Efficient fine-grain parallelism in shared memory for real-time avionics

Philippe Baufreton(1), Vincent Bregeon(2), Keryan Didier(3), Guillaume Iooss(4), Dumitru Potop(3), Jean Souyris(2)

(1) Safran Group (2) Airbus Operation S.A.S. (3) Inria (4) ENS

Abstract: Multi-task/parallel software design methods for critical embedded systems often enforce space and/or time isolation properties, which constrain resource sharing to facilitate the design process. For instance, ensuring that computing cores only interfere with each other during dedicated communication phases largely simplifies the timing analysis of parallel code. The downside of isolation is efficiency loss – longer latencies, smaller throughput, increased memory use.

We focus on hard real-time applications (or parts thereof) inside which isolation is not a requirement. We show that, in this case, fine-grain parallelism can be more efficiently exploited without isolation, while still providing the levels of safety and hard real-time guarantees required in critical industrial applications.

Resulting implementations allow multiple computations and communications to take place at the same time, provided that interferences can be controlled (which is possible on timing compositional platforms). We demonstrated this approach on two large avionics applications in a dedicated shared memory context, using an automatic parallelization method. Our method provides good parallelization results and, in one context, has reached TRL4.

Keywords: Automatic parallelization, Hard real-time, Avionics, multi-core, many-core, Lustre, synchronous

1. Introduction

Resource sharing is not new in critical systems engineering [A653,A664], but with the current drive towards increased functionality on increasingly parallel hardware its use cannot be avoided, even in the most critical systems. Resource sharing leads to interferences. When not properly controlled, interferences can reduce both the efficiency and the predictability of a system. In turn, reduced predictability complicates safety demonstration, and is unacceptable in critical systems.

The main engineering instrument for controlling resource sharing is isolation. It allows the simplification of system analysis by reducing the interferences between various components, or by confining these interferences to precise space/time envelopes. The best-known use of isolation is the full time-space isolation between partitions enforced by the ARINC 653 avionics standard [A653] (in this context, time-space isolation is called time-space partitioning, or TSP). However, less thorough isolation hypotheses are used in most implementation methods for critical parallel software proposed in recent years [CPP+18,RMM+16, PFF+18]. For instance:

- The spatial isolation between memory regions accessed by individual tasks or cores, with explicit copy operations whenever one task/core needs information produced by another [CPP+18, RMM+16, PFF+18]. This property is used to render code generation more portable [CPP+18] or, in conjunction with memory allocation, to facilitate timing analysis [RMM+16, PFF+18].

- The temporal isolation between computation and communication phases. This can be enforced system-wide, as in the bulk synchronous parallel (BSP) model [Val90], where execution is globally divided in computation phases, where cores do not exchange information, and communication phases, where no computation is performed. It can also be enforced on separate tasks [PFF+18,CPP+18, partially in RMM+16], ensuring that during the computation phase the task accesses only its stack and state (as well as code memory and constant memory), but not shared variables. This form of isolation facilitates WCET analysis of computation code. This form of isolation is echoed in the read-compute-write (RCW) execution model of control loops encoded in Simulink or SCADE.

- Time-triggered execution [GMR+18, PFF+18] is often used to enforce strict timing conformity between a schedule (timetable) computed off-line and actual execution. This form of timing isolation helps reduce interferences and facilitates timing analysis, by confining tasks to specific time intervals. Many variants exist, from fully time-triggered, where each task start, deadline, and possibly even preemption dates are pre-computed, to mixes with event-driven scheduling, where only certain start dates and deadlines are enforced using timers.

- The total absence of resource access interferences between cores [CDP+14, PFF+18] or applications [A653] is a comprehensive isolation property. It means that the shared resource accesses of one core/partition need not be taken into account in the analysis of another. This property is obtained through the use of the mechanisms defined above, in conjunction with dedicated scheduling and code generation methods.

Such isolation properties are the result of an implementation process, but sometimes they are required at the specification level on partitioning units such as
applications [A653] or tasks [CDA+05]. Such specification-level isolation requirements constrain the implementation process.

In many cases, isolation properties are explicitly mandated by regulation. In Integrated Modular Avionics (IMA), robust partitioning by means of time-space isolation between applications is required even in single-core environments [DO297].

