Programming and analysis of critical real-time systems

Julien Forget

▶ **To cite this version:**

Julien Forget. Programming and analysis of critical real-time systems. Embedded Systems. Université de Lille, 2023. tel-04148643

HAL Id: tel-04148643
https://theses.hal.science/tel-04148643

Submitted on 3 Jul 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Lille

Mémoire d'Habilitation à Diriger des Recherches

Préparé au sein de *l'Université de Lille*

Spécialité *Informatique*

Présenté et soutenu par

Julien Forget

Le *Lundi 12 Juin 2023*

Programming and analysis of critical real-time systems

Devant le jury composé de :

Sebastian Altmeyer	Examinateur
Alan Burns	Rapporteur
Laure Gonnord	Examinatrice
Giuseppe Lipari	Garant
Christine Rochange	Rapportrice
Marc Pouzet	Président

École doctorale MADIS

Acknowledgements

I would first like to thank Alan Burns, Marc Pouzet, and Christine Rochange, for reviewing my manuscript. I learnt a lot from their work, I hope they learnt a few things in return reading this document. Many thanks to Sebastian Altmeyer for making the trip from Germany and for our stimulating discussions. My gratitude to Laure Gonnord for her comeback to the source, as well as for her guidance in dire times.

This work would not exist without my collaborators. Frédéric Boniol and Claire Pagetti kindly guided my first steps as a researcher. Emmanuel Grolleau and Pascal Richard helped me unravel the inner workings of the real-time community. Ph.D. students and post-docs have been the main drive for my research activities: many thanks to Antoine Bertout, Andrei Florea, Frédéric Fort, Sandro Grebant, Jordy Ruiz and Rémy Wyss. Our trio with Clément Ballabriga and Giuseppe Lipari was the most stimulating research environment I could have hopped for. A warm thank you to my colleagues from SyCoMoRES, SEAS, and Polytech, for countless scientific and non-scientific discussions, and for being such uniformly caring persons.

To my family, my southwestern second family, my mexican-connection, my adoptive northern family: I am so grateful that you are part of my life.

Contents

1. Overview

This document summarizes my research since my appointment as an associate professor in 2010 at the University of Lille. Since then, I have been doing my research in the CRIStAL laboratory (Centre de Recherche en Informatique, Signal et Automatique de Lille), and I have been teaching at the Polytech Lille engineer school. I was a member of the following research teams, chronologically:

1. The DaRT team (until 2011), whose focus was on high performance systems on chip;
2. The Émeraude team (until 2021), whose focus was on software systems embedded on heterogeneous hardware platforms;
3. The SyCoMoRES team (since 2021), whose focus is on modular analysis and development of real-time systems.

My research as an associate professor concerns the development of real-time systems. I focused on three main topics in this context. First, I studied programming languages dedicated to the development of critical real-time systems, and their compilation. Second, I studied high-level timing analysis of real-time systems modeled as a set of concurrent tasks. Third, I studied low-level timing analysis of the code of a single real-time task. Work presented in this document is the result of my collaborations with several colleagues from CRIStAL, Onera Toulouse, the University of Lyon, and the University of Poitiers.

1.1. Context

Real-time systems are computer systems that are required not only to produce the correct output values, as a reaction to the system inputs, but also to produce these values at the correct time. We can distinguish two subclasses of real-time systems: *hard real-time systems*, where failing to respect some constraints may have catastrophic consequences, and *soft real-time systems*, where failures only cause a degradation in the quality-of-service of the system. In my work, I focus on hard real-time systems. The flight-control system of an airplane, or the autonomous driving system of a car, are good examples of hard real-time systems.

Real-time systems are a sub-class of *reactive systems*, their behaviour consists in indefinitely repeating the following sequence: wait for inputs from sensors, compute a reaction, send outputs to actuators. At the highest level of abstraction, a real-time system can be represented as a set of concurrent tasks, where each task either senses, computes, or actuates.

Each task is subject to real-time constraints. *Periodicity constraints* dictate the rate at which tasks must be repeated. *Deadline constraints* are usually derived from periodicity constraints and define the latest date at which each repetition of a task, also called a *job*, must complete. Real-time constraints typically stem from the physical characteristics of the device under control. The rate at which a device must be controlled is upper-bounded by the maximum rate at which the sensors and actuators of the system can operate (e.g. the maximum samples per second of

a LIDAR). It is lower-bounded by the minimum rate required by the control/command laws of the system (e.g. the minimum rate to ensure the stability of an airplane). Since different sensors and actuators have different capabilities, and since the control/command of a system is usually structured as a collaboration of several laws, different tasks of the systems are subject to different real-time constraints. Choosing the actual task rates amounts to a compromise between the quality of the control/command laws and the overall computation demand. Once real-time constraints are set, reacting faster than these constraints usually does not improve the quality of the system. Instead, the main focus of hard real-time systems development is to ensure that the constraints will always be met, so as to ensure the system safety.

1.2. Contributions

The programming of hard real-time systems involves several research domains, which are traditionally studied by separate research communities. An important part of my work focuses on the connection between these domains, through the development of the Prelude language and its compiler. This section outlines my contributions and the three main domains they cover.

1.2.1. Programming with Prelude

Because real-time systems are critical, development based on formal methods has become popular in this domain. Formal models (e.g. timed-automata [AD94], timed-petri nets [Wan12], or synchronous programs [Ben+03]), provide unambiguous specifications of the system behaviour and enable to mathematically prove safety and security properties for it. Formally defined compilation chains (e.g. synchronous languages compilers [Hal+91; Bou+17]), automate the translation from a high-level formal model to lower-level code, and preserve the semantics of the formal model in the corresponding low-level code. This ensures that the properties proved on the model are still valid in the generated low-level code.

Synchronous languages [Ben+03] have proved successful for the development of critical embedded systems. Their structure is particularly well-adapted to the specification of reactive systems. Their formally defined semantics and compilers improve the system safety and security. However, they traditionally lack primitives to specify real-time constraints. Prelude is a synchronous language dedicated to the programming of real-time systems, which was designed during my Ph.D. thesis. It extends synchronous data-flow languages, such as Lustre [Hal+91], with language constructs to specify real-time constraints. Its compiler generates multi-threaded C code, to be executed by a real-time operating system. This document presents several extensions to my initial work on Prelude. They are summarized below.

Implementation on multi-core with distributed memory The Prelude compiler initially generated code for a mono-core hardware platform with the MarteOS operating system [RH01]. It has since then been extended to support several different types of hardware platforms and operating systems. In particular, during the internship of Frédéric Fort, the Prelude compiler was extended to generate code for a multi-core hardware with distributed memory (a larger central memory, along with smaller scratchpad memories for each core), with the Erika real-time operating system[1].

[1] http://erika.tuxfamily.org/drupal/

Partial delays specification In collaboration with colleagues from Onera Toulouse (Frédéric Boniol, Claire Pagetti, Rémy Wyss), the Prelude language was extended to support incomplete specifications. A new operator was introduced to enable to specify communications with flexible non-deterministic communication semantics (either synchronized or not). The compiler then chooses among several possible deterministic implementations, its objective being to satisfy end-to-end latency constraints.

Multi-mode multi-periodic systems In the Ph.D. of Frédéric Fort, we extended Prelude to support the programming of multi-mode real-time systems. In such a system, each mode corresponds to a different behaviour, to a different set of computation tasks, with different real-time constraints. The Prelude language and its clock calculus have been extended to support mode automata where tasks of the same mode can have different periodicity constraints.

1.2.2. High-level timing analysis

The Prelude compiler translates the input program into a C program structured as a set of concurrent real-time tasks. In order to ensure that the task set will always respect its real-time constraints at run-time, a timing analysis must be performed. This consists of several sub-analyses applied at different abstraction levels. *High-level timing analysis* considers a system modeled as a set of concurrent tasks, where each task is characterized solely by its real-time characteristics. *Low-level timing analysis* focuses on the code executed by one task and mostly abstracts from other tasks. The following contributions to high-level timing analysis are presented in this document.

Real-time scheduling Real-time scheduling considers a set of tasks characterized by their periods, deadlines, and worst-case execution time, and ensures that all tasks will meet their deadlines during execution. In collaboration with colleagues from the University of Poitiers (Emmanuel Grolleau, Pascal Richard) and from Onera Toulouse (Frédéric Boniol, Claire Pagetti), we studied the scheduling of dependent tasks, that is to say tasks related by precedence constraints, as it is the case for task sets generated by the Prelude compiler.

End-to-end constraints analysis While periodicity and deadline constraints are per-task constraints, end-to-end constraints involve a chain of tasks. For instance, an end-to-end latency constraint imposes an upper-bound to the total delay from the start time of the first task of the chain to the completion time of the last task of the chain. In collaboration with colleagues from Onera Toulouse (Frédéric Boniol, Claire Pagetti, Rémy Wyss), we proposed a general framework for the analysis of such properties.

Task clustering In a Prelude program, each functionality of the system is implemented as a *node*. The Prelude compiler then translates each node into a separate real-time task. However, for complex industrial systems, mapping each functionality to a different task would produce a large number of tasks, incurring a significant time and memory overhead. In the Ph.D. thesis of Antoine Bertout, we studied the *task clustering* problem, where the objective is to reduce the number of real-time tasks used in the node-to-task mapping, while preserving schedulability.

1.2.3. Low-level timing analysis

High-level timing analysis requires as input the Worst-Case Execution Time (WCET) of each task. This information is provided by a low-level timing analysis, which considers the code executed by a single task in order to determine its WCET. WCET analysis is most often performed by static analysis of the task code. Due to the difficulty of the considered problem, WCET computation by static analysis makes simplifications that yield an over-approximated, yet safe, value. Even though the higher-level of abstraction of source code would simplify WCET analysis, compiler optimizations have a huge impact on WCET, and this impact is hard to predict, thus binary code is usually analysed instead. The following contributions to low-level timing analysis are presented in this document.

Symbolic WCET computation Traditional WCET analysis produces a constant numeric upper-bound to the WCET. Thus, if some parameters of the program change (e.g program inputs, loop bounds, cache size, ...), the analysis must be re-run. An alternative approach is to produce a parametric WCET formula. In collaboration with colleagues from the CRIStAL laboratory (Clément Ballabriga, Giuseppe Lipari), we proposed a new approach to parametric WCET analysis based on symbolic computation. The original motivation of this work was to enable the design of adaptive real-time systems in Prelude: we compute off-line a WCET formula, instantiate the formula on-line, and adapt the system behaviour if the WCET reaches a certain threshold. Such an adaptive behaviour can be specified using the multi-mode extensions of Frédéric Fort.

Relational abstract interpretation of assembly code *Abstract interpretation* is a static analysis technique that provides a sound over-approximation of the possible behaviours of a program. *Relational* abstract interpretation establishes relations (e.g. linear relations) between variables of the program under analysis. Abstract interpretation is usually performed on the program source code. Instead, in collaboration with colleagues from the CRIStAL laboratory (Clément Ballabriga, Giuseppe Lipari), and from the University of Lyon (Laure Gonnord), and in particular during the post-doc of Jordy Ruiz, we proposed an abstract interpretation technique for the analysis of assembly code. The original motivation of this work was to establish relations between software parameters appearing in WCET formulae produced by our symbolic WCET analysis.

1.3. Reading this document

The structure of this document does not follow the chronology of my research, but instead groups my contributions by research domains: programming real-time systems with Prelude (chapter 3), high-level timing analysis (chapter 4), and low-level timing analysis (chapter 5). The document summarizes the main research results, but it is not a comprehensive presentation of my work. The reader should refer to my cited publications for more details. In particular:

- Proofs and intermediate lemmas are not presented, only the main theorems are provided;
- The presentation of related works is succinct and limited to works that are the most tightly related to my contributions;
- When possible, formal definitions are replaced by illustration through examples.

A *Conclusion* subsection is provided at the end of the presentation of each contribution. It highlights the main contribution, detailing where it was published, collaborations, tutored students, and funded research projects.

Glossaries listing the main symbols and acronyms used in each chapter are provided in the appendix part of this document.

Part I.

Introduction

2. Background on real-time systems

This chapter provides a description of the background on which my contributions on real-time systems rely.

The development of hard real-time systems usually involves several development teams, each separately responsible for the development of a subset of the system tasks. These tasks are then assembled together to form the complete system, at which point real-time properties and inter-task communication schemes are integrated. In Section 2.1, I present the Prelude language, which I designed and developed during my Ph.D. thesis for specifying how real-time tasks are integrated together to form a complete real-time system. In Section 2.2, I recall classic definitions and results on real-time task models and their scheduling. In Section 2.3, I provide a short summary on the Worst-Case Execution Time analysis of a task.

2.1. Prelude, a synchronous data-flow language with real-time primitives

Prelude builds upon synchronous data-flow languages, such as Lustre [Hal+91]. It inherits their clean and formal semantics, and extends them with language concepts dedicated to the description and management of real-time constraints. This section presents the core language defined during my Ph.D. thesis.

The Prelude compiler translates the input program into a set of concurrent and inter-dependent C tasks. The compilation process is defined formally, so as to ensure that the program semantics is preserved throughout the compilation (semantics preservation is only proved on paper, not with a proof assistant such as e.g. [Bou+17]). The generated code is independent of the target Operating System. Several options for the compiler back-end have been developed since the initial compiler release, in order to support various types of hardware architectures: single-core, multi-core, centralized memory, as well as distributed memory.

2.1.1. Informal presentation

Data-flow The data-flow nature of Prelude is reminiscent of Lustre, and illustrated in Figure 2.1. When the program receives new input values, it reacts by computing new output values. Variables and expressions denote infinite sequences of values called *flows*. Figure 2.1 details the values of each flow, for each reaction of the node `main`.

Computations described in the program *equations* (the `let...tel` block) are implicitly repeated indefinitely. In the example of Figure 2.1, whenever the program receives new values for the input flows `a` and `b`, it computes the new value of output flow o as the sum of the new values of `a` and `b`. Note that computation order depends on data-dependencies and on real-time constraints, not on text order (the node body is a set of unordered equations, not of sequential instructions): we need values for variables on the right-hand side of an equation to compute values for variables on the left-hand side. Execution order is determined by the compiler and the scheduler, based on the program data-dependencies and real-time constraints.

```
imported node add(a,b:int) returns(o:int) wcet 6;
imported node plus_one(a:int) returns(o:int) wcet 3;

node main(a,b: int)
returns(o, p: int)
let
  o=add(a,b);
  p=0 fby (plus_one(o));
tel
```

a	0	2	4	6	8	...
b	1	3	5	7	9	...
o	1	5	9	13	17	...
p	0	2	6	10	14	...

Figure 2.1.: The data-flow semantics of a Prelude program

Nodes are the structuring unit of the language, much like functions/procedures in procedural languages: a Prelude program consists of a set of nodes that can be instantiated (called) in the equations of other nodes. The *main* node is the entry point of the program, it communicates with the program environment by acquiring its inputs from sensors and applying its outputs on actuators. Because it is designed as an architecture description language, Prelude does not include arithmetic or logic operators. Instead, computations are performed by *imported nodes* (`add`, `plus_one`), whose behaviour is described outside the Prelude program. Imported nodes are the leaves of the nodes hierarchy, which will ultimately be translated into tasks/threads in the generated low-level code.

The program of Figure 2.1 consists of a single non-leaf node (the main node), although there can be several non-leaf nodes in general. This node instantiates two imported nodes (`add` sums its arguments and `plusone` adds one to its argument). The program also uses the builtin operator `fby`, which is the delay operator: the value of expression `cst fby e` is `cst` for the first reaction *followed by* the value of `e` at its previous reaction.

Synchronous real-time Real-time constraints can be specified in Prelude as illustrated in Figure 2.2. The programmer either declares the rate of a flow explicitly (`c: int rate(40,0)`) or lets it be inferred by the compiler. A rate declaration specifies the *period* of a flow along with its *phase*, thus defining the dates at which the flow produces values. In the example, inputs a, b have period 40, while input c has period 20.

Due to the data-flow nature of the language, the execution rate of a node depends on that of its inputs. More precisely, when instantiating an imported node, all its inputs are required to be *synchronous*, that is to say they must have the same rate. This rate is also the execution rate of the node instance. In the example of Figure 2.2, the execution rate of node instance `add` in node `main` is (40,0), because both its inputs have rate (40,0). This is also the rate of its outputs. Trying to apply `add` to `a` and `c` would cause a compilation error, since they do not have the same period: for instance, the value of a cannot be accessed at date 20, while the value of b can.

Worst-case execution times (WCET) are specified in the declaration of imported nodes. The programmer must also specify a WCET for each input (sensor) and output (actuator) of the

```
imported node add(a,b:int) returns(o:int) wcet 6;
imported node plus_one(a:int) returns(o:int) wcet 3;
sensor a wcet 5; sensor b wcet 5; sensor c wcet 4;
actuator o wcet 1; actuator p wcet 1;

node main(a,b: int rate (40,0); c: rate (20,0))
returns(o, p: int)
let
  o=add(a,b);
  p=0 fby (plus_one(c));
tel
```

date	0	20	40	60	80	…
a	0		1		2	…
b	2		4		6	…
c	1	3	5	7	9	…
o	2		5		8	…
p	0	2	4	6	8	…

Figure 2.2.: A program with inputs with two different rates

main node. WCETs are only used for the schedulability analysis of the program (see Section 4.1 for details), they are not enforced at execution.

Rate transitions *Rate transition* operators are used to combine flows that have different rates, as illustrated in Figure 2.3. In expression `e*^k` the operator $*\hat{}$ *over-samples* expression `e`, duplicating each value of `e` k times. Conversely, in `e/^k` the operator $/\hat{}$ *under-samples* `e`, keeping only one out of k successive values of `e`. In the example, the compiler infers that the rate of `swap` is $(50, 0)$ (because its input `i` has rate (50,0)), and that the rate of `id` is $(150, 0)$ (because `vf` has the same rate as `swap`, and thus `vf/^3` has rate (150,0)). Since flow `vs` has rate (150,0), it is oversampled before being passed as input to `swap` so as to be synchronous with (`i`). A delay (`fby`) is also applied, to avoid overconstraining the deadline of `id`. Without this delay, `id` would have to complete no later than the deadline of `swap`. The imported nodes *swap* and *id* are defined outside Prelude such that $swap(i, j) = (j, i)$, and $id(i) = i$.

```
imported node swap(i, j: int) returns (o, p: int) wcet 10;
imported node id(i: int) returns (o: int) wcet 15;
sensor i wcet 5; actuator o wcet 5;

node sampling(i: rate(50,0)) returns (o)
  var vf, vs;
let
  (o, vf)=swap(i, (5 fby vs)*^3);
  vs=id(vf/^3);
tel
```

date	0	50	100	150	200	250	300	...
i	0	1	2	3	4	5	6	...
vf	0	1	2	3	4	5	6	...
vs	0			3			6	...
o	5	5	5	0	0	0	3	...

Figure 2.3.: Rate transition operators

Phase offsets The phase of an expression e can be shifted by a fraction q using the construction `e~>q`. This effectively shifts every value of e by q multiplied by the period of e.

Activation conditions In addition to activation rates (periods and phases), Prelude supports Boolean activation conditions à la Lustre (see Figure 2.4). Expression `e when c` only keeps the values of `e` when `c` is true. Expression `merge(c,e1,e2)` merges two expressions that have complementary activation conditions: when `c` is true the expression is equal to `e1`, when `c` is false it is equal to `e2`. For instance, in Figure 2.4, node `add` is evaluated only if `c` is true, otherwise `plusone` is evaluated. Again, because the language is data-flow, the activation condition of an imported node instance is that of its inputs. Note that flows `ctrue` and `cfalse` have complementary activation condition, ie either one produces a value or the other does, but not both. They are merged to compute the flow `o` which always produces values at rate (40,0).

```
imported node add(a,b:int) returns(o:int) wcet 6;
...
node main(a,b: int rate (40,0); c: bool) returns(o: int)
  var ctrue, cfalse;
let
  ctrue=add(a when c,b when c);
  cfalse=plus_one(a whennot c);
  o=merge(c,ctrue,cfalse);
tel
```

date	0	40	80	120	160	...
a	0	1	2	3	4	...
b	2	4	6	8	10	...
c	T	T	F	T	F	...
ctrue	2	5		11		...
cfalse			3		5	...
o	2	5	3	11	5	...

Figure 2.4.: Activation conditions

2.1.2. Formal semantics

The Prelude model of flows and clocks is based on the *tagged-signal model* [LSV96]. A *flow* is defined as a set of pairs $(v_i, t_i)_{i\in\mathbb{N}}$, where v_i is a value in some set $\mathcal{V}$ and the tag t_i (in $\mathbb{N}$) represents a date associated to v_i. Tags define the order in which values are produced. Intuitively, the value v_i is the value carried by the flow in interval $[t_i, t_{i+1})$, where t_{i+1} is the smallest tag in the flow that is greater than t_i. The *clock* of a flow is its projection on $\mathbb{N}$. Two flows are *synchronous* iff they have the same clock.

The formal definition of real-time constraints in Prelude relies on a dedicated class of clocks called *Strictly Periodic clocks*, defined as follows:

Definition 2.1.1 (Strictly Periodic Clock)**.** A strictly periodic clock is denoted as a pair (n, p), with n, p in $\mathbb{N}$, and:

- The infinite sequence of tags generated by (n, p), denoted $(n, p)^{\#}$, is defined as follows: $(n, p)^{\#} = \{n * i + p \mid i \in \mathbb{N}\}$.
- $\pi(n, p) = n$ is the *period* and $\varphi(n, p) = p$ is the *offset* of (n, p).

Operators on strictly periodic clocks are illustrated in Figure 2.5. The acceleration $(*.)$, deceleration $(/.)$, and phase offset $(\rightarrow.)$, are defined as follows:

Definition 2.1.2 (Strictly Periodic Clock Operators)**.** Let (n, p) be a strictly periodic clock, and k be in $\mathbb{N}$. Then, by definition:

$$
\begin{aligned}
(n, p) *. k &= (n/k, p) \\
(n, p) /. k &= (n * k, p) \\
(n, p) \rightarrow . k &= (n, p + k)
\end{aligned}
$$

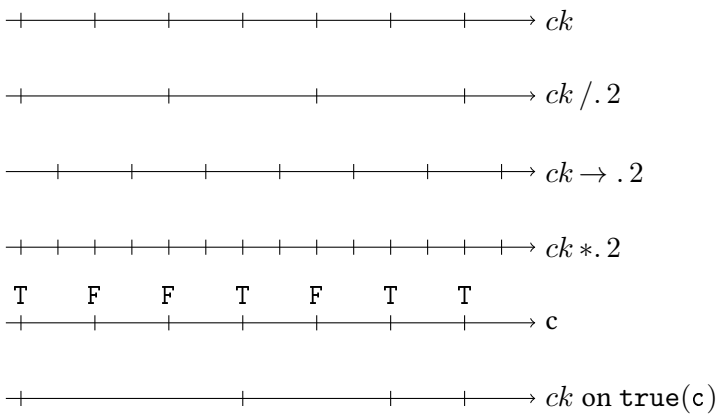


Figure 2.5.: Strictly periodic clocks and clock operations

Prelude combines strictly periodic clocks, which define the activation rate of a flow, with Boolean clocks, which define the activation condition of a flow. To formalise Boolean clocks, we adapt the definition of the clock operator *on* from [CP03] as follows.

Definition 2.1.3. Let ck be a clock, c be a flow whose values are of some enumerated type *ty*, and $C \in ty$. Then, by definition:

$$(ck \text{ on } C(c))^{\#} = \{t \mid t \in ck^{\#} \wedge (C, t) \in c^{\#}\}$$

We will now detail the formal semantics of the language operators. We let $\widehat{s}$ denote the clock of flow s. The term $\diamond^{\#}(s_0, \ldots, s_n)$ denotes the flow resulting from the application of the operator $\diamond$ on flows $s_0, \ldots, s_n$. The formal semantics of Prelude operators is detailed in Figure 2.6 (op_f denotes an operator over scalars from the compiler target language). The relation $x \text{ div } y \Leftrightarrow y \bmod x = 0$ reads as "x divides y". The denotational semantics of Prelude is rather standard and can be found in [For09].

$$\begin{aligned}
op^{\#}(s_0, \ldots, s_n) &= \{(op_f(v_0, \ldots, v_n), t) \mid (v_0, t) \in s_0^{\#}, \ldots, (v_n, t) \in s_n^{\#}\} \\
*^{\wedge\#}(s, k) &= \{(v, t + i * \pi(\widehat{s})/k) \mid (v, t) \in s^{\#}, i \in [0..k)\} \\
/^{\wedge\#}(s, k) &= \{(v, t) \mid (v, t) \in s^{\#} \wedge (t - \varphi(\widehat{s})) \text{ div } (\pi(\widehat{s}) * k)\} \\
\sim>^{\#}(s, k) &= \{(v, t + k) \mid (v, t) \in s^{\#}\} \\
\texttt{fby}^{\#}(v, s) &= \{(v, t_0)\} \cup \{(v_i, t_{i+1}) | (v_i, t_i) \in s^{\#}\} \quad (t_0 \text{ the smallest tag in s}) \\
\texttt{when}^{\#}(s, c, C) &= \{(v, t) \mid (v, t) \in s^{\#}, t \in (\widehat{s} \text{ on } C(c))^{\#}\} \\
\texttt{merge}^{\#}(c, s_0, s_1) &= s_0^{\#} \cup s_1^{\#}
\end{aligned}$$

Figure 2.6.: Semantics of Prelude operators

Note that the semantics of some operators is well-defined only if the clock of its operands respect some specific clock constraints. For instance, $\texttt{merge}^{\#}$ requires flows that do not bear

values at the same dates. Node application requires arguments that are synchronous, i.e. that have the same clock. $*^{\wedge\#}(s,k)$ is defined iff k div $\pi(\widehat{s})$, etc. The clock calculus (see below) is responsible for checking such clock constraints and inferring the clocks of the program.

2.1.3. Compilation overview

The structure of the Prelude compiler follows the classic decomposition into a front-end, which checks the program validity, and a back-end, which translates a valid input program into C code. The front-end starts with a standard syntax analysis, and ML-like type inference [Pie02]. Then it performs two analyses that are specific to synchronous data-flow languages: *clock calculus* and *causality analysis*. The causality analysis checks that the program data-dependencies do form *immediate* cycles (cycles that do not contain at least one delay), its definition is similar to that of Lustre [HRR91]. The clock calculus computes a clock for each element of the program (variables, expressions, etc). In doing so, it verifies that clock synchronization constraints are respected, which ensures that flow values are only accessed at dates at which they are well-defined. The clock calculus is implemented as a type inference system, where the usual types are replaced by clock types, and with some noticeable differences such as subtyping rules, and arithmetic simplifications of periodic clock expressions. For more details, refer to [For+08; For09].

The back-end consists of two steps. First, the program is translated into a *Task Set* intermediate representation. Basically, each imported node is translated into a real-time task, where periods and offsets are deduced from clocks, while deadlines and WCETs are deduced from the corresponding declarations in the program. In addition, tasks are related by data-dependencies due to the program causality constraints. For instance, Figure 2.7 represents the task graph obtained for the program of Figure 2.3. Boxes represent tasks, edges represent data-dependencies and are annotated with operators that detail how tasks communicate. Tasks i, o and $swap$ have period 50, while task id has period 150. The task relative deadlines are equal to their periods. Data-dependencies have a *causal* semantics, which means that a task producing data must complete before task(s) consuming this data can start.

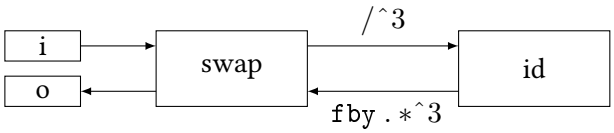


Figure 2.7.: Task graph for the program of Figure 2.3

In the second step of the back-end, the task set is translated into C code. This includes a fairly trivial generation of a data-structure describing the characteristics of each task (akin to a Process Control Block). More importantly, a tailor-made communication protocol is generated for each inter-task data-dependency to ensure that task communications respect the synchronous semantics. The code generated for each protocol depends on the operator annotations on the corresponding data-dependency. See [For09] for more details on the compiler back-end.

2.2. High-level timing analysis of a real-time task set

The code generated by Prelude is structured as a set of real-time tasks. This section recalls classic definitions for real-time tasks and provides an introduction to real-time scheduling.

In the classic model of the real-time scheduling theory [BW01], a real-time task is characterized by its duration (Worst-Case Execution Time), its repetition period, and its deadline (relative to its period). In order to take the functional semantics of the program into account, we also add data-communications to this model. To ensure the functional determinism of the program, we need to control the order in which communicating tasks execute. Typically, data-production must precede data-consumption, so data-communications induce precedence constraints. As a consequence, we consider a *dependent task model*, that is to say tasks whose start time depends on the completion time of other tasks.

For more details on real-time scheduling, the reader unfamiliar with this topic can refer to my introductory course at École Temps Réel 2017 [For17].

2.2.1. Real-time attributes

The software architecture of a real-time system can be defined as a set of tasks denoted $\mathcal{S}= \{\tau_i(O_i, C_i, D_i, T_i)\}_{0\leq i<n}$. O_i is the first release date of the task τ_i, also called *offset* in the literature. T_i is the *period* of the task and defines the exact duration between two successive releases of the task. We denote $\tau_{i.k}$ the k^{th} ($k \geq 0$) repetition, or *job*, of τ_i. The job $\tau_{i.k}$ is released at the date $o_{i.k}= O_i + kT_i$. D_i is the relative deadline of the task, every job $\tau_{i.k}$ must be completed before its absolute deadline $d_{i.k}= o_{i.k} + D_i$. We denote $\mathcal{J}$ the set of jobs (generated by $\mathcal{S}$). Finally, C_i is the worst-case execution time (WCET) of the task and represents the longest possible processing time required to compute a job of τ_i. These definitions are illustrated in Figure 2.8. Additionally, we define the *hyperperiod* H of a task set as the least common multiple (lcm) of the task periods.

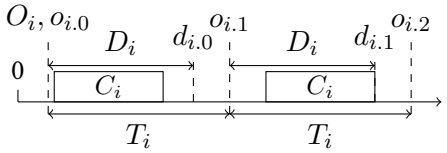


Figure 2.8.: Real-time attributes

2.2.2. Precedence constraints

Precedence constraints impose constraints on the relative execution order of tasks. In the following, we distinguish simple and extended precedence constraints.

2.2.2.1. Simple precedence constraints

Precedence constraints that relate tasks with the same period are called *simple precedence constraints*. They are formalized by a relation $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$, where $\tau_i \rightarrow \tau_j$ states that for all $k \in \mathbb{N}$, $\tau_{i.k}$ must complete before $\tau_{j.k}$ starts. As such, precedence constraints define a partial order between tasks. We assume that the graph of precedence constraints is acyclic (a Direct Acyclic Graph, or DAG), otherwise the system is not causal, meaning that there exists no execution order that respects all the precedence constraints. We define the predecessors of a task τ_i as $preds(\tau_i)= \{\tau_j | \tau_j \rightarrow \tau_i\}$ and its successors as $succs(\tau_i)= \{\tau_j | \tau_i \rightarrow \tau_j\}$. We let $\tau_i \overset{*}{\rightarrow} \tau_j$ denote the transitive closure of $\rightarrow$.

2.2.2.2. Extended precedence constraints

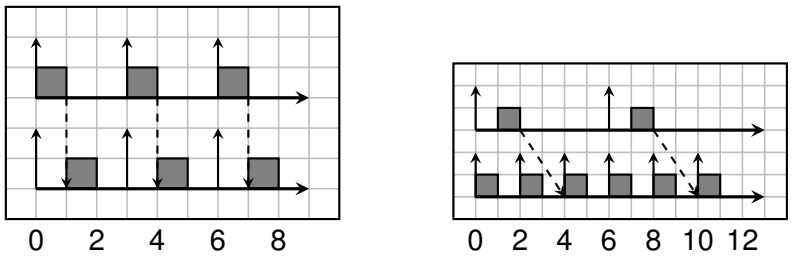
Prelude programs are often made up of several *computation chains*. Tasks within a chain all have the same period, while tasks of different chains may have different periods. The different computation chains eventually join, since they must collaborate to implement the complete system behaviour. Such junctions imply communications between tasks of different periods.

Precedence constraints that relate tasks with different periods are called *extended precedence constraints*. In that case, only a subset of the jobs of the related tasks are concerned by the constraint. We let $\tau_{i.k} \rightarrow \tau_{j.k'}$ denote a precedence constraint from $\tau_{i.k}$ to $\tau_{j.k'}$. We define the predecessors of a job $\tau_{i.k}$ as $preds(\tau_{i.k}) = \{\tau_{j.k'} | \tau_{j.k'} \rightarrow \tau_{i.k}\}$ and its successors as $succs(\tau_{i.k}) = \{\tau_{j.k'} | \tau_{i.k} \rightarrow \tau_{j.k'}\}$. We use a *precedence matrix* $M_{i,j} \in \mathcal{M}$ (with $0 \leq i < |\mathcal{S}|$, $0 \leq j < |\mathcal{S}|$), to specify the pairs (p, q) such that $\tau_{i.p} \rightarrow \tau_{j.q}$. Precedence matrices represent sets of precedence constraints that follow patterns repeated periodically. Let $\mathbb{N}_{<n}$ denote the set of natural integers strictly smaller than n. Let $lcm(n, n')$ denote the least common multiple of n and n'.

