Correct-by-Construction Parallelization of Hard Real-Time Avionics Applications on Off-the-Shelf Predictable Hardware

KERYAN DIDIER and DUMITRU POTOP-BUTUCARU, Inria, France

GUILLAUME IOOSS and ALBERT COHEN, Inria and ENS, France

JEAN SOUYRIS, Airbus, France

PHILIPPE BAUFRETON, Safran, France

AMAURY GRAILLAT, Kalray and Université Grenoble-Alpes, France

We present the first end-to-end modeling and compilation flow to parallelize hard real-time control applications while fully guaranteeing the respect of real-time requirements on off-the-shelf hardware. It scales to thousands of dataflow nodes and has been validated on two production avionics applications. Unlike classical optimizing compilation, it takes as input non-functional requirements (real time, resource limits). To enforce these requirements, the compiler follows a static resource allocation strategy, from coarse-grain tasks communicating over an interconnection network all the way to individual variables and memory accesses. It controls timing interferences resulting from mapping decisions in a precise, safe, and scalable way.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; • **Software and its engineering** → **Real-time systems software**; **Data flow languages**; **Compilers**;

Additional Key Words and Phrases: Parallelization, timing analysis, static scheduling, code generation

ACM Reference format:

Keryan Didier, Dumitru Potop-Butucaru, Guillaume Iooss, Albert Cohen, Jean Souyris, Philippe Baufreton, and Amaury Graillat. 2019. Correct-by-Construction Parallelization of Hard Real-Time Avionics Applications on Off-the-Shelf Predictable Hardware. *ACM Trans. Archit. Code Optim.* 16, 3, Article 24 (July 2019), 27 pages. <https://doi.org/10.1145/3328799>

24

1 INTRODUCTION

Full automation is possible and needed in real-time scheduling. The implementation of complex embedded software relies on two fundamental and complementary engineering disciplines: real-time scheduling and compilation. Real-time scheduling covers¹ the upper abstraction levels of the implementation process, which determine how a *functional specification* is transformed into

¹ Along with other disciplines such as operating systems, software engineering, and so on.

This work was partially funded by the ITEA3 ASSUME and PIA ES3CAP projects.

Authors' addresses: K. Didier and D. Potop-Butucaru, Inria, France; emails: {keryan.didier, dumitru.potop}@inria.fr; G. Iooss and A. Cohen, Inria and ENS, France; emails: {guillaume.iooss, albert.cohen}@inria.fr; J. Souyris, Airbus, France; email: jean.souyris@airbus.com; P. Baufreton, Safran, France; email: philippe.baufreton@safrangroup.com; A. Graillat, Kalray and Université Grenoble-Alpes, France; email: amaury.graillat@univ-grenoble-alpes.fr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/07-ART24

<https://doi.org/10.1145/3328799>

a set of *tasks* and how the tasks are mapped and scheduled onto the resources of the *execution platform* in a way that ensures functional correctness while respecting *non-functional requirements*. By comparison, compilation covers the low-level code generation process, where each task (usually a piece of sequential code written in C, Ada, etc.) is transformed into machine code, allowing actual execution.

In the early days of embedded systems design, both high-level and low-level implementation activities were largely manual. However, two factors led to rapid automation at low-level: the increasing amount and complexity of software, and the standardization of both general-purpose programming languages and instruction set architectures (ISAs) of execution platforms.

At the high level, many activities remained largely manual for a long time. Such is the case for the partitioning of the application into sequential tasks, the production of *glue code* ensuring task orchestration, communication, and synchronization, or even timing analysis, where adding experience-based margins to computed worst-case execution time (WCET) estimates is still commonplace. This lack of automation can be attributed to two factors: (1) the lack of standardization in execution platforms and in functional and non-functional modeling languages made the construction of (qualified) tools expensive and inefficient; (2) penalties associated with the lack of automation were acceptable on low-complexity systems featuring few processors and tasks, and often featuring tasks that require little synchronization [6].

Both factors are now gone. Languages for functional modeling of control applications such as Simulink [51], Scade [41], or LabView [34], and languages for non-functional specification such as SysML [56] or UML/MARTE [38] are standard practice in industry. There are also well-established official standards like those describing execution platforms in avionics—ARINC 653 [2]—and automotive industries—Autosar [4]. Furthermore, the rapidly increasing complexity of execution platforms and software leads to prohibitive development costs for manual processes.

Fully automating the mapping is difficult. The key difficulty of real-time scheduling is that timing analysis and resource allocation depend on each other. This difficulty is particularly acute in the case of safety-critical control systems where certification regulations do not tolerate the weakening of timing characterizations in multi-processor environments; this is the context of our work.

An exhaustive search for the optimal solution not being possible for complexity reasons, heuristic approaches are used to break this dependency cycle. Two such approaches are typical in real-time systems design.

The most common is to first build the system, and then check the respect of real-time requirements through a global analysis. Building the system uses unsafe timing information² such as measures, WCETs in isolation plus arbitrary margins, and the like. The second approach is to ensure *by construction* the respect of requirements. In this case, system construction uses task timing characterizations that are safe for all possible resource allocations (worst-case bounds). The drawback of the first approach is the lack of traceability between resource allocation decisions and timing analysis results. If the system does not respect its real-time requirements, mapping changes are needed, but these changes may also change the timing analysis, and so on, without guarantee of convergence to a solution. The second approach is much more appealing from an automation perspective and considering the fine-grained control it offers to platform engineers; its drawback is pessimism, as all resource allocations are made for the worst case.

So far, the practicality of the second approach has never been established. Automated real-time parallelization flows still rely on simplified hypotheses ignoring much of the timing behavior of concurrent tasks, communication, and synchronization code. And even with such unsafe

²This is similar to classical compilation, where the timing models used for software pipelining or VLIW instruction scheduling are not meant to provide worst-case timing bounds, but average-case optimization figures on chosen benchmarks.

hypotheses, few studies and tools considered the—harmonic—multi-periodic task graphs of real-world control applications, and the problem of statically managing all their computational, memory, synchronization, and communication resources.

Contributions. We present the first demonstration of the feasibility of the second approach, showing good practical results for classes of real-world applications and off-the-shelf multi-core hardware whose timing predictability allows keeping pessimism under control. This requires something that is missing in previous work: *the tight orchestration of all implementation phases*: WCET analysis, resource allocation, generation of *glue code* ensuring the sequencing of tasks on cores, and the synchronization and memory coherency between the cores, compilation, and linking of the resulting C code. This orchestration is conducted on a very detailed timing model that considers both the tasks and the generated glue code, and which includes resource access interferences due to multi-core execution. While orchestration is our main contribution, it should not be understood as a mere combination of existing tools and algorithms. The whole point of our approach is to carefully coordinate every analysis, mapping, and code generation phase to enable predictable execution and to keep pessimism under control. To this end, we contributed in this article:

- *an application normalization phase*, presented in Section 3.1, which facilitates the mapping of multi-periodic applications through hyper-period expansion, and which enforces a clear separation of inter-node data flow from internal side-effecting computations, enabling to defer all memory allocation decisions to a coordinated memory management and scheduling phase;
- *an original code generation algorithm*, presented in Sections 4 and 5.3, including the generation of the synchronization code; it has been designed to provide mapping-independent worst-case execution time bounds, inaccessible to previous semaphore-based code generation approaches;
- *new real-time scheduling algorithms* capable of orchestrating memory allocation and scheduling, presented in Section 5; the main originality is to provide a safe accounting of memory access interferences through an incremental allocation and scheduling process; these algorithms are scalable, and they rely on list scheduling heuristics inspired from classical compilation.

Besides these technical contributions, we believe this article makes a strong case for a design automation approach combining the strengths of real-time scheduling and optimizing compilation. We validate the benefits of such a combined approach on two production avionics applications. This experimental validation stands out as a milestone in terms of the scale of these applications, and in terms of effectively solving the integrated resource allocation, scheduling, and timing analysis problem on a shared-memory multi-processor.

Outline. The remainder of the article is organized as follows: Section 2 reviews the most closely related work. Section 3 defines the mapping and code generation problem, presenting the functional and non-functional modeling formalism, the target execution platform, and the desired form of the implementation. Section 4 presents the timing model at the core of our method. The mapping and code generation algorithms are presented in Section 5. Section 6 presents experimental results, before the conclusion in Section 7.

2 RELATED WORK

Much of the classical work on real-time scheduling (in both research and industry) relies on a two-phase process that clearly separates the construction of the implementation from its verification and validation (V&V) [17]. Verification activities, including determining whether the

system satisfies its non-functional requirements (a process known as *schedulability analysis*) are performed on the completed implementation. The construction of the implementation uses incomplete/unsafe/unformalized versions of the timing analysis algorithms to guide its mapping decisions and code transformations. For instance, the construction of tasks and memory allocation are often guided by potentially unsafe WCET and/or memory footprint estimations, derived from previous experience and partial code analysis. Furthermore, significant parts of the implementation process remain to this day manual or unformalized in many industrial settings.

Recent advances have largely automated the construction of task code and the generation of real-time implementations on specific sequential or multi-core targets. Industrial solutions include Simulink Real-Time from MathWorks, and Scade KCG6 Parallel from ANSYS/Esterel Technologies [41]. Academic results in this direction include Refs [22], [27], [44], and [61]. However, none of these tools provide strong schedulability guarantees when integrating multiple synthesized tasks: separate timing and schedulability analysis must be performed after synthesis. Several methods have been proposed [40, 47, 49], but they come with the shortcomings of the first approach sketched in the introduction. In particular, in the event of a global non-schedulability diagnosis, it is difficult to pinpoint its source so as to guide subsequent re-engineering efforts.

A few approaches have gone further by letting timing analysis results guide mapping and code generation under simplifying hypotheses common in real-time scheduling, e.g., assuming that task WCET values include *overheads* related to parallel/concurrent execution. Among these approaches, we cite the industrial tool Asterios Developer from KronoSafe, based on the Ψ C language [37], as well as the academic tools and toolboxes SynDEx [55], BIP [7], SchedMCore [52], Prelude [42], Lopht [14, 18], SigmaC [3, 10], the work on the time-triggered mapping of Lustre [15], Xoncrete [11], or the work of Baruah et al. on the synthesis of multi-core cyclic executives [23]. We defer the reader to Ref. [14] for a longer description.