In multi-core processing (MCP) contexts, where hardware resources are shared between applications, these isolation principles are maintained and required by regulation [CAS16]. Regulation requires exposing all hardware configuration settings. It also requires identifying all possible communication resources/channels (including side channels), to allow their characterization and the mitigation of undesired interferences. When applications share resources, regulation requires that interferences through all resources/channels are mastered (kept at “acceptable” levels) in order to provide both functional determinism and execution time (WCET) guarantees. This often requires enforcing isolation properties. This explains why various platform standards (e.g. [A653,A664]) and programming languages (e.g.[CDA+05]), as well as the parallel implementation methods cited above provide dedicated constructs to enforce isolation properties.

But when hardware resources (cores, memory) are shared inside a single application domain, there are cases where strong isolation is not required:

- When a large, single-criticality flight control or engine control application is parallelized, being the sole application to execute on the multi-core processor, isolation between its components is not required by regulation. It is only required to ensure its functional and timing predictability, allowing its integration in the avionics systems.
- When the application shares the multi-core hardware with other applications, then the application developer must prove the respect of the resource envelope assigned to it by the system integrator. However, inside this envelope, resource sharing is possible as in the previous case.
- The same is true when only one component (e.g. task) of a larger application subject to TSP is parallelized. In this case, isolation between components/tasks is needed, but not between the parallel threads implementing the component/task that is parallelized (inside the resource envelope assigned to the component/task).

Some implementation methods may enforce isolation even in such cases – as a form of over-engineering – to facilitate timing analysis. However, this is not needed on platforms whose HW and SW architecture allow keeping interferences at an “acceptable” level by other means. Such is the case for statically scheduled bare metal code on timing predictable HW.

In such cases, isolation needlessly limits the implementation space, and impacts efficiency.

We expect such cases to become more common, for two fundamental reasons:

- As more computational power is available, computation-intensive code such as Kalman filters, vibration analysis, or other “digital twin” predictive simulations will increasingly find their way into the control loop, requiring parallelization.
- Common COTS multi-cores (e.g. ARM, POWER) have little support for time isolation between cores running in parallel. Thus, ensuring time isolation properties such as those required by IMA may require the use of parallelized partitions spanning all available cores, as opposed to allowing different partitions to execute on different cores at the same time.

It is therefore important to quantify the cost of isolation, and to propose implementation methods for such applications, components, or partitions that require efficiency, safety, and real-time guarantees, but not strong internal isolation.

1.a. Contribution. This paper provides first elements allowing the quantification of the cost of isolation on multi-cores under a non-preemptive symmetric multiprocessing (SMP) approach. To do this, we use an existing parallelization method for critical real-time applications [DPI+19]. Previous work on this method defined its tool-flow and mapping algorithms, and evaluated its effectiveness (efficiency, safety, and ability to integrate in the existing industrial process) on industrial use cases.

In this paper, we determine that the parallelization method we consider enforces significantly less isolation properties than previous methods. Using two large industrial use cases, we also show that enforcing stronger isolation properties – as other methods do – would significantly decrease the efficiency of the generated code (in terms of speed or memory use), or make the integration in the industrial process more complicated.

1.b. Related work. Closest to our work is [RMM+16], which evaluates the effect of isolating computation from communication at task level on the precision of timing analysis for shared memory parallel code. Similarly, [HJB+18] shows that relaxing the time isolation of static time division multiplexing results in considerable efficiency gains, while preserving real-time guarantees. By comparison, the parallelization method of [DPI+19], which we study here, enforces even less isolation properties.

More generally, a significant amount of work in the field of real-time scheduling evaluated the differences between scheduling algorithms enforcing different partitioning/isolation properties (e.g. partitioned vs. global scheduling, off-line vs. on-line scheduling).
1.c. Outline. The remainder of the paper is organized as follows: Section 2 uses a simple example to intuitively explain how enforcing the various isolation properties can result in more or less severe performance penalties. We also present here a mapping and code generation method (the one of [DPI+19]) that does not enforce these isolation properties. Section 3 takes a higher-level view of a common industrial process, by considering not only mapping and code generation phases of the implementation process, but also its front-end, which builds a deterministic specification, ready for mapping, from an initially non-deterministic specification. Design choices made in the front-end (and often represented with isolation decisions) may negatively impact the performance of the implementation, unless these choices take into account target platform. Section 4 provides experimental data based on two industrial use cases supporting our claims. Section 5 concludes.

