Automated generation of time-predictable executables on multi-core

Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert, Arno Luppold

▶ To cite this version:

Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert, Arno Luppold. Automated generation of time-predictable executables on multi-core. RTNS 2018, Oct 2018, POITIERS, France. hal-01888728

HAL Id: hal-01888728
https://hal.science/hal-01888728v1

Submitted on 10 Oct 2018

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.

Automated generation of time-predictable executables on multi-core

Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert and Arno Luppold

October 8, 2018

Abstract

In this paper, we are interested in the implementation of control-command applications, such as the flight control system of an aircraft for instance, on multi-core hardware. Due to certification and safety issues, *time-predictability* – in the sense that the timing behavior must be analysable and validable off-line – is a mandatory feature. We present a complete framework, from high-level system specification in synchronous languages, to implementation on a multi-core hardware platform, which enforces time-predictability at every step of the development process. The framework is based on automated code generation tools to speed-up the development process and to eliminate error-prone human-made translation steps.

1 Introduction

Developing safety critical applications requires to respect standardized guidelines, concerning for instance the development process (e.g. the ARP 4754 [44] for aeronautics) or the software development (e.g. the DO-178C [43]). In the aeronautics domain, certification authorities require the designers to show, prove and argue that the system has been developed correctly and in compliance with the standards, such as the aforementioned ones. *Time-predictability* is one important objective according to the DO-178C, in the sense that the timing behavior of a system must be analyzable and validable off-line. Dealing with this objective for multi-core is known to be complex or even unfeasible in the general case [47, 48], because of intensive resource sharing, complex internal behavior and lack of documentation. In the academic literature, there has been a lot of effort to propose WCET analysis techniques and predictable programming languages; in the industry, dedicated standards have been defined. The most recent documents are the CAST-32A [10] and some recommendations published by the FAA [32]: they provide a set of guidelines, among which the applicant should identify the *interference channels* (e.g interference on memory buses) and argue that they are properly mitigated by adequate means.

In this paper, we focus on the software development process, to ensure *time-predictability* at every step of the process. Here, the term *time-predictability* encompasses: eliminating bus interferences thanks to the *AER (Acquisition Execution Restitution) execution model*, simplifying the computation of WCET upper-bounds, enabling the schedulability analysis of a task set, and preserving the semantics of the program throughout successive code generation steps.

1.1 Applicative context

A control-command application aims at controlling and supervising a physical entity in its environment. A typical example is the flight control system of an airplane. We consider the longitudinal flight controller of the Research Open-Source Avionics and Control Engineering (or ROSACE) case study [35]. Although of modest size, this controller is representative of real avionics applications.

Figure 1 depicts the ROSACE architecture. The *controller* is composed of 8 functions (depicted by boxes) that run at different periods and exchange data (depicted by arrows). A series of filters (named *X_filter*) consolidate the data measured on the aircraft and two controllers manage the airspeed (*va_control*) and the vertical speed (*vz_control* and *altitude_hold*). Data-dependencies are causal, for instance, the function *vz_filter* produces the variable *vz_f*, which is consumed by functions *vz_control* and *va_control*. The controller receives two inputs from the cockpit, which are the orders requested by the pilot on the altitude h_c and on the vertical air speed Va_c. The *Aircraft* receives the orders δ_{ec} and δ_{thc} computed by the controller.

1.2 Current industrial development process

The current development process followed by air-framers [46] for implementing such a controller on a single core architecture is the following. Each function is coded in the SCADE language [15] (the industrial version of LUSTRE [9]) and the assembly (that is how functions are interconnected and scheduled) is implemented in an ad-hoc language.

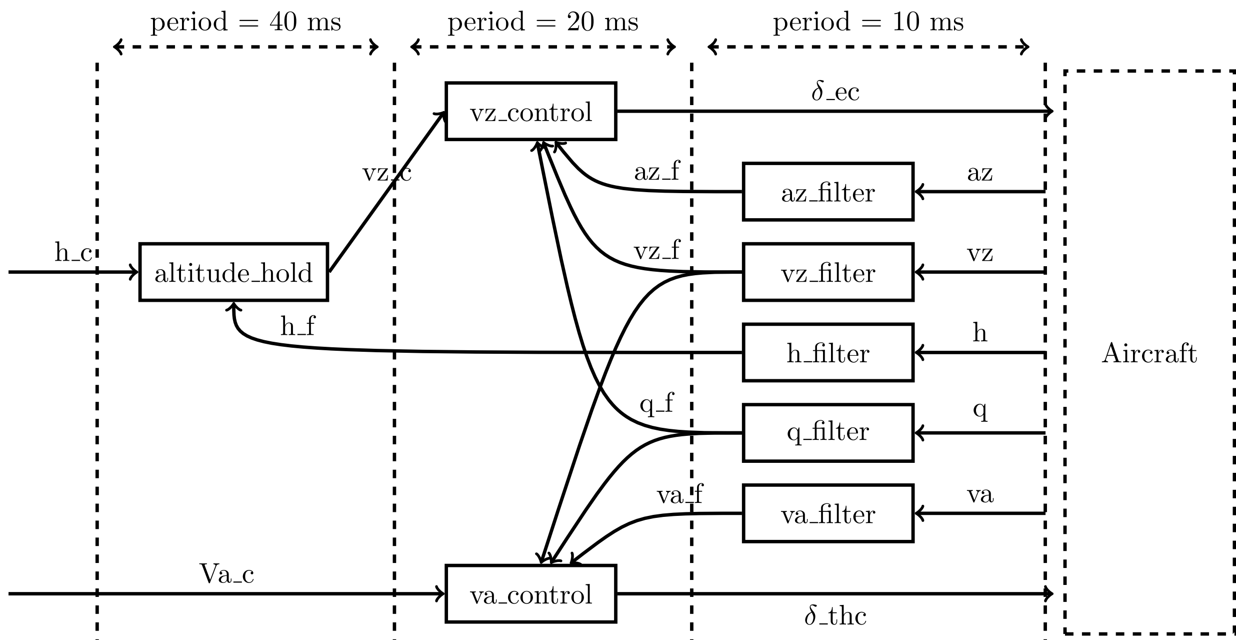

Figure 1: Rosace architecture

The proprietary qualified tool KCG of Ansys [16] translates each Scade program into an equivalent C code. The generated C code uses no dynamic memory allocation and only bounded loops, in order to ensure low timing variability. gcc is then used to generate the executable.

The WCET of the different functions is analyzed by the aiT tool from Absint [22]. Each function is analyzed in isolation using static analysis. Functions execute non-preemptively, following a schedule computed statically.

1.3 Contribution

In this work, we adapt the existing industrial process to support multi-core hardware, and also to rely on freely available software. Our framework covers the whole development process, from the high-level specification with synchronous languages, to the implementation on the target hardware platform. Thanks to automated code generation, the time-predictability of the developed system is established on its high-level specification and preserved during the subsequent steps of the development.

