] is the least recently used block

In addition to these notation, to define our abstract model, we introduce the following notations: \( A \), represents the set of abstract cache states, \( \varepsilon_a \in A \), represents cache states that do not contain \( a \), and \( [S]_a \in A \), represents cache states that contains \( a \) and some other blocks younger than \( a \) (forming the set \( S \)), where \( a \in M \) is a memory block.

Definition 4. Set of Abstract Cache States

\[ A = \mathcal{P}(C^{LRU}) \] is the power set of reachable cache states.

Definition 5. Abstract Cache States

\[ \varepsilon_a = \left\{ [b_1, ..., b_k] \in C^{LRU}, \forall i \in [1, k], b_i \neq a \right\} \in A \]

\[ [S]_a = \left\{ [b_1, ..., b_k] \in C^{LRU}, \exists i \in [1, k], b_i = a \land (b_j \in S \Leftrightarrow j < i) \right\} \in A \]

The idea behind the abstract model we define below is to track only one block (noted \( a \)). Indeed, to know whether a block is in a LRU cache, you only have to count the number of accesses made to pairwise different blocks since the last access to it. In other words, you do not have to know the age of others blocks, you are only interested in knowing if they are younger than the block you are tracking. Therefore, we group together cache states that have the same set of blocks younger than the block we want to track.

To every cache state \( p \), we associate an abstract state \( \alpha_a(p) \) which consists of the set of values younger than \( a \) in the cache or a special value in the case where \( a \) is not in the cache.

Definition 6. Abstraction of Cache States

\[ \alpha_a : C^{LRU} \to A \]

\[ \alpha_a([b_1, ..., b_k]) = \begin{cases} \varepsilon_a & \text{if } \forall i \in [1, k], b_i \neq a \\ {[b_1, ..., b_{i-1}]_a} & \text{if } \exists i \in [1, k], b_i = a \end{cases} \]

For example, when tracking block \( a \), the abstract cache state associated to the exit of block 1 of Figure 1 is \([\{\}[_a, symbolizing that \( a \) is the least recently used block at this point (the set of younger blocks is empty). At the exit of block 4, the abstract cache state is \([\{b, c, d\}_a].\]

Additionally, we define the partial function \( update_a^{LRU(k)} \), which models the effect of accessing a block on an abstract state. When the abstract cache state does not contain \( a \) (i.e. is equal to \( \varepsilon_a \)), it remains unchanged until an access to \( a \) is made. When \( a \) is accessed, every new block access appears into the set \( S \). When the cardinal of \( S \) reaches \( k-1 \) (a is the least recently used block), a new access to a different block evicts \( a \) (and new abstract cache state is reset to \( \varepsilon_a \)). If an access to \( a \) is done in the meantime, the set \( S \) of younger block is reset to \( \{\} \).

Definition 7. Abstract State Update

\[ update_a^{LRU(k)} : A \times M \to A \]

\[ update_a^{LRU(k)}(\varepsilon_a, c) = \begin{cases} [\{\}][a] & \text{if } a = c \\ \varepsilon_a & \text{if } a \neq c \end{cases} \]

Considering the example of Figure 1, the model checker associates two different abstract states to block 5 depending on the incoming flow from block 1 or block 4. These states are respectively \([\{\}][a\] and \([\{b, c, d\}][a\]. Thus, applying the update function for treating the access made to \( b \) in block 5, we obtain \([\{b\}][a]\) and \([\{b, c, d\}][a]\). Therefore, we know that \( a \) is not evicted from the cache by block 5 and access made to \( a \) in block 6 is not classified as unknown anymore but as a hit.

The second part of our model is the model of the program. Since we focus on instruction caches, the model we use for the program is a graph obtained from the CFG by splitting basic blocks (when needed) into blocks of the size of a memory block. Thus, a path in the model represents the sequence of memory access that the instruction cache handles during the execution of the program. However, because we only track one memory block at a time, it is also possible to simplify the control flow graph according to this block. Indeed, one can slice the CFG according to the cache set associated to the block we want to track, removing every memory access to an other cache set. Moreover, we can remove from the obtained graph every node that is not an access to \( a \) and that does not contribute to \( a \) eviction. Thus, it appears that every node that does not contain \( a \) in their entry may cache can be removed.

![Figure 2: Simplifying CFG according to access to a](image)

This simplification of the CFG is illustrated on figure 2. Plain arrows represent program flow potentially manipulating block \( a \), whereas dashed arrows represent flow that does not and that can be simplified in only one arrow. At some point after an access to block \( a \), we can be sure that \( a \) is not in the cache anymore. Therefore, it is possible to remove all nodes (dashed arrows) from this point until the next access to \( a \).

5. IMPLEMENTATION / EXPERIMENTS

This section describe the prototype we build and the experiments we made to valid our proof of concept. The workflow of our analysis is illustrated on Figure 3.

Our implementation does not use directly the binary code to analyze but runs on the LLVM bytecode representation of it. We first build the CFG of the program from the bytecode using the LLVM framework. Since the
Table 1: Precision of May/Must analysis and Model Checker

<table>
<thead>
<tr>
<th>Program</th>
<th>Size</th>
<th>4 ways</th>
<th>8 ways</th>
<th>16 ways</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Un Nc</td>
<td>Un Nc</td>
<td>Un Nc</td>
</tr>
<tr>
<td>recursion</td>
<td>25</td>
<td>34.6%</td>
<td>11.1%</td>
<td>53.8%</td>
</tr>
<tr>
<td>fac</td>
<td>26</td>
<td>34.6%</td>
<td>11.1%</td>
<td>46.1%</td>
</tr>
<tr>
<td>binarysearch</td>
<td>48</td>
<td>12.5%</td>
<td>0%</td>
<td>56.2%</td>
</tr>
<tr>
<td>prime</td>
<td>57</td>
<td>10.5%</td>
<td>0%</td>
<td>29.8%</td>
</tr>
<tr>
<td>insertsort</td>
<td>14</td>
<td>24.7%</td>
<td>29.5%</td>
<td>28.8%</td>
</tr>
<tr>
<td>bsort</td>
<td>62</td>
<td>30.6%</td>
<td>57.8%</td>
<td>53.2%</td>
</tr>
<tr>
<td>duff</td>
<td>64</td>
<td>10.9%</td>
<td>0%</td>
<td>37.5%</td>
</tr>
<tr>
<td>countnegative</td>
<td>65</td>
<td>21.5%</td>
<td>21.4%</td>
<td>43.0%</td>
</tr>
<tr>
<td>st</td>
<td>137</td>
<td>14.5%</td>
<td>30.0%</td>
<td>43.7%</td>
</tr>
<tr>
<td>hldcmp</td>
<td>179</td>
<td>11.1%</td>
<td>5.6%</td>
<td>39.6%</td>
</tr>
<tr>
<td>minver</td>
<td>265</td>
<td>20.7%</td>
<td>29.0%</td>
<td>44.1%</td>
</tr>
<tr>
<td>statemate</td>
<td>582</td>
<td>7.5%</td>
<td>2.2%</td>
<td>7.9%</td>
</tr>
</tbody>
</table>

Figure 3: Workflow of our prototype

L1VM bytecode does not affect any address to instructions, we have to provide a mapping of instructions to the main memory. For our prototype, we assume that every instruction has the same size in memory. Thus, memory blocks contain a fixed number of instructions and can be obtained by splitting basic blocks of the CFG into fixed size blocks. Using this mapping and the CFG, our prototype performs a may/must analysis of the program. For every block access classified as unknown, we build an abstract model of the cache and provide it to the model checker together with the CFG (simplified as explained above). It would be possible to use real addresses from binary code and a correspondence between LLVM bytecode and binary code, as done in [4], but this requires significant engineering and falls outside of the scope of this experiment.

We experiment our prototype with benchmarks of the TacleBench\(^1\). Table 1 contains the results of our experiments. Size of program is given in number of memory block. We run our experiments with caches of only one cache set, with different sizes: 4, 8 or 16 ways. For every experiment, we measure both the amount of accesses classified as “unknown” by the may/must analysis (column “Un”) and the amount of accesses newly classified as “always in the cache” or “always out of the cache” among the accesses left “unknown” by the may/must analysis (column Nc). During these experiments, our analysis classifies up to 57.8% of the accesses left unclassified by the abstract interpretation analysis.

6. CONCLUSION

We proposed to refine classical cache analysis by using a model checker. To avoid the common problem of state space explosion meet when dealing with model checking, we introduce a new abstract cache model. This model allows us to compute the exact age of a memory block along an execution path of the program. Thus, we can select the memory block we want to refine. Moreover, it allows us to simplify the program model too, by removing some nodes useless to the refinement. Finally, we implement a prototype and test it on a benchmark. Our experiments shows that our approach is able to refine up to 60% of the memory access classified as unknown by the abstract interpretation.

Our prototype runs on LLVM bytecode, and use an unrealistic memory mapping. As future work, we aim at implementing an analyzer that runs on the binary code. To finally validate our approach, it is also possible to compare the performance of our analysis to other analysis refining may/must analysis, like persistence analysis or analysis performing virtual inlining and unrolling.

7. REFERENCES


\(^1\)http://www.tacle.eu/index.php/activities/taclebench
On Scheduling Sporadic Non-Preemptive Tasks with Arbitrary Deadline using Non-Work-Conserving Scheduling

Homa Izadi
Islamic Azad University, E-Campus, Tehran, Iran.
hi.zadi@iaue.ac.ir

Mitra Nasri
Max Planck Institute for Software Systems, Kaiserslautern, Germany.
mitra@mpi-sws.org

ABSTRACT
In this paper, we consider the problem of scheduling a set of non-preemptive sporadic tasks with arbitrary deadlines on a uni-processor system. We modify the Precautious-RM algorithm which is an online non-preemptive scheduling algorithm based on rate monotonic priorities to cope with arbitrary deadlines and sporadic releases. Our solution is an $O(1)$ non-work-conserving scheduling algorithm.

1. INTRODUCTION
Non-preemptive scheduling reduces both run-time and design time overheads. At run-time, it avoids context switches, and hence, preserves the state of the cache and CPU pipelines for the task. At design-time, it simplifies the required mechanisms for guaranteeing mutual exclusion. Besides, due to these properties, it improves the accuracy of estimating the worst-case execution-time (WCET) of a task. Thus, it allows system designers to have a more realistic and less pessimistic configuration setup for the system.

Moreover, non-preemptive execution improves the quality of service in control systems because it efficiently reduces the delay between sampling and actuation [9]. By doing so, it increases the freshness of a sampled data. In a preemptive system, if a task is preempted after reading its data from the environment (and before using that data), the data becomes old, and hence, after resuming, the task uses the old data to calculate the output command. Consequently, controller's accuracy may reduce.

