Efficient parallelization of large-scale hard real-time applications

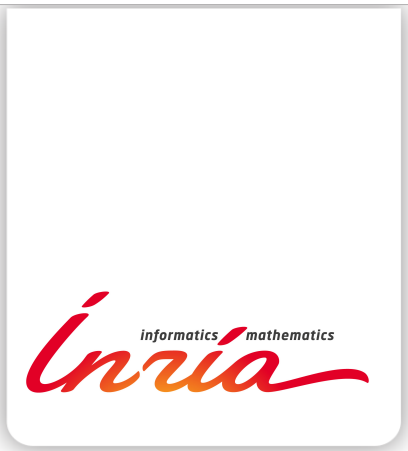
Keryan Didier, Dumitru Potop-Butucaru, Guillaume Iooss, Albert Cohen, Jean Souyris, Philippe Baufreton, Amaury Graillat

Efficient parallelization of large-scale hard real-time applications

Keryan Didier, Dumitru Potop-Butucaru, Guillaume Iooss, Albert Cohen, Jean Souyris, Philippe Baufreton, Amaury Graillat

RESEARCH REPORT
N° 9180
June 7, 2018
Project-Teams AOSTE,PARKAS

ISSN 0249-6399 ISRN INRIA/RR--9180--FR+ENG



Efficient parallelization of large-scale hard real-time applications

Keryan Didier, Dumitru Potop-Butucaru, Guillaume Iooss, Albert Cohen, Jean Souyris, Philippe Baufreton, Amaury Graillat

Project-Teams AOSTE,PARKAS

Research Report n° 9180 — June 7, 2018 — ?? pages

Abstract: We present a parallel compilation method for embedded control applications. The method is fully automatic and scales up, being based on low-complexity heuristics. Unlike classical compilation, it also takes as input non-functional requirements, e.g. real-time or resource limits. The main objective is not optimization per se, but the respect of requirements. To this end, static resource allocation and code generation algorithms perform a safe accounting of non-functional properties. Accounting starts from per-component time and memory footprint worst-case bounds, automatically obtained through calls to state-of-the-art static analysis tools. Experiments show that our method produces efficient code for large-scale, real-life avionics applications.

Key-words: Hard real-time, Compilation, Parallelization, Offline scheduling, Code generation, Timing analysis, Many-core, Lustre, Scade, Shared memory

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Parallelisation efficace de larges applications temps-reel

Résumé : Pas de résumé

Mots-clés : Pas de motclef

Contents

1 Introduction

Full automation is 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 covers[1] the upper abstraction levels of the implementation process, which determine how the *functional specification* is transformed into a set of *tasks* and how the tasks are allocated 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 (C, Ada...) 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 relying on independent tasks [**?**] that require little synchronization.

Both factors are now gone. Languages for functional modeling of control applications such as Simulink [**?**], Scade [**?**], or LabView [**?**], and languages for non-functional specification such as SysML [**?**] or UML/MARTE [**?**] are standard practice in industry. There are also well-established official standards like those describing execution platforms in avionics—ARINC 653 [**?**]—and automotive industries—Autosar [**?**]. Furthermore, the complexity of execution platforms and software leads to prohibitive development costs for manual processes. For instance, the sets of independent tasks where performance issues were mostly confined inside each sequential task are today replaced with parallelized software formed of tightly synchronized tasks. In such software, performance strongly depends on all mapping and code generation parameters (allocation, scheduling, synchronization, resource management).

Full automation of mapping is difficult The key difficulty of real-time scheduling is that timing analysis and resource allocation depend on each other. 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 first approach uses unsafe timing characterizations for the tasks (e.g., measurements) to build the system,[2] and then checks the respect of real-time requirements through a global timing analysis. The second approach uses a formal model of the hardware platform enabling

[1] Along with other disciplines such as systems engineering, software engineering, *etc.*

[2] This 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.

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

Contribution We present the first demonstration of the feasibility the second approach, showing good practical results for classes of real-world applications and multiprocessor execution platforms whose timing predictability allows keeping pessimism under control. This requires the tight integration 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. Integration is done around 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.

The approach we propose is scalable, as it relies on static mapping and code generation algorithms derived from classical compilation, and optimization heuristics based on list scheduling. We validate the approach experimentally on two large-scale avionics applications.

Outline Section **??** reviews the most closely related work. Section **??** 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 **??** presents the timing model central to our method. The mapping and code generation algorithms are presented in Section **??**. Section **??** presents experimental results, and Section **??** concludes.

2 Related work

Much of the classical work on real-time scheduling (in both research and industry) relies on a two-phase process which clearly separates the construction of the implementation from its verification and validation (V&V) [**?**]. 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 even the generation of real-time implementations on specific sequential or multi-core targets. Industrial solutions include here Simulink Real-Time from MathWorks, and Scade KCG6 Parallel from ANSYS/Esterel Technologies [**?**]. However, these tools do not provide schedulability guarantees. Separate timing and schedulability analysis must be performed after synthesis, using various methods [**?**, **?**, **?**]. In

the event of a global non-schedulability diagnosis, it is difficult to pinpoint its source so as to guide subsequent 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 ΨC language [**?**], as well as the academic tools and toolboxes SynDEx [**?**], BIP [**?**], SchedMCore [**?**], Lopht [**?**] or the work on the time-triggered mapping of Lustre [**?**]. We defer the reader to [**?**] for a survey.