Definition 2.2.1. Let τ_i and τ_j be two tasks. Let $H = lcm(T_i, T_j)$. A precedence matrix M associated to tasks τ_i, τ_j, is such that $M \subseteq \mathbb{N}_{<H/T_i} \times \mathbb{N}_{<H/T_j}$, where for all $(p, q), (p', q') \in M^2$, $p = p' \Rightarrow q = q'$ and $p' > p \Rightarrow q' \geq q$. Then, $\tau_i \xrightarrow{M_{i,j}} \tau_j$ denotes a precedence relation defined as follows:

$$\forall (p, q) \in M^\omega, \tau_{i.p} \rightarrow \tau_{j.q}$$
$$\text{with } M^\omega \equiv \{(n, n') | \exists k \in \mathbb{N}, (m, m') \in M, (n, n') = (m, m') + (k\frac{H}{T_i}, k\frac{H}{T_j})$$

This definition is illustrated in Figure 2.9. Intuitively, M lists all job precedence constraints over one hyperperiod of the related tasks. The pattern is then repeated indefinitely.



(a) $M = \{(0, 0)\}$ (b) $M = \{(0, 2)\}$

(c) $M = \{(0, 0)\}$ (d) $M = \{(0, 0), (2, 1)\}$

Figure 2.9.: Some communication patterns ($\tau_i \xrightarrow{M} \tau_j$)

2.2.3. Scheduling

Real-time scheduling consists in finding an order for the execution of a set of tasks, such that the set of real-time constraints of the tasks is respected. It involves two related parts. First, defining a *scheduling policy*, that is to say an algorithm whose purpose is to choose which task to execute at each step of the execution. Second, given a scheduling policy, performing a *schedulability analysis*, prior to execution, to ensure that the task set will respect all its constraints when scheduled with that policy.

In the following, we focus on **preemptive**, **on-line**, **priority-based** scheduling policies. A scheduling policy is *preemptive* if it allows interrupting a job during its execution and resuming it later. With *on-line scheduling*, the scheduler computes the schedule as execution progresses, based on the chosen scheduling policy. Most on-line scheduling policies are *priority-based*, meaning that they only define how to assign priorities to tasks and that the scheduler then always chooses to execute the highest priority task ready for execution. With a *fixed-task priority* scheduling policy, the priority of a task remains unchanged during the whole system execution. With a *fixed-job priority* scheduling policy, the priority can differ between jobs of the same task, but remains unchanged for a given job. Given a priority assignment Φ, we define two functions $s_{\mathcal{S}\Phi}, e_{\mathcal{S}\Phi} : \mathcal{J} \to \mathbb{N}$, where $s_{\mathcal{S}\Phi}(\tau_{i.k})$ is the start time and $e_{\mathcal{S}\Phi}(\tau_{i.k})$ is the completion time of $\tau_{i.k}$ in the schedule produced by this assignment. In the sequel, $\mathcal{S}$ and Φ are omitted when clear from context. We say that a schedule obtained for a dependent task set under a given priority assignment is *feasible* if it respects all the temporal constraints of the task set and all its job precedence constraints. More formally:

Definition 2.2.2. Let $\mathcal{S} = (\{\tau_i\}_{0 \leq i < n}, \to)$ be a dependent task set and Φ be a priority assignment. Let $\sigma_{\mathcal{S}\Phi}$ be the schedule of $\mathcal{S}$ under Φ. $\sigma_{\mathcal{S}\Phi}$ is *feasible* if and only if:

$$\begin{cases} \forall \tau_{i.k},\ e(\tau_{i.k}) \leq d_{i.k} \wedge s(\tau_{i.k}) \geq o_{i.k} \\ \forall \tau_{i.k} \to \tau_{j.k'},\ e(\tau_{i.k}) \leq s(\tau_{j.k'}) \end{cases}$$

A task set is *schedulable* by a given scheduling policy if and only if the schedule produced by that policy is feasible. A scheduling policy $\mathcal{P}$ is *optimal* within a certain class of policies (e.g. the class of fixed-task policies) if the following holds: if a task set is schedulable by some policy of this class, then it is schedulable by $\mathcal{P}$.

In monocore[1], Liu and Layland [LL73] proposed the *rate-monotonic* (RM) fixed-task priority policy, where tasks with a shorter period are affected a higher priority, and the *earliest-deadline-first* (EDF) fixed-job priority policy, where jobs with a shorter absolute deadline are affected a higher priority. RM is optimal within the class of fixed-task priority policies for periodic task sets with $T_i = D_i$ and $O_i = 0$. It can be extended to the *deadline-monotonic* policy (DM) [LW82], to schedule optimally a set of tasks with $D_i \leq T_i$ and $O_i = 0$. For the case where $O_i \geq 0$, an optimal algorithm was defined by Audsley in [Aud91]. EDF is optimal within the class of fixed-job priority policies, for jobs with arbitrary offsets and deadlines.

A *schedulability test* determines whether a task set is schedulable with a given scheduling policy. A schedulability test is called *sufficient* if all task sets considered schedulable by the test are indeed schedulable. A schedulability test is called *necessary* if all task sets considered unschedulable by the test are indeed unschedulable. Schedulability tests that are both sufficient and necessary are referred to as *exact*.

[1] To avoid overburdening this introduction, multicore scheduling will be discussed later in the document.

2.3. Low-level timing analysis of a single task

High-level timing analyses require as input an upper bound to the execution time of each task. Deriving such upper bounds is an undecidable problem in the general case, as it can be reduced to the halting problem. However, programming conventions for real-time systems require the number of iterations of loops and recursions to be explicitly bounded, thus making execution time upper-bounds computable. It remains nevertheless a difficult problem, because it requires to identify the worst case input software-hardware configuration, which leads to the Worst-Case Execution Time (WCET). The software-hardware state space is usually too large to be explored exhaustively, thus WCET analysis produces approximate results. Ideally, a WCET value should be *safe*, in the sense that it must be greater than all possible execution times of the task, and should also be *tight* to avoid hardware resource over-provisioning. The right trade-off between safety and tightness depends on the application domain; the critical real-time domain favors **safety**.

There exists several different approaches to WCET analysis, a survey of which can be found in [Wil+08]. They can be categorized as either *measurement-based*, typically unsafe but less pessimistic and complex, or *static*, typically safe but more pessimistic and complex. As my work targets critical real-time system, in the following I will focus on **static** WCET analysis.

Static analysis can be applied to either source code or binary code. Source code analysis is generally easier, thanks to the higher level of abstraction of the program. However, it requires to make assumptions about the (complex) compiler behaviour, thus machine code analysis is often favored for WCET analysis. In my work, I focus on **machine code** analysis.

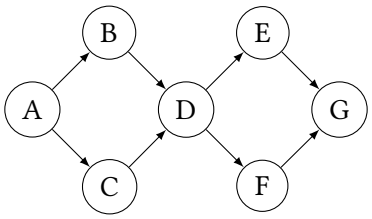
Static WCET analysis consists of four main steps. *Control-flow analysis* (CFA) studies the different possible execution paths of the program. *Hardware analysis* determines the execution time of a path on the considered hardware. *Value analysis* studies the values computed by the program, providing information useful both for CFA (e.g. loop bounds) and hardware analysis (e.g. instruction addresses needed for cache analysis). Finally, the *bound computation* puts all information of the previous steps together to estimate a safe over-approximation of the WCET.

2.3.1. Control-flow analysis

A *Control Flow Graph* (CFG) is the traditional model used to represent the possible execution paths of a machine code program. In a CFG, each node or *basic block* corresponds to a sequence of instructions with no jump or jump target, while edges represent jumps. Figure 2.10 details an example of Arm program, with the corresponding C code as comment for better readability, and the CFG representing it. CFG construction from machine code is a difficult problem, but it is out of the scope of the work presented in this manuscript.

```
1     @ ...                   @     /* A */
2     str r0, [fp, #-32]      @     /* A */
3     @ ...                   @     /* A */
4     ldr r3, [fp, #-32]      @     /* A */
5     cmp r3, #10             @     if (n <= 10) /* A */
6     bgt .L2                 @     {   /* still A */
7     @ ...                   @         /* C */
8     b   .L3                 @     }   /* C */
9  .L2:                       @     else {
10    @ ...                   @         /* B */
11 .L3:                       @     }
12    @ ...                   @     /* D */
13    ldr r3, [fp, #-32]      @     /* D */
14    cmp r3, #-1             @     if (n <= -1)  /* D */
15    bgt .L4                 @     {   /* still D */
16    @ ...                   @         /* F */
17    b .L5                   @     }   /* F */
18 .L4:                       @     else {
19    @ ...                   @         /* E */
20 .L5:                       @     }
21    @ ...                   @     /* G */
```

(a) Arm program



(b) CFG

Figure 2.10.: Control-flow representation

A CFG represents a superset of the set of paths that are actually feasible in the corresponding program. A path that is structurally feasible in the CFG may be infeasible if we take the semantics of the program into account. For instance, in Figure 2.10, path $A.B.D.F.G$ is structurally feasible in the CFG, but semantically infeasible in the program (n cannot be greater than 10 *and* smaller than -1).

Control-flow analysis (CFA) provides information on semantically infeasible execution paths, to be combined with the purely structural information provided by the CFG. The more information it can extract, the tighter the WCET bound gets. There exists many CFA approaches, for instance detection of mutually exclusive branch constraints [HW02], detection of infeasible paths across several loop iterations [RCM17], or analysis of loop bounds and their iteration structures [Wil+08].

2.3.2. Hardware analysis

In its simplest form, hardware analysis provides the execution time of each instruction of an instruction set on the target hardware. However, on modern hardware the execution time of an instruction can be subject to large variations caused by various optimization components: caches, pipeline, branch prediction, ... Hardware analysis consists of a set of sub-analyses whose objective is to determine the state of the different hardware components at each program point. The execution time of a program instruction is then derived based on the hardware state at that

point.

Hardware analysis is performed on an abstract conservative model of the actual hardware, which means that execution time predicted based on the model is never less than actual execution time on the concrete hardware. As different execution paths may lead to the same program point, the exact hardware state usually cannot be derived. Instead, weaker invariants on it are established, typically using abstract interpretation [CC77]. WCET analysis tools usually include at least hardware analyses for the instruction cache [Fer+99], and the pipeline (e.g. [RS09]).

2.3.3. Value analysis

To produce accurate information, CFA and hardware analysis both require information on the values computed by the program. As we analyse machine code, these values are stored in registers or in memory. Value analysis infers properties on the content of processor registers, and memory addresses accessed by the program. Abstract interpretation is a popular approach to value analysis as it provides a safe over-approximation of the possible values computed at each program point. It is most noticeably used to infer loop bounds (e.g. [GEL05]), and to infer instruction addresses (e.g. [The+03]), required for cache analysis.

2.3.4. WCET bound computation

The final step of WCET analysis consists in combining information provided by CFA, hardware analysis and value analysis, to compute an upper-bound to the WCET. There are mainly two approaches to WCET bound computation: *tree-based* (also called structure-based), and *implicit-path enumeration* (IPET).

A tree-based approach represents the program to analyse as a tree (akin to an abstract syntax tree), and computes the WCET bound by a recursive evaluation on the tree structure. Essentially:

- The WCET of a sequence of nodes is the sum of the WCET of the nodes;
- The WCET of an alternative between nodes is the maximum of the WCET of the nodes;
- The WCET of a loop multiplies the WCET of the loop body by the loop bound.

For instance, the WCET for the program of Figure 2.10 would be computed as: $t_A + \max(t_B, t_C) + t_D + \max(t_E, t_F) + t_G$ (where t_I denotes the WCET of basic block I).

In IPET, the information and constraints provided by CFA, hardware analysis and value analysis, are all combined into a single *Integer Linear Program* (ILP) and the WCET bound is obtained by solving this ILP. Essentially:

- A timing t_e is associated to each node of the CFG. It is an upper bound to the WCET of the corresponding basic block, inferred by hardware analysis.
- An execution count x_e is associated to each node and edge of the CFG. It represents the number of execution of the node or edge during a complete program execution;
- The goal function of the ILP is to maximize the term $\sum_{i \in CFG} x_i * t_i$, thus obtaining an upper-bound to the WCET.

Constraints related to the CFG structure are expressed as constraints on execution counts. For instance, for the program of Figure 2.10 the following ILP constraints would be deduced from the CFG: $x_A = x_G = 1, x_A = x_{A\to B} + x_{A\to C}, x_B = x_{A\to B}, \ldots$ Constraints obtained by CFA or value analysis, are also expressed as constraints on execution counts. For instance, for the previous program, an infeasible path analysis would add the constraint $x_A = x_B + x_F$.

IPET has become a popular WCET analysis technique, thanks to its generality; indeed, results of new analyses can easily be integrated as ILP constraints. Its main drawback is its high complexity (potentially exponential in the program size). On the contrary, tree-based approaches have low complexity (a low-degree polynomial in the size of the tree), but struggle to integrate constraints provided by CFA. The popular WCET analysis tools Heptane [HRP17], OTAWA [Bal+10] and AiT [Abs] are currently based on IPET.

Part II.

Contributions

3. Programming real-time systems with Prelude

This chapter summarizes my contributions on Prelude that are posterior to my Ph.D. thesis. First, I summarize extensions of the compiler back-end that target multi-core hardware platforms (Section 3.1). Then, I present two extensions to the language definition and to the compiler front-end: the introduction of a new operator to support incomplete specifications (Section 3.2), and support for mode-automata (Section 3.3).

3.1. Implementation on multi-core with distributed memory

Over the last decade, the code generation of Prelude has been extended to target several types of hardware platforms and Operating Systems. The available code generation options are summarized in Table 3.1. *Centralized memory* corresponds to a hardware architecture with a large shared memory, and a smaller cache memory for each core. *Distributed memory* corresponds to a hardware architecture with a large shared memory, and a smaller private locally addressable memory for each core (typically a scratchpad memory).

MarteOS [RH01] on unicore was the first code generation target, implemented during my Ph.D. thesis. The code generation was then adapted so as to generate OS-independent code. Porting Prelude to a new target OS now only requires to implement a small set of OS-specific functions (task creation, and communication primitives).

SchedMCore [Cor+11], was the first multi-core target. SchedMCore runs on top of an existing Operating System, and allows to run a set of tasks written in C using various real-time multicore scheduling policies (somewhere between a pure simulator and a true hard real- time execution environment).

The code generation was then adapted to enable execution on standard Linux, relying on the SCHED_DEADLINE scheduling policy and on the ptask API[1] to enforce real-time constraints. Due to unbounded latencies in the Linux kernel, this is not a truly hard real-time port. Nevertheless, it enables easy prototyping and testing of real-time applications.

A distributed memory target was then implemented. This required to significantly extend the code generation, so as to explicitly handle copies between core local memories and the global shared memory. The first port [Pag+18b], referred to as *Sequencer* in Table 3.1, targeted an architecture with no real OS. Instead, the task schedule was generated offline and tasks were executed by a simple sequencer. Finally, we ported code generation to *Erika*, a true real-time Operating System[2]. In the remainder of this section, I present our work on code generation for distributed memory architectures.

[1] Ptask is a Periodic Real-Time Task interface to pthreads: https://github.com/glipari/ptask
[2] http://erika.tuxfamily.org/drupal/

OS	unicore	multicore	centralized mem.	distributed mem.
MarteOS	✓		✓	
SchedMCore	✓	✓	✓	
Linux	✓	✓	✓	
Sequencer	✓	✓		✓
Erika	✓	✓	✓	✓

Table 3.1.: Code generation for Prelude on different platforms

3.1.1. Motivation

Implementing real-time tasks on a multi-core platform is hard, mainly because cores share access to a central memory. This leads to contentions, which cause significant execution delays that are hard to predict, because they require to finely analyse task codes, task interferences and the contention resolution mechanisms [PC10].

In order to simplify the analysis of task interferences, the PRedictable Execution Model (PREM) [Pel+11] advocates to decouple communication phases from computation phases. For instance, the AER task model [Dur+14], a declination of the PREM model, splits each task of the system into three phases. The *Acquisition* phase loads task data and instructions from the main memory into the core local memory. Then, the *Execution* phase performs the task computations using only local memory. Finally, the *Restitution* phase copies the results of the E-phase back into the main memory, for use by other tasks. This simplifies timing analysis because: 1) communication phases are clearly identified, so the system scheduler can schedule communications [AP14a; Mai+17] and avoid contentions; 2) worst-case execution time analysis (WCET) of computation phases does not need to take bus contentions into account [Pel+11].

Manually implementing a PREM-compliant program is tedious, unintuitive and error-prone. Instead, we propose an extension of the Prelude compiler that automatically generates PREM-compliant code. The synchronous semantics is close to the PREM model, making the translation into PREM natural. We target a multi-core platform with distributed memory: one shared main memory plus one private scratchpad memory (SPM) for each core. According to a predefined distribution of tasks onto cores, the compiler generates a separate C code for each core. The main advantage of this approach is to simplify the development process, by automating the translation from the high-level specification in Prelude to the low-level implementation in C. In particular, concerns related to task communications become the responsibility of the compiler.

3.1.2. Model

First, we define the considered hardware and software model.

3.1.2.1. Distributed memory

We consider a multi-core hardware architecture with distributed memory. Each core ρ_i has access to a global shared memory denoted $\mathcal{M}_G$ and to a private scratchpad memory denoted $\mathcal{M}_i$. We assume a static allocation of code and data to SPMs. Compared to a cache-based architecture, in our case distributed memory is apparent in the program code (local memory is explicitly addressable). Thus, memory transfers between private and global memories are handled by the Prelude compiler. This implies more predictable memory accesses without

Phase	Dependency
E_A R_A	$\tau_A \rightarrow \tau_C$
E_B R_B	$\tau_B \rightarrow \tau_C$
A_C	$\tau_B \rightarrow \tau_C, \tau_A \rightarrow \tau_C$
E_C	$\tau_C \rightarrow \tau_D$
E_D	$\tau_C \rightarrow \tau_D$

Table 3.2.: Phases and related data-dependencies

overburdening the programmer.

3.1.2.2. Multi-phase tasks

The translation of the Prelude program into a set of real-time tasks remains unchanged compared to previous works. Here, we consider a task set as the starting point for the code generation (for readers unfamiliar with the definition of a real-time task set, see Section 2.2 for more details). Following the AER model of [Dur+14], each task τ_i is divided into three phases. In the *Acquisition* phase (A_i), data is copied from $\mathcal{M}_G$ into $\mathcal{M}_i$. The *Execution* phase (E_i) then executes using only $\mathcal{M}_i$. Finally, in the *Restitution* phase (R_i), the results of the Execution phase are copied back from $\mathcal{M}_i$ into $\mathcal{M}_G$. In our implementation, not all tasks have A and E and R phases. Tasks without incoming data-dependencies, have no A-phase. Tasks without outgoing data-dependencies have no R-phase. Similarly, A/R phases are removed when all predecessors/successors are located on the same processor.

Figure 3.1 shows a simple example, that will be used as an illustration in further sections. The program consists of four tasks/nodes ($\tau_A, \tau_B, \tau_C, \tau_D$) distributed on two processors (ρ_0, ρ_1). The program is multi-periodic (periods 5, 6, 10). Phases and the data-dependencies they implement are depicted in Table 3.2. For instance, τ_C copies both its inputs during A_C. Since τ_C and τ_D are colocated (i.e. located on the same core), their data-dependencies are directly handled by E_C and E_D. In this example, none of the tasks have three phases.

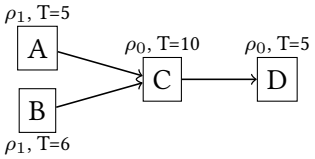


Figure 3.1.: Running example

3.1.3. Code generation

Figure 3.2 details the files involved in the production of an executable for a Prelude program. The Prelude program is compiled into one C file per core and one C file for the global memory $\mathcal{M}_G$. The code of each core contains one function per phase allocated to that core, and related communication and synchronisation code (see Figure 3.3 for instance). The $\mathcal{M}_G$ code contains

data shared for inter-core communication purposes. In addition, the C application contains some code that is not generated by Prelude: 1) for each task, a user-provided imported function, to be executed during the corresponding E-phase; 2) the OS specific code that integrates the generated files into the final application. The compilation of the C code produces one binary per core ρ_i, to be stored in $\mathcal{M}_i$, which contains the instructions and local data of ρ_i. Shared communication data is stored into $\mathcal{M}_G$.

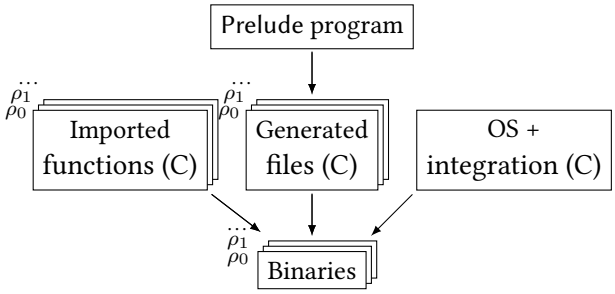


Figure 3.2.: Files involved in the production of the executable code

So as to illustrate the structure of the generated code, Figure 3.3 details the code generated for tasks τ_A, τ_C. For each input or output of each task, the compiler allocates a *working variable* in $\mathcal{M}_i$ that is only accessed by the phases of that task (variables suffixed by `_loc` or `_out`). It allocates a *communication buffer* for each $\tau_i \rightarrow \tau_j$ (variables suffixed by `_buff`). If τ_i and τ_j are located on the same core, the buffer resides in $\mathcal{M}_i$ (e.g. `C_D_buff` in $\mathcal{M}_0$), otherwise it resides in $\mathcal{M}_G$ (e.g. `A_C_buff`).

In the E-phase code, the call to the imported function only operates on working variables (e.g. Line 16). Before this call, we copy input data from communication buffers into working variables. After this call, we copy output data from working variables into communication buffers. For intra-core communications, these copies are directly performed by the E-phase (Line 17). For inter-core communications, they are performed by the A/R-phases (Lines 7 Column 1, and 8 Column 2). We use the OS-specific functions `read_val` and `wrive_val` to perform copies between $\mathcal{M}_i$ and $\mathcal{M}_G$.

```
1  // core 0
2  void C_A() {
3    wait_sem(sem_A_C);
4    if (must_wait_B_C())
5      wait_sem(sem_B_C);
6
7    a_loc = read_val(A_C_buff, A_C_idx);
8    b_loc = read_val(B_C_buff, B_C_idx);
9
10   A_C_idx += 1;
11   if (must_change_B_C())
12     B_C_idx += 1;
13   }
14
15 void C_E() {
16   c_out = C(a_loc, b_loc);
17   C_D_buff = c_out;
18   post_sem(sem_C_D);
19 }
```

```
// core 1
void A_E() {
  a_loc = A();
}

void A_R() {
  if (must_write_A_C())
    write_val(A_C_buff, a_loc);

  if (must_post_A_C())
    post_sem(sem_A_C);
}
```

Figure 3.3.: C code for τ_A and τ_C

Table 3.3.: Size of memories for the experiments

Memory (SPM architecture)	Size
Data SPM	ρ_0: 5kB, ρ_1: 4kB
Instruction SPM	ρ_0: 12kB, ρ_1: 8kB
Main	2kB
Memory (cache architecture)	**Size**
Data cache	2kB
Instruction cache	4kB
Main	29kB

We do not detail here how the Prelude compiler generates the code of the inter-task communication protocols depending on the rate transition operators involved. It remains as detailed in [For09]. To summarize, during that step the compiler determines for each $\tau_i \rightarrow \tau_j$:

- The size of the communication buffer `i_j_buff` (e.g. `C_D_buff` has size 2);
- A function `must_change_i_j`, which tells when to change the cell of `i_j_buff` jobs of τ_j read from (e.g. `must_change_C_D` always returns true);
- A function `must_write_i_j`, which tells for each job of τ_i whether it must write in `i_j_buff` or not (e.g. `must_write_A_C` alternates between true and false, meaning that only one out of two successive jobs of τ_A writes in the buffer);
- A function `must_wait_i_j`, which tells if τ_j must wait on the communication semaphore;
- A function `must_post_i_j`, which tells if τ_i must post on the communication semaphore.

3.1.4. Comparing memory architectures

The original code generation of Prelude, implemented during my Ph.D. thesis, targets an architecture with centralized memory. As a consequence, we can now use the Prelude compiler, with its new distributed code generation target, to compare the performance of an application on a hardware platform with centralized memory against the same platform with distributed memory instead.

In order to allow the comparison between different hardware architectures, we rely on an FPGA development board, a Cyclone III by Altera with two NIOS-II softcores, depicted in Figure 3.4. The data and instruction ports connect the cores to the Avalon Interconnect Fabric, a crossbar which serves as a hub to access shared resources of the board. Each core has access to a tightly-coupled memory for data and to another for instructions. These memories serve as scratchpad memories ($\mathcal{M}_i$). Processors share access to an on-chip shared RAM ($\mathcal{M}_G$). Finally, processors are also connected through the Avalon to an on-chip mutex (used to implement semaphores), on-board IOs and timers.

In addition to the scratchpad architecture we just detailed, we implement a cache-based architecture on the FPGA. It features a cache for each core, with access performances similar to the scratchpads. The FPGA has tight space limitations, its memory sizes are reported in Table 3.3. Space reserved for SPM in the scratchpad-based architecture is instead reserved for the main memory in the cache-based architecture.

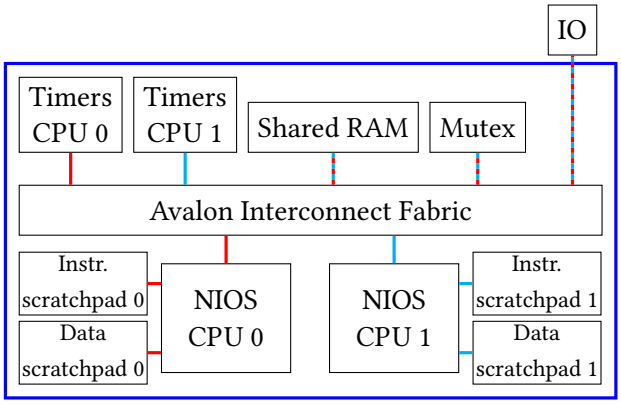


Figure 3.4.: The hardware design.

We perform experiments on the Rosace case study [Pag+14]. We compare the different hardware architectures through the response time of each task of Rosace. Figure 3.5 shows the speedup of PREM code on the SPM-based architecture with respect to non-PREM on the cache-based architecture (e.g. speedup of 2 means that SPM+PREM is twice as fast as cache+non-PREM). We provide results for different RAM clock speeds: either the same as the global clock (red), 4 times slower (green) or 8 times slower (blue, which corresponds to observed latencies on an external SRAM on similar boards). We provide mean results for 20 executions for each configuration (variance is very low).

The observed speedup is proportional to the RAM clock. When the shared RAM is the slowest, the average speedup is 6.29 with a standard deviation of 2.19. When the shared RAM has the same clock as the global clock, the SPM implementation barely outperforms the cache one. The average speedup is 1.09 with a standard deviation of 0.31.

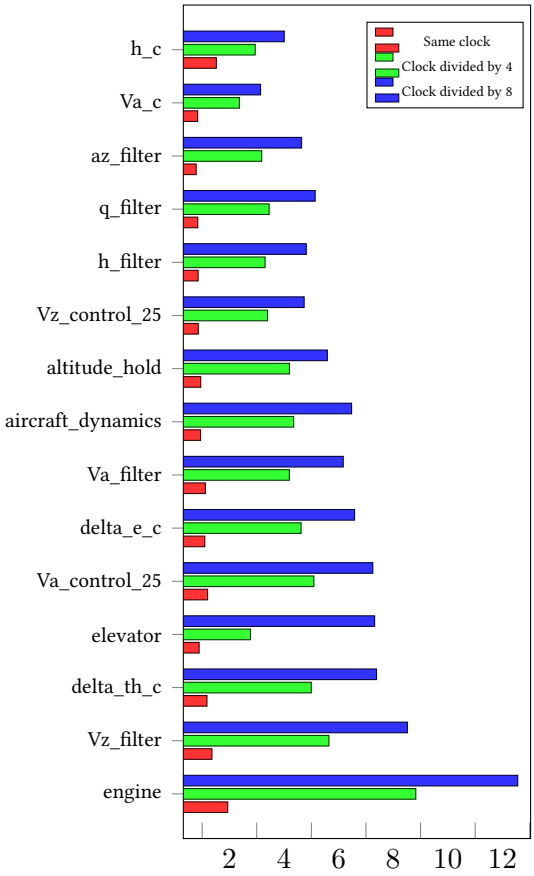


Figure 3.5.: Observed speedup (higher is better)

3.1.5. Related works

Before our publications The majority of works on PREM task models concerns schedulability analysis [Yao+12; AP14a; AP14b; WP14; AWP15; Mel+15; Bec+16; Yao+16; Mai+16; Mai+17; RDP17]. Schedulability analysis is out of the scope of our work, instead we focus on code generation.

Other works have focused on the development of PREM-compliant applications. OS-level support or hardware drivers for the execution of PREM tasks have been proposed in [WP13; Cap+17; Tab+16; Tab+17]. Automated production of PREM-compliant code has also been considered, by C-code refactoring in [MDC14; Mat+18], by compilation from C-code [SP17], also for a hardware target with GPU kernels [FBM18].

Our work is orthogonal to these approaches, in that we start from a high-abstraction synchronous language, which naturally fits with the hypotheses of the PREM model. Compilation of synchronous languages for distributed hardware platforms was studied in [ALGM96; GLS99; GNP06], but with a single execution thread per CPU. Compilation into multi-thread/multi-task code was proposed for control-flow synchronous languages in [YYR11; Yip+16] and for Scade in [Pag+18a].

After our publications The analysis and implementation of PREM-compliant applications remains an active topic. For instance, compilation from a high-level language into PREM-

compliant code has recently been studied for SCADE in [Sch+20], and for ForeC in [HGJ19].

3.1.6. Conclusion

The main highlights concerning this work are listed below:

- A first version of the multi-phase code generation, with off-line scheduling, was presented at the RTNS'18 conference [Pag+18b];
- The multi-phase code generation was then adapted for execution with an on-line scheduler and validated on a real hardware architecture, as presented at the RTCSA'19 conference [FF19];
- This work was a collaboration with a colleague from Onera (Claire Pagetti);
- This work was the main topic of the internship of Frédéric Fort (03/2018-08/2018), which I supervised.

3.2. Partial delays specification

Prelude ensures, through the language structure and the compiler static analyses, that the program semantics, and in particular the semantics of data-communications, are completely deterministic. While this improves the reliability of critical systems, this also requires the system developer to *design* a completely deterministic program, which can be difficult for large systems. In some cases, the developed system is such that its specification can support some degree of freedom without jeopardizing its reliability. Thus, requiring a completely deterministic specification can actually lead to overspecification, in the sense that the developer has to make arbitrary choices to fulfill the determinism requirement.

We proposed to extend Prelude to support incomplete specifications. The designer can specify that some communications can either be immediate or delayed. It is then up to the compiler to choose where to introduce delays in the program, in a way that breaks causality cycles and satisfies some specified latency requirements.

3.2.1. Motivating example

Let us consider the simplified mono-periodic Flight Control System depicted in the Figure 3.6. Its objective is to control the position, speed and attitude of the vehicle through its control surfaces. The right part of the figure depicts the control of the ailerons, while the left part depicts the control of the elevators. Vertices depict computation nodes, while edges depict data-communications between nodes. Plain edges stand for immediate communications, which induce a precedence constraint from the data-producer to the data-consumer. Dashed arrows stand for less constrained communications that do not induce precedence constraints.

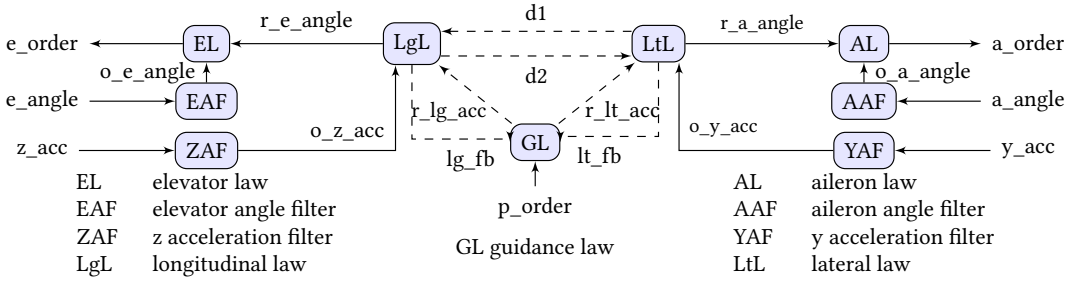


Figure 3.6.: A simplified flight control system

Variable `a_angle` corresponds to the current angle of the aileron and is acquired by node AAF (Aileron angle filter). This node consolidates the data and sends its output `o_a_angle` (the observed angle) to the function AL (Aileron law). AL controls the aileron and maintains the required angle `r_a_angle`. According to the observed angle and the required angle, it sends an order `a_order` that enables to reach safely the required angle. Thus, the command of the aileron surface is implemented by the computation path $L_1 =$ `a_angle` $\rightarrow$ `o_a_angle` $\rightarrow$ `a_order`. We use the term *functional chain* for such a computation path that starts with a sensor acquisition and ends with an actuator order. L_1 corresponds to the discretisation of a command law and communications must all be immediate for the computation to be correct. The elevator law behaves similarly.

The control laws of the ailerons and of the elevators communicate through the nodes `LtL` and `LgL`, to ensure that the orders sent to the different actuators (ailerons and elevators) are consistent. This consolidation step does not require strict synchronization between the two control laws, a function can compute using data produced by another node during the previous reaction instead of the current reaction (delayed communication). Still, the comparison and consolidation must be done on sufficiently "fresh" data, thus the number of delayed communications that the functional chain $L_2 =$ `z_acc` $\rightarrow$ `o_z_acc` $\rightarrow$ `d2` $\rightarrow$ `r_a_angle` $\rightarrow$ `a_order` can tolerate is upper-bounded by a maximum latency constraint.