In the non-preemptive scheduling, a processing resource (or other resources) can be blocked for a long time, causing a timing violation for the tasks with shorter deadlines. Thus, in a feasible task set, the maximum execution time of a task is restricted. Cai et al., [1] have shown that in a periodic task set, the maximum permissible execution time of the tasks except $\tau_1$ (the task with the smallest period), is at most $2(T_1 - C_1)$ (where $T_1$ and $C_1$ are the period and WCET of $\tau_1$). For example, if the smallest period is 1ms and the largest period is 1000ms (which is regularly the case for the runnables in the automotive industry), then the largest execution time cannot be greater than 2ms. It means that those tasks with large period must be heavily under-utilized, otherwise, the system will not be schedulable.

In order to cope with the limitation of non-preemptive scheduling, many researches have focused on finding a safe speed-up factor, using which no task will miss its deadline (under a particular scheduling algorithm such as non-preemptive rate monotonic (NP-RM), or non-preemptive EDF (NP-EDF) [8]). However, as shown in [3], this speed-up factor may become an unreasonably large number (e.g., 120). Such a speed-up factor may not be useful for a real system where the weight, size, power consumption, and many other factors affect the choice of processor.

In many real-time systems (e.g., control or multimedia systems), finishing a task after its deadline will not cause a failure for the system (the system might not even be a safety-critical system to begin with). For example, in control systems, usually the stability of the controller will not be affected if one of the samples is taken later than the expected sampling period [9]. Such event may affect the quality of control, however, it may probably not affect the stability of the controller as long as the completion time is within a certain deadline. In other words, deadlines are not necessarily smaller than or equal to periods. As a result, the tasks can tolerate longer blocking time caused by a lower priority task. It allows us to have a non-preemptive system with longer execution times than one mentioned previously.

Recently, there have been some advancements in the state of non-work-conserving scheduling algorithms [4,5,7]. These algorithms may leave the processor idle even if there are pending jobs in the system. The idea of these works is to add an idle-time insertion policy (IIP) to the existing job scheduling policies such as RM and EDF to increase schedulability. The role of the job scheduling policy is to select a job with the highest priority, and the role of IIP is to decide whether to schedule that job or instead, to leave the processor idle. It has been shown [5] that by adding an IIP to the job scheduling policy, a significant number of task sets that are not schedulable by the job scheduling policy itself (e.g., RM and EDF) become schedulable if an idle-time insertion policy is used.

In our work, we focus on the problem of scheduling non-preemptive sporadic tasks with arbitrary deadlines upon a uni-processor system. We use the Precautious-RM (P-RM) from [7] as the basis of our algorithm, and modify it in three ways: 1) we increase its eagerness to schedule tasks rather than scheduling idle intervals, 2) we replace the existing IIP with the one which is based on arbitrary deadlines rather than implicit deadlines, and 3) we add mechanisms for enabling it to schedule sporadic tasks. Then we discuss how a schedulability test can be developed for the new algorithm. Our final solution is an online non-preemptive scheduling algorithm, called AD-PRM (where AD stands for arbitrary deadline). It has $O(1)$ computational complexity per activation of the if the number of priorities in the system is a reasonable constant number.

In the rest of the paper, we describe system model in Sect. 2. The AD-PRM, is presented in Sect. 3. Open challenges and future work are presented in Sect. 4. Sect. 5 presents the experimental results. In Sect. 4 we discuss...
about future work and then conclude the paper in Sect. 6.

2. SYSTEM MODEL AND BACKGROUND

We consider a set of independent, non-preemptive sporadic tasks, with arbitrary deadlines which must be scheduled on a single processor. Task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ has $n$ tasks and each task is identified by $\tau_i = (C_i, T_i, D_i)$, where $C_i$ is the worst-case execution time (WCET), $T_i$ is the period (the minimum inter-arrival time), and $D_i$ is the deadline of the task. We assume $C_1, T_1, D_1$ are integer multiples of the system's clock, thus they are integer values. We denote system utilization by $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ where $u_i = C_i/T_i$ is the utilization of task $\tau_i$. The hyperperiod is the least common multiple of the periods. Tasks are indexed according to their period, i.e., $\tau_1$ and $\tau_n$ are the tasks with the highest and the lowest priorities. The IIP of P-RM is based on one future job of $\tau_i$; if executing the current low priority task $\tau_i$ at time $t$ will cause a deadline miss for the next instance of $\tau_i$, P-RM schedules an idle interval until the next release of $\tau_i$. More accurately, in order to schedule $\tau_i$, one of the following conditions must hold: a) $\tau_i$ must be finished before the next release of $\tau_i$, or b) the previously executed task must be $\tau_i$ and the current task must be finished before the latest time at which the next instance of $\tau_i$ can start and finish its execution. This instant is $r_{\text{next}} + (T_i - C_i)$. If none of these conditions is satisfied, P-RM schedules an idle interval until the next release of $\tau_i$ (Line 6).

Fig. 1-(a) shows an example of a schedule produced by P-RM. However, as shown in Fig. 1-(b), the P-RM may be too cautious due to the condition $\tau_i$ is the latest executed task (in Line 3 of Alg. 1). Also, the first late job of $\tau_2$ (the one released at 36) will cause another late job for its next job as well. By a late job we mean a job that is not finished before the next job of its own task is released.

3. NON-WORK-CONSERVING SOLUTION

As it has been shown in Fig. 1-(b), P-RM is not always efficient since due to Line 2 (of Alg. 1), it adds more idle-times than it may be necessary. The original motivations for having this condition was to make the algorithm optimal to schedule harmonic tasks with $\forall i: T_i/T_{i-1} = 2$ (proven by Nasri et al., [7]). However, in general, this condition just makes the P-RM very cautious.

In the current implementation of P-RM (see Alg. 1), $\tau_1$ is only allowed to be scheduled if it can finish its execution (at $t + C_i$) before $r_{\text{next}} + T_1 - C_1$ which is supposedly the latest valid start time for the next job of $\tau_1$. However, if deadlines are arbitrary, this condition becomes very pessimistic because $\tau_1$ can be scheduled even after its deadline. Besides, if one of the tasks has $C_i > 2(T_i - C_1)$, it will never be scheduled by the current implementation of P-RM. Moreover, if a job of $\tau_1$ is not finished before the next job of $\tau_1$, it cannot be scheduled since none of the two conditions in Line 2 of Alg. 1 allows it.

Algorithm 1: P-RM

Input: $t$: the current time

1. $\tau_1 \leftarrow$ the highest priority task with a pending job;
2. $r_{\text{next}} = \left\lfloor \frac{t}{T_1} \right\rfloor T_1$ which is the release time of the previous job of $\tau_1$ before or at $t$;
3. if $(t + C_i < r_{\text{next}})$ or $(t + C_i \leq r_{\text{next}} + T_1 - C_1)\) then
   4. Schedule $\tau_1$;
   5. else
      6. Schedule an idle interval from $t$ to $r_{\text{next}}$;
end

Our new solution is shown in Alg. 2. If $i = 1$, i.e., current pending task is $\tau_1$, AD-PRM schedules it without checking any other condition. Hence, it allows late jobs of $\tau_1$ to be safely scheduled. Also, if the finish time of the current high priority job at $t + C_i$ satisfies the following condition, it will be scheduled (it does not need to be scheduled only after one job of $\tau_1$):

$$t + C_i \leq r_{\text{last}} + T_1 + D_1 - C_1$$

(1)

In Equation (1), the right-hand-side represents the latest safe start time of $\tau_1$ which is not violating its deadline. Doing so, our algorithm can be applied for arbitrary deadline tasks.

Yet, another important feature in AD-PRM is to use $r_{\text{last}} + T_1$ instead of $r_{\text{next}}$. In a fully periodic task set which does not have any sort of release jitter, $r_{\text{last}} + T_1$ is always equal to $r_{\text{next}}$ however, if the task set is sporadic, we may not know the exact release time of the next job of $\tau_1$ in the future. As a result, the old version of the P-RM could not

Figure 1: Examples of P-RM schedules in comparison with an optimal schedule.
handle sporadic tasks. To handle sporadic tasks efficiently, we assume that at time $t$ a sporadic task $\tau_i$ can be scheduled if $C_i \leq D_i - C_i$. It means that if a job of $\tau_i$ is released slightly after $t$, still it will be able to meet its deadline since the execution time of the low priority task is smaller than $D_i - C_i$. However, if $C_i > D_i - C_i$, AD-PRM will not allow it to be scheduled until one job of $\tau_i$ is released in the system (see Line 6 of Alg. 2). When that job is scheduled, if $\tau_i$ is still the highest priority job in the system, it can be scheduled through (1) condition. Note that AD-PRM will not let any job of $\tau_i$ to be missed due to a low priority task.

The last modification is in Line 6 of Alg. 2, where instead of scheduling an idle interval until $r_{\tau_i}^{\text{new}}$ which is a fixed value in time, we schedule the idle interval until the next release event in the system. If the next release is for one of the jobs of $\tau_i$, AD-PRM will behave almost similar to the P-RM, but if it is another high priority job, then that job may be able to satisfy conditions of Line 3 (Alg. 2), and hence, has a chance to be scheduled even without waiting for the next job of $\tau_i$. Note that a release of a low priority job will not change the situation because AD-PRM is a fixed priority algorithm and as long as $\tau_i$ has a pending job, it will not schedule the jobs of lower priority tasks.

It is worth noting that if the release of higher priority tasks does not happen within a bounded amount of time, we cannot provide any efficient schedulability test for AD-PRM due to the condition in Line 6 of the algorithm. Thus, in order to propose a schedulability test for AD-PRM we need to have some assumptions about the maximum inter-arrival time of the tasks (rather than just having an information about the minimum inter-arrival times). Providing a schedulability test is our next future work.

As can be seen in Alg. 2, computational complexity of AD-PRM is $O(1)$ (per activation of the algorithm), provided that the highest priority task with a pending job (Line 2 of Alg. 2) can be found in $O(1)$. It is worth noting that since AD-PRM inserts idle-times in the schedule, it may happen that a particular job, for example from task $\tau_i$, is postponed several times by the algorithm, as long as it finds a better chance to be scheduled. Thus, if we count the total overheads of the algorithm in the hyperperiod, it might be slightly larger than RM.

4. DISCUSSIONS AND FUTURE WORK

In this section first we discuss potential challenges and then provide a list of future work.

