Contributions to the safe and efficient parallelisation of hard real-time systems

Keryan Didier

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité Informatique

École doctorale Informatique, Télécommunications et Électronique (EDITE)

Présentée par

Keryan Didier

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Contributions to the safe and efficient parallelisation of hard real-time systems

soutenue le 19 septembre 2019

devant le jury composé de :

Prof. Christine Rochange	Rapporteur
Prof. Reinhard Von Haxleden	Rapporteur
M. Philipe Baufreton	Examinateur
Dr. Albert Cohen	Examinateur
Prof. Alix Munier	Examinateur
Dr. Dumitru Potop-Butucaru	Directeur de thèse

Abstract

We propose an automatic parallelization method for applications featuring fine-grain internal parallelism. Our method adds the needed overheads to the WCET values, ensuring safety by construction without the need for subsequent schedulability analysis. It is aimed at applications with fine-grain parallelism where excessive per-task overheads would result in poor parallelization gains. 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. By covering all aspects of resource allocation and code generation, our work belongs to the compilation realm. What fundamentally differentiates it from previous compilation work is the choice of performing a *safe*, worst-case timing analysis *incrementally during compilation* around which are integrated parallelization, real-time scheduling, and memory allocation. This method is suitable for very large-scale applications, as it uses low-complexity mapping heuristics, which *guarantees* scalability.

We also propose a language, named InteLus, for the description of parallel multi-threaded implementations of dataflow specifications. It is a sub-set of Lustre extended with annotations representing mapping and code generation choices. While such extensions are common in literature, our language and modeling approach go further in one fundamental way: implementation models specified in InteLus are *strictly richer* than the multi-threaded C code we want to generate. InteLus allows the representation of *all* mapping decisions needed for multi-threaded code generation in our context. InteLus's representation of threads and thread synchronization is a sub-case of the C11/pthread concurrency model. Therefore, C code can be obtained by selectively putting elements of the InteLus program into C and linker script syntax without making any further mapping decision. Annotations are covered by the operational semantics of InteLus. This allows us to formally define the correctness of implementation models. To facilitate the definition of the correctness properties, implementation models are endowed with not one, but two semantics: the *synchronous semantics* of Lustre (which simply discards mapping annotations) and the *machine semantics*, which interprets the program and its annotations as a multi-threaded imperative program. This dual semantic nature of our implementation models enables us to envision an original approach to proving implementation correctness.

Contents

Chapter 1

Introduction

1.1 Context

Full automation is possible and needed in real-time scheduling. The implementation of complex embedded software relies on two fundamental and complementary engineering disciplines: real-time scheduling and compilation. Real-time scheduling covers[1] the upper abstraction levels of the implementation process, which determine how a *functional specification* is transformed into a set of *tasks* and how the tasks are mapped and scheduled onto the resources of the *execution platform* in a way that ensures functional correctness while respecting *non-functional requirements*. By comparison, compilation covers the low-level code generation process, where each task (usually a piece of sequential code written in C, Ada, *etc.*) is transformed into machine code, allowing actual execution.

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

At the high level, many activities remained largely manual for a long time. Such is the case for the partitioning of the application into sequential tasks, the production of *glue code* ensuring task orchestration, communication and synchronization, or even timing analysis, where adding experience-based margins to computed worst-case execution time (WCET) estimates is still commonplace. This lack of automation can be attributed to two factors:

- The lack of standardization in execution platforms and in functional and non-functional modeling languages made the construction of (qualified) tools expensive and inefficient.
- Penalties associated with the lack of automation were acceptable on low-complexity systems featuring few processors and tasks, and often featuring tasks that require little synchronization [1].

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

Both factors are now gone:

- Languages for functional modeling of control applications such as Simulink [2], Scade [3], or LabView [4], and languages for non-functional specification such as SysML [5] or UML/MARTE [6] are today standard practice in industry. There are also well-established official standards like those describing execution platforms in fields such as avionics—ARINC 653 [7]—and automotive industries—Autosar [8].

- The rapidly increasing complexity of execution platforms and software leads to prohibitive development costs for manual processes when efficiency is desired.

Note that efficiency is a key property in real-time systems design. While the *mathematical modeling* of schedulability problems is in the form of constraint systems (there is a solution or not), in *practice*, efficiency of the implementation method will determine how much software can be executed, therefore how many functions can be performed and with which precision.

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

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

The most common is to first build the system, and then check the respect of real-time requirements through a global analysis. Building the system uses here unsafe timing information such as measures, WCETs in isolation plus arbitrary margins, *etc.* 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. The second approach is to ensure *by construction* the respect of requirements. In this case, system construction uses task timing characterizations that are safe for all possible resource allocations (worst-case bounds).

The drawback of the first approach is the lack of traceability between resource allocation decisions and timing analysis results. If the system does not respect its real-time requirements, mapping changes are needed, but these changes may also change the timing analysis, and so on without guarantee of convergence to a solution. The second approach is much more appealing from an automation perspective and considering the fine-grained control it offers to platform engineers. Its drawback is pessimism, as all resource allocations are made for the worst case.

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

multiperiodic task graphs of real-world control applications, and the problem of statically managing *all* their computational, memory, synchronization, and communication resources.

Ensuring the safety of the generated code is difficult. Regardless of the chosen approach (using unsafe timing information or using safe timing information), automatic mapping methods and tools follow a "correct-by-construction" paradigm. However, like for C compilers, this is not sufficient to provide the levels of confidence required by the most critical embedded software. Supplementary arguments are needed, based on both engineering experience (*e.g.* extensive test) and formal arguments (*e.g.* proof of correctness of the tool). In both cases, formalizing the correctness of the mapping tools is essential, and one key step in doing this is defining the *formal semantics of parallel implementations*.

As hardware design resolutely moves towards massive parallelism based on the use of chip-multiprocessors, mastering concurrency, and thus exploiting the full potential of such hardware, becomes increasingly difficult. Threads are one of the major programming paradigms for such multi- and many-core systems. They arguably provide the best portability and the finest control over resource allocation, which are both essential in the design of embedded applications that need to get the best guaranteed performance out of resource-constrained hardware.

However, the expressiveness of threads comes at a price. As a model of computation, threads are wildly non-deterministic and non-compositional [9], making programming, formal analysis, and implementation difficult [10, 11]. This explains why multi-threaded software is often bug-ridden even in the context of critical systems [12].

But there are also good news: in many industrial contexts (avionics, automotive, *etc.*) the use of threads is tightly controlled, and the threaded code implements a functionality specified in a high-level concurrent formalism.

We consider in our work the particular case where the functional specification of the system is done in a dataflow synchronous language such as Lustre/Scade [13] or a sub-set of Simulink [2]. In this case, multi-threaded implementations, possibly obtained using automatic mapping tools, have particular structure and properties: The number of threads is fixed, each one implementing a recurrent task obtained by sequencing a fixed set of dataflow blocks (or parts thereof, obtained by parallelization).

When taken individually, such properties already facilitate the formal analysis of multi-threaded systems. But *in many cases, the multi-threaded implementation preserves a fundamentally dataflow structure*, with specific rules on the way platform resources (shared memory, semaphores) are used. Such implementations are not only *data race free (DRF)* in the restricted sense of [11], but also *deterministic*.

1.2 Related work and contribution

In this context, my thesis is concerned with the *safe and efficient parallelization of hard real-time systems*. This string of keywords hints at several topics in computer science and engineering.

Hard real-time systems. My work concerns the implementation of *real-time systems* [1]. These are systems whose execution is subject to real-time *requirements*, such as *worst-case bounds* on the *latency* between arrival of an event and appropriate *reaction* to it. These systems exist in a large spectrum ranging from multimedia applications such as music player to cyber-physical systems found in transportation, medicine, or robotics.

Hard real-time systems are those where requirements must always be respected. This is not the case in a set-top box, where losing a video frame should not happen too often, but is ultimately largely innocuous. However, this is usually the case in *critical cyber-physical systems*. In such systems, the non-respect of the real-time requirements can lead to life- or mission-threatening situations, either because the requirement is part of the very description of the system function (as in an airbag, which must be inflated in at most 80 ms after the crash) or because the real-time requirements are meant to ensure the controllability of the system (as in avionics flight control software).

Much of the classical work on real-time systems implementation (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) [14]. 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.

Automatic parallelization The problem we consider is the automatic parallelization of hard real-time applications. More specifically, we target shared memory multi-core processors, and our objective is the fully automatic construction of the executable implementation code in an approach similar to classical compilation. In our hard real-time context, this amounts to not only performing the allocation and scheduling of computations onto the various cores, but also synthesizing the control and synchronization code, choosing the memory allocation, and finally ensuring that the non-functional requirements are satisfied given these allocation and code generation choices.

Our work is by no means the first to contribute to this bold objective.

Recent advances have largely automated the construction of task code and the generation of real-time implementations on specific sequential or multi-core targets. Industrial solutions include here Simulink Real-Time from MathWorks, and Scade KCG6 Parallel from ANSYS/Esterel Technologies [3]. Academic results in this direction include [15, 16, 17, 18]. However, none of these tools provide strong schedulability guarantees when integrating multiple synthesized tasks: separate timing and schedulability analysis must be performed after synthesis. Several methods have been proposed [19, 20, 21] but they come with the shortcomings of the first approach sketched in the in-

troduction. In particular, in the event of a global non-schedulability diagnosis, it is difficult to pinpoint its source so as to guide subsequent re-engineering efforts.

A few approaches have gone further, by letting timing analysis results guide mapping and code generation under simplifying hypotheses common in real-time scheduling, *e.g.*, assuming that task WCET values include *overheads* related to parallel/concurrent execution. Among these approaches we cite the industrial tool Asterios Developer from KronoSafe, based on the ΨC language [22], as well as the academic tools and toolboxes SynDEx [23], BIP [24], SchedMCore [25], Prelude [26], Lopht [27, 28], SigmaC [29, 30], the work on the time-triggered mapping of Lustre [31], Xoncrete [32], or the work of Baruah *et al.* on the synthesis of multi-core cyclic executives [33]. We defer the reader to [27] for a longer description.

While these methods guarantee correctness and have the potential of providing more feedback in case of non-schedulability diagnostics, the simplifying hypotheses are seldom (if ever) satisfied in practice. To our knowledge, they are satisfied only when using prototype hardware designed for predictability, *e.g.* by ensuring the absence of memory access interferences [34, 35]. But the hypotheses are never satisfied on modern off-the-shelf multi- and many-cores, where the overheads due to concurrent execution include contributions that may be difficult to estimate, depending on the hardware or software architecture of the system: memory access and bus access interferences, cache-related delays, synchronization costs, scheduler execution time, *etc.*

A step further is taken in [36, 37], where the tool itself adds the needed overheads to the WCET values. In [36], overheads are large (several hundred cycles per task), to account for the time-triggered execution mechanism where so-called monitors are used to dynamically update triggering dates. In [37], overheads are not even discussed, and no comparison with the sequential case without communication costs is given. More important, in both cases the objective of the method is optimization, not implementation under constraints, as it is in our case.

Contribution 1: Real-time systems compilation We propose an automatic parallelization method for applications featuring fine-grain internal parallelism. Like in [36, 37], our method adds the needed overheads to the WCET values, ensuring safety by construction without the need for subsequent schedulability analysis. However, unlike these methods, it is aimed at applications with fine-grain parallelism 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 [38], 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 [39, 40]. What fundamentally differentiates our current work from previous compilation work is the choice of performing a *safe*, worst-case timing analysis *incrementally during compilation*.

My thesis does *not* provide advances on the complexity of real-time scheduling algorithms. Recent papers provided conflicting evidence—pro [41, 42] and contra [43]—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.

Contribution 2: An implementation model for dataflow multi-threaded software. We propose a language, named InteLus, for the description of parallel multi-threaded implementations of dataflow specifications. It is a sub-set of Lustre extended with annotations representing mapping and code generation choices. Such extensions are common in existing literature, but our language and modeling approach go beyond previous work in one fundamental way: implementation models specified in InteLus are *strictly richer* than the multi-threaded C code we want to generate. InteLus allows the representation of *all* mapping decisions needed for multi-threaded code generation in our context—sequencing of dataflow blocks into threads executed by processors; code, stack, data variables to memory locations; synchronizations to hardware locks, *etc.* InteLus's representation of threads and thread synchronization is a sub-case of the C11/pthread concurrency model [44]. Therefore, C code can be obtained by selectively putting elements of the InteLus program into C and linker script syntax without making any further mapping decision—a process we call pretty-printing, exemplified in Section 1.3.

Annotations are covered by the InteLus operational semantics. This allows us to formally define (but not yet prove) the correctness of implementation models. To facilitate the definition of the correctness properties, implementation models are endowed with not one, but two semantics: the *synchronous semantics* of Lustre (which simply discards mapping annotations) and the *machine semantics*, which interprets the program and its annotations as a multi-threaded imperative program. This dual semantic nature of our implementation models enables us to envision an original approach to proving implementation correctness, based on 3 proof obligations:

1. *Refinement* Under synchronous semantics the Lustre specification and the InteLus implementation model are equivalent modulo a pipelining transformation.

2. *Mapping* Executing the implementation model under machine semantics produces the same sequences of values as those produced by the same model under synchronous semantics.

3. *Code generation* The machine semantics of the implementation model faithfully describes the behavior of the multi-threaded C code produced by pretty printing.

Previous work on implementation modeling. Much work exists on parallel application mapping (*e.g.* [23, 25, 45]), but it always involves a non-trivial code generation phase that escapes formal modeling and analysis, covering at least some of the aspects we consider here: thread construction, synthesis of synchronization and memory consistency protocols, *etc.* The language we propose *allows formal reasoning on the correctness of all these mapping and code generation decisions*. The only step not covered by our correctness formalization is pretty printing, which moves to a C syntax while preserving unchanged the thread structure and the allocation.

Our modeling language and method currently do not cover timing aspects, due to the limited time we had in the context of this thesis. Thus, it does not allow reasoning about real-time correctness. For time-triggered systems [45] this also means that we cannot reason about functional correctness. However, having such a framework to define the functional correctness of multi-threaded implementations of dataflow systems has its merits in its own right, even outside of the real-time domain. Furthermore, timing aspects can be considered in the future and built on top of this current work.

Our results are closely related to previous work on providing (operational) semantics to synchronous languages and to concurrent C. Our machine semantics is close in form to the operational C11 semantics of [11], most notably to the variant without the "promise" rule, which is adapted to DRF programs like the ones we synthesize. Main differences are that we consider a very restricted concurrency model, and that we consider a particular type of shared memory architecture and the associated memory allocation problem (which previous semantics [11, 44] do not cover). From a dataflow language perspective, my thesis includes novel operational semantics for Lustre covering the language extensions (triggers and synchronization-only variables) and the machine semantics that formalizes execution on platform at the dataflow level.