2. Motivating example

To intuitively illustrate the consequences of isolation we consider the simple, synthetic application of Fig. 1, and the problem of mapping it on a dual-core.

Like often in critical avionics, the application is a dependent task set specified in a dialect of the Lustre language.1 Lustre [HCR+91] allows the specification of dataflow applications in textual form. Our small specification is formed of 4 tasks (f,g,h,n), also called dataflow nodes. Lustre programs have a cyclic execution model. Inside each cycle, the 4 tasks are executed in an order compatible with the intra-cycle data dependencies determined by variables x and t. Data can be transmitted from one cycle to the next using the fby constructs, which also provide a value to use in the first cycle (7 and 9, respectively, in our case). In our application, inside each cycle, task f, which produces variable x, must be executed before both g and n, which use x. Task h must be executed before n.

2.1. Multi-core real-time implementation

This application is mapped onto a dual-core processor using the method of [DPI+19]. By mapping we understand here the allocation and scheduling of nodes on cores, the allocation of variables and code into memory, and code generation enforcing these allocations and scheduling decisions. To achieve high functional and temporal predictability, allocation and scheduling are static. The C implementation code and part of the linker script synthesized by the chosen parallelization method (simplified to facilitate reading) are provided in Fig. 2.

The linker script fragment we provide here shows how variable x and how the code and local data of task f are allocated. The full linker script allocates all remaining variables, tasks, and also the code and local data of thread functions. In conjunction with the use of mutex-based synchronization (described below), static memory allocation allows us to ensure functional determinism and correction, and to provide (tight) upper bounds on the execution time penalty a piece of code (e.g. dataflow function) incurs due to interferences from other cores.

![Figure 1. Simple Lustre program (left) and graphical representation (right)](image)

The cyclic execution model of the Lustre program is enforced at C level using two infinite loops running in lockstep. Each cycle of the loops implements one cycle of the Lustre model. Lockstep execution is enforced using a global barrier, which is traversed synchronously on both cores. This barrier also enforces the periodic triggering of cycles (the sole real-time requirement in this example). The efficiency metric optimized (minimized) by the parallelization method we used [DPI+19] is the triggering period T of the cycles. To ensure that one cycle finishes before the previous one is triggered, T must be a safe upper bound of the worst-case execution time (WCET) of the parallel code of one cycle.

```c
int x,y,z,t;
void init(){ y=7; z=9; }

void thread_cpu0(){
  init();
  for();
}

void thread_cpu1(){
  for();
}

/* FUNCTION f */
= 0x60400 : f_text ALIGN(ICACHE_LINE_SIZE) : {
  f_step.o(.text) f_step.o(.scade_code)
}

/* DATA f */
= 0x60400 : f_data ALIGN(DCACHE_LINE_SIZE) : {
  f_step.c.o(.data) f_step.c.o(.bss) f_step.c.o(.rodata)
}

/* VARIABLE x */
= 0x604e8 : x ALIGN(DCACHE_LINE_SIZE) : {
  *(.x) }
```

---

1 The industrial dialect of Lustre is SCADE [CPP+18]. The dialect directly used by our tools is Heptagon [DMP+].
Figure 2. Dual-core implementation of our example (C code and part of the linker script). In the C code, the barrier performs periodic triggering of computations.

Synchronization between cores inside one cycle (e.g. to enforce the data dependency between f and n) is event-driven, based on the use of lock/unlock operations on mutexes. Furthermore, as our test platform (Kalray MPPA256 Bostan) lacks hardware cache coherency, explicit cache operations are needed to ensure the transfer of data between cores.

To synthesize the code of Fig. 2, our parallelization method relies (like other similar methods, e.g. [PFF+18]) on off-line real-time scheduling. We provide in Fig. 3 (left) the scheduling table produced by our method, assuming that the WCET of the tasks f, g, h, and n, in isolation, are respectively 200, 400, 300, and 300 time units. In addition to these values, each task is allocated (the yellow bars) a budget covering cache coherency, synchronization, and interferences from other cores (e.g. interferences due to the concurrent access of n and g to variable x).

2.2. Isolation properties and their impact on efficiency

The implementation of Fig. 2 lacks most of the isolation properties listed in the introduction.