The guidance law (`GL`), computes the acceleration to apply in order to reach the position `p_order` ordered by the pilot. This corresponds to the chain $L_3 =$ `p_order` $\rightarrow$ `r_lt_acc` $\rightarrow$ `r_a_angle` $\rightarrow$ `a_order`. Again, this law is loosely coupled with the other laws and communications between them can be delayed. Note also that inter-law communications form cycles, for instance `r_lg_acc` $\rightarrow$ `lg_fb`. Thus, at least one communication per cycle must be delayed to ensure that the program remains causal, otherwise we cannot find an execution order that satisfies all the precedence constraints induced by immediate communications.

To summarize, some communications can be implemented as either immediate or delayed communications. At the same time, the number of delayed communications in a functional chain is upper-bounded by maximum latency constraints, and lower-bounded by causality constraints. Our extension consists in allowing the programmer to specify that some communications can be either immediate or delayed, and ensuring by compilation that latency constraints and causality constraints are met.

3.2.2. Incomplete program specification

Prelude is extended with a new operator denoted `dc` (for "don't care") to specify communications that are allowed to be either immediate or delayed. We call such communications

dc-communications. We also enable the specification of maximum latency constraints. This work only supports the use of operator `dc` in monoperiodic systems, so latencies are expressed as a number of reactions. The language syntax is reduced to the following:

$$
\begin{array}{lcl}
vars & ::= & id \mid vars, id \\
e & ::= & cst \mid id \mid (e, e) \mid cst \texttt{ fby } e \mid id(e) \mid cst \texttt{ dc } e \\
eqs & ::= & vars = e; \mid eqs* \\
topDecl & ::= & \texttt{node } id(vars)(vars) \; [\, \texttt{var } vars; \,] \; \texttt{let } eqs \; \texttt{tel} \\
 & & \mid \texttt{imported node } id(vars) \; \texttt{returns } (vars) \; \texttt{wcet } n; \\
 & & \mid \texttt{req } (vars) < k \\
prog & ::= & decl*
\end{array}
$$

Example 3.2.1. We illustrate this syntax with the following program, corresponding to the example of Figure 3.6.

```
req (z_acc, o_z_acc, d2, dc2, r_a_angle, a_order) < 1; --L2
req (p_order, r_lt_acc, dc6, r_a_angle, a_order) < 4; --L3

node FCS_dc(a_angle, y_acc, e_angle, z_acc, p_order)
returns (a_order,  e_order)
  var o_e_angle, o_a_angle, r_a_angle, o_y_acc, r_e_angle, o_z_acc,
      lg_fb, r_lg_acc, d1, d2, r_lt_acc, lt_fb,  dc1, dc2, dc3, dc4,
      dc5, dc6;
let
  o_a_angle = AAF(a_angle);
  o_z_acc = ZAF(z_acc);
  o_z_acc = ZAF(z_acc);
  o_y_acc = YAF(y_acc);
  o_e_angle = EAF(e_angle);
  (lg_fb, d2, r_e_angle) = LgL(dc1, o_z_acc, dc4);
  (r_a_angle, lt_fb, d1) = LtL(dc2, o_y_acc, dc6);
  (r_lt_acc, r_lg_acc) = GL(dc3, dc5, p_order);
  a_order = AL(o_a_angle,r_a_angle);
  e_order = EL(r_e_angle, o_e_angle);

  dc1 = 0 dc d1; dc2 = 0 dc d2;
  dc3 = 1 dc lt_fb; dc4 = 0 dc r_lg_acc;
  dc5 = 1 dc lg_fb; dc6 = 0 dc r_lt_acc;
tel
```

The program first specifies maximum latency constraints on two functional chains ($L2$, $L3$). Then, the node `FCS_dc` assembles all the nodes of the different laws. For immediate communications (plain edges in Figure 3.6), the data-consumer directly takes as input the output of the data-producer. For dc-communications (dashed edges in Figure 3.6), we apply operator `dc` on the output of the data-producer before feeding it to the data-consumer.

In this work we extend the Prelude compiler so that it performs a source-to-source transformation that replaces each `dc` operator either by `fby` or by the identity (i.e. removes the `dc`

operator). First, we formalize the behaviour of operator `dc` using a non-deterministic semantics. Since the language is monoperiodic and contains no sampling operators, we do not consider clocks in this semantics and represent flows as sequences of values. Given a set of values $\mathcal{V}$ and a set of values E in $\mathcal{V}^*$, let $E.s$ denote the flow whose head is non-deterministically chosen among the values in E and whose tail is s. We abusively denote $v.s$ instead of $\{v\}.s$ when clear from context. We let $\prod_{i=1}^{n} E_i = E_1 \times \ldots \times E_n$ denote the Cartesian product of the sets E_i. We also define:

$$< E_1.s_1| \ldots |E_n.s_n >= \cup_{1 \leq i \leq n} E_i. < s_1| \ldots |s_n >$$

The semantics of flow expressions is as follows:

$$\begin{aligned} cst^{\#} &= cst.cst^{\#} \\ op^{\#}(E_1.s_1, \ldots, E_m.s_m) &= \cup_{t \in (\prod_{i=1}^{m} E_i)} \{op_f(t)\}.op^{\#}(s_1, \ldots, s_m) \\ \texttt{fby}^{\#}(cst, s) &= cst.s^{\#} \\ \texttt{dc}^{\#}(cst, s) &=< \texttt{fby}^{\#}(cst, s)|s^{\#} > \end{aligned}$$

Example 3.2.2. We detail the set of values that can be produced at each reaction of the following program (node `plus` simply sums its inputs).

```
imported node plus (i1,i2) returns (o);
node ex (i) returns (o)
var v1, v2;
let
  v1 = 0 dc i;
  v2 = 1 fby v1;
  o = plus(v1, v2);
tel
```

i	5	3	7	...
v_1	$\{0, 5\}$	$\{5, 3\}$	$\{3, 7\}$	
v_2	$\{1\}$	$\{0, 5\}$	$\{5, 3\}$	
o	$\{1, 6\}$	$\{3, 5, 8, 10\}$	$\{6, 8, 10, 12\}$	

3.2.3. Program concretisation

Program concretisation is the compilation step that translates a program with `dc` operators into a program without. This amounts to choosing one behaviour among the different possible ones specified by the program non-deterministic semantics, effectively producing a deterministic program.

Definition 3.2.1. A program is *concrete* if and only if it contains no `dc` operators. Otherwise it is *abstract*.

Let a denote an abstract program. Let $dc(a) = \{dc_1, \ldots, dc_n\}$ denote the set of `dc` operators in a ordered by appearance in the program text. Let $c = (dc_i \mapsto op)a$, denote the program resulting from the substitution of dc_i by op in a (with $op \in \{\texttt{fby}, id\}$). Let $(dc_1 \mapsto op_1, \ldots, dc_j \mapsto op_j)a$, denote the composition of program substitutions (with $dc_1, \ldots, dc_j$ in `dc` (a)).

Definition 3.2.2. Let c be a concrete program and a be an abstract program such that $\texttt{dc}\ (a) = \{dc_1, \ldots, dc_n\}$. We say that c is an *instance* of a iff there exists a set of substitutions $dc_1 \mapsto op_1, ..., dc_n \mapsto op_n$ such that:

$$p = (dc_1 \mapsto op_1, ..., dc_n \mapsto op_n)a$$

In the following, $a[b_1, \ldots, b_n]$ denotes the instance $p = (sub_1, \ldots, sub_n)a$ such that:

$$\begin{cases} sub_i = dc_i \mapsto id & \textit{if} \quad b_i = 0 \\ sub_i = dc_i \mapsto \texttt{fby} & \textit{otherwise} \end{cases}$$

Example 3.2.3. The abstract program of Example 3.2.2 has two instances $ex0 = ex[0]$, $ex1 = ex[1]$, shown below.

```
node ex0 (i) returns (o)
var v1, v2;
let
  v1 = i;
  v2 = 1 fby v1;
  o = plus(v1, v2);
tel

node ex1 (i) returns (o)
var v1, v2;
let
  v1 = 0 fby i;
  v2 = 1 fby v1;
  o = plus(v1, v2);
tel
```

Let $x \overset{0}{\leftarrow} x'$ denote that the variable x' immediately depends on the variable x. Let $x \overset{1}{\leftarrow} x'$ denote a delayed dependency. The latency of a functional chain corresponds to the number of delayed dependencies in the chain:

Definition 3.2.3. Let c be a *concrete* program. Let $C = (x_1, \ldots, x_n)$ be a functional chain of c. The latency of C in c is denoted $Lat_c(C)$ and computed inductively as follows:

$$Lat_c(x_1, \ldots, x_n) = \begin{cases} 1 + Lat_{sys}(x_1, \ldots, x_{n-1}) & \text{if } x_{n-1} \overset{1}{\leftarrow} x_n \\ Lat_{sys}(x_1, \ldots, x_{n-1}) & \text{if } x_{n-1} \overset{0}{\leftarrow} x_n \end{cases}$$
$$Lat_c(x) = 0$$

Example 3.2.4. In Example 3.2.3, considering $C = (i, v_1, v_2, o)$, we have $Lat_{ex0}(C) = 1$ and $Lat_{ex1}(C) = 2$.

Definition 3.2.4. A program is considered to be a *valid* instance of an abstract program iff:

- It respects the latency requirements specified in the abstract program;

- It is causal.

The problem of finding a valid instance for an abstract program amounts to a pseudo-Boolean problem, which we solve using SAT4J [LBP10]. Let a be an abstract program, and $c = a[b_1, \ldots, b_n]$ be an instance of a. For any functional chain $\mathcal{C}$ of a, let $B(\mathcal{C})$ denote the subset of $b_1, \ldots, b_n$ corresponding to the `dc` operators involved in $\mathcal{C}$. Then, finding a valid instance c for a amounts to finding a solution to the following pseudo-Boolean constraints:

- For any requirement $Lat_a(\mathcal{C}) < k$, we must have:

$$(\sum_{b_i \in B(\mathcal{C})} b_i) < k - Lat_a(\mathcal{C})$$

- For any cycle $\mathcal{C}$ in the data-dependencies of a (which can be enumerated using classic graph algorithms such as [Tar73]), we must have:

$$(\sum_{b_i \in B(\mathcal{C})} b_i) + Lat_a(\mathcal{C}) > 0$$

Example 3.2.5. The abstract program for the Flight Control System in Example 3.2.1 has four valid instances, `FCS_dc`[101111], `FCS_dc`[101011], `FCS_dc`[101010], `FCS_dc`[101101].

Experiments showed that the resolution scales to large programs. For an avionic application with 3994 imported nodes, 16186 variables, and 2000 latency requirements, it take less than 25s.

3.2.4. Related works

Before our publications In software architecture description languages such as AADL [FGH06], Marte [OMG07] or SysML [OMG10], communication patterns can be either immediate or delayed, but not undeterministic.

Undeterministic communication operators have been considered for Simulink[3], and also in other synchronous languages [MH96], but compilation is out-of-scope of these works. An important specificity of our work is that even though the initial specification is undeterministic, it is then translated into a deterministic one.

After our publications In [Ioo+20] authors use a `fby?` operator, that has the same semantics as the `dc` operator, and that is also later translated into a deterministic communication. Their objective is to balance the load of a synchronous program over a multi-periodic real-time schedule.

3.2.5. Conclusion

The main highlights concerning this work are listed below:

- This work was published at the APLAS'12 conference [Wys+12];
- This work is a collaboration with colleagues from Onera Toulouse (Frédéric Boniol, Claire Pagetti, Rémy Wyss).

[3] https://www.mathworks.com/help/simulink/slref/ratetransition.html

3.3. Multi-mode multi-periodic systems

In this work, we are interested in the programming of real-time applications that exhibit a multi-moded behaviour. In such a system, each mode produces a different behaviour, characterised by a different set of tasks to execute. For instance, the behaviour of an aircraft control can be decomposed into take-off, cruise, and landing modes. Synchronous State Machines (SSM) have been proposed in [CPP05] as a way to program multi-moded systems with Synchronous Languages. The purpose of this work is to extend Prelude with SSM.

3.3.1. Motivating example

As a motivating example, Figure 3.7 presents the implementation of the control software of an Unmanned Aerial Vehicle (UAV) based on [KS03; HHK03]. The system perceives its environment via a GPS and an Inertial Navigation System (INS). In addition, it receives via a wireless communication an enabling signal specifying in which mode it shall execute (`isEnabled`) and a destination point (`waypoint`). The system actuates via servo motors. The application has two modes. In the `Estimate` mode, the UAV preserves its previous course and measurements serve only to update the UAV position. In the `Actuate` mode, the UAV computes orders for the servo motors so as to reach the current waypoint.

Figure 3.7 shows the corresponding program written using SSM with Prelude. Recall that node execution rates are determined by the rate of their inputs. So, for instance `control` has period 10, while `servo_driver` has period 20, even though both nodes are executed within the same mode *Actuate*. The SSM semantics requires the two modes to compute the same set of non-local flows (`pos` and `servos`). The automaton switches from mode `Estimate` to mode `Actuate` when expression `isEnabled` is true, and from `Actuate` to `Estimate` when `not isEnabled` is true. In other words, *Mode Change Requests* (MCR) are emitted at rate $(10, 0)$.

Because flows and transitions are multi-periodic, not all flows should react to MCRs at the same rate. For instance, if at instant 10 the automaton is in state `Estimate` and `isEnabled` evaluates to true, most of the dataflows will immediately become computed according to equations of state `Actuate`. The only exception is `servos`, which will still be computed by its equation in state `Actuate`, until time 20 (non-included).

Figure 3.8 shows an excerpt of the automaton transpilation, which replaces automata constructs by the usual `when` and `merge` synchronous operators (see [CPP05] for details on the transpilation process). Flow `state_X` is true when the automaton is in state X. Flow `servos_X` defines the value of `servos` when the automaton is in state X. The output flow `servos` is obtained by merging the flows that define its values in the two automaton states (flow `pos` is computed similarly).

```
node main(GPS: GPSMessage rate(10,0); INS : INSMessage rate(10,0);
          isEnabled: bool rate(10,0); waypoint: real[4] rate(10,0))
returns(servos: ServoMessage)
var pos;
let
  automaton
  | Estimate ->
    unless isEnabled then Actuate;
    var GPS_f, INS_f, pos_f;
    GPS_f,INS_f,pos_f = h_f(GPS,INS, init_pos fby* pos);
    pos = filter(GPS_f, INS_f, pos_f);
    servos = init_servos fby* servos;
  | Actuate ->
    unless not isEnabled then Estimate;
    var GPS_c, INS_c, pos_c, waypoint_c, controls;
    GPS_c,INS_c,pos_c,waypoint_c =
      h_c(GPS, INS, init_pos fby* pos, waypoint);
    pos,controls = control(GPS_c,INS_c, pos_c,waypoint_c);
    servos = servo_driver(controls/^2);
  end
tel
```

Figure 3.7.: A multi-mode synchronous automaton

```
...
prev_state = Estimate fby state;
state = merge(prev_state, Estimate->s_Estimate, Actuate->s_Actuate);
s_Estimate =
  if isEnabled when Estimate(state) then Actuate
  else Estimate;

s_Actuate =
  if not (isEnabled when Actuate(state)) then Estimate
  else Actuate;

servos = merge(state, Estimate->servos_Estimate,
  Actuate->servos_Actuate);
servos_Estimate = (init_servos fby servos) when Estimate(state);
servos_Actuate = servo_driver(controls/^2);
...
```

Figure 3.8.: Automaton transpilation excerpt

3.3.2. Language extension

In the state machines of [CPP05], all flows within a state must have the same rate, i.e. only mono-rate states are allowed. Our objective is to transpose state machines to Prelude, and in doing so to extend them to support programs with multi-rate states. We rely on the same transpilation process as [CPP05] to translate automata constructs. Our work focuses on the extension of the semantics and clock calculus of operators `when` and `merge`.

We want to extend the definition of the clock operator ck on $C(c)$ to allow ck and c to have different periods. To this intent, we introduce the concept of *clock views*. The clock ck on $C(c, w)$ denotes the clock ck sub-sampled on condition c, such that it produces tags only if c, perceived according to view w, is equal to C. A view is a clock that specifies the rate at which the condition is observed. The semantics of the on operator with views is defined as follows.

$$(ck \text{ on } C(c,(n,p)))^{\#} = \{t \mid t \in ck^{\#}, \exists (C,t') \in c^{\#},\ t' \bmod n = p, t' \leq t < t' + n\}$$

Consider for instance the clock $(2,0)$ on $true(c,(6,0))$ depicted in Figure 3.9. The clock first produces tags 0, 2 and 4 because `c` is `true` at date 0. The fact that `c` is `false` at date 3 is ignored because the view $(6,0)$ considers `c` only at tags that are multiples of 6. The next tag produced by the clock is 18, because at dates 6 and 12 `c` is `false`, and values of `c` at instants 9 and 15 are ignored. The figure shows that changing the view, for instance clock $(2,0)$ on $true(c,(12,0))$, produces a different set of tags.

Formally, the semantics of Prelude flow operators remains exactly that presented for the core language in Figure 2.6, we just extend the semantics of the on clock operator. As shown in Figure 3.9, `x when C(c,w)` filters x on the clock $\widehat{x}$ on $C(c,w)$ (only the introduction of the clock view is new here), while the `merge` still merges flows with complementary clocks.

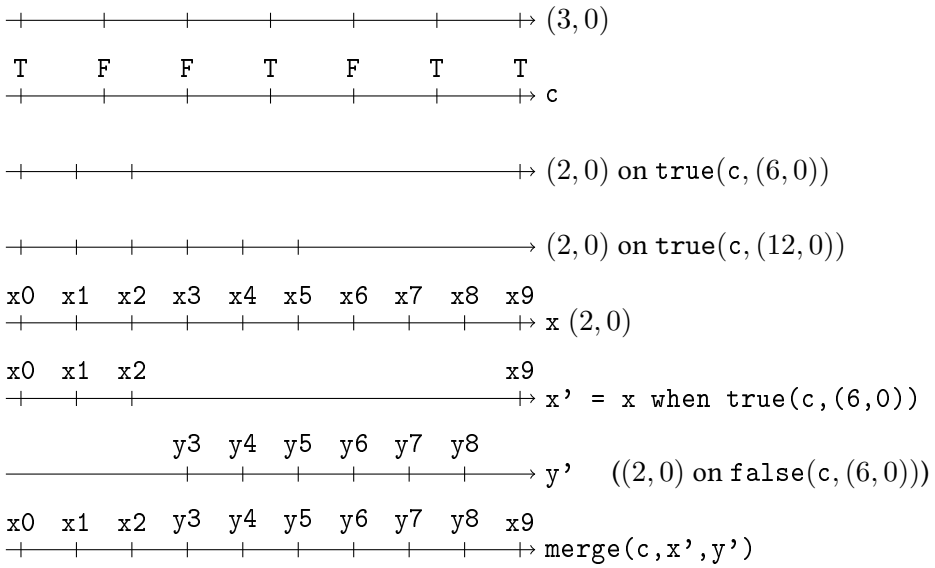


Figure 3.9.: Clocks and clock views

Clock views play a key role in the implementation of mode changes. A *Mode Change Request*

(MCR) is an event that triggers a mode change. In our case, MCRs are emitted by automata transitions. Let us illustrate the relation between the triggering of a transition, the automaton state, and clock views, on the example of Figure 3.8. The automaton state is defined by the variable `state`, which has clock $(10, 0)$. Outgoing transition conditions of state S (either `Estimate` or `Actuate`) have clock $(10, 0)$ on $S(\texttt{state}, (10, 0))$. Thus a MCRs may arrive each 10 time units. A MCR instantly updates the value of the variable `state`. However this state change is observed by tasks depending on their view. The *Mode Change Promptness* (MCP) is the amount of time a task requires to react to a MCR. In our work, views allow to reason about the MCP of tasks. For instance, task `control` in Figure 3.7 has view $(10, 0)$ while task `servo_driver` has view $(20, 0)$. Task `control` will respond to a change of `state` immediately, while `servo_driver` can in some cases respond with a delay of 10 time units. Thus, views provide an upper-bound to task *worst-case* Mode Change Promptness.

3.3.3. Clock calculus

In order to support our extended *on* clock operator, we define a clock calculus based on refinement typing [FP91]. In a refinement typing system, types may be ascribed with predicates. For instance `(/)` would have type $a{:}\texttt{int} \to b{:}\{\nu{:}\texttt{int} \mid \nu \neq 0\} \to \{\nu{:}\texttt{int} \mid \nu = a/b\}$, meaning "a function taking an argument a of type int and an argument b of type int whose value is not equal to 0, returning an int whose value is equal to a/b". In our clock calculus, we use linear integer arithmetic predicates and rely on the Z3 SMT solver [DMB08] to check the satisfiability of these predicates.

The refinement typing process is divided into two passes. In the first pass, only the structure of types is inferred. This pass is very similar to classic Hindley-Milner typing, except that types are annotated with *refinement holes*, i.e. refinement placeholders. In the second pass, refinement holes are replaced by actual type refinements. Inference rules use *bi-directional typing* [DK21]. *Synthesis judgements* $H \vdash e \Rightarrow t$ mean that in environment H, the type t is associated to expression e. *Checking judgements* $H \vdash e \Leftarrow t'$ mean that in environment H, the type t' is valid for expression e (even though e might be associated to a different type t). This is usually done by first producing a judgement $H \vdash e \Rightarrow t$ and then verifying the subtyping relation $t' <: t$. The subtyping relation is defined as follows.

Definition 3.3.1. $\{\nu{:}t \mid r_0\} <: \{\nu{:}t \mid r_1\} \Leftrightarrow \forall \nu : t.\ r_0 \implies r_1$

3.3.3.1. Clock system

The structure of the clock types of the Prelude clock calculus is shown below. A clock calculus environment H maps variables of the program to their clock types. In addition to the classic functional $(x{:}ck_r \to ck_e)$ and product clock types $(ck_e \times ck_e)$, we have refined clocks $(\{\nu{:}ck_c \mid r\})$. Note that refinements are not entirely within the logic of linear arithmetic due to divisibility constraints. We will show in Section 3.3.3.3 how we can reduce this non-linear problem to a linear one. The type `pck` is the type for all periodic clocks.

$$
\begin{array}{lcl}
H & ::= & \emptyset \mid H; x{:}ck_e \\
ck_e & ::= & x{:}ck_r \to ck_e \mid ck_e \times ck_e \mid ck_r \\
ck_r & ::= & \{\nu{:}ck_c \mid r\} \\
ck_c & ::= & \texttt{pck} \mid ck_b \text{ on } c(C, ck_r)
\end{array}
$$

$$
\begin{array}{lcl}
r & ::= & r \wedge r \mid p = a \mid a \text{ div } p \mid p \geq a \mid true \\
p & ::= & \pi(\nu) \mid \varphi(\nu) \\
a & ::= & k \mid \pi(x) \mid \varphi(x) \mid a + a \mid a - a \mid a * k \mid a/k
\end{array}
$$

$$x : \text{Identifier} \quad k : \text{Constant}$$

Refinement types tend to be quite verbose, thus in the following we use $\{\nu{:}ck_c \mid \langle n, p\rangle \wedge r\}$ as a shorthand for the clock type $\{\nu{:}\alpha \mid (\pi(\nu) = n) \wedge (\varphi(\nu) = p) \wedge r\}$.

3.3.3.2. Clock inference

During the first pass of the clock calculus, namely the *Structural Clock Calculus*, the structure of types is inferred, yielding types where refinements are unknown and represented by *refinement holes* (denoted $\star$ below).

Example 3.3.1. Consider the following two equations:

```
x = i when c;
y = x *^ 2;
```

Assume that the environment contains the following bindings:

$$H(\texttt{i}) = \{\nu{:}\texttt{pck} \mid \star_0\}$$
$$H(\texttt{c}) = \{\nu{:}\texttt{pck} \mid \star_1\}$$

The structural clock calculus produces the judgements below. Note how the inferred clocks only represent the global clock structures, only stating that both expressions involve the same clock condition. The relation between the periods and the phases of the different clocks are unspecified. They will only be introduced by the next pass of the clock calculus (refinement clock calculus).

$$H \vdash \texttt{i when c} \overset{S}{\Rightarrow} \{\nu{:}\texttt{pck on true}(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_2\}) \mid \star_3\}$$
$$H \vdash \texttt{x *^ 2} \overset{S}{\Rightarrow} \{\nu{:}\texttt{pck on true}(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\}) \mid \star_5\}$$

In the *Refinement Clock Calculus* pass, type refinements are inferred in place of refinement holes. Figure 3.10 details the corresponding inference rules. Rule Var states that the clock type of variables can be accessed from the environment. Rule Cst states that constants may have any clock type. The rule Appl infers a clock for the function and its argument. Then it verifies that the argument clock is a subtype of the expected clock. The clock of the function application is the output clock of the function where the placeholder name (x) has been substituted by the actual (a) within refinements.

$$\textsc{Var}\ \frac{}{H; x{:}\sigma \vdash x \Rightarrow \sigma} \qquad \textsc{Cst}\ \frac{}{H \vdash c \Rightarrow \sigma} \qquad \textsc{Appl}\ \frac{H \vdash f \Rightarrow x{:}ck_c \to ck_e \qquad H \vdash a \Leftarrow ck_c}{H \vdash f(a) \Rightarrow ck_e[x := a]}$$

Figure 3.10.: Inference rules of the refinement clock calculus

The initial environment of the refinement clock calculus provides the (functional) clock type of each operator of the language. Thus, it details the constraints, expressed in terms of clock refinements, relating the clocks of its inputs and outputs. Most notably, it defines the constraints relating their periods and offsets, and also the constraints on their views. For instance, the type for the `when` operator is detailed below. It takes an argument e of any clock and an argument c with offset equal to e but arbitrary period because n is free. It returns a clock with period and offset equal to e but which is only present when c perceived along view w equals C. View w has an offset equal to the offset of e and a period that is divisible by the periods of e and c. Note that there is no unique solution for the period of the view. The clock calculus will choose this period during the *view closing phase*.

$$\begin{aligned} \texttt{when}: \quad & \forall \alpha.e{:}\{\nu{:}\alpha \mid true\} \to c{:}\{\nu{:}\alpha \mid \langle n, \varphi(e)\rangle\} \to \{\nu{:}\alpha \text{ on } C(c,w) \mid \langle \pi(e), \varphi(e)\rangle\} \\ & \text{where } w = \{\nu{:}\texttt{pck} \mid \langle n', \varphi(e)\rangle \wedge \pi(e) \text{ div } \pi(\nu) \wedge \pi(c) \text{ div } \pi(\nu)\} \end{aligned}$$

Example 3.3.2. Consider the equations of Example 3.3.1. The type structures have already been inferred, now we need to add the refinements. The clock environment now contains information on clock periods and offsets (obtained from the input declarations in the program):

$$\begin{aligned} H(\texttt{i}) &= \{\nu{:}\texttt{pck} \mid \langle 20, 0\rangle\} \\ H(\texttt{c}) &= \{\nu{:}\texttt{pck} \mid \langle 10, 0\rangle\} \end{aligned}$$

We obtain the judgement below for the first equation. The expression has the same period and phase as i. However, there is no unique solution for the period n of the view w, we only know that it must be divisible by both $\pi(\texttt{i})$ and $\pi(\texttt{c})$. The period will be obtained in the final step of the clock calculus (view closing).

$$\begin{aligned} H \vdash \texttt{i when c} &\overset{S}{\Rightarrow} \{\nu{:}\texttt{pck on true}(\texttt{c}, w) \mid \langle \pi(\texttt{i}), \varphi(\texttt{i})\rangle\} \\ w &= \{\nu{:}\texttt{pck} \mid \langle n, \varphi(\texttt{i})\rangle \wedge \pi(\texttt{i}) \text{ div } \pi(\nu) \wedge \pi(\texttt{c}) \text{ div } \pi(\nu)\} \end{aligned}$$

We obtain the judgement below for the second equation. Note how operator $*\hat{}$ modifies the period of the output ($\pi(\texttt{x})/2$), but the period of the view remains unchanged. Intuitively, an expression cannot view a clock condition faster than its sub-expressions.

$$\begin{aligned} H \vdash \texttt{x}\ *\hat{}\ 2 &\Rightarrow \{\nu{:}\texttt{pck on true}(\texttt{c}, w') \mid \langle \pi(\texttt{x})/2, \varphi(\texttt{x})\rangle\} \\ w' &= \{\nu{:}\texttt{pck} \mid \langle \pi(w), \varphi(w)\rangle\} \end{aligned}$$

3.3.3.3. View closing

Constraints of the form $\pi(\mathtt{i})$ div $\pi(\nu)$ are nonlinear. However, these constraints only appear within refinements of views and can be simplified to linear ones as follows. First, as an additional constraint to the clock calculus, we impose that all user-defined node inputs must have rate annotations, and that the clocks of all expressions and variables must derive from the clocks of the node inputs (except for constants, which must have explicit clock annotations). Then, it can be verified that the structure of the inference rules of the refinement clock calculus along with the previously mentioned assumptions guarantee that the dependencies between view refinements form a forest where each root of the forest is a view with decidable constraints. Therefore, we can close views (i.e. compute their periods and offsets) by a breadth-first traversal of the forest. At each step, we propagate previously computed periods and offsets, removing nonlinearity from the unclosed views. The solver is then responsible for finding the lowest possible period and offset.

Example 3.3.3. Example 3.3.2 introduced the following views:

$$w = \{\nu{:}\mathtt{pck} \mid \langle n, \varphi(\mathtt{i})\rangle \wedge \pi(\mathtt{i}) \text{ div } \pi(\nu) \wedge \pi(\mathtt{c}) \text{ div } \pi(\nu)\}$$
$$w' = \{\nu{:}\mathtt{pck} \mid \langle \pi(w), \varphi(w)\rangle\}$$

Propagating constants from the typing environment yields the following clocks, with decidable constraints. Clock solutions are provided on the right:

$$w = \{\nu{:}\mathtt{pck} \mid \langle n, 0\rangle \wedge 20 \text{ div } \pi(\nu) \wedge 10 \text{ div } \pi(\nu)\} \Rightarrow \{\nu{:}\mathtt{pck} \mid \langle 20, 0\rangle\}$$
$$w' = \{\nu{:}\mathtt{pck} \mid \langle \pi(w), \varphi(w)\rangle\} \Rightarrow \{\nu{:}\mathtt{pck} \mid \langle 20, 0\rangle\}$$

3.3.4. Evaluation

The implementation of the new clock calculus in the Prelude compiler represents around 1500 new lines of OCaml code. Compared to the previous clock calculus (as defined in [For+08]), the new clock calculus incurs a noticeable but still very reasonable overhead in compilation time. For instance, the compilation time of the ROSACE case study [Pag+14] increases from $10ms$ to $50ms$ (including the constraints resolution time of Z3).

3.3.5. Related works

Mode change protocols have been studied extensively by the real-time scheduling community (see [RC04] for a survey). However, these works focus on the timing analysis of the system, and do not consider the semantics of the corresponding program.

As mentioned previously, synchronous state machines [CPP05; Tal+06] lack the ability to specify real-time constraints. Many other languages provide constructs to specify multi-mode applications. In the real-time domain, this includes AADL [Ber+08], Giotto [HHK03], and Statecharts [Har87]. However, these languages opt for one specific mode change protocol, while our language allows to specify which mode change protocol to use in a program.

Finally, clock calculus based on refinement typing was studied for Signal in [TJS15].

3.3.6. Conclusion

The main highlights concerning this work are listed below:

- This work has been published at the SAC'22 conference [FF22];
- The extended clock calculus was implemented in the Prelude compiler;
- This work is pivotal in the ANR PRCE Corteva project[4] (2018-2022, leader CRIStAL);
- This work is the main topic of the Ph.D. thesis of Frédéric Fort (10/2018-03/2022). The thesis is advised by Giuseppe Lipari, tutored by myself, and funded by the ANR Corteva project;
- Frédéric Fort currently has a post-doctoral position at IRT Saint-Exupery in Sophia Antipolis.

[4] https://corteva.cristal.univ-lille.fr/

4. High-level timing analysis

This chapter summarizes my contributions to the timing analysis of multi-rate dependent tasks. First, it summarizes my contributions on real-time scheduling (Section 4.1). After that, I present my work on two other kinds of high-level timing analyses. First, the analysis of end-to-end constraints is presented in Section 4.2. Second, Section 4.3 presents my work on task clustering, that is to say techniques to reduce the number of tasks used to implement a system.

4.1. Real-time scheduling

When we started working on Prelude, the integration of the separate tasks into a complete system often required to manually sequence the task set offline. The advent of reliable real-time operating systems has enabled to replace the manual task sequencing by a multi-task implementation, where tasks are scheduled concurrently online by the operating system. However, common real-time scheduling policies, namely RM, DM and EDF, do not support dependent tasks as is. Therefore, the determinism of task communications was usually ensured manually by the programmers. For instance, the dependent tasks can first be ordered manually, by sequencing them inside another larger grain task, then the operating system schedules this partially "pre-scheduled" task set. This is unfortunately tedious and time consuming. The purpose of our work was to investigate scheduling policies that directly support dependent tasks.