Carry-in jobs over hyperperiods: unlike work conserving algorithms, AD-PRM may face a situation in which some jobs are late even after the hyperperiod and even if the total utilization is smaller than 1. Fig. 2 shows such an example. As can be seen, the last job of $\tau_i$ took one unit of time from the beginning of the next hyperperiod. In this particular example, the next hyperperiod will only have one carry-in job and the schedule repeats afterwards. However, since 1) the amount of carry-in load, and 2) the priority of carry-in load, and 3) the number of carry-in jobs, affect decisions of the IHP, we may need to consider many more hyperperiods before we can reach to a conclusion about the schedulability of the jobs. Note that even if $U < 1$, a carry-in job may appear. Two important consequences of having carry-in jobs before a hyperperiod are: 1) experiments must be performed on a longer periods of time (longer than 1 hyperperiod) and 2) if the schedule is stored in a timetable, the size of the time table may be much larger than the number of jobs in one hyperperiod.

Effect of utilization of $\tau_i$ on the schedulability test: As shown in Fig. 2, if $\tau_i$ has high utilization and a low priority job forces $\tau_i$ to finish just before its deadline, then a long blackout happens for the system, namely, for a long duration of time, the system cannot respond to any other job until alllate jobs of $\tau_i$ are finished. Duration of this blackout will be longer if $\tau_i$ has higher utilization. For example, in Fig. 2, when the first job of $\tau_i$ which is released at 20 becomes late, for 28 units of time (equivalent to 4 executions of $\tau_i$), no other low priority task will find any chance to be executed. The blackouts are important for our scheduling algorithm because $\tau_i$ plays an important role in the schedule.

An idea for a schedulability test: In [4, 7], a schedulability test has been suggested for the P-PRM which was based on counting the number of vacant intervals and checking whether each low priority task can have at least one vacant interval in its priority level before its deadline. A vacant interval is basically the slack between two consecutive jobs of $\tau_i$, and has the length $2(T_i - C_i)$. It may appear every $2T_i$ units of time because the P-PRM synchronizes the schedule of the jobs with the releases of $\tau_i$ using idle intervals. A vacant interval can be used to schedule a low priority task, as long as $C_i \leq 2(T_i - C_i)$, because then in the worst-case, each job takes one vacant interval to be scheduled. Thus, for each priority level, we can count the minimum number of available vacant intervals and make sure that each task has at least one of them.
we will extend it for sporadic tasks. Since idle-time insertion policies need to have some information about the future, without knowing anything about the maximum inter-arrival-time of the tasks, we may not be able to design a schedulability test. Unlike work-conserving solutions, here, a sequence of periodic releases may not create the worst-case scenario because of the idle intervals.

5. EXPERIMENTAL RESULTS

We compare AD-PRM with CW-EDF [5]. Group-based EDF [2], NP-RM, and NP-EDF. Group-based EDF has been developed for soft real-time systems. It groups the tasks based on deadlines, and then, among the group with the earliest deadline selects the task with the shortest execution time. This algorithm does not provide any guarantee for tasks with arbitrary deadlines.

The parameter of our experiment is the deadline of $\tau_i$. We have measured the schedulability ratio by dividing the number of randomly generated task sets that could actually be scheduled by each algorithm divided by the total number of task sets. Note that we have discarded those task sets that had some late jobs at the end of hyperperiod. To generate a random task set, first we choose random periods from range [10, 500] for 5 tasks. Then we assign $u_i$ from [0.3, 0.95] and $D_i$ from [$T_i[D_{max}^iT_i]$] where $D_{max}^i$ is the parameter of the experiment. Then we calculate $C_i = u_i T_i$. For the other tasks, we assign $D_i = T_i$ and $C_i$ is randomly selected from interval [0.0001, $T_i + D_i - 2C_i$]. We generate 1000 task sets per each value of the horizontal axis to obtain Fig. 3. Average utilization of the each generated task set using these parameters is about 0.8.

Fig. 3 shows that AD-PRM can efficiently schedule about 80% of the task sets. It is not yet as good as CW-EDF which is a dynamic priority scheduling. We also observe that with the increase in $D_i$, EDF has better chance to schedule the task set because most missed jobs of EDF are jobs of $\tau_i$. Consequently, if $\tau_i$ has larger deadline, NP-EDF will have higher schedulability. NP-EDF has almost 0 miss ratio for the jobs of $\tau_i$, and the highest miss ratio for the jobs of $\tau_j$. Note that vertical axis in Fig. 3-(b) shows the ratio of missed jobs of a task to the total number of its jobs. Thus, sum of values of a column won’t give the total miss ratio.

6. CONCLUSIONS

In this work, we have shown how to modify Precautious-RM to handle tasks with arbitrary deadlines and sporadic releases. We have discussed the effect of the late jobs, i.e., those that cannot be finished before their next release, on the next hyperperiod. We have shown that even if $U \leq 1$, some jobs may become late because our algorithm may schedule idle intervals to guarantee deadlines. We have also presented initial ideas for a schedulability test and discussed the future work.

Acknowledgment

This work is supported by Alexander von Humboldt foundation.

7. REFERENCES


Tester Partitioning and Synchronization Algorithm For Testing Real-Time Distributed Systems

Deepak Pal, Jüri Vain
Department of Computer Science
Tallinn University of Technology
Tallinn, Estonia 12186
{deepak.pal, juri.vain}@ttu.ee

ABSTRACT

Large-scale cyber-physical systems (CPS) have grown to the size of global geographic distribution with numerous services and applications forming an ubiquitous computing network. Timing latencies introduced by geographically distributed ports of such CPS (or the system under test) affect the response time of the centralized tester, thereby making the testing infeasible. Consequently, the remote testing approaches replaced by a distributed test architecture. Sufficient timing conditions for remote online testing have been proposed by David et al for remote Δ-testing method. We extend the Δ-testing by deploying testers on fully distributed test architecture. This approach reduces the test reaction time by almost a factor of two wherein the timing of coordination messages is implemented based on model-based testing platform DTRON.

Keywords

Model-based testing; Distributed systems; Low-latency systems

1. INTRODUCTION

The distributed large-scale systems have grown to the size of global geographic distribution with numerous services and applications forming an ubiquitous computing network. Examples of such systems are smart energy grids, transportation systems, manufacturing processes etc. which are expected to behave in a correct manner in real-time. The geographical distribution, communication and presence of numerous components introduces timing imperfections which, if not considered during the design and deployment phases can lead to catastrophic outcomes that affects the reliability of the system as a whole. Building a reliable CPS require a system to be tested for performance in the presence of these timing imperfections induced by various components and the coordination between them. Timing latencies introduced by geographically distributed ports of such CPS set strict response time constraints also to the testing tools of such systems.

Reaching sufficient test coverage by integration testing of such systems in the presence of numerous latency factors and their interdependency, is out of the reach of manual offline testing. Moreover, off-line testing of such systems is not possible due to the non-deterministic nature of system under test (SUT). Consequently, the off-line testing approaches replaced by on-line distributed testing.

2. STATE OF THE ART

Testing distributed systems has been one of the model based testing (MBT) challenges since the beginning of 90s. One of the first attempt to standardize the test interfaces for distributed testing was made in [1]. Early MBT approaches represented the test configurations as systems that can be modeled by finite state machines (FSM) with several distributed interfaces, called ports. An example of abstract distributed test architecture is proposed in [2]. This architecture suggests the SUT contains several ports that can be located physically far from each other. The testers are located in these nodes that have direct access to ports. There are also two strongly limiting assumptions: (i) the testers cannot communicate and synchronize with one another unless they communicate through the SUT, and (ii) no global clock is available. Under these assumptions a test generation method was developed in [2] for generating synchronizable test sequences of multi-port finite state machines. Two major issues which occur in the phase of test execution are synchronization and fault detectability problems. However, it was shown in [3] that no method that is based on the concept of synchronizable test sequences can ensure full fault coverage for all the testers. The investigation in [4] used the concept of controllability and observability to study the problems of synchronization and fault detectability. A coordination algorithm that allows the testers to exchange messages through external reliable communication independent of the SUT to solve the controllability and observability problems was proposed in [5]. However, the introduction of extra communication messages creates undesirable delays and communication overhead to be addressed by tester middleware.

In [6], it is proposed to construct test sequences that cause no controllability and observability problems. But the approach relies on assumption that SUT is deterministic which makes it impractical for realistic testing applications. Alternatively, an online tester synthesis method is proposed in [7] capable of handling non-deterministic behavior of SUT. This approach can be used for centralized remote testing of distributed systems but the issue of satisfying test timing constraints remains. A mechanism to solve the controllability and observability problems in the presence of timing constraints for testing distributed systems was proposed in [8]. The investigation proved that even when the SUT does not have critical time requirements, still it should respect some timing constraints for solving controllability and observability issues.

More recently, pioneering results on testing timing cor-
rectness with remote testers was proposed in [9] where a remote abstract tester was proposed for testing distributed systems in a centralized manner. It was shown that if the SUT ports are remotely observable and controllable then $2\Delta$-condition is sufficient for satisfying timing correctness of the test. Here, $\Delta$ denotes an upper bound of message propagation delay between tester and SUT ports. Though this approach works reasonably well for systems with sufficient timing margins, but it cannot be extended to systems with the timing constraint close to $2\Delta$. This means that the test inputs may not reach the input port in time and as a result the testing becomes infeasible in such systems.

To shorten the tester reaction time, our recent work [10] proposed an approach for improving the performance of $\Delta$-testing method (proposed originally for single remote tester) by introducing multiple local testers attached directly to the ports of SUT on a fully distributed architecture. It was shown that this mapping to distributed architecture preserves the correctness of testers so that if the monolithic remote tester meets $2\Delta$ requirement then the distributed testers meet (one) $\Delta$-controllability requirement.

In this paper, the main contribution is an improvement of approach proposed in [10], which includes following
(i) An algorithm that maps a central remote tester model to a set of communicating local tester models while preserving the I/O behavior at testing ports;
(ii) A new adapter model to synchronize the communication between SUT model and local tester and for sending coordination messages among local testers.

3. PROBLEM STATEMENT

3.1 Model-Based Testing

Model-Based Testing is an approach to the test construction and test execution process [12]. It is generally understood as black-box conformance testing where the goal is to check if the behavior observable on system interface conforms to a given requirements specification. The formal requirements model of SUT describes how the SUT is required to behave and allows generating test oracles. The test purpose most often used in MBT is conformance testing. In conformance testing the SUT is considered as a black-box, i.e., only the inputs and outputs of the system are externally controllable and observable respectively. During testing, a tester executes selected test cases on the SUT and emits a test verdict (pass, fail, inconclusive). The verdict shows correctness in the sense of input-output conformance (IOCO) [11]. Due to native non-determinism of distributed systems the natural choice is online testing where the test model is executed in lock step with the SUT. The communication between the model and the SUT involves controllable inputs of the SUT and observable outputs of the SUT which makes easy to detect IOCO violations. For detailed overview of MBT and related tools we refer to [12].