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. This is especially true in modern multi- and many-core architectures, 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 [**?**], where the tool itself adds the needed overheads to the WCET values. However, overheads are here 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. Furthermore, the objective of the method is optimization, which allows decomposing the mapping problem into several successive allocation and scheduling steps solved through constraint programming.

Our parallelization method also adds the needed overheads to WCET values. However, unlike in [**?**], 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.

By covering all aspects of resource allocation and code generation, our work is clearly related to previous work on compilation. In previous work [**?**], we already noted and exploited the formal and algorithmic proximity between off-line real-time scheduling and various results on software pipelining for super-scalar and VLIW processors, where the scheduling burden is mostly supported by the compilers [**?**, **?**]. What fundamentally differentiates our current work from previous compilation work is the choice of performing a *safe*, worst-case timing analysis *incrementally during compilation.*

This paper does *not* provide advances on the complexity of real-time scheduling algorithms. Recent papers provided conflicting evidence—pro [**?**, **?**] and contra [**?**]—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 ILP, 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.

```
1  open Io (* declarations of platform-dependent I/O functions read_int, write_int *)
2  open Func (* declarations of externally-defined functions f, g, h *)
3  node simple (i:int) returns (o:int)
4  var y,d : int; x : float;
5  let
6    deadline(1500) x = f(i);
7                   y = g(d);
8                   o = h(x,y);
9                   d = 0 fby o;
10 tel
11
12 period(3000) node main () returns ()
13 var i,o : int;
14 let
15                  i = read_int();
16           inline o = simple (i);
17                 () = write_int(o);
18 tel
```

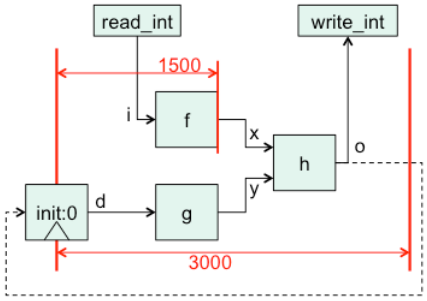


Figure 1: On the left, functional specification provided as a Heptagon program (in black) with non-functional requirements specified through annotations (in red). On the right, a graphical representation of the dataflow of node `main`, after inlining of node `simple`.

3 Problem statement and solution overview

Our compilation problem is similar to that solved by a compiler and linker flow for a sequential imperative language: it produces *correct executable code* to statically orchestrate the machine resources, in a *fully automatic* and *scalable* way. But there are also important differences. Following long-standing practice in the avionics industry, the input program—also known as the *functional specification*—is provided in a data-flow synchronous language with a *cyclic execution scheme*. Also, the target low-level semantics is multithreaded with explicit resource allocation and mapping for communication and synchronization. Furthermore, as the example of Fig. **??** 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 with its functional semantics and non-functional annotations will be presented in Section **??**.

We target shared memory multiprocessors with uniform memory access. To facilitate *timing analysis*, hardware and low-level libraries must satisfy a number of properties detailed in Section **??**.

```
void* thread_cpu0(void* unused){
  lock_init_pe(0); init();
  for(;;){
    global_barrier_reinit(2);
    time_wait(3000);
    global_barrier_sync(0);

    dcache_inval();
    f(i,&x);
    dcache_flush();
    unlock(1);

    lock(0,0);
    dcache_inval();
    h(x,y,&z);
    dcache_flush();
  }
}
```

```
1  void* thread_cpu1(void* unused){
2    lock_init_pe(1);
3    for(;;){
4
5
6      global_barrier_sync(1);
7
8      dcache_inval();
9      g(z,&y);
10     dcache_flush();
11     lock(1,1);
12     unlock(0);
13
14
15
16
17   }
18 }
```

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

For such hardware, we want to generate *statically scheduled, statically allocated, bare metal code* whose structure facilitates timing analysis [**?**]. The threads generated from our example for a dual-core target are presented in Fig. **??**. To allow compilation and execution, they must be accompanied by the boot code launching the threads, by the sequential code of the functions implementing the dataflow blocks `f`, `g`, and `h`, by the communication and synchronization library, and by a linker script enforcing memory allocation (provided later, in Fig. **??**).

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 Fig. **??**. The front-end normalizes and simplifies the input program, bringing it to a form that satisfies the requirements of *static single assignment (SSA)* form [**?**]. In the back-end, the sequential code of the basic data-flow blocks (`f`, `g`, `h` in our example[3]) 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 Fig. **??** and the linker script of Fig. **??**.