Beyond defining the formal semantics of implementation models, we formalize the proof obligations ensuring implementation correctness. This is the first step towards formal proof of correctness for the multi-threaded implementation of Lustre programs. In this sense our work is related to previous results on the formally verified compilation of dataflow synchronous languages [46, 47]. By comparison, our work extends dataflow modeling to cover multi-processor implementation issues (mutex synchronization, memory consistency), but we have not yet proved the correctness of the method.

1.3 Motivating example

Our compilation problem is similar to that solved by a compiler and linker flow for a sequential imperative language: to produce *correct executable code* that statically orchestrates the machine resources, in a *fully automatic* and *scalable* way. But there are also important differences. Following long-standing practice in the avionics industry [48], the input program—also known as the *functional specification*—is provided in a dataflow synchronous language with a *cyclic execution scheme*. As exemplified in

Figure 1.1, we use for functional specification the Heptagon language [49, 50]. The second major difference is that the functional specification—the Heptagon program—is annotated with *non-functional requirements* the implementation must respect. In our example, annotations specify *real-time* requirements (period and deadline). The third difference is that the target low-level semantics is multithreaded with explicit resource allocation and mapping for communication and synchronization.

```
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 (z:int)
4  var y,d : int; x : float;
5  let
6    deadline(1500) x = f(i);
7                   y = g(d);
8                   z = h(x,y);
9                   d = 0 fby z;
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
19
20
```

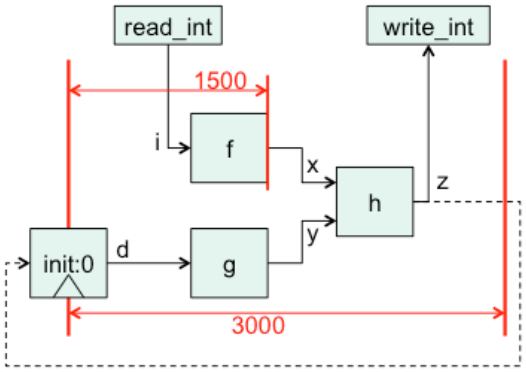


Figure 1.1: On the left, functional specification provided as a Heptagon program (in black) with non-functional requirements specified through annotations (in red). On the right, a graphical representation of the dataflow of node `main`, after inlining of node `simple`.

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 Chapter 3. For such hardware, we generate *statically scheduled, statically allocated, bare metal code* whose structure facilitates timing analysis, along the guidelines of [20]. The threads generated from our example for a dual-core target are presented in Figure 1.2. 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 of all code and data.

To ensure that generated code is not only functionally correct, but also satisfies *by construction* the real-time requirements, we rely on the compilation flow of Figure 1.3. The front-end normalizes and simplifies the input program, bringing it to a form that satisfies the requirements of *static single assignment (SSA)* form [51]. In the back-end, the sequential code of the basic dataflow blocks (`f`, `g`, `h` in our example[2]) is

[2]I/O is performed through shared variables, so `read_int` and `write_int` require no code in Figure 1.2.

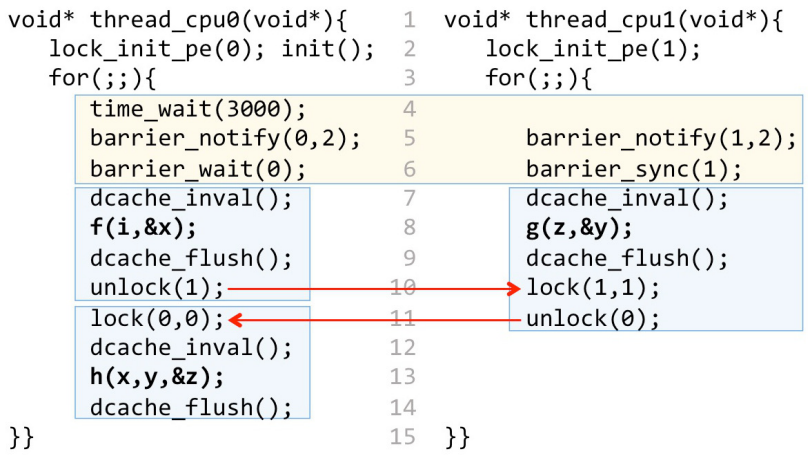


Figure 1.2: Parallel C code generated from the Heptagon program in Figure 1.1 for a two-core implementation

separately compiled and analyzed to determine their WCET, worst-case number of accesses to shared communication resources (memory banks), and memory footprint. This information is used in the parallel back-end, which performs real-time resource allocation and code generation, building the parallel threads of Figure 1.2 and the linker script of Figure 1.4.

Of this compilation flow, several components have been extensively studied in previous work: the compilation of dataflow synchronous programs to sequential code [52], C compilation [53], and WCET analysis[54]. My thesis focuses on the remaining topics: the front-end normalization phase in Chapter 2, the parallel back-end in Chapter 3, and the integration of all back-end tools around the timing model of Chapter 4 which guarantees *by construction* the respect of real-time requirements.

Formal implementation modeling The multi-threaded code of Figure 1.2 preserves a fundamentally dataflow structure, which can be represented with an extended Lustre syntax as pictured in Figure 1.4(left). Note that, even for this trivial example, the parallel C implementation and its implementation model is already quite complex. To ensure correct operation sequencing and communication on the multi-core, calls to `f`, `g`, and `h` are surrounded by:

- Mutex operations (`lock` and `unlock` API calls) ensuring that production happens before consumption, and that consumption is completed when a communication variable is reused for production. Barrier synchronization operations are also used to ensure that execution is performed in lockstep. However, we shall not insist on these, as they can be decomposed into mutex operations.

- Data cache operations (`dflush` and `dinval` API calls) implementing the memory coherency protocol ensuring that the consumer uses the correct data. Explicit cache operations are only needed on platforms without hardware cache coherency

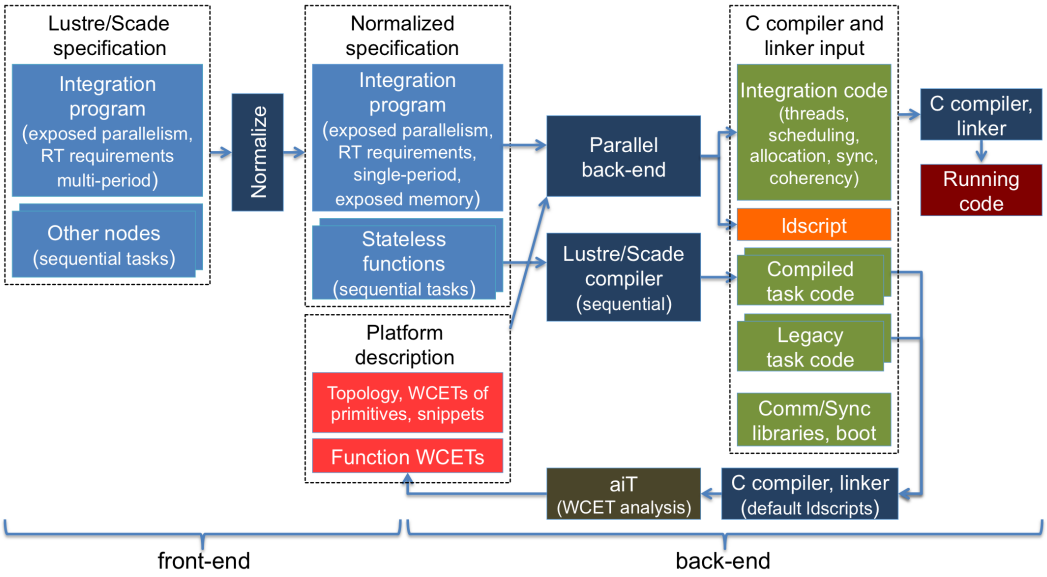


Figure 1.3: Proposed implementation flow—tools and artifacts

such as our test platform. On more classical POWER or ARM multi-cores, they can be simply discarded, as the semantics of `lock` and `unlock` (of either the pthread or C11 mutexes) ensures the needed coherency.

On the right of the implementation model, we have pictured the C code. Note that our modeling does not (yet) cover real-time behavior. For this reason, the `time_wait` API call of Figure 1.2 has been discarded here.

As mentioned above, the multi-threaded implementation consists not only of C code, but also comprises GCC annotations and the linker script defining memory allocation. Such tightly-controlled mapping is common in critical embedded systems. In avionics applications like our case study, the worst case execution time must be demonstrated for normal conditions, but the application must also be robust to "external factors". The choice of a mutex-synchronized implementation improves robustness by guaranteeing the respect of the functional semantics regardless of timing aspects. Providing execution time guarantees can then be done through tight control of memory allocation and synchronization, and through the use of hardware with good support for timing predictability. These design choices, covered elsewhere [55], reduce timing variability and facilitate timing analysis [20].

The implementation model of Figure 1.4 (left) consists of a dataflow program (in black and blue) extended with annotations defining all the aspects of its mapping (in red). The dataflow program uses some extensions to Lustre (in blue) allowing the description of synchronization. These extensions include the synchronization data type `event` and the `wait` and `done` constructs that allow the definition of sequencing constraints not implied by data dependencies. The dataflow implementation program provides a precise functional model of the execution on platform. For instance, specifi-

```
fun f:(int)->(int) at 0x20100
fun g:(int)->(float) at 0x30200
fun h:(float,int)->(int) at 0x20500
var input i:int at 0x22064 i_cpu0:int at i on cpu0
    x:int at 0x22000 x_cpu0:int at x on cpu0
    y:float at 0x32000
    y_cpu0:float at y on cpu0 y_cpu1:float at y on cpu1
    z:int at 0x22004 z_cpu0:int at z on cpu0
    d:int at z d_cpu1:int at z on cpu1
    u:event at 1 v:event at 0
    s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,t1,t2,t3,t4,t5,t6,b:event
let
  d  = 0 fby z    // memory, C code in the init0 function
  b  = barrier(b0,b1) // HW barrier semantics, no C code
  s9 = top fby s8 // thread 0 self-activation, no C code
  t6 = top fby t5 // thread 1 self-activation, no C code

thread on cpu0 at 0x20000 stack 0x30000
  s9                    [br_notify:0,2]     b0 = top
  s9          done(s0) [br_wait:0]           _ = b
  s9 wait(s0) done(s1) [inval:0x22064] i_cpu0 = i
  s9 wait(s1) done(s2)                 x_cpu0 = f(i_cpu0)
  s9 wait(s2) done(s3) [flush:0x22000]      x = x_cpu0
  s9 wait(s3) done(s4) [unlock:1]           u = top
  s9 wait(s4) done(s5) [lock:0]             _ = v
  s9 wait(s5) done(s6) [inval:0x32000] y_cpu0 = y
  s9 wait(s6) done(s7)                 z_cpu0 = h(x_cpu0,y_cpu0)
  s9 wait(s7) done(s8) [flush:0x22004]      z = z_cpu0


thread on cpu1 at 0x30000 stack 0x40000
  t6                    [br_notify:1,2]     b1 = top
  t6          done(t0) [br_wait:1]           _ = b
  t6 wait(t0) done(t1) [inval:0x22004] d_cpu1 = d
  t6 wait(t1) done(t2)                 y_cpu1 = g(d_cpu1)
  t6 wait(t2) done(t3) [flush:0x32000]      y = y_cpu1
  t6 wait(t3) done(t4) [lock:1]             _ = u1
  t6 wait(t4) done(t5) [unlock:0]           v = top
tel
```

```
ldscript fragment:
  x=0x22000; y=0x32000; z=0x22004; i=0x220064;
  stack0=0x30000; stack1=0x40000;
  .=0x20000; .bank2:{*(.text.cpu0);
                     .=0x100 ; *(.text.f) ;
                     .=0x500 ; *(.text.h) ;
  }
  .=0x30000; .bank3:{*(.text.cpu1);
                     .=0x200 ; *(.text.g) ;
  }

-------------------------------------
extern void f(int,int*);
extern void g(int,float*);
extern void h(int,float,int*);
extern int i,x,z; extern float y;
__attribute__((section(.text.cpu0)))
void thread_cpu0(){ init0(); for(;;){
  barrier_notify(0,2) ;
  barrier_wait(0) ;
  dcache_inval() ;
  f(i,&x) ;
  dcache_flush() ;
  unlock(1) ;
  lock(0) ;
  dcache_inval() ;
  h(x,y,&z) ;
  dcache_flush() ;
}}
__attribute__((section(.text.cpu1)))
void thread_cpu1() { init1(); for(;;){
  barrier_notify(1,2) ;
  barrier_wait(1) ;
  dcache_inval() ;
  g(z,&y) ;
  dcache_flush() ;
  lock(1) ;
  unlock(0) ;
}}
```

Figure 1.4: Implementation model corresponding to the example of Figure 1.2 (real-time excluded)

cation variable `y` is replaced here with three variables `y`, `y_cpu0`, and `y_cpu1` allowing the representation of the various states of the memory system where the value produced on processor `cpu0` has not yet been propagated to the RAM or to the cache of processor `cpu1`. The implementation model provides dataflow interpretations for the various API calls. For instance, in line 36 the dataflow interpretation of `flush` ensures that the local value of `y_cpu0` has been propagated onto its RAM counterpart `y`, and in line 38 the dataflow interpretation of `unlock` produces a token (the special literal `top`) that can be consumed later by the `lock` call in line 25. The equations in lines 13-16 are not part of threads. They perform the initialization of state variables, represent the self-reactivation of the thread bodies, and provide the semantics of the platform HW barrier.

Under the application-specific code structuring hypotheses detailed in Chapter 3,

the mapping information of Figure 1.4(left) is exhaustive. It allows the generation of all the C code, GCC annotations, and linker script of Figure 1.4(right) by simple pretty-printing, as no new mapping decisions are needed. For instance, the list of equations of a thread is transformed line-by-line into the sequence of function calls forming the body of the infinite loop of the corresponding C thread.

Such implementations and implementation models can be automatically synthesized, using our automatic mapping method or other existing tools [34, 56]. They can also be manually written.

1.4 Thesis plan

This thesis is organized in two distinct parts corresponding to the two main contributions identified above.

Efficient parallelization of real-time applications We start in Chapter 2 with a presentation of the specifications we handle, namely, dataflow synchronous programs. There, we present functional specification with Heptagon[49] as it exists today through an intuitive semantics of the main components of its language and the structure of the code generated by its compiler. We then propose some extensions to the language to specify real-time requirements, thus allowing the encoding of all the *platform-independent* aspects of the specification to be parallelized (both functional and non-functional). We also present in this chapter the transformations we apply to our specifications to obtain a normalized form that our back-end can process efficiently.

In Chapter 3, we focus on the hardware and software architecture. We start by identifying and discussing predictability properties required to allow the efficient parallel implementation of our applications. We describe the simple application programming interface (API) we rely on to synthesize our code, and explain how it can be implemented on some existing multi-cores, most notably on the platform we use for experimental evaluations—the Kalray MPPA 256 Bostan many-core. We also present in this chapter the structure of the implementation we generate: code organization, how the primitives are used, and how the memory is organized.