4.1.1. Scheduling tasks with simple precedence constraints

In [CSB90], Chetto et al. proposed a technique to schedule dependent task sets with simple precedence constraints on a uniprocessor hardware platform. They propose to *encode* precedence constraints in the task real-time attributes. The encoding is defined as follows:

$$O_i^* = \max(O_i, \max_{\tau_j \in preds(\tau_i)}(O_j^*)) \tag{4.1}$$

$$D_i^* = \min(D_i, \min_{\tau_j \in succs(\tau_i)}(D_j^* - C_j)) \tag{4.2}$$

Intuitively, a task must finish early enough for its successors to have sufficient time to complete before their own deadline (Equation 4.2 defining D_i^*). In addition, we must ensure that a task is not released before its predecessors (Equation 4.1 defining O_i^*). Authors proved that, for fixed-job policies, the encoded task set is equivalent to the original task set in terms of schedulability:

Theorem 4.1.1 (From [CSB90]). *Let $\mathcal{S} = \{\tau_i(O_i, C_i, D_i, T_i)\}$ be a dependent task set with simple precedence constraints. Let $\mathcal{S}^* = \{\tau_i'(O_i^*, C_i, D_i^*, T_i)\}$ be a set of independent tasks such that O_i^* and D_i^* are given by Equations 4.1, 4.2:*

$\mathcal{S}$ is schedulable with a fixed-job priority scheduling policy if and only if $\mathcal{S}^*$ is.

As a corollary, since EDF is optimal in the class of fixed-job priority scheduling policies, $\mathcal{S}$ is schedulable if and only if $\mathcal{S}^*$ is schedulable with EDF. The encoding of the complete task set proceeds by iterating the encoding process over a topological traversal of the precedence graph.

In [For+10], we extended this result to fixed-task scheduling policies. First, we proved that the encoding-based method also works for tasks without offsets, when scheduled with DM.

Theorem 4.1.2. *Let $\mathcal{S} = \{\tau_i(0, C_i, D_i, T_i)\}$ be a dependent task set with simple precedence constraints. Let $\mathcal{S}^* = \{\tau_i'(0, C_i, D_i^*, T_i)\}$ be a set of independent tasks such that D_i^* is given by Equation 4.2:*

$\mathcal{S}$ is schedulable with a fixed-task priority scheduling policy if and only if $\mathcal{S}^*$ is schedulable with DM.

For tasks with offsets, DM is not an optimal scheduling policy (even without precedence constraints). Instead, we adapt Audsley's scheduling policy [Aud91] to account for precedence constraints. This consists of two parts: 1) adjusting offsets according to Equation 4.1; 2) adapting Audsley's priority assignment so that a task always gets a higher priority than its successors. Also, since we manipulate deadlines relative to release times, we have to modify deadlines as follows:

$$D_i^* = D_i + O_i - O_i^* \tag{4.3}$$

The corresponding policy is detailed in Algorithm 1 (priority 1 is the highest priority). The only differences compared to the original algorithm are lines 1 (adjusting offsets), and 5 (checking task dependencies). Priorities are assigned starting from the lowest priority and up to the highest priority. At each step, the algorithm tries to find a task that is feasible if we assign it this priority. If no tasks can be assigned that priority, then the task set is unschedulable. The test at line 5 ensures that a task has higher priority than its successors.

We established the optimality of this policy:

Theorem 4.1.3. *Let $\mathcal{S} = \{\tau_i(O_i, C_i, D_i, T_i)\}$ be a dependent task set with simple precedence constraints. Then:*

$\mathcal{S}$ is schedulable with a fixed-task priority scheduling policy if and only if it is schedulable with the policy of Algorithm 1.

4.1.2. Scheduling tasks with extended precedence constraints

To schedule tasks with extended precedence constraints, we also rely on precedence encoding. However, the encoding is more complicated because we need to consider in detail which jobs of the tasks are related by precedence constraints.

4.1.2.1. Fixed-task priority

Let us first consider fixed-task scheduling policies. Let $\tau_i \xrightarrow{M_{i,j}} \tau_j$ be an extended precedence constraint. The fixed-task priority nature of the scheduling policy means that, even though some jobs of τ_i and τ_j may not be subject to precedence constraints, we still have to assign a higher priority to all jobs of τ_i, with respect to jobs of τ_j. Thus, we reuse the scheduling policy defined previously for fixed-task priority scheduling with simple precedence constraints (the

Algorithm 1 Scheduling policy for a task set $\mathcal{S}$ with simple precedence constraints and arbitrary offsets

```
1: S* ← adjust(S)
2: for lvl = |S| to 1 do
3:     assigned ← false
4:     for τ_i ∈ S* do
5:         if ∀τ_j ∈ succs(τ_i), Φ(j) > lvl then
6:             if respects_deadline(τ_i, lvl) then
7:                 Φ(i) ← lvl
8:                 S* ← S*\{τ_i}; assigned ← true
9:                 break the current loop
10:            end if
11:        end if
12:    end for
13:    if assigned=false then the system is not feasible
14:    end if
15: end for
```

extension of Audsley's policy detailed in Algorithm 1). The only modification concerns the adjustment of release dates:

$$O_i^* = O_i + max(0, \max_{M_{i,j}}(\max_{(n,n') \in M_{i,j}} ((O_j^* + nT_j) - (O_i + n'T_i)))) \tag{4.4}$$

Then, as a straightforward generalization of Theorem 4.1.3, we have:

Theorem 4.1.4. *Let $\mathcal{S} = \{\tau_i(T_i, C_i, O_i, D_i)\}$ be a dependent task set, with extended precedence constraints defined by a set of precedence matrices $M = \{M_{i_1,j_1}, \ldots, M_{i_l,j_l}\}$. Let $\mathcal{S}^* = \{\tau_i'(T_i, C_i, O_i^*, D_i^*)\}$ be a set of independent tasks such that O_i^* and D_i^* are given by Equations 4.4, 4.3:*

$\mathcal{S}$ is schedulable with a fixed-job priority scheduling policy if and only if $\mathcal{S}^*$ is schedulable with Algorithm 1.

4.1.2.2. Fixed-job priority

Let us now consider fixed-job scheduling policies. Let $\tau_i \xrightarrow{M_{i,j}} \tau_j$ be an extended precedence constraint. With fixed-job priority scheduling, we can now modify the precedence encoding, so that deadlines and release dates of jobs of τ_i that are unrelated to jobs of τ_j are not adjusted. Since job precedence constraints expressed with precedence matrices follow periodic patterns, the sequence of adjusted attributes of successive jobs is also periodic. To represent such periodic values, we use *periodic words*, where each value of the word corresponds to the deadline, or release date, of a job. Let $(u)^\omega$ denote the infinite sequence of integers consisting of the infinite repetition of the finite sequence of integers u. Let us also introduce the following notations:

- For a finite word v, $|v|$ denotes its length;
- $w[n]$ denotes the n^{th} value of the periodic word w, i.e for $w = (u)^\omega$, we have $w[n] = u[(n) \mod |u|]$;

- The minimum of two periodic words $w = min(w_i, w_j)$ is such that for all n, we have $w[n] = min(w_i[n], w_j[n])$. In case $|w_i| \neq |w_j|$, we unfold w_i and w_j over length $lcm(|w_i|, |w_j|)$, before computing the pointwise minimum;
- The maximum of two periodic words is defined similarly.

The *deadline word* dw_i and *offset word* ow_i of τ_i are obtained as follows:

$$ow_j = \max_{\tau_i \in preds(\tau_j)} ow_{i,j} \tag{4.5}$$

$$dw_i = \min_{\tau_j \in succs(\tau_i)} dw_{i,j} \tag{4.6}$$

Where, for all $\tau_i \xrightarrow{M_{i,j}} \tau_j$, for all n, n', in $\mathbb{N}$, we have:

$$ow_{i,j}[n'] = \max_{(n,n') \in M^{\omega}_{i,j}} (O_j, ow_i[n] + nT_i - n'T_j) \tag{4.7}$$

$$dw_{i,j}[n] = \min_{(n,n') \in M^{\omega}_{i,j}} (D_i - (ow_i[n] - O_i), ow_j[n'] + n'T_j + dw_j[n'] - C_j - ow_i[n] - nT_i) \tag{4.8}$$

Example 4.1.1. Let $\tau_i(0, 1, 2, 2)$ and $\tau_j(0, 1, 3, 3)$ be two tasks with $\tau_i \xrightarrow{M} \tau_j$ and $M = \{(0, 0), (2, 1)\}$ (this corresponds to the tasks of Figure 2.9d). We get the following adjusted attributes:

$$dw_i = (2.2.1)^{\omega} \quad ow_i = (0)^{\omega}$$
$$dw_j = (3)^{\omega} \quad ow_j = (0.1)^{\omega}$$

Although formulae 4.7,4.8 are defined over an infinite number of jobs, we can compute $ow_{i,j}$ and $dw_{i,j}$ over finite lengths. When computing periodic words for the complete task set, we start with words of size 1, and then expand their size when needed. When the encoding is complete, word sizes are bounded as follows:

Property 1. *Let $\mathcal{S}$ be a dependent task set, with extended precedence constraints defined by a set of precedence matrices $M = \{M_{i_1,j_1}, \ldots, M_{i_l,j_l}\}$. For all task τ_i of $\mathcal{S}$, the size of both ow_i and dw_i is upper-bounded by $lcm(\{T_j\})/T_i$, where $\{\tau_j\}$ is the set of tasks in the connected component of τ_i.*

Finally, we establish the optimilaty of this encoding approach:

Theorem 4.1.5. *Let $\mathcal{S} = \{\tau_i(T_i, C_i, O_i, D_i)\}$ be a dependent task set, with extended precedence constraints defined by a set of precedence matrices $M = \{M_{i_1,j_1}, \ldots, M_{i_l,j_l}\}$. Let $\mathcal{S}^* = \{\tau'_i(T_i, C_i, ow_i, dw_i)\}$ be a set of independent tasks such that ow_i and dw_i are given by Equations 4.5, 4.6:*

$\mathcal{S}$ is scheduling with a fixed-job priority scheduling policy if and only if $\mathcal{S}^*$ is schedulable with EDF.

4.1.3. Conclusion

The main highlights concerning this work are listed below:

- The schedulability analysis of tasks with simple and extended precedence constraints, with a fixed-task priority scheduler, was presented at the RTAS'10 conference [For+10];
- The schedulability analysis of tasks with extended precedence constraints, with a fixed-job priority scheduler, was presented at the ETFA'11 conference [For+11];
- This work is a collaboration with colleagues from the University of Poitiers (Emmanuel Grolleau, Pascal Richard) and from Onera Toulouse (Frédéric Boniol, Claire Pagetti);
- The precedence encoding techniques developed in this work have been integrated in Prelude, and in the work on task clustering presented in Section 4.3.

Related works, before our publications Scheduling simple precedence constraints has been a well-understood problem since the 90's [CSB90; SS94]. Surprisingly though, our work was the first to propose a schedulability analysis for simple precedence constraints with a fixed-task priority scheduler (i.e. to transpose the results from fixed-job priority scheduling). It was also the first to consider extended precedence constraints.

Related works, after our publications The problem of scheduling dependent tasks can be considered a closed research topic in unicore, and the focus has now shifted to multicore (where it is known as *DAG scheduling*). A recent survey on multicore dependent task scheduling can be found in [Ver20].

4.2. End-to-end constraints analysis

Over the past five decades, the real-time community has developed an extensive set of scheduling algorithms and analysis techniques to ensure that, at system execution, each task completes before its deadline. In this section, I present my work on a different type of real-time analysis, called *end-to-end analysis*, which analyses the time required for performing a chain of tasks, from input acquisition to output production.

Control-command systems development typically starts by specifying a high-level model of the system, such as a Matlab/Simulink model, or a Prelude program in our case. The model is then translated into lower-level code, typically C, either manually or using code generation tools. In our approach, we perform the end-to-end analysis at the model level (i.e. at the task-set level), which simplifies the analysis by abstracting from implementation details. Furthermore, we can detect an end-to-end constraint violation during the model conception and modify the model accordingly, which is less error-prone and less time-consuming than performing corrections on the final implementation.

4.2.1. Motivating example

The notion of end-to-end constraints is illustrated on a subset of the longitudinal Flight Control System (Figure 4.1) extracted from the Rosace case-study [Pag+14]. It corresponds to a subset of the example presented previously in Figure 3.6, but in a multi-periodic version. This system

controls the angle of the control surface ($order$) based on: the current surface angle ($angle$), the current aircraft vertical speed (vz), the current aircraft altitude ($altitude$) and the altitude required by the pilot ($required\ altitude$). The software architecture consists of six tasks that make up three computation chains. Tasks operate at three different rates (30ms, 40ms, 60ms) and some tasks of different rates communicate.

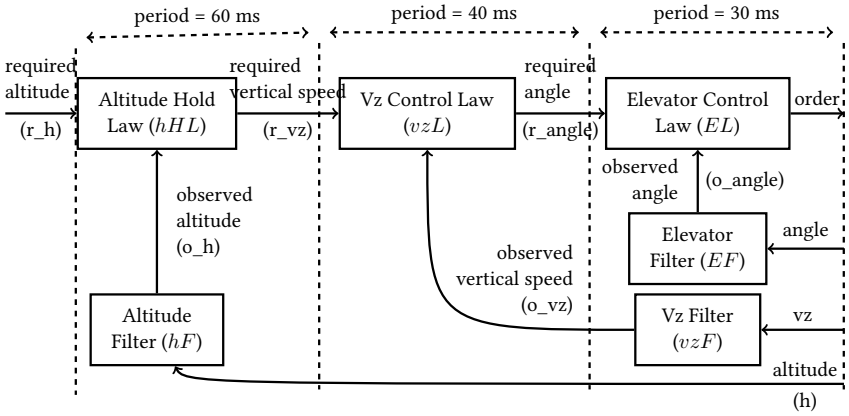


Figure 4.1.: Vertical speed control from Rosace

The most widely considered end-to-end constraints are *latency constraints*. As an example, let us consider the navigation computation chain (r_h, hHL, vzL, EL, $order$). The time elapsed between a pilot order ($required\ altitude$) and the modification of the control surface angle ($order$) is required to be less than 1s. Assuming that the control surface requires 400ms to reach an ordered angle, the end-to-end latency of the computation chain must thus be less than 600ms.

A possible temporal behaviour of this chain is detailed in Figure 4.2. Arrows stand for backwards data-dependencies, for instance $\tau_{order.4}$ depends on $\tau_{r\_h.0}$. Arrows between two jobs of the same task correspond to internal delays (like the `fby` synchronous operator). The worst-case latency of a pair of dependent task instances ($\tau_{order.p}$,$\tau_{r\_h.q}$) is when $\tau_{r\_h.q}$ is executed at the beginning of its period and $\tau_{order.p}$ is executed at the end of its period. Then, we consider the worst-case latency for each input paired with the first output that depends on it (e.g. we ignore the latency of ($\tau_{order.5}$,$\tau_{r\_h.0}$)). The behaviour described in this diagram satisfies the 600ms latency constraint: the worst-case latency is 150ms, achieved for ($\tau_{order.4}$,$\tau_{r\_h.0}$) and ($\tau_{order.6}$,$\tau_{r\_h.1}$).

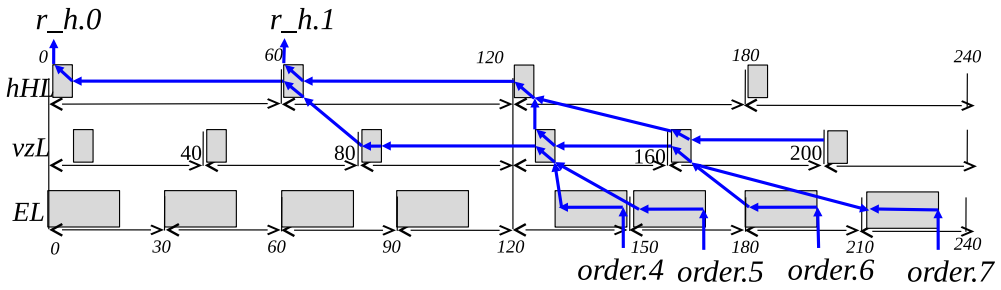


Figure 4.2.: Chain from *r_h* to *order*

4.2.2. End-to-end properties definition

In this section we present a small language dedicated to the specification of end-to-end real-time constraints and use this language to define some classic end-to-end constraints. The complete constraints language is detailed in Figure 4.3. Constraints are defined with respect to some (transitive) dependence $\tau_j \leftarrow^* \tau_i$, and involve the following terms :

- $etime(\tau_{i.p})$ denotes the earliest completion time, and $ltime(\tau_{i.p})$ the latest completion time, of job $\tau_{i.p}$;
- $\tau_{i.rlv(p)}$ denotes the p^{th} relevant job of τ_i. The job $\tau_{i.p}$ is relevant to τ_j iff some job of τ_j depends on it;
- $\tau_{j.last(rlv(p))}$ denotes the last job of τ_j that depends on $\tau_{i.rlv(p)}$
- $\tau_{j.first(rlv(p))}$ denotes the first job of τ_j that depends on $\tau_{i.rlv(p)}$

$$
\begin{array}{lcl}
x_S & ::= & x \mid x-1 \mid x+1 \mid k \in \mathbb{N} \\
idx & ::= & rlv(x_S) \mid idx-1 \mid idx+1 \\
 & & \mid last(idx) \mid first(idx) \\
date & ::= & etime(\tau_{task.idx}) \mid ltime(\tau_{task.idx}) \\
duration & ::= & date - date \mid \min_{x\in\mathbb{N}}(date - date) \\
 & & \mid \max_{x\in\mathbb{N}}(date - date) \\
formula & ::= & duration <= k \in \mathbb{N}
\end{array}
$$

Figure 4.3.: End-to-end constraints specification language

For instance, in the example of Figure 4.2, for the dependence $vzL \leftarrow^* EL$, we have $rlv(0) = 2$, meaning that $\tau_{vzL.0}$ and $\tau_{vzL.1}$ are unused (irrelevant). Then, $first(0) = 4$ and $last(0) = 5$, because $\tau_{vzL.rlv(0)=2}$ is first used by $\tau_{EL.4}$ and last used by $\tau_{EL.5}$. We can then use this constraints language to define classic end-to-end properties, as detailed below.

Best-case latency The latency of a functional chain can be defined informally as the time needed for an input value to propagate to an output value (also called *first-to-first* delay in [Fei+08]). The best-case latency (BCL) of a pair of dependent values is achieved when a relevant input is read at the latest and its first successor output is produced at the earliest. Considering all pairs of dependent values, we get the following definition.

Definition 4.2.1. Let $\tau_n \leftarrow^* \tau_1$:

$$BCL(\tau_1, \tau_n) = \max(0, \min_{x\in\mathbb{N}}(etime(\tau_{n.first(rlv(x))}) - ltime(\tau_{1.rlv(x)})))$$

Worst-case latency To define the worst-case latency (WCL) of a chain, we need to consider all of its pairs of dependent values. However, as illustrated in Figure 4.4, we also need to consider unused values: since $\tau_{vz.2}$ is irrelevant, if the value of vz changes during its third sampling period (i.e. for $\tau_{vz.2}$) and provided it remains unchanged until the fourth sampling period (i.e.

for $\tau_{vz.3}$), then *order* will be impacted only at its fifth period (i.e. for $\tau_{order.4}$). This leads us to the definition below.

Definition 4.2.2. Let $\tau_n \leftarrow^* \tau_1$:

$$WCL(\tau_1, \tau_n) = \max_{x \in \mathbb{N}^*}(ltime(\tau_{n.first(rlv(x))}) - etime(\tau_{1.rlv(x-1)+1}))$$

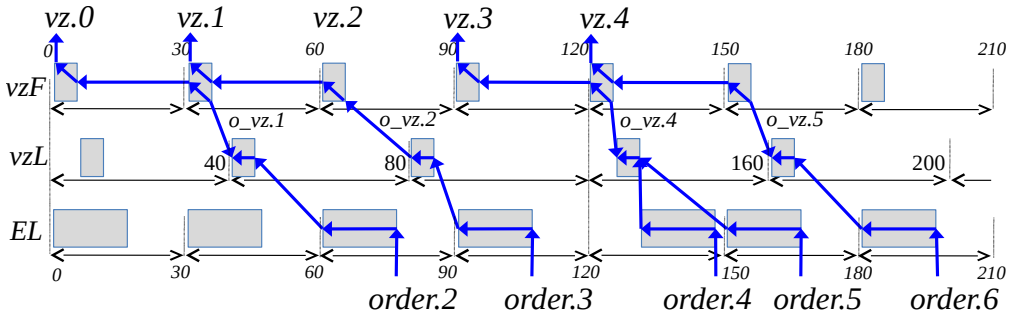


Figure 4.4.: Chain from vz to $order$

Freshness Let $\tau_n \leftarrow^* \tau_1$. At any time t, the freshness is the difference $t - t_1$, where t_1 is the reading date of the value of τ_1 used to compute the current value of τ_n (also called *last-to-last* delay in [Fei+08]). Consequently, a freshness constraint $Freshness(\tau_1, \tau_n) \leq \Delta_f$ requires that, at every time t, the value of τ_1 used to compute the current value of τ_n has been read no earlier than $t - \Delta_f$. Let us consider the chain $(vz, ..., order)$ depicted Figure 4.4. We have $\tau_{order.2} \leftarrow \tau_{vz.0}$. Considering that $\tau_{vz.0}$ is acquired at $t \in [0, 30)$ and that $\tau_{order.2}$ is produced at $t' \in [60, 90)$, then the freshness of $\tau_{order.3}$, that is $t' - t$, is bounded as follows: $60 - 30 < t' - t < 90 - 0$. The worst-case freshness (WCF) is achieved for the pair $(\tau_{vz.3}, \tau_{order.5})$ and is $180 - 90 = 90$. More formally:

Definition 4.2.3. Let $\tau_n \leftarrow^* \tau_1$:

$$WCF(\tau_1, \tau_n) = \max_{x \in \mathbb{N}}(ltime(\tau_{n.last(rlv(x))}) - etime(\tau_{1.rlv(x)}))$$

Reactivity The reactivity of a chain between two tasks τ_1 and τ_n characterizes the minimal duration of any value change on τ_1 such that this change is eventually propagated to τ_n. For instance, Figure 4.4, shows that $\tau_{vz.2}$ does not impact any occurrence of *order*. Let us consider the following scenario: (1) $\tau_{vz.1}$ is read at the earliest, i.e. at 30; (2) the value of *vz* changes just after 30 but then gets back to its previous value just before 120; (3) $\tau_{vz.3}$ is read at the latest, i.e. at 120. In this scenario, the variation on the value of *vz* goes "unnoticed" by the system, because $\tau_{vz.2}$ is not a relevant occurrence of *vz*.

Definition 4.2.4. Let $\tau_n \leftarrow^* \tau_1$. $WCR(\tau_n, \tau_1) =$

$$\max_{x \in \mathbb{N}}(ltime((\tau_{\tau_1.rlv(x+1)})) - etime(\tau_{\tau_1.rlv(x)}))$$

4.2.3. End-to-end properties verification

Our verification procedure is applied to a set of periodic dependent tasks with implicit deadlines $\{\tau_i(0, C_i, D_i = T_i)\}$, where dependence constraints are defined by dependence matrices. Dependence matrices define dependence relations between jobs in the exact same way as precedence matrices define precedence relations between jobs (see Section 2.2.2).

To enable the verification of end-to-end properties on a given task set, we only need to specify how to compute the different terms of the constraints language (relevant jobs, earliest completion times, etc).

4.2.3.1. Transitive dependence relation

First, we focus on how to determine which pair of jobs of a functional chain are dependent, that is to say, how to compute the transitive closure of the dependence relation based on dependence matrices.

Definition 4.2.5. Let M_1, M_2, be dependence matrices. $\tau_i \overset{M_1 \circ M_2}{\leftarrow^*} \tau_j$ denotes a dependence relation defined as follows:

$$\forall p, q \in \mathbb{N}^2, \tau_{\tau_i.p} \leftarrow^* \tau_{\tau_j.q} \text{ iff } \exists t \in \mathbb{N} | (p,t) \in M_1^\omega, (t,q) \in M_2^\omega$$

Algorithm 2 describes a procedure to compute the composition of two dependence matrices, using the following auxiliary functions:

- $unfold(M, n, T_o, T_i)$ unfolds precedence matrix M over duration n;
- $pat\_size(M, T_o, T_i)$ returns the duration covered by one pattern of M. It is equal to $|M| * T_o$. For instance, in Figure 4.2, for $(r\_h, ..., order)$, we have $pat\_size(M, 30, 60) = 4 * 30 = 120$;
- $closure(M_1, M_2)$ takes two precedence matrices M_1,M_2 and returns the precedence matrix consisting of the pairs (p, q) such that there exists t with $(p, t) \in M_1$ and $(t, q) \in M_2$.

Algorithm 2 Procedure $\mathcal{C}$ for composing M_1 with M_2

Require: $\tau_i \overset{M_1 \circ M_2}{\leftarrow} \tau_j$
$hp \leftarrow lcm(pat\_size(M_1, T_i, T_j), pat\_size(M_2, T_i, T_j))$
$M_1' \leftarrow unfold(M_1, hp, T_i, T_j)$
$M_2' \leftarrow unfold(M_2, hp, T_i, T_j)$
return $closure(M_1', M_2')$

In the following, we consider that the composition is computed with Algorithm 2. To compute the dependence matrix of a whole functional chain, we also need to account for internal task delays.

Definition 4.2.6. Let $\tau_a \overset{M_1}{\leftarrow}{}^* \tau_b$ and $\tau_b \overset{M_2}{\leftarrow} \tau_c$. Let $del_b(c, a)$ denote the internal delay in τ_b, when processing data from τ_a to τ_c. We have $\tau_a \overset{M}{\leftarrow}{}^* \tau_c$, with:

$$M = M_1' \circ M_2$$
$$M_1' = \{(p, q) | \exists p', p = p' + del_b(c, a), (p', q) \in M_1\}$$

4.2.3.2. Completion times

We abstract from actual execution dates and from execution times, and only consider best and worst-cases. Thus, we let $etime(\tau_{i.p}) = T_i * p$ and $ltime(\tau_{i.p}) = T_i * (p + 1)$. This leads to a safe over-approximation of end-to-end properties. In a way, this is similar to how synchronous languages abstract from real-time to simplify programming. Furthermore, as shown in [Fei+08; MMTS13], the contribution of task periods to end-to-end delays is far more important than that of task execution times.

To compute completion times in our formulas, we also need to compute the indices of relevant jobs, and the indices of their successors. This is done based on the following definition.

Definition 4.2.7. Let M be a dependence matrix. We define an equivalence relation $\sim$ on M and a partitioning of M into subsets $[M]_i$ as follows:

- $\forall (p, q), (p', q') \in M^2, ((p, q) \sim (p', q')) \equiv (q = q')$
- $M/\sim$ is the quotient of M by $\sim$ (its set of equivalence classes);
- $\forall i, 1 \leq i < |(M/\sim)|, \forall (p, q), (p', q') \in M^2$:

$$[M]_i \in (M/\sim)$$
$$(p, q) \in [M]_i \wedge (p', q') \in [M]_{i+1} \Rightarrow q < q'$$

Additionally:

- Let $in([M]_i)$ be such that for all $(p, q) \in [M]_i$, $q = in([M]_i)$;
- Let $outs([M]_i)$ denote the first projection of $[M]_i$.

For instance, for the precedence matrix $M = \{(3, 1), (4, 2), (5, 4), (6, 4)\}$, we have $[M]_1 = \{(3, 1)\}, [M]_2 = \{(4, 2)\}, [M]_3 = \{(5, 4), (6, 4)\}$. We also have $in([M]_3) = 4$ and $outs([M]_3) = \{5, 6\}$.

Then, for $\tau_n \overset{M}{\leftarrow}{}^* \tau_1, \forall q \in \mathbb{N}^*$, we have:

$$rlv(x) = in([M^\omega]_x)$$
$$last(rlv(q)) = \max(outs([M^\omega]_q))$$
$$first(rlv(q)) = \min(outs([M^\omega]_q))$$

To verify that an end-to-end property holds, according to the formulas of our constraints language we need to consider every pair of dependent values of the concerned tasks. However, since dependence constraints are defined by dependence matrices, to check a formula of our language we only need to check the formula for each relevant input in one pattern of the

dependence matrix. Finally, the complexity of the verification of an end-to-end property is stated below.

Theorem 4.2.1. *Let $(\tau_1, \ldots, \tau_n)$ be a functional chain where data-dependencies are specified by dependence matrices. Any formula of the constraints language (Figure 4.3) for this functional chain can be verified with complexity $\mathcal{O}(lcm_{1\leq k\leq n}(T_k) \times n)$.*

4.2.4. Related works

Before our publications Following initial work in [Fei+08], end-to-end properties analysis has been studied extensively over the past decade. In [Raj+10; Moh+13; Mub+15], it has been studied for *register buffer communications*, where the producer task writes in the buffer when it completes and the consumer task reads from it when it starts. Synchronous/causal communications have also been studied in [GTW11; Bec+16; Kha+16].

After our publications End-to-end analysis has recently been studied in new settings, in particular: with sporadic tasks [Dür+19], taking the schedule into account [KBS20], or considering the Logical Execution Time model [BDN18].

4.2.5. Conclusion

The main highlights concerning this work are listed below:

- The analysis of end-to-end latency properties was first presented at the SAC'13 conference [Wys+13];
- A more systematic approach to the analysis of general end-to-end properties (as presented in this section) was published at the ETFA'17 conference [FBP17];
- This work is a collaboration with colleagues from Onera Toulouse (Frédéric Boniol, Claire Pagetti, and Rémy Wyss).

4.3. Task clustering

A real-time system is usually designed as a set of functionalities, or nodes in Prelude. Implementing such a design requires to map functionalities to real-time tasks. The most straightforward solution consists in mapping each functionality to a different task. However, complex industrial systems such as flight control systems can consist of up to 1000 functionalities [Bon+08]. Such a large number of tasks would incur a significant processing time overhead in context switching [Lee06] and an important memory footprint. The thesis of Antoine Bertout studied the *task clustering* problem, i.e. how to compute a functionality-to-task mapping where several functionalities are mapped to (clustered in) the same task. Clustering several functionalities implies to choose only one deadline for the cluster, which effectively reduces the deadlines of some functionalities and may cause the system to become unschedulable. The objective of the task clustering is to reach a minimal number of tasks, while preserving the schedulability of the system.

4.3.1. Problem definition

From a schedulability point-of-view, functionalities can be considered as finer grain tasks. Then, mapping functionalities to tasks amounts to clustering several tasks into a single one, thus the term *task clustering*. The input of our task clustering problem is a set of periodic tasks with constrained deadlines, i.e. $\mathcal{S} = \{\tau_i(C_i, D_i, T_i)\}_{0 \leq i \leq n}$ with $D_i \leq T_i$. The output of the clustering has the same formal definition, though the number of tasks and their attributes are changed.

The clustering of two tasks produces a new task, as defined below.

Definition 4.3.1. Clustering τ_i and τ_j, where $D_i \leq D_j$, produces a task τ_{ij} with the following parameters:

$$C_{ij} = C_i + C_j \tag{4.9}$$

$$T_{ij} = T_i = T_j \tag{4.10}$$

$$D_{ij} = \begin{cases} D_j & \text{if } (D_j - C_j \leq D_i) \vee (R_j - C_j \leq D_i) \quad \text{(4.11c)} \\ D_i & \text{otherwise} \quad \text{(4.11d)} \end{cases}$$

Note that in our work we only cluster tasks of the same period. The WCET of the cluster is simply the sum of the WCETs of its constituents. Concerning the deadline, setting $D_{ij} = D_i$ ensures that if the cluster meets its deadline, then so do its constituents (since D_i is the smallest of the two). Case 4.11c is not always applicable, but induces a less pessimistic deadline. It also requires to execute τ_i before τ_j in the cluster (while the order can be arbitrary in case 4.11d). In the following, we let $\tau_{i'}$ and $\tau_{j'}$ denote the respective portions corresponding to the clustered tasks inside τ_{ij}. The following property states that clustered tasks respect their original constraints as long as the cluster respects its own.

Theorem 4.3.1. *Let $\mathcal{S} = (\{\tau_x(C_x, D_x, T_x)\}_{1 \leq x \leq n})$ and $\mathcal{S}' = (\mathcal{S} \setminus \{\tau_i, \tau_j\} \cup \{\tau_{ij}\})$ be two task sets, where τ_i and τ_j are two tasks of $\mathcal{S}$ with $D_i \leq D_j, T_i = T_j$. Let Φ be a priority assignment for $\mathcal{S}'$. Then:*

$$\mathcal{S}' \text{ schedulable under } \Phi \Rightarrow \forall \tau_{ij.k}, \begin{cases} e_\Phi(\tau_{i'.k}) \leq d_{i.k} \wedge e_\Phi(\tau_{j'.k}) \leq d_{j.k} \\ s_\Phi(\tau_{i'.k}) \geq o_{i.k} \wedge s_\Phi(\tau_{j'.k}) \geq o_{j.k} \end{cases}$$

The choice of the deadline for the cluster has an impact on the schedulability of the task set. Indeed, when setting $D_{ij} = D_i$, we effectively reduce the deadline for τ'_j, compared to its non-clustered counterpart τ_j. Even if the task set does not immediately become unschedulable, this kind of clustering makes the task set harder to schedule, thus further clusterings become less likely to yield schedulable task sets. On the contrary, in case we set $D_{ij} = D_j$, which is only allowed when the conditions of Equation 4.11c are satisfied, the clustering does not degrade the schedulability of the task set. Such a clustering is called a *zero-cost clustering*.

We define the notion of *valid cluster* below, and let the set of all valid clusters of $\mathcal{S}$ be denoted $\mathcal{C}(\mathcal{S})$. Then, we state the clustering problem.

Definition 4.3.2. Let $\mathcal{S} = \{\tau_i(C_i, D_i, T_i)\}_{0 \leq i < n}$ and $\mathcal{S}' = \{\tau'_i(C'_i, D'_i, T'_i)\}_{0 \leq i < n'}$ be two task sets with $n' \leq n$. We say that $\mathcal{S}'$ is a valid cluster for $\mathcal{S}$ iff:

- $\mathcal{S}$ schedulable $\Rightarrow$ $\mathcal{S}'$ schedulable;

- $\mathcal{S}'$ can be obtained by applying to $\mathcal{S}$ a succession of clusterings performed as defined in Definition 4.3.1.