While these methods guarantee correctness and have the potential of providing more feedback in case of non-schedulability diagnostics, the simplifying hypotheses are seldom (if ever) satisfied in practice. To our knowledge, they are satisfied only when using prototype hardware designed for predictability, e.g., by ensuring the absence of memory access interferences [12, 43]. But the hypotheses are never satisfied on modern off-the-shelf multi- and many-cores, where the overheads due to concurrent execution include contributions that may be difficult to estimate, depending on the hardware or software architecture of the system: memory access and bus access interferences, cache-related delays, synchronization costs, scheduler execution time, etc.

A step further is taken in Refs [50] and [53], where the tool itself adds the needed overheads to the WCET values. In Ref. [53], overheads are large (several hundred cycles per task), to account for the time-triggered execution mechanism where so-called monitors are used to dynamically update triggering dates. In Ref. [50], overheads are not even discussed, and no comparison with the sequential case without communication costs is given. More important, in both cases, the objective of the method is optimization, not implementation under constraints, as it is in our case.

Our parallelization method also adds the needed overheads to WCET values. However, unlike in Ref. [53], it is aimed at applications with fine-grain parallelism like our case studies, where excessive per-task overheads would result in significantly reduced parallelization potential. To keep overheads under control, we make strong hypotheses on the target execution platform, on the form of generated code, and on the integration of the various tools of the back-end. These hypotheses allow our tool-flow to *perform a full-fledged timing and schedulability analysis incrementally during allocation and scheduling*. Allocation and scheduling are performed jointly, using scalable compilation-like heuristics. The resulting schedule and code are correct by construction. If construction of the schedule is impossible, the partial mapping and schedulability analysis allow the engineer to pinpoint the immediate causes of the scheduling failure.

This article does *not* provide advances on the complexity of real-time scheduling algorithms. Recent papers provided conflicting evidence—pro [20, 45] and contra [26]—on the ability of methods based on constraint solving to find solutions to large-scale real-time scheduling problems. Like most others, the scheduling problem we address can be encoded as an integer linear programming (ILP), satisfiability modulo theories (SMT), or constraint program. However, we consider parallelization, real-time scheduling, memory allocation, and safe and precise timing analysis for very large-scale applications going much beyond the most complex constraint programming and complexity studies. Furthermore, we consider the use of low-complexity mapping heuristics a positive point, as it *guarantees* scalability.

Our compilation problem is similar to that solved by a compiler and linker flow for a sequential imperative language: to produce *correct executable code* that statically orchestrates the machine resources, in a *fully automatic* and *scalable* way. But there are also important differences. Following long-standing practice in the avionics industry [28], the input program—also known as the *functional specification*—is provided in a dataflow synchronous language with a *cyclic execution scheme*; we use the Heptagon language [24, 29], introduced in Section 3.1. Also, the target low-level semantics is multithreaded with explicit resource allocation and mapping for communication and synchronization. Furthermore, as the example of Figure 1 shows, this program can be annotated with *non-functional requirements* the implementation must respect. In our example, annotations specify *real-time* requirements (period and deadline). The programming language and the non-functional annotations will be presented in Section 3.1.

ACM Transactions on Architecture and Code Optimization, Vol. 16, No. 3, Article 24. Publication date: July 2019.


```

void* thread_cpu0(void*){
    lock_init_pe(0); init();
    for(;;){
        barrier_reinit(2);
        time_wait(3000);
        barrier_sync(0);
        dcache_inval();
        f(1,&x);
        dcache_flush();
        unlock(1);
        lock(0,0);
        dcache_inval();
        h(x,y,&z);
        dcache_flush();
    }
}

void* thread_cpu1(void*){
    lock_init_pe(1);
    for(;;){
        barrier_sync(1);
        dcache_inval();
        g(z,&y);
        dcache_flush();
        lock(1,1);
        unlock(0);
    }
}

```

Fig. 2. Parallel C code generated from the Heptagon program in Figure 1 for a two-core implementation.

Fig. 3. Proposed implementation flow—tools and artifacts.

and h , by the communication and synchronization library, and by a linker script enforcing memory allocation (provided in Figure 5).

To ensure that generated code is not only functionally correct, but also satisfies *by construction* the real-time requirements, we rely on the compilation flow of Figure 3. The front-end normalizes and simplifies the input program, bringing it to a form that satisfies the requirements of *static single assignment* (SSA) form [21]. In the back-end, the sequential code of the basic dataflow blocks (f , g , h in our example³) is separately compiled and analyzed to determine their WCET, worst-case number of accesses to shared communication resources (memory banks), and memory footprint. This information is used in the parallel back-end, which performs real-time resource allocation and code generation, building the parallel threads of Figure 2 and the linker script of Figure 5.

Of this compilation flow, several components have been extensively studied in previous work: the compilation of dataflow synchronous programs to sequential code [9], C compilation [39], and WCET analysis [58]. This article focuses on the remaining topics: the front-end normalization phase in Section 3.1, the parallel back-end in Section 5, and the integration of all back-end tools

³I/O is performed through shared variables, so `read_int` and `write_int` require no code in Figure 2.

around the timing model of Section 4, which guarantees *by construction* the respect of real-time requirements.

3.1 Platform-independent Modeling

In safety-critical avionics, the use of synchronous languages is meant to facilitate both the specification and the implementation of systems [28]. One key advantage of these languages is the *deterministic* execution model where *concurrency* is permitted inside bounded *execution cycles*, whereas the cycles are strictly sequenced in time. The *de facto* industry standard is the proprietary language Scade [28, 41] from ANSYS/Esterel Technologies, itself an evolution of Lustre [16]. Code fragments in this article use the syntax of the open-source Heptagon dialect of Lustre [9, 29], which implements many of the features of the Scade 6 language and is natively used by our tools [24].

3.1.1 The Heptagon Language. Heptagon allows the functional description of systems in a *hierarchical dataflow* style. The programming unit of Heptagon is the dataflow *node*, which has *inputs and outputs* and a (possibly empty) *internal state*. The execution of a node is *cyclic*. At each cycle, the node reads its inputs and internal state and computes the value of outputs and the state of the next cycle. The code of a node can be either implemented directly as dataflow equations built of dataflow primitives and instances of other nodes, or provided externally in C following a well-defined programming interface.⁴ In Figure 1 (left), node *simple* has one input *i* and one output *o*. It is defined as the dataflow composition of three nodes (*f*, *g*, and *h*) and the dataflow primitive *fby* introduced below. Nodes *f*, *g*, and *h* are externally defined, their interface being declared in the include *Func*. Recursion is not permitted in the construction of the dataflow hierarchy—a node cannot be instantiated in its own definition, either directly or through the instantiation of other nodes. This article introduces several extensions to Heptagon to expose parallelism and convey real-time requirements.

3.1.2 Exposing Parallelism. Classical work in real-time scheduling makes a clear distinction between two specification and programming levels:

- Components meant to become sequential code, which are usually known as *tasks* or *runnables*.
- The system-level specification defines how tasks/runnables interact, exposing the *potential parallelism* that can be exploited during real-time mapping.

A synchronous language like Heptagon (and Lustre, Scade) does not make this distinction. Its naturally concurrent programming style allows to combine both levels as a single dataflow hierarchy going all the way from system level to low-level machine operations.

However, expressing parallelism of too fine a grain generally proves unprofitable for performance and unsuitable to harness in a real-time embedded context. This is primarily due to the synchronization and runtime scheduling overheads, and also to limitations of the timing analysis and parallelization algorithms. For this reason, we need a mechanism to specify which part of the parallelism of a Heptagon program is exposed to the parallelization algorithms.

We shall make the convention that parallelization algorithms only exploit the parallelism of the topmost node of the application and of the nodes that are *inlined* into it. Inlining is specified with a specific keyword. In Figure 1 (left), the node *simple* is inlined into node *main* (the topmost node of the Heptagon program), meaning that the dataflow exposed to parallelization algorithms is the

⁴The legacy task code of Figure 3 is provided in this way.

one pictured at right. We call this part the *integration program*. All other nodes (in our case f , g , and h) are compiled to sequential code.

As the integration program corresponds to a system-level specification, it is required that it has no inputs and outputs. Input and output operations are performed by some of the node instances, which sample hardware devices. In Figure 1, `read_int` and `write_int` are dedicated I/O functions, declared in library `Io`. Their implementation only requires that variables `i` and `o` are placed at specific addresses in memory.

Other ways of exposing parallelism exist. In Refs [19] and [41], the convention is that compilation is sequential except for groups of nodes (identified through program annotations) that are executed in a fork/join fashion. This may be better adapted to exposing (regular) data parallelism, but seemed less adequate to represent the unstructured, fine-grained parallelism of our two case studies. In addition, note that these works do not model real-time requirements.

3.1.3 Real-time Requirements. Heptagon is a functional programming language for synchronous, real-time reactive systems. *Non-functional requirements* constrain how and when the computations of the Heptagon program are executed. We specify them through *annotations* (in red in Figure 1) that do not change the functional semantics of the Heptagon program. We have already introduced the inlining annotation. We shall use three more annotations to represent real-time requirements: period, release date, and deadline. This choice of real-time requirements is not new [14]. The only novelty here is that we represent the requirements with annotations of the Heptagon program. The Heptagon language, extended with these four annotations, allows us to represent the *platform-independent specification* of our system.

The *period* of the system is represented with an annotation of its topmost node. We require that all the computations and communications of cycle n of this node (including all computations and communications of nodes it instantiates) are executed after date $n \times p$ and before date $(n + 1) \times p$.⁵

Release date and *deadline* requirements can be set on each of the statements of the integration program. If the period of the specification is p and if the release date and the deadline of a statement are, respectively, r and d , with $0 \leq r < d \leq p$, then the execution of the statement inside cycle n must happen between dates $n \times p + r$ and $n \times p + d$. When applied to an inlined node instance, these requirements are transferred onto all the statements of the inlined node.

The timing requirements do not specify the time unit (ms, μ s, CPU cycle, etc.). It is the task of the engineer to ensure that all values use a consistent unit. In Figure 1, the application period is 3,000 time units, and node f has a deadline constraint of 1,500 time units.