In Chapter 4, we present the timing model and the timing analysis methodology. First, we discuss the way a worst-case execution time (WCET) analysis tool for sequential code is used to perform the analysis of the sequential functions of the functional specification. This analysis must ensure that computed WCET values are independent of function code and data mapping. We then propose an interference model for the target architecture. This model precisely accounts for memory access interferences due to parallel execution. Finally, we derive a global timing analysis method for parallel software satisfying the code structuring hypotheses of Chapter 3.

In Chapter 5, we present our automatic mapping method and algorithms. We first discuss the difference between our approach, traditional compilation, and traditional real-time implementation processes. We then present the real-time scheduling algorithms, which also perform memory allocation and the allocation of computations to

the various cores in a way that incrementally ensures the respect of non-functional requirements. This is only possible by incrementally performing, at scheduling time, the timing interference analysis. We also present here the code generation method ensuring, and in particular the original synchronization synthesis method.

We conclude this part with the experimental results, presented in Chapter 6. After presenting our two industrial use-cases, we propose a method for quantifying the efficiency of parallelization algorithms for real-time applications, and then we present our results.

Formal implementation modeling We start in Chapter 7 with the presentation of a new language called InteLus. This language is a stripped-down version of Lustre extended with dataflow constructs allowing the representation of pure synchronization and operation sequencing, and with non-functional annotations defining the mapping of the data, computations, and communications onto the hardware resources (RAM, CPUs, mutexes, *etc.*).

Chapter 8 defines the dataflow semantics of Intelus. Two distinct semantics are defined. The first one is a synchronous semantics, whose only particularity is that it follows a structural operational semantics (SOS) paradigm. The second one interprets Intelus programs as asynchronous Kahn process networks (KPN). A strong semantics preservation property links the two semantics. We also introduce in this chapter a series of necessary properties all correct implementation models must satisfy.

Chapter 9 introduces the machine semantics of Intelus, which takes into account the non-functional annotations of the language. This semantics, which defines the state of the execution platform and state transitions, is necessarily platform-dependent. We formalize the correctness of implementation models with respect to functional specifications. Correctness properties cover two aspects: the absence of run-time errors during execution on the platform, and the semantics preservation between the functional specification and the implementation model.

We conclude this thesis and present the perspectives offered by this work in Chapter 10.

Part I

Efficient parallelization of real-time applications

Chapter 2

Platform-independent specification in Heptagon

To provide the functional specification of a real-time system, we use Heptagon [49], a dataflow synchronous language close in form and semantics to Lustre [13] and its industrial dialect Scade [3]. We also extend Heptagon with non-functional annotations allowing the representation of platform-independent non-functional requirements. Heptagon, extended with these annotations, allows us to define all aspects of our system that are independent from details of the execution platform. We call this the *platform-independent specification* of the system.

We start this chapter by presenting the core concepts of the Heptagon language, starting with the principles of the synchronous paradigm. Then, we introduce the language annotations that allow the specification of non-functional requirements. Finally, we explain how platform-independent specifications are normalized before being given to our parallelization and code generation algorithms.

2.1 Functional specification and code generation with Heptagon

Heptagon is both a language and a research compiler. In this thesis we shall extensively discuss both – the language as the support of our specifications, and the compiler as a key tool in our code generation flow.

In this section, we provide a brief introduction to synchronous programming, an overview of the subset of Heptagon we use, and a brief presentation of the code generation conventions we use.

2.1.1 Synchronous programming

Synchronous programming originated in the real-time community [57, 58]. It was designed as a way to non-ambiguously specify the functional part of complex real-time embedded control systems. These systems have very particular specification needs.

Reactive and real-time systems. First of all, they are *reactive systems* [59], whose role is to respond, in a timely and continuous manner, to events produced by their environment at the pace set by the environment. In the description of a reactive system, the focus is less on the description of the computation to be performed, but on the description of the interaction between system and its environment, and by extension between sub-systems. A key property of most reactive systems is concurrency. Concurrency occurs at both specification and implementation level. At specification level, various sub-systems may require interaction with the environment (or with other sub-systems) possibly at different paces. At implementation level, software reactive systems often have multi-task implementations.

Concurrency complicates both the design and the analysis of systems. As sub-systems/tasks compete for access to resources, undesired phenomena such as deadlocks, starvation, or data races can occur. More generally, concurrent execution can lead to functional non-determinism. This is bad news, as in the industrial engineering of real-time embedded control systems, determinism is a highly desirable property, at it can largely facilitate activities at all steps of the design cycle – specification, implementation, verification and validation (V&V).

Real-time systems are reactive systems where the system/environment interaction or the interaction between sub-systems must satisfy precise timing requirements. Typical requirements are:

- An airbag must be deployed at most 80ms after a shock event has been detected.
- The cyclic computation and interaction of a flight controller must happen on a period of 50ms, to ensure the controllability of the plane.

Real-time systems are often described and implemented as a set of elementary behaviors without internal concurrency, usually called *tasks*, which are activated (or interrupted, disabled) following specific rules depending on environment input, task interaction, a timed pattern, or a combination thereof. Specifying a real-time system mainly consists in specifying the set of tasks and their (concurrent) activation pattern.

The synchronous programming paradigm [60] and the synchronous languages are one of the solutions proposed for the programming of reactive and real-time systems. In this paradigm, *system execution is semantically divided into a sequence of completely ordered execution instants*. Concurrency is confined inside execution instants, and each instant must involve a statically bounded number of computations[1]. Furthermore, the concurrency inside an instant satisfies the requirements of the *static single assignment form* [51], which enforces[2] a *causal* ordering between the computations: inside each instant, a variable is assigned at most once, and no variable is assigned a value after being read inside the same instant. Causality implies the determinism of the computations

[1] Some languages have been introduced that do not have this restriction [61], but they are rather exceptions from the common use (at least in the industry).

[2] In conjunction (as in virtually all work in compilation) with the property of dominance of variables definitions over their uses.

inside an instant, which in turn implies the global determinism of correct synchronous programs. Data races and deadlocks are not possible.

Synchronous languages provide constructs allowing the natural description of elementary computations and their hierarchic composition involving relations of (data and control) dependency, concurrency, priority, and (conditional) activation.

To represent activation, the synchronous paradigm proposes the notion of *logical clocks*. A logical clock represents the sequence of execution instants where some computation or communication is performed, or where a variable is assigned a value. Every operation and variable of a system has exactly one clock, describing its (deterministic) activation/communication pattern. The *base clock* of a system represents the set of all execution instants of the system. All other clocks are derived from the base clock through subsampling.

From its beginning, synchronous reactive programming has been declined in several languages, the three best known being:

- Esterel [62], an imperative programming language with explicit expression of concurrency and hierarchic control. Its emphasis on control flow is particularly suited for control-dominated applications in robotics or hardware design.
- Lustre [13], a declarative dataflow programming language. Scade [63] is a industrial dialect of Lustre used in the engineering of critical systems, *e.g.* in avionics.
- Signal [64], another dataflow language which facilitates the definition of systems with multiple, unrelated logical clocks.

Throughout this thesis, we will see the synchronous paradigm through the lens of Lustre. In particular, the next section will illustrate the concepts of the dataflow synchronous paradigm through the description of the Heptagon dialect of Lustre.

Related formalisms. Synchronous languages are not the only class of formalisms used to describe (reactive) real-time systems.

Synchronous data-flow (SDF) [65] and derived formalisms (such as CSDF [66], BDF [67]) also feature a deterministic and cyclic execution model. However, while in synchronous languages all computations and communications are synchronized with respect to the global base clock, the SDF execution model is fundamentally asynchronous and decentralized. An SDF block can be executed as soon as enough input is available, and execution is not constrained with respect to global events.

Traditionally used in real-time scheduling, task models [68, 69] focus on the coarse-grain organization of the application—the partitioning of its code into tasks, and the real-time properties of the tasks (durations, periods, deadlines, *etc.*). The functional behavior of the tasks is abstracted away, as task models are not designed as full-fledged programming languages.

While synchronous languages are not concerned with the *duration* of computations, time-triggered languages such as Giotto [70] (with its current LET [71] dialect) or PsyC [22] introduce duration as a first-class language construct, to allow its explicit specification and analysis.

Heavily used in industry, Simulink[72] is a language and tool for design and simulation of control systems, in discrete or continuous time. Its syntax is similar to that of dataflow synchronous languages like Lustre. However, the semantics of Simulink is defined by the simulation engine, and it varies depending on numerous simulation parameters, some of which trade semantic determinism and consistency for simulation speed. Thus, while certain sub-sets of Simulink can be considered synchronous, the whole language (in all its simulation configurations) is not. The situation is similar for LabView [4].

2.1.2 Syntax and intuitive semantics of Heptagon

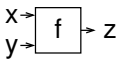
A Heptagon program is a textual representation of a dataflow graph with dedicated constructs specifying activation. The language syntax allows to express dependence or concurrency of operations, conditional activation and communication, and hierarchical program composition.

This section will present *the subset of Heptagon we need in our work* using simple examples to illustrate the intuitive semantics of the constructs.

Equation

In Heptagon, vertices of the dataflow graph are called *equations*. The simplest kind of equation is the *function call*, of which an example is provided in Figure 2.1(left), along with its informal graphical representation (middle) and an execution trace (right). This

```
x = f(x,y);
```



	Cycle					
	0	1	2	3	4	5
x	1			3		0
y	1			-1		3
z	f(1,1)			f(3,-1)		f(0,3)

Figure 2.1: Function call equation in Heptagon. Syntax (left), graphical representation (middle), six cycles long example trace (left).

function takes two inputs and produces one output. The execution of the equation is cyclic. At each execution cycle where it is activated, the equation reads one input from each of `x` and `y` and produces `f(x,y)` on the variable `z`. The function call equation synchronizes its inputs and outputs. If at least one input is present in an execution cycle, then all inputs are present and all outputs are produced. Formally, all input and output variables have the same clock. We call this a *clock constraint*.

To facilitate the use of common mathematical operators, Heptagon allows their use with natural infix syntax. Thus, if a function call equation must compute the addition of two integers, we will write:

```
z = x + y;
```

While the function in our example has one output, Heptagon functions can have any fixed number of outputs. Such functions are called using the following syntax:

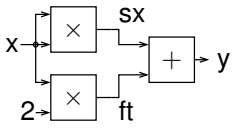
```
()    = output0(x,y,z); (* Function with no output and three inputs *)
(a,b) = output2(x) ; (* Function with two outputs and one input *)
```

Note the OCaML-style comments, which can be nested (unlike in C/C++).

Dataflow

Equations can be connected through variables to form a dataflow allowing the computation of more complex functions.

```
 y = sx + ft ;
sx = x * x ;
ft = x * 2 ;
```



	0	1	2	3...
x	1		3	0...
sx	1		9	0...
ft	2		6	0...
y	3		15	0...

Figure 2.2: Dataflow computing $y = x^2 + 2x$

We provide in Figure 2.2 the dataflow encoding the computation of the mathematical function $y = x^2 + 2x$. In this dataflow, the input variable `x` is used thrice. Variables `sx` and `ft` are internal to our dataflow. They are outputs of multiplication equations and input to the addition equation. Variables can be inputs of any number of dataflow nodes (or none), but each variable that is not an input of the dataflow must be produced by exactly one dataflow node.

This example also shows a constant value `2` in the second equation. Constants can be seen as functions without inputs that produce the same value at each execution cycle where the output is needed.

Recall from Section 2.1.1 that the dataflow must satisfy the structuring requirements of the SSA form, and in particular the causality requirement. In the textual representation of the dataflow, equations need not be ordered to satisfy causal dependencies. The causality analysis and the ordering of equations for code generation is the task of the compiler. In our example, the equations are in reverse topological order, which would produce an incorrect result should the fragment be interpreted as a C program (but the fragment is correct and will produce the correct C code with Heptagon). More generally, the semantics of a Heptagon program do not depend on the order of its equations.[3]

The following program is incorrect due to a violation of the causality requirement:

```
x = f(y);
y = g(x);
```

Function `f` depends on the execution of `g` through the variable `y`, while `g` depends on `f` through `x`. Execution is therefore not possible, and the program is rejected by the compiler. Similarly, a function cannot have as both input and output the same variable.

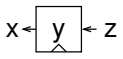
[3]The form of the C code generated by the Heptagon compiler may depend on the equation order, but without changing the computed function.

Function equations that are not causally ordered in the dataflow are concurrent. They can be executed in any order or in parallel. In Figure 2.2, the two multiplication equations are concurrent.

State

In the example of Figure 2.2, the computation of a cycle only depends on the value of the input `x`. Heptagon also gives the programmer the ability to specify stateful behaviors by storing *state* values from one execution cycle to the next. This is done using the operator `fby` (followed-by).

```
x = y fby z ;
```

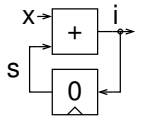


Cycle	0	1	2	3...
y	0	3		-5...
z	1	2		3...
x	0	1		2...

Figure 2.3: The `fby` (followed-by) statement of Heptagon

The `fby` statement has two inputs (denoted `y` and `z` in Figure 2.3) and one output (denoted `x` in our figure). The input and output variables of the construct are present in the same cycles (they have the same clock). In the first cycle where it is present, `x` takes the same value as `y`. In all other cycles where it is present, `x` takes the previous value of `z`. Operationally, the execution of `fby` takes place in two phases: it produces the value of `x` at the beginning of a cycle, allowing execution to proceed, and then stores the value of `z` for the next cycle, at the cycle end.

```
i = x + s ;
s = 0 fby i ;
```



Cycle	0	1	2	3	4...
x	1	0		3	2...
s	0	1		1	4...
i	1	1		4	6...

Figure 2.4: State in Heptagon

As pictured in Figure 2.4, the `fby` allows the specification of feedback loops without breaking the causality requirements. This example implements an integrator—`i` is the sum of all previous values of `x`.

Expressions. Syntax restrictions.

The syntax of Heptagon allows grouping dataflow equations into expressions. For instance, instead of:

```
y = 0 fby z ;
x = f(y) ;
```

one can simply write `x = f(0 fby z)`. It is always possible to expand a complex expression into a set of dataflow equations containing each exactly one dataflow operator.

Grouping operators into expressions or expanding expressions into simpler equations does not change the semantics of a program, which remains subject to the same causality requirements and clock constraints.

To simplify the definition of the semantics and to reduce the complexity of code generation, we simplify throughout this thesis the syntax of the language. We focus on the dataflow core of the language, and do not cover Heptagon constructs such as the modular reset or the automata [49]. Furthermore:

- We assume that each equation contains at most one dataflow operator.[4]
- We assume that the first argument of a `fby` operator is a constant.

The last two constraints can be removed with light to moderate effort. We did not do it, because the language sub-set we consider covers the needs of our industrial use cases. Covering modular reset and automata poses no theoretical problem—as they can always be translated into pure dataflow—but providing an efficient practical solution may require more work (and use cases).

Modularity

Once built, a dataflow—*i.e.* its equations and variables—can be encapsulated under the form of a dataflow *node*. The node is the programming unit of Heptagon. A node definition provides:

Name An identifier, similar to a function name, which must uniquely identify the node in its namespace.