Definition 4.3.3 (Clustering problem)**.** Given a task set $\mathcal{S} = (\{\tau_x(C_x, D_x, T_x)\}_{1 \leq x \leq n})$, the *clustering problem* consists in finding the element of $\mathcal{C}(\mathcal{S})$ with the least cardinality.

We can easily establish that the clustering problem is at least as hard as co-NP-complete problems. Indeed, solving the clustering problem requires to perform schedulability tests as the clustering progresses. It is well-known that the testing feasibility for periodic tasks with constrained deadlines is a co-NP-complete problem [LW82; BRH90]. In the following, we detail heuristics whose objective is to produce a valid cluster whose cardinality is as close as possible to the theoretical minimum.

4.3.2. Guiding principles

In this section, we examine a set of guiding principles that we applied to design our clustering heuristics. First, the overall structure of our heuristic is as follows. We start from an initial task set where each task is considered as a cluster with one element. We progressively try to group more and more clusters together to minimize the cardinality of the task set. At each step, we try to group one cluster with another. To select the best clustering candidate, we rely on a heuristic *cost function* that estimates which candidate is the most likely to lead to the minimal clustering. Formulated that way, the clustering becomes an optimisation problem and can thus be solved using classic heuristics based on cost functions, such as for instance Greedy Best-First Search, A*, or Simulated Annealing. In our experiments, we rely on Greedy BFS.

Second, we note that when a task set is unschedulable, applying a clustering to it also yields an unschedulable task set, since this will yield an even more constrained task set. This is tightly related to the notion of *sustainable schedulability* introduced by Baruah and Burns in [BB06]. In a nutshell, a task set deemed unschedulable remains so when some of the task parameters are changed by one of the following means: (i) increasing the execution time; (ii) decreasing the period; (iii) decreasing the deadline. Thus, it is useless to apply further clusterings to a task set that was deemed unschedulable.

We need a schedulability test to determine valid task clusterings. In the following, we only consider exact or sufficient tests. This ensures that the task sets obtained after clustering are schedulable, even though this reduces the chance to obtain the minimum number of clusters. We can distinguish two types of schedulability tests: Boolean schedulability tests and response time tests. On the one hand, Boolean tests give a Boolean answer, determining only whether a task set is schedulable or not. On the other hand, exact tests based on *response time analysis* (RTA, e.g. [JP86]) provide worst response time for each task. We favor RTA tests, because they provide insight on how close the task set is to becoming unschedulable, and thus can be used as cost functions for our heuristics.

4.3.3. Independent tasks, uniprocessor

In this section, we present our results for the task clustering problem for independent tasks on a uniprocessor hardware platform.

4.3.3.1. Heuristic

We experimented with different cost functions, listed below.

- Deadline-based density (favor task sets with the highest remaining scheduling margin):

$$\min \sum_{i=1}^{n} \frac{C_i}{D_i}$$

- Response-time-based density (more precise, but high time-complexity to compute R_i):

$$\min \sum_{i=1}^{n} \frac{R_i}{D_i}$$

- Laxity over the hyper-period (laxity weighted with task periods):

$$\max \sum_{i=1}^{n} (D_i - C_i) \cdot \frac{H}{T_i}$$

- Random (to compare with other cost functions).

As described in Algorithm 3, we recursively enumerate clusterings. At each recursive call, we first try to apply a zero-cost clustering on each generated child. If the zero-cost condition is respected we make a recursive call with the new cluster, if not, we accumulate a 3-tuple containing the task set and indices of the two tasks we want to group in a buffer. Then, we compute a set of valid children, and choose the most promising child according to the heuristic cost function. This heuristic can be used with different schedulability tests. The choice of the schedulability test depends on the considered scheduling policy (fixed-task or fixed-job scheduling policy), but also on the complexity we want to achieve.

4.3.3.2. Experimental results

Experiments are performed on a quad-core Intel Xeon, 2.4 GHz, 32Go RAM. We apply our heuristic to randomly generated task sets. Each task utilization ($U_i = \frac{C_i}{T_i}$) is computed following the classic UUnifast [BB04] method. We denote as u the overall utilization factor of the processor. Each task period T_i is uniformly distributed between a set of a maximum of 10 different periods by task set using method [GM01], ensuring that the simulation can be limited to a reasonable hyper-period. Concerning WCETs, we take $C_i = T_i \times U_i$. For deadlines, following [GM01] we take $D_i = round((T_i - C_i) \times rand(d1, d2)) + C_i$ with $0 \leq d1 \leq d2$.

First, we show the number of clusters obtained depending on the initial number of tasks of the task set and on the processor utilization. Figure 4.5 shows the results for DM, while Figure 4.6 shows the results for EDF. The number of clusters cannot be lower than 10 (the number of different periods of the task set). Each point corresponds to the average value obtained over 1000 task sets. The cost function used in these experiments is the response-time-based density. We observe that the task clustering reduces the number of tasks by at least a factor 10, and the number of clusters increases with the utilization of the task set.

Algorithm 3 Task clustering algorithm
Function clustering(S)

Require: $\mathcal{S} = (\{\tau_i\}_{0 \leq i < n})$: initial set of tasks in non-decreasing deadline order
  children $\leftarrow \varnothing$
  ▷ Try zero-cost clustering.
  for $i = n - 1$ to 0 **do**
    for $j = i - 1$ to 0 **do**
      if $T_i == T_j$ **then**
        if $(C_i + C_j \leq D_j) \wedge ((D_j - C_j \leq D_i) \vee (R_j - C_j \leq D_i))$ **then**
          $S' \leftarrow \{S \setminus \{\tau_i, \tau_j\}\} \cup \tau_{ij}$ **return** clustering(S')
        else
          children $\leftarrow \{$children $\cup (\mathcal{S}, i, j)\}$
        end if
      end if
    end for
  end for
  childrenSched $\leftarrow \varnothing$
  ▷ Find valid children
  for all $(M, x, y) \in$ children **do**
    if $C_x + C_y \leq min(D_x, D_y)$ **then** ▷ Check for non-negative laxity (optimisation)
      $M' \leftarrow \{M \setminus \{\tau_x, \tau_y\}\} \cup \tau_{xy}$
      if schedulable(M') **then**
        childrenSched $\leftarrow$ childrenSched $\cup \{M'\}$
      end if
    end if
  end for
  ▷ continue with child with highest cost
  if childrenSched $\neq \varnothing$ **then return** clustering(highestCost(childrenSched))
  else
    return $\mathcal{S}$
  end if

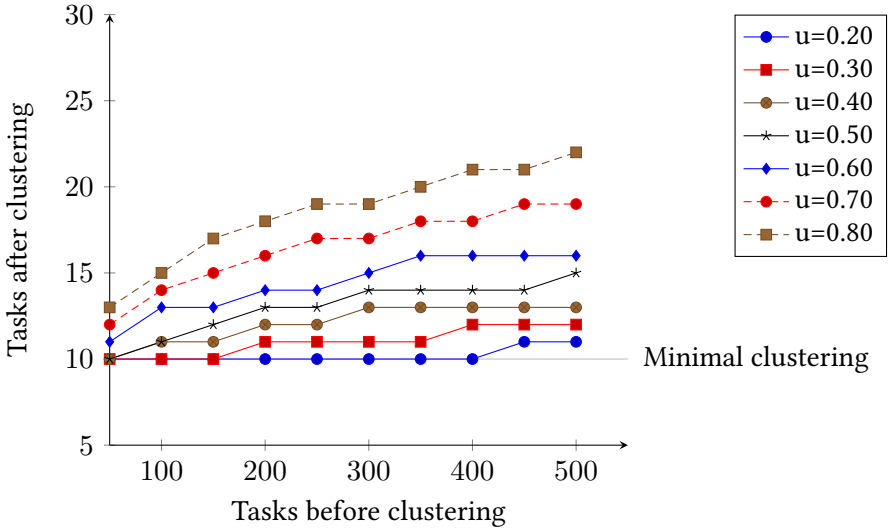


Figure 4.5.: Task clustering with DM (d_{min} = 0, d_{max} = 1)

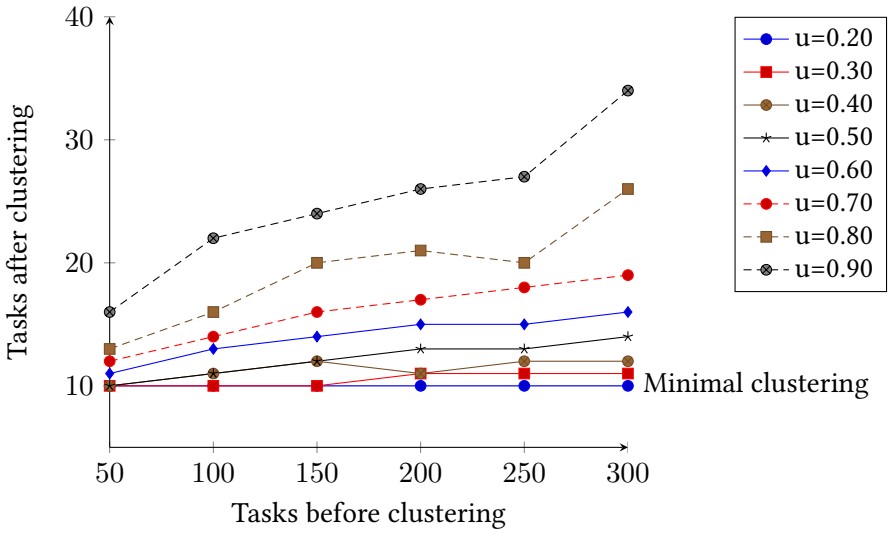


Figure 4.6.: Task clustering with EDF (d_{min} = 0, d_{max} = 1)

As a second set of experiments, we compared the impact of different cost functions on the efficiency of the clustering. Results are depicted in Figure 4.7 for DM (results for EDF have similar trends. We show the number of clusters obtained depending on the density of the initial task set. The cost function being compared are: purely random (random), minimal response-time-based density (RTDensity), minimal deadline-based density (MinDensity), maximal deadline-based density (MaxDensity). The maximal deadline-based density is meant as a reference to compare with other cost functions. We observe that the number of clusters increases with density. We also observe that the choice of the cost function does not have a very significant impact on the number of clusters.

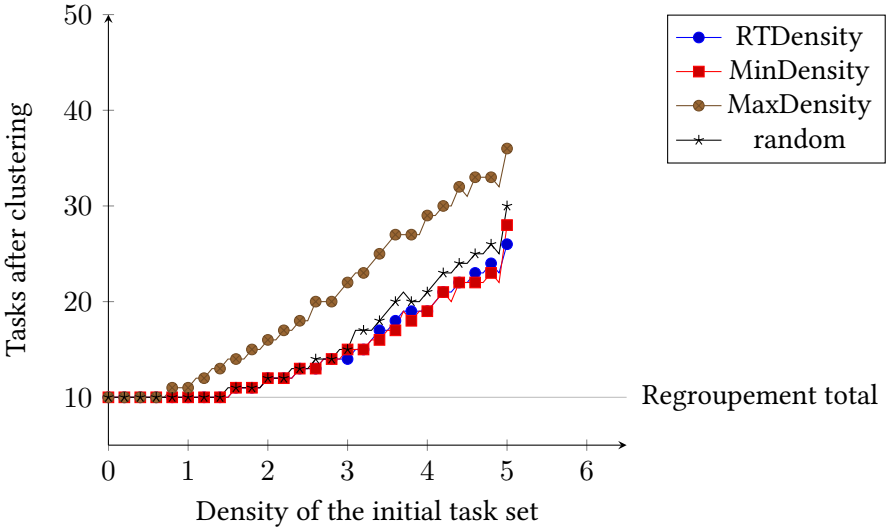


Figure 4.7.: Comparing cost functions with DM as a function of density (n = 300)

4.3.4. Dependent tasks, uniprocessor

In this section, we present our results for the task clustering problem for dependent tasks on a uniprocessor hardware platform.

4.3.4.1. Adapting the heuristic

Our results apply to tasks with constrained deadlines, no offset, and simple precedence constraints. Precedence constraints restrict the possible clusterings of the task graph, as illustrated in Figure 4.8 (tasks related by a red dashed arrow cannot be clustered together). A cluster is *valid* if and only if, in addition to being schedulable, one of the following conditions is satisfied:

- There exists a precedence constraints between the two. In that case, we order the tasks inside the cluster according to the precedence constraint (e.g. τ_1 before τ_3);
- Tasks are *isolated* in the sense that they have no precedence constraint with the rest of the task graph;
- Tasks are independent (there is not transitive precedence relation between them) and one of them is isolated (e.g. τ_1 and τ_9).

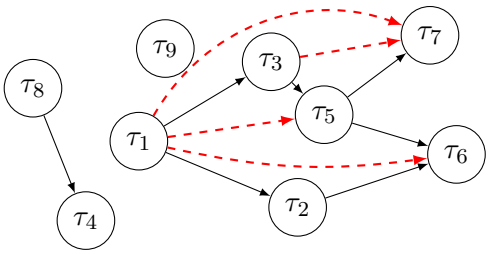


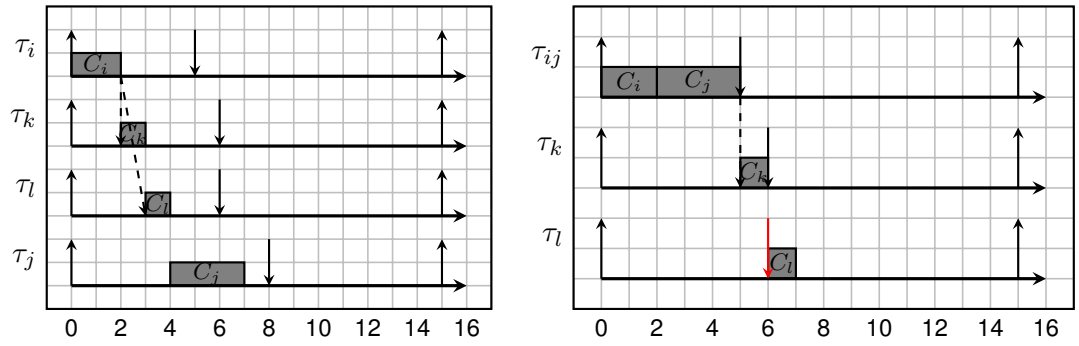
Figure 4.8.: Infeasible task clusterings

The real-time attributes of a cluster are computed as previously. Provided that tasks are ordered according to their precedence constraints in their cluster, clustered tasks respect their original constraints as long as the cluster respects its own.

Theorem 4.3.2. *Let $\mathcal{S} = (\{\tau_i\}_{0 \leq i < n\}, \rightarrow)$ be a dependent task set, and τ_i, τ_j be two tasks of $\mathcal{S}$ with $T_i = T_j$. Let $\mathcal{S}' = (\mathcal{S} \setminus \{\tau_i, \tau_j\} \cup \{\tau_{ij}\})$. Then:*

$$\mathcal{S}' \text{ schedulable under } \Phi \Rightarrow \begin{cases} \forall \tau_{ij.k}, e(\tau_{i'.k}) \leq d_{i.k} \\ \forall \tau_{ij.k}, s(\tau_{i'.k}) \geq o_{i.k} \\ \forall \tau_{i.k} \rightarrow \tau_{j.k}, e(\tau_{i'.k})) \leq o_{j.k} \end{cases}$$

Unfortunately, schedulability preservation in the case of Equation 4.11c (i.e. when $(R_j - C_j \leq D_i)$) does not hold for dependent tasks. This is illustrated in Figure 4.9. We assume that $\tau_i \rightarrow \tau_j$ and $\tau_i \rightarrow \tau_k$. After the clustering, we have $\tau_{ij} \rightarrow \tau_k$ and the system becomes unschedulable. Even though this implies that we cannot benefit from zero-cost clustering as before (i.e. we must always check schedulability after a clustering), checking the condition of Equation 4.11c is still beneficial, since it yields a less constrained deadline.



(a) Schedule before the clustering

(b) Schedule after the clustering

Figure 4.9.: Schedulability not preserved due to constraint $\tau_{ij} \rightarrow \tau_k$

The heuristic used to cluster dependent tasks is very similar to the one previously proposed for independent tasks. The main notable differences are the following:

- We check that the clustering does not yield unfeasible precedence constraints (such as the red arrows of Figure 4.8);
- We favor zero-cost clustering of isolated tasks;
- When clustering tasks related by a precedence constraint, we order them accordingly in the cluster;
- We rely on precedence encoding (from Section 4.1.1) to check schedulability. We also apply the cost functions on the encoded task set.

4.3.4.2. Experimental results

Experiments are performed with the same hardware settings, and with the same real-time attributes generation mechanisms as for independent tasks (Section 4.3.3.2). The generation of precedence constraints is performed according to two criteria. First, tasks are partitioned into a desired number of groups, where each group is called a *precedence level* of the task set. Then, we choose the probability for each pair of tasks belonging to two successive levels to be related by a precedence constraint.

Figure 4.10 shows the impact of the precedence probability for a fixed number of tasks and levels. We observe that the number of clusters increases with the probability of precedence constraints, which seems logical since precedence constraints reduce the number of feasible clusterings.

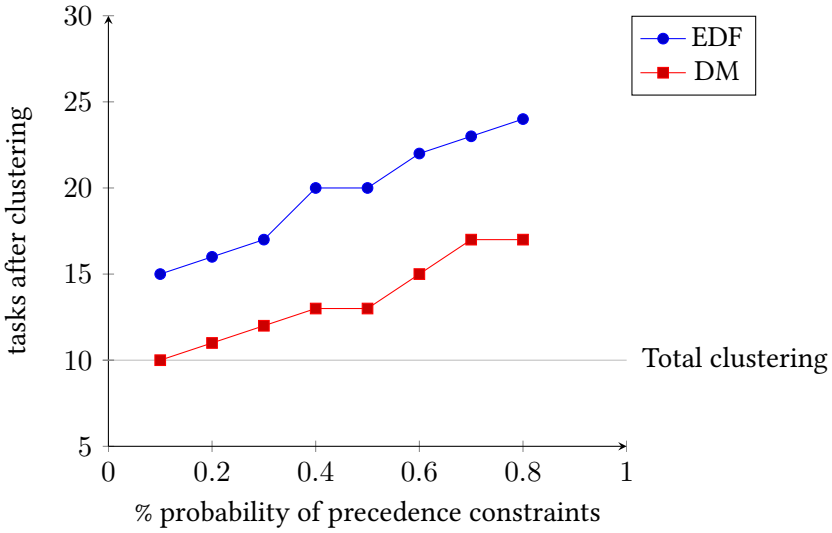


Figure 4.10.: Impact of the probability of precedence constraints on the task clustering (n = 200, u = 0.7, d_{min} = 0, d_{max} = 0.8); number of levels is 0.9 times the number of tasks

Tables 4.1, 4.2 show the number of clusters obtained for varying numbers of levels and precedence probability (the other parameters are the same as for Figure 4.10). We observe that the number of levels has a limited impact on the number of clusters, while the precedence probability has a more significant impact. Note that the number of levels of task graphs found in case studies of the literature (e.g. Rosace [Pag+14]) are around 0.5 times the total number of tasks, with a precedence probability around 0.25. We also observe that results are significantly worse for EDF. This is mainly because we had to disable the optimization based on response time analysis, due to the high complexity of its computation.

4.3.5. Dependent tasks, multiprocessor

In this section, we present our results for the task clustering problem for dependent tasks on a multiprocessor platform. We assume a platform with *identical processors*, meaning that all processors have the same characteristics and compute at the same speed. We consider *partitioned scheduling*, meaning that each task is assigned to one fixed processor for the whole execution of the system.

	0.25	0.50	0.75
0.4	12	15	18
0.5	11	15	18
0.6	11	15	18
0.7	11	14	18

Table 4.1.: Varying the number of levels and precedence probability with DM

	0.25	0.50
0.4	23	23
0.5	24	24
0.6	24	26
0.7	24	25

Table 4.2.: Varying the number of levels and precedence probability with DM

4.3.5.1. Scheduling

Since we consider partitioned scheduling, task are partitioned between the processors. Our partitioning strategy, is a slight adaption of that of [BBW11]. It partitions tasks based on the critical path of the task set (the chain of tasks whose sum of WCETs is the highest). It first assigns all tasks of the critical path, called a *flow*, to one processor. These tasks are then removed from the task set and the process iterates until all tasks are assigned. If the number of processors is reached before all tasks are assigned, the remaining tasks are assigned to the processors with the objective of keeping the processors utilization balanced. The original strategy of [BBW11] is targeted for a task set where all tasks have the same period. In order to support multiple periods, we weigh paths by their periods and define the weight W_j of a chain j as follows. Then the critical path is the chain of maximal weight in the task graph.

$$W_j = \sum_{\tau_i \in \text{ chain } j} \frac{C_i}{T_j} \tag{4.12}$$

The precedence encoding technique presented in Section 4.1.1 is targeted for uniprocessor. Unfortunately, it cannot directly be transposed to partitioned scheduling, because some dependent tasks might be assigned to different processors. Instead, to encode inter-processor precedence constraints, the release date of a task τ_i assigned to processor m_k is adjusted as follows:

$$O_i^* = \max(O_i, \max_{\tau_j \in preds(\tau_i), \tau_j \notin m_k} D_j^*) \tag{4.13}$$

This encoding is sufficient, since a task will always execute before its successors, but not necessary, in the sense that adjusted release dates might be unnecessarily small [WGD14].

4.3.5.2. Heuristic

Due to precedence constraints, clustering two tasks on one processor can impact tasks on another processor, so we cannot simply apply the clustering strategy proposed for uniprocessor separately on each processor. The clustering procedure for multiprocessor is described in Algorithm 4. We distinguish flows, depending on whether the tasks of a flow have precedence constraints with the other flows (interdependent flow) or not (independent flow). Independent flows can be clustered as described previously for the uniprocessor case. We iterate the procedure *clusterDepNonRec* on interdependent flows until no further clustering is feasible. Procedure *clusterDepNonRec* differs from the uniprocessor clustering procedure in that: 1) it performs a single clustering; 2) it adjusts release dates for inter-flow precedence constraints according to Equation 4.13; 3) it checks the feasibility of all flows after the clustering.

Algorithm 4 Multiprocessor task clustering algorithm
Function clustering($\mathcal{G}$)

Require: $(\mathcal{S}, \rightarrow)$: initial task set
  $\mathcal{F} \leftarrow$ partition($\mathcal{S}$)
  $invalid \leftarrow false$
  for all $Flow \in \mathcal{F}$ **do**
    if ! schedulable($Flow$) **then** ▷ After encoding
      $invalid \leftarrow true$
    end if
  end for
  if ! $invalid$ **then**
    $\mathcal{F}^{ind} \leftarrow$ independent($\mathcal{F}$)
    $\mathcal{F}^{dep} \leftarrow$ dependent($\mathcal{F}$)
    ▷ Process independent flows
    for all $Flow \in \mathcal{F}^{ind}$ **do**
      $Flow \leftarrow$clusterDep($Flow$)
    end for
    ▷ Process interdependent flows
    $CurrentFlow \leftarrow \mathcal{F}^{dep}[0]$
    $idxFlow \leftarrow 0$
    $nbInfeasibleClustering \leftarrow 0$
    while $nbInfeasibleClustering < |\mathcal{F}^{dep}|$ **do**
      $CurrentFlow' \leftarrow$clusterDepNonRec($CurrentFlow$)
      if $|CurrentFlow'| = |CurrentFlow|$ **then** ▷ No feasible clustering
        $nbInfeasibleClustering++$
      else
        $nbInfeasibleClustering = 0$
        $\mathcal{F}^{dep}[idxFlow] \leftarrow CurrentFlow'$
      end if
      $idxFlow++$
      $CurrentFlow \leftarrow \mathcal{F}[idxFlow \mod |\mathcal{F}^{dep}|]$
    end while
  end ifreturn $\mathcal{F}^{ind} \cup \mathcal{F}^{dep}$

4.3.5.3. Experimental results

For generating task utilization in multiprocessor, we use the *RandFixedSum* method [ESD10]. The other parameters of the task set are generated as in previous experiments. Figure 4.11 shows the number of clusters obtained for increasing processor utilizations. The precedence probability is set to 0.25 and we show results for a number of levels equal to 0.4 and 0.6 multiplied by the number of tasks. Each point corresponds to the average for 100 generated task sets. Experiments were performed with EDF, using the "Offset Analysis" test of [PL05]. As expected, the number of clusters grows with the processor utilization. The clustering is however significantly worse than in uniprocessor. One reason is that the minimal number of clusters is likely to be higher in multiprocessor: it can be up to the number of different periods multiplied by the number of processors for unfavorable partitionings. Another reason is that the partitioning generates some flows with very high utilization or density, making the clustering significantly harder. We can also observe in this figure that the number of levels has a limited impact on the efficiency of the task clustering.

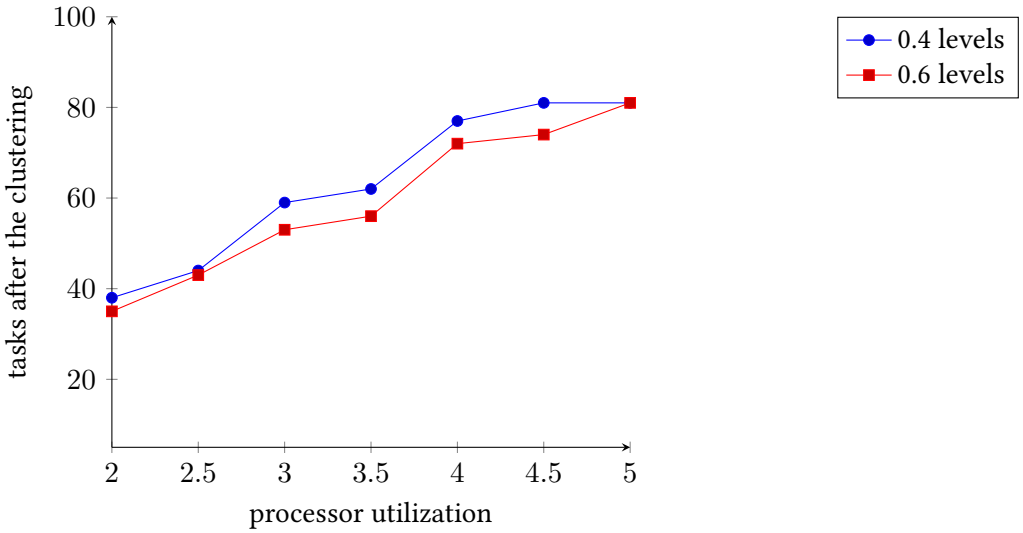


Figure 4.11.: Impact of processor utilization on the task clustering (n = 200, p = 0.25, d_{min} = 0, d_{max} = 1)

4.3.6. Related works

Before our publications The most closely related works are found in the domain of embedded systems design. For instance, [Mzi+13] regroup tasks to reduce the required number of task priorities, [KCH00] regroup tasks that share resources, [SKW00; KWS03] regroup tasks to reduce communications. *Runnable-to-task mapping* is identified as a key step of the development in AUTOSAR and is studied in [ZG11; ZDN12; Woz+13; FLN13]. However, these works do not consider the impact of the clustering on schedulability.

After our publications Runnable-to-task mapping in AUTOSAR remains an active research domain. Most notably, minimizing the number of clusters while preserving schedulability has been studied in [Bou+15; KCC20], where clustering of tasks with different periods is allowed.

4.3.7. Conclusion

The main highlights concerning this work are listed below:

- A proper definition of the task clustering problem and a first heuristic for the case with a uniprocessor and independent tasks was published at the SAC'14 conference [BFO14b];
- Several improvements to the heuristic were then proposed in a publication at the RTNS'14 conference [BFO14a];
- This work started during the internship of Antoine Bertout (2012/3-2012/8);
- This work was the main topic of the Ph.D. thesis of Antoine Bertout (10/2012-10/2015). The thesis was advised by Richard Olejnik, tutored by myself, and co-funded by the Nord-Pas-de-Calais region and by the LIFL laboratory (now CRIStAL);
- Antoine Bertout is now Associate Professor at the University of Poitiers, since 2017.

5. Low-level timing analysis

In this chapter, I present my work on WCET analysis, which consists of two main contributions. First, we proposed a new parametric WCET analysis. While standard WCET analysis produces a constant upper-bound to the WCET, parametric WCET analysis produces a formula, where parameters of the formula are parameters of the program (e.g. loop bounds). Second, we proposed an abstract interpretation procedure that analyses assembly code and establishes linear relations between data locations (memory values and register values) accessed by the program. This enables to produce more accurate estimations of loop bounds than previous works, and thus to produce tighter WCET estimates.

5.1. Symbolic Worst-Case Execution Time analysis

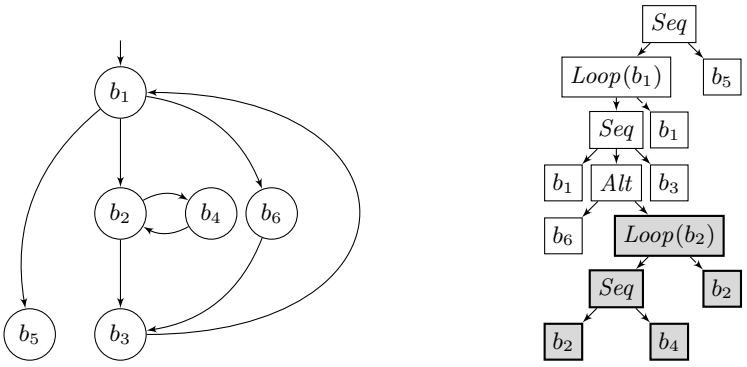
With traditional WCET analysis, if any program parameter that has an effect on the WCET (e.g. loop bounds) is changed, it is necessary to re-run the analysis. Thus, it is difficult to analyze the impact of different parameter values on the final WCET estimate. For instance, the developer may want to know the impact of the number of iterations of a certain loop on the WCET, the impact of the cache size, etc. To answer these questions, it would be necessary to run the analysis several times with different parameter values, which could be a very time consuming process.

An alternative approach is to calculate directly a *parametric WCET* formula instead of a constant value. If the parameter changes, it is possible to recompute the WCET by simply substituting the parameter value into the formula. Thus, it is possible to quickly explore the parameters space. Furthermore, parametric WCET simplifies the analysis process when third-party software is involved, since the developer can provide a parametric WCET along with the component, that can be adapted to the target system.

In addition, if the obtained formula is simple enough, it can be used to efficiently implement an *adaptive* real-time system. We can compute off-line a WCET formula that depends on dynamic parameters and instantiate this formula on-line, at which point parameter values become known. As a result, with low overhead, we can obtain a tighter estimate of the task WCET and take better scheduling decisions.

Finally, large execution time values may happen only very rarely, for instance for unlikely combinations of input data. By using parametric WCET analysis, it is possible to design the system according to an upper bound that is safe for the vast majority of executions of the system, and then evaluate a parametric WCET formula at run-time to trigger an alternate, less time-consuming computation when the formula returns a value exceeding the safe bound (and thus remain under the safe bound). Such an adaptive behaviour can for instance be specified in Prelude using the multi-mode extensions proposed by Frédéric Fort.

The following sections present our approach to parametric WCET analysis, which is based on *symbolic computation.*



(a) CFG representation (b) CFT representation

Figure 5.1.: A program with two nested loops.

5.1.1. Control Flow Tree

A popular approach for performing WCET analysis is to apply the Implicit Path Enumeration Technique (IPET) [LMW95] on the Control Flow Graph of the program. IPET encodes WCET computation as an Integer Linear Programming (ILP) problem that is then solved with standard ILP-solving techniques. Instead, we represent a program as a tree and perform the WCET analysis by a recursive analysis of that tree. The main benefit of a tree-based approach is that recursive WCET analysis is more amenable to symbolic computation than IPET.

In our approach, we represent a program as a *Control Flow Tree* (CFT), instead of a CFG. A CFT also represents the possible execution paths of a program, but with a tree structure instead of a graph. As an example, Figure 5.1b shows the tree corresponding to the CFG of Figure 5.1a. The set of Control-flow Trees $\mathcal{T}$ is defined inductively as follows:

Definition 5.1.1. Let $n, m \in \mathbb{N}^*$, $t_1, \ldots, t_n, \in \mathcal{T}^n$. A control-flow tree $t \in \mathcal{T}$ is one of:

- $Leaf(b)$, which represents the execution of basic block b;
- $Alt(t_1, \ldots, t_n)$, which represents an alternative between the execution of trees $t_1, \ldots, t_n$;
- $Seq(t_1, \ldots, t_n)$, which represents a sequential execution of trees $t_1, \ldots, t_n$;
- $Loop(h, t_1, n, t_2)$, which represents a loop with *header* h, that repeats the execution of its *loop body* t_1, with a maximum number of iterations n, and exits from the loop executing the tree t_2.

In [BFL17], we proposed a procedure to translate a CFG into a CFT, which works for any CFG without irreducible loops (i.e. loops with multiples entries). Any execution path in the input CFG is also an execution path in the corresponding Control-flow Tree produced by this procedure. However, some paths that are valid in the tree may not be valid in the CFG, therefore, the two representations are not equivalent. Still, the translation is *safe*, since the presence of additional paths in the CFT can only lead to an *over-approximation* of the WCET. Experiments show that the over-approximation is very small in practice (see Section 5.1.4).