3.1.4 Exposing Node States. The state of a node consists in all values passed from one cycle to the next. The state holder primitive is `fby`, used in line 9 of our example. The primitive produces on its unique output the value it acquired on its unique input *in the previous cycle*. During the first execution cycle, `fby` outputs the initialization constant (0 in our example). The state of node `simple` consists in the state of its `fby` statement, plus the state of all nodes it instantiates (in our case f , g , h).

When the state of a node is empty, we say that it is a *dataflow function*. It is always possible to transform a stateful node instantiation into a dataflow function instantiation. The transformation is done by exposing the node state using dataflow variables and an `fby` primitive. Assume the original node instantiation is: $(o_1, \dots, o_k) = f(i_1, \dots, i_n)$. To perform the transformation, we need to explicitly define three Heptagon objects: the type of f 's state, denoted `f_state`, the initial state of f denoted `f_init` (of type `f_state`), and the function `f_step`, whose signature extends that of f with one input and one output of type `f_state`. This function, called the *transition function* of f ,

⁵This definition can be extended to allow the pipelining of cycles following the approach of Ref. [13].


```

1  period(100) node main () returns ()
2  var
3  x,y,z,c : int ; cb : bool ;
4  let
5  c = 0 fby ((c+1)%2); cb = (c=1); (* Modulo 2 counter *)
6  x = m(z) ;
7  y = n(x when cb) ;
8  z = 0 fby (merge cb (true -> y)
9             (false -> z whenot cb)) ;
10 tel

period(200) node main_norm () returns ()
var
x1,x2,z : int ;
s1,s2 : m_state ; s3 : n_state ;
let
deadline(100)
(s1,x1) = m_step(m_init fby s2,0 fby z) ;
release(100) (s2,x2) = m_step(s1,0 fby z) ;
release(100) (s3,z) = n_step(n_init fby s3,x2) ;
tel

```

Fig. 4. Multi-period specification using periodic activation conditions (left). Result of normalization (right).

computes the new state and outputs of f starting from the current state and inputs, but does not need an internal state because the state is explicitly represented with an input and an output. The instantiation of f can then be replaced with:

$(s, o_1, \dots, o_k) = f_step(ps, i_1, \dots, i_n) ; ps = f_init \text{ fby } s ;$

The compilation of various Lustre dialects to sequential code [9] always builds these three objects, but only at C level. What we do is move this transformation up to dataflow level, as a source-to-source transformation. Resulting C code preserves its efficiency.

Transforming all node instantiations of the integration program into function calls has the advantage of exposing all data memory used by the application under the form of dataflow variables and fby primitives. We can then perform memory allocation at the level of the dataflow semantics of Heptagon.

3.1.5 Hyper-period Expansion and Normal Form. The Heptagon language provides conditional control constructs allowing the representation of complex execution modes, data-dependent control, and multi-period execution. We assume in this article that multi-period specifications come under the form of Heptagon programs like the one in Figure 4 (left), where (strictly) periodic activation is represented with periodic counters. In this integration program, nodes n and m are, respectively, executed with periods 200 and 100. The period of the topmost node being 100, the multi-period behavior is achieved by using the Boolean variable cb to ensure that n is executed every other cycle. Line 5 ensures that the value of cb alternates between false and true. Expression “ x when cb ” is present with the value of x only when cb is present and true, so that n receives input and executes only in cycles where cb is present and true. The over-sampling allowing communication of values from n to m is achieved in lines 8 and 9 using operator `merge`. For more information on the syntax and semantics of the language operators, readers are deferred to [9, 29].

Note the high complexity of the sub- and over-sampling operations (even for a Heptagon program defining only two tasks), and the fact that the execution of n is subject to the implicit release date and deadlines imposed by the period of 100, meaning that its execution cannot take more than 100 time units, even though it is executed with a longer period. For these reasons, such a presentation of a multi-period specification may not be a good input for mapping algorithms. This is especially true when scheduling is performed offline and its output is a scheduling table whose length is the *hyper-period* of the system, i.e., the least common multiple of the node periods. In this case, common in industrial contexts, we believe that a better model of the application can be obtained by “unrolling” the cycles of the integration program over the length of the hyper-period (twice for the example in Figure 4), and then simplifying the expression of the periodic activations and sub- and over-samplings. We call this transformation *hyper-period expansion*.

Hyper-period expansion can also be accompanied by a relaxing of the release date and deadline requirements for nodes with larger periods, according to application-specific rules. For instance,

in Figure 4 (right), the release date requirement on node n could be removed (replaced with the implicit one of 0), potentially allowing its computation to take more than 100 time units.

When unrolling creates multiple instances of the same node, like for m in our example, the state of the node must be exposed as explained in Section 3.1.4, to allow its transmission between instances. We actually expose the state of all stateful nodes, which facilitates memory allocation. The result of hyper-period expansion and state exposal is the *normalized integration program* of Figure 3.

Normalized programs satisfy the properties of SSA form [21]—there are no hidden data dependencies, each variable is assigned exactly once, and every variable is defined before it is used in a cycle. Thus, it is a good starting point for optimized resource allocation (e.g., memory optimization, using register allocation techniques).

The complexity of the normalization step is low: exposing node states is linear in time and space; the ratio between the size of the output of hyper-period expansion and its input is in $O(\text{MAF}/\text{MIF})$, where major frame (MAF) and minor frame (MIF) are defined in Section 6; inlining is simpler than classical C macro expansion.

3.2 Parallel Execution Platform

In avionics systems, the respect of execution time bounds must be demonstrated for normal conditions, but the system must also be robust to errors. To ensure robustness, we rely on event-driven semaphore-based synchronization. This guarantees the functional semantics in the presence of timing errors. It also ensures some degree of runtime robustness to timing errors through scheduling elasticity.⁶ The respect of execution time bounds for normal conditions can then be achieved through tight control of scheduling, memory allocation, and synchronization [12]. This is different from time-triggered approaches [15, 20, 37], which place timing predictability first, potentially at the expense of functional determinism and robustness.

3.2.1 Hardware. The method we propose can be applied to shared memory architectures satisfying two hypotheses: (H1) processing cores allow the computation of static WCET bounds for code executed in isolation and (H2) WCET bounds in the presence of interferences on shared resources can be computed as the sum between the WCET bound in isolation plus a term depending on the amount of accesses to the shared resources. This is always possible for architectures that are fully timing compositional [60].

While the design of such architectures is an active field of research (see, e.g., Refs [12] and [43]), we are aware of only two compliant processors that are available off-the-shelf—the Kalray MPPA many-core series [33] and the Infineon Aurix TC27x tri-core series [31].

We focus in this article on the architecture of a single compute cluster of the Kalray MPPA Bostan many-core. This simplifies the definition of the timing model because: (a) there are no shared caches, (b) L1 caches have an LRU replacement policy and no hardware coherency, (c) the interconnect is a full crossbar with fair arbiters, and (d) without interferences, memory accesses take the same time regardless of core and memory bank (uniform memory access). The timing model of this architecture, defined in Section 4, can be easily extended to cover the non-uniform memory access architecture (NUMA) of the TC27x (which can also be configured to satisfy properties (a)–(c)).

A compute cluster of the Kalray MPPA 256 many-core has 16 computing cores and 2MB of static RAM divided into 16 banks of 128kB each. Each core has 32kB of 8-way set-associative

⁶One component overstepping its timing bounds can, under certain conditions, be compensated by other components executing faster than provisioned.

Fig. 5. Memory organization and part of the linker script for the code in Figure 2. Memory organization for execution on two cores of the Kalray MPPA256 compute cluster, *prior to node mapping* (a). Blue space is available for allocation to node code and local data and to dataflow variables. Allocation of thread code and data (b), function code and data (c), and dataflow variable (d).

instruction cache and 8kB of 2-way set-associative data cache. Special hardware devices allow hardware synchronization of very low latency between the computing cores (a few clock cycles) without the need to perform memory sampling, as classical multi-core spinlocks do.

3.2.2 Software Organization and API. To comply with industrial requirements, our tool flow produces *statically scheduled, statically allocated, bare metal implementations*. The code we produce is showcased in Figures 2 and 5. Each CPU is assigned one sequential thread—a function that never terminates and that is never preempted. This function consists of an *initialization section* followed by an infinite loop. A global barrier synchronizes the starts of the loop bodies for all CPU threads, so that execution advances in lockstep on all CPUs. Each iteration of the loop bodies performs one execution cycle of the normalized integration program. In Figure 2, the global synchronization code of both threads is contained in the yellow box. The initialization section sets the state variables and semaphores to their initial values.

Real time. In Figure 2, the `time_wait` call before the synchronization barrier of thread 0 is traversed exactly 3,000 time units after either the initialization of the program or the previous call to `time_wait`. It ensures that the synchronization barrier is traversed with the period prescribed by the integration program of Figure 1. Calls to `time_wait` can also be used inside the loop body to enforce release date requirements (not present in our example). The compilation process presented in this article ensures that control always reaches a call to `time_wait` before the specified timeout elapsed. In other terms, *no deadline is missed*. In contexts where the execution platform cannot provide static timing guarantees or when explicitly required to do so (e.g., for certification purposes), calls to `time_wait` can be used to enforce deadline requirements of the integration program. In this case, the implementation of `time_wait` must be extended with code that detects and handles deadline misses.

Locks. Thread synchronization during one loop iteration is performed using a fixed set of *locks*. They are simplified versions of the POSIX or C++11 mutexes [8] that can be given a *time-predictable* implementation using specialized hardware. When a thread calls `unlock(l)`, the state of lock `l` becomes true. A call to `lock(l, c)` waits until the state of `l` is true and then changes its state back to false. The `c` argument identifies the requesting CPU to avoid obtaining it at runtime. Like in C++11, behavior is undefined when calling `unlock` on an already true lock, and the choice of thread to unlock is not specified when two or more lock calls are active on the same lock. Our compilation process will ensure that these two conditions never occur.

Barriers. We do not use synchronization to isolate computation from communication phases, either in the execution of the whole system (as in approaches based on the bulk synchronous parallel (BSP) model [41, 57]) or in that of individual nodes (as in Ref. [49]). Doing so would enforce space/time isolation during computation phases, which largely facilitates timing analysis. However, on embedded platforms and on the Kalray, many-core memory is in short supply, and isolation requires that each thread has its own memory space containing a copy of all variables it uses. This would lead to significant memory replication, conflicting with our optimization objectives. Another approach providing the same isolation effect would be to ensure that the execution of each node fits into the processor cache. In this case, memory replication is not needed, and the communication phases consist of cache operations alone [44]. However, when user-specified nodes do not fit into the cache (like in our case studies), heavy modifications are needed in the C compiler to automatically slice the original functions into smaller code intervals. This poses a problem in avionics contexts, because it would require not only the development, but also the qualification of the modified C compiler (a long and costly process). For this reason, we do not do it.