2.2.1. Memory isolation.

While the parallelization method allocates memory to reduce interferences, it does not attempt to privatize memory banks or isolate individual tasks/nodes, applications, or cores. A first, very practical consequence is that, unlike in [RMM+16], two cores can share a memory bank for code, stack, and local data, allowing the use of all 16 cores of a cluster on the test platform, even though less than 16 banks are available for the application.

Communication variables are not replicated on a per-node basis (as in [CPP+18], which would result in 9 C-level variables) or a per-core basis (as in [RMM+16,PFF+18], which would result in 6 C-level variables). In our case, there are 4 variables, directly corresponding to the 4 data-flow variables. Note that no supplementary variable is needed to encode the fby constructs.

2.2.2. Isolation between computation and communication phases.

In the code we synthesize, there is no isolation between computation and communication phases at the level of the system (as in BSP) or at the level of nodes, as in [PFF+18,CPP+18,RMM+16]. At any moment of its execution, a node can access any input, output, or local data. Nodes are also allowed to interfere during access to shared variables (e.g. g and n can interfere due to accesses to x).

Figure 3. Three dual-core schedules for our example. In yellow, synchronization, coherency, and interference costs. In brown, global barriers. Under BSP, barriers perform communications and enforce coherency, thus incurring all interference, synchronization, and coherency costs.

To illustrate the effect of isolation, we have pictured in Fig.3 (middle and right) the optimal schedules under respectively:

i. BSP isolation between computation and communication phases

ii. Scheduling with our method, which does not replicate variables, but requiring the absence of interferences due to access to shared variables.

We can see that BSP scheduling results in some idle time, whereas enforcing the absence of interferences in the absence of variable replication seriously affects efficiency. For instance, the execution of task n in Fig. 3(right) cannot be performed on Core 1 in parallel with g, because n and g both access variable x, and so executing them in parallel can result in interferences.

2.2.3. Time isolation

While the scheduling table of Fig.3 (left) is a time-triggered activation pattern like that of [GMR+18,PFF+18], we chose to implement it (in Fig.2) using mostly event-driven mutex synchronizations, thereby losing the strong timing determinism. In doing so, we lose neither functional correctness nor the respect of real-time requirements [DPI+19]. Instead, we gain some degree of robustness to timing errors. For instance, if f takes some more time to execute than predicted\(^2\), then there is still a chance that this is absorbed by g and n executing faster.

The only isolation properties we need to enforce are:

- The respect of periods and deadlines. These are necessarily enforced with timers, which can

\(^2\) This can happen if a static WCET was incorrectly computed, or if the WCET is obtained using a probabilistic or stochastic method, and a rare event occurs.
become partition switch barriers if multiple applications are executed on the same multi-core.

- That no interferences can happen in the C implementation that were not covered by the interference analysis applied on the scheduling table of Fig. 3. These are enforced using mutex synchronization. For instance, in Fig. 3(left), the time reservations for tasks f and n do not overlap, so the interference analysis assumes they cannot interfere. To ensure that f and n cannot interfere, the mutex synchronization in Fig. 2 ensures that inside each cycle task f completes before task n starts.

3. Multi-periodic systems specification and implementation

The motivating example of the previous section is very simple in order to keep the focus on classical time-space isolation properties and their effects. In particular, the application is mono-periodic, and deterministic.

However, in many industrial settings, the system specification is multi-periodic, and it is initially non-deterministic. The exact dependencies between task instances are not fully determined. Instead, we know which tasks exchange information, and we have real-time requirements on so-called functional flows. For instance, the original specification of the example in Fig. 1 may specify that data produced by task f is used by g and n, that output produced by h is used by n, that output of g is used by f and h, and that the functional flow formed of f and g must take less than 10ms. This specification does not determine, for instance, which instances of f and g exchange data. If timing properties allow it, we could set up fby operators on each of the dataflow arcs.

As systems implementations must usually be deterministic, the design process will progressively transform the specification into a deterministic one, which is an artifact of the design process. Other intermediate artifacts may also be built in the process, where part of the non-determinism has been removed, e.g. one where the dependencies between task instances have been fully determined, but the exact triggering dates of tasks not yet set.

This process of building a deterministic specification is reducing the implementation space, and may impact efficiency if it is not conducted as part of the global optimized implementation process. One key method used in determinizing a specification is enforcing time isolation properties.

In this section we explain how we perform this determinization process. In our case, it is accompanied by a normalization of the specification, meant to facilitate the work of mapping algorithms. Normalization includes, among others, ensuring that all data memory (including all task states) is exposed to the parallelization algorithms under the form of dataflow variables.