In our approach Uppaal Timed Automata (UTA) [13] and [14].

Figure 1: Remote tester communication architecture

Figure 2: SUT and Remote Tester Model

3.2 From Remote to Distributed Testing

Centralized remote testing is an approach that can be used when the test implementation and the SUT are in different locations. Consider a distributed CPS with geographically distributed and interacting applications that needs to be tested for timing performance using a remote tester as shown in Figure 1. This means that the remote tester will generate an input for the SUT, waits for the result and continues with the next set of inputs until the test scenario has been finished. Thus, the tester has to wait for the duration it takes to transmit the signal from the tester to the SUT’s ports and the responses back from SUT ports to the tester. Since, in most real-time distributed CPS, the timing constraints introduced are significant and time-varying thereby introducing non-determinism. Under circumstances where message propagation time is close to required test reaction time, remote testing may lead to situations wherein the centralized remote tester is unable to generate the necessary inputs for the SUT within the expected timing window, thereby rendering the testing infeasible [15]. Consequently, the centralized remote testing approach is not suitable for testing a distributed system if the system has strict timing constraints. The shortcomings of the centralized remote testing approach are mitigated with extending the $\Delta$-testing idea by mapping the monolithic remote tester into multiple local testers as shown in Figure 3. This modification eliminates the message propagation time between the local tester and SUT as the testers are attached directly to the ports and it reduces the overall testing response time.
4. PARTITIONING ALGORITHM

We apply Algorithm 1 to transform the centralized testing architecture depicted in Figure 1 into a set of communicating distributed local testers, the architecture of which is shown in Figure 3. Let us consider the remote testing architecture in Figure 1 and the corresponding model depicted in Figure 2. The SUT has 3 geographically distributed ports (p₁, p₂, p₃) that interact within the system, inputs in₁[1], in₂[2] and in₃[3] at ports p₁, p₂ and p₃ respectively and outputs out₁[1] at port p₁, out₂[2] at port p₂, out₃[3] at port p₃. Now, the local testers are generated in two steps:

(i) a centralized remote tester is generated, e.g. by applying the reactive planning online-tester synthesis method of [7];
(ii) a set of synchronizing local testers are derived by mapping the monolithic tester into a set of location specific local tester instances.

Each tester instance needs to know only the occurrence of those I/O events at other ports that influence its behavior. Possible reactions of the local tester to these events are already specified in the initial remote tester model and do not need further feedback to the event sender. After mapping the monolithic tester into distributed local testers, the message propagation time between the local tester and the SUT port is eliminated because the tester is attached directly to the port. This means the overall testing response time is also reduced, since previously the messages were transmitted twice: from remote tester to the port and back from port to the tester. The resulting architecture mitigates the timing issue by replacing the bidirectional communication with a unidirectional broadcast of the SUT output signals between the distributed local testers.

Algorithm Description: Assume SUT has n geographically distributed ports. Algorithm 1 generates n communicating local testers from monolithic remote tester. Let M^RT denote a monolithic remote tester model generated by applying the reactive planning online-tester synthesis method [7]. Loc(SUT) denotes a set of geographically different port locations of SUT. The number of locations Loc(SUT) = \{l_n | n \in \mathbb{N}\}. Let Pₙ denotes a set of ports accessible in the location lₙ. For each port locations lᵢ, lₙ \in Loc(SUT) we copy the monolithic remote tester M^RT to M^lₙ to be transformed to a location specific local tester instance (Line 4-6). Line 7 adds an adapter model to each local tester instance. The purpose of adding an additional automata i.e. adapters to each location is that it synchronize the local communica-

tion between SUT local ports and a local tester with other testers in different locations. Its model is derived from remote tester model by adding original channels of SUT and by renaming channels of local testers. The loop in Line 9 and 10 says that for each local testers model M^lₙ we go through all the edges in M^lₙ. If the edge has a synchronizing channel and the channel belongs to same port location Pₙ then we rename the chan[1] shown in Figure 4 and add the same chan[1] to adapter model (Line 11-13). Second case in Line 14-20, if the edge has a synchronizing channel and the channel does not belong to same port location Pₙ then we do the following (i) if the channel’s action is Emission i.e in[1][!!], we replace it with the co-action Reception channel, rename it and add it to the adapter model, (ii) if the channel’s action is Reception i.e out[1]?, we rename it and add it to adapter model.

Figure 4 represents the generated local tester models with their corresponding parameterised adapter model where parameter L denotes the geographical location. In this composition of models it is shown that local testers work together with local adapters. The adapters synchronize the local communication between SUT local ports and a local tester with other testers in different locations i.e. in[1] and out[1] are channels between SUT and adapter; in[1] and out[1] are channels between the local adapter and local tester; and in_[-1] and out_[-1] are the channels between the local adapter and other local testers.

5. CORRECTNESS OF TESTER DISTRIBUTION ALGORITHM

The generated local testers are correct by construction which means that the partitioning preserves their correctness because the synchronization of tester local instances is preserved due to auxiliary synchronization channels introduced by the algorithm. Synchronization is preserved also among local testers and with SUT. The communication be-
between adapters and local testers are synchronized without affecting SUT actions and actions on other tester ports which guarantee that generated local testers are bi-similar by construction. As for method implementation, the local testers are executed and communicating via distributed test execution environment DTRON [16]. We demonstrate that the distributed deployment architecture supported by DTRON and its message serialization service allows reducing the total test reaction time by almost a factor of two.

6. CONCLUSION

We have proposed a methodology and algorithm to transform the centralized testing architecture into a set of communicating distributed local testers which extend our previous work [10] by showing well-defined model transformations and introducing test adapter model templates. As for method implementation, the local testers are executed and communicating via distributed test execution environment DTRON.

7. ACKNOWLEDGMENTS

This research is partially supported by Project IUT33-13 “Strong warranties software methodologies, tools and processes” and doctoral studies TTU, Estonia.

8. REFERENCES

Framework to Generate and Validate Embedded Decision Trees with Missing Data

Arwa Khannoussi*, Catherine Dezan*, Patrick Meyer+
* Université de Bretagne Occidentale
+ Institut Mines-Télécom, Télécom Bretagne
UMR CNRS 6285 Lab-STICC
arwa.khannoussi@etudiant.univ-brest.fr
catherine.dezan@univ-brest.fr
patrick.meyer@telecom-bretagne.eu

ABSTRACT

Autonomous vehicles or devices like drones and missiles must make decisions without human assistance. Embedded inference engines, based for example on decision trees, are used to reach this goal which requires the respect of hardware and real-time constraints.

This paper proposes a generic framework which can generate automatically the adequate hardware solution taking into account the real-time and the hardware constraints. Moreover, the proposed solution supports the case of faulty sensors which generate missing data.

Experimental results obtained on FPGA and CPU suggest that it is better to consider missing data from the learning phase instead of considering it just in the classification phase. Besides, the use of the pipeline technique provides a better latency, which is adequate for real-time applications.

Keywords
Decision Trees; Missing Data; Hardware Implementation; Real-time; Deadline

1. INTRODUCTION

Autonomous vehicles or devices such as drones or missiles need embedded inference engines to make decision without human assistance. These engines, in their embedded version, should respect hardware constraints in terms of available resources, performance and energy consumption. The embedded decision is made with respect to data provided by different sensors (GPS, IR,...) with different frequencies. This leads to the definition of a set of deadlines necessary to manage in real-time the flow of values provided by the sensors. In this study, we suppose that the sensors can malfunction and consequently produce no data.

In machine learning such inference engines include artificial neural networks [5], decision trees (DTs) [10], support vector machines [13], and many others. In this work, we study decision trees, because they are one of the most popular models and they can easily be interpreted and validated by humans.

Decision trees are typically constructed from the data using several algorithms like CART (Classification and Regression Tree) [6], C4.5 [9], ... Those algorithms can be implemented and executed on traditional systems. This implementation can not bring a big improvement in term of latency which is an important characteristic to fulfill the real-time constraints. In order to improve data processing latency and the throughput when considering different instances, a hardware implementation is considered as a solution. However, hardware implementations of decision trees has not been investigated thoroughly so far.

To our knowledge, [4, 8, 7] are the only three papers available in the literature which discuss the hardware implementation of decision trees. Other papers are based on these seminal references [11, 12]. A., Bermak and D. Martinez [4] propose a hardware realization of decision trees using the 2-dimensional systolic array architecture. J.R. Struharik [8] describes a hardware implementation of decision trees for a specific problem only, while in [7], several architectures are proposed for the hardware realization of arbitrary decision trees. However, in all these references, the architectures are implemented manually and each tree has its own implementation.

In this paper we propose a generic framework that allows generating decision trees using the C4.5 algorithm because it handles training data with missing values and generating the adequate hardware solution automatically using Vivado HLS tools [1]. Data quality is a major concern in our work. It is well known that the amount of missing data clearly affects the prediction accuracy of decision trees. Our framework can therefore generate, from a configuration, the missing data and treat them using different methods. As we will show in this article, this framework actually is a tool to validate the hardware solution by comparing its results with those of the software solution.
The rest of the article is structured as follows. In Section 2, we present the different architectures of the decision trees hardware implementations that are already proposed in the literature. Section 3 is devoted to the presentation of our framework, and in Section 4 we present some experiments and the results with the framework, before concluding in Section 5.

2. RELATED WORK

In general, the input of a machine learning model is a data set composed of a set of instances evaluated on a set of attributes. For autonomous vehicles or devices, an attribute corresponds to one of the sensors and an instance includes the values of the different sensors at a given instant. In decision trees, every node of a tree represents a test that includes one or more attributes, every branch descending from a node represents one of the possible outcomes of the test, and every leaf matches with one of the classes. To evaluate the performance of decision trees we use the classification accuracy, i.e., the rate of correct predictions made by the model over a data set.

In [8], the realization of the decision tree requires to implement every node as a separate module. The problem with this architecture is that new instances cannot be classified before the end of the classification of the previous ones. A more advanced hardware realization is proposed in [4]. In order to provide a faster throughput, only two levels are used, irrelevant of the depth of the original decision tree.

Several architectures suitable for the hardware realization of decision trees are proposed in [7] and are based on the concept of universal node. Actually, to classify an instance, one node needs to be evaluated per level so only a subset of nodes will be visited, which corresponds to the path from the root to the leaf node. The number of modules required for the realization of a decision tree is equal to the depth of the tree. One universal node module per level is enough for the realization. This Single Module per Level (SMpL) architecture [7] is represented in Figure 1. The classification of an instance using the SMpL architecture can be executed sequentially or with the use of a pipeline. It is also possible to use a single universal node to evaluate every tree node. This architecture, called Universal Node (UN) [7] (Figure 2), is used when the speed of classification is not critical.