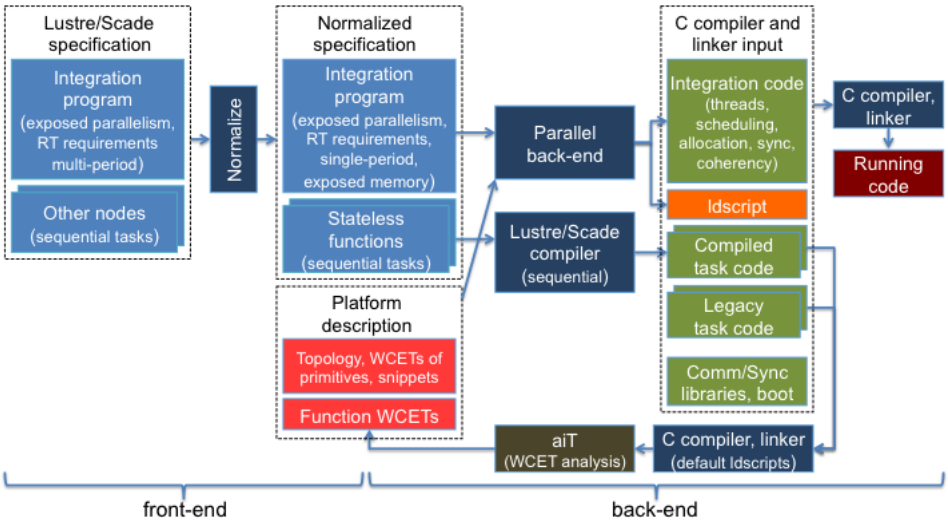


Figure 3: Proposed implementation flow—tools and artefacts

Of this compilation flow, various components have been extensively studied in previous work: the compilation of dataflow synchronous programs to sequential code [**?**], C compilation [**?**], and WCET analysis[**?**]. This paper focuses on the remaining topics: the front-end normalization phase in Section **??**, the parallel back-end in Section **??**, and most importantly, the integration of all back-end tools around the timing model of Section **??**, which allows us to guarantee *by contruction* 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. 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 [**?**] from ANSYS/Esterel Technologies, itself an evolution of Lustre [**?**]. Code

[3] I/O is performed through shared variables, so `read_int` and `write_int` require no code in Fig. **??**.

fragments in this paper use the syntax of the open-source Heptagon dialect of Lustre [**?**, **?**] which implements many of the features of the Scade 6 language and is natively used by our tools.

3.1.1 The Heptagon language

Heptagon allows the functional description of systems in a *hierarchic 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 provided directly in C, following a well-defined programming interface,[4] or provided as a dataflow built by connecting dataflow primitives and instances of other nodes through dataflow variables. In Fig. **??**(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.

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 which 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 recursively *inlined* into it. Inlining is specified with a specific keyword. In Fig. **??**(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 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 Fig. **??**, `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.

[4] The legacy task code of Fig. **??** is provided in this way.

3.1.3 Real-time requirements

Heptagon is a functional programming language for synchronous, real-time reactive systems. The *non-functional requirements* constrain how and when the computations of the Heptagon program are executed. We specify these through *annotations* (in red in Fig. **??**). We have already introduced the inlining annotation. We shall use three more annotations to represent real-time requirements: period, release date, and deadline. 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 time unit (e.g., ms, μs, CPU cycle) is not specified. It is the task of the toolflow user to ensure that all values are specified using the same unit. In Fig. **??**, the application period is 3000 time units, and node `f` has a deadline constraint of 1500 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 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 a `fby` primitive. Assume the original node instantiation is: `(`o_1`,...,`o_k`) = f(`i_1`,...,`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` (of type `f_state`) denoted `f_init`, 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`, 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,o1,...,ok) = f_step(ps,i1,...,in) ; ps = f_init fby s ;
```

The compilation of various Lustre dialects to sequential code [**?**] builds these three objects. In particular the exposure of nodes' states is always possible, and we automated it as a source-to-source transformation in our compilation flow.

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.

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

```
1  period(100) node main () returns ()
2  var
3    x,y,z : int ; c : boolean ;
4  let
5    c = false fby (not c) ; (* Modulo 2 counter *)
6    x = f(z) ;
7    y = g(x when c) ;
8    z = 0 fby (merge c (true -> y)
9                 (false -> (z when (not c)))) ;
10 tel
```