We consider only harmonic multi-periodic applications, where a larger period is always a multiple of a smaller period. For instance, an application featuring task periods of 5ms, 10ms, 20ms, 40ms and 120ms is of harmonic multi-period type, whereas an application featuring periods 15ms and 20ms is not. In such systems, we shall call major frame (MAF) the largest period, and we call phase or minor frame (MIF) the smallest period of the application.

3.1. Multi-periodic specification methods

We consider two methods of specifying a multi-period application, which correspond to two different industrial processes.

The first one uses a dependence graph, like the one in Fig. 4, to provide the initial, non-deterministic description of the system. In the dependence graph, each node corresponds to a computation/task and has a period. The edges of the graph represent variables, stored in global memory, which are produced by a (single) source node and consumed by the destination nodes. Source and destination nodes of an edge may have different periods.

Non-determinism comes from several sources:

i. For an edge connecting two nodes of the same period, we do not know if we use the data produced during the current period or during the previous one (absence or presence of a fby construct).

ii. For an edge where the source has a smaller period than the destination, we do not know which instance of the source to sample.

iii. For an edge where the source has a larger period than the destination, we do not know at which instance of the consumer we start to use the data produced during one period.

We allow specifying latencies on functional flows (chains of dependences between nodes).

![Figure 4. Representation of a multi-periodic integration program based on a dependence]

3 Like the one in Fig. 1.
The nodes are annotated with their period ("Per n" = "period n").

Transforming such a specification into a deterministic one involves both determining the exact dependencies between task instances, and determining the moment in time (e.g. the MIF) where each task instance is released.

The second specification method we consider amounts to directly providing the result of the determinization process. This can be done in a synchronous language such as SCADE. One popular method of doing this is to assume that one cycle of the synchronous SCADE specification implements one generic MIF. Then, conditional execution (e.g. using the «condact» construct of SCADE) is used to represent the periodic triggering of tasks. A specification of this type is provided in Fig. 5. Here, the boolean variables controlling the triggering of tasks f and g by the condact constructs are clkf and clkg. In our program, clkf is always true, meaning that an instance of f is triggered at every cycle/MIF, whereas g is only triggered at cycles 0,2,4… The first condact construct will compute a value for variable y at each execution cycle. When clkg is true, y is computed as the output of g(x). When clkg is false, y takes its value from the previous cycle (or 0, if it’s the first cycle). In the following line, clkf is always true, so the condact is not needed (we could simply write “x = f(0 fby y)”). However, we preserve it for uniformity (all task activations have the same form).

```plaintext
period(10000)
node main() returns ()
var clkf, clkg : bool; x, y : int;
let
  (* Computation of activation conditions *)
  clkf = true;
  clkg = true fby false fby clkg;
  (* Conditional activation of the tasks *)
  y = condact(clkg, g(x), 0 fby y);
  x = condact(clkf, f(0 fby y), 0);
```

**Figure 5. Representation of a multiperiodic integration program, based on condact.**

While the «condact» construct of SCADE allows the specification of task activations with respect to the cyclic activation of the main node, the SCADE/Lustre language does not allow the specification of the real-time period of activation of the main node itself. This period – the MIF, equal in Fig. 5 to 10000 time units – is specified in our approach using the «period» annotation of the main node. This annotation completes the platform-independent specification of our application.

3.2. Determining the minor frames

Moving from a specification like the one in Fig. 4 to one like in Fig. 5 requires choosing for each task the MIFs where the task is triggered, and requiring that each instance of the task instance is performed inside the chosen MIF period. This amounts to choosing activation conditions compatible with the chosen periods (without changing the conditional activation code). In Fig. 5, the only other choice of activation conditions is the one where clkf is unchanged and “clkg = false fby true fby clkg”. The choice of activation conditions is made to ensure the respect of real-time properties such as end-to-end latencies of flows, or to ensure that the workload is well balanced between MIFs.

This scheduling process can be performed automatically. We do it using a cost function that chooses among all schedules that respect the periodicity and flow latency constraints one that minimizes a cost function (e.g. one that balances the computational load among the MIFs). Automating this process is particularly interesting when the specification has a large number of tasks: choosing a schedule by hand is tedious and finding an optimal one might be difficult if the architecture is complex.