The implementation of those architectures is done manually and each decision tree is therefore implemented separately. In this contribution, we propose a framework which handles this problem. It is an automatic tool that can generate an arbitrary decision tree’s structure code on hardware, based on the SMpL architecture. It also allows generating and treat missing data using several methods, and to handle the whole process of decision trees learning, from the generation of the decision tree, to the result of the classification in terms of accuracy.

3. PROPOSED FRAMEWORK

Figure 3 details the structure of our framework and the different phases of the implementation.

Besides generating the decision tree structure code, another essential purpose of the framework that we propose is to study the behavior of decision trees when confronted with missing data. Therefore, the first part of our framework is to generate automatically the missing data by taking as input the data set and the configuration of the missing data. Actually, we consider three such missing data configurations:

- Uni-variate: missing data appear only in one attribute or sensor;
- Monotonic: one or more attributes or sensors are affected;
- Non-monotonic: missing data appears randomly in the data set.
Note that in our case, typically missing data is caused by a faulty sensor. Missing data can be taken into account during both the learning and the classification phase. The input of the framework is the data set and missing data configuration to define which type and the percentage of missing data will be generated and also the number of attributes that will be affected.

The learning phase involves the generation of the decision tree from the learning data set. In this phase, an implementation of C4.5 algorithm on R [2] has been used. C4.5 works with the concept of information entropy. The training data is a set of classified samples having p-dimensional vectors defining the attributes of the sample. It generates a decision tree where at each node C4.5 chooses the attribute that most effectively splits its set of samples into subsets enriched in one class or other. The splitting criterion is the normalized information gain. The attribute with the highest normalized information gain is chosen to make the decision. The output of this phase is the description file of the decision tree that will be used in the next phase.

The classification phase, can be executed on two different device targets, CPU and FPGA. For the first one, we also used the C4.5 algorithm for the classification and the treatment of missing data. In fact C4.5 uses a probabilistic approach to handle missing data. This part of the framework allows testing the decision tree with a number of data sets fixed by the user, and the outputs are the mean and the standard deviation of the different accuracy rates.

From the decision tree, that has already been generated by the learning phase, our framework generates automatically the decision tree structure code, based on the SMpL architecture [7]. The framework is a generic tool that can handle any decision tree regardless of the number of nodes or the type of attributes, given that the framework is a generic tool that can handle any decision tree. For the missing data, the mean imputation method is used. It consists in replacing the missing value for a given attribute by the mean of all known values of that attribute. Vivado HLS tools are used to perform the High-Level Synthesis [1].

This framework can manipulate different type of attributes, which facilitates the study of the resources used in the hardware implementation of the decision trees. It produces, the resources used and the latency as output. In order to validate our hardware implementation, this framework allows comparing it with the software version.

4. EXPERIMENTS

In this section, we first study the behavior of decision trees when confronted with missing data in both the learning and the classification phase. Then, the effect of the attribute type and the use of pipeline technique on the required resources are experimented in the hardware version of the classification phase.

The following data sets are used to illustrate the use of the framework: “Iris” (4 numerical attributes and 150 instances) and “Balance Scale” (4 nominal attributes and 625 instances); see [3] for further details.

As part of the study of the effects of missing data on the accuracy, we launched a series of experiments on the software solution. Testing takes place as follows: from the initial data set, the framework generates 1000 different data sets with different types and percentage of missing data. These are then used in both the learning and classification phases. The results are presented in Figures 4 and 5. Both curves show that classification accuracy is higher if missing data is already taken into account in the learning phase.

For the classification phase, we suppose that the decision tree is already generated. The target device is an FPGA Zynq of the Xilinx Zedboard. For each data set, the following output is produced: the accuracy with missing data, resource occupation in terms of percentage of LUT (Lookup Table) and (FF) Flip Flop and the latency for the performance aspect.

The type of attributes is one of features that can affect the performances of decision trees. We therefore used three different types of attributes for the hardware implementation: Float a number coded on 32 bits, ap-fixed<8,6> a number on 8 bits with 6 integer bits and 2 decimal places, and, ap-fixed<4,2> a number coded on 4 bits with 2 integer bits and 2 decimal places.

<table>
<thead>
<tr>
<th>Data set</th>
<th>Iris</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float</td>
<td>&lt;8,6&gt;</td>
<td>&lt;4,2&gt;</td>
</tr>
<tr>
<td>10</td>
<td>97.15</td>
<td>96.91</td>
</tr>
<tr>
<td>30</td>
<td>94.90</td>
<td>94.57</td>
</tr>
<tr>
<td>50</td>
<td>93.66</td>
<td>93.41</td>
</tr>
<tr>
<td>80</td>
<td>90.49</td>
<td>90.35</td>
</tr>
<tr>
<td>95</td>
<td>88.58</td>
<td>88.40</td>
</tr>
</tbody>
</table>

Table 1: Classification accuracies with respect to missing data percentages and various types of attributes.

Table 1 shows that classification accuracy decreases with increasing amounts of missing data and number of affected attributes. Both the type Float and the type
ap-ufixed<8,6> have nearly the same accuracy because the value of the attributes can be represented by 8 bits without degrading the accuracy.

The use of the pipeline directive was also one the features used in our study. If we want decision trees to be suitable for real-time applications, those features can be used to respect deadlines by reducing the latency.

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>HLS Directive</th>
<th>Benchmark</th>
<th>% LUT</th>
<th>% FF</th>
<th>Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float</td>
<td>no pipeline</td>
<td>Iris</td>
<td>2.0</td>
<td>2.0</td>
<td>7051</td>
</tr>
<tr>
<td></td>
<td>Balance</td>
<td>11.0</td>
<td>3.0</td>
<td>730001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Iris</td>
<td>5.0</td>
<td>6.0</td>
<td>6081</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Balance</td>
<td>11.0</td>
<td>8.0</td>
<td>65002</td>
<td></td>
</tr>
<tr>
<td>Ap-ufixed&lt;8,6&gt;</td>
<td>no pipeline</td>
<td>Iris</td>
<td>1.5</td>
<td>1.3</td>
<td>7051</td>
</tr>
<tr>
<td></td>
<td>Balance</td>
<td>10.0</td>
<td>2.2</td>
<td>730001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Iris</td>
<td>1.7</td>
<td>4.0</td>
<td>6081</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Balance</td>
<td>11.0</td>
<td>6.5</td>
<td>65002</td>
<td></td>
</tr>
<tr>
<td>Ap-ufixed&lt;4,2&gt;</td>
<td>no pipeline</td>
<td>Iris</td>
<td>0.4</td>
<td>3.1</td>
<td>6081</td>
</tr>
<tr>
<td></td>
<td>Balance</td>
<td>9.0</td>
<td>4.1</td>
<td>65002</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: FPGA implementation results.

Table 2 shows that the type Float consumes more resources than ap-ufixed<8,6> and ap-ufixed<4,2>. The use of the pipeline directive, one of the directives of Vivado HLS, requires more resources but it ensures a better latency for a set of instances while maintaining a sequential access to the data set.

What we can conclude from the results presented in this section is that it is better to consider the missing data right from the learning phase (instead of considering it just in the classification phase). For the hardware implementation the type of attributes can affect both the accuracy and the use of resources. We notice that the type ap-ufixed<8,6> is a better choice in our experiments given that it ensures a good accuracy and uses less resources. The use of pipeline technique provides a better latency but consumes more resources than non-use of pipeline.

5. CONCLUSIONS

In this paper, we presented a framework which, to our knowledge, is original. It handles a process which can be summarized as follows: first the generation and handling of missing data, second the learning phase by generating the decision tree, third the automatic generation of the decision trees code, and finally the validation of the embedded decision trees with missing data by comparing the software and hardware results.

This framework gives us the possibility to deal with different types of attributes and can provide parallel implementations on FPGA by means of High Level Synthesis Tool, such as Vivado HLS. This latter ability allows generating the decision trees suitable for real-time embedded applications.

The perspective of generalizing the framework to different machine learning technologies was considered since the beginning. It can be extended to use Neural Networks (NNs) or Bayesian Networks (BNs) or Support Vector Machines (SVM) as machine learning models.

6. ACKNOWLEDGMENTS

This work was supported by the LATERAL laboratory, an industrial cooperation Thales Optronique (TOSA)/Lab-STICC of UBO.

7. REFERENCES

Quantifying the Flexibility of Real-Time Systems

Rafik Henia\(^1\), Alain Girault\(^2\), Christophe Prévote\(^{1,2}\), Sophie Quinton\(^2\), Laurent Rioux\(^1\)

\(^1\) Thales Research & Technology
\(^2\) Inria Grenoble Rhône-Alpes

1. INTRODUCTION

The life cycle of many safety or mission critical real-time systems, such as avionic systems, satellites or software defined radios, is superior to 10 years whereas the supporting ICT technologies have a much faster evolution rate. This mismatch between system life cycle on one side and software life cycle on the other side is a challenge for performance engineers. It is therefore essential for commercial offers to allow technology evolutions and upgrades of the architecture and functions after the initial system deployment. Besides, this must be done while preserving the satisfaction of timing performance requirements over the entire system lifetime.

Our objective is to develop novel techniques to quantify the ability of a real-time system to cope with future software evolutions, while remaining schedulable. We propose to define the flexibility of a system as its ability to schedule a new software function and show how this concept can be used to anticipate at design time schedulability bottlenecks that may prove problematic, after deployment, for system evolution. In this paper we focus on an evolution scenario where a single task is added to an existing system running on a single-core processor. Our task model here is restricted to independent periodic tasks with implicit deadlines and the scheduling policy is static priority preemptive. Of course our goal is to generalize our approach to more complex systems.

The research area that is most closely related to our problem is sensitivity analysis \([9, 2, 5, 7]\). Sensitivity analysis is used (1) to provide guarantees on the schedulability of a system in case of uncertainty on the system parameters, or (2) given a non schedulable system, to find a set of changes that lead to a schedulable system.

Most related work on sensitivity analysis addresses changes of one type of parameter, e.g., changes of task worst-case execution times (WCETs) or changes of periods — but not both. In \([2]\) for example, the authors analyze the sensitivity of a system to changes in the arrival frequency of all tasks, or in the WCET of all tasks. In \([7]\), the authors are interested in sensitivity analysis of systems with shared resources, for which they study the impact of an increase of the WCETs on system schedulability.

Dealing with different parameters (i.e. periods and priorities) is the approach used in \([8]\). Here the authors define the sensitivity using evolutionary algorithms and a stochastic approach. It will be interesting to compare our results with this method.

In contrast with the above mentioned approaches, we do not allow parameters of the deployed system to be modified in according with industrial practice. But we need to handle at the same time changes in the period, priority and WCET of the task to be added. As a result, the set of possible system configurations that need to be checked is much smaller than in classical sensitivity analysis and we hope to be able to handle much more complex systems that what state-of-the-art sensitivity analysis can handle.