We only use global synchronization barriers to enforce the separation between successive iterations of the control loop, using primitives `barrier_reinit` and `barrier_sync`.

Memory coherency. To allow shared memory communication on platforms without hardware coherency, we need to ensure coherency through software. This is done using two functions: `dcache_inval` invalidates the content of the data cache, so that the following reads take their value from the shared RAM; `dcache_flush` forces the writing of the write buffer contents into the shared RAM before giving control in sequence. This completes the definition of the platform application programming interface (API), which is formed of seven API *primitives*: `time_wait`, `dcache_inval`, `dcache_flush`, `lock`, `unlock`, `barrier_reinit`, and `barrier_sync`.

Memory allocation. It is fully static, specified using linker scripts like those in Figure 5. The allocation of dataflow functions and variables is an output of the mapping algorithms defined next. The allocation of threads, thread stacks, system code, and pre-allocated (e.g., I/O) data is decided prior to scheduling so as to reduce interferences.

Note in Figure 5 that the code, data, and stack of the two threads are allocated on different memory banks (to reduce interferences). When the number of threads is larger than 12, allocation on separate banks is no longer possible. In this case, multiple threads must be allocated on the same memory bank.⁷

Also note that, unlike in other approaches [19, 41, 43], communication is by shared variables alone, and no explicit memory copy is performed by thread code, for either communication or for fby implementation. This largely reduces the data memory footprint, as well as code size, for generated code. This proved to be very important on our memory-constrained evaluation platform, given the size of the use cases.

4 TIMING MODEL

The structure of the generated code, exemplified in Figures 2 and 5, has been chosen to allow the computation of tight bounds on the execution time of one cycle of the for loops running in lock-step. Each iteration of these top-level for loops implements one hyper-period of the integration program. Each loop body is formed of the global barrier code (in the yellow box in Figure 2) followed by a sequence of *code snippets*, each one corresponding to an instance of a dataflow function of the normalized integration program (the blue boxes).

⁷E.g., threads 13 and 0 on the same bank, threads 14 and 1 on the same bank, threads 15 and 2 on the same bank.

Fig. 6. The DAG of operations and dependencies for the parallel code in Figure 2. Snippets are represented with dotted boxes grouping the operations.

Each snippet contains a call to the C function⁸ implementing the dataflow function. Cache operations placed before and after the function call ensure memory coherency. Lock operations are placed at the beginning and at the end of the snippet. They are always paired to enforce order relations between specific points of different threads (the red arrows of Figure 2). One unlock operation marks the dependency start and one lock operation on the same lock marks the dependency end. The code generation process ensures that no other lock operation can access the same lock between the points in time where the unlock and lock calls corresponding to the dependency are executed.

The operations of the snippets—dataflow function calls and API primitive calls—are naturally organized as a directed acyclic graph (DAG). This DAG represents all the dependencies between operations performed during one cycle of the for loops running in lockstep (between two global barriers). The nodes of this DAG are the individual operations. Two operations o_1 and o_2 are connected by an edge $o_1 \rightarrow o_2$ if either: (1) o_2 is sequenced immediately after o_1 in the body of one of the threads, or (2) o_1 and o_2 form one unlock/lock pair encoding one of the inter-thread dependencies. We also add to this DAG one special *release operation* r per release date annotation in the integration program. This operation is connected with a single edge $r \rightarrow o$ to the first operation o of the snippet associated with the annotated dataflow node.

The scheduling and code generation process ensure that this graph is acyclic. Figure 6 presents the DAG associated with the example in Figure 2.

PROPERTY 1 (PARALLEL WCET COMPUTATION). *Assume that for each operation o of this DAG we can compute an upper bound on its duration $WCET(o)$ (for release operations, we consider instead the value of the release annotation), and that B is an upper bound on the duration of the barrier synchronization (including the call to `time_wait`). Then, according to Ref. [47], an upper bound on the duration of an iteration of the loops running in lockstep is obtained by adding B to the critical path of the DAG.*⁹

To allow parallel WCET computation, we still need to compute $WCET(o)$ for all dataflow function and primitive call o . The remainder of this section details how these values are computed.

4.1 Dataflow Function Analysis

To derive the non-functional characterization of the C functions implementing dataflow nodes, we use the WCET analysis tool *aiT* from AbsInt. The tool is the de facto industry standard for static timing analysis [59].

We use *aiT* as a black box.¹⁰ It works on binary code. To obtain the characterization of one function, its code is compiled and linked separately, but using the same conventions as for the final

⁸In the general case, the call can be guarded by an **if** statement representing conditional activation. The examples of our article do not feature conditional activation, which simplifies the presentation.

⁹From the WCET analysis perspective of Ref. [47], this DAG is the CFG of the application after adding the communication arcs, assuming that each operation has its own basic block. The fundamental difference w.r.t. Ref. [47] is that we cannot build this CFG starting from existing parallel code, because this analysis must be performed *incrementally during parallelization*.

¹⁰Readers interested into the theory and practice of WCET analysis are deferred to Refs [58] and [59].

implementation code. To simplify the presentation, we assume, for the scope of this article, that each function inlines all calls to external functions.¹¹ For a function h , the resulting object code file has exactly one code section (`.h_text`) and one data section (`.h_data`), which are allocated sequentially in memory (grouping them facilitates memory allocation).

Characterization. By using *aiT*, we obtain for each function h a characterization consisting of:

- an upper bound $WCET(h)$ on the worst-case execution time of h ;
- an upper-bound $WCSS(h)$ on the worst-case size of the stack needed by h ;
- upper bounds $WCAT(h, r)$ on the worst-case number of memory accesses to the memory regions r containing the code section, the data section, the stack, and to pre-allocated data;¹²
- the sizes $CS(h)$ and $DS(h)$ of the generated code and data sections.

Analysis and mapping conventions. The mapping algorithms perform allocation by choosing the start address of the code section (`f_ALLOC` in Figure 5(c)). To make sure that the figures we obtained with *aiT* are worst-case bounds covering all possible mappings, we perform WCET analysis with a fixed stack value computed to match (modulo cache size) the execution time one. It is also required that `*_ALLOC` values computed by the mapping algorithms are a multiple of the instruction cache line size. Furthermore, `*_ALLOC` must be a multiple of 4Ko¹³ if the code or data sections are larger than 4Ko or if pre-allocated data and the stack include addresses that overlap modulo 4Ko.

4.2 Individual Operations in Isolation

Function call operations. In the previous section, we explained how to use *aiT* to derive the WCET of each dataflow function f called by the threads. But to apply the parallel WCET computation method of Property 1, we need WCET estimations covering the whole cost of each function call operation:

$$WCET(o) = WCET(f_o) + call_WCET(f_o) + interf(o)$$

Here, f_o is the function called by o , so $WCET(f_o)$ is computed using *aiT*. The term $call_WCET(f_o)$ is an upper bound on the duration (in isolation) of the thread code that calls f_o : placing the arguments on the stack, obtaining the address of the function, branching to it, and, finally, placing the results (if any) in the locations passed by reference. The term $interf(o)$ is an upper bound on interferences from other threads, and its computation is covered in the next section.

To allow the computation of $call_WCET(f_o)$, the code that calls f_o is analyzed to produce a full non-functional characterization, like that produced using *aiT*. But *aiT* can only be applied on binary code, so it cannot be applied on the exact assembly code sequences for function call and return before mapping and code generation take place. For this reason, we rely on manual reasoning to derive the needed upper bounds for the target systems' application binary interface (ABI). The analysis starts from a description of the calling convention used by the C compiler. The resulting characterization is a function on the number, type, and order of arguments of a dataflow function. This manual analysis is done at compiler construction time, for each ABI/C compiler pair.

In our example of Figure 1, calling the C function associated with `f` requires placing on the stack the value of variable i and the pointer to x . It requires read accesses to the thread code and to the location of i , and a single write access to the location of x , at the end of the function execution.

Primitive call operations. We also need non-functional characterizations for the API primitive call operations. Given that the number of arguments is here fixed, the WCET of a primitive call

¹¹We made this *presentation* choice because it has a negligible effect on our case studies. Inlined functions are either trivial (e.g., access to an element of a vector) or used in a single place.

¹²Dataflow communication variables are taken into account in Section 4.2.

¹³The size of a way of the processor's data cache on our test platform. Changing the address of a piece of code by a multiple of this value preserves its computation time, and a similar property holds when translating data.

operation can be computed as: $WCET(o) = WCET(p_o) + interf(o)$. Here, the first term only depends on the primitive type $p_o \in \{\text{lock}, \text{unlock}, \text{inval}, \text{flush}\}$. To derive the characterization of $WCET(\text{lock})$, $WCET(\text{unlock})$, $WCET(\text{inval})$, and $WCET(\text{flush})$, we rely on a manual process (as for $call_WCET(f_o)$). For $WCET(\text{lock})$, we assume that the primitive does not need to wait.

4.3 Memory Access Interferences

Consider the operation o . Once we have a bound on its worst-case duration in isolation, the only ingredient we need to compute $WCET(o)$ is an upper bound $interf(o)$ on the interferences from other threads. In the absence of shared caches, these interferences come from the interleaving of requests at the level of the multiplexers that guard the access to memory banks.