To find a valid schedule, we extract linear constraints on the phases from the dependences between nodes. We can add additional constraints such as latency requirement or resource occupancy. Then, given a cost function, this problem can be expressed as an Integer Linear Program. Once a solution is found, we can use it to have a fully specified Scade program.

| MIF 0: f, m, n, p |
| MIF 1: f, m, n, p, g |
| MIF 2: f, m, n, p, h |
| MIF 3: f, m, n, p, g |

**Figure 6. Scheduling the tasks/nodes of Fig. 4 into the four MIFs of the application MAF**

Note that, like all form of static scheduling, assigning MIFs to nodes/tasks is a form of timing partitioning. However, in the method we propose this partitioning is an output of the method, chosen to optimize the performance of the system.

This is very different from approaches where a a MIF assignment from a previous implementation of a specification (e.g. on a different platform, possibly sequential) is used as a requirement for the implementation on a new platform. Such an approach has the advantage of preserving unchanged the deterministic specification artefact of the first implementation, which facilitates functional testing and integration, but can significantly reduce the efficiency, as we will see in Section 4.

3.3. Normalization

Once a deterministic multi-periodic specification is built (encoded as a Lustre program), we process it to facilitate the work of the mapping algorithms. We apply a program transformation called hyper-period expansion [DPI+19]. The purpose of this
transformation is to remove all the complex conduct logic that conditionally triggers the tasks, and especially make explicit the dependencies between instances of the tasks along the MAF. This is achieved by unrolling MAF/MIF times the initial Lustre deterministic specification, in which one cycle represents the execution of one MIF. One cycle of the resulting Lustre specification represents the execution of one MAF. Unrolling is accompanied by a propagation of constants which removes unneeded code. The result of hyper-period expansion for the simple Lustre program in Fig. 5 is provided in Fig.7. For each node/task t of the original specification, hyper-period expansion will create MAF/period(t) instance nodes. For this reason, in Fig.7, f is instantiated 2 times.

Tasks can have an internal state, and the instantiation process must ensure that this state is passed from one instance to the next in order to preserve functional correctness. To this end, hyper-period expansion exposes this internal state as an additional input and output of each node/task.

The outputs of a node/task are also instantiated multiple times, as under the synchronous model each variable can be assigned only once during an execution cycle. For example, variable x in Fig. 5 is replaced in Fig. 7 with variables x1 and x2. Once instantiation of variables is performed, the inputs of nodes can be set to express the correct dependencies. For instance, in Fig. 7, the second instance of f uses the value output by g in the current MAF, whereas the first instance of f uses the value output by g in the previous MAF.

More complex buffering schemes can be represented in this way.

```plaintext
period(20000)
node main() returns ()
var x1, x2, y1 : int;
let
  sf1, sf2 : state_f ; sg1 : state_g ;
  deadline(10000)
  (sg1,y1) = g(sg_init fby sg1,x1) ;
  deadline(10000)
  (sf1,x1) = f(sf_init fby sf2,0 fby y1) ;
  release(10000)
  (sf2,x2) = f(sf1,y1);
end
```

Figure 7. Program from Fig. 5, after hyperperiod expansion (MIF=1, MAF=2).

The final program is mono-periodic and does not have any temporal operator other than the fby (Fig. 7). This program can also be larger, as each function call may be instantiated multiple times (MAF/MIF at most). Inputs and outputs of the program are also duplicated, which causes an issue with a real-time execution of the program. Indeed, the inputs would be progressively available (instead of directly from the start), and, symmetrically, the output must be released progressively. Thus, we introduce annotations on the equations of the program to inform the scheduler that a given node must be executed after a certain logical time ("release") and must be executed before a certain logical time ("deadline"). For instance, the second instance of function f must start in the second MIF, so it starts after date 10000, as specified by the "release" annotation, and before date 20000 (the end of the period). Note that the period of the program changed from the MIF (10000 in our case) to the MAF (20000=2*MIF). Once hyper-period expansion performed, these annotations can be relaxed (manually) to allow more flexibility during scheduling, allowing a function to be scheduled outside its MIF (subject to data dependencies).

3.4. Resource allocation, real-time scheduling, and code generation

Once a deterministic, normalized Lustre specification is produced, it is given to the back-end of our implementation flow which performs:

- Static real-time scheduling, i.e. allocation of CPU time to the tasks/nodes of the Lustre program.
- Memory allocation of all code and data.
- Code generation.