We make a few additional definitions. First, we let $\mathsf{time}(b)$ denote the WCET of basic block b. Its computation is out of the scope of our work, and relies on OTAWA [Bal+10] in our experiments. Second, we define a lattice on the loops of a CFT as follows:

- We let l_h denote the loop corresponding to the tree whose root is the loop node with header h;
- We let L_t denote the set of loops of CFT t;
- We say that loop l_h *contains* loop $l_{h'}$ and denote $l_{h'} \sqsubseteq l_h$ iff $l_{h'}$ is a sub-tree of l_h;
- $\top$ is a loop such that for all loop l_h, $l_h \sqsubseteq \top$. In other words, $\top$ is a fictive loop whose body is the whole CFT;
- $\bot$ is such that for all l_h, $\bot \sqsubseteq l_h$. In other words, $\bot$ is a fictive empty loop;
- $(L_t \cup \{\top, \bot\}, \sqsubseteq)$ is the loop lattice;
- $l_1 \sqcup l_2$ denotes the least upper of l_1 and l_2;
- $l_1 \sqcap l_2$ denotes the greatest lower bound of l_1 and l_2.

5.1.2. Context-sensitive execution time

The main drawback of a tree-based WCET analysis, is that it lacks means to incorporate the results of auxiliary software and hardware analyses (e.g. cache analysis). In order to cope with this drawback, we enrich the control-flow tree with *context annotations* designed to represent the result of extra-CFT analyses, that will help us reduce the pessimism in WCET estimation.

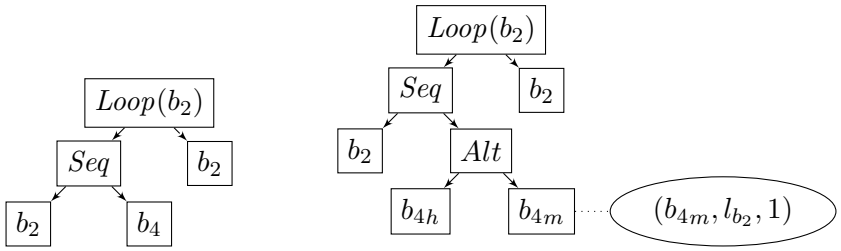
5.1.2.1. Context annotations

A *context annotation*, attached to a CFT node, constrains the conditions under which a sub-tree can be executed. In this work, annotations only represent constraints related to loops. Note that with IPET-based approaches, this information would be represented as an ILP constraint. We will detail the role of context annotations in parametric WCET in Section 5.1.3.

Definition 5.1.2. An annotation on tree t is denoted $\mathsf{ann}(t, l, m)$, where l contains t, and m is a positive integer. This annotation represents the following constraint: m is the maximum number of times t can be executed each time l is entered.

We motivate the need to represent context-sensitive information on two examples. First, let us consider a triangular loop: a *for* loop $i = 1..10$, containing an inner *for* loop $j = i..10$. The maximum iteration count for each loop considered separately is 10, but the inner loop body can be executed at most $\sum_{i=1}^{10} i$ times. Knowing this information will enable us to produce a tighter WCET estimation. To model this example, we have a $\mathit{Leaf}(b)$ node representing the block inside the inner loop. This node has an annotation $(\mathit{Leaf}(b), l_{outer}, 55)$ where l_{outer} represents the outer loop. This annotation represents the fact that, due to the triangular loop, the block b can be executed at most $\sum_{j=1}^{10} j = 55$ times in a complete execution of l_{outer}.

As a second example, we consider the instruction cache analysis by categorization (see e.g. [Fer+99]). In this approach, blocks can be categorized as *persistent* with respect to a loop, meaning that the block will stay in the cache during the whole execution of the loop (only the first execution results in a cache miss). For instance, in the control-flow tree of Figure 5.2a, let us assume that the block corresponding to $\mathit{Leaf}(b_4)$ is persistent. For every complete execution of loop l_{b_2}, b_4 can only cause a cache miss once. Thus the execution time of b_4 must account



(a) Tree before annotation (b) Tree after annotation

Figure 5.2.: Context annotations

for the cache miss only once per complete execution of loop l_{b_2}. To model this example, we proceed in two steps. First, we modify the CFT by splitting the block b_4 from Figure 5.2a into two leaves, representing respectively the cache hit and cache miss cases. This is shown in Figure 5.2b: $Leaf(b_{4m})$ corresponds to the miss and $Leaf(b_{4h})$ to the hit. Then, we add an annotation $(b_{4m}, l_{b_2}, 1)$ to represent the fact that b_{4m} can be executed only once per execution of loop l_{b_2}.

5.1.2.2. Abstract WCET

With context annotations, the WCET of a tree whose execution is iterated inside a loop can vary at each iteration. We introduce the concept of *multi-WCET* to represent the set of WCETs associated with a tree node. Multi-WCETs are defined using *multi-sets* (or *bags*), a generalization of sets where multiple instances of the same element are allowed. The number of instances of a given element in a multiset is called its *multiplicity*. A multi-WCET is a multi-set over $\mathbb{N}$, where the smallest element has an implicit infinite multiplicity. To simplify the presentation, in the following the elements of a multi-WCET are assumed to be sorted non-increasingly. We make the following definitions on multi-WCET:

Definition 5.1.3. Let $\mathbb{W}^{\#}$ denote the set of multi-WCET. Let $\eta, \eta' \in \mathbb{W}^{\#}$ and let $n \in \mathbb{N}$. The following notations and operations are defined on multi-WCETs:

- $\eta[n]$, denotes the $(n+1)$-th greatest element of η. For instance, if $\eta = \{4, 3\}$ then $\eta[0] = 4, \eta[1] = \eta[2] = \eta[3] = \cdots = 3$;
- $\eta|_n$ denotes the multi-WCET that contains the n greatest elements of η (i.e. $\eta[0], \ldots, \eta[n-1]$), and zero as its smallest element;
- $\eta \uplus \eta'$ is a modified version of the traditional multi-set sum, which we will denote $\uplus_{trad}$. Like $\uplus_{trad}$, $\uplus$ sums multiplicities. The difference is as follows. Let min_η, $min_{\eta'}$ denote respectively the smallest elements of η and η'. Then, we have: $\eta \uplus \eta' = \eta \uplus_{trad} \eta' \backslash \{k | k \leq max(min_\eta, min_{\eta'})$. So for instance, $\{8, 8, 4\} \uplus \{9, 8, 3, 2\} = \{9, 8, 8, 8, 4\}$;
- $\eta \otimes k$ denotes the multi-WCET for which each member has k times the multiplicity it has in η;
- $\eta'' = \eta \oplus \eta'$ is the multi-WCET such that: $\forall i \in \mathbb{N}, \eta''[i] = \eta[i] + \eta'[i]$.

The *abstract WCET* of a CFT is now defined as follows:

Definition 5.1.4. For any tree t, its abstract WCET is a pair $\alpha = (l, \eta)$, where l is a loop and η is a multi-WCET. The presence of an integer n in η means that the code associated with t may have an execution time n, but only once, each time l is entered.

In our cache example from Figure 5.2b, the abstract WCET computed for the Alt node would be $(l_{b_2}, \{\mathsf{time}(b_{4m}), \mathsf{time}(b_{4h})\})$, meaning that the WCET of that node is $\mathsf{time}(b_{4m})$ for the first iteration of loop l_{b_2} and then it is $\mathsf{time}(b_{4h})$ for all subsequent iterations of the loop. Note that, if we exit and re-enter the loop, the WCET of the Alt node will again be $\mathsf{time}(b_{4m})$, then $\mathsf{time}(b_{4h})$, $\mathsf{time}(b_{4h})$, etc.

The abstract WCET $\omega(t)$ of a CFT t can be computed inductively as follows:

- $\omega(\mathit{Leaf}(b)) = (\top, \{\mathsf{time}(b)\})$
- $\omega(\mathsf{ann}(t, l_2, n)) = (l_1 \sqcap l_2, \eta_1|_n)$
  where $(l_1, \eta_1) = \omega(t)$.
- $\omega(\mathit{Alt}(t_1, \ldots, t_n)) = (l_1 \sqcap \cdots \sqcap l_n, \eta_1 \uplus \cdots \uplus \eta_n)$
  where $(l_1, \eta_1) = \omega(t_1)$, …, $(l_n, \eta_n) = \omega(t_n)$;
- $\omega(\mathit{Seq}(t_1, \ldots, t_n)) = (l_1 \sqcap \cdots \sqcap l_n, \eta_1 \oplus \cdots \oplus \eta_n)$
  where $(l_1, \eta_1) = \omega(t_1)$, …, $(l_n, \eta_n) = \omega(t_n)$;
- $\omega(\mathit{Loop}(h, t_1, n, t_2) = \begin{cases} (l_2, (\{\sum_{i=0}^{n-1} \eta_1[i]\}) \oplus \eta_2) & \text{if } l_h \equiv l_1 \\ (l_1 \sqcap l_2, \eta \oplus \eta_2) & \text{otherwise} \end{cases}$
  where $(l_1, \eta_1) = \omega(t_1)$ and $(l_2, \eta_2) = \omega(t_2)$ and $\eta[i] = \sum_{j=i \cdot n}^{i \cdot n + n - 1} \eta_1[j]$;

Example 5.1.1. Let us consider an Alt node with two children t_1 and t_2, such that $\omega(t_1) = (l, \{5, 4, 2, 1\})$ and $\omega(t_2) = (l, \{6, 2\})$. The WCET for the Alt node is 6 (from t_2), the second WCET is 5 (from t_1), then 4, and so on. As such, we compute the abstract WCET for the Alt node by taking the union of the multi-WCET components of the two children abstract WCET. Therefore, in our example, $\omega(t) = (l, \{6, 5, 4, 2\})$.

Example 5.1.2. Let us consider a Seq node with two children t_1 and t_2, such that $\omega(t_1) = (l, \{5, 4\})$ and $\omega(t_2) = (l, \{2, 1\})$. The WCET of the Seq node is $5 + 2 = 7$, the second WCET is $4 + 1 = 5$. As such, we compute the abstract WCET for the Seq node by adding elements of the same ranks. In the example, $\omega(t) = (l, \{7, 5\})$.

Example 5.1.3. Let us consider a node $\mathit{Loop}(h, t_1, n, t_2)$, with $\omega(t_1) = (l_h, \{5, 4, 3\})$ (case $l_h \equiv l_1$), $n = 2$, and let t_2 be empty. Then the WCET for one execution of the loop is always $5 + 4 = 9$ (the sum of the n first ranks of the multi-WCET) and we have $\omega(t) = (\top, \{9\})$.

Example 5.1.4. Let us consider a node $\mathit{Loop}(h, t_1, n, t_2)$, with $\omega(t_1) = (l_1, \{5, 4, 3, 2\})$ (case $l_h \not\equiv l_1$), $n = 2$, and let t_2 be empty. Then the WCET of the loop is $5 + 4 = 9$ (the sum of the first n ranks of the multi-set), while the second WCET is $3 + 2 = 5$ (the sum of the subsequent n ranks of the multi-set), and subsequent WCETs are always $2 + 2 = 4$. Therefore, $\omega(t) = (l_1, \{9, 5, 4\})$.

The soundness of the CFT based WCET analysis is stated by the following theorem. MakeCFT corresponds to the procedure that translates a CFG into a CFT. $\mathsf{gpaths}(G)$ denotes the set of all possible execution paths of CFG G. We overload notation $\mathsf{time}(.)$ and let $\mathsf{time}(p)$ denote the total WCET of the execution path p.

Theorem 5.1.1. *Let G be a CFG. Let $t = \mathsf{MakeCFT}(G)$. Let $(l, \eta) = \omega(t)$. We have:*

$$\forall p \in \mathsf{gpaths}(G), \mathsf{time}(p) \leq \eta[0]$$

5.1.3. Symbolic computation

In this section we study the problem of computing the abstract WCET of a tree when some parameters of the tree are unknown. We show that, using simple syntactic sugaring, our definition of $\omega(t)$ produces formulae akin to arithmetic expressions. Then we rely on existing work on symbolic computation of arithmetic expressions to simplify abstract WCET formulae. The simplification step is mainly useful in case of on-line formula instantiation, to reduce the memory and execution time overhead.

First, we introduce several operators on abstract WCET, which act as syntactic sugar, to be able to express WCET computation as arithmetic computation.

Definition 5.1.5. Let t_1 and t_2 be two control-flow trees. We define a set of operations on abstract WCET such that:

$$\begin{aligned}
\omega(t_1) \oplus \omega(t_2) &= \omega(Seq(t_1, t_2)) \\
\omega(t_1) \uplus \omega(t_2) &= \omega(Alt(t_1, t_2)) \\
(\omega(t_1), \omega(t_2), h)^n &= \omega(Loop(h, t_1, n, t_2)) \\
\omega(t_1)_{\downarrow(h,n)} &= \omega(\mathsf{ann}(t_1, l_h, n)) \\
n \odot (l, \eta) &= (l, \eta \otimes n) \\
k^\infty &= \{k\}
\end{aligned}$$

Furthermore, we let $\theta \equiv (\top, 0^\infty)$. We define the following grammar to represent the set of formulae $\mathcal{W}$ corresponding to the computation of the abstract WCET of a control-flow tree ($w \in \mathcal{W}$):

$$\begin{array}{lcl}
w & ::= & const \mid id \mid w_{\downarrow(h,it)} \mid w \oplus w \mid w \uplus w \mid (w, w, b)^{it} \\
h & ::= & b \mid id \\
it & ::= & i \mid id
\end{array}$$

The simplest formula is a literal abstract WCET value ($const \in (L_G \times W^\#)$). A formula can also be a variable corresponding to an unknown WCET value (id). A formula can also be the sum ($w \oplus w$), the product ($w \uplus w$) or the repetition of two formulae ($(w, w, b)^{it}$). Finally, a formula can also consist of the application of an annotation to a formula ($w_{\downarrow(h,it)}$). The factor of a repetition, and the factor of an annotation (it), can either be a constant integer value (i) or a variable (id). The loop header of an annotation (h) can either be a basic block name (b) or a variable (id).

Several elements of these formulae can be symbolic values (denoted by id), i.e. variable parameters: symbolic WCET value (w), symbolic loop iteration bound (it), symbolic loop header

(h). When symbolic values appear in a WCET formula, we cannot reduce the formula to a literal abstract WCET value. However, in many cases the formula can be transformed into a simpler, yet equivalent formula. For instance, we have: $(x \oplus 2 \odot x) \oplus 3 \odot x \oplus y = 6 \odot x \oplus y$. Figure 5.3 lists all the rewriting rules we use in order to simplify WCET formulae. Most of the rules are transpositions of integer arithmetic simplification rules [Coh02] to the case of WCET formulae. We make the following comments:

- We rely on an order relation $\lhd$ on formulae, so as to ensure that the commutativity rules are only applied in one direction for two given formulae. Classically, the order relation is defined based on the syntactic structure of the formulae (see e.g. [Coh02] for details);
- Distributivity is applied in reverse order and only to factor constant terms;
- Concerning the annotation rewriting rule, the strategy consists in reducing the number of annotation applications;
- Concerning the loop rule, since we have no rule for combining loops, we only extract the loop exit tree from the loop;
- Combination of constant formulae is not detailed here but is applied as well. For instance, $(l, 2^\infty) \oplus (l, 3^\infty)$ is simplified to $(l, 5^\infty)$.

Associativity.

$$(w_1 \oplus w_2) \oplus w_3 \mapsto w_1 \oplus w_2 \oplus w_3 \quad (5.1)$$
$$w_1 \oplus (w_2 \oplus w_3) \mapsto w_1 \oplus w_2 \oplus w_3 \quad (5.2)$$
$$(w_1 \uplus w_2) \uplus_3 \mapsto w_1 \uplus w_2 \uplus w_3 \quad (5.3)$$
$$w_1 \uplus (w_2 \uplus w_3) \mapsto w_1 \uplus w_2 \uplus w_3 \quad (5.4)$$

Commutativity.

$$(w_1 \oplus w_2) \mapsto (w_2 \oplus w_1) \text{ if } w_2 \lhd w_1 \quad (5.5)$$
$$(w_1 \uplus w_2) \mapsto (w_2 \uplus w_1) \text{ if } w_2 \lhd w_1 \quad (5.6)$$

Distributivity.

$$(cst_1 \oplus w_3) \uplus (cst_2 \oplus w_3) \mapsto (cst_1 \uplus cst_2) \oplus w_3 \quad (5.7)$$

Neutral element.

$$w_1 \oplus \theta \mapsto w_1 \quad (5.8)$$
$$w_1 \uplus \theta \mapsto w_1 \quad (5.9)$$

Multiplication.

$$0 \odot w_1 \mapsto \theta \quad (5.10)$$
$$(k_i \odot w_1) \oplus w_1 \mapsto (k_i + 1) \odot w_1 \quad (5.11)$$

Annotation.

$$\theta_{\downarrow(h,it)} \mapsto \theta \quad (5.12)$$
$$w_{1\downarrow(h,it)} \oplus w_{2\downarrow(h,it)} \mapsto (w_1 \oplus w_2)_{\downarrow(h,it)} \quad (5.13)$$

Loop.

$$(w_1, w_2, b)^{it} \mapsto (w_1, \theta, b)^{it} \oplus w_2 \quad (5.14)$$

Figure 5.3.: Abstract WCET formula rewriting rules

5.1.4. Experiments

Our symbolic WCET analysis has been implemented in the WSymb tool, available online as open-source[1]. It is implemented as a plugin to the OTAWA WCET analysis tool [Bal+10]. In this

[1] https://gitlab.cristal.univ-lille.fr/otawa-plugins/WSymb

Bench	*Source*	*Parameter*	*Algorithm*	*Function*
matmult	ML	Matrix size	Matrix multiplication	*Initialize (twice)*
cnt	ML	Matrix size	Matrix sum	*Sum*
fft	TB	Number of samples	FFT	*main*
compress	ML	Data size	Data compression	*main*
lift	TB	Number of sensors	Factory lift control	*main*
adpcm	ML	Trigo. computation steps	ADPCM encoding	*main*
aes_enc	TB	Data size	AES encryption	*main*
powerwindow	TB	Sensor data input size	Car window control	*main*
fbw	PB	Task activation count	fly-by-wire	*main*
audiobeam	TB	Audio source count	Audio beamforming	*main*
mpeg2	TB	Video resolution	MPEG2 decoding	*main*

Table 5.1.: Benchmarks summary

section, we report the results of our experiments with WSymb. The benchmarks we selected for our experiments are summarized in Table 5.1. For each benchmark, we mention its source (ML for Mälardalen[2], TB for TACleBench[3], or PB for PapaBench[4]), provide a short description of the kind of algorithm it performs, and specify the function whose WCET is analyzed. We only introduce one parameter per benchmark because precision is independent of the number of parameters in our approach. The analyses have been executed on a PC with an Intel core i5 3470 at 3.2 Ghz, with 8 Gb of RAM. Every benchmark has been compiled with ARM crosstool-NG 1.20.0 (gcc version 4.9.1) with -O1 optimization level.

The results of our experiments are shown in Table 5.2. First, we detail the size of the WCET formulae computed by our approach. Column *CFG* shows the number of basic blocks in the CFG. Column *Initial* shows the size (the number of operands) of the WCET formula before simplification, while Column *Final* shows the formula size after simplification. In most cases, the size of the non-simplified formula, which also corresponds to the size of the CFT, is close to the size of the CFG. Differences are due to the presence of structure-breaking instructions (such as *goto*, *break*, *continue*, *return* in the middle of a function). This is especially true for the *mpeg2*, and to a lesser extent for *lift*, *audiobeam*, and *fbw* benchmarks. For all benchmarks, the size of the simplified formula is very small and is related to the number of loops whose iteration count depends on the parameter.

Then, we compare our approach with an IPET approach. Comparison is performed according to two criteria: WCET analysis time, and pessimism of the resulting WCET. The target hardware is an ARM processor with a set-associative LRU instruction cache (the data cache is not taken into account). The processor pipeline is analyzed with the exegraph method [RS09]. The instruction cache is analyzed using the cache categorization of [Fer+99], and its impact is represented in the CFT using annotations as explained in Section 5.1.2.1. The target instruction cache used in the analysis has 64 Kbytes, 16 ways, and blocks of 16 bytes. The instruction cache miss latency was assumed to be 10 cycles. Each benchmark is analyzed as a standalone task executed on baremetal. To perform the preliminary steps of the WCET analysis (program path analysis, CFG building, loop bounds estimation, pipeline and cache modeling), we rely on

[2] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[3] https://github.com/tacle/tacle-bench
[4] https://github.com/stefanct/papabench

		Formula size		*Time (ms)*			*Pessimism (%)*			
Bench	*CFG*	*Initial*	*Final*	*Common*	*WSymb*	*ILP*	*WSymb*	*Min*	*Max*	*MPA*
matmult	111	130	5	1105	1	0	0.01	0.00	3.88	0.31
cnt	153	284	3	2278	2	8	0.15	0.00	3.59	30.4
fft	391	453	8	2968	4	16	0.00	0.00	1.51	-
compress	694	906	3	4760	11	40	0.02	0.01	0.03	-
lift	814	1799	5	5130	19	40	1.51	0.05	2.29	-
adpcm	2032	2211	3	10688	67	272	0.01	0.01	0.33	-
aes_enc	2205	2651	2	4914	30	260	0.04	0.03	0.04	-
powerwindow	3738	4453	24	45702	224	4192	0.01	0.01	1.43	-
fbw	10612	27251	2	36940	1198	8960	2.62	0.03	7.05	-
audiobeam	12299	47248	37	56566	1222	12824	0.12	0.00	0.49	-
mpeg2	38612	1658109	3	267332	12221	$>$ 1 week	-	-	-	-

Table 5.2.: Benchmarking results

OTAWA (version 1.0). These steps are common to the IPET approach and to our approach. For the remaining steps, in the case of the IPET approach, we use GNU lp_solve ILP solver[5]. To compare the WCET estimates, we instantiate our WCET formula by assigning to the parameter the constant value used in the IPET experiment.

The *Common* column represents the time spent by OTAWA for the preliminary steps (common to IPET and our approach), while the *WSymb* and *ILP* columns correspond to the time spent for the remaining steps. The WCET evaluation time is essentially linear in the size of the CFT in our approach and noticeably lower than the evaluation time for the IPET approach. Notice that lp_solve did not find a solution for mpeg2 after one week of execution time. Furthermore, let us emphasize that computing the WCET for different parameter values with the IPET approach requires to run the whole analysis (*Common+ILP*) for each parameter value, while we only need to do the analysis (*Common+WSymb*) once and then instantiate the formula for each parameter value.

WCET pessimism is measured in comparison with the IPET result. The *WSymb* column represents the value of the pessimism with our approach for a fixed value of the parameter (the same value as the one used for the IPET approach). The *Min* and *Max* columns represent respectively the minimum pessimism and maximum pessimism (in percentage) for varying values of the parameter between 1 and 1000. We observed that, in general, the percentage of pessimism decreases with the value of the parameter, approximately with an hyperbolic shape. The pessimism of our approach is much lower than that of the MPA approach (results extracted from [BEL09] are reported in column MPA). It is also extremely low compared to the IPET approach. Pessimism in our approach can be attributed to the following causes: (1) the reduced expressiveness of our CFT annotations (as opposed to ILP constraints) and (2) paths existing in the CFT but not in the CFG. Experiments show that the amount of pessimism does not depend on the size of the CFG.

5.1.5. Related works

Before our publications Several parametric/symbolic WCET analyses have been proposed before. Source code analyses have been proposed in [Viv+01; Moh+05; Cof+07; Moh+11]. One limitation of source code analysis is the need to account for complex compiler optimizations

[5] https://sourceforge.net/projects/lpsolve/

that may change the structure of the Control-Flow Graph. As a consequence, source code analyses usually make conservative simplifying hypotheses about the compiler behaviour, which result in a more pessimistic WCET. Binary-level analyses have also been proposed, based on parametric ILP in [Alt+08], or based on a non-IPET ad-hoc approach in [BEL11]. Further works have focused either on reducing the complexity of the analysis or on improving its tightness [AAN11a; AAN11b; Če+15]. Tree-based parametric WCET analysis has been considered in [BB00; CB02], from which our work is inspired. However, in that approach the WCET of a tree node does not depend on its execution context, resulting in potentially high overapproximation.

In comparison to these works, our approach is the only one that is simultaneously scalable, because the complexity of the formula production is polynomial, and tight, thanks to context annotations.

After our publications Recently, our symbolic WCET computation method was applied in [Búr+21] to enable the on-line computation of the WCET of query-based monitors, which is highly dependent on the program inputs.

5.1.6. Conclusion

The main highlights concerning this work are listed below:

- This work was published in the TECS journal (2017) [BFL17];
- The symbolic WCET analysis has been implemented in the WSymb tool, available online as open-source[6];
- This work was pivotal in the ANR PRCE Corteva project[7] (2018-2022, leader CRIStAL);
- The ANR JCJC Sywext project[8] (2020-2023, leader Clément Ballabriga) is a prolongation of this work;
- The Ph.D. thesis of Sandro Grebant (10/2020-) is a prolongation of this work. The thesis is co-advised by Giuseppe Lipari and myself, tutored by Clément Ballabriga, and funded by Sywext.

5.2. Relational abstract interpretation of assembly code

Parameters of the WCET formulae produced by our symbolic WCET analysis are often related to one another. For instance, the loop bound of an inner loop can depend on the loop bound of an outer loop. We would be interested in establishing such properties. To this intent, we developed a static analysis of assembly code based on abstract interpretation using a polyhedra-based abstract domain. The analysis is capable of automatically inferring linear relations between values used in an assembly program. Although our original motivation was the need to enhance existing WCET analyses, and in particular to improve the computation of upper bounds on the

[6] https://gitlab.cristal.univ-lille.fr/otawa-plugins/WSymb
[7] https://corteva.cristal.univ-lille.fr/
[8] https://sywext.cristal.univ-lille.fr/

number of iterations of loops, our abstract domain has other potential applications as well, such as buffer-overflow analysis or infeasible paths analysis for instance.

Most analyses by abstract interpretation proposed in the literature are performed on *source code*. Instead, we analyze *assembly code*. Assembly code analysis offers high fidelity reasoning about the software behaviour. In comparison, source code analysis requires to make assumptions about the (complex) compiler behaviour. Combined with assembly code reconstruction from binary code (e.g. using OTAWA [Bal+10]), assembly code analysis enables the analysis of software whose source code is unavailable, such as third-party libraries, legacy programs, or malware. A wide range of applications can benefit from these advantages, including Worst-Case Execution Time analysis, reverse engineering, binary code rewriting, binary code reuse, vulnerability detection, and more [CL16]. However, one major challenge is that, while analysis of source code can reason on program *variables*, this information is unavailable in the assembly code. Instead, assembly code analysis must reason at a lower level of abstraction, analysing properties on *data-locations*, that is to say registers and memory addresses. We propose to identify the subset of registers and memory locations to be represented in the abstract state as the analysis progresses. This representation enables us to design a relational abstract interpretration procedure for binary code.

5.2.1. Motivating example

As a motivating example, we present a snippet of C code, inspired from packet processing network drivers in Figure 5.4[9]. We consider C code to ease the comprehension, however let us remind that our analysis is performed on assembly code. The `send_request` function sends a request in some application-layer protocol that runs over UDP/IP. Lines 12-13 build a packet composed of a variable-length IP header, a fixed-length UDP header, and a variable-length UDP payload (some operations on IP or UDP fields have been omitted). Note that the starting address of the UDP header depends on the size of the IP header (`h1->hdr_len`). At line 17, we call the function responsible for putting the payload into the packet. At line 18, the packet is sent using the `send_packet` function, which belongs to the lower-level network layer API. This function does not take the packet size as parameter, since it can be deduced from the header: in lines 2-3, the function parses the packet to obtain the UDP payload size, and the UDP checksum is computed by iterating over the payload.

To automatically compute a bound on the number of iterations of the loop at line 4, the analysis has to discover that `udp_l` equals `udp_size` (due to line 16). This can be done with an appropriate use of a *relational abstract domain*. However, very few of the existing analyses running on binary code use a relational domain, and to the best of our knowledge, none support relations between addresses that are not know statically (`udp_l`, `udp_size`). Let us emphasize that such a use of pointers and memory buffers is typical of many embedded systems: for instance in network packet processing, but also in many device drivers.

5.2.2. Target language

To simplify the presentation, we consider programs of a minimalist assembly language that we call `MEMP` (our implementation actually supports the more complex ARM A32 instruction set). `MEMP` makes the following simplifying assumptions: all data locations have the same size, memory accesses are aligned to the word size and are values in $\mathbb{Z}$, there are no integer overflows,

[9] The original bench listing is available here: https://pastebin.com/C5UPYRx3

```
void send_packet(char *buf) {
   int iphdr_l = ((struct ip*)buf)->hdr_len;
   int udp_l = ((struct udp*)(buf + iphdr_l))->len;
   for (int i = 0; i < udp_l; i++) { /* do CRC */ }
   ethernet_write(buf);
}

void send_request(int iphdr_size, int udp_size) {
   char buf[1024];
   if ((iphdr_size >= 20) && (iphdr_size <= 60) &&
       (udp_size >= 4) && (udp_size <= 100)) {
      struct ip *h1 = buf;
      struct udp *h2 = buf + iphdr_size;

      h1->hdr_len = iphdr_size;
      h2->len = udp_size;
      fill_packet_payload(buf);
      send_packet(buf);
   }
}
```

Figure 5.4.: Network-inspired benchmark

and function calls are inlined (these limitations could be lifted using for instance [BLH12; SP78]). Its complete syntax is detailed in Figure 5.5. A *concrete program state* is a pair $(\mathcal{R}, *)$, where $\mathcal{R}$ maps registers to their content and $*$ maps addresses to their content.

Programs	$::=$	$l_1 : I_1, l_2 : I_2, \ldots, l_n :$ `END`
Labels	$::=$	$\{l_1, l_2, \ldots\}$
Registers	$::=$	$\{r_1, r_2, \ldots\}$
Constants	$::=$	$\{c_1, c_2, \ldots\}$
Instructions	$::=$	
$r_1 \leftarrow \mathrm{OP}^c(r_2, r_3)$	$\mid$	`OP r1 r2 r3`
$r \leftarrow c$	$\mid$	`SET r c`
Emulate undefined r	$\mid$	`RAND r`
$r_1 \leftarrow *(r_2)$	$\mid$	`LOAD r1 r2`
$*(r_1) \leftarrow r_2$	$\mid$	`STORE r1 r2`
Branch to l if $r = 0$	$\mid$	`BR r l`
Halt	$\mid$	`END`

Figure 5.5.: Syntax of `MEMP`

5.2.3. The POLYMAP domain

We introduce the POLYMAP abstract domain to represent (an over-approximation of) the set of possible concrete states at an execution step of a program of `MEMP`. POLYMAP relies on

Polyhedra, although it could be adapted to rely to a different relational domain, e.g. Octagons, to reduce complexity, as long as that domain supports equality constraints between two variables of the abstract domain.

Let $|S|$ denote the cardinality of set S. Let C_n denote the set of linear constraints in $\mathbb{Z}^n$ on a set of n variables taken in some set $\mathcal{V}$. We denote $\langle c_1, c_2, \ldots, c_m \rangle$ the polyhedron p consisting of all the vectors in $\mathbb{Z}^n$ that satisfy constraints c_1, c_2, ..., c_m (with $c_i \in C_n$, $1 \leq i \leq m$). We denote $dim(p) = n$ the number of dimensions of p. In the following, the term *variable* implicitly refers to polyhedron variables (a.k.a the domain dimensions). This should not be confused with source code variables, which are never considered. Instead, we analyse the contents of *data-locations*, that is to say of registers and memory locations accessed by the program. We denote:

- $\mathcal{P}$ the set of polyhedra;
- $s \in p$ when s (with $s \in \mathbb{Z}^{dim(p)}$) satisfies the constraints of polyhedron p;
- $p \sqsubseteq_\diamond p'$ iff $\forall s \in p, s \in p'$;
- $p'' = p \sqcup_\diamond p'$ the *convex hull* of p and p';
- $p'' = p \sqcap_\diamond p'$ the union of the constraints of p and p';
- $vars(p)$ the set of variables of p, where $|vars(p)| = dim(p)$ by definition;
- $proj(p, x_1 \ldots x_k)$ the projection of p on space $x_1 \ldots x_k$, with $k < |dim(p)|$;
- $cyl(p, x)$ the cylindrification of p by x, as defined in [Mon76] (which basically removes x from the constraints of p);
- $p[x_i/x_j]$ the substitution of variable x_j by x_i in p, which first applies $cyl(p, x_i)$ and then substitutes x_i for x_j in the remaining polyhedra constraints;
- We say that "c holds for p" when $p \sqsubseteq_\diamond \langle c \rangle$.

An abstract state in POLYMAP is a triple $(p, \mathcal{R}^\sharp, *^\sharp)$. The polyhedron p specifies the constraints on the variables of the abstract state. The *register mapping* $\mathcal{R}^\sharp$ maps registers to variables. We have $\mathcal{R}^\sharp(r) = v$ iff variable v represents the value of register r in p. The *address mapping* $*^\sharp$ maps address variables to content variables. We have $*^\sharp(x_1) = x_2$ iff variable x_2 represents the value at the memory address represented by variable x_1. These mappings evolve during the analysis, because the polyhedra variables are not known when starting the analysis. Instead they are created/removed as the analysis progresses, and so associations are added/removed from the mappings.

Example 5.2.1. Consider the following abstract state of POLYMAP:

$$(\langle x_2 = x_0, x_3 = x_1, x_0 = 4, x_1 \geq 5 \rangle, \{r_0 : x_0, r_1 : x_1\}, \{x_2 : x_3\})$$

Registers r_0, r_1, are respectively mapped to variables x_0, x_1. The content of the address represented by x_2 is represented by x_3. Polyhedra constraints state that memory address 4 ($x_2 = x_0 = 4$) contains a value greater than 5 ($x_3 = x_1 \geq 5$).