Consider the memory bank b , and assume that operation o makes $r_o(b)$ read accesses and $w_o(b)$ write accesses to bank b , for a total of $a_o(b) = r_o(b) + w_o(b)$ accesses. We make a difference between read and write accesses, which often have different durations. On our test platform, read accesses transfer an entire cache line, keeping the bank input multiplexer occupied for $RD = 8$ hardware clock cycles, whereas write accesses only concern one word at a time and last for only $WR = 1$ clock cycle. Also assume operation t makes $a_t(b) = r_t(b) + w_t(b)$ accesses to bank b . Then, under the Round Robin arbitration policy, each memory access of o can be delayed by at most one memory access of t . Among these delays, the ones caused by read accesses of t take at most $RD = 8$ cycles, and there can be a maximum of $dr(o, t, b) = \min(a_o(b), r_t(b))$ delays of this type. Therefore, an upper bound on the global delay that t may impose on o due to accesses to b is:

$$interf(o, t, b) = RD \times dr(o, t, b) + WR \times \min(a_o(b) - dr(o, t, b), w_t(b))$$

An upper bound on the global delay that t may impose on o is $interf(o, t) = \sum_b interf(o, t, b)$. An upper bound on the delay that the code of other threads may impose on o is

$$interf(o) = \sum_{t \text{ concurrent with } o} interf(o, t).$$

This formula extends previous work on timing analysis of parallel applications [49, 50] by considering the different contributions of read and write accesses to the interference budget.

Note that the synchronizations enforced by `lock`, `unlock`, and `barrier_sync` do not exactly match the frontiers of operations. The computation of $r_o(b)$ and $w_o(b)$ must take this into account.

5 PARALLEL BACK-END: SCHEDULING AND CODE GENERATION

The parallel back-end of our tool flow has the same general functions as the back-end of a compiler for an imperative language. Starting from the intermediate representation—in our case, the normalized platform-independent program of Section 3.1.5—it performs memory allocation and scheduling, and then generates target-specific code. Like a back-end compiler for a sequential imperative language [35, 54], our back-end uses a static scheduling heuristic based on list scheduling (presented in Section 5.4) for the sake of scalability.

5.1 Schedulability vs. Optimization

But there are also major differences with respect to classical compilation. The main performance-related goal here is not to optimize some metric such as speed (throughput), memory footprint, or energy consumption. Instead, it is to produce an implementation that is functionally correct and that respects the non-functional requirements [6].

To provide schedulability guarantees, our parallel back-end must perform a *safe* accounting of non-functional properties such as real time or memory use. Safety here means that actual resource use in the implementation must never overstep the reservations made by the back-end. To this

end, the back-end maintains *worst-case* bounds on resource use that is updated at each step of the mapping process, and checked against reservation sizes. Computing safe and tight resource use bounds requires not only knowledge of the major mapping decisions (allocation, scheduling) but also tight control over code generation details such as the structure of the thread code, or C compiler optimization choices. By comparison, classical optimizing compilers provide no timing safety guarantees.

5.2 Implementation and Abstraction Issues

Like in other static scheduling approaches, resource reservations are organized in a *reservation table* (also known as *scheduling table* or *timetable*). The data structures manipulated by our scheduling algorithms are the following (in OCaml syntax):

```

1   type interval = { starti : int ; endi : int } (* time/RAM interval *)
2   type reservation_table = {
3     length : int ;                               (* size of scheduling table *)
4     inst_cpu : cguid instance_map ;               (* allocation of snippets to CPUs *)
5     inst_time : interval instance_map ;           (* time reservations of snippets *)
6     fun_ram : interval fun_map ;                 (* RAM reservations of functions *)
7     var_ram : interval var_map ;                 (* RAM reservation of variables *)
8   }
9   type valid_scheduling_state = {
10    ns : instance_map ;                           (* set of yet unmapped dataflow functions *)
11    free_ram : free_ram ;                         (* yet unallocated RAM *)
12    free_cpu_time : free_cpu_time ;               (* yet unallocated CPU time *)
13    inst_current_wcet : int instance_map ;         (* current WCET of allocated snippets *)
14    inst_wcat : wcat instance_map ;               (* computed WCAT of allocated snippets *)
15    rt : reservation_table ;                      (* reservation table *)
16  }
17  type scheduling_state = NonSchedulable | Schedulable of valid_scheduling_state

```

A reservation table is defined by its length and by the static resource reservations it makes. In our case, the length is an input to the scheduling routine and must be equal to the period of the integration program. The scheduling table describes how the resources are allocated to the various computations and communications during one generic cycle of the integration program.

We need to allocate two resources: CPU time and memory. CPU time is allocated to snippets (each containing one function call operation and some primitive calls). Recall that one function call operation is needed to implement each function instance of the integration program. Allocation of CPU time to snippets is done using the `inst_cpu` and `inst_time` fields of the `reservation_table` structure. The two fields are maps (partial functions) from function instances to, respectively, CPU identifiers and time intervals. When scheduling succeeds, they associate to each function instance, i.e., to each snippet, exactly one CPU and one time interval $[s, e]$ with $0 \leq s < e \leq l$, where l is the length of the reservation table. Each dataflow function f is allocated one memory interval, used to store all its code and local data (sections `.f_text` and `.f_data` of its compiled version). Each dataflow variable is allocated one memory interval.

Memory allocation for the boot and thread code, as well as the allocation of stacks is done *before* the mapping of dataflow function instances, with fixed-size memory reservations done on predefined memory banks, as described in Section 3.2.2 and in Figure 5.

The reservation table produced by the scheduler must safely abstract the functional and timing behavior of the actual C code. For this to be true, a table and the code associated with it must satisfy a number of *well-formedness properties*, such as the sequential use of resources¹⁴ or the respect

¹⁴Two reservations on the same processor cannot overlap in time, unless they are conditional, subject to exclusive conditions.

of the control and data dependencies. Most of these properties have been covered in previous literature, e.g., see Refs [12], [46], and [53]. We only mention two of them that are important for subsequent developments:

- Interf1.** The CPU time reservation of a function instance must be longer than the sum of the WCETs of all the operations of the associated snippet (including interferences).
- Interf2.** When two snippets access at least one common memory bank and have non-overlapping time reservations, their executions must always be sequenced, e.g., by mutex operations.

The second property implies that in the computation of $interf(o)$ (in Section 4.3), we can make the sum over t whose time reservation overlaps with that of o .

5.3 Timing Closure

The reservation table of the full application is built incrementally, using the algorithms of Section 5.4. Function instances of the integration program are considered one by one, in an order compatible with the data dependencies. When a function instance is considered, resources are allocated to it, and to all yet unallocated dataflow variables it uses. Once the mapping choices are made for an instance, function, or variable, they are never changed (there is no backtracking).

The main difficulty in this approach is to ensure the respect of property **Interf1**. When a function instance is mapped, its CPU time reservation must be chosen without knowledge of interferences from function instances yet to be mapped. Given the structure of the generated code, these instances may introduce new memory access interferences, and may change the synchronization code.

To bound the duration of synchronization code in the absence of the full schedule, we assume a particular method for synthesizing it. To each snippet, we associate two synchronization points: one at the beginning (before the `inval` operation preceding the function call), and one at the end (after the `flush` operation following the function call). All the synchronization points of all the snippets are fully sequenced using mutex operations, in a way that enforces both the dataflow dependencies between functions and the respect of property **Interf2**. In the C code, each synchronization point is translated into at most two mutex operations—one lock waiting for the completion of the previous synchronization point, and one unlock to give control to the next point. The code of Figure 2 shows the result of synchronization synthesis for our small example. Some synchronization points do not require here encoding (the start of `f` and `g`, the end of `h`). The total order between the remaining synchronization points is represented with the red arrows.

Under this code generation method, the total synchronization overhead of one snippet is up-bounded by $2 \times (\text{WCET}(\text{lock}) + \text{WCET}(\text{unlock}))$. The sequencing of synchronization points must also be taken into account at scheduling time by ensuring that the beginning and end of each snippet reservation are mutually exclusive, for a time span of $\text{WCET}(\text{lock}) + \text{WCET}(\text{unlock})$.

To break out of the circular dependency between scheduling and timing analysis, we set a bound on access interferences as a scheduling constraint called the *interference provision budget*. It is defined as a percentage of the dataflow function WCET. In the current implementation of the back-end, this percentage is the same for all dataflow functions and is an input to the scheduling algorithms (the provision input in Figure 7). Then, in accordance with Section 4, the minimal length of the time reservation made for a snippet calling function f is:

$$\begin{aligned} \text{reservation}(f, \text{provision}) = & \text{WCET}(f) \times (1 + \text{provision}) + \text{call_WCET}(f) \\ & + 2 \times (\text{WCET}(\text{lock}) + \text{WCET}(\text{unlock})) + \text{WCET}(\text{inval}) + \text{WCET}(\text{flush}) \end{aligned}$$

```

1  procedure ListSchedulingDriver
2  inputs: d:dataflow; /* integration program */
3         provision:float; /* interference provisions (% of function WCET) */
4         cpus:int; /* number of CPUs to use on the architecture (≤ 16) */
5  outputs: s:scheduling_state
6  begin
7      vs := build_init_scheduling_state(d,cpus) /* set vs.st to be a scheduling table of
8         length equal to the period of d and no reservations, vs.ns to contain all
9         function instances of d, vs.free_cpu_time to contain all CPU time,
10         and vs.free_ram to contain all free memory (in blue in Fig. 5(a)) */
11  while vs.ns ≠ ∅ do
12      fi := choose(vs.ns) /* choose the dataflow function to map */
13      vs.ns := remove(vs.ns,fi) /* remove f from the set of non-scheduled functions */
14      /* memory reservation needs - assume each function instance allocates one
15         interval, which includes yet non-allocated code and dataflow variables */
16      na := needed_align(d,vs,fi) /* alignment depends on size, cf. Section 4.1 */
17      nr := needed_ram(d,vs,fi) /* size and organization of RAM to allocate */
18      res := reservation(get_function(fi,d),provision) /* time reservation size, cf. Section 5.3 */
19      /* earliest start date depends on release date and predecessors'end */
20      sd := max(d.release[fi],max(vs.rt.fun_time[gi].endi | gi precedes fi in d))
21      /* find the first date where the function can be scheduled (if any) */
22      found := false ;
23      while ((sd+res≤d.deadline[fi]) and not found) do
24          /* attempt allocation on all CPUs and retain all valid scheduling results */
25          sres := ∅
26          /* Reservation start date must allow the scheduling of a synchronization
27             point exclusive in time with all others synchronization points */
28          sd := next_sync_free(vs,sd) /* returns -1 upon failure */
29          if sd ≥ 0 then
30              for cpu := 0 to cpus-1 do
31                  sprime := ScheduleBlockAtDateOnCPU(d,vs,fi,cpu,sd,res,na,nr)
32                  if sprime ≠ NonSchedulable then sres := sres ∪ {sprime} end
33              done
34          end
35          if sres = ∅ then
36              /* scheduling not possible at date sd, advance date */
37              sd := advance_time(sd,d,fi,vs)
38          else
39              /* scheduling possible at date sd. If sres contains multiple solutions
40                 (i.e. CPU allocations allowing scheduling), choose one using a cost function */
41              s := choose_optimal(sres,fi,cost_function) /* cf. Section 5.4, paragraph 4 */
42              found := true
43          end
44      done
45      if not found then return NonSchedulable end /* scheduling of f was not possible */
46  done
47  return s /* application scheduling was successful */
48  end procedure

```

Fig. 7. List scheduling driver pseudo-code.