Other papers like \([10, 1, 3]\) deal with flexibility related to task allocation, priority assignment or scenario based optimization. In \([3]\), the authors use concepts similar to ours to define an optimal or a robust priority assignment. In \([10]\), the authors deal with task allocation and priority assignment to maximize the extensibility of each task chain, where extensibility is the maximum execution time it is possible to increase the WCET before missing the deadline. In \([1]\) the flexibility of a system is defined according to scenarios. To define the flexibility it is necessary to define possible scenarios of change.

Related work that deals with flexibility is used to define at design time what is the best task allocation or priority assignment. But most of the time in an industrial context, the designer cannot take into account only timing constraints and some functionalities have fixed parameters. Moreover we are interested in system evolution when most parameters of the system are already defined. It is at this point unclear if our problem (adding a task) can be precisely encoded into existing methods dealing with increased execution times.

As a summary, by restricting ourselves to a simple change scenario with industrial relevance, we hope to come up with an approach that can provide guidance for the evolution of complex systems, which cannot be covered by existing sensitivity analysis techniques. Finally, to the best of our knowledge, no existing work discusses the notion of limiting task, i.e. the task which will miss its deadline first. The final aim is to help the designer by giving him information to allow a system to evolve while respecting timing constraints.

2. SYSTEM MODEL

Unless otherwise specified all the parameters defined in the following have positive integer values. In particular, we assume a discrete time clock.

We consider a uni-processor real-time system \(S\) consisting of a finite set of \(n\) independent tasks scheduled according to the Static Priority Preemptive (SPP) scheduling policy. Each task \(\tau_i, i \in [1, n]\), is defined by a tuple \((\pi_i, T_i, C_i)\) where \(\pi_i\) is the priority, \(T_i\) the period, and \(C_i\) an upper-bound on the worst case execution time (WCET). Each task \(\tau_i\) has an implicit deadline \(D_i = T_i\), meaning that a given activation of \(\tau_i\) must finish before the next activation of \(\tau_i\).
We assume that different tasks have different priorities and use as convention that \( \pi_i < \pi_j \) means that \( \tau_i \) has a higher priority than \( \tau_j \). For the purpose of our evolution scenario, we also suppose that for any two tasks \( \tau_i \) and \( \tau_j \) with \( \pi_i < \pi_j \), it is always possible to define a new task \( \tau_{new} \) with \( \pi_i < \pi_{new} < \pi_j \). This is done without loss of generality as priorities are only used to define a total order on tasks.

For each \( \tau_i \) in \( S \), \( hp(i) \) is the set of tasks of \( S \) that have a higher priority than \( \tau_i \), while \( lp(i) \) is the set of tasks of \( S \) that have a lower priority than \( \tau_i \).

The execution of a task \( \tau_i \) is triggered periodically with period \( T_i \) and each activation of \( \tau_i \) generates a new instance. A task instance is defined by its activation time, possible preemption delays, and finish time. Preemption delays are due to the task instance being blocked by higher priority task instances. Each instance of \( \tau_i \) finishes at the latest after having been scheduled (i.e., not counting the preemption delays) for \( C_i \) units of time.

### 3. PROBLEM STATEMENT

We are interested in system evolutions that may happen after system delivery. We focus here on a simple evolution scenario where a new task \( \tau_{new} \) defined by \((\pi_{new}, T_{new}, C_{new})\) is added to a schedulable system \( S \), thus yielding the system \( S_{new} = S \cup \{ \tau_{new} \} \). From now on, \( \tau_{new} \) will denote the task \((\pi_{new}, T_{new}, C_{new})\) and \( S_{new} \) will denote the system \( S \cup \{ \tau_{new} \} \).

We wish to answer the following question: Given a schedulable system \( S \) and a task \( \tau_{new} \) with unknown parameters: how can we quantify the flexibility of \( S \), that is, its ability to accommodate \( \tau_{new} \) while remaining schedulable?

To address these issues, we start below by formalizing the notions of evolution and flexibility of a system.

**Definition 1.** Evolution: An evolution of a system \( S \) is a task \( \tau_{new} \) where \( \pi_{new} \) belongs to \([\min \pi_i - 1, \max \pi_i + 1] \backslash \{ \pi_i \}\), and \((T_{new}, C_{new})\) belongs to \( \mathbb{N}^+ \times \mathbb{N}^+ \), such that the CPU load after the evolution does not exceed 100%:

\[
\frac{C_{new}}{T_{new}} + \sum_i \frac{C_i}{T_i} \leq 1
\]

**Definition 2.** Valid evolution: \( S \) being a schedulable system, an evolution \( \tau_{new} \) is valid for \( S \) if \( S_{new} \) is schedulable.

We denote by \( F_S \) the set of all valid evolutions for system \( S \). Note that if \( \tau_{new} \) is in \( F_S \), then for any \( 0 < C'_{new} < C_{new} \), the task \((\pi_{new}, T_{new}, C'_{new})\) is also in \( F_S \). Moreover, for any \( T'_{new} > T_{new} \), the task \((\pi_{new}, T'_{new}, C_{new})\) is also in \( F_S \). The flexibility of a system quantifies its ability to schedule a new task. We focus in this paper on the flexibility w.r.t. \( C_{new} \).

The notion of flexibility w.r.t. \( T_{new} \) is left for future work.

**Definition 3.** Flexibility: Let \( S \) be a schedulable system. The flexibility of \( S \) is the partial function \( flex_S \) from \( \mathbb{N}^+ \times \mathbb{N}^+ \) into \( \mathbb{N}^+ \) such that \((\pi_{new}, T_{new}, flex_S(\pi_{new}, T_{new}))\) is in \( F_S \) and \( flex_S(\pi_{new}, T_{new}) \) is maximal.

The function \( flex \) returns, for all \((\pi_{new}, T_{new})\), the value of the maximum WCET such that \( S \cup \{ (\pi, T, flex_S(\pi, T)) \} \) is schedulable. This function is partial as such an WCET may not exist. In the rest of the paper, we first focus on methods for computing \( flex \) and then we will show how we can use it for system design.

### 4. SLACK ANALYSIS

Let us first briefly recall the standard analysis used to establish the schedulability of systems of independent periodic tasks on a single processor under the SPP policy [6].

The response time of an instance of a task \( \tau_i \) is the delay between its activation and its finish time. The worst case response time of \( \tau_i \) (WCRT) is the maximum response time over all instances of \( \tau_i \); it is denoted \( r_i \). A system is schedulable if \( r_i \leq T_i \) for any task \( \tau_i \).

A critical instant of \( \tau_i \) is an activation scenario that maximises the response time of \( \tau_i \). A task is said to be schedulable if and only if its WCRT is smaller than its (implicit) deadline: \( r_i \leq T_i \). For the systems we consider, a critical instant of \( \tau_i \) occurs whenever an activation of a task instance occurs simultaneously with the activation of all higher-priority tasks. The WCRT of \( \tau_i \) can thus be obtained by computing:

\[
r_i = C_i + \sum_{j \in \{ hp(i) \}} \left\lfloor \frac{r_i}{T_j} \right\rfloor C_j
\]

To quantify the flexibility of a system, we start by introducing the slack of a task, which takes into account all the preemptions from higher priority tasks [4].

**Definition 4.** The slack of a task \( \tau_i \) is the maximum value it is possible to increase \( C_i \) while keeping \( \tau_i \) schedulable.

The slack \( Sl_i \) of a task \( \tau_i \) can be computed using the algorithm presented in [4].

**Theorem 1.** Let \( S \) be a schedulable system and \( \tau_{new} \) a task. \( S_{new} \) is schedulable if:

\[
\forall \tau_i \in \{ hp(new) \}, \left\lfloor \frac{T_i}{T_{new}} \right\rfloor C_{new} \leq Sl_i \tag{2}
\]

and

\[
\tau_{new} \leq T_{new} \tag{3}
\]

**Proof sketch of Theorem 1.** We have to prove that, for each \( \tau_i \in S \cup \{ \tau_{new} \} \), we have \( r_i \leq T_i \). Since \( S \) is schedulable, this holds for each \( \tau_i \in \{ hp(new) \} \). For \( \tau_{new} \) itself, this holds directly by Eq. (3). And for each \( \tau_i \in \{ hp(new) \} \), the preemptions of the tasks in \( hp(\ell) \) are already taken into account in the slack \( Sl_i \), hence there only remains to take into account the preemptions of \( \tau_{new} \) on \( \tau_i \), which is precisely the goal of Eq. (2).

This gives a sufficient but non necessary condition because the slack \( Sl_i \) is a lower bound on the slack available for \( \tau_i \). The reason the criterion is not necessary is that the preemption delays induced by \( \tau_{new} \) may be overapproximated.

### 5. FLEXIBILITY ANALYSIS

Recall that the flexibility of a system is quantified through the function \( flex \). We now show how to approximate \( flex \) using the above sufficient schedulability condition for \( S_{new} \).

**Definition 5.** For a given period \( T_{new} \) and priority \( \pi_{new} \) of \( \tau_{new} \), denote

- \( C_S \) the largest WCET of \( \tau_{new} \) such that \( S \) is schedulable after evolving,
• $C_{\tau_{\text{new}}}$ the largest WCET of $\tau_{\text{new}}$ such that $\tau_{\text{new}}$ is schedulable after evolving.

**Theorem 2.** $S$ being a system and $\tau_{\text{new}}$ an evolution, let $C_{S_{\text{new}}}^\text{max}$ be the largest WCET allowed by Eq. (2) i.e. a lower bound of $C_S$. Let $C_{\tau_{\text{new}}}^\text{max}$ be the largest WCET allowed by Eq. (3) i.e. a lower bound of $C_{\tau_{\text{new}}}$. Then

\[
C_{S_{\text{new}}}^\text{max} = \min \left\{ C_{S_{\text{new}}}^\text{max}, C_{\tau_{\text{new}}}^\text{max} \right\}
\]

is a lower bound of $\text{flex}_S(\tau_{\text{new}}, T_{\text{new}})$.

**Proof sketch of Theorem 2.** Theorem 1 gives a guarantee on the schedulability of $S_{\text{new}}$ depending on $\tau_{\text{new}}$ which is a sufficient non necessary condition. As a consequence, Theorem 2 based on Theorem 1 is also a sufficient and non necessary condition, i.e., a lower bound of the largest $C_{\tau_{\text{new}}}$ that guarantees the schedulability of $S_{\text{new}}$ for fixed $\pi_{\text{new}}$ and $T_{\text{new}}$ — which is given by $\text{flex}_S$.