```
period(200) node main_norm () returns ()
var
  x1,x2,z : int ;
  s1,s2 : f_state ; s3 : g_state ;
let
  deadline(100)
     (s1,x1) = f_step(f_init fby s2,0 fby z) ;
  release(100) (s2,x2) = f_step(s1,0 fby z) ;
  release(100) (s3,z) = g_step(g_init fby s3,x2) ;
tel
```

Figure 4: Multi-period specification using periodic activation conditions (left). Result of normalization on the right.

3.1.5 Hyper-period expansion and normal form

The Heptagon language provides conditional control constructs allowing the representation of complex execution modes, data-depedent control, and multi-period execution. In the integration program of Fig. **??**(left), nodes `g` and `f` 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 `c` to ensure that `g` is executed every other cycle. Line 5 ensures that the value of `c` alternates between `false` and `true`. Expression "`x when c`" is present with the value of `x` only when `c` is present and `true`, so that `g` receives input and executes only in cycles where `c` is present and true. The over-sampling allowing communication of values from `g` to `f` 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 [**?**, **?**].

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 `g` is subject to the implicit release date and deadlines imposed by the period of 100, meaning that its execution cannot take more of 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, 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 Fig. **??**), and then simplifying the expression of the periodic activations and sub- and over-samplings.

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 Fig. **??**(right) the release date requirement on `g` 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 `f` in our example, the state of the node must be exposed as explained in Section **??**, to allow its transmission between instances. But we do not expose only the state of nodes with multiple instances. We do it for all nodes with non-void state. This exposes the state of all nodes under the form of variables, facilitating memory allocation. The result of the hyper-period expansion and state exposal process is the *normalized integration program* of Fig. **??**.

The normalized program satisfies the properties of Static Single Assignment (SSA) form [**?**]—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).

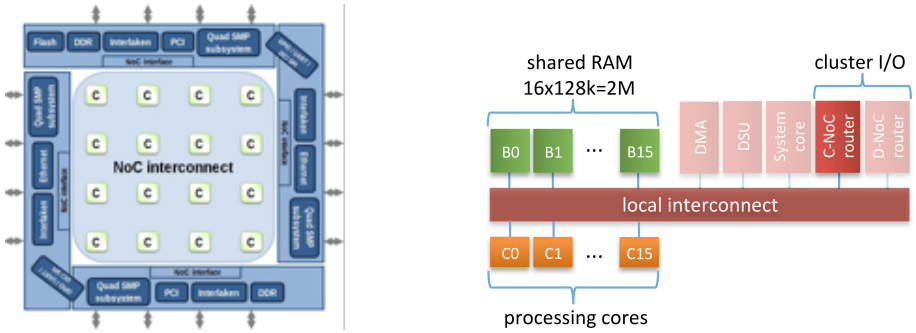


Figure 5: Kalray MPPA256 many-core. Global organization, with the 16 compute clusters figured within the NoC interconnect (left) and structure of a compute cluster (right).

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 to guarantee the functional semantics in the presence of timing errors, and to ensure some degree of run-time 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 [**?**]. This is different from time-triggered approaches [**?**, **?**], which place timing predictability first, potentially at the expense of functional determinism and robustness.

3.2.1 Hardware

The back-end of our tool flow allows code generation for shared memory architectures satisfying a number of hypotheses that facilitate timing analysis.

We require that processing cores allow the computation of static worst-case execution time (WCET) bounds for code executed in isolation. We also require that 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 [**?**].

For the scope of this paper, to simplify the presentation and allow the computation of tight WCET bounds (and thus allow efficient resource allocation), we shall make the following assumptions: a) we use only L1 non-shared caches (shared caches, if present, must be shut off); b) all caches have an LRU replacement policy; c) access from CPU cores to memory banks is done through a full crossbar interconnect (so that accesses from different processors to different memory banks do not interfere) with fair arbiters at the level of memory banks; d) no hardware memory coherency is employed.

We also require that event-driven synchronization can be done in bounded time and generate bounded interferences on other cores. The implementation used on our test platform uses a hardware semaphore implementation, whose API is defined below.

One particular architecture satisfying our requirements is that of the shared memory *compute clusters* of the Kalray MPPA 256 many-core processor [**?**], which we used for the experimental evaluations. The high-level architectural view of such a cluster is presented in Fig. **??**(right). The

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

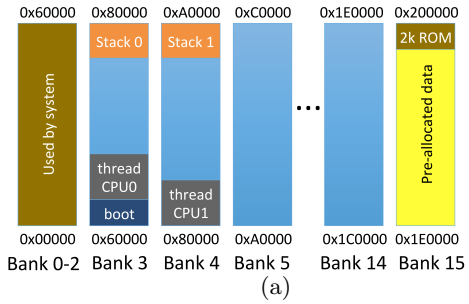
16 computing cores (C0-C15) of a cluster access memory banks (B0-B15) and memory-mapped devices through the local interconnect. Of the remaining devices (in red), our work only uses the C-NoC router, which provides the hardware semaphores. Each cluster has 2 MB of static RAM divided into 16 banks of 128 kB each. We are not interested here in cluster I/O (which should traverse the on-chip network connecting the clusters), and hence we may safely assume that the cluster interconnect is a full crossbar with fair (Round Robin) arbiters. This architecture is fully timing compositional. Its timing model is presented in Section **??**.

3.2.2 Software organization and API

Our tool flow produces *statically scheduled, statically allocated, bare metal implementations* like the one in Figures **??** and **??**. Each CPU is assigned one sequential thread—a function that never terminates and that is never preempted. This function consists in 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 Fig. **??** 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 Fig. **??**, the `time_wait` call before the synchronization barrier of thread 0 is traversed exactly 3000 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 Fig. **??**. 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 paper 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 statically fixed set of *locks*. They are simplified versions of the POSIX or C++11 mutexes [**?**] that can be given a *time-predictable* implementations using hardware devices. 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



(a)

```
. = 0x80000 ;
.text_thread1 ALIGN(64) : {
  thread_cpu1.o(.text)
}
.data_thread1 ALIGN(32) : {
  thread_cpu1.o(.data)
  thread_cpu1.o(.bss)
  thread_cpu1.o(.rodata)
}
. = 0xa0000 ;
_user_stack_start1 = .;
```

(b)