In this formula, the interference budget for function f is $WCET(f) \times provision$. The choice of $provision$ is currently done using a dichotomic search.

Through the `inst_current_wcet` field of the `scheduling_state` data structure, the scheduling routine maintains at all times a safe upper bound on the execution time of all function instances that were already scheduled. These figures include memory access interferences from already scheduled function instances. Whenever a new function instance fi is mapped, `inst_current_wcet(fi)` is first computed, and `inst_current_wcet(gi)` is updated to include interferences from operations of fi for all function instances gi that have already been scheduled. It is required that all times during scheduling, the scheduling state satisfies $inst_current_wcet(fi) \leq length(rt.inst_time(fi))$. Mapping choices not respecting this requirement are rejected.

5.4 Scheduling Algorithm

The scheduling algorithm is structured as a classical list scheduling heuristic. The use of list scheduling in real-time and embedded systems is by no means new [5, 12, 36, 55], and for this reason


```

1  procedure ScheduleBlockAtDateOnCPU
2  inputs: d:dataflow; vs:valid_scheduling_state; fi:function_instance;
3          cpu:int; start:int; time_len:int; mem_align:int; mem_res:memory_reservation;
4  outputs: sprime:scheduling_state
5  begin
6      /* Check if CPU has free time interval starting at date start, of length at least
7         time_len and allowing the scheduling of the end synchronization point */
8      f_res := find_free_interval(vs,cpu,start,time_len)
9      if invalid_interval(f_res) then return NonSchedulable end
10     vs.rt.inst_cpu[fi] := cpu; vs.rt.inst_time[fi] := f_res; vs := update_free_time(vs,cpu,f_res);
11     /* Reserve a large-enough, well-aligned free interval (if possible). */
12     vs := reserve_mem(vs,cpu,fi,mem_align,mem_res)
13     if invalid_state(vs) then return NonSchedulable end
14     /* Worst-case accesses of snippet fi to various banks under given allocation, (upper bounds
15        over  $r_{fi}(b)$  and  $w_{fi}(b)$ , for all  $b$ , cf. Section 4.3) */
16     vs.inst_wcat[fi] := wcat(d,vs,mem_res,fi)
17     f := get_function(fi,d) /* get the function of the instance */
18     vs.inst_current_wcet[fi] :=
19         WCET(f)+call_WCET(f)+2*(WCET(lock)+WCET(unlock))+WCET(inval)+WCET(dflush)
20     forall gi function instance already scheduled do
21         if intervals_overlap(vs.rt.inst_time[gi],res) then
22             vs.inst_current_wcet[gi] :=
23                 vs.inst_current_wcet[gi] + interf(vs.inst_wcat[gi],vs.inst_wcat[fi])
24             /* if provisions on gi are not sufficient due to fi */
25             if vs.inst_current_wcet[gi] > length(vs.inst_time[gi]) then return NonSchedulable end
26             vs.inst_current_wcet[fi] :=
27                 vs.inst_current_wcet[fi] + interf(vs.inst_wcat[fi],vs.inst_wcat[gi])
28             /* if provisions on fi are not sufficient due to gi */
29             if vs.inst_current_wcet[fi] > length(vs.inst_time[fi]) then return NonSchedulable end
30         end
31     done
32     /* allocation and scheduling are possible */
33     return (Schedulable vs)
34 end procedure

```

Fig. 8. Function that makes the actual reservations and checks schedulability.

(as well as for space reasons), we do not present here all subroutines. We focus on the top-level routines that include the major originality points: accounting for time interferences and ensuring timing closure.

With the notations of the previous sections, the scheduling routines are presented in Figures 7 and 8. The first algorithm is the list scheduling driver that makes most mapping decisions. The second algorithm updates the scheduling state based on these decisions and determines if it is well-formed. Thus, it can be seen as a schedulability test.

The scheduling driver consists of an initialization phase, followed by a *while* loop that schedules at each iteration one function instance of the dataflow. Scheduling will fail if the scheduling of one function instance fails. At each iteration, the function instance to schedule is chosen using function choose among those whose immediate predecessors have all been scheduled. Choice is deadline-driven. For function instances with the same deadline, we choose the one with least laxity. For the example in Figure 1, the instance of *f* is chosen first due to its deadline (the choice being between *f* and *g*). Then, *g* is scheduled (no choice exists because *h* depends on *g*), and, finally, *h*.

Scheduling is performed at the earliest date possible after the release date and after the end of all predecessors. Starting at this date, scheduling will be attempted on every CPU (by the *for* loop in lines 30–33). If scheduling at a specific date fails on all CPUs, time is advanced (line 37) and scheduling attempted again until a solution is found or scheduling is no longer possible given the duration and deadline of the function instance (line 23). If for a given start date multiple CPUs allow the scheduling of the current function instance, we select one that minimizes a cost function. The selection is performed by function choose_optimal (in line 41), and the cost_function currently used is the end date of the time reservation made for the function instance based on the CPU allocation choice.

Fig. 9. Scheduling result for the specification of Figure 1.

In Figure 9, we provide the CPU time reservations produced by the scheduling algorithms for the small example in Figure 1, under the following assumptions:¹⁵ $WCET(f) = 600$, $WCET(g) = 400$, $WCET(h) = 200$, $call_WCET(f) = call_WCET(h) = 100$, $call_WCET(g) = 200$, $provision = 0.5$, $WCET(dcache_inval) = WCET(dcache_flush) = 100$, and worst-case length of a synchronization point $\delta = WCET(lock) + WCET(unlock) = 200$.

The algorithms allocate f and h on CPU0 and g on CPU1. The length of the reservation table is the period of the application (3,000). The reservation made for f spans from date 0 to date 1,600. That of g spans from 200 to 1,800, and that of h from 1,800 to 2,800. Reservation length must cover the WCET of the function (in light blue), that of the function call code (in dark blue), the interference provisions (in purple), the cache operations (in green), and the start and end synchronization points (in orange).

The allocation and scheduling of f is possible on both CPU0 and CPU1, at date 0. The two possible schedules have the same real-time properties (and, thus, the same cost), and allocation on CPU0 is retained. When mapping g , the earliest start date determined by release date requirement and predecessor dependences is computed (in line 20) as $sd = 0$. However, starting g at date 0 would mean that the start synchronization points of f and g overlap in time, which is not permitted according to Section 5.3. The call to `next_sync_free` (in line 28) finds the first date after sd allowing the mapping of a synchronization point that is non-overlapping with synchronization points of previously-mapped function instances. Recall that the length of a synchronization point is $\delta = WCET(lock) + WCET(unlock)$. Therefore, the mapping of g is first attempted at date $\delta = 200$ on all processors, through calls to function `ScheduleBlockAtDateOnCPU`.

This function takes the remaining scheduling decisions, builds the new scheduling state, and performs schedulability analysis to determine if the mapping choices allow the respect of real-time requirements. The reservation end date is chosen in line 8. The time interval must be not yet allocated, longer or equal to $reservation(f, provision)$ (which is used to initialize `time_len`), and its end (for a length of δ) must not overlap with synchronization points of previously mapped functions. Function `find_free_interval` always chooses the smallest end date ensuring this (or fails).

Ensuring that synchronization points do not overlap means that time reservations are often longer than $reservation(f, provision)$. In our example, the end synchronization point of g must not overlap with that of f , meaning that the reservation end is set to 1,800 (hence, the white space of 300 time units inside the reservation made for g). If an end date cannot be found, the function returns `NonSchedulable`. If a date is found, the scheduling state is updated with the new processor time reservation (in line 10).

In building the scheduling of Figure 9, we implicitly assumed that the interference budgets are not overstep during scheduling. If, for instance, the interferences of f on g are larger than 200, then the scheduling of g is not possible before the end of the reservation of f .

¹⁵These figures are unrealistic, but make for a clearer presentation.

Memory allocation is performed (and the scheduling state updated) in line 12. For each function instance, we reserve one memory interval, possibly of size 0, which must fit inside one memory bank. This interval must allow the allocation of the code and local data of the C function (if it has not already been allocated for another instance of the same function) and of all dataflow variables that the instance uses that were not allocated with a previous function instance. Allocation inside this interval is fixed by the call to `needed_ram` in line 17 of algorithm `ListSchedulingDriver`. The code and data of the function always come first, according to the rules of Section 4.1. If memory allocation cannot be performed, the function returns `NonSchedulable`.

The remainder of function `ScheduleBlockAtDateOnCPU` performs schedulability analysis and updates scheduling state fields that are only used to speed up this analysis (`inst_current_wcet` and `inst_wcet`). Field `inst_current_wcet` maintains along the execution, for each function instance that has already been mapped, an over-estimation of its WCET, including the worst-case interferences from other already-mapped function instances, according to the timing model of Section 4. When `ScheduleBlockAtDateOnCPU` is called on a function instance f_i , the code in lines 16–31 sets up the initial WCET estimate for f_i and updates the estimation for all already mapped instances g_i , according to the model of Section 4.3. The schedulability test is performed in lines 25 and 29 by checking that the current estimations do not become greater than the reserved time intervals, as explained in Section 5.3.