We can compute both $C_{S_{\text{new}}}^\text{max}$ and $C_{\tau_{\text{new}}}^\text{max}$ using Theorem 3.

**Theorem 3.** Let $S$ be a schedulable system and $\tau_{\text{new}}$ a task. Then we have $C_{S_{\text{new}}}^\text{max}$ and $C_{\tau_{\text{new}}}^\text{max}$:

\[
C_{S_{\text{new}}}^\text{max} = \min_{\tau_i \in lp(\text{new})} \left\{ \frac{\text{Sl}_i}{\tau_{\text{new}}} \right\}
\]

and

\[
C_{\tau_{\text{new}}}^\text{max} = T_{\text{new}} - \sum_{\tau_j \in lp(\text{new})} \left\{ \frac{T_{\text{new}}}{T_j} C_j \right\}
\]

**Proof.** Eq. (2) implies that, for all task $\tau_i$ in $lp(\text{new})$, we have $C_{\text{new}} \leq \text{Sl}_i / \left( \frac{T_{\text{new}}}{T_j} \right)$. Since $C_{\text{new}}$ must be in $\mathbb{N}^+$, it follows that

\[
C_{\text{new}} \leq \min_{\tau_i \in lp(\text{new})} \left\{ \frac{\text{Sl}_i}{T_{\text{new}}} \right\}
\]

hence Eq. (4). We found the largest $C_{\text{new}}$ by computing the largest $\text{Sl}_{\text{new}}$. Again, since $C_{\text{new}}$ must be in $\mathbb{N}^+$, this leads to Eq. (5).

Intuitively, if $\tau_{\text{new}}$ has a high priority, then it will be easy to schedule but it will have a stronger impact on $S$. And this impact will be greater if the period $T_{\text{new}}$ is small. In contrast, a task $\tau_{\text{new}}$ with a low priority will have a smaller impact on the other tasks and it will be harder to schedule with a small period.

Note that the function $\text{flex}_S$ is partial. Both Conditions (4) and (5) on the flexibility of $\tau_{\text{new}}$ imply that some tuples $(\pi_{\text{new}}, T_{\text{new}})$ lead to non schedulable systems. These tuples imply that at least one task in $S_{\text{new}}$ will have not enough time to be executed. This is the case in a system if we add $\tau_{\text{new}}$ with a high priority and a small period.

### 6. LIMITING TASK OF THE SYSTEM

We now focus on the fact that for a priority $\pi_{\text{new}}$ and a period $T_{\text{new}}$, $C_S^\text{max}$ is limited by one, or possibly several tasks in $lp(\text{new})$. We call the lowest priority task among them the limiting task.

**Definition 6.** The limiting task of $S$ w.r.t. a priority $\pi_{\text{new}}$ and a period $T_{\text{new}}$ denoted $\tau_\ell$, is the lowest priority task $\tau_\ell$ such that:

\[
\ell = \arg \min_{\tau_i \in lp(\text{new})} \left\{ \frac{\text{Sl}_i}{\tau_{\text{new}}} \right\}
\]

Note that our definition of limiting task is based on the sufficient condition in Theorem 1. As a result, it is a good indicator of where the bottlenecks for software evolution are, but it is not an exact criterion (yet).

We are particularly interested in the fact that some tasks never limit $C_S^\text{max}$, whatever the priority and period of $\tau_{\text{new}}$.

**Theorem 4.** Consider the system $S$. The tasks that will never limit the system are all the tasks $\tau_i$ such that:

\[
\exists \tau_j \in S \text{ s.t. } \tau_j \in lp(i) \land T_j \leq T_i
\]

**Proof sketch of Theorem 4.** Tasks for which there exists another task of lower priority (hence with more preemptions) and lesser or equal period (hence with less or the same available time to finish) can never be the limiting task.

A good strategy to increase the flexibility of the system is to decrease the interference of higher priority tasks on the limiting task $\tau_i$, i.e., to increase its slack. One source of complexity here is that the limiting task is not necessarily the same for different values of $T_{\text{new}}$ and $\pi_{\text{new}}$.

We therefore identify the values of the period $T_{\text{new}}$ upon which $\tau_i$ may change. The set of these values is:

\[
\mathcal{F} = \left\{ t \in \mathbb{N}^+ \mid \exists \tau_i \in S, \left[ \frac{T_i}{T} \right] \neq \left[ \frac{T_i}{t-1} \right] \right\}
\]

This corresponds to the values of $T_{\text{new}}$ for which the number of preemptions of $\tau_i$ on $\tau_j$, with $j \in lp(\text{new})$, changes. It follows that $T_{\text{new}}$ remains constant in any interval $[T_{\text{new}}^l, T_{\text{new}}^u]$ where $T_{\text{new}}^l$ and $T_{\text{new}}^u$ are two consecutive periods in $\mathcal{F}$. It is thus sufficient to compute the limiting task of $S$ w.r.t. each priority $\pi_{\text{new}}$ and period of $T_{\text{new}} \in \mathcal{F}$ using Eq. (6).

### 7. CASE STUDY

Let us now show how the concepts presented in this paper are applied on an example. Consider a system:

\[
S = \{(2, 10, 1), (4, 5, 1), (6, 15, 1), (8, 10, 2), (10, 30, 2)\}
\]

**Case 1: Priority and period of $\tau_{\text{new}}$ are known**

Suppose that we want to add a task $\tau_{\text{new}}$ to $S$, with $\pi_{\text{new}} = 1$ and $T_{\text{new}} = 5$. By applying Theorem 3 and Definition 6 we determine that $C_S^\text{max} = 1$ and $\tau_1$ is the limiting task of $S$. This means that $\tau_{\text{new}}$ should not have an execution time larger than 1 otherwise $\tau_1$ may miss its deadline.

**Case 2: $C_S^\text{max}$ as a function of $T_{\text{new}}$**

Let us now assume that we know nothing about $\tau_{\text{new}}$ and we want to quantify the flexibility of $S$.

Table 1 shows the values of $C_S^\text{max}$ obtained for the possible values of $\pi_{\text{new}}$. Whenever any WCET larger than 0 would make one task in $S$ unschedulable we put $\perp$. Note that we group periods of $\tau_{\text{new}}$ for which the number of preemptions of tasks in $S$ does not change, based on the computation of $\mathcal{F}$. In addition, if $\pi_{\text{new}} \geq 11$ then $\tau_{\text{new}}$ will have no impact on the system so we do not represent $C_S^\text{max}$.

Similarly, Table 2 shows $\text{flex}_S$ for possible values of $\pi_{\text{new}}$ and $T_{\text{new}}$ (we leave out periods larger than 15). Finally, based on Table 1 and Table 2 we easily obtain Table 3 which shows the possible values of $C_S^\text{max}$.
the new task; (2) identify, through the concept of limiting task, which task in the system will “break” first in case of a software update.

8. CONCLUSION

In this paper we have defined the flexibility of a system as its capability to schedule a new task. We also presented an approach to quantify the flexibility of a system. More importantly, we show that it is possible under certain conditions to identify the task that will directly induce the limitations on a possible software update. If performed at design time, such a result can be used to adjust the system design by giving more slack to the limiting task. We illustrate how these results apply to a simple system.

We plan to extend our work in several main directions. First, we want to replace our sufficient conditions for by giving more slack to the limiting task. We illustrate how such a result can be used to adjust the system design by on a possible software update. If performed at design time, we expect to be able to provide useful results even for such complex systems.

9. REFERENCES

Towards schedulability analysis of real-time systems with precedence constraints and different periods

Slim Ben-Amor  
INRIA de Paris  
slim.ben-amor@inria.fr  

Dorin Maxim  
INRIA de Paris  
dorin.maxim@inria.fr  

Liliana Cucu-Grosjean  
INRIA de Paris  
liliana.cucu@inria.fr  

ABSTRACT
The complexity of modern architectures has increased the timing variability of programs (or tasks). In this context new approaches based on probabilistic methods are proposed to decrease the pessimism by associating probabilities to the worst case values of the programs (tasks) time execution. In this paper we present preliminary work to extend the original work of Chetto et al. [3] on precedence constraints tasks to the case of tasks with probabilistic periods and arrival pattern.

1. INTRODUCTION
Time critical systems are currently facing an increased complexity of the hardware architectures with a direct impact on the timing variability of the programs (and tasks). This increased complexity of the hardware architectures is motivated by numerous new functionalities that others industries than time critical systems require. Unfortunately the time critical systems do not have any impact on the design of such hardware architectures and need to adapt their timing analysis. Mainly worst case reasoning in presence of larger timing variability is becoming importantly more pessimistic and this conclusion has motivated the appearance of mixed-criticality models [12] with different possible values for the execution time of a program (or task). To our best knowledge the only existing methods providing such estimation for the execution time of a program belong to the realm of probabilistic and statistical methods [1, 6–8]. In this context the concept of probabilistic worst case execution time (pWCET) of a program (or task) has been proposed [5] as well as associated new schedulability analyses for task systems with worst case execution times described by probability distributions [9]. This paper proposes a schedulability analysis of task systems with probabilistic arrival times and this in presence of precedence constraints. To our best knowledge there is currently no solution to this problem.

Related work: Precedence constraints are together with deadlines probably the most used real-time constraints. This is motivated by the reactivity that real-time systems should ensure. This reactivity to some sensors input is obtained by regularly checking the sensors for new inputs and then producing a result that is sent to the actuators. The order of the execution is usually imposed by precedence constraints.

The precedence constraints are defined by (directed) graphs and the numerous results [3, 4, 11, 13] indicate the importance as well as the maturity of this topic. The existing results cover schedulability analyses, scheduling policies in both uniprocessor and multiprocessor case as well as shared resources.

2. EXISTING MODEL OF TASKS WITH PRECEDENCE CONSTRAINTS
In Chetto et al. [3] the authors study the schedulability analysis of task systems with precedence constraints on one processor. Let \( \tau \) be a set of tasks \( \tau_i \) defined by a Tuple \((r_i, C_i, d_i)\), \( \forall i \in \{1, \cdots, n\} \), where \( r_i \) is the release time, \( C_i \) the execution time and \( d_i \) the deadline of \( \tau \). Let \( G \) be a directed acyclic graph \((\tau, E)\) defining a partial order between the tasks of \( \tau \). Two tasks \( \tau_i \) and \( \tau_j \) \((i \neq j)\) are related by a precedence if and only if \((\tau_i, \tau_j)\) corresponds to an edge in \( E \). All tasks have the same period and they are released only once.

For instance let \( \tau \) be a task system \( \tau = \{\tau_1, \cdots, \tau_4\} \) with an associated oriented precedence graph \( G \) as described in Figure 1.