The *concretization function* γ defines the set of concrete states represented by an abstract state. Intuitively, a concrete state belongs to the concretization of an abstract state iff the values of its registers and memory respect the constraints of the abstract state (see [Bal+19] for a formal definition).

Example 5.2.2.

$$\begin{aligned} a &= (\{1 \leq x_1 \leq 2, x_2 = x_1, x_3 = 1\}, \{r_0 : x_1\}, \{x_2 : x_3\}) \\ \gamma(a) &= \{(\{r_0 = 1\}, \{*(1) = 1\}\}) \\ &\quad (\{r_0 = 2\}, \{*(2) = 1\})\} \end{aligned}$$

5.2.4. Abstract interpretation

Our analysis proceeds by forward abstract interpretation [CC77], adapted to the analysis of programs of `MEMP`. In order to concisely define abstract state transformers we use $(p', [r_i : x_i], [x_j : x_k])(\cdot)$ as a shorthand for $\lambda(p, \mathcal{R}^\sharp, *^\sharp).(p \sqcap_\diamond p', \mathcal{R}^\sharp[r_i : x_i], *^\sharp[x_j : x_k])$, and denote "$-$" when a state component remains unchanged. $\mathcal{R}^\sharp[r_i : x_i]$ associates r_i to x_i in $\mathcal{R}^\sharp$, replacing the previous association of r_i, if any. $*^\sharp[x_i : x_k]$ behaves similarly. Whenever an unbound polyhedron variable appears in the lambda body, we implicitly assume that it is a fresh variable, that has never been used before during the analysis.

The complete interpretation procedure is described in Algorithm 5. It applies to a program P of `MEMP`. During the interpretation, we keep a subset L of labels of interest. Abstract values are stored in a map M from labels to abstract values. We assume that loop header labels L_W of P have previously been identified using an existing analysis (e.g. Tarjan's algorithm [Tar72]). Figure 5.6 reports a running example, that will be used as illustration throughout the rest of the section. Procedures to compute the join ($\sqcup$), widening ($\triangledown$), and the transfer function $(I)^\sharp$ of instruction I, dictate how to compute the abstract state at some program label based on abstract states computed at other program labels. They are detailed in the remainder of this section.

```
1: RAND r0
2: RAND r7
3: SET r1 4
4: SET r2 5
5: ADD r3 r0 r1
6: STORE r3 r1
7: SUB r5 r7 r1
8: BR r5 10
9: STORE r3 r2
10: LOAD r6 r3
11: END
```

Label	*Polyhedron*	*Registers*	*Memory*
5	$p_1 = \langle x_1 = 4, x_2 = 5\rangle$	$\mathcal{R}_1^\sharp = \{r_0 : x_0, r_1 : x_1, r_2 : x_2, r_7 : x_7\}$	
6	$p_2 = p_1 \sqcap_\diamond \langle x_3 = x_0 + x_1\rangle$	$\mathcal{R}_2^\sharp = \mathcal{R}_1^\sharp[r_3 : x_3]$	
7	$p_3 = p_2 \sqcap_\diamond \langle x_4 = x_3, x_5 = x_1\rangle$	$\mathcal{R}_2^\sharp$	$*_1^\sharp = \{x_4 : x_5\}$
8	$p_4 = p_3 \sqcap_\diamond \langle x_8 = x_7 - x_1\rangle$	$\mathcal{R}_3^\sharp = \mathcal{R}_2^\sharp[r_5 : x_8]$	$*_1^\sharp$
10 (from 9)	$p_5 = p_4 \sqcap_\diamond \langle x_9 = x_2\rangle$	$\mathcal{R}_3^\sharp$	$*_2^\sharp = \{x_4 : x_9\}$
10′ (from 8)	$p_6 = p_4 \sqcap_\diamond \langle x_8 = 0\rangle$	$\mathcal{R}_3^\sharp$	$*_1^\sharp$
$unify(10, 10')$	$p_7 = p_6[x_9/x_5]$	$\mathcal{R}_3^\sharp$	$*_3^\sharp = \{x_4 : x_9\}$
$10 \sqcup 10'$	$p_8 = p_2 \sqcap_\diamond \langle x_4 = x_3, x_8 = x_7 - x_1, x_1 \leq x_9 \leq x_2\rangle$	$\mathcal{R}_3^\sharp$	$*_3^\sharp$
11	$p_8 \sqcap_\diamond \langle x_{10} = x_9\rangle$	$\mathcal{R}_3^\sharp[r_6 : x_{10}]$	$*_3^\sharp$

Figure 5.6.: Running example of analysis

Algorithm 5 INTERPRET(P)

```
1:  procedure UPDATE(ℓ, a, L)                          ▷ Auxiliary procedure
2:      a ← antialias(a)
3:      if ℓ ∈ L_W then                                ▷ Check if l is a loop header
4:          new ← M[l] ▽(M[l] ⊔ a)
5:      else
6:          new ← M[ℓ] ⊔ a
7:      end if
8:      if new ⋢ M[ℓ] then                             ▷ Abstract value for ℓ changed, propagate
9:          M[ℓ] ← new; L ← L ∪ ℓ
10:     end if
11: end procedure
12:
13: for all (ℓ, I) ∈ P do                              ▷ Start of main procedure
14:     M[ℓ] ← ⊥                                       ▷ Begin with empty abstract states
15: end for
16: M[ℓ_1] ← ⊤; L ← {ℓ_1}                              ▷ Program starting label
17: while L ≠ ∅ do                                     ▷ Fixpoint iteration
18:     Pick and remove ℓ from L
19:     match P[ℓ]
20:         with BR r ℓ'
21:             UPDATE(ℓ', (⟨r = 0⟩, −, −)(M[ℓ]), L)    ▷ Branching case
22:             UPDATE(ℓ + 1, (−, −, −)(M[ℓ]), L)       ▷ Not branching case
23:         with END
24:             skip
25:         with _
26:             UPDATE(ℓ + 1, ((P[ℓ])♯)(M[ℓ]), L)       ▷ Abstract semantics of I
27: end while
28: return M
```

The following theorem states that the interpretation procedure is sound, in the sense that it always computes abstract states that over-approximate the set of possible concrete states. Relation $\xrightarrow{c}{}^*$ represents the state transitions allowed by the concrete program semantics.

Theorem 5.2.1. *Let P be a MEMP program. Let $M = Interpret(P)$. Then, for any concrete state s_{init}:*

$$(P \vdash (l_1, s_{init}) \xrightarrow{c}{}^* (\ell, s)) \Longrightarrow (s \in \gamma(M[\ell]))$$

5.2.4.1. Aliasing

Aliases play an important role in the analysis. In a general sense, aliasing occurs in a program when a data location can be accessed through several symbolic names. In our case, we define the aliasing relation between two variables x_1 and x_2 of a polyhedron p as follows:

- Cannot alias: whenever $\langle x_1 = x_2 \rangle \cap p = \emptyset$;
- May alias: whenever $\langle x_1 = x_2 \rangle \cap p \neq \emptyset$;
- Must alias, denoted $x_1 \equiv x_2$: whenever $p \sqsubseteq_\diamond \langle x_1 = x_2 \rangle$.

In the following, we assume that abstract states are *alias free* so as to simplify the analysis (see [Bal+19] for details on how to enforce this assumption). Still, testing aliasing relations is required to compute several abstract state transformers.

5.2.4.2. Non-memory instructions

The transfer functions for non-memory instructions are defined below:

- A binary operation (`OP`) binds the target register (r_1) to a variable (x) constrained to be equal to the combination of the variables bound to the operand registers ($\mathcal{R}^\sharp(r_2)$, $\mathcal{R}^\sharp(r_3)$). If the constraint cannot be expressed as a linear relation, x remains unconstrained;
- Concerning the branching instruction, the branching condition holds at the target label (l). Its negation cannot be represented as a linear relation so it is ignored;
- Transfer functions for $(SET)^\sharp$ and $(RAND)^\sharp$ are straightforward.

$$(\texttt{OP r}_1\ \texttt{r}_2\ \texttt{r}_3)^\sharp = \begin{cases} (\langle x = \mathrm{OP}^c(\mathcal{R}^\sharp(r_2), \mathcal{R}^\sharp(r_3))\rangle,\ [r_1 : x],\ -)(\cdot) & \text{if } linear(\mathrm{OP}^c) \\ (-,\ [r_1 : x],\ -)(\cdot) & \text{otherwise} \end{cases}$$

$$(\texttt{BR r l})^\sharp = \begin{cases} (\langle \mathcal{R}^\sharp(r) = 0\rangle,\ -,\ -)(\cdot) & \text{at } l \\ (-,-,\ -)(\cdot) & \text{at current label+1} \end{cases}$$

$$(\texttt{SET r}_1\ \texttt{c})^\sharp = (\langle x = c\rangle,\ [r_1 : x], -)(\cdot)$$

$$(\texttt{RAND r}_1)^\sharp = (-,\ [r_1 : x], -)(\cdot)$$

Example 5.2.3. In Figure 5.6, at label 6 (i.e. the label immediately following the `ADD` operation) we introduce the constraint $x_3 = x_0 + x_1$ and the register mapping $\mathcal{R}_1^\sharp(r_3) = x_3$.

Example 5.2.4. In Figure 5.6, at label 10, when coming from label 8 (i.e. from `BR r5 10`), we add the constraint $x_8 = 0$.

5.2.4.3. Memory instructions

Let us now consider the `LOAD` instruction. If the input state contains a memory address variable that is equivalent to the load address (note that for alias free states, if such a variable exists, it is unique), then in the output state the value of the destination register is the value of the memory value mapped to this address. Otherwise, the value of the destination register is undefined:

$$(\texttt{LOAD r}_1\ \texttt{r}_2)^\sharp = \begin{cases} (\langle x = *^\sharp(a)\rangle,\ [r_1 : x],\ -)(\cdot) & \text{if } a \equiv r_2 \\ (-,\ [r_1 : x],\ -)(\cdot) & \text{otherwise} \end{cases}$$

Example 5.2.5. In Figure 5.6, at label 10 we have $x_4 \equiv r_3$ and $*^\sharp(x_4) = x_9$, so at label 11 we introduce the constraint $x_{10} = x_9$ and the mapping $\mathcal{R}_3^\sharp[r_6] = x_{10}$.

Let us now consider the `STORE` instruction. If there exists an address variable equivalent to the target register, then there already exists a memory mapping for this address. The previous content at this address is replaced by the content of the source register (see *Replace* below).

Otherwise, we create a new memory mapping (see $Create$ below). An alias free state contains at most one address variable that must-alias with r_1. It may however contain several may-alias address variables a'. For each such a', this means that a' either equals r_1, which requires a $Replace$, or is different from r_1, which has no impact. We apply operator $\sqcup$ on both cases to manage this uncertainty, and add the constraints for each may-alias address (see May below).

$$(\texttt{STORE r}_1\ \texttt{r}_2)^\sharp = \begin{cases} \lambda s.Replace(a)(May(s)) & \text{if } \exists a \in vars_A(p), a \equiv r_1 \\ \lambda s.Create(May(s)) & \text{otherwise} \end{cases}$$

With ($\bigcirc$ denotes function composition):

$$\begin{aligned} Replace(a) &= (\langle x = \mathcal{R}^\sharp(r_2)\rangle,\ -,\ [a : x])(\cdot) \\ Create &= (\langle x_i = \mathcal{R}^\sharp(r_1), x_j = \mathcal{R}^\sharp(r_2)\rangle, -, [x_i : x_j])(\cdot) \\ May &= \bigcirc_{\{a \in A \mid a \text{ may-alias } r_1\}} \lambda s.(Replace(a)(s) \sqcup s) \end{aligned}$$

Example 5.2.6. In Figure 5.6, at label 7, we create a new memory mapping $*_1^\sharp(x_4) = x_5$ and we introduce the constraints $x_4 = x_3$, $x_5 = x_1$.

Example 5.2.7. In Figure 5.6, at label 10, when coming from label 9, we replace a previous mapping, x_4 is mapped to x_9 (instead of x_5 previously), and we introduce the constraint $x_9 = x_2$.

5.2.4.4. Abstract domain operators

We just detailed how to compute the impact of the execution of an instruction on an abstract state. Now, we define procedures to merge two abstract states. The *join* operator $\sqcup$ is used to handle branching: we join the abstract states corresponding to two different program paths leading to the same label. The *widening* operator $\triangledown$ is used on loop headers to ensure that the analysis reaches a fixpoint, despite the presence of loops.

The correspondence between polyhedra variables and data locations is neither predefined nor fixed. Therefore, a specificity of our analysis is that it may happen that two abstract states use different variables to designate the same data location. To enable a more accurate comparison of these two states, we must *unify* them first, which consists in trying to assign the same variables in the two states to the same data locations. Unification is used for inclusion testing, and also in the join and widening operators.

Unification The unification procedure is detailed in Algorithm 6. It replaces address variables and address content variables of $s_2^\sharp$ by their equivalent in $s_1^\sharp$ and does the same for register variables. Function $matchVar(v_1, v_2, p_1, p_2)$ is a heuristic that returns true if variable v_1 of p_1 is equivalent to variable v_2 of p_2[10].

Example 5.2.8. In Figure 5.6, when computing $unify(10, 10')$, s_1 corresponds to the state of 10 and s_2 to the state of $10'$. Trivially, $matchVar$ detects that x_4 represents the same address in both states. Since $*_2^\sharp(x_4) = x_9$ (in s_1) and $*_1^\sharp(x_4) = x_5$ (in s_2), we replace x_5 by x_9 in s_2.

[10] Basically, the heuristic tries to express v_1 and v_2 as linear expressions of some variables common to p_1 and p_2.

Algorithm 6 *unify*$(s_1^\sharp = (p_1, \mathcal{R}_1^\sharp, *_1^\sharp), s_2^\sharp = (p_2, \mathcal{R}_2^\sharp, *_2^\sharp))$

$(p_2', \mathcal{R}_2^{\sharp\prime}, *_2^{\sharp\prime}) \leftarrow (p_2, \mathcal{R}_2^\sharp, *_2^\sharp)$
for all $(x_1, x_2) \in Dom(*_1^\sharp) \times Dom(*_2^\sharp)$ **do**
  if $matchVar(x_1, x_2, p_1, p_2)$ **then**
    Replace x_2 by x_1 and $*_2^\sharp(x_2)$ by $*_1^\sharp(x1)$ in $(p_2', \mathcal{R}_2^{\sharp\prime}, *_2^\sharp)$
  end if
end for
for all $r \in Dom(\mathcal{R}_1^\sharp) \cap Dom(\mathcal{R}_2^\sharp)$ **do**
  Replace $\mathcal{R}_2^\sharp(r)$ by $\mathcal{R}_1^\sharp(r)$ in $(p_2', \mathcal{R}_2^{\sharp\prime}, *_2^{\sharp\prime})$
end for
return $(p_2', \mathcal{R}_2^{\sharp\prime}, *_2^{\sharp\prime})$

Join The join procedure is described in Algorithm 7. Function *comLoc* filters the register and address mappings of two states, to keep only locations that are mapped to the same variable in both states. The join procedure first unifies the two states to join, then joins the two polyhedra, and finally filters common data locations.

Algorithm 7 $(p_1, \mathcal{R}_1^\sharp, *_1^\sharp) \sqcup (p_2, \mathcal{R}_2^\sharp, *_2^\sharp)$

$(p_2', \mathcal{R}_2^{\sharp\prime}, *_2^{\sharp\prime}) = unify((p_1, \mathcal{R}_1^\sharp, *_1^\sharp), (p_2, \mathcal{R}_2^\sharp, *_2^\sharp))$
$p \leftarrow p_1 \sqcup_\diamond p_2'$
$(\mathcal{R}^\sharp, *^\sharp) \leftarrow comLoc((\mathcal{R}_1^\sharp, *_1^\sharp), (\mathcal{R}_2^{\sharp\prime}, *_2^{\sharp\prime}))$
return $(p, \mathcal{R}^\sharp,)$

Example 5.2.9. In Figure 5.6, when computing $10 \sqcup 10'$, we obtain identical register and memory mappings for 10 and $unify(10, 10')$. The convex hull $p_5 \sqcup_\diamond p_7$ groups the constraints on x_9 ($x_1 \leq x_9 \leq x_2$) and lifts those on x_8.

Widening The widening procedure is defined just like $\sqcup$, except that we use a polyhedra widening operator $\triangledown_\diamond$ (e.g. that of [GR06]) in place of $\sqcup_\diamond$.

Inclusion Finally, to determine when the analysis reaches a fix-point, we must test abstract states inclusion: the fix-point is reached when, for all program labels, the analysis computes an abstract state that is included in the abstract state computed previously at that label. Let $s_1^\sharp = (p_1, \mathcal{R}^\sharp{}_1, *^\sharp{}_1)$ and $s_2^\sharp = (p_2, \mathcal{R}^\sharp{}_2, *^\sharp{}_2)$. The inclusion operator $\sqsubseteq^\sharp$ is defined as follows:

$$s_1^\sharp \sqsubseteq s_2^\sharp \Leftrightarrow p_1' \sqsubseteq_\diamond p_2 \wedge \mathcal{R}_2^\sharp \subseteq \mathcal{R}_1^{\sharp\prime} \wedge *_2^\sharp \subseteq *_1^{\sharp\prime}$$
$$\text{with } (p_1{}', \mathcal{R}_1^{\sharp\prime}, *^{\sharp\prime}) = unify(a_2, a_1)$$

5.2.5. Experiments

Our analysis is implemented in the Polymalys tool, available online as open-source[11]. To illustrate the benefits of our approach, we compute loop bounds on a set of assembly programs. We compare the results obtained by Polymalys with state-of-the-art loop bound analysis tools, namely SWEET [Lis14], PAGAI [HMM12] and oRange [BMS08]. To compute loop bounds with Polymalys, for each loop header label we create a "virtual" register. We instrument the program so that the register is set to 0 when entering the loop, and incremented at each loop iteration. Then, the loop bound is the maximum possible value of that register.

We illustrate the differences between tool capabilities on some synthetic program examples below (more extensive experiments are available in [Bal+19]). To ease the comprehension, we provide the C source code.

Example 5.2.10. The following example contains pointer aliasing and pointer arithmetic:

```
foo() {
  int i, bound = 10;
  int *ptr = &bound;
  ptr++; ptr--; *ptr = 15; k = 0;
  for (i = 0; i < bound; i++);
}
```

PAGAI does not find the loop bound (the loop is considered unbounded), because it does not infer that `ptr` = `&bound` when executing the instruction `*ptr=15`. Other tools bound the loop correctly (15 iterations).

Example 5.2.11. The following example contains an off-by-one array access:

```
1 #define SIZE 10
2 foo(int offset) {
3   int i, bound = 10;
4   int tab[SIZE];
5   if ((offset > SIZE) || (offset < 0))
6     return -1;
7   tab[offset] = 100;
8   for (i = 0; i < bound; i++);
9 }
```

The off-by-one error (lines 5-6) may cause the array cell assignment (line 7) to overwrite the `bound` variable with the value 100. Polymalys correctly detects that the loop may iterate 100 times, while oRange and SWEET detect a maximum of 10 iterations. PAGAI also bounds to 10 iterations, but warns about a possible undefined behavior and unsafe result. Note that the bound depends on the stack variable allocation layout, which is unknown when analysing the source code. In our experiments, the compiler allocates the `bound` variable next to the array.

Example 5.2.12. The following example shows the benefits of a relational domain:

```
1 #define MAXSIZE 10
2 foo() {
```

[11] https://gitlab.cristal.univ-lille.fr/otawa-plugins/polymalys

```
3     int base, end, i;
4     if (end - base > MAXSIZE)
5       end = base + MAXSIZE;
6     for (i = base; i < end; i++);
7   }
```

Here, we do not know statically the value of `end` and `base`. However, due to the `if` statement (line 4), Polymalys introduces the constraint $\texttt{end} - \texttt{base} \leq 10$. Thus, Polymalys bounds the loop correctly (10 iterations), while PAGAI, oRange and SWEET do not.

Example 5.2.13. Finally, we report analysis results for the motivating example of Figure 5.4. Polymalys correctly infers that the loop bound is equal to the maximum size of the UDP payload. PAGAI, oRange and SWEET fail to provide any bound.

5.2.6. Related works

Our experiments established that in many cases Polymalys is capable of more tightly bounding the number of loop iterations than existing tools (SWEET [Lis14], PAGAI [HMM12] and oRange [BMS08]).

Computing loop bounds is only one possible application of this abstract interpretation analysis. The main contribution of our work is actually to automatically discover memory locations of interest, and to track their contents. To this regard, *Value Set Analysis* (VSA) [BR04] is the closest related work. VSA is integrated in many tools for analysing binary programs, such as CodeSurfer [Bal+05], angr [Sho+16] BAP [Bru+11] and Jakstab [KV10]. VSA is also based on abstract interpretation of assembly code, but the abstract domain is non-relational. As a consequence, only constant addresses (or constant offsets to the data-stack) can be identified as memory locations of interest. Our analysis identifies a wider range of memory locations of interest and thus infers more properties on the assembly code.

5.2.7. Conclusion

The main highlights concerning this work are listed below:

- This work was published at the VMCAI'19 conference [Bal+19], and received the *best paper award* of the conference;
- An extension to efficiently handle arrays was published in the FMSD journal (2022) [BFR22];
- The abstract interpretation procedure was implemented in the Polymalys tool, available as open-source[12];
- This work was done in collaboration with colleagues from CRIStAL (Clément Ballabriga, Giuseppe Lipari), and from the University of Lyon (Laure Gonnord);
- The post-doctoral study of Jordy Ruiz (6/2018-6/2019) was a follow-up to this work;
- The internship of Guillaume Person (5/2021-8/2021) was a follow-up to this work;
- The Ph.D. thesis of Sandro Grebant (10/2020-), already mentionned previously, partly relies on this work;

[12] https://gitlab.cristal.univ-lille.fr/otawa-plugins/polymalys

- The Ph.D. thesis of Andrei Florea (10/2022-) is a continuation of this work. The thesis is advised by Vlad Rusu, tutored by Clément Ballabriga and myself, and jointly funded by the Haut-de-France region and the University of Lille.

Part III.

Perspectives

6. Conclusion

In the previous chapters I presented my work on the programming and analysis of critical real-time systems, carried out since my appointment in 2010 at the University of Lille. My contributions stem from the development of the Prelude language and its compiler, and extend over several research domains. This chapter summarizes my main contributions and presents my future research projects.

6.1. Summary

The development of real-time systems involves a wide range of research areas. An important part of my work has focused on the connections between these areas.

Chapter 3 presented my work and collaborations on the programming of real-time systems with Prelude. First, the code generation process of the Prelude compiler was extended to support multi-core architectures with distributed memory [Pag+18b; FF19]. Second, the Prelude language was extended to support communications that may or may not be synchronized [Wys+12]. Third, support for multi-mode real-time systems based on mode automata was presented [FF22].

Chapter 4 presented my work and collaborations on high-level timing analysis. First, we proposed scheduling policies and associated schedulability tests for the analysis of a set of real-time tasks with precedence constraints [For+10; For+11]. Second, we proposed a general method for the analysis of end-to-end timing constraints, that is to say constraints that involve a chain of tasks [Wys+13; FBP17]. Third, we studied the task clustering problem, which consists in reducing the number of tasks used to implement a system while preserving the schedulability of the system [BFO14b; BFO14a].

Chapter 5 presented my work and collaborations on low-level timing analysis. First, we proposed a parametric Worst-Case Execution Time analysis based on symbolic computation [BFL17]. Second, we defined a relational abstract interpretation technique for the analysis of assembly code [Bal+19; BFR22].

6.2. Future research projects

In future works, I will build on my expertise on the programming of real-time systems to develop new research directions. In this section I detail one short term (1-3 years), one medium term (2-4 years), and one long term (4+ years) future research project.

6.2.1. Modular WCET analysis (short term)

In our symbolic WCET computation approach (Section 5.1), elements to be considered as symbolic values must be identified *manually* by the programmer. In the Ph.D thesis of Sandro Grebant (started 10/2020) our objective is to extend the analysis so that it *automatically* identifies the arguments of a procedure as parameters and produces a WCET formula that represents the

WCET of the procedure as a function of its arguments. Intuitively, the formula accounts for the impact of arguments on the control-flow (conditionnal statements, loops) of the procedure, and thus on its WCET. This extensions relies on both our work on symbolic WCET computation and on abstract interpretation of binary code (Section 5.2). First, we are extending the abstract interpreter to infer conditions of conditionnal statements, and loop bounds, that depend on procedure arguments. Then, we are extending the symbolic WCET computation to incorporate this information in the CFT and in the WCET formulae. For instance, we aim at producing formulae such as $(((\texttt{r0} \geq 11) \circledast \omega(B)) \uplus ((\texttt{r0} \leq 10) \circledast (l, \{5\}))$, which states that the WCET of a procedure is that of basic block B when its first argument (`r0`) is greater than 11, otherwise it is equal to 5.

Our next step will be to derive a *modular WCET analysis*, a feature that no existing WCET analysis approach currently provides. For instance, let us assume that procedure f calls procedure g. To compute the formula of f, we will instantiate the WCET formula of g with the arguments values provided by f. This splits the problem of computing the WCET of a complete program into separate sub-problems, one for each procedure. This will improve the scalability of the analysis, because the sub-problems can be solved independently, and also because we can factorise the analysis of a procedure called several times in the same program.

To achieve this objective, we will extend the CFT formalism and WCET formulae to represent procedure calls. Considering our current progress, we hope to complete this part during the Ph.D thesis of Sandro Grebant. We will also extend our abstract interpretation of binary code to make it compositional. This requires to analyse and represent the impact of a procedure call on the abstract state of the program. This should be fairly simple for pure functions, i.e. functions without side-effects, since they only impact the caller through return values. We could for instance adapt the symbolic relational separate analysis proposed in [CC02] to the case of binary code. The analysis of impure functions is more difficult, because they can modify arbitrary memory addresses in the caller stack and also in memory addresses of global variables. This work will be the main topic of the Ph.D. thesis of Andrei Florea (started 10/2022).

6.2.2. Synthesis of Prelude programs (medium term)

As mentionned in our work on partial delays specification (Section 3.2), designing a completely deterministic real-time system can be difficult. Instead, we can opt for an initial non-deterministic specification, which is simpler to devise and avoids overspecification, and delegate the generation of a deterministic program to the compiler. This can be seen as a form of *program synthesis*.

In future works, I will consider program synthesis in the following setting. The non-deterministic input model will consist of a set of communicating nodes subject to real-time constraints. A minimum and maximum period will be provided for each node[1] and end-to-end constraints will be specified for functionnal chains (such as those of Section 4.2). When generating the output program, the compiler will automatically insert delays, choose communication patterns and periods, so that the program is causal and satisfies end-to-end constraints.

I will build on related works to achieve this objective. First, to formalize the non-deterministic semantics of the initial specification, I will collaborate with colleagues from CRIStAL (David Nowak and Vlad Rusu). We will rely on their recent work on co-induction [RN22], which is well-adapted to formalize semantics that apply on sets of flows (non-determinism induces several possible resulting flows for the same computation). Second, for the compilation step I

[1] These minimum and maximum values were discussed in the Overview (Chapter 1).

plan to collaborate with colleagues from Inria Paris/ENS (Timothy Bourke, Marc Pouzet), who recently worked on similar synthesis problems for a Lustre-like language, for partial delays specification e.g. [Ioo+20], and task chain synthesis (to-be-published). Synthesis of real-time attributes for functional chains has also been studied in the real-time litterature [GHS94; Li+13; Bec+16], although less extensively than the real-time analysis of functional chains.

6.2.3. Formally verified real-time programs (long term)

The high-level of abstraction of Prelude simplifies the development of large scale real-time systems, by abstracting from low-level implementation concerns, which are delegated to the compiler. Compilation transforms the Prelude program into a set of concurrent communicating real-time tasks, to be executed by a real-time operating system. Because Prelude targets critical systems, we want to ensure that the semantics of the generated multi-task program is consistent with the semantics of the input Prelude program.

While parts of the compilation of Prelude are defined formally on paper, providing a full correctness proof that relates the source and target semantic models is still an open problem. In future works, I would like to devise a complete formally verified framework for the development of real-time systems. The core of this project will consist in devising a Prelude compiler verified formally with the Coq Interactive Theorem Prover [The22]. Related works on Vélus [Bou+17], a verified Lustre compiler, will serve as a starting point. However, compared to Prelude, the source semantics of Vélus does not contain information on production dates, and the target semantics is single-threaded instead of multi-threaded. The framework will also include a formally verified real-time operating system (RTOS), as Prelude generates multi-task code. I will collaborate with colleagues from CRIStAL (2XS team) who developed the Pip Protokernel[2XS22], a minimalist kernel proved correct in Coq that includes an EDF scheduler [Van+22]. We will rely on the CompCert compiler [Ler09] to ensure the correctnes of the compilation from C to machine code. As a result, this will provide a complete proof that ensures that the program executed by the RTOS behaves as specified by the semantics of the corresponding Prelude program.

As an additional objective, I would like to study the automatic verification of temporal properties on Prelude programs. The model-checker Lesar [Ray08] has a similar objective, but it only deals with logical-time, not real-time. In particular, I would like to be able to check constraints on time-domain performance indicators of control-command systems, such as for instance the settling time[2] of a controller in response to a command. Such properties require to model the program along with the physical environment it interacts with. Developping a verification tool with such capabilities would combine nicely with the compilation chain and RTOS described previously, as properties proved on the Prelude program would provably be preserved by the corresponding embedded code.

[2]The time required to settle close to the steady-state value [Pag+14].

Part IV.

Appendices

A. Main symbols and acronyms

Programming with Prelude

$/\hat{}$ Periodic flow undersampling. 13

$/_{.}$ Periodic clock deceleration. 15

$\pi(n, p)$ The period of clock (n, p). 15

$\rightarrow_{.}$ Clock phase offset. 15

$\varphi(n, p)$ The phase of clock (n, p). 15

$*_{.}$ Periodic clock acceleration. 15

$*\hat{}$ Periodic flow oversampling. 13

$\mathcal{M}_G$ The global shared processor memory. 18

$\mathcal{M}_i$ The private scratchpad memory of core i. 18

A_i The acquisition phase of τ_i. 18

E_i The execution phase of τ_i. 18

R_i The restitution phase of τ_i. 18

ρ_i The core i. 18

AER Acquisition Execution Restitution. 17

PREM PRedictable Execution Model. 17

SPM Scratchpad memory. 18

SSM Synchronous State Machines. 28

UAV Unmanned Aerial Vehicle. 28

High-level timing analysis

C_i The worst-case execution time of τ_i. 37

D_i^* The relative deadline of τ_i, adjusted to encode precedence constraints. 41

D_i The relative deadline of τ_i. 37

$M_{i,j}$ The precedence matrix from τ_i to τ_j. 38

O_i^* The first release date of τ_i, adjusted to encode precedence constraints. 41

O_i The first release date of τ_i. 37

T_i The period of τ_i. 37

$d_{i.k}$ The absolute deadline of $\tau_{i.k}$. 37

$etime(\tau_{i.p})$ The earliest completion time of job $\tau_{i.p}$. 46

$\tau_{i.k}$ The k^{th} repetition, or job, of τ_i. 37

$ltime(\tau_{i.p})$ The latest completion time of job $\tau_{i.p}$. 46

$\mathcal{J}$ The job set. 37

$\mathcal{S}$ The task set. 37

$\tau_{i.rlv(p)}$ The p^{th} relevant job of τ_i. 46

$\tau_{j.first(rlv(p))}$ The first job of τ_j that depends on $\tau_{i.rlv(p)}$. 46

$\tau_{j.last(rlv(p))}$ The last job of τ_j that depends on $\tau_{i.rlv(p)}$. 46

$o_{i.k}$ The release date of $\tau_{i.k}$. 37

$preds(\tau_i)$ The predecessors of τ_i. 38

Φ A priority assignment. 39

$\mathbb{N}_{<n}$ The set of natural integers strictly smaller than n. 38

$succs(\tau_i)$ The successors of τ_i. 38

τ_i The task of index i. 37

$\tau_i \rightarrow \tau_j$ A precedence constraint from τ_i to τ_j. 38

$e_{\mathcal{S}\Phi}(\tau_{i.k})$ The completion time of $\tau_{i.k}$ in the schedule produced by Φ. 39

$s_{\mathcal{S}\Phi}(\tau_{i.k})$ The start time of $\tau_{i.k}$ in the schedule produced by Φ. 39

DM Deadline Monotonic. 40

EDF Earliest-Deadline First. 40

lcm Least Common Multiple. 38

RM Rate Monotonic. 40

WCET Worst-Case Execution Time. 37

Low-level timing analysis

$(\mathcal{R}, *)$ Concrete state with registers $\mathcal{R}$ and memory $*$. 74

$(p, \mathcal{R}^\sharp, *^\sharp)$ Abstract state with polyhedron p, registers $\mathcal{R}^\sharp$ and memory $*^\sharp$. 76

$Alt(t_1, \ldots, t_n)$ An alternative between the execution of CFTs t_1, ..., t_n. 64

$Leaf(b)$ The CFT for basic block b. 64

$Loop(h, t_1, n, t_2)$ A loop CFT with header h, that repeats t_1, n times, and exits executing t_2. 64

$Seq(t_1, \ldots, t_n)$ A sequential execution of CFTs t_1, ..., t_n. 64

γ The concretization function. 76

$\langle c_1, c_2, \ldots, c_m \rangle$ The polyhedron for constraints $c_1, c_2, \ldots, c_m$. 76

$\eta \oplus \eta'$ The point-wise sum of η and η'. 67

$\eta \otimes k$ Mulplies the multiplicities in η by k. 67

$\eta \uplus \eta'$ The multi-set-style sum of multi-WCET η and η'. 67

$\eta|_n$ The n greatest elements of η. 67

$\sqcup_\diamond$ Convex hull (polyhedra join). 76

$\sqsubseteq_\diamond$ Polyhedra inclusion. 76

$\sqcap_\diamond$ Polyhedra intersection (union of polyhedra constraints). 76

$\mathbb{W}^\#$ The set of all multi-WCET. 67

$*^\sharp[x_i : x_k]$ Associates x_k to x_i in $*^\sharp$. 77

$\omega(t)$ The abstract WCET of CFT t. 67

$proj(p, x_1 \ldots x_k)$ The projection of p on space $x_1 \ldots x_k$. 76

$\mathcal{R}^\sharp(r)$ The variable for the content of register r. 76

$\mathcal{R}^\sharp[r_i : x_i]$ Associates r_i to x_i in $\mathcal{R}^\sharp$. 77

$(I)^\sharp$ The transfer function of instruction I. 77

$vars(p)$ The variables (dimensions) of polyhedron p. 76

$\mathsf{time}(b)$ The WCET of basic block b. Also denotes the WCET of an execution path. 64

$p[x_i/x_j]$ The substitution of variable x_j by x_i in p. 76

CFG Control-Flow Graph. 64

CFT Control-Flow Tree. 64

ILP Integer Linear Programming. 64

IPET Implicit Path Enumeration Technique. 64

Bibliography

Personal publications

Journal papers

[BFL17] Clément Ballabriga, Julien Forget, and Giuseppe Lipari. "Symbolic WCET Computation". In: *ACM Transactions on Embedded Computing Systems (TECS)* 17.2 (Dec. 2017), pp. 1 –26. DOI: 10.1145/3147413.