The real-time mapping is not an original contribution of this paper, but of [DPI+19]. We only present here some details facilitating our presentation. All three back-end operations are performed jointly, in a way that ensures both the functional correctness of the generated implementations, and the respect of the real-time requirements.

The back-end produces, as an intermediate artefact, a scheduling table, as introduced in Sec. 2. The scheduling table of the example in Figs. 4 and 6 is provided in Fig. 8.

The originality of the mapping approach, and the cause of its efficiency is that it follows the principles laid out in Sec.1. It does not enforce strong timing isolation properties between tasks of an application that does not require them. Thus, it can allow, and control, interferences, allowing efficient resource sharing between the computing cores. Interference analysis is performed during RT scheduling, based on the choices of the scheduling table, which the implementation must preserve, as explained in Sec. 2.2.3 (a form of timing isolation).

<table>
<thead>
<tr>
<th></th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>g/h</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8. Table of scheduling for the program of Fig. 4 on 2 processors
4. Experimental results

4.1. Use case 1

The first use case we present in this abstract is a large application formed of >5000 unique dataflow nodes connected with >30000 dataflow variables. It is a classical multi-period major frame/minor frame (MAF/MIF) avionics application. Execution involves 5 harmonic periods of ratios 5:10:20:40:120, the smallest being the MIF, the greatest the MAF. Each dataflow node is assigned a period among the five and a phase (the first MIF where it is activated). Each node execution must be contained inside a MIF.

The objective is to parallelize the application and synthesize the code. The industrial requirements were the following: (1) parallelization (allocation, scheduling, and code generation) must be fully automatic and performed in reasonable time; (2) the parallelization method must not rely on the user providing WCET values taking into account interferences; (3) generated code must be free of deadlocks and data races, and functionally correct with respect to the specification; (4) it must respect real-time requirements (each node must be executed in the prescribed MIFs); (5) it must achieve a 5x speed-up on 8 cores; (6) code generation must be compatible with industrial code generation and testing infrastructure. The requirements were reached, and the parallelization method attained TRL4 on the chosen test platform (Kalray MPPA256 Bostan).

4.1.1. The cost of isolation

Given existing work, the key difficulty concerned the performance requirement (5). Early in the design process we realized that enforcing the isolation properties employed in other methods was incompatible with the fine-grain parallelization problem we considered. To give (partial) insight into why this happens, we provide Figs. 9 and 11. The first shows the number of shared C variables in the code we generate (bottom line), compared with the number of variables when using per-node replication [CPP+18] (top), or per-core replication [RMM+16,PFF+18], assuming the same allocation of nodes to cores as for our code. The significantly larger number of variables does not only count as extra memory allocation, but also as copy operations, which take significant time and space.

The second table shows the effect of requiring the absence of interferences in our method that does not replicate variables among cores/nodes. We can see that parallelization is significantly lower.

4.2. Use case 2

The second use case is also a MIF/MAF application, with MIF=15ms and MAF=240ms. It is smaller than the first use case, yet still large, with 4792 function instances after normalization.

The specification of this use case was done using the second specification method described in Sec. 3.1. It consists in Lustre/SCADE code describing the execution of one generic MIF.
The main difficulty in producing an efficient implementation for this example is that the specification itself was heavily tailored for efficient sequential implementation. Most notably, the critical path [DPI+19] of the specification (the longest chain of computations that must be performed in sequence inside a cycle) limits parallelization to 2.69x. This theoretical, hard limit is represented with the red, dashed line. The parallelization attained by our method (which takes into account inter-core interferences due to concurrent execution on the multi-core) is represented with the blue curve – we virtually reach the theoretical limit with 4 cores. The green curve provides the result of our mapping heuristic if we assume interferences cost nothing.

While more research is needed on the topic, it is likely that relying on the deterministic specification level built for another, sequential target architecture results in a severe penalty, as noted in Sec. 3.2, and that the only solution allowing further parallelization gains is to move (at least partially) towards a specification of the first type. This amounts to partitioning nodes into MIFs in a way that is takes into account the target architecture.

6. References


[CDP+05] D. Chabrol, V. David, C. Aussagrèes, S. Louise, F. Daumas. Deterministic distributed safety-critical real-time systems within the OASIS approach. Proc. PDCS 2005