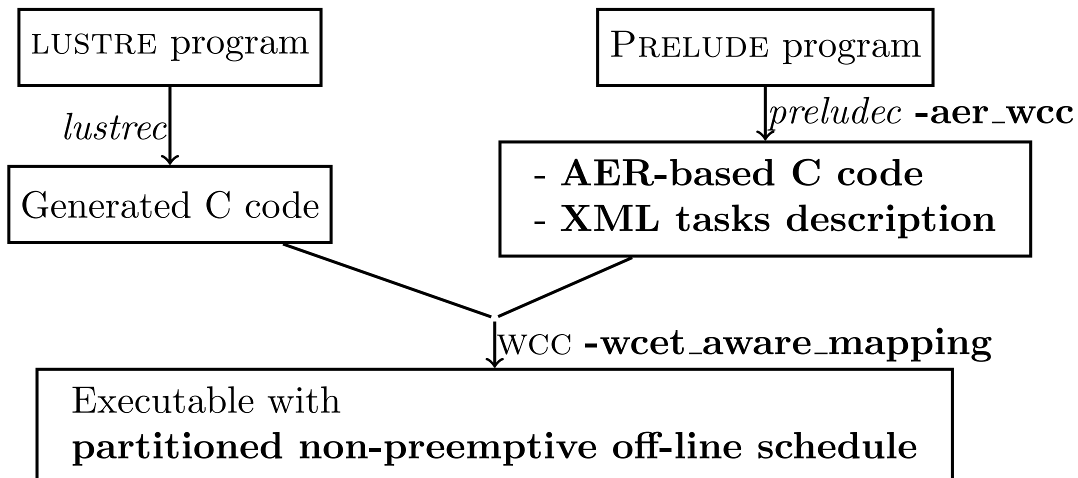

Figure 2: Framework overview

Figure 2 summarizes the framework and the novelties of the paper are highlighted in bold. The high-level specification is performed combining lustre, for the functions, and Prelude [34], for the assembly.

LUSTRE functions are compiled into C code using the open-source *lustrec* compiler from ONERA [21] [1]. The *preludec* compiler produces a C code that assembles the LUSTRE functions into a multi-task program. The C code is compiled using the WCC (WCET-aware C Compiler) compiler [17]. The back-end of WCC generates a partitioned non-preemptive schedule. The WCET of each function is analyzed separately using AIT. The application executes on an ARM-based multi-core architecture, which has the particularity of combining a global flash memory with private local addressable scratchpad memory (SPM), instead of local cache memory.

Time-predictability is addressed at each step of the framework. First, the synchronous approach [7], followed by LUSTRE, simplifies the programming of real-time aspects by reasoning on logical time, focusing only on the execution order instead of the exact execution dates. Second, PRELUDE transposes the synchronous semantics to the assembly level and introduces specific constructs to ensure that multi-rate communications have a deterministic semantics. *preludec* has been extended to generate multi-phase AER tasks [13], which separate computation phases from communication phases (as in [36, 8]). This effectively eliminates memory access contentions during computation phases, which simplifies the WCET analysis of these phases. Third, the schedule computed off-line ensures that real-time constraints will be respected. Furthermore, because we adopt a non-preemptive partitioned schedule, there is no interference between tasks due to preemption, which again simplifies the WCET analysis. Fourth, WCC is designed specifically to produce an internal representation that is more amenable to WCET analysis, meaning that the WCET over-estimation, performed by AIT, is reduced. Finally, relying on local addressable memory (scratchpad) instead of caches further simplifies the WCET analysis.

The main novelty of our work is to provide a complete development framework focusing on time-predictability on multi-core. Our approach extends and integrates previous ideas. The elements of the framework that are new are the following: the compilation of PRELUDE programs into AER tasks, the off-line scheduling of AER tasks, the integration of AER tasks and a task sequencer (off-line schedule) into WCC, and the hardware architecture choice with its dedicated execution model.

2 Framework overview

In this section we provide an overview of the development framework presented in Figure 2. This section focuses on the integration of the different steps of the framework. The extensions we made to the different elements of the framework will be detailed in subsequent sections.

2.1 Initial program

The system is programmed as a combination of PRELUDE [34] and LUSTRE [9], two synchronous languages[7].

Each function is first programmed separately as a LUSTRE *node*. The inputs and outputs of one node are all synchronous, which means in our context that they are all consumed or produced at the same periodic rate. Each node is compiled separately by the *lustrec* compiler, which translates LUSTRE code into C code.

LUSTRE nodes are then assembled together in a PRELUDE program, which details the real-time constraints of the system and the semantics of the communications between the LUSTRE nodes. This assembly is not done directly in LUSTRE because communications relate nodes executing at different periodic rates; PRELUDE is better suited to the specification, analysis and efficient compilation of such systems[34]. An example of PRELUDE program is provided in Code 1. *vz_filter* and *vz_control* are imported nodes, programmed separately in LUSTRE. The main node *rosace_main* details the data-flows between the imported nodes. For instance *vz_filter* produces value *vz_f*, which is consumed by *vz_control*. The rate of flow *vz_f* is reduced by 2 (*vz_f^/2*), so that *vz_control* executes twice slower than *vz_filter*. Periods are specified on the inputs and outputs of the main node, for instance *h_c* has a period of 100 and an offset of 0. *preludec* deduces the periods of the imported node calls from the period of its inputs.

Code 1 (PRELUDE program)**.**

```
imported node vz_filter (vz :real) returns (vz_f :real) wcet 1;
imported node vz_control (vz_f :real, [...])
  returns (delta_ec :real) wcet 1;
[...]
node rosace_main (h_c: real rate(100,0); va_c: real rate(100,0))
returns (delta_thc, delta_ec)
  var vz_f, delta_dec, [...] : real;
  [...]
  vz_f = vz_filter([...]);
  delta_ec  = vz_control(vz_f/^ 2, [...]);
  [...]
tel
```

[1] This work could easily be adapted to rely on *ec2c* and *poc* from the Verimag lab [39] instead.

preludec translates the PRELUDE program into a multi-task C code, where each LUSTRE node corresponds to one task. Tasks are executed periodically, the code executed during each period is the C code generated by LUSTRE for the corresponding node. In order to prevent memory access contentions during the execution of a task, the code of each task follows the AER model: there is a separate C function for the Acquisition phase, the Execution phase and the Restitution phase. The Acquisition phase is in charge of collecting all data required for the computation. This phase accesses to the shared memory (the flash memory), to fetch data into local memory. On the contrary, the Execution phase is pure computation, no access to shared memory occurs. The Restitution phase consists in copying all the results of the Execution phase from the local memory to the shared memory. Splitting the interference-free phases from the non interference-free phases greatly simplifies the computation of the WCET of each phase (see Section 3.2 for more details). Finally, *preludec* also generates a description of the task set in an xml file for WCC (see below, Section 2.2.2). The AER code generation process is a new contribution of this work, it is detailed in Section 4.

2.2 WCC

In this section, we give a brief overview of WCC, of the inputs needed in order to compile multi-task programs, and of how the off-line mapping of Section 5 was integrated in WCC.

2.2.1 Compiler overview

A compiler makes several transformation steps in order to transform a C code into an executable for a given target [1]. The transformation steps of WCC are detailed in Figure 3. The *parser* translates the C code into a high level representation (denoted ICD-C in WCC). The ICD-C representation is then translated into a low level representation (LLIR – Low-Level Intermediate Representation), which already depends partially on the processor target. Finally, the *assembler* and *linker* produce the executable. Compared to a classic C compiler, the specifics of WCC are:

1. The LLIR is analyzed with static WCET analysis tools. In this work, we use AIT of Absint [22];
2. WCC includes some optimizations on the LLIR targeted for the reduction of the WCET. In classic compilers, most standard optimizations are targeted for the reduction of the *average* execution-time not the *worst-case* execution time.