Signature The set of input and output variables along with their types and possibly clocks.

Local variables They are grouped in the `var` section of the node definition. Local variables are invisible from outside the node.

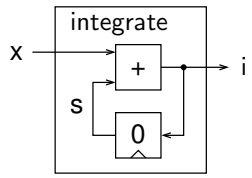
Dataflow A set of dataflow equations using only the input, output, and local variables of the node. These equations must provide a value to all local and output variables.

In Figure 2.5, we define a node called `integrate`, which encapsulates the dataflow of Figure 2.4. This node has one input `x`, one output `i`, and one local variable `s`. The code of the node – the equations – is placed between the keywords `let` and `tel`. The node `integrate` can now be instantiated in other programs with the function call syntax presented previously.

A node is said to be *stateful* when it contains at least one `fby` operator, either directly, or in one of the nodes it hierarchically instantiates. A node that is not stateful is called a *function* and its definition may be prefixed with `fun` instead of `node`. The Heptagon compiler checks the correct use of `fun` and `node` during compilation.

[4] Equations without an operator are mere copy equations of the form `x = y;`

```
node integrate (x:int) returns (i:int)
var s : int
let
  i = x + s ;
  s = 0 fby i ;
tel
```



Cycle	0	1	2	3
x	1	0	3	2
s	0	1	1	4
i	1	1	4	6

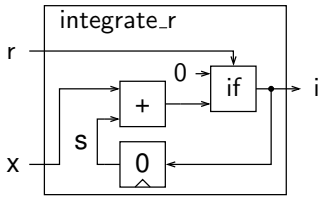
Figure 2.5: Node definition in Heptagon

Data-dependent behavior

We define in Figure 2.6 another node that extends the functionality of the integrator. Its behavior is as follows: The node has a new input `r`, of Boolean type. In a cycle where this input is true, `o` is set to 0. In cycles where `r` is false, the value of `o` is increased by that of `i` (as the integrator does).

To implement this new behavior we introduce another primitive construct of Heptagon: the ternary operator `if then else`. The operator takes three arguments. The first one is a Boolean expression. In cycles where this expression evaluates to true, the value of the second argument is output. When the expression evaluates to false, the value of the third argument is output. As with `fby` we shall consider in this thesis only a simplified syntax, where expressions can be only variables or constants.

```
node integrate_r (r:bool, x:int) returns (i:int)
var s : int ;
    t : int ;
let
  i = if r then 0 else t ;
  s = 0 fby i ;
  t = s + x ;
tel
```



	0	1	2	3	4...
r	F	F		T	F...
x	1	0		3	2...
s	0	1		1	0...
i	1	1		0	2...

Figure 2.6: Simple data-dependent behavior: integrator with reset

It is required that all three arguments and the outputs have the same clock—all inputs are present at each cycle where an output is produced. Therefore, `if` is not a conditional execution operator. It only chooses between the last two inputs based on the value of the Boolean. In this, the `if then else` operator of Heptagon significantly differs from `if` operators of classical imperative languages.

Typing

Heptagon is a statically typed language. All variable declarations—inputs, outputs, and local variables—must be explicitly given a data type. A typing analysis of the

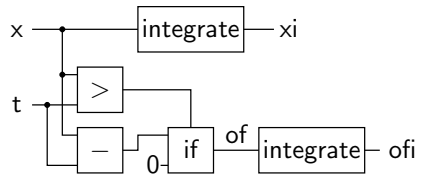
compiler will ensure the consistency of typing between the variable and node signature definitions. Programs not satisfying this requirement are rejected.

In Figure 2.6, the variables `x`, `i`, `s`, and `t` have type `int`, and `r` has type `bool`. The type checking phase of the Heptagon compiler will check that the output and last two inputs of the `if then else` equation have the same type, and that the first input is a Boolean. It will also check whether the inputs and outputs of the `+` operator are all integers, and that the input and the output of the `fby` have the same type.

Node instantiation

Once defined, a node be used in the definition of other nodes, using the same syntax as function calls. Thus we can hierarchically build complex specifications.

```
node twice(x,t:int) returns (xi,ofi:int)
var c:bool; d:int;
let
  xi  = integrate(x) ;
  ofi = integrate(of) ;
  c   = x > t ;
  d   = x - t ;
  of  = if c then d else 0 ;
tel
```



Cycle	0	1	2	3	4
x	1	5	-3		2
t	4	3	1		1
xi	1	6	3		5
of	0	2	0		1
ofi	0	2	2		3

Figure 2.7: Two instantiations of node `integrate` into node `twice`

In Figure 2.7 we use the node `integrate` (from Figure 2.5) to define a new dataflow. The node is *instantiated* twice. The first instance integrates the values of `x` and the other the overflow of `x` with respect to threshold `t`.

Semantically, node instantiation can be seen as the inlining of the instantiated node dataflow into that of the instantiating node. During instantiation, local variables of the instantiated node are renamed to ensure uniqueness. Thus, instances of the same node are entirely independent from each other. In particular, they do not share their states. The state of a node consists of the state of its `fby` equations and that of all the stateful nodes it instantiates.

Code generation choices associated with modularity may impose further causality semantic constraints on the behavior of a Heptagon program. Typical methods for generating sequential code (presented below, in Section 2.1.3) produce one C function per node. One node instantiating another is translated at C level into one function calling another. In turn, this requires that the semantic actions associated with the instantiated node can be executed together, in an atomic fashion.

From a compilation point of view, this means that causality analysis is performed at the level of each node, separately, without taking into account the behavior of the instantiated or instantiating nodes.

```
node hiddenfby(i:int) returns (o:int)
```

```
var v:int ;
let
  v = 0 fby i ;
  o = f(v) ;
tel
node inst(i:int) returns (o:int)
var v:int ;
let
  o = hiddenfby(v) ;
  v = o + i ;
tel
```

In the previous example, such constraints mean that the program is rejected during causality analysis, which finds a causality cycle inside node `inst`, not being able to determine that the `fby` construct inside `hiddenfby` breaks this cycle.

To control code generation and allow the compilation of such programs, Heptagon introduces the `inline` construct, presented in Section 2.1.4.

Conditional execution and communication

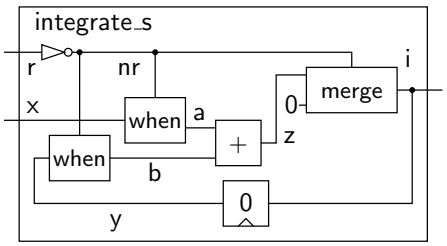
While the use of `if-then-else` is sometimes sufficient, there are common situations where genuine conditional execution is needed:

- Access to physical devices such as sensors or actuators. When accessing such devices:
  - Commands may be required at precise instants, *i.e.* not at every cycle.
  - Different command functions may be needed depending on the execution context. For instance, a send or a receive command (but never both inside a cycle) on a simplex network interface.
- Limited platform resources. In embedded contexts, the resources of the execution platform are often limited, not allowing the execution of all node instances at each cycle. It is then important to reduce the amount of code executed at each cycle. In classical real-time contexts, these requirements resulted in the use of multi-period task systems. In our synchronous context, multi-period execution is represented with periodic clocks controlling the activation of nodes. One particular problem here is to ensure that activation clocks balance the computation load over successive execution cycles, so that peak load does not exceed the platform resources.

To cover such situations, Heptagon provides two specialized dataflow primitives—`when` and `merge`—exemplified in Figure 2.8, where we provide a new version of the resetable integrator of Figure 2.6.

The input/output behavior of node `integrate_s` of Figure 2.8 is identical to that of node `integrate_r`. However, its internal behavior is very different: the addition is

```
node integrate_s (r:bool; x:int) returns (i:int)
var y,z,a,b : int;
    nr : bool ;
let
 nr = not r ;
  i = merge nr z 0 ;
  a = x when nr ;
  b = y when nr ;
  z = a + b ;
  y = 0 fby i ;
tel
```



Cycle	0	1	2	3
r	F	F	T	F
x	1	0	3	2
y	0	1	1	0
a	1	0		2
b	0	1		0
z	1	1		2
i	1	1	0	2

Figure 2.8: Conditional execution in Heptagon

executed only during cycles where its value is used to produce an output. This conditional execution and communication behavior is obtained using the dataflow operators `when` and `merge`.

The `when` operator sub-samples its input variable. Its two inputs must be present in the same cycles (they have the same clock). The second input is a Boolean condition. The `when` operator outputs a value only during cycles where its condition is present and true. The output value is the current value of its first input. Any computation using the output of the `when` will be triggered only when the output has been produced. In Figure 2.8, variables `a` and `b` are produced and the addition is executed only when `r` is false. When `r` is true, the two `when` operators do not produce any value. This means that `a`, `b`, and `z` – as the addition does not execute – are *absent* during those cycles.

The `merge` operator has three inputs and one output. The first input is a Boolean. If the Boolean is true, `merge` outputs the value of its second input. If it is false, it outputs the value of the third input. If it is absent, then no value is output. The difference with the `if then else` operator is that, while the latter requires all its inputs to be present at the same time, `merge` require its second and third inputs to have exclusive presence, or more specifically, that the second is present when the condition is true and the third when its not.

Note in our example how the constant 0 used as second argument in `merge` is assigned the clock that matches the needs of the equation in which it is used. Implicitly, it will be present at cycles where `r` is present and true.

Clock calculus In Figure 2.8, variables `a` and `b` are present at the same cycles, thus ensuring the correct functioning of the adder. This is achieved by sub-sampling on the same condition. Similarly, the `merge` operation requires its inputs to be present in mutually exclusive cycles.

The Heptagon compiler checks the respect of such *clock constraints* through a static analysis called *clock calculus*. The clocks explicitly specified by the program and the clock constraints associated with each equation are then used to build a clock for each

variable and equation of the program. The process is organized as a *type inference*. Programs are only accepted for code generation if every variable or equation can be successfully assigned a type so that all clock constraints are satisfied.

In the Heptagon language sub-set considered in this thesis, `when` is the only statement allowing the creation of new clocks, by sub-sampling. Subsampling starts in each node from the *base clock, or tick* of the node, denoted ".". This allows the non-ambiguous notation of each clock inside a node as an expression of the form ". `when s1 when s2`...`when sn`", where `s1`... `sn` are the Boolean signals used for sub-sampling. In Figure 2.8, the clock of `a` is ". `when nr`". Type inference proceeds in a constructive fashion. For instance, once `a` is assigned clock ". `when nr`", the clock constraints associated with equation "`z = a + b`" are used to infer that `b` and `z` have the same clock. Alternatively, the clock of `z` can be done using the requirements of the `merge` equation.

2.1.3 Code generation API

Lustre dialects, such as Heptagon, are traditionally compiled to sequential code such as C, Ada or Java. We will only cover here the C code generation.

With few exceptions [73], compilers for Lustre dialects follow the same conceptual translation scheme. For each dataflow node, a compiler generates:

- A "step" function that performs the computation of one instance of the node inside one cycle of execution. If two node instances execute in one cycle, two calls to the step function will be performed.
- Only if the node is stateful:
  - A structured type representing its state. This is a C `struct` with fields corresponding to `fby` statements and stateful nodes instantiated in its dataflow.
  - An initialization function that resets the node state to its initial value, as specified by the constants of `fby` statements.

In our work, we respected this code generation convention. However, we modified aspects of the Heptagon call convention to facilitate our mapping, code generation, and timing analysis work. We did not explore the effects of such code generation choices on the performance (size, speed) of generated code.

Figure 2.9 provides an example of an Heptagon node (top left), the result of its compilation to C using the standard Heptagon compiler (right) and the result of compilation when using the convention used throughout this thesis (bottom left). In both cases, the compiler generates:

- A C `struct` definition for the memory state of the node, containing the integer state of the `fby` equation and the internal state of the `integrate` node instance.
- A reset function for setting the state variables to their initial values. Note the recursive call allowing the initialization of the instance of `integrate`.

```
(* File func.ept *)
open Lib
node f(x:int) returns (o:int)
var d:int, r:bool;
let
  o = integrate(r,x);
  r = x > (d * 2) ;
  d = 0 fby o;
tel
```

```
/* File func.h */
typedef struct Func__f_mem {
  int d;
  Lib__integrate_mem integrate;
} Func__f_mem;

typedef struct Func__f_out {
  int o;
} Func__f_out;

/* File func.c */
void Func__f_reset(Func__f_mem* self) {
  Lib__integrate_reset(&self->integrate);
  self->d = 0;
}

void Func__f_step(int x,
                  Func__f_out* _out,
                  Func__f_mem* self) {
  Lib__integrate_out Lib__integrate_out_st;
  int v;
  int r;
  v = (self->d*2);
  r = (x>v);
  Lib__integrate_step(r, x,
              &Lib__integrate_out_st,
              &self->integrate);
  _out->o = Lib__integrate_out_st.i;
  self->d = _out->o;;
}
```

```
/* File func.h */
typedef struct Func__f_mem {
  int d;
  Lib__integrate_mem integrate;
} Func__f_mem;

/* File func.c */
void Func__f_reset(Func__f_mem* self) {
  Lib__integrate_reset(&self->integrate);
  self->d = 0;
}

void Func__f_step(int* x, int* o,
                  Func__f_mem* self) {
  int v;
  bool r;
  v = (self->d*2);
  r = ((*x)>v);
  Lib__integrate_step(&r, x, o,
                      &self->integrate);
  self->d = (*o);;
}
```

Figure 2.9: A Heptagon program (top left), code generated by the standard Heptagon compiler (right), code generated by our modified Heptagon compiler (bottom left)

- A step function.

The differences between the translation schemes only concern the step function. In the original translation, outputs of a node are grouped into one C `struct` which is transmitted by reference to the step function, whereas inputs are passed by value. In our case:

- To allow variable-level allocation by parallelization algorithms, outputs are no longer grouped together, but are passed by reference individually.
- To facilitate WCET analysis and code generation (especially in the presence of `at` annotations defined below) we also pass by reference inputs of the functions.

Note that in C the names of the functions are prefixed by the capitalized name of the file there are defined and have either `_step` or `_reset` as suffix. Various instances of

the node must be independent so the reset and step function takes as last argument a pointer to the state corresponding to the executed instance. While in Heptagon the equations can be given in any order, the C code must respect the causality so the C code lines associated to various equations have been reordered during scheduling.

Note how local variables that are not part of the state are translated into local C variables of the step function (allocated on the stack).

Modules and interfaces

By convention, a Heptagon source code file (with extension `.ept`) also defines a *module* containing all the types, constants, functions, and nodes of defined in the file. The name of the module is obtained from the source file name by removing the extension and capitalizing the name.

Heptagon provides two mechanisms allowing the use of the definitions of a module inside another:

- Opening of the used module inside the user module. For instance, in Figure 2.9 (top left) module Func opens module Lib to allow the use of node `integrate`.
- Prefixing a name with the module declaring it. This approach does not expose all definitions of the used module, but can result in less readable code.