6 EXPERIMENTAL RESULTS

We evaluated our compilation flow on two large-scale avionics applications from two major companies, denoted A1 and A2 for confidentiality reasons. They are multi-period applications following a classical MAF/MIF execution pattern: the hyper-period of the application, called the *major frame (MAF)* (120ms in A1, 240ms in A2) is divided into a sequence of *minor frames (MIFs)* of equal length (5ms in A1, 15ms in A2). Thus, there are 24 MIFs in A1 and 16 in A2. Each function instance of the normalized integration program is statically confined to one of the MIFs through a combination of release and deadline requirements. A1 has more than 5,124 dataflow nodes and more than 36,000 variables before hyperperiod expansion (more than 5Mo of text and data after compilation), and 18,672 function instances after normalization. All these functions are directly instantiated in the integration program, which consists of a single, very large node. This integration program directly exposes dataflow concurrency at the top level of the integration program, so that further inlining of nested nodes is not necessary. A2 is quite different. It has a very hierarchical structure so that parallelism must be exposed through inlining. It has 4,792 function instances after normalization. The WCET of dataflow functions in A1 ranges from 37.5ns to 60.66 μ s and from 1 to 994 μ s in A2. By comparison, the total synchronization and coherency overhead (cf. Section 5.3) is less than 265ns per function instance.

The experiments with A1 and A2 have three objectives: to evaluate the scalability of the compilation toolflow, to evaluate the efficiency of the generated parallel code, and, finally, to validate the correctness of the generated code through execution on actual hardware. We validated the first two objectives on the full applications by virtually ignoring the SRAM limits of the Kalray compute cluster (1.65MB available with the system configuration available for our experiments, cf. Section 3.2.1). For this purpose, we configured the parallel back-end to assume larger memory banks. We validated the third objective on A1 alone (due to code availability constraints). Since the application is too large to fit inside the SRAM of a Kalray compute cluster, a work-around consisted in parallelizing and compiling each one of the 24 MIFs separately. Their size is already quite significant, with 690 function instances and 687ko of text and data after compilation, on average; these MIFs, when considered separately as applications, are denoted $A1_j$, with $0 \leq j < 24$. Note that we could also have considered an overlay mechanism to manage code/text memory at

Fig. 10. Compilation speed (log scale).

Fig. 11. Guaranteed speedup figures for the two avionics applications.

runtime. Such an abstraction remains challenging to implement in a hard real-time environment and is not yet available in our framework.

To evaluate *scalability*, we focused on the performance bottleneck of the compilation flow—the scheduling and allocation algorithms defined in Section 5.4. On a 4-core Intel Core i7 with 16Gb of RAM, the scheduling and allocation of the largest example (A1) on 8 (resp. 16) cores took 29.22s (resp. 42.97s), and that of A2 3.72s (resp. 3.51s). For A1₀, we plotted in Figure 10 the compilation speed as a function of the number of target cores (resulting code is unimplementable on the target platform beyond 16 cores). The graph is quasi-linear, as the scheduling algorithm attempts mapping on each core before choosing the best solution.

To evaluate the *parallelization efficiency* of our compilation flow, we must take into account the MAF/MIF organization of our case studies. For an application a parallelized on $c \geq 2$ cores with p interference provisions, *guaranteed performance* is measured as the minimal MIF duration allowing application scheduling with our tool, denoted $\min_mif(a, c, p)$. We denote with $op(a, c)$ the optimal interference provisions, i.e., the value of p that minimizes $\min_mif(a, c, p)$ for given a and c , and with $\min_mif_opt(a, c) = \min_mif(a, c, op(a, c))$. For the sequential case ($c = 1$), $\min_mif(a, 1)$ is computed as the maximum of the per-MIF sum of block WCETs.¹⁶ *Guaranteed speedup* of the parallel code with respect to the sequential reference is then defined as $gso(a, c) = \min_mif(a, 1) / \min_mif_opt(a, c)$. An upper limit on guaranteed speedup for a given application is obtained using the critical path method [30]. For each MIF m , we determine its critical path $cp(a, m)$ and then the parallelization limit of the application is computed as $pl(a) = \min_mif(a, 1) / \max_m cp(a, m)$.

The speedup figures $gso(A1, c)$ and $gso(A2, c)$ for $c = \{2, \dots, 16\}$ are provided (in blue) in Figure 11, along with the upper bounds $pl(A1)$ and $pl(A2)$ (in red). As a measure of the efficiency of the resource allocation algorithms (regardless of timing analysis precision), we also provide

¹⁶Including function call overhead, but no synchronization, cache coherency, on interference.

Fig. 12. Guaranteed speedup figures for the MIF applications $A1_j$, $0 \leq j < 24$.

(in green) the guaranteed speedup figures produced by our tool if we assume that synchronization, cache coherency, and interferences impose no overhead.

Clearly, the parallelism exposed in $A2$ is quite limited ($pl(A2) = 2.69$). Our back-end virtually reaches this limit when mapping to $c \geq 4$ processors (difference of less than 1% with respect to $pl(A2)$). This is made possible by our choice of mapping algorithm, but also by the fact that function WCETs are significantly larger than the various overheads, and that the critical path is significantly longer than other dependency paths.

$A1$ exposes significantly more parallelism at a fine grain. The guaranteed speedup is very good, coming within 12% of the theoretical limit. However, overheads come to dominate parallelization gains beyond a certain number of cores. To determine the causes of this behavior, it is interesting to consider the parallelization results on $A1_j$, $0 \leq j < 24$, in Figure 12 (left). Notice that most $A1_j$ parallelize better than the whole $A1$ does. This is normal because the results in Figure 11 must consider the worst-case among the MIFs, and because during parallelization of $A1$, the MIFs constrain one another, resulting in lower overall performance. Also note that for all $A1_j$, performance is lost beyond a certain number of parallelization cores. To provide insight into the contribution of the various overheads in this performance loss, we provide in Figure 12 (right) the guaranteed speedup for $A1_0$ (in blue), as well as the guaranteed speedup obtained if we assume that all overheads are zero (in green) or that interference overheads alone are zero (in red). Note that the green line shows almost perfect parallelization. For the full applications (in Figure 11) parallelization is almost perfect until the parallelization limit is reached. But for $A1_0$, this limit is larger than 16, so it is never reached in our experiments. The red line shows that synchronization and coherency overheads have an impact on performance, but this one still increases quasi-linearly with the number of cores.

The performance loss is therefore clearly due to memory access interferences, also known in the literature as saturation of the memory bandwidth [32]. This phenomenon is well-known in single- and multi-core scheduling, and our timing model predicts it, providing support to engineers in platform dimensioning.

To *validate* the correctness of the toolflow output, we have executed the parallel implementations of $A1_j$, $0 \leq j < 24$ on the test platform. Execution time measures were compared with the worst-case bounds provided by our tool, and measured execution time was always smaller than the worst-case bound. Note that while our real-time parallelization method is of “correct-by-construction” type, validation is by no means sufficient to guarantee the correctness of the implementations generated for real-life applications. It is done using only one application, and covers only real-time aspects and the absence of execution-time deadlocks. Significantly more validation work is needed to provide evidence that the resulting implementations can be used in real-life systems.

Fig. 13. Predicted vs. measured performance (average for $A1_j$, $0 \leq j < 24$).

We have also tried to ascertain the efficiency of the generated code from a purely measurement-based perspective. To obtain a measurement-based counterpart of gs and gso , we have measured execution time and compared it to the measured execution time of a sequential implementation. We have done this for each of the 24 tasks $A1_j$, and then taken the average. Results are provided in Figure 13 (in red), along with the statically predicted execution time (strict upper bounds of execution time, in blue). The statistical significance of these results is low because measure was performed for a single test vector for each $A1_j$.¹⁷ The performance of the code is lower than the statically predicted one, which is normal because the code was parallelized based on worst-case quantitative data. However, it remains efficient and exhibits the same pattern of performance loss due to memory bandwidth saturation.

7 CONCLUSION AND FUTURE WORK

We designed and validated the first fully automated code generation flow capable of compiling a real-world control application into a parallel implementation that is both functionally correct and respects non-functional real-time requirements *without* making simplifying hypotheses concerning the overheads related to parallel/concurrent execution on off-the-shelf hardware. In particular, our flow does *not* require adding experience-based margins to computed WCET estimates, and thus guarantees the respect of real-time requirements.

One key element of our approach consists in embedding safe and precise timing analyses into the scheduling loop, and conveying precise memory mapping and interference information throughout the compilation and analysis flow. This avoids the pitfalls of methods that first build an implementation and only then perform schedulability analysis. Achieving this requires a tight integration of analysis and synthesis steps: the normalization phase that produces an SSA-like intermediate representation, the code generation steps of the dataflow synchronous program, the back-end C compiler and binary utilities (linker and loader), all the way to the real-time parallel real-time scheduling and timing analysis. Integration ensures global consistency with respect to the timing model of the execution platform. The method provides good results on real-world applications, and is scalable. These results may percolate into industrial processes in the future, yet this may involve evolutions of the certification procedures and will certainly require major efforts in qualifying parallel-capable versions of the leading tools (e.g., Scade Suite and its KCG compiler).

From a scientific perspective, many challenges remain. The experimental evaluation emphasized the importance of evenly distributing the application load among its minor frames. This optimization problem is related to I/O latency requirements specific to each application and industrial

¹⁷Comprehensive test vector sets were only available for the whole application.

workflow, which are generally not exposed in the source model, but rather at more abstract system design levels. In the longer term, our success on one specific shared-memory architecture motivates an extension toward more complex or different hardware platforms. Extension to TC27x [31] requires taking into account the NUMA model. Extension to the full Kalray MPPA many-core and other time-predictable architectures [25, 62] requires taking into account network-on-chip (NoC) interconnects and/or shared caches (e.g., by cache partitioning). On more classical multi-cores (ARM, POWER, x86) featuring speculative out-of-order execution or hardware cache coherence, it does not seem realistic to provide static hard real-time guarantees, and different approaches with lower safety expectations should be considered.