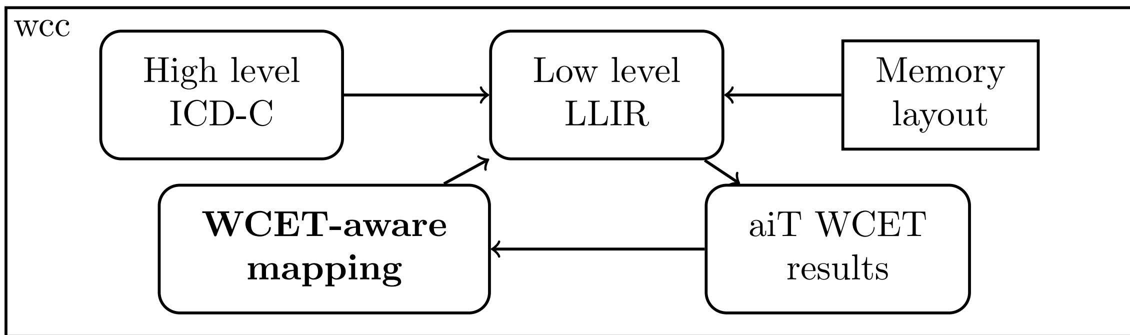

Figure 3: WCC internal architecture

2.2.2 Input description: application and architecture

In order to deal with timing aspects, WCC requires more information than just the C file(s). First, it requires a file that describes some characteristics of the physical architecture such as the number of cores, the memory sizes, the maximal read and write access time to each memory. Because we rely on a TDMA bus (see Section 3.1 for details), a second file describes the TDMA features and contains the slots length and the repetition length of the TDM. In a third file, the user specifies in an xml file the function set with the real-time attributes such as the period. In our case this file is generated by *preludec*, as well as the C files.

2.2.3 Code architecture

A specific mapping aware compilation procedure has been implemented in WCC. The algorithm 1 summarizes its main steps. First, we read the hardware characteristics. We consider that each core has direct access to a private scratchpad memory (SPM), and access to a shared flash via a private bus. Next, the first *for* loop reads the application description (stored in the xml file and in the C code). Inside the loop, AIT is called in order

to compute the WCET and the size of each function of the application and each of its AER phases. Thanks to the AER model, the aiT analysis can assume that code on each core executes in isolation from other cores. The access to the flash is possible only at pre-defined time slots (see Section 3 for more details on the hardware architecture). Once we have the WCET for each function, a constraint programming problem is generated, which describes the scheduling problem to solve with OPL IBM [27] (see Section 5 for more details). Once the scheduling problem is solved, the second *for* loop reads the off-line schedule computed by the solver. Finally, the local scheduler code is generated for each core.

Algorithm 1 WCET-aware mapping

```
 1: procedure LLIR_ExecutionModel (Config appli)
 2:     get SPM_size, SPM_stack_size, flash_size
 3:     get nb_core, bus_slot                                   ▷ Hardware input
 4:     for function: t in appli do
 5:         get t.period, t.name, t.subfunctions
 6:         call aiT
 7:         get t.wcet_x, t.size_x (all sections)
 8:     end for                                                 ▷ Application input
 9:     call OPL IBM solver to solve the mapping problem
10:     for function: t in appli do
11:         get t.core, t.start_x
12:     end for                                                 ▷ Off-line schedule
13:     for core: c in Cores do
14:         generate C local scheduler
15:     end for                          ▷ Integration code for off-line schedule
16: end procedure
```

2.3 Off-line schedule and mapping

The program schedule is computed off-line with OPL. An OPL problem description generally consists of two parts:

- A *.dat* file, containing the input data. This is the file generated by wcc, which depends on the application and on the hardware description;
- A *.mod* file, containing the constraints and the script that generates the results. This file is independent of the actual program to analyze. It only specifies constraints related to the execution model and to general rules related to the kind of scheduling problem we want to solve.

The mapping problem formulation in OPL is detailed in Section 5. The solver computes for each function on which core it must be executed and when each of its AER phases must start executing.

3 Real-time executive

In this section we detail the hardware and software platform on which the execution of the final embedded program relies.

3.1 ARM-based multi-core

wcc supports two types of processors: a single core Tricore and several ARM-based architectures that range from 1 to 8 cores. In this work, we rely on the ARM-based architecture that has initially been implemented by Timon Kelter [28], which is depicted in Figure 4.

Each core is cadenced at 1Ghz and is associated with a private local SPM. Such a memory is better suited for predictable implementation than traditional cache memories, which generate implicit traffic to maintain cache coherency. The main memory is a flash memory and cores access it through a bus arbitrated with a TDMA (Time Division Multiple Access) protocol. The access is managed by the hardware and no special additional code must be added by the developer. Even if this architecture does not reflect an existing platform, it is still realistic because the next generation of embedded processors for control-command application is expected to share similar features.

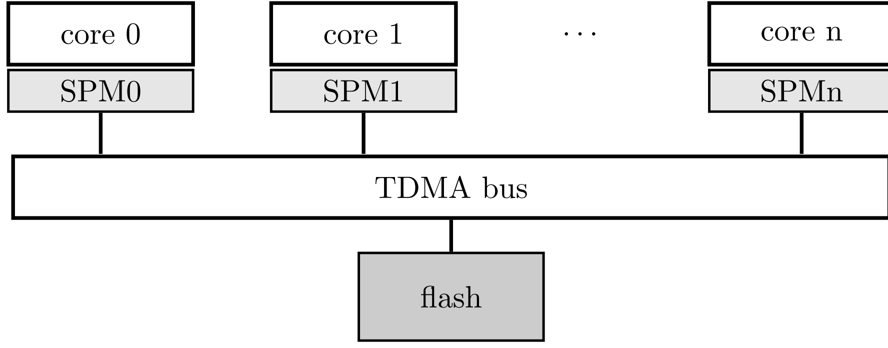

Figure 4: ARM-based multi-core architecture

3.2 Execution model

The structure of the WCC code on this platform follows an execution model detailed in [12, 33]. To summarize:

- There is a unique function per core;
- The local SPM is partitioned in two parts: one for the instructions (*section .text_spm*) and one for the stack (*section .stack*). Everything else is stored in the flash;
- Only local addressing on local SPM is supported, meaning that core i cannot access the SPM of core j when $i \neq j$.

In this work, we extend WCC to implement the AER model [13] on this platform. This is done as follows:

Rule 1: Several functions are sequenced in a non preemptive way on each core. No migration is allowed, meaning that a function will always execute on the same core. The sequence is computed off-line;

Rule 2: The function sections are allocated on the local SPM where they execute, except the exchanged data which are stored in the flash. Since there is no global addressing on the local memory, the flash is the only area where *all* cores can access;

Rule 3: A function is decomposed in 3 AER phases. Data consumed by the function is first read during an acquisition phase A from the flash and copied to local variables. Then the function executes in isolation from other cores and without accessing any shared resources during the execution phase E. Finally, data produced by this function and consumed by other functions is copied to the flash during the restitution phase R;

Rule 4: The A and R phases always occur during the TDMA slots of the core hosting the function.