```
. = f_ALLOC ;
.f_text ALIGN(64) : {
  f.o(.text)
}
.f_data ALIGN(32) : {
  f.o(.data)
  f.o(.bss)
  f.o(.rodata)
}
```

(c)

```
x = 0x88e88;
```

(d)

Figure 6: Memory organization and part of the linker script for the code in Fig. **??**. Memory organization for execution on 2 cores of the Kalray MPPA256 compute cluster, prior to mapping (a). Blue space is available for allocation to node code and local data and to data-flow variables. Allocation of thread code and data (b), function code and data (c), and dataflow variable (d).

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.

We do not use synchronization to isolate computation from communication phases, either in the execution of the whole system (as in BSP-based approaches [**?**, **?**]) or in that of individual nodes (as in [**?**]). Doing so would enforce space/time isolation during computation phases, which largely facilites 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.

Memory coherency To allow shared memory communication on platforms without hardware coherency support, we need to ensure coherency trough software. To do this, we use two primitives that can enforce consistency between the L1 data cache of a core and the shared RAM: `dcache_inval` invalidates the content of the data cache, and `dcache_flush` forces the writing of the write buffer contents into the shared RAM before giving control in sequence.

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

4 Timing model

The structure of the generated code, exemplified in Figures **??** and **??**, has been chosen to allow the computation of tight bounds on the execution time of one cycle of the `for` loops running in lockstep. Each iteration of these top-level `for` loop implements one cycle of the integration program. Each loop body it is formed of the global barrier code (in the yellow box in Fig. **??**) 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).

Each snippet contains a call to the C function [7] 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 (one `unlock` and one `lock` operation on the same lock) to enforce order relations between specific points of different threads (the red arrows of Fig. **??**). Together with the sequencing of snippets inside the loop bodies, these explicit dependencies enforce a directed acyclic graph (DAG) structure over the snippets forming the threads.

Assume that for each snippet s of this DAG we can compute an upper bound on its duration WCET(s), and that g is an upper bound on the duration of the barrier synchronization (including the call to `time_wait`). Then, an upper bound on the duration of a loop iteration is obtained by adding g to the duration of the critical path of the DAG [**?**]. To compute this upper bound, we still need to compute WCET(s) for all s. The remainder of this section details how these values are computed.

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

4.1 Dataflow function analysis

We start by using the state-of-the-art WCET analysis tool *aiT* from AbsInt [8] to derive a non-functional characterization of the C functions implementing the functions of our dataflow.

Compilation The *aiT* tool 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 implementation code. To simplify the presentation, we shall assume for the scope of this paper that the compilation of each function inlines all calls to external functions, and that the output arguments (passed by reference to the C function) are only accessed at the end of function execution. The resulting code makes no direct reference to dataflow communication variables (which are passed as arguments). However, it may directly access pre-allocated variables representing memory-mapped I/O (in yellow in Fig. **??**(a)). Given that it concerns a single function, the linker script is similar to that of Fig. **??**(c). For a function `f` the resulting ELF file has exactly one code (`.f_text`) and one data section (`.f_data`) which are allocated sequentially in memory (grouping them facilitates memory allocation). For WCET analysis, the start of the code section (`f_ALLOC` in Fig. **??**(c)) is set to 0, as all other code can be safely ignored. The exact addresses of pre-allocated data are provided through absolute symbols.

Characterization By using *aiT* we obtain for each function f a characterization consisting of:

- an upper bound WCET(f) on the worst-case execution time of the function;
- an upper-bound WCSS(f) on the worst-case size of the stack during function execution;
- upper bounds WCAT(f, 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[9];
- the sizes CS(f) and DS(f) 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 Fig. **??**(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 `f_ALLOC` values computed by the mapping algorithms are a multiple of the instruction cache line size. Furthermore, `f_ALLOC` must be a multiple of 4Ko[10] if the `.f_data` section is larger than 4Ko, or if pre-allocated data and the stack include addresses that overlap modulo 4Ko.

4.2 Primitives and code snippets in isolation

As *aiT* can only be applied on binary code, it cannot be applied on code snippets before mapping. For this reason, the non-functional characterizations of primitives and code snippets are derived manually. For this manual analysis, we use very conservative assumptions, e.g., considering the worst-case cache configuration at every access.

An upper bound on the duration of a snippet in isolation (without interference during execution, and assuming `lock` calls never wait for their mutex) can be computed as the sum of the worst-case durations of 1) the synchronization and coherency code, 2) the called function, 3) the time needed

[8] `https://www.absint.com/ait/index.htm` Accessed on 05/19/2018

[9] Dataflow communication variables are taken into account in Section **??**.

[10] The size of a way of the processor's data cache on our test platform.

to build the context for the call to the dataflow function (placing arguments on the stack and obtaining the address of the function), and 4) the cost of placing the function results in the global variables passed by reference. Duration (1) is computed once for the platform. Duration (2) is computed using *aiT*. For the last two contributions, we derived a formula (not detailed here) that depends on the number, type, and allocation of function arguments. This formula makes strong assumptions on the function call and code generation conventions of the compiler.

4.3 Memory access interferences