![Figure 1: A graph describing the precedence constraints of a task system](image)

In order to decide the schedulability of these task systems the authors of [3] propose the modification of the releases and the deadlines of each task as follows:

\[
\tau_i^* = max(r_i, r_j + C_j : \tau_j \rightarrow \tau_i)
\]

(1)

\[
d_i^* = min(d_i, d_j^* - C_j : \tau_i \rightarrow \tau_j)
\]

(2)

The notation \( \tau_i \rightarrow \tau_j \) in equations (1) and (2) mean that there is an edge from \( \tau_i \) to \( \tau_j \) in graph \( G \). Thus, we can say that \( \tau_i \) is an immediate predecessor of \( \tau_j \) and that \( \tau_j \) is an immediate successor of \( \tau_i \).

Equation (1) imposes that the release of a task cannot be done before the release of all its successors (with respect to the partial order defined by \( G \)). Equation (2) imposes that a task cannot be scheduled after its successors.

Equation (1) is applied from the sources (the tasks without predecessors) of \( G \) to the sinks (the tasks without successors) of \( G \). Equation (2) is applied from the sinks of \( G \) to the sources of \( G \).

Earliest Deadline First (EDF) is the scheduling policy applied preemptively to the new task systems \( \tau^* \) containing only independent tasks and Theorem 1 describes a schedulability condition.
THEOREM 1. [3] Let \( \tau \) be a task set. \( \tau \) is schedulable under a preemptive uni-processor EDF scheduling if and only if \( \forall j \in \{1, \ldots, n\} \) and \( \forall i \in \{1, \ldots, n\} \) such that \( r_i \leq r_j \), \( d_i \leq d_j \),

\[
\sum_{k=1}^{k=n} C_k \leq d_j - r_i
\]  

Our contribution In [2] we have presented a probabilistic extension of the results presented in Section 2 by introducing probabilistic worst case execution times. Our previous extension does not allow different periods and in order to prepare a probabilistic description of the periods we are first considering the problem with sporadic arrivals and different minimal inter-arrival times. In the following section we present a task model supporting such extension and an associated architecture model.

3. TASK AND ARCHITECTURE MODEL

In this section we present our enriched task model and the architecture model.

3.1 Functional task model

The task model or the functional model describes functions and operations that may correspond to inputs, outputs and computing performed by the system. As previously functional blocks are related to each other through arrows indicating data dependencies and/or precedence constraints. Thus the model is somehow similar to a data flow model. Each function corresponds to a set of operations to be executed and it has temporal characteristics such as execution time and minimal inter-arrival time. We can distinguish two types of functions:

- Aperiodic or acyclic functions: as a general rule these functions are the input to the system and they can represent different operations like various sensors, buttons and external commands. For this reason the functions release are sporadic and a minimal inter-arrival time may be derived, for example as the refresh rate of the sensors (see Figure 2). Furthermore, the aperiodic functions have execution times directly related to the time required to acquire data from sensors.

- Cyclic functions: these functions are characterized only by an execution time and they are activated as soon as data is available at one of their inputs. They are called cyclic because they are scheduled periodically thus they are checked in a cyclic manner for data availability.

Starting from this flow model, end-to-end and latency constraints are defined on chains of functions. These constraints require that each function belonging to a chain is executed at least once within a time duration defined by the latency. More complex constraints can also be defined, for example imposing that a certain function is executed some number of time \( k \) every \( t \) time units.

3.2 Architecture model

Architecture models used in the industry are often different from the models used in academic research. This fact has several motivations such as the adaptation of the system (and hence of the model) to the surrounding environment and different types of constraints which differ from one domain to another. The details of each model needs to be studied in order to be able to propose an analysis that is adapted to that model. In this section we present a model of architecture commonly used by both academia and industry. Architectures belonging to this models are used in the avionics domain for instance to describe a flight management system.

![Figure 2: Functional model with corresponding operations](image)

![Figure 3: A joint model of architecture and tasks](image)
non-active even if new data has appeared since its last execution.

• If the activation of a function comes after checking its status then the function will be executed only for the next release of the associated task.

• A function may be within an ongoing execution when the next release of the task containing this function arrives. In this case we consider a non-drop mode and the function continues its previous execution once it is possible (when all higher priority functions are not active).

• The functional model may contain cycles in the sense that a function may wait for some data to arrive and part of this data is its own output. Thus, if other data is not available the function will use its last produced data. Cycles are allowed as they do not represent data dependences. Indeed a function may be activated once data has arrived on one of its inputs without being blocked if some inputs are missing. Moreover, in our model cycles don’t overuse the computation resources. In fact, executions of a function that represent a cycle is not successive to the point of overloading the processor because this function is scheduled according to the period of the task that contains it.

4. ANALYSIS

After the definition of our model we propose a feasibility analysis to verify the respect of the latency constraints on a functional flow or chain. Our proposal relies on a response time analysis. We compute the worst-case response time \( r_f \) of each function of the flow separately. Then, the worst-case response time \( R \) of the entire flow \( L \) is obtained according to the following equation:

\[
R = \sum_{f \in L} r_f + \sum_{f \in L} T_{task}(f) \tag{4}
\]

In fact, we sum up all the worst-case response times \( r_f \) of functions on the flow and we add the worst-case waiting time for a function from its activation to the begin of its execution. This duration is bounded by the period \( T_{task}(f) \) of the task that contains the analysed function. Finally, we compare the worst-case response time of the flow to the latency constraint to decide the system feasibility.

The computation of \( r_f \) is based upon the classical response time analysis used for tasks scheduled by a fixed-task priority algorithm that we adapt to our model. First, we proceed by period inheritance as described in Algorithm 1. Therefore, We use the inter-arrival times of the input functions as a period according to the worst-case analysis. Then, we set the function period \( T_f \) to the task period \( T_{task}(f) \) that contains it. If the function period is less than the task, the operation will be released and executed at every task period. Otherwise, if the task period is less than the function period, every two successive executions of the function can be separated by the greatest multiple of \( T_{task}(f) \) that remain less than the function period \( T_f \).

In order to compute the worst-case response time \( r_f \) of a given function \( f \), we add to that functions execution time \( C_f \) the impact of preemptions it may suffer. The preemption may be caused by the execution of a function belonging to a higher priority task or higher order function in the same task. The response time is initialized to \( C_f \) then it is computed iteratively by the following equation:

\[
r_f = r_f + \sum_{f' \in hp(f)} X_{f' \times C_{f'}} \tag{5}
\]

where \( X \) is a state vector of 0 and 1 indicating if the function \( f' \) is active or not. \( hp(f) \) is the set of functions that belong to higher priority task than the task containing \( f \) or that belong to the same task as \( f \) but have higher order.

At each iteration, we check if the response time is superior to the next activation date of one of the higher priority operations. If so, we take this activation as a preemption and we update the response time according to equation 5. If not, the response time analysis ends (the task can not be preempted anymore) and we analyze the next function of the functional flow. Notice that we stop the computation if the task response time is already larger than the latency constraint because the latency constraint is trivially unsatisfied.

If the function is not an input operation and it has no period or activation date, we save the status of this function in a vector. If the function is active, we take it into account during the preemption. This state vector \( X \) is updated after each function execution. Indeed, when a function ends its execution it becomes non-active while all of its successors become active.

The tasks of the model we study can be considered as tasks of the non-cyclic generalized multi-frame task model (NC-GMF) proposed by Moyo [10]. That is, a task of our system model has several possible execution times depending on the different activation combinations of its active functions. For the same reason, each instance of this task is released in a non cyclic manner. Nevertheless, an analysis based on the NC-GMF model would be very pessimistic as it does not consider the dependencies between the functional blocks. These dependencies have an impact on the possible combinations of active functions and consequently, on the execution time of tasks.

5. SIMULATOR AND RESULTS

In this section we evaluate the performance of the proposed schedulability analysis. To this extent we apply the analysis on a task-set conforming to the presented model. Also, in order to have a reference for comparison, we also simulate the task-set on an in-house simulator that we have developed.

5.1 Simulator

Similar to the majority of simulators that exist we try to imitate the passage of time through a variable that is incremented continuously. At each iteration we check which active task is at the head of the queue (i.e. has the highest priority) and we decrease the execution time of its job by one unit of time. At the same time the state of the system is updated, e.g. the priority queue is change according to new arrivals in the system.

As each task is composed of several functions, when a task is executed the precedence constraints of the functions need to be respected. If a function finishes its execution, we flag it as inactive and all its successors are activated. Then the next active function of the same task is executed. If the current highest priority task has no more active functions we move on to the next highest priority task.
Furthermore, when a task or an input function is activated we also calculate the time instant of its next activation. The next activation of a task is equal to the time of the current activation plus its period, but in the case of an input function we also add a random value to its current arrival time together with its minimal inter-arrival time in order to represent the sporadic nature of their arrivals.

In order to keep track of certain information from one iteration to another we also use several global variables to represent the state of the system:

- the status ‘active’ or ‘inactive’ of each function
- the ID of the active task with highest priority
- the order of the executing functions for each task
- the time instant for the next activation of each task
- the time instant for the next activation of each function

**Studied example:** The system that we analyze contains the functions described by the functional model on Figure 2 and by timing characteristics on the Table 1. The architecture model is describe in the Figure 4.

<table>
<thead>
<tr>
<th>Operation</th>
<th>MIT&lt;sub&gt;i&lt;/sub&gt;</th>
<th>C&lt;sub&gt;i&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Func_A1</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>Func_A2</td>
<td>150</td>
<td>15</td>
</tr>
<tr>
<td>Func_C1</td>
<td>-</td>
<td>25</td>
</tr>
<tr>
<td>Func_C2</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>Func_C3</td>
<td>-</td>
<td>10</td>
</tr>
</tbody>
</table>

**Table 1: Inter-arrival times and worst case execution times of the studied example functions**

![Figure 4: Architecture model of studied example](image)

### 5.2 First numerical results

While working on a proof for our analysis and in order to evaluate its pessimism we consider the task set given in the previous example. We estimate the latency values with our analysis for two given chains of functions and compare against the maximum observed values provided by our simulator. The calculated values are always larger than the observed ones. Nevertheless our analysis is up to 37% larger than the observed values. This over-estimation may be decreased by introducing probabilistic description since large execution times values for instance are rare events.

### 6. CONCLUSION

In this paper we have presented partial results on the schedulability analysis of precedence constraints tasks in presence of sporadic arrivals of data. Our tasks are partitioned on a processor and their executions is using a fixed-priority policy. The analysis extends the classical fixed-point reasoning for response time calculation and we compare its pessimism with respect to observed response time. This work is the first step towards for the proposing of a probabilistic model with different periods. As future work we plan to extend the analysis by introducing the probabilistic description of the timing parameters like execution times and periods.

### 7. REFERENCES