Example 1. *The implementation shown in the Figure 5 fulfills the 4 rules of the execution model. The diagram shows part of a* ROSACE *execution on two cores.* vzcx *stands for* vz_control *in mode* $x \in \{A, E, R\}$*;* vffx *for* vf_filter*;* vacx *for* va_control*;* ahe *for* altitude_hold *in mode E. The variables that are written or read in the flash are shown on the flash line.*

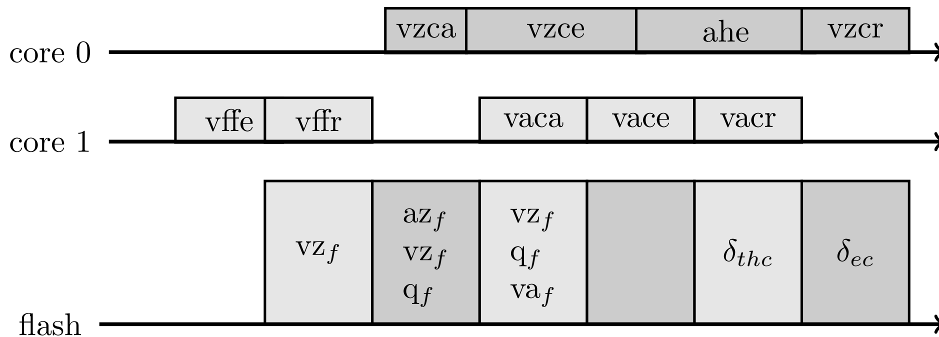

Figure 5: Dual core execution of part of ROSACE

Regarding rule 3, we assume that the code and data of the three AER phases can fit in the SPM permanently. Another solution would be to fetch the code of phase E during the A phase. This was more or less our approach in previous work [8].

3.3 Scheduler

Since the system schedule is computed off-line, and since we consider partitioned scheduling, the scheduler consists of a simple sequencer on each core. The sequencer is implemented in C, its structure is provided in Code 2.

A local table *table* stores the schedule computed off-line. The function *fill_table* fills the table with the start time and the address of each function. In this example, *vz_filter* and *altitude_hold* are the two functions allocated on the core. They are initialized by calling all the *[function name]_init*. At the end of the initialization phase, a synchronization barrier ensures that all cores start executing at the same time. In the infinite *while* loop, each sequencer scans its table, waits for the next function activation of the sequence and executes it.

Code 2 (C local scheduler)**.**

```
void main(){
  struct sched_table table;
  fill_table(&table);
  vz_filter_init();
  altitude_hold_init();
  init_barrier();

  while (1) {
    for (i=0; i< table.length; i++) {
      // should start at time t
      wait_time(table.task[i].t);
      // execute task
      (*table.task[i].function)();
    }}}
```

4 Prelude AER code generation

In this section, we detail how the inputs for WCC are generated by the *preludec* compiler. The main focus of PRELUDE is the implementation of deterministic multi-rate communications. Initially, communications were implemented based on Sofronis et al. ideas [45], using a shared memory approach. PRELUDE was later extended in two steps to support hardware platforms with distributed memories:

1. Puffistch et al. in [38] developed INTERLUDE, an extension of PRELUDE, which allowed *push* and *pull* communications on the many-core Intel Single-chip Cloud Computer (SCC). These ideas have since been introduced directly in *preludec* with the option *-extern_buffers* in the distribution 1.6.0. This option is generic and it is up to the designer to implement the read and write functions for a specific target.

2. The developers of *lustrec* and *preludec* eased the integration of C code generated by *lustrec* in the *preludec* imported node infrastructure.

In this work, we rely on these extensions and we adapt them to support the AER multi-phase task model.

4.1 Task code generation

For each LUSTRE node f, *lustrec* generates two C functions: an initializing function *f_init*, and a "step" function *f_step* corresponding to the code executed by one repetition of f. The initialization function is scheduled only once, at the beginning of the system execution, while the step function is scheduled periodically.

preludec wraps the function *f_step* with code dedicated to the implementation of communications with other functions. The option *-extern_buffers* generates a step function that includes code for acquiring inputs, then a call to the step function and then code for the restitution of outputs. In this work, we have introduced a new AER code generation option (*-aer_wcc*). This mode splits the "wrapping" code in three separate A (acquisition code), E (step function), R (restitution code) functions. It also separates initialization code in a dedicated I function.

Communication buffers. Communication buffers are allocated in a separate file, that goes into the flash memory. An example is provided in Code 3. The section "buffer" contains buffer declarations. For instance, variable *vz_filter95_Vz_f_vz_speed_control104_Vz_f* is the communication buffer allocated for the communication of variable *Vz_f* from *vz_filter* to *vz_speed_control*. Note that *preludec* allocates a buffer of size 2, because *vz_filter* executes twice faster than *vz_speed_controller*. *preludec* groups the addresses of all communication buffers in an array (*buffer_addresses*). Step functions can access to communication buffers through this array, using the buffer identifier generated in section "buffers id": values of the enumeration are declared in the same order as buffers, thus, for instance, *buffer_addresses[vz_filter95_Vz_f_vz_speed_control104_Vz_f_id]* is the address

of *vz_filter95_Vz_f_vz_speed_control104_Vz_f*.

Code 3 (C external buffers)**.**

```
// buffers
double vz_filter95_Vz_f_vz_speed_control104_Vz_f[2];
...
// buffers id
enum {
vz_filter95_Vz_f_vz_speed_control104_Vz_f_id,
...}
// table to access the buffers
void * buffer_addresses [PLUD_BUFFER_NUMBER] =
{ (void *) vz_filter95_Vz_f_vz_speed_control104_Vz_f,
  ...}
```

AER functions. Code 4 shows the code generated by *preludec* for the function *vz_filter*. *preludec* allocates variables to store the inputs and outputs of the function (variables *locread* and *locwrite*). These variables are local copies, allocated in the local SPM, of the global variables allocated in the flash memory. For instance, *vz_filter_95_fun_Vz_locread* is a copy of the buffer for the communication from *aircraft_dynamics* to *vz_filter*. The copy is performed by a call to *read_val*, during the A phase of *vz_filter* (i.e. *vz_filter_95_A*). Because the communication buffer contains two cells, the index variable *Vz_rcell* is used to read alternatively from the two cells. Note also that we use the buffer identifier in *read_val* to denote which buffer we want to access.

Similarly, *vz_filter_95_fun_Vz_f_locwrite* is a copy of the buffer for the communication from *vz_filter* to *vz_speed_control*. The copy is performed by a call to *write_val*, during the R phase of *vz_filter* (i.e. *vz_filter_95_R*). The function *must_write* determines whether the current value must be copied to the buffer or not, because *vz_control* only reads one out of two successive values produced by *vz_filter*.