The implementation of a module, including types, constants, functions, or nodes, can also be provided in C. To this end, the module objects must be declared in a Heptagon interface file, with extension `.epi`. Each interface file declares a module following the same conventions as for `.ept` files. However, interface files only contain the signatures of the declared objects, assuming availability of the code nodes and the compiler assumes the availability of their code at link time, following the same code generation conventions as if the code were generated by Heptagon.

If we want to implement `integrate` in C, we would need to define the interface file `lib.epi` containing the declaration:

```
node integrate (bool;int) returns (int)
```

Then, we would need to provide C data structures and functions with the same signatures as if it were generated by Heptagon:

```
typedef struct Func__integrate_mem {
  int s;
} Func__integrate_mem;
void Func__integrate_step(bool*, int*, int*, Func__integrate_mem*);
void Func__integrate_reset(Func__integrate_mem*);
```

In addition to respecting this interface, provided implementation code must not contain side-effects, be statically bounded in execution time, memory, and stack usage, to comply with the requirements of the synchronous paradigm.

2.1.4 Non-functional annotations

While Heptagon, as a dialect of Lustre, is mainly dedicated to functional specification, it also provides a few constructs specifying non-functional requirements.

These constructs do not change the semantics (the function) of a program[5]. Instead, they can change code generation modularity and variable allocation, which result in changes in scheduling and code generation, to the point where some programs are accepted or rejected (but never changing the semantics of accepted programs). In this section, we present the two non-functional annotations of standard Heptagon.

Inlining

```
node subs2_1 (x:int) returns (o:bool)
var y:float;
let
  y = p(x) ;
  o = q(y) ;
tel
node subs2 (x:int,i:float) returns (o1:bool,o2:bool)
let
  o1 = h(i) ;
  inline o2 = subs2_1(x) ;
tel
```

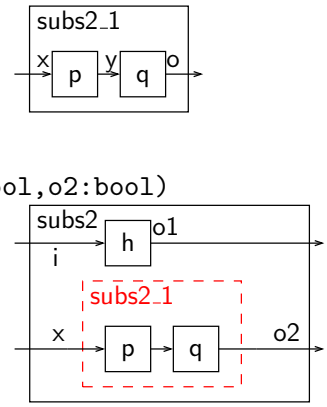


Figure 2.10: Inlining in Heptagon

Inlining is enforced using the `inline` keyword. When placed in front of a node instance, it states that it must not follow the standard modular compilation convention. Instead, the dataflow of the inlined node is exposed at the level of the instantiating node.

Inlining can be seen as a source-to-source transformation of the Heptagon source that opens the inlined instances prior to C code generation. The inlining process ensures the uniqueness of the inlined variables, so the semantics is unchanged. However, modularity is affected, which potentially affects causality, as explained in Section 2.1.2 (paragraph "Node instantiation").

In the example of Figure 2.10, node `subs2` instantiates node `subs2_1`. The dataflow obtained through (conceptual) source-to-source translation is the following:

```
node subs2 (x:int,i:float) returns (o1:bool,o2:bool)
var y:float ;
let
  o1 = h(i) ;
  y  = p(x) ;
```

[5]Hence the name non-functional.

```
  o2 = q(y) ;
tel
```

Code generation is performed on this inlined version of the original node, so that the step function generated for `subs2` directly calls the step functions of `p` and `q`.

Allocation requirements

Allocation annotations were introduced in Heptagon [50] as language support for memory optimization. They allow specifying that two or more variables of a node (inputs, outputs, or local) share the same memory location, materialized as a C-level variable.

```
node incr (x:int at r) returns (y:int at r)
tel
  y = x + 1 ;
tel
```

Figure 2.11: Allocation constraint in Heptagon

In Figure 2.11, allocation annotations (in red) specify that variables `x` and `y` share the same memory location, labelled `r`. An allocation annotation has the form "`at a`" for some identifier `a`, called the *location*. All input, output, or local variables of a node sharing the same location must share the same C variable.

Allocation annotations impact code generation in two fundamental ways:

- When they occur on interface variables, they may change the code generation convention. When generating code for the example in Figure 2.11, variables `x` and `y` can no longer be distinct in the interface of the generated step function. Instead, a single variable `r` replaces both.
- Scheduling must ensure that sharing of the same C variable by several dataflow variables does not change the semantics of the application. In our example, the allocation constraint means that the value of `x` is overwritten in the process of computing `y` on location `r`. When node `incr` is instantiated, the scheduling of the instantiating node must ensure that the variable provided as input to `incr` is never used after the call to `incr` in the generated C code.

2.2 Extensions and integration specification

In this section, we extend Heptagon with additional annotations allowing the expression of real-time requirements, and we present an original way of using the pre-existing inlining annotations to specify which part of the concurrency of a Heptagon specification can be exploited by our parallelization method.

These additions allow us to use Heptagon to represent all the platform-independent functional and non-functional aspects taken as input by our parallelization method. We call the resulting specification the *platform-independent specification*. We also call it

integration specification to emphasize its position in the design methodology. Indeed, we do not use Heptagon to specify the behavior of sequential tasks. Instead, we use it to specify the way tasks are integrated—the way they communicate, synchronize, the way they are activated.

This section provides an original contribution of my thesis.

2.2.1 Exposing parallelism

Classical work in real-time scheduling [8, 7] 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 does not make this distinction. Its naturally concurrent programming style allows the joint specification of 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. One fundamental reason for this concerns WCET analysis. To provide accurate estimates, WCET analysis tools must be applied on functions whose size allows the computation (through static analysis) of precise cache hit patterns. Furthermore, too fine a grain of parallelism would result in prohibitive synchronization costs.

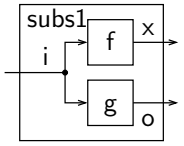
We do *not* provide here a method for automatically choosing which part of the parallelism of a Heptagon program is exposed to the parallelization algorithms. Instead, we provide a method for specifying it. The actual choice is left to the user. The mapping of the exposed parts itself is done through our method, and the choice of what to expose has a direct impact on the efficiency of the resulting implementation (as we will see in Chapter 6).

We make the convention that parallelization algorithms only exploit the parallelism of the topmost node of the application, that we refer to as *main node*. This includes the parallelism exposed through inlining of nodes into the main node. In particular, our method allows exposing all the parallelism of a hierarchic Heptagon specification. Indeed, if all node instances are inlined, then the inlining process proceeds recursively.

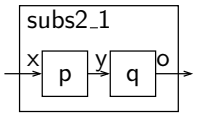
In the example of Figure 2.12, the main node is `main`. As it represents a closed system, it has no input or output variables. Instead, I/O is performed by means of explicit hardware operations encoded with external functions called from the dataflow.

In our example, there are several levels of nested nodes. We assume every node instantiated but not defined in the figure comes from module `Extern`. Without inlining the main node does not allow much paralellisation. We choose to inline `subs1` and `subs2`. Inlining `subs2_1` is possible, but since it contains no parallelism we chose not to. Then, the result of inlining is illustrated by the dataflow on the right. All nodes not exposed to parallelization are compiled directly to sequential code.

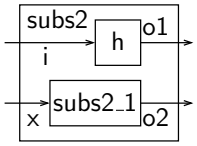
```
open Extern (* definitions of f,g,p,q,h,read_*,write_* *)

node subs1 (i:int) returns (x:int,o:bool)
let
  x = f(i) ;
  o = g(i) ;
tel
```




```
node subs2_1 (x:int) returns (o:bool)
var y:float;
let
  y = p(x) ;
  o = q(y) ;
tel
```




```
node subs2 (x:int,i:float) returns (o1:bool,o2:bool)
let
  o1 = h(i) ;
  o2 = subs2_1(x) ;
tel
```




```
node main () returns ()
var i1,x,o3:int; i2:float;
    o1,o2:bool;
let
  i1 = read_int() ;
  i2 = read_float() ;
  inline (x,o1) = subs1(i1) ;
  inline (o2,o3) = subs2(x,i2) ;
  () = write_bool(o1) ;
  () = write_bool(o2) ;
  () = write_int(o3) ;
tel
```

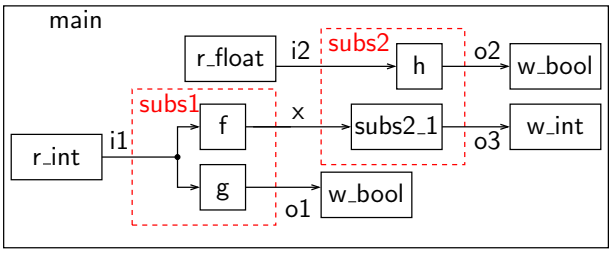


Figure 2.12: Inlining in a hierarchical Heptagon program

Inlining produces a flat (non-hierarchic) dataflow graph that is a natural entry point for parallelization or real-time scheduling algorithms. Our use of Heptagon and its inlining annotations allows the construction of this artefact using existing language constructs and transformations, but applied in an innovative way.

2.2.2 Real-time requirements

We extend Heptagon with three more annotations to represent real-time requirements: period, release date and deadline. The choice of requirements is not new [27], the original point is their representation with annotations of the Heptagon program, and the integration of their automatic processing in the Heptagon compiler.

```
period(300)
node main () returns ()
var i1,x,o3:int; i2:float;
    o1,o2:bool;
let
  i1 = read_int() ;
  i2 = read_float() ;
  deadline(100) (x,o1) = subs1(i1) ;
  (o2,o3) = subs2(x,i2) ;
  () = write_bool(o1) ;
  () = write_bool(o2) ;
  () = write_int(o3) ;
tel
```

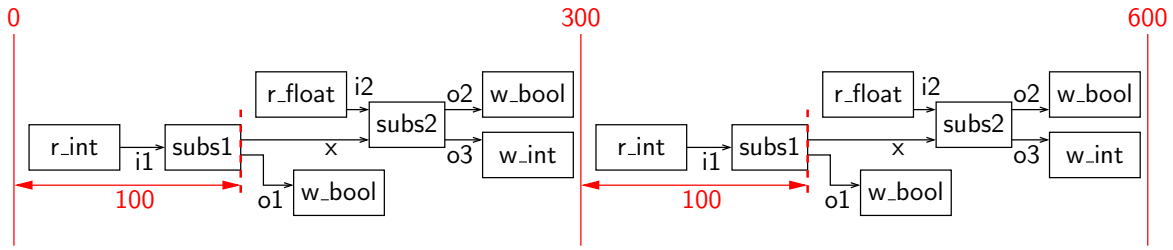


Figure 2.13: Real-time annotations over a Heptagon program

The *period* of the system is represented with an annotation of the main node. We require that all the computations and communications of an execution cycle of this node (including all computations and communications of nodes it instantiates) are executed within the prescribed period, assuming strict periodicity. In other words, for a given period p and given execution cycle n, every computation of cycle n starts after date $n \times p$ and ends before date $(n + 1) \times p$. This definition can be extended to allow the pipelining of cycles following the approach of [38].

Release date and *deadline* requirements can be set on each of the statements of the integration specification. These requirements impose timing constraints within each cycle. 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 recursively transferred onto all the statements of the inlined node. Note that the period specification imposes implicit

release date and deadline requirements even on nodes not having them explicitly.

The time unit (e.g., ms, μs, CPU cycle) is not specified. It is the task of the user to ensure that all values are specified using the same unit, and that all WCET analysis results are converted to this unit.

We provide an example in Figure 2.13. It is the example of Figure 2.12 without the inlining annotations (which changes exposed parallelism) and with period and deadline annotations. The application period is 300 time units, and the node `subs1` has a deadline constraint of 100 time units after the start date of each cycle.

Recall that non-functional requirements do not change the functional specification. As a corollary, the deadline on `subs1` also applies to `r_int` due to the data dependency requiring that `r_int` completes execution before `subs1` in every cycle.

2.2.3 Integration specification

We shall call integration specification a Heptagon specification whose main node has no input or output variables, where exploitable parallelism has been exposed through inlining annotations, and where real-time requirements have been annotated on the main node and possibly other nodes inlined into the main node.

An integration specification provides the system-level description of a real-time application destined to be implemented on parallel hardware. It corresponds to the top-level of the application, built only of node calls, up-sampling and down-sampling operations, and `fby` operators, along with non-functional annotations annotations. Code below this level of specification is standard Heptagon code (or interfaces towards C code) that can fully use the features of the language and is compiled to sequential code. The focus of this thesis shall be on the implementation of the integration specification.

2.3 Specification normalization

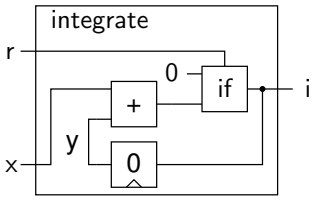
While integration specifications use a restricted sub-set of Heptagon, their structure can be further simplified to facilitate the application of mapping and code generation algorithms. This section will introduce two source-to-source transformations we apply on integration specifications to produce a *normalized integration specification* which is the input of the parallelization algorithms of the following sections.

2.3.1 Exposing node states

As explained in Section 2.1.2, the state of a node consists in all values passed by `fby` statements from one cycle to the next. When a node contains no `fby`, we say that it is a *dataflow function*. It is always possible to transform a stateful node instantiation into a dataflow function instantiation. The transformation is done by exposing the node state using dataflow variables and a `fby` primitive.

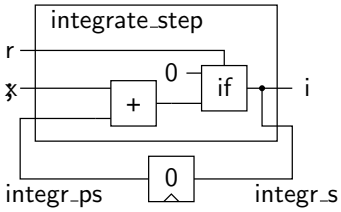
To understand the principle of the transformation, consider the following node instantiation:

```
i = integrate(r,x) ;
```



The transformation takes the `fby` equation and moves it outside the node, which becomes a function. The function has one more input and one more output than the original node, which are used to connect to the `fby` equation (using two new local variables). The result of the transformation is:

```
(integr_s,r,x) = integrate_step(integr_ps,i) ;
integr_ps = 0 fby integr_s ;
```



For a general stateful node, the process must expose the state of all `fby` equations, hierarchically. This still requires the use of only one new delay equation and one input and one output signal to the new function, all having the record type grouping all state elements.

Exposing the state of a node in this way is always possible. In fact, all compilers of Lustre dialects we know of (save the one of [73]) perform this transformation implicitly, in order to produce the three C language objects described in Section 2.1.3:

- The state `struct` is the record grouping all state elements.
- The `reset` function defines the initial value of the state.
- The `step` function adds to the signature of the node the state argument, which can be seen as a pair of an input and output argument allocated on the same location with an `at` annotation.

During the ASSUME project, the process has been automated (at our request) by our collaborators of the Inria PARKAS team inside as part of the Heptagon compiler.

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. This allows a unified approach to the allocation of all data memory, and also allows the representation of allocation under the form of variable annotations in the specification once the states are exposed.