[BFR22] Clément Ballabriga, Julien Forget, and Jordy Ruiz. "Relational abstract interpretation of arrays in assembly code". In: *Formal Methods in System Design* (Oct. 2022), pp. 1–32. DOI: 10.1007/s10703-022-00399-3.

Conferences with proceedings

[Bal+19] Clément Ballabriga, Julien Forget, Laure Gonnord, Giuseppe Lipari, and Jordy Ruiz. "Static Analysis Of Binary Code With Memory Indirections Using Polyhedra". In: *VMCAI'19 - International Conference on Verification, Model Checking, and Abstract Interpretation*. Cascais, Portugal, Jan. 2019. DOI: 10.1145/3147413.

[BFO14a] Antoine Bertout, Julien Forget, and Richard Olejnik. "A heuristic to minimize the cardinality of a real-time task set by automated task clustering". In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 2014, pp. 1431–1436. DOI: 10.1145/2554850.2554958.

[BFO14b] Antoine Bertout, Julien Forget, and Richard Olejnik. "Minimizing a real-time task set through task clustering". In: *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. 2014, pp. 23–31. DOI: 10.1145/2659787.2659820.

[Cor+11] Mikel Cordovilla, Frédéric Boniol, Julien Forget, Eric Noulard, and Claire Pagetti. "Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset". In: *19th International Conference on Real-Time and Network Systems*. Nantes, France, Sept. 2011. URL: https://hal.archives-ouvertes.fr/inria-00618587.

[FBP17] Julien Forget, Frédéric Boniol, and Claire Pagetti. "Verifying end-to-end real-time constraints on multi-periodic models". In: *ETFA2017 - 22nd IEEE International Conference on Emerging Technologies And Factory Automation*. Limassol, Cyprus, Sept. 2017. DOI: 10.1109/ETFA.2017.8247612.

[FF19] Frédéric Fort and Julien Forget. "Code generation for multi-phase tasks on a multi-core distributed memory platform". In: *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE. 2019, pp. 1–6. DOI: 10.1109/RTCSA.2019.8864558.

[FF22] Frédéric Fort and Julien Forget. "Synchronous semantics of multi-mode multi-periodic systems". In: *37th ACM/SIGAPP Symposium On Applied Computing (SAC'22)*. Virtual, Apr. 2022, pp. 1248–1257. DOI: 10.1145/3477314.3507271.

[For+10] Julien Forget, Frédéric Boniol, Emmanuel Grolleau, David Lesens, and Claire Pagetti. "Scheduling Dependent Periodic Tasks Without Synchronization Mechanisms". In: *16th IEEE Real-Time and Embedded Technology and Applications Symposium*. Stockholm, Sweden, Apr. 2010. DOI: 10.1109/RTAS.2010.26.

[For+11] Julien Forget, Emmanuel Grolleau, Claire Pagetti, and Pascal Richard. "Dynamic priority scheduling of periodic tasks with extended precedences". In: *ETFA2011*. IEEE. 2011, pp. 1–8. DOI: 10.1109/ETFA.2011.6059015.

[Pag+18b] Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert, and Arno Luppold. "Automated generation of time-predictable executables on multi-core". In: *RTNS 2018*. Proceedings of the 26th International Conference on Real-Time Networks and Systems. POITIERS, France, Oct. 2018. DOI: 10.1145/3273905.3273907.

[Wys+12] Rémy Wyss, Frédéric Boniol, Julien Forget, and Claire Pagetti. "A synchronous language with partial delay specification for real-time systems programming". In: *10th Asian Symposium on Programming Languages and Systems*. Kyoto, Japan, Dec. 2012. DOI: 10.1007/978-3-642-35182-2_16.

[Wys+13] Rémy Wyss, Frédéric Boniol, Claire Pagetti, and Julien Forget. "End-to-end latency computation in a multi-periodic design". In: *28th Symposium On Applied Computing (SAC'13)*. Coimbra, Portugal, Apr. 2013, pp. 1682–1687. DOI: 10.1145/2480362.2480678.

Invited speaker

[For17] Julien Forget. "Ecole Temps Réel 2017 - Uniprocessor real-time scheduling". Doctoral. Lecture. France, Aug. 2017. URL: https://hal.archives-ouvertes.fr/hal-03193898.

Publications by other authors

References

[2XS22] 2XS-CRIStAL. *The Pip Protokernel*. 2022. URL: https://pip.univ-lille.fr/.

[AAN11a] Ernst Althaus, Sebastian Altmeyer, and Rouven Naujoks. "Precise and efficient parametric path analysis". In: *SIGPLAN Not.* 46.5 (Apr. 2011), pp. 141–150. ISSN: 0362-1340. (Visited on 04/26/2022).

[AAN11b] Ernst Althaus, Sebastian Altmeyer, and Rouven Naujoks. "Symbolic Worst Case Execution Times". en. In: *Theoretical Aspects of Computing – ICTAC 2011*. Ed. by Antonio Cerone and Pekka Pihlajasaari. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 25–44. ISBN: 978-3-642-23283-1.

[Abs] AbsInt. *aiT*. Version 22.1. URL: https://www.absint.com/ait/.

[AD94] Rajeev Alur and David L Dill. "A theory of timed automata". In: *Theoretical computer science* 126.2 (1994), pp. 183–235.

[ALGM96] Pascal Aubry, Paul Le Guernic, and Sylvain Machard. "Synchronous distribution of Signal programs". In: *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on,* IEEE. 1996.

[Alt+08] S. Altmeyer, C. Hümbert, B. Lisper, and R. Wilhelm. "Parametric Timing Analysis for Complex Architectures". In: *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Aug. 2008, pp. 367–376.

[AP14a] Ahmed Alhammad and Rodolfo Pellizzoni. "Schedulability analysis of global memory-predictable scheduling". In: *Proceedings of the 14th International Conference on Embedded Software*. ACM. 2014.

[AP14b] Ahmed Alhammad and Rodolfo Pellizzoni. "Time-predictable execution of multithreaded applications on multicore systems". In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2014.

[Aud91] Neil C. Audsley. *Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start Times*. Tech. rep. YCS 164. Dept. Computer Science, University of York, Dec. 1991.

[AWP15] Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. "Memory efficient global scheduling of real-time tasks". In: *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2015.

[Bal+05] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. "CodeSurfer/x86—A platform for analyzing x86 executables". In: *International Conference on Compiler Construction*. 2005.

[Bal+10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. "OTAWA: An Open Toolbox for Adaptive WCET Analysis". In: *Software Technologies for Embedded and Ubiquitous Systems*. Vol. 6399. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010.

[BB00] Guillem Bernat and Alan Burns. "An approach to symbolic worst-case execution time analysis". In: *IFAC Proceedings Volumes* 33.7 (2000), pp. 43–48.

[BB04] E. Bini and G.C. Buttazzo. "Biasing effects in schedulability measures". In: *16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004. Proceedings.* July 2004, pp. 196 –203.

[BB06] S. Baruah and A. Burns. "Sustainable Scheduling Analysis". In: *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International.* 2006, pp. 159–168.

[BBW11] G. Buttazzo, E. Bini, and Yifan Wu. "Partitioning Real-Time Applications Over Multicore Reservations". In: *IEEE Transactions on Industrial Informatics* 7.2 (May 2011), pp. 302–315.

[BDN18] Alessandro Biondi and Marco Di Natale. "Achieving predictable multicore execution of automotive applications using the LET paradigm". In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE. 2018, pp. 240–250.

[Bec+16] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. "Synthesizing Job-Level Dependencies for Automotive Multi-Rate Effect Chains". In: *The 22th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications.* 2016.

[BEL09] S. Bygde, A. Ermedahl, and B. Lisper. "An Efficient Algorithm for Parametric WCET Calculation". In: *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA'09.* Beijing, China: IEEE, 2009, pp. 13–21.

[BEL11] Stefan Bygde, Andreas Ermedahl, and Björn Lisper. "An efficient algorithm for parametric WCET calculation". In: *Journal of Systems Architecture.* Design and Optimization for Embedded and Real-Time Computing Systems and Applications 57.6 (June 2011), pp. 614–624. ISSN: 1383-7621.

[Ben+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. "The synchronous languages 12 years later". In: *Proceedings of the IEEE* 91.1 (2003), pp. 64–83.

[Ber+08] Dominique Bertrand, Anne-Marie Déplanche, Sébastien Faucou, and Olivier H Roux. "A study of the aadl mode change protocol". In: *13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008).* IEEE. 2008, pp. 288–293.

[BLH12] Stefan Bygde, Björn Lisper, and Niklas Holsti. "Fully bounded polyhedral analysis of integers with wrapping". In: *Electronic Notes in Theoretical Computer Science* 288 (2012), pp. 3–13.

[BMS08] Armelle Bonenfant, Marianne de Michiel, and Pascal Sainrat. "oRange: A tool for static loop bound analysis". In: *Workshop on Resource Analysis, University of Hertfordshire, Hatfield, UK.* 2008.

[Bon+08] Frédéric Boniol, Pierre-Emmanuel Hladik, Claire Pagetti, Frédéric Aspro, and Victor Jégu. "A framework for distributing real-time functions". In: *International Conference on Formal Modeling and Analysis of Timed Systems.* 2008, pp. 155–169.

[Bou+15] Rahma Bouaziz, Laurent Lemarchand, Frank Singhoff, Bechir Zalila, and Mohamed Jmaiel. "Architecture exploration of real-time systems based on multi-objective optimization". In: *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE. 2015, pp. 1–10.

[Bou+17] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. "A formally verified compiler for Lustre". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 586–601.

[BR04] Gogul Balakrishnan and Thomas Reps. "Analyzing memory accesses in x86 executables". In: *Compiler Construction*. Springer. 2004, pp. 2732–2733.

[BRH90] Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor". In: *Real-Time Systems* 2.4 (1990-11), pp. 301–324.

[Bru+11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. "BAP: A binary analysis platform". In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 463–469.

[Búr+21] Márton Búr, Kristóf Marussy, Brett H Meyer, and Dániel Varró. "Worst-Case Execution Time Calculation for Query-Based Monitors by Witness Generation". In: *arXiv preprint arXiv:2102.03116* (2021).

[BW01] Alan Burns and Andrew J Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[Cap+17] Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. "SiGAMMA: Server based integrated GPU arbitration mechanism for memory accesses". In: *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. ACM. 2017.

[CB02] A. Colin and G. Bernat. "Scope-tree: a program representation for symbolic worst-case execution time analysis". In: *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*. June 2002, pp. 50–59.

[CC02] Patrick Cousot and Radhia Cousot. "Modular static program analysis". In: *International Conference on Compiler Construction*. Springer. 2002, pp. 159–179.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (PLDI'77)*. ACM. 1977, pp. 238–252.

[CL16] Juan Caballero and Zhiqiang Lin. "Type inference on executables". In: *ACM Computing Surveys (CSUR)* 48.4 (2016).

[Cof+07] Joel Coffman, Christopher Healy, Frank Mueller, and David Whalley. "Generalizing parametric timing analysis". In: *SIGPLAN Not.* 42.7 (June 2007), pp. 152–154. ISSN: 0362-1340. (Visited on 11/18/2020).

[Coh02] J.S. Cohen. *Computer Algebra and Symbolic Computation: Mathematical Methods*. Ak Peters Series vol. 1. Natick, MA, USA: Peters, 2002.

[CP03] Jean-Louis Colaço and Marc Pouzet. "Clocks as First Class Abstract Types". In: *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT'03)*. Vol. 2855. Lecture Notes in Computer Science. Philadelphia, USA, 2003, pp. 134–155.

[CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. "A conservative extension of synchronous data-flow with state machines". In: *Proceedings of the 5th ACM international conference on Embedded software*. 2005, pp. 173–182.

[CSB90] Houssine Chetto, Marilyne Silly, and T. Bouchentouf. "Dynamic Scheduling of Real-Time Tasks under Precedence Constraints". In: *Real-Time Systems* 2 (1990).

[DK21] Jana Dunfield and Neel Krishnaswami. "Bidirectional typing". In: *ACM Computing Surveys (CSUR)* 54.5 (2021), pp. 1–38.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[Dur+14] Guy Durrieu, Madeleine Faugere, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. "Predictable flight management system implementation on a multicore processor". In: *Embedded Real Time Software (ERTS'14)*. 2014.

[Dür+19] Marco Dürr, Georg Von Der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. "End-to-end timing analysis of sporadic cause-effect chains in distributed systems". In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.5s (2019), pp. 1–24.

[ESD10] Paul Emberson, Roger Stafford, and Robert I Davis. "Techniques for the synthesis of multiprocessor tasksets". In: *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*. 2010, pp. 6–11.

[FBM18] Björn O Forsberg, Luca Benini, and Andrea Marongiu. "HePREM: Enabling predictable GPU execution on heterogeneous SoC". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2018).

[Fei+08] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. "A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Paths semantics". In: *Proceedings of the IEEE Real-Time System Symposium Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*. Barcelona, Spain, 2008.

[Fer+99] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. "Cache Behavior Prediction by Abstract Interpretation". In: *Sci. Comput. Program.* 35.2 (1999), pp. 163–189.

[FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. *The Architecture Analysis & Design Language (AADL): An Introduction*. Tech. rep. CMU/SEI-2006-TN-011. Software Engineering Institute, Carnegie Mellon University, 2006.

[FLN13] H.R. Faragardi, B. Lisper, and T. Nolte. "Towards a communication-efficient mapping of AUTOSAR runnables on multi-cores". In: *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*. 2013, pp. 1–5.

[For+08] Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. "A multi-periodic synchronous data-flow language". In: *2008 11th IEEE High Assurance Systems Engineering Symposium*. IEEE. 2008, pp. 251–260.

[For09] Julien Forget. "A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints". PhD thesis. Toulouse, France: Université de Toulouse - ISAE/ONERA, Nov. 2009.

[FP91] Tim Freeman and Frank Pfenning. "Refinement types for ML". In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 1991, pp. 268–277.

[GEL05] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. "Towards a flow analysis for embedded system C programs". In: *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. 2005, pp. 287–297.

[GHS94] R. Gerber, S. Hong, and M. Saksena. "Guaranteeing end-to-end timing constraints by calibrating intermediate processes". In: *Real-Time Systems Symposium, Proceedings*. 1994.

[GLS99] Thierry Grandpierre, Christophe Lavarenne, and Yves Sorel. "Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors". In: *Proceedings of the seventh international workshop on Hardware/software codesign*. ACM. 1999.

[GM01] Joel Goossens and Christophe Macq. "Limitation of the Hyper-Period in Real-Time Periodic Task Set Generation". In: *In Proceedings of the RTS Embedded System (RTS'01*. 2001.

[GNP06] Alain Girault, Xavier Nicollin, and Marc Pouzet. "Automatic rate desynchronization of embedded reactive programs". In: *ACM Transactions on Embedded Computing Systems (TECS)* 5.3 (2006), pp. 687–717.

[GR06] Denis Gopan and Thomas Reps. "Lookahead widening". In: *International Conference on Computer Aided Verification*. Springer. 2006, pp. 452–466.

[GTW11] Marc Geilen, Stavros Tripakis, and Maarten Wiggers. "The Earlier the Better: A Theory of Timed Actor Interfaces". In: *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control*. Chicago, IL, USA, 2011.

[Hal+91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. "The synchronous data-flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320.

[Har87] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of computer programming* 8.3 (1987), pp. 231–274.

[HGJ19] Nicolas Hili, Alain Girault, and Éric Jenn. "Worst-Case Reaction Time Optimization on Deterministic Multi-Core Architectures with Synchronous Languages". In: *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2019.

[HHK03] Thomas A Henzinger, Benjamin Horowitz, and Christoph M Kirsch. "Giotto: A time-triggered language for embedded programming". In: *Proceedings of the IEEE* 91.1 (2003), pp. 84–99.

[HMM12] Julien Henry, David Monniaux, and Matthieu Moy. "Pagai: A path sensitive static analyser". In: *Electronic Notes in Theoretical Computer Science* 289 (2012), pp. 15–25.

[HRP17] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. "The heptane static worst-case execution time estimation tool". In: *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[HRR91] Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. "Generating efficient code from data-flow programs". In: *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP '91)*. Passau, Germany, 1991, pp. 207–218.

[HW02] C.A. Healy and D.B. Whalley. "Automatic detection and exploitation of branch constraints for timing analysis". In: *IEEE Transactions on Software Engineering* 28.8 (Aug. 2002), pp. 763–781. ISSN: 1939-3520.

[Ioo+20] Guillaume Iooss, Marc Pouzet, Albert Cohen, Dumitru Potop-Butucaru, Jean Souyris, Vincent Bregeon, and Philippe Baufreton. *1-Synchronous Programming of Large Scale, Multi-Periodic Real-Time Applications with Functional Degrees of Freedom*. Tech. rep. Inria, 2020.

[JP86] Mathai Joseph and P. Pandya. "Finding Response Times in a Real-Time System". In: *The Computer Journal* 29.5 (1986), pp. 390–395.

[KBS20] Tomasz Kloda, Antoine Bertout, and Yves Sorel. "Latency upper bound for data chains of real-time periodic tasks". In: *Journal of Systems Architecture* 109 (2020), p. 101824.

[KCC20] Fouad Khenfri, Khaled Chaaban, and Maryline Chetto. "Efficient mapping of runnables to tasks for embedded AUTOSAR applications". In: *Journal of Systems Architecture* 110 (2020), p. 101800.

[KCH00] Saehwa Kim, Sukjae Cho, and Seongsoo Hong. "Schedulability-aware mapping of real-time object-oriented models to multi-threaded implementations". In: *Seventh International Conference on Real-Time Computing Systems and Applications, 2000. Proceedings*. 2000, pp. 7–14.

[Kha+16] Jad Khatib, Alix Munier-Kordon, Enagnon Cedric Klikpo, and Kods Trabelsi-Colibet. "Computing Latency of a Real-time System Modeled by Synchronous Dataflow Graph". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. Brest, France, 2016.

[KS03] H.Jin Kim and David H. Shim. "A flight control system for aerial robots: algorithms and experiments". In: *Control Engineering Practice* 11.12 (2003). Award winning applications-2002 IFAC World Congress. ISSN: 0967-0661.

[KV10] Johannes Kinder and Helmut Veith. "Precise static analysis of untrusted driver binaries". In: *Formal Methods in Computer Aided Design*. 2010.

[KWS03] Sharath Kodase, Shige Wang, and Kang G. Shin. "Transforming Structural Model to Runtime Model of Embedded Software with Real-Time Constraints". In: *Proceedings of the Conference on Design, Automation and Test in Europe: Designers' Forum - Volume 2*. DATE '03. Washington, DC, USA: IEEE Computer Society, 2003.

[LBP10] Daniel Le Berre and Anne Parrain. "The SAT4J library, Release 2.2, System Description". In: *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), pp. 59–64.

[Lee06] Edward A Lee. "The problem with threads". In: *Computer* 39.5 (2006), pp. 33–42.

[Ler09] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Commun. ACM* 52.7 (2009), 107–115. ISSN: 0001-0782.

[Li+13] Jianjun Li, Ming Xiong, Victor Lee, LihChyun Shu, and Guohui Li. "Workload-efficient deadline and period assignment for maintaining temporal consistency under edf". In: *Computers, IEEE Transactions on* 62.6 (2013), pp. 1255–1268.

[Lis14] Björn Lisper. "SWEET–a tool for WCET flow analysis". In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer. 2014, pp. 482–485.

[LL73] Cheng L. Liu and James W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment". In: *Journal of the ACM* 20.1 (1973).

[LMW95] Y-TS Li, Sharad Malik, and Andrew Wolfe. "Efficient microarchitecture modeling and path analysis for real-time software". In: *Proceedings of the 16th IEEE Real-Time Systems Symposium*. Pisa, Italy: IEEE, 1995, pp. 298–307.

[LSV96] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. "Comparing models of computation". In: *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided design (ICCAD'96)*. San Jose, USA: IEEE Computer Society, 1996, pp. 234–241.

[LW82] Joseph Y. T. Leung and Jennifer Whitehead. "On the complexity of fixed-priority scheduling of periodic, real-time tasks". In: *Performance Evaluation* 2.4 (1982).

[Mai+16] Cláudio Maia, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. "A closer look into the aer model". In: *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*. IEEE. 2016.

[Mai+17] Cláudio Maia, Geoffrey Nelissen, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. "Schedulability analysis for global fixed-priority scheduling of the 3-phase task model". In: *Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE. 2017.

[Mat+18] Joel Matějka, Björn Forsberg, Michal Sojka, Zdeněk Hanzálek, Luca Benini, and Andrea Marongiu. "Combining PREM Compilation and ILP Scheduling for High-performance and Predictable MPSoC Execution". In: *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*. Vienna, Austria, 2018.

[MDC14] Renato Mancuso, Roman Dudko, and Marco Caccamo. "Light-PREM: Automated software refactoring for predictable execution on COTS embedded systems". In: *RTCSA*. IEEE Computer Society, 2014.

[Mel+15] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. "Memory-processor co-scheduling in fixed priority systems". In: *International Conference on Real Time and Networks Systems (RTNS)*. Lille, France, 2015.

[MH96] Florence Maraninchi and Nicolas Halbwachs. "Compositional semantics of non-deterministic synchronous languages". In: *European Symposium On Programming*. Springer. 1996, pp. 235–249.

[MMTS13] Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. "Support for End-to-End Response-Time and Delay Analysis in the Industrial Tool Suite: Issues, Experiences and a Case Study". In: *Computer Science and Information Systems* 10.1 (2013).

[Moh+05] S. Mohan, F. Mueller, W. Hawkins, M. Root, C. Healy, and D. Whalley. "ParaScale: exploiting parametric timing analysis for real-time schedulers and dynamic voltage scaling". In: *Proceedings of the 26th IEEE International Real-Time Systems Symposium*. San Antonio, TX, USA: IEEE, 2005, pp. 232–242.

[Moh+11] Sibin Mohan, Frank Mueller, Michael Root, William Hawkins, Christopher Healy, David Whalley, and Emilio Vivancos. "Parametric timing analysis and its application to dynamic voltage scaling". In: *ACM Trans. Embed. Comput. Syst.* 10.2 (Jan. 2011), 25:1–25:34. ISSN: 1539-9087. (Visited on 04/27/2022).

[Moh+13] Swarup Mohalik, Devesh B. Chokshi, Manoj G. Dixit, A. C. Rajeev, and S. Ramesh. "Scalable model-checking for precise end-to-end latency computation". In: *2013 IEEE International Symposium on Computer-Aided Control System Design (CACSD)*. Hyderabad, India, Aug. 2013.

[Mon76] J. D. Monk. "Mathematical Logic. Graduate Texts in Mathematics". In: vol. 37. Springer, 1976. Chap. Cylindric Algebras.

[Mub+15] Saad Mubeen, Mikael Sjödin, Thomas Nolte, John Lundbäck, Mattias Gålnander, and Kurt-Lennart Lundbäck. "End-to-end Timing Analysis of Black-box Models in Legacy Vehicular Distributed Embedded Systems". In: *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 2015.

[Mzi+13] Rania Mzid, Chokri Mraidha, Asma Mehiaoui, Sara Tucci-Piergiovanni, Jean-Philippe Babau, and Mohamed Abid. "DPMP: A Software Pattern for Real-time Tasks Merge". In: *Proceedings of the 9th European Conference on Modelling Foundations and Applications*. ECMFA'13. Berlin, Heidelberg: Springer-Verlag, 2013.

[OMG07] OMG. *A UML Profile for MARTE*. Tech. rep. Object Management Group, Inc, 2007.

[OMG10] OMG. *Systems Modeling Language*. Tech. rep. Object Management Group, Inc, 2010.

[Pag+14] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. "The ROSACE case study: From Simulink specification to multi/many-core execution". In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2014, pp. 309–318.

[Pag+18a] Bruno Pagano, Cédric Pasteur, Günther Siegel, and R Knizek. "A model based safety critical flow for the aurix multi-core platform". In: *Proceedings ERTS2, Toulouse, France* (2018).

[PC10] Rodolfo Pellizzoni and Marco Caccamo. "Impact of peripheral-processor interference on WCET analysis of real-time embedded systems". In: *IEEE Transactions on Computers* 59.3 (2010), pp. 400–415.

[Pel+11] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. "A predictable execution model for COTS-based embedded systems". In: *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2011.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. Cambridge, USA: MIT Press, 2002. ISBN: 0-262-16209-1.

[PL05] Rodolfo Pellizzoni and Giuseppe Lipari. "Feasibility Analysis of Real-Time Periodic Tasks with Offsets". In: *Real-Time Syst.* 30.1 (May 2005), pp. 105–128.

[Raj+10] A. C. Rajeev, Swarup Mohalik, Manoj G. Dixit, Devesh B. Chokshi, and S. Ramesh. "Schedulability and End-to-end Latency in Distributed ECU Networks: Formal Modeling and Precise Estimation". In: *Proceedings of the Conference on Embedded Software (EMSOFT'10)*. Scottsdale, USA, 2010.

[Ray08] Pascal Raymond. "Synchronous program verification with lustre/lesar". In: *Modeling and Verification of Real-Time Systems* (2008), p. 7.

[RC04] Jorge Real and Alfons Crespo. "Mode change protocols for real-time systems: A survey and a new proposal". In: *Real-time systems* 26.2 (2004), pp. 161–197.

[RCM17] J. Ruiz, H. Cassé, and M. de Michiel. "Working Around Loops for Infeasible Path Detection in Binary Programs". In: *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Sept. 2017, pp. 1–10.

[RDP17] Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. "Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures". In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.5s (Oct. 2017).

[RH01] Mario Aldea Rivas and Michael González Harbour. "MaRTE OS: An Ada kernel for real-time embedded applications". In: *International Conference on Reliable Software Technologies*. Springer. 2001, pp. 305–316.

[RN22] Vlad Rusu and David Nowak. "Defining Corecursive Functions in Coq Using Approximations". In: *ECOOP*. 2022.

[RS09] Christine Rochange and Pascal Sainrat. "A Context-Parameterized Model for Static Analysis of Execution Times". In: *Transactions on High-Performance Embedded Architectures and Compilers II*. Ed. by Per Stenström. Vol. 5470. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 222–241.

[Sch+20] Matheus Schuh, Claire Maiza, Joël Goossens, Pascal Raymond, and Benoît Dupont de Dinechin. "A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory". In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2020, pp. 283–295.

[Sho+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. "Sok:(state of) the art of war: Offensive techniques in binary analysis". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 138–157.

[SKW00] M. Saksena, P. Karvelas, and Y. Wang. "Automatic synthesis of multi-tasking implementations from real-time object-oriented models". In: *Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2000. (ISORC 2000) Proceedings*. 2000.

[SP17] Muhammad Refaat Soliman and Rodolfo Pellizzoni. "WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching". In: *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. 2017.

[SP78] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences. Computer-Science Department, 1978.

[SS94] Marco Spuri and John A Stankovic. "How to integrate precedence constraints and shared resources in real-time scheduling". In: *IEEE Transactions on Computers* 43.12 (1994), pp. 1407–1412.

[Tab+16] Rohan Tabish, Renato Mancuso, Saud Wasly, Ahmed Alhammad, Sujit S Phatak, Rodolfo Pellizzoni, and Marco Caccamo. "A real-time scratchpad-centric os for multi-core embedded systems". In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*. IEEE. 2016.

[Tab+17] Rohan Tabish, Renato Mancuso, Saud Wasly, Sujit S. Phatak, Rodolfo Pellizzoni, and Marco Caccamo. "A Reliable and Predictable Scratchpad-centric OS for Multi-core Embedded Systems". In: *RTAS*. IEEE Computer Society, 2017.

[Tal+06] Jean-Pierre Talpin, Christian Brunette, Thierry Gautier, and Abdoulaye Gamatié. "Polychronous mode automata". In: *Proceedings of the 6th ACM & IEEE International conference on Embedded software*. 2006, pp. 83–92.

[Tar72] Robert Tarjan. "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[Tar73] Robert Tarjan. "Enumeration of the elementary circuits of a directed graph". In: *SIAM Journal on Computing* 2.3 (1973), pp. 211–216.

[The+03] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. "An abstract interpretation-based timing validation of hard real-time avionics software". In: *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings*. 2003, pp. 625–625.

[The22] The Coq team. *The Coq proof assistant*. Version 8.16. Sept. 5, 2022. URL: http://coq.inria.fr/.

[TJS15] Jean-Pierre Talpin, Pierre Jouvelot, and Sandeep Kumar Shukla. "Towards refinement types for time-dependent data-flow networks". In: *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE. 2015, pp. 36–41.

[Van+22] Florian Vanhems, Vlad Rusu, David Nowak, and Gilles Grimaud. "A Formal Correctness Proof for an EDF Scheduler Implementation". In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2022, pp. 281–292.

[Ver20] Micaela Verrucchi. "A comprehensive analysis of DAG tasks: solutions for modern real-time embedded systems". PhD thesis. University of Modena and Reggio Emilia, 2020.

[Viv+01] Emilio Vivancos, Christopher Healy, Frank Mueller, and David Whalley. "Parametric Timing Analysis". In: *Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems*. OM '01. New York, NY, USA, Aug. 2001, pp. 88–93. ISBN: 978-1-58113-426-1.

[Wan12] Jiacun Wang. *Timed Petri nets: Theory and application*. Vol. 9. Springer Science & Business Media, 2012.

[WGD14] Yifan Wu, Zhigang Gao, and Guojun Dai. "Deadline and activation time assignment for partitioned real-time application on multiprocessor reservations". In: *Journal of Systems Architecture* 60.3 (2014), pp. 247–257.

[Wil+08] Reinhard Wilhelm et al. "The worst-case execution-time problem - overview of methods and survey of tools." In: *ACM Trans. Embedded Comput. Syst.* 7 (Jan. 2008).

[Woz+13] E. Wozniak, A. Mehiaoui, C. Mraidha, S. Tucci-Piergiovanni, and S. Gerard. "An optimization approach for the synthesis of AUTOSAR architectures". In: *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*. 2013.

[WP13] Saud Wasly and Rodolfo Pellizzoni. "A Dynamic Scratchpad Memory Unit for Predictable Real-Time Embedded Systems". In: *ECRTS*. IEEE Computer Society, 2013.

[WP14] Saud Wasly and Rodolfo Pellizzoni. "Hiding memory latency using fixed priority scheduling". In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2014).

[Yao+12] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. "Memory-centric scheduling for multicore hard real-time systems". In: *Real-Time Systems* 48.6 (2012), pp. 681–715.

[Yao+16] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo. "Global Real-Time Memory-Centric Scheduling for Multicore Systems". In: *IEEE Trans. Comput.* 65.9 (Sept. 2016).

[Yip+16] Eugene Yip, Alain Girault, Partha S Roop, and Morteza Biglari-Abhari. "The ForeC synchronous deterministic parallel programming language for multicores". In: *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*. IEEE. 2016.

[YYR11] Simon Yuan, Li Hsien Yoong, and Partha S Roop. "Compiling esterel for multi-core execution". In: *2011 14th Euromicro Conference on Digital System Design*. IEEE. 2011.

[ZDN12] Haibo Zeng and M. Di Natale. "Efficient implementation of AUTOSAR components with minimal memory usage". In: *2012 7th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2012.

[ZG11] Ming Zhang and Zonghua Gu. "Optimization issues in mapping AUTOSAR components to distributed multithreaded implementations". In: *2011 22nd IEEE International Symposium on Rapid System Prototyping (RSP)*. 2011, pp. 23–29.

[Če+15] Pavol Černý, Thomas A. Henzinger, Laura Kovács, Arjun Radhakrishna, and Jakob Zwirchmayr. "Segment Abstraction for Worst-Case Execution Time Analysis". en. In: *Programming Languages and Systems*. Ed. by Jan Vitek. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2015, pp. 105–131. ISBN: 978-3-662-46669-8.