Consider the snippet s. Once we have a bound on its worst-case duration in isolation, the only ingredient we need to compute WCET(s) is an upper bound on the interferences from other threads. In the absence of shared caches, these interferences can only 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 the snippet s makes $r_s(b)$ read accesses and $w_s(b)$ write accesses to bank b, for a total of $a_s(b) = r_s(b) + w_s(b)$ accesses. We make a difference between read and write accesses, which often have different durations. On our test platform read accesses are bursty, and keep 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 snippet 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 s can be delayed by at most one memory access of t. Among these delays, the ones caused by read accesses of t take at morst $RD = 8$ cycles, and there can be a maximum of $dr = \min(a_s(b), r_t(b))$ delays of this type. Therefore, an upper bound on the global delay t may impose on s due to accesses to b is:

$$\text{interf}(s, t, b) = RD \times dr + WR \times \min(a_s(b) - dr, w_t(b))$$

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

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

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

Note that the synchronizations enforced by `lock`, `unlock`, and `global_barrier_sync` do not exactly match the frontiers of snippets. The computation of $r_s(b)$ and $w_s(b)$ 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 **??**—it performs memory allocation and scheduling, and then generates target-specific code. Like a back-end compiler for a sequential imperative language [**?**, **?**], our back-end uses a static scheduling heuristic based on list scheduling (presented in Section **??**) 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 is here 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 which respects the non-functional requirements [**?**].

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 means here 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 which are 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 of code generation details such as the structure of the thread code, or C compiler optimization choices. By comparison, classical optimizing compilers rely on an *average-case* accounting of non-functional properties for optimization purposes only, which does not provide any safety guarantees, but in turn requires comparatively less integration between scheduling, timing analysis, and code generation.

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 : cpuid 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 allocate two resources: CPU time and memory. Allocation of CPU time to code snippets (which correspond to instances of dataflow functions in the integration program) 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 instance 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 **??** and Fig. **??**(a).

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 as the sequential use of resources [11] or the respect of the control and data dependencies. Most of these properties have been covered in previous literature, e.g., [**?**, **?**, **?**]. We only mention two of them, which are important for subsequent developments:

Interf1. The CPU time reservation of a function instance must cover the worst-case duration of the associated snippet WCET(s), including all memory access interferences from other cores.

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 calls). Thus, in the computation of interf(s) we can assume that they do not interfere.

5.3 Timing closure

The reservation table of the full application is built incrementally, using the algorithms of Section **??**. Function instances of the integration program are considered one by one, in an order compatible with the dataflow 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 form of the synchronization code.

To bound the duration of synchronization code in the absence of the full schedule, we assume a particular method for generating the synchronization code. To each snippet, we associate two synchronization points: one at the beginning, and one at the end. 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 Fig. **??** 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 $2 \times (\text{WCET}(\texttt{lock}) + \text{WCET}(\texttt{unlock}))$. 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 $\text{WCET}(\texttt{lock}) + \text{WCET}(\texttt{unlock})$).

To bound memory access interferences, we provision an interference budget, 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 Fig. **??**). Then, in accordance with Section **??**, the length of the time reservation made for an instance fi of function f is:

$$\begin{aligned}\text{fun\_time}(fi) = {} & \text{WCET}(f) \times (1 + \texttt{provision}/100) + \text{call\_WCET}(f) \\ & + 2 \times (\text{WCET}(\texttt{lock}) + \text{WCET}(\texttt{unlock})) + \text{WCET}(\texttt{inval}) + \text{WCET}(\texttt{flush})\end{aligned}$$

[11] Reservations on the same CPU or on overlapping memory intervals cannot overlap in time, modulo conditional execution).

where call_WCET(f) is an upper bound on the duration of code that builds the context for the call to the dataflow function. We denote with ib(fi) = WCET(f) × `provision`/100 the interference budget for fi.

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 fi (if any) for all function instance gi that has been already scheduled. It is required that all times during scheduling, the scheduling state satisfies `inst_current_wcet`(fi) ≤ length(`rt.inst_time`(fi)). Mapping choices not respecting this requirement must be rejected.

5.4 Scheduling agorithm

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 [**?**, **?**, **?**, **?**], and for this reason (as well as for space reasons) we do not present here all subroutines. We focus on the top-level routines which 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 **??** and **??**. 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 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 among those whose immediate predecessors have all been scheduled.

Scheduling is performed at the earliest date possible after the release date and the end of all predecessors. Starting at this date, scheduling will be attempted on every CPU (`for` loop in lines 26–29). If scheduling at a specific date fails on all CPUs, time is advanced (line 32) 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 scheduling, one mapping is chosen using a cost function (line 34).

While scheduling a function instance, our algorithm will 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 and which were not allocated with a previous instance. Allocation inside this interval is fixed by the call to `needed_ram` in line 17. The code and data of the function always come first, according to the rules of Section **??**.

Both the length of the CPU reservation and that of the memory reservation are set by the scheduling driver, along with the scheduling date and CPU choice. They are all passed to the routine of Fig. **??**, which makes the actual reservations and modifies the scheduling state. In the process, it checks whether reservation is possible, and whether time budgets are respected once the current node has been scheduled, as explained in Section **??**. To this end, the routine maintains the various fields of the `valid_scheduling_state` data structure.