Code 4 (C code generated by *preludec*)**.**

```
// local copy of communication buffers
static double vz_filter_95_fun_Vz_locread;
static double vz_filter_95_fun_Vz_f_locwrite;
...
// information on buffer sizes and communication patterns
struct write_proto_t
       vz_filter95_Vz_f_vz_speed_control104_Vz_f_write;
struct write_proto_t
      vz_filter95_Vz_f_va_speed_control92_Vz_f_write;
...
int vz_filter_95_I()
{
  int _idx_i=0;
  vz_filter95_Vz_f_vz_speed_control104_Vz_f_write.wpref_size=0;
...
  return 0;
}

int vz_filter_95_A(void* args)
{
  static int Vz_rcell=0;
  // copy from flash to local SPM
  read_val(aircraft_dynamics73_vz_vz_filter95_Vz_id, ...,
           &vz_filter_95_fun_Vz_locread);
  Vz_rcell=(Vz_rcell+1)%2;
  return 0;
}

int vz_filter_95_E(void* args)
{
  // vz_filter(...) is the step function generated by lustrec
  vz_filter_95_fun_Vz_f_locwrite=
    vz_filter(vz_filter_95_fun_Vz_locread);
  return 0;
}

int vz_filter_95_R(void* args)
{
  static int Vz_f_vz_speed_control104_Vz_f_wcell=0;
  static int instance=0;
  if(must_write(vz_filter95_Vz_f_vz_speed_control104_Vz_f_write,
                instance)){
    // copy from local SPM to flash
    write_val(vz_filter95_Vz_f_vz_speed_control104_Vz_f_id, ...,
              &vz_filter_95_fun_Vz_f_locwrite);
    ...
  return 0;
}
```

Finally, the execution phase *vz_filter_95_E* simply consists in calling the step function generated by *lustrec*.

Note that the function uses the local copies discussed above.

Local/global memory copies. Copies from/to the flash memory are performed by functions *read_val* and *write_val*. These functions are platform specific. For our ARM-based architecture, since we only manipulate *double* values, we implemented a monomorphic version of these functions. The code for *read_val* is shown in code 5. The function *write_val* is very similar. A polymorphic version that relies on *memcpy* is also available in the PRELUDE distribution.

Code 5 (Function read_val)**.**

```
void read_val(int id, int cell, int size, void* addr){
  double *p = (double *)addr;
  *p=buffer_address[id][cell];
}
```

4.2 Task set description

preludec computes a *clock* for each node call during the *clock calculus* analysis[19]. The activation period, offset and deadlines of each node are directly derived from its clock. The production of the task set description as an xml file, as required by WCC, is thus a mere syntactic transformation step. The xml file lists the different A/E/R/I functions, along with their associated real-time attributes.

5 Off-line mapping problem

In this section, we describe the mapping problem formulation in OPL [27]. Computing off-line mapping and scheduling with constraint programming is not new. The novelties here are twofold:

- We use the *Conditional Time-Intervals* that have been introduced into IBM ILOG CP Optimizer since version 2.0 for AER model. This approach, as described in [37], has been shown very efficient to solve non preemptive off-line mapping problems;
- We propose a procedure to reduce the window size which is very efficient in average.

In scheduling theory, the elements being scheduled are called *tasks* and *jobs*. In our case, each LUSTRE function of the application corresponds to a task. Each periodic repetition of a task corresponds to a job.

5.1 Background on conditional time-intervals

Detailed definitions and descriptions can be found in [29] and [30]. The Figure 6 shows the classic way to represent a non preemptive task that could execute on two different cores with a conditional time interval-based approach. The formulation of this representation is done as follows. A task t_1 is associated with:

- an interval i_1 of length t_1.wcet (not represented on the figure);
- two conditional intervals $i_{1,1}$ and $i_{1,2}$.

To express that only one conditional interval will be present in the result (t_1 executes only on one core) and that it will be synchronized with i_1 (to determine the start time of t_1 and to impose the execution length to be the WCET), the constraint is:

$$alternative(i_1, all(x \in 1,2) i_{1,x})$$

The solver then computes the values of $presenceOf(i_{1,1})$, $presenceOf(i_{1,2})$ and $startOf(i_1)$. In the Figure 6, the task executes on the second core.

Let us consider a second task t_2. t_2 is associated with i_2, $i_{2,1}$ and $i_{2,2}$ as for t_1. If t_2 is also mapped to the second core, the execution of the two tasks must not overlap, indeed the CPU must be accessed in mutual exclusion. To ensure this constraint, OPL provides the notion of cumulative function, named *pulse*, that computes the sum of access quantity made by the different intervals. The Figure 7 shows the cumulative of two intervals. This schedule does not satisfy exclusive core access condition code and thus is not correct. We must add a constraint to ensure the exclusive core access condition. For instance, to ensure this on the second core, we add the following constraint:

$$pulse(i_{1,2}, 1) + pulse(i_{2,2}, 1) \leq 1$$

Such a formulation is really compact compared to usual formulation and also much more efficient thanks to the algorithms implemented in OPL. For the same problem with two tasks, in the usual formalization, the

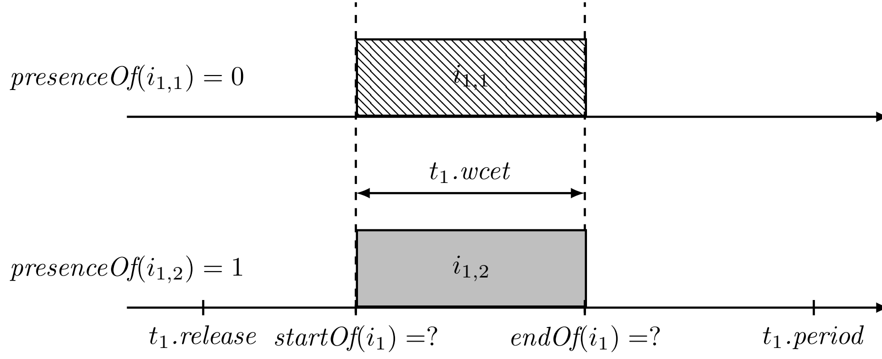

Figure 6: Conditional time intervals

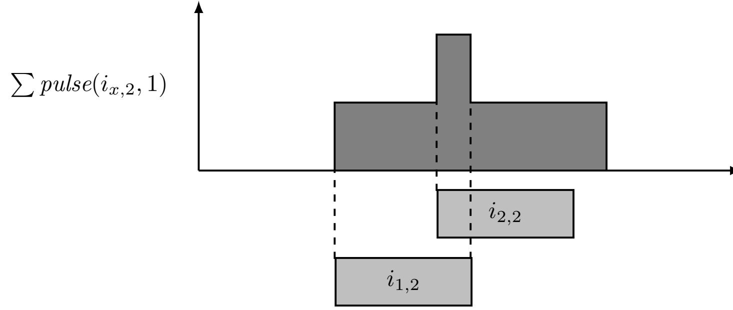

Figure 7: Cumulative function of two intervals

variables would be $c_1, c_2 \in \{0, 1\}$ representing on which core t_1 and t_2 execute; $s_1, s_2 \in \{0, .., L\}$ representing at which dates t_1 and t_2 start their execution. Then to ensure the mutual exclusion, the constraint would be:

$$c_1 = c_2 \implies (s_1 + t1.wcet \leq s_2) \vee (s_2 + t2.wcet \leq s_1)$$

Using $\implies$ and $\vee$ in constraint programming is very costly and should be avoided as much as possible. However, since most of our problem relates to mutual exclusion or bounded quantity accesses, these constraints are needed.