2.3.2 Hyper-period expansion

Through the primitives introduced in Section 2.1.2, the Heptagon language provides conditional control constructs allowing the representation of complex execution modes, data-dependent control, and multi-period execution.

```
period(100)
node main () returns ()
var
  cnt : int; c_g,c_h : boolean;
  x,y,z : int;
let
  (* computation of clocks *)
  cnt = 0 fby ((cnt+1)%4) ;
  c_g = (cnt%2)==1 ;
  c_h = cnt==0 ;
  (* task activation *)
  x   = f(z) ;
  y   = g(x when c_g) ;
  ()  = h(x when c_h) ;
  (* communications *)
  z = 0 fby (merge c_g y (z when (not c_g))) ;
tel
```

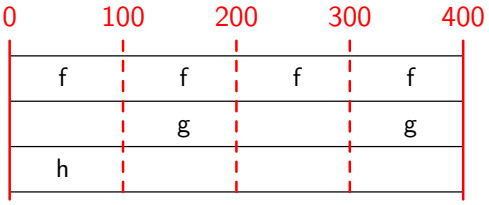


Figure 2.14: Example of multi-period specifications using sub-sampling

We illustrate multi-period specification with the example of Figure 2.14. It features three tasks/functions `f`, `g` and `h`. The variables `c_g` and `c_h` act as activation conditions for `g` and `h`, triggering them every two, respectively every four cycles. The three tasks have therefore periods 100, 200, and 400, with a stricly periodic activation pattern. Specific to the use of a pure synchronous modeling approach is the requirement that each task completes execution in the same 100 time units cycle where it started, even though its period can be much larger.[6]

The *hyper-period* of a multi-period specification is the least common multiple of the nodes periods. In our example, the hyper-period is of 400 time units. The hyper-period extension is the transformation that unrolls the original integration program until its period is equal to the hyper-period. This transformation is usually accompanied by a propagation of constants that simplifies the conditional control of the application. In our case, since conditional control only enforces periodic execution, and since unfolding is performed over the hyper-period, the result of the transformation presented in Figure 2.15.

In the unrolled specification, `f`, `g` and `h` are respectively instantiated four, two and one time. However, the code replication also has the good consequence of completely making disappear the complex sub- and over-sampling operators.

To preserve the real-time requirements unchanged from the original specification, release dates and deadlines are used to enforce the implicit release and deadline requirements, which also confine each dataflow function to one 100 time units interval.

[6]Other formalisms, such as Giotto[70] or PsyC[22] allow more flexibility in the use of larger periods.

```
period(400)
node main () returns ()
var
  x1,x2,x3,x4: int at x;
  y1,y2,z0: int at y;
  z0,z1,z2 : int at z;
  fs0,fs1,fs2,fs3,fs4 : f_state at fs;
  gs0,gs1,gs2 : g_state at gs;
  hs0,hs1 : h_state at hs;
let
               deadline(100) (fs1,x1) = f_step(fs0,z0) ;
  release(100) deadline(200) (fs2,x2) = f_step(fs1,z0) ;
  release(200) deadline(300) (fs3,x3) = f_step(fs2,y1) ;
  release(300)               (fs4,x4) = f_step(fs3,y1) ;
  release(100) deadline(200) (gs1,y1) = g_step(gs0,x2) ;
  release(300)               (gs2,y2) = g_step(gs1,x4) ;
               deadline(100)      hs1 = h_step(hs0,x1)

  init<<fs>> fs0 = f_init fby fs4;
  init<<gs>> gs0 = g_init fby gs2;
  init<<hs>> hs0 = h_init fby hs1;
  init<<y>>   z0 = 0 fby y2 ;
tel
```

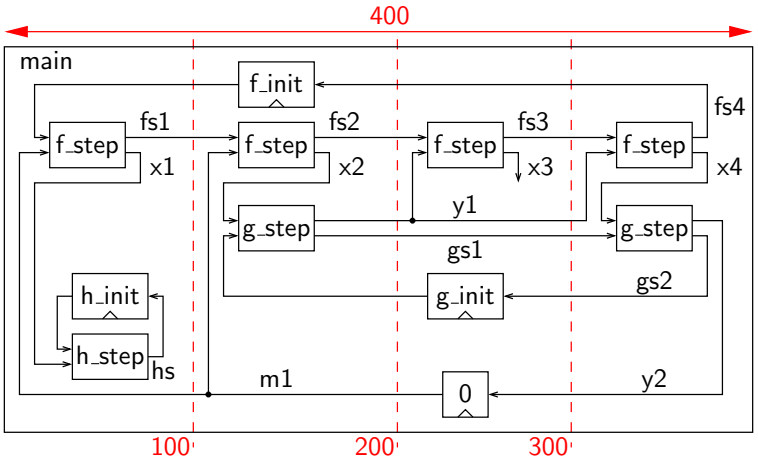


Figure 2.15: Result of the hyper-period expansion transformation of Figure 2.14

The replication of function instances is accompanied by a replication of dataflow variables. The states of the nodes must also be exposed and chained through the different instances of a same original node to respect the initial specification semantics. However, this replication does not result in increased memory requirements at code generation time. Indeed, all replicas of an original specification variable share the same location.

No real-time requirements are needed on the delay equations of the exposed state variables. Given the use of allocation annotations, these nodes require no code generation, as input and output are both allocated to the same memory location. In fact, we will require that the same property holds for all `fby` equation, so that no code or extra variable is required in their encoding. This property will be further discussed in Section 8.3.2.

Hyper-period expansion does not change the functional semantics or the non-functional requirements of the integration specification, nor does it increase the complexity of the generated code. It can increase the size of the generated code (exponentially, in the worst case)[7]. It facilitates mapping by removing all conditional control related to periodic execution.

While the expansion itself is deterministic, it can also be accompanied by a relaxing of the release date and deadline requirements for nodes with larger periods. For instance, the deadline requirement on `h` could be removed, potentially allowing its computation to take more than 100 time units. However, this relaxation changes the non-functional requirements of the system, and should be carefully evaluated. Such a relaxation is not performed in our experiments.

2.3.3 Normalized integration specification

We call *normalized integration specification* a Heptagon integration specification where the node states have been exposed and hyper-period expansion performed. This specification is the input of the mapping method of the next chapters.

A normalized integration specification satisfies the properties of Static Single Assignment form [51]—there are no hidden data dependencies, each variable is assigned exactly once, and every variable is defined before it is used in a computation. Thus, it is a good starting point for optimized resource allocation, such as memory optimizations using register allocation techniques.

[7] For example, a specification with tasks of periods 30, 50 and 70ms would have a hyper-period of 1050ms, replicating the code for tasks of the fastest period 35 times. Fortunately, in most practical applications the tasks have mostly harmonic periods, so that the hype-period is often (in practice) equal to the largest period.

Chapter 3

Hardware and software architecture

With the increasing need for computational power in real-time systems, parallelization becomes inevitable. However, unlike in general-purpose (GPC) or high-performance computing (HPC), the evaluation of real-time software is not based on performance alone. In an embedded/reactive context with a cyclic execution model, improving performance often consists in improving computation throughput—number of computation cycles per time unit—or decreasing the latency of some computation flow—the duration between the start of its first operation to the end of the last. In classical real-time research, reducing the latency of selected computation flows was achieved by carefully prioritizing the execution of computations. However, as explained in the introduction, we believe that for the computationally-intensive software of the future prioritizing alone is not a solution. We therefore need to consider the classical approaches from GPC and HPC—addition of hardware resources and the use of efficient resource allocation algorithms performing parallelization, pipelining, speculative execution, *etc.* In GPC and HPC, experience shows that such performance-improving solutions lead to a decrease of both:

- Functional predictability, through complexification of the software, which leads to more bugs and at the same time makes it difficult to understand and analyze.
- Temporal predictability, through the use of hardware and software dynamic resource allocation mechanisms whose timing is difficult to analyze (untractable in practice) and difficult to statically and tightly bound. Such mechanisms include out-of-order pipelining, speculative execution, hardware memory coherency, dynamic resource OS-based allocation and scheduling.

While loss of predictability was acceptable in many GPC and HPC applications in exchange for increased *average case* performance, this is not true in critical, hard real-time applications, where functional and temporal predictability are a requirement.

To improve *guaranteed* performance, existing parallelization methods must be revisited, first of all the hardware and software mechanisms they rely upon.

This topic is by no means new [55]. In this chapter, we detail the requirements on hardware and software architecture allowing the application of our shared memory parallelization method. We also explain that the architecture of our test platform

satisfies these requirements due to a combination of predictable hardware (the Kalray MPPA256 Bostan many-core), very simple and predictable basic software consisting in 7 communication and synchronization primitives, and finally a very simple, predictable, and efficient structuring of the generated code.

3.1 Hardware architecture

When parallelizing under hard real-time constraints, we have to provide static timing guarantees on our implementations. The parallelization method I propose in this thesis can be applied on multi-core architectures satisfying two fundamental properties:

H1 Processing cores and the memory hierarchy must allow the computation of static worst-case execution time (WCET) bounds for code executed on a single core, *in isolation*, *i.e.* without interaction nor interferences with other core or the environment.

H2 The on-chip communication and synchronization interconnect must allow the computation of upper bounds on the worst-case response time (WCRT) of parallel software. To ensure a certain degree of separation of concerns and to ensure tractability, we require here that WCET bounds for pieces of sequential code executed in the presence of interferences[1] 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 [74].

The design of such architectures is an active field of research (*e.g.* [55, 75]), but most resulting multi-cores are still at the level of model simulations and FPGA implementations. This is potentially due to the fact that they use non-standard features such as exposed pipelines, exposed memory hierarchies, time-triggered arbitration, *etc.* These features make it more difficult or less efficient to deploy general-purpose software on them, which limits their adoption in high-volume GPC or HPC markets. In turn, this limits the ability to turn them into off-the-shelf products.

We know of only two off-the-shelf multi-core processors with strong support for timing predictability:

- The Infineon TC27X family of multi-cores for the automotive market, using cores with a TriCore architecture.
- The Kalray MPPA family of many-cores.

Both of them combine predictable processing pipelines with the absence of hardware coherency and partially exposed memory hierarchies allowing the programming of timing-predictable behaviors. At the same time, they avoid the use of non-standard features such as specialized caches or time-triggered arbitration, thus allowing the use of a traditional compilation toolflow for executable code generation.

[1] From other cores, due to concurrent accesses to shared resources such as memory, I/O devices, shared buses, *etc.*

Experimentation platform

In this thesis, experiments were conducted on a Kalray MPPA 256 Bostan many-core [76, 77]. More specifically, we target a single compute cluster of the many-core.

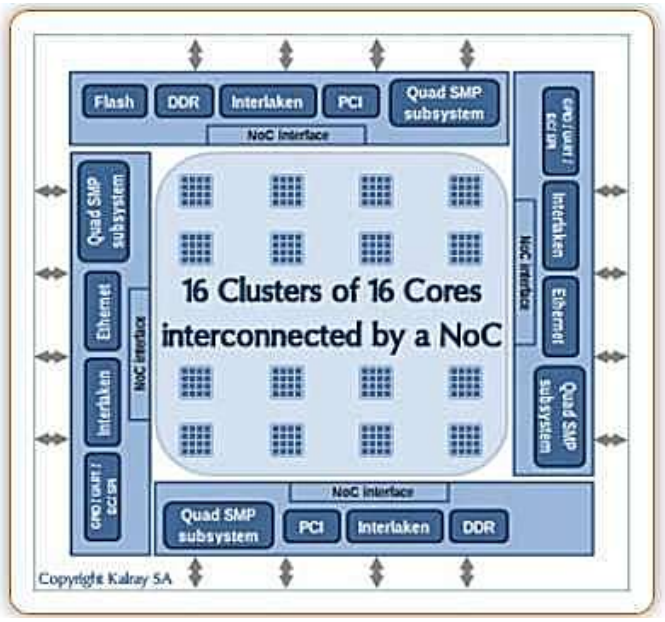


Figure 3.1: Kalray MPPA-256 chip organization

The structure of the Kalray MPPA 256 Bostan many-core is pictured in Figure 3.1. It features 172 processing cores and 16 resource management (RM) cores running at 400MHz. The cores are organized into 16 compute clusters, or tiles and 4 I/O clusters (on the borders). Each compute cluster contains one RM core and 16 processing cores. From our perspective, the RM core is not programmable.[2] The 16 compute clusters and the 4 I/O clusters are connected only through a network-on-chip (NoC).

Each core has separate L1 code and data caches (9kB each). In addition to cores, each compute cluster also contains 16 memory banks of 128kB each, for a total of 2MB of local tile memory. Processors of a cluster are connected to the memory banks through a full crossbar local interconnect allowing non-interfering accesses from different processing cores to different memory banks. Of the 2MB of total memory, the last 2kB are ROM, and the rest are SRAM.

Of the other hardware devices of a compute cluster we shall use throughout this thesis only the C-NoC router and the timers. The C-NoC router is one of the NoC access controllers. It provides hardware synchronization of very low latency between the processing cores. It is controlled through memory-mapped registers.

Each core contains a special register, a timer updated by the hardware at every tick of the hardware clock. Within a cluster, the timers are all synchronous but not in phase, meaning they can have different initial values, but cannot drift apart from each other.

[2] It handles system-level functions such as booting/initialization, error recovery, *etc.*

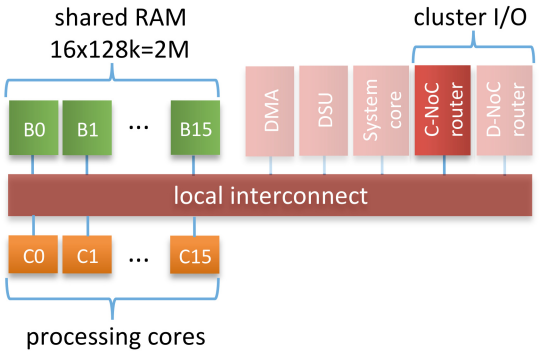


Figure 3.2: Structure of a Kalray MPPA 256 Bostan compute cluster

To arbitrate between concurrent requests from different processing cores to the same memory bank, the full crossbar local interconnect uses a fair (Round Robin) arbitration policy. Accesses from the NoC and the RM core are prioritary, but we shall not consider them in this thesis. Arbitration is done at the entry of each memory bank.

The caches of each core are not shared and there is no hardware cache coherency mechanism. Caches replacement policy is LRU, which provides a better timing predictability than other popular policies [78].

We assume in this work that the execution platform is timing compositional [74]. While we have no formal proof that our test platform satisfies this property, we rely on the fact that this architecture is marketed as such and on arguments in the same direction from previous work [77].

3.2 System software

Allocation of CPU time and memory in SMP architectures like our experimentation platform is often delegated to an operating system (OS), which takes allocation decisions at execution time. Such an approach has two inconvenients:

- Operating systems are complex pieces of software, difficult to predict and analyze.
- On-line allocation and scheduling decisions are known to render timing analysis more difficult and imprecise.

In our efforts toward predictable and efficient implementations, we choose instead to synthesize *bare-metal implementations where all allocation and scheduling decisions are taken before execution*. This choice facilitates precise timing analysis. Furthermore, generated implementations are simple, facilitating both code review and static analysis.