```
1  procedure ListSchedulingDriver
2  inputs: d:dataflow; /* integration program */
3          provision:int; /* 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. ??(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 ?? */
17     nr := needed_ram(d,vs,fi) /* size and organization of RAM to allocate */
18     res := fun_time(fi,provision,d) /* time reservation size, cf. Section ?? */
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       for cpu := 0 to cpus-1 do
27         sprime := ScheduleBlockAtDateOnCPU(d,vs,fi,cpu,sd,res,na,nr)
28         if sprime ≠ NonSchedulable then sres := sres ∪ {sprime} end
29       done
30       if sres = ∅ then
31         /* allocation not possible at date sd, advance date */
32         sd := advance_time(sd,d,fi,vs)
33       else
34         s := choose_optimal(sres,fi,cost_function)
35         found := true
36       end
37     done
38     if not found then return NonSchedulable end /* scheduling of f was not possible */
39   done
40   return s /* application scheduling was successful */
41 end procedure
```

Figure 7: List scheduling driver pseudo-code

```
1  procedure ScheduleBlockAtDateOnCPU
2  inputs: d:dataflow; vs:valid_scheduling_state; fi:function_instance;
3          cpu:int; start:int; time_len:int;
4          mem_align:int; mem_res:memory_reservation;
5  outputs: sprime:scheduling_state
6  begin
7    /* Check if CPU is free on the desired interval. If yes, reserve it.*/
8    f_res := { starti = start; endi = start+time_len; }
9    if not cpu_free_on_interval(vs,cpu,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   if not ram_free(vs,cpu,mem_align,mem_res) then return NonSchedulable end
13   vs := reserve_mem(vs,cpu,fi,mem_align,mem_res)
14   /* Worst-case accesses of snippet fi to various banks under given allocation
15      (upper bounds over r_fi(b) and w_fi(b), for all b, cf. Section ??) */
16   vs.inst_wcat[fi] := wcat(d,vs,mem_res,fi)
17   f := get_function(fi) /* 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       S'.current_use(blk') := tmp_use
27       vs.inst_current_wcet[fi] :=
28         vs.inst_current_wcet[fi] + interf(vs.inst_wcat[fi],vs.inst_wcat[gi])
29       /* if provisions on fi are not sufficient due to gi */
30       if vs.inst_current_wcet[fi] > length(vs.inst_time[fi]) then return NonSchedulable end
31     end
32   done
33   /* allocation and scheduling are possible */
34   return (Schedulable vs)
35 end procedure
```

Figure 8: Function that makes the actual reservations and checks schedulability

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)* (120 ms in A1, 240 ms in A2) is divided into a sequence of *minor frames (MIFs)* of equal length (5 ms in A1, 15 ms 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 5124 dataflow nodes and more than 36000 variables before hyperperiod expansion, and 18672 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. Its has a very hierarchical structure so that parallelism must be exposed through inlining. It has 4792 function instances after normalization. The WCET of dataflow functions in A1 ranges from 37.5 ns to 60.66 μs and from 1 to 994 μs in A2. By comparison, the total synchronization and coherency overhead (cf. Section **??**) is less than 265 ns 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 (1.65 MB available with the system configuration available for our experiments, cf. Section **??**). For this purpose we configured the parallel back-end to assume larger memory

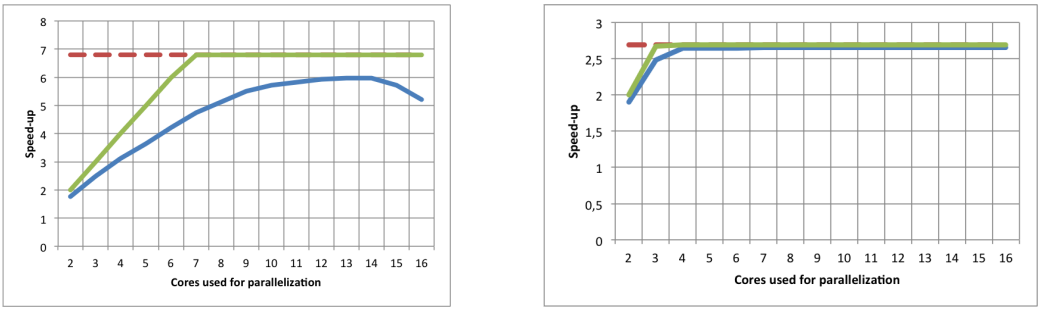


Figure 9: Speed-up figures for A1 (left) and A2 (right)

banks. We validated the third objective on A1 only. 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 on average; these MIFs, when considered separately as applications, are denoted $\text{A1}_j = \{0, \ldots, 23\}$. Note that we could also have considered an overlay mechanism to manage code/text memory at run time. 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 of Section **??**. On a 4-core Intel Core i7 architecture with 16 Gbytes of RAM, the scheduling and allocation of the largest example (A1) on 8 (resp. 16) cores cores took 29.22 s (resp. 42.97 s), the compilation of the $\text{A1}_j = \{0, \ldots, 23\}$ took 0.23 s (resp. 0.51 s) on average, and that of A2 3.72 s (resp. 3.51 s).