5.2 Problem inputs

We can now express the mapping problem for AER synchronous programs executing on the ARM-based architecture. The problem inputs are the hardware specification encoded as:

- *nbCores* the number of cores, *SPMsize* the size of the local SPM, *StackSPMsize* the size of the local memory area dedicated to store the stack, *flashsize* the size of the external memory;
- *MAF* is the length of the TDMA cycle on the bus, *StartBusSlot[nbCores]* is the beginning of the bus temporal slot allocated to each core on the bus in the MAF;

and the application description encoded as:

- *TaskList* the name of the application functions, *TaskProps [TaskList]* the attributes associated to each function. In particular, *TaskProps[t].period* gives the period of function t; $TaskProps[t].wcet_x$ provides the WCET for each phase $x \in \{A, E, R\}$; $TaskProps[t].size_{x,y}$ details for each phase $x \in \{A, E, R\}$ the size of the section $y \in \{.data, .stack, .others\}$. The term *others* refers to all possible sections, except the *.stack*.

An OPL script first unrolls the jobs on the study window, the length of which will be discussed in the very next section. After this unrolling, we obtain:

- the set of *Jobs*,
- *JobProps[Jobs]* the attributes associated to each job. More precisely, *JobProps[j].function* gives the name of the function j is a job of; *JobProps[j].release* provides the release date of j; whereas *JobProps[j].deadline* is its deadline.

5.3 Study window

Since the system is composed of functions with different periods, we must compute the schedule on the hyper-period H of the tasks (where $H{=}lcm_t\,TaskProps[t].period)$). Unfortunately, this means that the size of the schedule H can theoretically be exponential in the number of tasks in the worst-case. In order to improve the resolution time, we try to reduce the size of the problem by computing a schedule on a shorter *study window*, such that we can deduce the schedule on H from the schedule on the study window. We propose to define this shorter window with length $SWsize$ as:

$$\begin{cases} SWsize \leq H \\ SWsize | H \\ MAF | SWsize \\ \forall t \in TaskList, \begin{cases} either\ TaskProps[t].period | SWsize \\ or\ SWsize | TaskProps[t].period \end{cases} \\ a\ valid\ schedule\ can\ be\ found \end{cases}$$

The symbol | means *divides*, thus $MAF|SWsize$ is satisfied if the MAF divides $SWsize$. We need to take into account the MAF of the TDMA bus for computing the study window size. However since we can configure this parameter, we choose it wisely – for instance, such that $MAF| \gcd_t(TaskProps[t].period)$. Function periods are usually much slower than hardware capacities, thus this constraint should be easily reachable. As a consequence, the MAF does not impact the study window size, the other constraints on the other hand really matter. From the constraints, in the best case, $SWsize$ is equal to the gcd of the periods and in the worst to the lcm. Moreover, a study window is valid if we are able to compute a schedule: all active jobs, i.e. such that $JobProps[j].release \leq SWsize$, must be scheduled in the study window, i.e. end their execution before $SWsize$.

Example 2. *Let us consider two functions t_1 and t_2 where*

name	*period*	$wcet_a$	$wcet_e$	$wcet_r$
t_1	*20*	*1*	*2*	*1*
t_2	*100*	*2*	*5*	*2*

then on a single core it is sufficient to compute an off-line schedule on a study window of size 20*, such as for instance:*

start time	*0*	*1*	*3*	*4*	*6*	*11*
	$t_1.a$	$t_1.e$	$t_1.r$	$t_2.a$	$t_2.e$	$t_2.r$

Since we do not know a priori the value of $SWsize$, we have to call the solver with different values of $SWsize$ as long as the solver does not find a solution. In the current implementation, we successively try values from gcd to lcm. From what we have observed, $SWsize$ is in general much lower than the hyper-period, but this is just an empiric knowledge. In the case of a geometric task set – that is when for all $t_1, t_2 \in TaskList$ either $TaskProps[t_1].period | TaskProps[t_2].period$ or $TaskProps[t_2].period | TaskProps[t_1].period$ – $SWsize$ is equal to the period of one of the tasks. This is commonly the case in controllers we studied, which simplifies the problem. Note that reducing $SWsize$ reduces the size of the scheduling table, which reduces the execution time and memory footprint of the embedded program.

5.4 Decision variables

The variables that must be computed by the solver are:

1. the interval *phaseX[j in Jobs]* of length *TaskProps[JobProps[j]. task].wcet*$_x$, which represents the execution of job j during the phase $X \in A, E, R$;

2. the interval *phaseX_c[j in Jobs][c in Cores]*, which is computed only if the job j runs on core c and which does not exist otherwise. When the interval exists, it is synchronized with *phaseX[j]*.

5.5 Constraints

The first constraint is directly related to the conditional intervals approach. We must associate each *phaseX* interval to a unique *phaseX_c* and their timing positions must be synchronized.

$$\begin{aligned} &\forall j \in Jobs, \\ &\quad alternative(phaseX[j], all(c \in Cores)\ phaseX_c[j][c]) \end{aligned} \tag{1}$$

The next constraint is related to the execution order of the different phases of a job, that is A must be before E, which must execute before R.

$$\begin{array}{l}\forall j \in Jobs, \\ \quad endBeforeStart(phaseA[j], phaseE[j]) \\ \quad \wedge\ endBeforeStart(phaseE[j], phaseR[j])\end{array} \tag{2}$$

Then we must ensure that each job will execute between its release date and its deadline.

$$\begin{array}{l}\forall j \in Jobs, \\ \quad startOf(phaseA[j]) \geq JobProps[j].release \\ \quad \wedge\ endOf(phaseR[j]) \leq JobProps[j].deadline\end{array} \tag{3}$$

All jobs of the same function must execute on the same core.

$$\begin{array}{l}\forall j \in Jobs, c \in Cores, X \in \{A, E, R\} \\ \quad presenceOf\ (phaseA_c[JobProps[j].function][c]) \\ \quad == presenceOf\ (phaseX_c[j][c])\end{array} \tag{4}$$

There is no job execution overlapping on a core.

$$\forall c \in Cores, \sum_{j \in Jobs} pulse(\sum_{X \in \{A,E,R\}} phaseX_c[j][c], 1) \leq 1 \tag{5}$$

The sections stored in a given memory area do not exceed the size of this memory.

$$\begin{array}{l}\forall c \in Cores, \\ \quad \big(\sum_{t \in TaskList} presenceOf(phaseA_c[t][c]) \\ \quad\quad \times(\sum_{x \in \{A,E,R\}} TaskProps[t].size_{x,.others})\big) \\ \quad \leq SPMsize \\ \\ \quad \big(\sum_{t \in TaskList} presenceOf(phaseA_c[t][c]) \\ \quad\quad \times(\sum_{x \in \{A,E,R\}} TaskProps[t].size_{x,.stack})\big) \\ \quad \leq StackSPMsize\end{array} \tag{6}$$

The acquisition and restitution phases must execute within the TDMA slot allocated to the core.

$$\begin{array}{l}\forall j \in Jobs, c \in Cores, X \in \{A, R\} \\ \quad presenceOf(phaseX_c[j][c]) \\ \quad\quad \implies ((startOf(phaseX[j]))mod\ MAF == StartSlotBus[c])\end{array} \tag{7}$$

6 Case study

The code developed in WCC is around 1200 lines of C++ and the code developed in *preludec* is around 300 lines of OCAML. We applied the framework on ROSACE and on the WATERS 2017 industrial challenge.