To allow the implementation of the fully static allocation and scheduling patterns, we assume that the platform provides the following services, implemented with library functions:

- Global synchronization barriers allowing high-precision synchronized triggering of computations on all cores.

- A restricted implementation of pthreads/C11 mutexes [44] that does not support concurrent `lock` waiting on a given mutex, nor the unlocking of an already unlocked mutex. Other common names for these constructs are binary semaphores or locks.
- Cache coherency operations allowing us to ensure that a value produced on one core can be read on another.
- Synchronization with real-time.

All common multi-core platforms provide these services, but the hardware sub-systems they use may significantly vary, especially if high efficiency is sought. For instance, cache coherency may have some hardware support (as in classical POWER or ARM multi-cores) or not (as on our test platform). For this reason, the exact set of primitives depends on the target platform. On our test platform, the low-level library API we rely on contains the 7 primitives listed in Figure 3.3. Their implementation was provided by Amaury Graillat from Kalray in the framework of the ITEA3 ASSUME collaborative project.

Call	**Precondition**	**Action**
`time_wait(`*d*`)`	*d* is a later date	Returns when the hardware clock reaches *d*
`lock(`*l*`,c)`	No other lock is waiting for *l*	Consumes the semaphore/lock *l* when available. The second argument *c* provides the id of the core on which the primitive is called, to reduce the duration of the call.
`unlock(`*l*`)`	Lock *l* is not available (locked)	Unlocks semaphore/lock *l*
`global_barrier_sync(c)`	Barrier is initialized	Terminates on all cores at the same cycle, after the last core calls the primitive. The argument is the core identifier.
`global_barrier_reinit(`*n*`)`	No core is waiting on the barrier	Initializes the global barrier for another cycle. The argument is the number of cores to synchronize.
`dcache_inval()`	None	Invalidates the whole data-cache
`dcache_flush()`	None	Flush the whole data-cache

Figure 3.3: Hardware primitives for MPPA-256

We require these primitives to have bounded execution time[3] and generate a bounded number of interferences on memory accesses. The second objective is facilitated by the hardware locks provided by the CNoC controller of the Kalray compute clusters, which do not require spinlock implementations (as high-frequency memory sampling can create large and/or unpredictable interferences).

[3]Counting only effective execution, not waiting time.

The rest of this section will detail the functioning of the primitives of our API.

3.2.1 Synchronization with real-time

We need a single primitive `time_wait(d)`, where `d` is the real-time date to wait. A call to the primitive will stall the execution of the core until the core timer reaches date `d`. Note that on Kalray the timer counters run downwards.

We assume in the context of our work that the primitive will always be called before the date passed as argument. In contexts where timing analysis does not provide a static guarantee of respect of respect of this requirement, or when required to do so (*e.g.* for certification purposes) the implementation of `time_wait` may verify that this requirement is met and, in cases it is not true, raise an error. Such an implementation amounts to actively checking for deadline misses. However, my work in this thesis focuses on providing static guarantees that such a situation never occurs. Of course, if the chosen certification methodology requires it, health monitoring can be use to provide further safety guarantees.

3.2.2 Event-driven synchronization

Point-to-point synchronization between cores is performed using a set of mutexes and the primitives `lock` and `unlock` which have the natural pthread/C11 semantics [44], subject to further restrictions, detailed below. These restrictions make the primitives easier to implement and result in very low synchronization overhead.

A call to `unlock(l)` will put the state of mutex `l` to unlocked/true. A call to `lock(l)` will stall the execution until mutex `l` is unlocked. Then, it changes its state back to locked/false and terminates. Like in C11, behavior is undefined when calling `unlock` on an already unlocked mutex. The choice of thread to unlock is not specified when two or more `lock` calls are wait on the same mutex (but we will ensure during code generation that this situation never occurs).

Given that the Kalray platform does not have hardware support for cache coherency, our mutex operations have no effect on memory coherency (unlike traditional pthread/C11 semantics of mutexes).

To reduce the duration of the `lock` operation on Kalray, we have added a second parameter to `lock`, which is the identifier of the core executing the operation. To allow implementation on platforms with native pthread/C11 mutexes, this second argument can be safely removed.

The use of the CNoC router hardware locks also means that we only dispose of 127 distinct mutexes on the test platform.

Global barriers also use the services of the CNoC router. Immediately after each barrier (materialized into calls to `global_barrier_sync(c)` on each core), one of the cores must perform the reinitialization of the barrier, before any core calls `global_barrier_sync` again.

3.2.3 Memory coherency

The pthread/C11 model has been essentially defined for cache-coherent multi-core architectures where mutex operations also enforced memory coherency. Our test platform has no hardware support for cache coherency. We can simulate by software means the coherency behavior of C11 mutex operations. However, doing so would severely impact the performance of our implementations. For this reason, we prefer to completely dissociate synchronization from memory coherency. We do so by introducing separate primitives performing explicit cache operations.

We use two primitives that on the test platform have low computational cost:

- `dcache_inval()` invalidates the content of the entire data cache of the calling core, ensuring that potentially outdated values present in the case are discarded, and that memory reads after the invalidation take their value directly from RAM.

- `dcache_flush()` forces the writing of all locally-modified variables to the RAM, by forcing the flush of all values in the write buffer of the core executing it. Control is given in sequence once the flush is complete.

On cache-coherent platforms with native pthreads/C11 implementations of mutex operations—such as ARM or POWER multi-cores—the cache operations can be safely removed, because the coherency function is ensured by mutexes, according to the C11 semantics [44].

3.2.4 Requirements on compilation tools

To apply the parallelization method described in the following chapters, we not only make requirements on the hardware and basic software libraries. We also require that the C compiler provides us tight control over code generation:

- We require the ability to fully control memory allocation of our bare metal implementation using linker scripts following the standard syntax and conventions of GNU ld.

- We assume the compiler supports three types of `gcc` attributes:

  - Section annotations that allow setting the code or data section of a function or variable.
  - The `always_inlined` attribute requiring that a function is always inlined.
  - The `noinline` attribute requiring that a function is never inlined.

  In the absence of inlining-related annotations, the heuristics of `gcc` can make decisions that our method does not support.

3.3 Structure of an implementation

In critical real-time systems, the respect of execution time bounds must be demonstrated for normal conditions, but the system must also be robust to errors. We rely on event-driven semaphore-based synchronization to ensure the two following properties:

- Guarantee the functional semantics in the presence of timing errors.
- Ensure some degree of run-time robustness to timing errors through scheduling elasticity (one component overstepping its timing bounds can under certain conditions be compensated by other components executing faster than provisioned).

The respect of execution time bounds for normal conditions can then be achieved through tight control of scheduling, memory allocation, and synchronization [34]. This is different from time-triggered approaches [22, 31], which place timing predictability first, potentially at the expense of functional determinism and robustness.

Building upon the presentation of the platform HW and low-level library API, we shall now present the software architecture we enforce through synthesis on our applications. Recall that we synthesize statically scheduled, statically allocated, bare metal, parallel C code. This section will cover both the form of the generated C code, with an emphasis on the usage of API primitives, and the memory organization, enforced through linker scripts. Details on the choices made leading to this general software organization are discussed in Chapter 5.

3.3.1 Threads running in lockstep

We synthesize parallel implementations consisting in multiple concurrent threads sharing the same memory space. Each CPU is assigned exactly 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. We use a global barrier to synchronize 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 of all threads performs one execution cycle of the normalized integration program described in Chapter 2. The loop bodies of different threads synchronize and communicate with each other and with real-time using the primitives of our low-level API.

We illustrate this with the small example of Figure 3.4. This is a single-period specification with three tasks/functions `f`, `g` and `h`, an input `i`, and an output `z`. I/O is handled with the I/O functions `read_int` and `write_int` which generate no code, but inform the code generator that the C variables associated with `i` and `z` are updated cyclically in memory.[4] The period of the program is 3000 time units and function `f` has a deadline at 1500 time units after the beginning of each cycle.

An implementation of this specification on a dual-core processor is provided in Figure 3.5. There are two threads, one for each core. Each thread contains some code initializing the hardware locks (`lock_init_pe`). Core 0 also initializes its timer, and sets state variables to their initial values. This is done by function `init` which in our

[4] *e.g.* by dedicated hardware components.

```
open Externc (* Declaration of task functions f,g,h *)
open Lib_io (* Declaration of I/O functions read_*,write_* *)

period(3000) node main () returns ()
var
  i,y : int ; x : float ;
  z,d : int at z ;
let
                  i = read_int();
                  ()= write_int(z);
  deadline(1500) x = f(i);
                  y = g(d);
                  z = h(x,y);
  init<<z>>       d = 123 fby z; (* no extra variable, no copy *)
tel
```

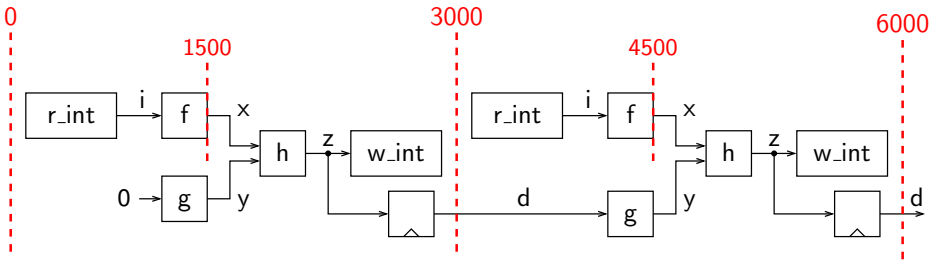


Figure 3.4: Single-period integration specification (top) and unrolling of its first two cycles of execution (bottom)

case sets `z` to 123. The variable `time` is set by `time_init` to the current value of the hardware timer of the core. We implicitly assumed here that the time units of the specification in Figure 3.4 are clock cycles.

The initializations are followed in each thread by an infinite loop containing the global barrier and all the code implementing one cycle of the dataflow specification.

The code in orange corresponds to the global barrier. The barrier is (re)initialized by the first CPU and the synchronization primitive is called on both cores to ensure the lockstep execution. Core 0 also uses `time_wait` to enforce the period of the specification: the variable `time` is adjusted to the date of the beginning of the next synchronous cycle by substracting[5] the period (in hardware cycles). We then wait for this date before reaching the global barrier. We do not consider the potential underflow of the timer for two reasons: First, on the test platform the timer register is 64-bit wide and the clock frequency is 400MHz, which gives gives enough runtime to current critical applications ($> 2^{34}$ seconds, with the timer initialized at the maximum value). Second, software

[5]On the test platform, the timer is not incremented, but decremented every cycle. This is why we do not add the period as one might expect.

```
void* thread_cpu0(void* unused){
   lock_init_pe(0); init();
   time_init(&time);
   for(;;){
      global_barrier_reinit(2);
      time-=3000; time_wait(time);
      global_barrier_sync(0);
      dcache_inval();
      f(i,&x);
      dcache_flush();
      unlock(1);
      lock(0,0);
      dcache_inval();
      h(x,y,&z);
      dcache_flush();
   }
}
```

```
void* thread_cpu1(void* unused){
   lock_init_pe(1);

   for(;;){
      global_barrier_sync(1);
      dcache_inval();
      g(z,&y);
      dcache_flush();
      lock(1,1);
      unlock(0);
   }
}
```

Figure 3.5: Dual-core implementation of the specification in Figure 3.4

work-arounds exist for such underflows.

The remaining code of the loop bodies contains the calls to the dataflow functions `f`, `g`, and `h` (in blue), surrounded by cache coherency primitives (in green) and mutex synchonizations (in red). Note that cache invalidations and flushes are systematic, immediately before and after each function call. The absence of any form of optimization on these operations is justified by their low cost on the test platform. Synchronizations are performed to ensure that the dataflow functions are executed in the correct order.

The blue boxes identify the *code snippets* generated by our method and tool for each of the dataflow functions. No code exists in the loops outside the code snippets and the global barrier code. Each snippet contains exactly one call to a dataflow function, the coherency operations, and a bounded number of synchronization operations (possibly none). The order of the snippets in the implementation, enforced by the synchronizations and the sequence, is a partial order stricter than the one of the data dependencies in the dataflow model.

In our example, `f` and `h` are executed in sequence on the same core so the dependency on `x` is respected. A synchronization (on mutex 0) between `g` and `h` ensures the dependency on `y`. Unlike in traditional code generation schemes for synchronous languages, the `fby` operators generate no code and no extra variable. This is possible under the assumptions detailed in Section 8.3.2, which allow allocating the input and output variables of a `fby` on the same C-level variables. In our case, Heptagon variables `z` and `d` are implemented by C variable `z`. The order between successive computation cycles is enforced by the global barrier.

The reader may have noticed that the synchronization on the lock 1 in the example serves no purpose. This is an artifact of the synchronization synthesis we use, which is presented in detail in Chapter 5.

We do not use synchronization to isolate computation from communication phases, either in the execution of the whole system, as in bulk synchronous parallel (BSP) based approaches [79, 3], or in that of individual nodes[19]. 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. Another approach providing the same isolation effect would be to ensure that the execution of each node fits into the processor cache. In this case, memory replication is not needed, and the communication phases consist of cache operations alone [17]. However, when user-specified nodes do not fit into the cache (like in our case studies of Chapter 6), heavy modifications are needed in the C compiler to automatically slice the original functions into smaller code intervals. This poses a problem in avionics contexts, because it would require not only the development, but also the qualification of the modified C compiler (a long and costly process). Furthermore, BSP-like barriers synchronize all threads, thus adding artificial constraints that make real-time scheduling problems more complex.

3.3.2 Explicit memory mapping

To bound and minimize inter-core interferences due to concurrent memory accesses, our mapping method produces not only C code for each core, but also a static memory mapping for every object of the implementation. This includes the dataflow variables, the code and data of the dataflow functions, the code of the threads and their stacks. This mapping is specified using a linker script.

The allocation of dataflow functions and variables is an output of the dataflow real-time scheduling and allocation process. The allocation of threads, thread stacks, system code,[6] and pre-allocated data (*e.g.* I/O variables) are chosen prior to real-time scheduling using a heuristic meant to reduce interferences.

To explain how this memory allocation heuristic works, we consider again the mapping of the specification in Figure 3.4 onto a dual-core. Prior to real-time scheduling of the dataflow nodes, the organization of the test platform (the Kalray compute cluster) is that of Figure 3.6. The first three banks (of indices 0-2) are used for system code—the low-level libraries automatically loaded under the standard configuration of the platform. The last bank is dedicated to potentially pre-allocated data, such as I/O variables, except for the very last 2kBytes, which are used by ROM.