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 $\text{min\_mif}(a, c, p)$. We denote with $\text{op}(a, c)$ the optimal interference provisions, i.e., the value of p that maximizes $\text{min\_mif}(a, c, p)$ for given a and c, and with $\text{min\_mif\_opt}(a, c) = \text{min\_mif}(a, c, \text{op}(a, c))$. For the sequential case ($c = 1$), $\text{min\_mif}(a, 1)$ is computed as the maximum of the per-MIF sum of block WCETs.[12] *Guaranteed speed-up* of the parallel code with respect to the sequential reference is then defined as $\text{gso}(a, c) = \text{min\_mif}(a, 1)/\text{min\_mif\_opt}(a, c)$. An upper limit on guaranteed speed-up for a given application is obtained using the critical path method [**?**]. For each MIF m we determine its critical path $\text{cp}(a, m)$ and then the parallelization limit of the application is computed as $\text{pl}(a) = \text{min\_mif}(a, 1)/\max_m \text{cp}(a, m)$.

The speed-up figures $\text{gso}(A1, c)$ and $\text{gso}(A2, c)$ for $c = \{2, \ldots, 16\}$ are provided (in blue) in Fig. **??**, 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 (in green) the guaranteed speed-up 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 speed-up is very good,

[12] Including function call overhead, but no synchronization, cache coherency, on interference.

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 $A1_j = \{0, \ldots, 23\}$, in Fig. **??**(left). Notice that most $A1_j$ parallelize better than the whole A1 does. This is normal, because the results in Fig. **??** 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 Fig. **??**(right) the guaranteed speed-up for $A1_0$ (in blue), as well as the guaranteed speed-up obtained if we assume that all overheads are zero (in green) or that interference overheads alone are zero (in red). Like in Fig. **??**, the green line shows almost perfect parallelization. Considering the synchronization and coherency overheads reduces the performance, but it still increases quasi-linearly with the number of cores.

The performance loss is therefore clearly due to memory access interference, also known in the literature as saturation of the memory bandwidth [**?**]. 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 $A1_j = \{0, \ldots, 23\}$ on the test platform. Execution duration measures were compared with the worst-case bounds provided by our tool, and measured execution time was always smaller than the worst-case bound. 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 duration and compared it to measured duration of a sequential implementation. We have done his for each of the 24 tasks, and then taken the average and variance. Results are provided in Fig. **??** (in red), along with the same averages for the static guarantees. The statistical significance of these results is low, because measure was performed for a single test vector for each $A1_j$.[13] 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. In particular, our flow does *not* require adding

[13] Comprehensive test vector sets were only available for the whole application.

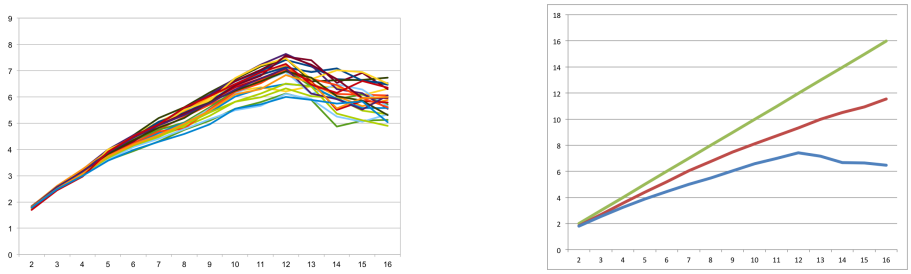


Figure 10: Guaranteed parallelization for $A1_j = \{0, \ldots, 23\}$ (left). Contribution of overheads for $A1_0$ (right).

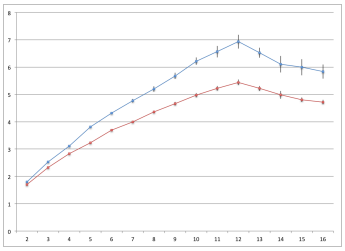
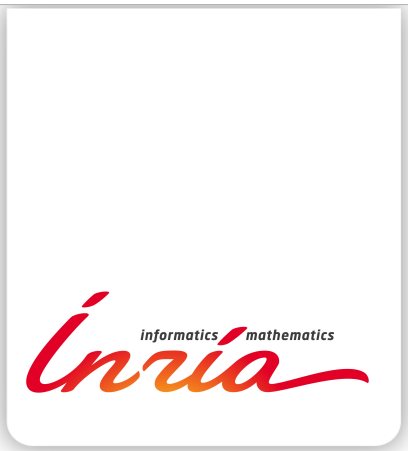


Figure 11: Predicted (in blue) vs. measured (in red) performance for $A1_j = \{0, \ldots, 23\}$

experience-based margins to computed worst-case execution time (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, through 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 execution 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 will involve modifications to the industrial Scade compiler and the design and implementation of qualified versions of the tools composing the flow.

From a scientific perspective, many challenges remain. In the near future, we will consider the problem of mapping on the full Kalray MPPA many-core, and that of mapping onto other multi- or many-core platforms. The experimental evaluation also emphasized the importance of evenly distributing the application load among its minor frames. We will consider this optimization problem, which must be performed under I/O latency requirements specific to the application and to the industrial process.



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399