ROSACE We have implemented ROSACE in LUSTRE and PRELUDE, then generated the AER-based code and compiled it with WCC. The execution times of the different steps are given in the table below.

preludec	WCC phase 1	OPL	WCC phase 2
0m0.114s	1m45.132s	0m0.601s	0m20.481s

WATERS 2017 Industrial Challenge The second use case was proposed by Arne Hamann, Simon Kramer, Martin Lukasiewycz and Dirk Ziegenbein from Robert Bosch GMBH as the WATERS 2017 challenge [26, 25]. The application is composed of:

- 1250 *runnables*, where a runnable is a piece of code that reads a set of *labels* (variables) and writes another set of labels. A runnable may be periodic or sporadic, in the sequel we only consider periodic ones;
- 10000 labels, that serve as the communication variables between runnables. A label can be read by several runnables but can only be written by at most one, which we call the *producer* of that label. A label that is not consumed by any runnable is an external output of the system (i.e. data sent to an actuator) and a label that is not produced by any runnable is an external input (i.e. data originating from a sensor).

Period	0.7ms	0.9ms	1ms	1.1ms	1.5ms	1.7ms	2ms	4.9ms	5ms
Number of runnables	4	5	41	3	8	5	27	5	26
Period	6ms	6.66ms	9.5ms	10ms	20ms	50ms	100ms	200ms	1000ms
Number of runnables	2	146	6	303	306	45	246	14	43

Table 1: Tasks description

We have translated the AMALTHEA tasks description [2] in PRELUDE, where each runnable becomes a node. More details on the translation and semantics can be found in [20]. The PRELUDE program has been compiled with the framework. Since we do not have the internal code of the runnables, we simply created an empty C function for each corresponding node. The size of the PRELUDE code is about 1Mo, while the size of the C code is about 11Mo. Due to the size of the C code, we had to increase the size of the local SPM to compute the memory mapping because of our choice to store code in each processor SPM. In future work we plan to modify A functions to pre-fetch task code at task release. The execution times of the different steps are given in the table below. The mapping is done on 4 cores.

preludec	WCC phase 1	OPL	WCC phase 2
0m9.163s	3550m15.365s	0m36,074s	30m22.548s

7 Related works

7.1 WCET assessment for synchronous programs

SCADE suite now comes with an integration of AIT and STACKANALYZER [18]. The processor model is quite generic but it provides the designer with a means to compare the memory/time cost of different programs.

We disabled most C compilation optimizations in our framework. Indeed, optimization are often avoided in safety critical embedded systems because they can break traceability and their correctness is difficult to prove. COMPCERT [31] is the first proved compiler that offers some optimization with a sufficient confidence to possibly be introduced in the design of future airplanes. COMPCERT has been assessed in the aeronautics context to implement a flight control system [6]. The system is specified in SCADE and the generated C code is then compiled for a Freescale MPC755 single-core, superscalar, pipelined microprocessor. The authors compared the performances between GCC and COMPCERT in terms of WCET and code size, all computed with AIT. The results were quite encouraging thanks to the optimization offered by COMPCERT since the WCET was in general 10% less than with a compilation with GCC and no optimization. The authors also summarize the expectation on a compiler regarding optimization from the DO 178: *an optimization is acceptable as long as a verification provides enough coverage*. Our purpose here is different since we target predictable execution on multi-core as a first objective. However, integrating safe optimization on predictable implementation would be a challenging issue worth to be considered in some future work.

The authors of [40, 41] focus on the WCET analysis and optimization of synchronous programs. In terms of tools, they rely on GCC for ARM processor, the e2c LUSTRE compiler and OTAWA [4] as static WCET analysis tool. They first remind the characteristics of the CFG (control flow graph) associated to a synchronous program: very simple structure mainly made of sequential basic blocks and nested conditionals; the calling tree of sub-functions is statically bounded. However, the designer may call external code or library for which a priori no assurance exists. In the implementation of ROSACE for instance, we called a math library provided with WCC and for which AIT is able to compute the WCET. From a LUSTRE program, the authors compute unfeasible paths by mixing model checking and IPET approaches in order to improve the quality of the WCET estimation. This work is also orthogonal to ours since it optimizes the WCET for a single core and a single function. Their solution could be used for the E phase of our execution model.

7.2 Predictable execution

Many works tackle the implementation real-time applications on multi-core platforms. In the family of software-based solutions to enforce the predictability, there are two main approaches: 1) interference-aware control software that manages, in a transparent way for the embedded application, the interference by sequencing the shared resources accesses; 2) application-aware control that consists in modifying the applications.

Since we work at the compiler level, we definitely belong to the second category which has the advantage to be more efficient, if the applied modifications are well realized. We already mentioned, in the introduction, the execution models [8, 36, 13]. Compared to those existing rules, the one proposed here is a direct reuse of standard ideas. The off-line computation with a constraint programming approach is also classic as for instance done in [38, 23, 5, 11]. In [42], the authors propose an off-line mapping of SCADE/LUSTRE programs on the Kalray MPPA to improve the WCET. In the continuation of the former work, Graillat and al. [24] have developed a

parallelization framework of LUSTRE programs on the Kalray MPPA that follows the off-line mapping computed before. Their idea is to manage the data-flow between distant nodes as an explicit communication, similarly to the push/pull communication of [38]. We reused the most state of the art modeling framework of conditional time interval as we initially performed in [37]. The main novelty here is to integrate all the steps in an automated way by generating the wrapping code from the synchronous compilers, by interacting with AIT and the OPL solver. With this framework, further interaction between the WCET analyzer, the off-line computation and the code transformation could be imagined.

In [12], we considered a unique function to be mapped on the ARM-based architecture. We implemented a WCC-optimization that computed off-line a memory mapping for the instructions, i.e. some are stored in the local SPM while the rest is in the flash; in order to reduce the WCET value.

Time-predictable implementation of multi-threaded programs has been studied in [3], for mono-core hardware, and later in [49], for multi-core hardware. However, the input language does not provide means to specify real-time constraints and thus ignores the real-time schedulability analysis problem.

Time-predictability is also the main concern of Precision Timed (PRET) machines [14] and many associated works, although authors work at a different level from us, since they focus on the hardware architecture.

8 Conclusion

We developed a framework that allows to produce time-predictable code for an ARM-based multi-core platform, starting from LUSTRE and PRELUDE input programs. The approach has been applied to two case-studies. The ROSACE case-study is solved very efficiently. The WATERS 2017 industrial challenge is much more complex, but our framework is still able to handle it. We will pursue our experiments to detect potential slowdown but we are optimistic on the mapping resolution since it has been shown very efficient in our previous benchmarks in a different context.

For future works, we plan to focus on the following improvements. 1) Improving the semantics and capacities of the xml file. There are many equivalent ways to express the same input, which should be avoided; and on the other hand new properties cannot be added by the user, e.g. precedence constraints between functions, as it is required in control-command applications; 2) Defining a data-flow dedicated internal representation. Due to the specifics of synchronous programs, one further step towards a really efficient and adequate compilation could be to define a fully specific representation; 3) On-line scheduling. We are currently working on an on-line scheduling policy, as an alternative for the current off-line solution.

Acknowledgments

This work has been funded by the DGA in the context of a *Stage ERE N°2016.60.0004.00.470.75.01*. The authors would like to thank Eric Noulard for his precious advice.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] AMALTHEA. An open platform project for embedded multicore systems. Available: `http://www.amalthea-project.org`.

[3] Sidharta Andalam, Partha Roop, and Alain Girault. Predictable multithreading of embedded applications using pret-c. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 159–168. IEEE, 2010.