REFERENCES

- [1] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. 1995. Software pipelining. *Comput. Surv.* 27, 3 (1995), 367–432.
- [2] ARINC653. 2010. *ARINC 653: Avionics Application Software Standard Interface. Part 1 – Required Services. Revision 3*.
- [3] P. Aubry, P.-E. Beaucamps, F. Blanc, B. Bodin, S. Carpov, L. Cudennec, V. David, P. Dore, P. Dubrulle, B. Dupont de Dinechin, F. Galea, T. Goubier, M. Harrand, S. Jones, J.-D. Lesage, S. Louise, N. Morey Chaisemartin, Thanh Hai Nguyen, X. Raynaud, and R. Sirdey. 2013. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. In *Proceedings ICCS*.
- [4] autosar [n.d.]. AUTOSAR. Retrieved March 20, 2018 from autosar.org.
- [5] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-Vincentelli. 1998. Scheduling for embedded real-time systems. *IEEE Des. Test* 15, 1 (Jan. 1998), 71–82.
- [6] S. Baruah, M. Bertogna, and G. Buttazzo. 2015. *Multiprocessor Scheduling for Real-Time Systems*. Springer.
- [7] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. 2011. Rigorous component-based system design using the BIP framework. *IEEE Software* 28, 3 (May 2011), 41–48.
- [8] M. J. Batty. 2014. *The C11 and C++11 Concurrency Model*. Ph.D. Dissertation. Wolfson College, University of Cambridge.
- [9] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. 2008. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings LCTES*.
- [10] B. Bodin, A. Munier Kordon, and B. Dupont de Dinechin. 2012. K-periodic schedules for evaluating the maximum throughput of a synchronous dataflow graph. In *SAMOS XII*.
- [11] V. Brocal, M. Masmano, I. Ripoll, A. Crespo, P. Balbastre, and J. J. Metge. 2010. Xoncrete: A scheduling tool for partitioned real-time systems. In *Proceedings ERTS*.
- [12] T. Carle, M. Djemal, D. Potop-Butucaru, R. de Simone, and Z. Zhang. 2014. Static mapping of real-time applications onto massively parallel processor arrays. In *Proceedings ACSD, Tunis, Tunisia*. DOI: <https://doi.org/10.1109/ACSD.2014.19>
- [13] T. Carle and D. Potop-Butucaru. 2014. Predicate-aware, makespan-preserving software pipelining of scheduling tables. *ACM Trans. Archit. Code Optim.* 11, 1 (Feb. 2014), 12:1–12:26.
- [14] T. Carle, D. Potop-Butucaru, Y. Sorel, and D. Lesens. 2015. From dataflow specification to multiprocessor partitioned time-triggered real-time implementation. *Leibniz Trans. Embedded Syst.* 2, 2 (2015), 01:1–01:30.
- [15] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. 2003. From Simulink to SCADE/Lustre to TTA: A layered approach for distributed embedded applications. In *Proceedings LCTES*.
- [16] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. 1987. LUSTRE: A declarative language for programming synchronous systems. In *POPL*.
- [17] M. Chetto (Ed.). 2014. *Real-time Systems Scheduling, Volumes 1 and 2*. ISTE Ltd. and John Wiley & Sons, Inc.
- [18] A. Cohen, V. Perrelle, D. Potop-Butucaru, M. Pouzet, E. Soubiran, and Z. Zhang. 2016. Hard real time and mixed time criticality on off-the-shelf embedded multi-cores. In *Proceedings ERTS2, Toulouse, France*.
- [19] J.-L. Colaço, B. Pagano, C. Pasteur, and M. Pouzet. 2018. Scade 6: From a Kahn semantics to a Kahn implementation for multicore. In *Proceedings FDL*. Munich, Germany.
- [20] S. Craciunas and R. Serna Oliver. 2016. Combined task- and network-level scheduling for distributed time-triggered networked systems. *Real-Time Syst.* 52, 2 (Mar 2016), 161–200.
- [21] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490.
- [22] P. Deng, F. Cremona, Qi Zhu, M. Di Natale, and Haibo Zeng. 2015. A model-based synthesis flow for automotive CPS. In *Proceedings ICCPS*.
- [23] C. Deutschbein, T. Fleming, A. Burns, and S. Baruah. 2017. Multi-core cyclic executives for safety-critical systems. In *Proceedings SETTA*. LNCS 10606.

- [24] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. 2012. A modular memory optimization for synchronous data-flow languages: Application to arrays in a Lustre compiler. In *Proceedings LCTES, Beijing, China*.
- [25] K. Goossens, M. Koedam, A. Nelson, S. Sinha, S. Goossens, Yonghui Li, G. Breaban, R. van Kampenhout, R. Tavakoli, J. Valencia, H. A. Balf, B. Akesson, S. Stuijk, M. Geilen, Dip Goswami, and M. Nabi. 2017. *NoC-Based Multiprocessor Architecture for Mixed-Time-Criticality Applications*. Springer Netherlands, Dordrecht, 1–40.
- [26] R. Gorcitz, E. Kofman, T. Carle, D. Potop-Butucaru, and R. de Simone. 2015. On the scalability of constraint solving for static/off-line real-time scheduling. In *Proceedings FORMATS*. Madrid, Spain.
- [27] A. Graillat, M. Moy, P. Raymond, and B. Dupont de Dinechin. 2018. Parallel code generation of synchronous programs for a many-core architecture. In *Proceedings DATE*.
- [28] N. Halbwachs. 2005. A synchronous language at work: The story of Lustre. In *Proceedings MEMOCODE*.
- [29] heptagon [n.d.]. The Heptagon/BZR distribution. Retrieved May 19, 2018 from <http://heptagon.gforge.inria.fr/>.
- [30] J. K. Hollingsworth. 1998. Critical path profiling of message passing and shared-memory programs. *IEEE Trans. Parallel Distrib. Syst.* 9, 10 (1998), 1029–1040.
- [31] Infineon [n.d.]. *AURIX™ TC27x D-Step User's Manual*. Infineon. V2.2 2014–12.
- [32] Intel 2010. *Detecting Memory Bandwidth Saturation in Threaded Applications*. Intel. Retrieved May 22, 2018 from <https://software.intel.com/en-us/articles/detecting-memory-bandwidth-saturation-in-threaded-applications/>.
- [33] Kalray S. A. 2016. *MPPA®-256 Bostan Cluster and I/O Subsystem Architecture*. Kalray S.A.
- [34] labview [n.d.]. LabView. Retrieved March 20, 2018 from ni.com/labview.
- [35] S. Larin and A. Trick. 2012. Instruction scheduling for Superscalar and VLIW platforms. Temporal perspective. <https://llvm.org/devmtg/2012-11/Larin-Trick-Scheduling.pdf>. 2012 LLVM Developers' Meeting presentation.
- [36] G. Liao, E. R. Altman, V. K. Agarwal, and G. R. Gao. 1994. A comparative study of multiprocessor list scheduling heuristics. In *1994 Proceedings of the 27th Hawaii International Conference on System Sciences*, Vol. 1. 68–77. DOI: <https://doi.org/10.1109/HICSS.1994.323184>
- [37] S. Louise, M. Lemerre, C. Aussagues, and V. David. 2011. The OASIS kernel: A framework for high dependability real-time systems. In *Proceedings HASE*.
- [38] F. Mallet and R. de Simone. 2008. MARTE: A profile for RT/E systems modeling, analysis. In *Proceedings Simutools*.
- [39] S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Academic Press.
- [40] H. Ozaktas, C. Rochange, and P. Sainrat. 2013. Automatic WCET analysis of real-time parallel applications. In *Proceedings of the WCET workshop*. Paris, France. <https://hal.archives-ouvertes.fr/hal-01239727>.
- [41] B. Pagano, C. Pasteur, G. Siegel, and R. Knizek. 2018. A model based safety critical flow for the AURIX™ multi-core platform. In *Proceedings ERTS2*. Toulouse, France.
- [42] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. 2011. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems* 21, 3 (2011), 307–338.
- [43] C. Pagetti, J. Forget, H. Falk, D. Oehlert, and A. Luppold. 2018. Automated generation of time-predictable executables on multi-core. In *Proceedings RTNS*. Poitiers, France.
- [44] R. Pellizzoni, E. Betti, S. Bak, Gang Yao, J. Criswell, M. Caccamo, and R. Kegley. 2011. PREM a predictable execution model for COTS-based embedded systems. In *Proceedings RTAS*.
- [45] Q. Perret, P. Maurère, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet. 2016. Mapping hard real-time applications on many-core processors. In *Real-Time Networks and Systems (RTNS2016)*. Brest, France.
- [46] D. Potop-Butucaru, R. de Simone, Y. Sorel, and J. P. Talpin. 2009. Clock-driven distributed real-time implementation of endochronous synchronous programs. In *Proceedings EMSOFT*.
- [47] D. Potop-Butucaru and I. Puaut. 2013. Integrated worst-case execution time estimation of multicore applications. In *Proceedings WCET*. Paris, France. DOI: <https://doi.org/10.4230/OASIS.WCET.2013.21>
- [48] B. R. Rau and C. D. Glaeser. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming, IEEE*.
- [49] H. Rihani, M. Moy, C. Maiza, R. Davis, and S. Altmeyer. 2016. Response time analysis of synchronous data flow programs on a many-core processor. In *Proceedings RTNS*.
- [50] B. Rouxel, S. Derrien, and I. Puaut. 2017. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Trans. Embedded Comput. Syst. (TECS)* 16, 5s (Oct. 2017), 1–20.
- [51] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. 2004. Defining and translating a “safe” subset of Simulink/Stateflow into Lustre. In *Proceedings EMSOFT*.
- [52] SchedMCore [n.d.]. SchedMCore toolset. Retrieved June 2017 from sites.onera.fr/schedmcore/.
- [53] S. Skalistis and A. Simalatsar. 2017. Near-optimal deployment of dataflow applications on many-core platforms with real-time guarantees. In *Proceedings DATE*.
- [54] M. Smotherman, S. Krishnamurthy, P. S. Aravind, and D. Hunnicutt. 1991. Efficient DAG construction and heuristic calculation for instruction scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO 24)*. <https://doi.acm.org/10.1145/123465.123482>.

- [55] Syndex [n.d.]. SynDEx mapping tool. Retrieved June 2017 from www.syndex.org.
- [56] sysml [n.d.]. SysML. Retrieved March 20, 2018 from <http://www.omg.sysml.org/>.
- [57] L. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111.
- [58] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. 2008. The worst-case execution-time problem — overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.* 7, 3 (2008), 36:1–36:53.
- [59] R. Wilhelm and D. Grund. 2014. Computation takes time, but how much? *Commun. ACM* 57, 2 (2014), 94–103.
- [60] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. 2009. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE TCAD* 28, 7 (2009), 966–978.
- [61] E. Yip, A. Girault, P. Roop, and M. Biglari-Abhari. 2016. The ForeC synchronous deterministic parallel programming language for multicores. In *Proceedings MCSoc*.
- [62] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. 2014. FlexPRET: A processor platform for mixed-criticality systems. In *Proceedings RTAS*.

Received November 2018; revised March 2019; accepted April 2019