[4] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *8th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, volume LNCS-6399, pages 35–46, 2010.

[5] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Åkesson, Vincent Nélis, and Thomas Nolte. Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform. In *28th Euromicro Conference on Real-Time Systems (ECRTS'16)*, July 2016.

[6] Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *ERTS2 2012: Embedded Real Time Software and Systems*, Toulouse, France, 2012.

[7] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. In *Readings in hardware/software co-design*. Kluwer Academic Publishers, 2001.

[8] Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic execution model on cots hardware. In *Proceedings of the 25th International Conference on Architecture of Computing Systems (ARCS'12)*, pages 98–110, 2012.

[9] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[10] CAST (Certification Authorities Software Team). Position Paper on Multi-core Processors - CAST-32A, 2016. Retrieved from https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32A.pdf.

[11] Silviu S. Craciunas and Ramon Serna Oliver. Combined task- and network-level scheduling for distributed time-triggered systems. *Real-Time Systems*, 52(2):161–200, 2016.

[12] Arno Luppold Dominic Oehlert and Heiko Falk. Bus-aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems. In *In Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS)*, Dubrovnik, Croatia, June 2017. Accepted for publication.

[13] Guy Durrieu, Madeleine Faugre, Sylvain Girbal, Daniel Gracia Prez, Claire Pagetti, and Wolfgang Puffitsch. Predictable flight management system implementation on a multicore processor. In *Proceedings of the 7th Conference on Embedded Real Time Software and Systems (ERTS'14)*, 2014.

[14] Stephen A Edwards and Edward A Lee. The case for the precision timed (pret) machine. In *Proceedings of the 44th annual Design Automation Conference*, pages 264–265. ACM, 2007.

[15] Esterel Technologies, Inc. *SCADE Language - Reference Manual*, 2012.

[16] Esterel Technologies, Inc. *SCADE Suite KCG C & ADA Code Generator*, 2012. http://www.esterel-technologies.com/products/scade-suite/automatic-code-generation/scade-suite-kcg-ada-code-generator/.

[17] Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, 46(2):251–300, 2010.

[18] C Ferdinand, R Heckmann, T Le Sergent, D Lopes, B Martin, X Fornari, and F Martin. Combining a high-level design tool for safety-critical systems with a tool for wcet analysis on executables. In *Embedded Real Time Software and Systems (ERTS2'12)*, 2008.

[19] Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A multi-periodic synchronous data-flow language. In *Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*, 2008.

[20] Julien Forget, Frédéric Boniol, and Claire Pagetti. WATERS Industrial Challenge 2017 with Prelude. In *8th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems WATERS'17*, 2017.

[21] Pierre-Loïc Garoche, Falk Howar, Temesghen Kahsai, and Xavier Thirioux. Testing-based compiler validation for synchronous languages. In *6th International Symposium of NASA Formal Methods (NFM'14)*, pages 246–251, 2014.

[22] AbsInt Angewandte Informatik GmbH. Worst-case execution time analyzer ait, 2016. https://www.absint.com/ait/.

[23] Raul Gorcitz, Emilien Kofman, Thomas Carle, Dumitru Potop-Butucaru, and Robert de Simone. On the scalability of constraint solving for static/off-line real-time scheduling. In *13th International Conference Formal Modeling and Analysis of Timed Systems (FORMATS'15)*, pages 108–123, 2015.

[24] Amaury Graillat, Matthieu Moy, Pascal Raymond, and Benoît Dupont De Dinechin. Parallel Code Generation of Synchronous Programs for a Many-core Architecture. In *Design, Automation and Test in Europe (DATE'18)*, Dresden, Germany, 2018.

[25] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, Falk Wurst, and Dick Ziegenbein. Waters industrial challenge 2017, 2017.

[26] Arne Hamann, Dick Ziegenbein, Simon Kramer, and Martin Lukasiewycz. Demo abstract: Demonstration of the fmtv 2016 timing verification challenge. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–1, 2016.

[27] IBM ILOG. CPLEX Optimization Studio, 2014. http://www.ibm.com/software/integration/optimization/cplex-optimization-studio/.

[28] Timon Kelter. *WCET Analysis and Optimization for Multi-Core Real-Time Systems*. PhD thesis, Technischen Universität Dortmund an der Fakultät für Informatik, 2014.

[29] Philippe Laborie and Jérôme Rogerie. Reasoning with Conditional Time-intervals. In *21st International Florida Artificial Intelligence Research Society Conference (FLAIRS'08)*, 2008.

[30] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. Reasoning with Conditional Time-Intervals. Part II: An Algebraical Model for Resources. In *22nd International Florida Artificial Intelligence Research Society Conference (FLAIRS'09)*, 2009.

[31] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 42–54, 2006.

[32] Laurence H. Mutuel, Xavier Jean, Vincent Brindejonc, Anthony Roger, Thomas Megel, and E. Alepins. Assurance of Multicore Processors in Airborne Systems, 2017. Retrieved from `http://www.tc.faa.gov/its/worldpac/techrpt/tc16-51.pdf`.

[33] Dominic Oehlert, Arno Luppold, and Heiko Falk. Practical challenges of ilp-based SPM allocation optimizations. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES'16)*, pages 86–89, 2016.

[34] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, 2011.

[35] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The rosace case study: From simulink specification to multi/many-core execution. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*, April 2014.

[36] 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 *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2011)*, 2011.

[37] Quentin Perret, Pascal Maurère, Éric Noulard, Claire Pagetti, Pascal Sainrat, and Benoît Triquet. Mapping hard real-time applications on many-core processors. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS'16, pages 235–244, 2016.

[38] Wolfgang Puffitsch, Éric Noulard, and Claire Pagetti. Off-line mapping of multi-rate dependent task sets to many-core platforms. *Real-Time Systems*, 51(5):526–565, 2015.

[39] Pascal Raymond. *Efficient Compilation of a Declarative Synchronous Language:the Lustre-V3 Code Generator*. PhD thesis, Grenoble Institute of Technology, France, 1991.

[40] Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, and Fabienne Carrier. Timing analysis enhancement for synchronous program. In *21st International Conference on Real-Time Networks and Systems (RTNS'13)*, pages 141–150, 2013.

[41] Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Fabienne Carrier, and Mihail Asavoae. Timing analysis enhancement for synchronous program. *Real-Time Systems*, 51(2):192–220, 2015.

[42] Hamza Rihani, Matthieu Moy, Claire Maiza, Robert I. Davis, and Sebastian Altmeyer. Response time analysis of synchronous data flow programs on a many-core processor. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, pages 67–76, 2016.

[43] RTCA, Inc. DO-178 ED-12B - Software Considerations in Airborne Systems and Equipment Certification, 2008.

[44] SAE. Aerospace Recommended Practices ARP4754a - development of civil aircraft and systems, 2010. SAE.

[45] Christos Sofronis, Stavros Tripakis, and Paul Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 21–33, 2006.

[46] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In *Proceedings of Formal Methods (FM'2009)*, pages 532–546, 2009.

[47] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.

[48] Reinhard Wilhelm and Jan Reineke. Embedded systems: Many cores - many problems. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 176–180, 2012.

[49] Eugene Yip, Alain Girault, Partha S Roop, and Morteza Biglari-Abhari. The forec synchronous deterministic parallel programming language for multicores. In *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2016 IEEE 10th International Symposium on*, pages 297–304. IEEE, 2016.