We allocate the code, local data, and stack of each thread on the same bank. If the number of threads is smaller than the number of free banks, then we allocate each thread on a different bank. If not, at most two threads can share the same bank. We allocate thread code and local data at the base of the memory bank and thread stack at

[6] Low level operations needed by `gcc`, like the software implementation of division

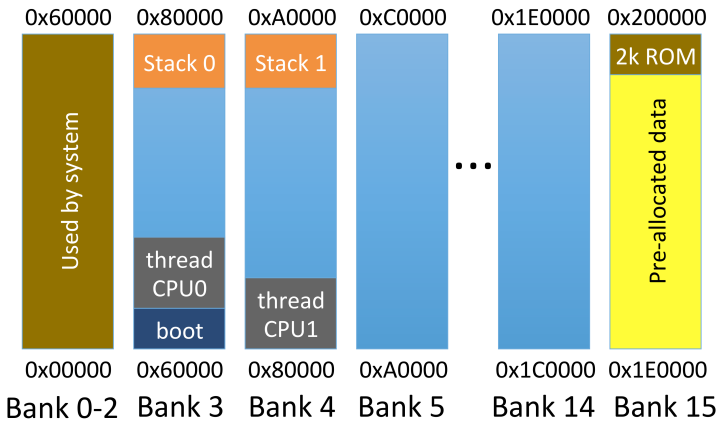


Figure 3.6: Memory organization before the mapping phase

the end. Given the shape of the generated code and the application size[7] we compute an upper bound on the size of thread code and local data, allowing its allocation. The size of the stack can also be over-approximated prior to scheduling and code generation based on the worst-case needs of the nodes. The small boot code executed on core 0 to start the two threads is allocated on bank 3. The part of the linker script corresponding to these allocation choices is provided in Figure 3.7.

```
1  . = 0x1e0000;
2  .i ALIGN(4) : {
3     *(.i)
4  }
5
6  . = 0x61000 ;
7  .text_thread0 ALIGN(64) : {
8     thread_cpu0.o(.text)
9  }
10 .data_thread0 ALIGN(32) : {
11    thread_cpu0.o(.data)
12    thread_cpu0.o(.bss)
13    thread_cpu0.o(.rodata)
14 }
15 . = 0x80000 ;
16 _user_stack_start0 = .;
```

```
18 . = 0x1e0008;
19 .z ALIGN(4) : {
20    *(.z)
21 }
22
23 . = 0x80000 ;
24 .text_thread1 ALIGN(64) : {
25    thread_cpu1.o(.text)
26 }
27 .data_thread1 ALIGN(32) : {
28    thread_cpu1.o(.data)
29    thread_cpu1.o(.bss)
30    thread_cpu1.o(.rodata)
31 }
32 . = 0xa0000 ;
33 _user_stack_start1 = .;
```

Figure 3.7: Memory allocation of Figure 3.5 prior to scheduling

To the reader unfamiliar with linker scripts, they organize the code and data of the application into *named sections* which are then organized in memory (placed at specific

[7] As well as allocation rules respected by the real-time scheduling and allocation algorithms.

adresses, ordered one after the other, aligned). Directives are read from the beginning to the end. The first line sets the current memory address to 0x1e0000. At this address is allocated the section `.i` containing only the the memory location of C variable `i`. The directive in line 6 changes the address to 0x61000. At that address are allocated, in order, the code and local data of thread 0. Stack allocation does not require a section, only a base address. For thread 0, this is done in lines 15 and 16. Recall that stacks grow down, which explains why they are set at the end of the memory bank allocated to the thread.

The allocation of the dataflow variables and nodes is done during real-time scheduling. The corresponding linker script fragment for our example is provided in Figure 3.8.

```
g_ALLOC = 0x60280;
f_ALLOC = 0x80280;
h_ALLOC = 0x80a80;

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

. = g_ALLOC ;
.g_text ALIGN(64) : {
   g.o(.text)
}
.g_data ALIGN(32) : {
   g.o(.data)
   g.o(.bss)
   g.o(.rodata)
}
```

```
. = h_ALLOC ;
.h_text ALIGN(64) : {
   h.o(.text)
}
.h_data ALIGN(32) : {
   h.o(.data)
   h.o(.bss)
   h.o(.rodata)
}

y_ALLOC = 0x60a6c;
x_ALLOC = 0x80a6c;

/* DATA y */
. = y_ALLOC;
.y ALIGN(4) : {
   *(.y)
}
/* DATA x */
. = x_ALLOC;
.x ALIGN(4) : {
   *(.x)
}
```

Figure 3.8: Memory allocation of dataflow variables and nodes for the example in Figure 3.5

For the nodes, the linker script is very similar to the one of the threads: their code and data are placed next to one another and aligned on cache line size (following the requirements detailed in Section 4.1.2). For the variables, each one is allocated as a separate section, aligned on the size of a machine word. The addresses, chosen during real-time scheduling, are highlighted in red.

The allocation performed by the real-time scheduling algorithms gives a different address to each variable. This is a constraint due to industrial validation process that

requires that every variable can be observed at any moment during the execution. If this constraint is lifted, data memory use can be significantly reduced through allocation of multiple variables to the same location, provided their lifetimes do not overlap during execution. Preliminary results in this direction are provided in Chapter 6.

The full linker script of the application is obtained by concatenating the scripts of Figures 3.7 and 3.8.

3.3.3 The case for resource sharing

Resource sharing is not new in critical systems engineering [7, 80], but with the current drive towards increased functionality on increasingly parallel hardware its use cannot be avoided, even in the most critical systems. The side effect of resource sharing is interferences. When not properly controlled, interferences can reduce not only the efficiency, but also the predictability of a system, which in turn complicates safety demonstration, and thus 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 envelops. The best-known use of isolation is the full time-space isolation between partitions enforced by the ARINC 653 avionics standard [7]. However, less thorough isolation hypotheses are used in most implementation methods for critical parallel software proposed in recent years [81, 19, 35]. For instance:

- The isolation between memory regions accessed by individual tasks or cores, with explicit copy operations whenever one task/core needs information produced by another [19, 35]. This property is used to render code generation more portable [81] or, in conjunction memory allocation, to facilitate timing analysis [19, 35].

- The isolation between computation and communication phases. This can be enforced system-wide, as in the bulk synchronous parallel (BSP) model [79], 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 [35, 81] partially in [19], 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 [18, 35] 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. Many variants exist, from fully time-triggered, to mixes with event-driven.

- The total absence of resource access interferences between cores [34, 35] or applications [7] 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 imple-mentation process, but sometimes they are required at the specification level on partitioning units such as applications [7] or tasks [82]. 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 [83]. In multi-core contexts [84] keeping interferences at an "acceptable" level (and providing safe WCET estimations) often requires enforcing isolation properties. This explains why various platform standards (e.g. [7, 80]) and programming languages (e.g.[82]), as well as the parallel implementation methods cited above provide dedicated constructs to enforce isolation properties.

But there are cases where isolation is not needed. When a large, single-criticality flight control or engine control application is parallelized, isolation between its components is not required by regulation (only its predictability). Some implementation methods may impose it—as a form of over-engineering—to facilitate timing analysis, but this is not needed on platforms where the HW and SW architecture allow keeping interferences at an "acceptable" level by other means (*e.g.* statically scheduled bare metal code on timing predictable HW). In such cases, isolation needlessly curtails the implementation space, impacting efficiency.

We expect such cases to become more common. 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. Furthermore, common COTS multi-cores (*e.g.* ARM, POWER) have little support for isolation. Thus, ensuring the isolation properties 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 or partitions that require efficiency, safety and real-time guarantees, but not strong internal isolation.

The remainder of this section will use a simple example to intuitively show the effects of enforcing various isolation properties, comparing the result with the output of our method which does not enforce them. Then, in Section 6.2.4, we provide first experimental results showing 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.

The application we consider is the integration program of Figure 3.9. It is a single-period application and its period annotation is missing, meaning that parallelization is performed in optimization mode, aiming at minimizing the worst-case duration of one cycle of the synchronous program. The specification is formed of four dataflow nodes `f`, `g`, `h`, `n`.

```
node main() returns ()
var x,y,z,t:int;
let
  x      = f(7 fby y);
  (y,z)  = g(x);
  t      = h(9 fby z) ;
  ()     = n(x,t) ;
tel
```

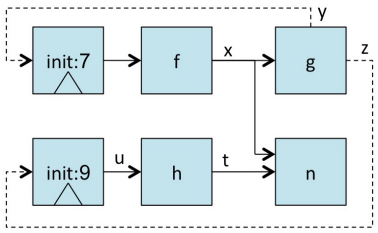


Figure 3.9: Simple integration program (left) and graphical representation (right)

This application is mapped onto a dual-core processor. Mapping is done using the method defined in the next chapters of this thesis, and the C implementation code (simplified to facilitate reading) is provided in Figure 3.10. We do not present here the linker script. To synthesize this code, our parallel mapping method relies on off-line real-time scheduling. We provide in Figure 3.11(left) the scheduling table produced by

```
int x,y,z,t ;

void thread_cpu0(){
   init_cpu0();
   for(;;){
      global_barrier_reinit(2);
      time-=PERIOD; time_wait(time);
      global_barrier_sync(0);
      dcache_inval();
      f(y,&x);
      dcache_flush();
      unlock(1);

      dcache_inval();
      g(x,&y,&z);
      dcache_flush();

}}
```

```
1  void init(){ y=7; z=9;}
2
3  void thread_cpu1(){
5     init_cpu1();
6     for(;;){
7
8
9        global_barrier_sync(1);
10       dcache_inval();
11       h(z,&t);
12       dcache_flush();
13
14       lock(1,1);
15       dcache_inval();
16       n(x,t);
17
18 }}
```

Figure 3.10: Dual-core implementation of the program in Figure 3.9

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`).

The implementation of Figure 3.10 lacks most of the isolation properties listed in the introduction:

Memory isolation. While the parallelization method allocates memory to reduce interferences, it does not attempt to privatize memory banks or isolate individual

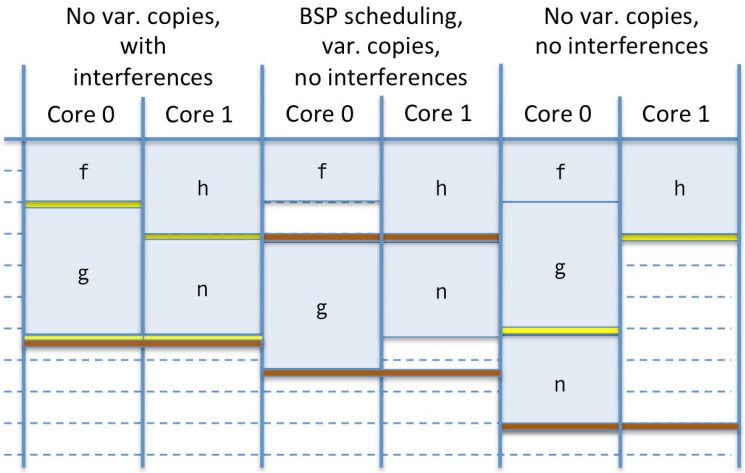


Figure 3.11: Three dual-core schedules for the example in Figure 3.9. In yellow, synchronization, coherency, and interference costs. In brown, global barriers (under BSP, they also perform coherency and communication and incur all interference, synchronization, and coherency costs).

tasks/nodes, applications, or cores. A first, very practical consequence is that, unlike in [19], 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 [81], which would result in 9 C-level variables) or a per-core basis (as in [19, 35], which would result in 6 C-level variables). In our case, there are 4 variables, directly corresponding to the 4 data-flow variables.

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 [35, 81, 19]. 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`).

To illustrate the effect of isolation, we have pictured in Figure 3.11 (middle and right) the optimal schedules under respectively:

- BSP isolation between computation and communication phases.
- 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.

Time isolation. While the scheduling table of Figure 3.11(left) is a time-triggered activation pattern like that of [18, 35], we chose to implement it using mostly event-driven mutex synchronizations, thereby losing the strong timing determinism. As we shall explain in more detail in Section 5.2.1, this does not lead to loss of functional correctness or to degraded respect of real-time requirements. On the contrary, as previously explained, the implementation is more robust to timing errors. For instance, if `f` takes some more time to execute than predicted, 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 MIF barriers, enforced with timers, which can become partition switch barriers if multiple applications are executed on the same multi-core.
- The absence at execution time of interferences not taken into account in the timing analysis performed during mapping. This property will be covered later, in Section 5.2.1, and is enforced through mutex synchronization.

Chapter 4

Timing model

The timing model is a core element of any method for real-time systems implementation. Its function is to provide *safe* and *precise* predictions on the timing behavior of systems satisfying a set of software and hardware architecture requirements. For a timing model to be practical, analysis should be *tractable* for systems in the chosen class.

In our case, the chosen class of systems is identified by the hardware and software architecture choices of the previous sections—statically scheduled, non-preemptive parallel implementations of dataflow specifications running on time-predictable hardware (one SMP compute cluster of the Kalray MPPA256 Bostan many-core) by relying on a specific API. These choices will be exploited throughout this chapter to ensure safety, precision, and tractability. While our implementation work concerns a very specific hardware architecture, the principles behind this timing model should be applicable to any timing predictable architecture.

Our timing model is derived from the one developed by H. Rihani at Verimag [19, 77]. By comparison, our model considers only one computing tile, but refines the analysis to consider both read and write memory accesses in the interference analysis. In Rihani's work, the choice of execution model[1] meant that no interferences can be due to read accesses from other cores.[2]

Another key property of our timimg model will become evident in the next chapter—it is possible to perform the timing analysis *incrementally during scheduling* with little impact on compilation time and while preserving precision (and thus efficiency). This allows the exploration of multiple mapping choices for each dataflow node under a safe timing model.

In this chapter, we show that we can compute a safe and tight upper-bound for the duration of one cycle of the loops running in lockstep *i.e.* for one cycle/period of the Heptagon (normalized) specification. We call this the *parallel WCET* of the implementation.[3] We do this in several steps:

[1] Where each processing core owns a memory bank, and where inter-core communications are performed by the producer writing into the consumer's memory bank.

[2] It also meant that the method cannot use all the cores of a compute cluster, as each core cannot own a bank due to the need to allocate basic software. Our method does not have this limitation.

[3] We do not use the term worst-case response time (WCRT), which hints towards the response to some input event. In our case, we measure the execution time of a piece of code with clearly identifiable start and

1. **WCET analysis.** We first derive a timing characterization for the sequential parts of the implementation (dataflow functions/tasks and code snippets forming the threads).

2. **Interference model.** We derive a model providing safe upper bounds on the interferences due to concurrent accesses to shared hardware resources.

3. **Parallel WCET analysis.** We finally provide the WCET analysis for parallel code, which is based on [20].

Sequential WCET analysis is performed using an external, state-of-the-art analysis tool.

4.1 Dataflow function analysis

The implementation of each dataflow function of the normalized integration specification is provided as either a (sequential) C function, or as Heptagon code compiled to a C function. These C functions satisfy well-formed properties defined in Section 2.1: there are no side-effects on global variables (only on output arguments), and execution time is statically bounded. To further simplify the tooling, we further assume that output dataflow variables are written only once by the C function (at the end of the function execution). This assumption allows us to simplify the function characterizations defined next.

The WCET analysis of such code is well-covered in previous work, and efficient tooling exists both in research [85] and the industry [86, 87], allowing analysis on the target platform. Such tools perform analysis on binary code, and their results strongly depend on low-level encoding and memory allocation choices done during compilation and linking. In our experiments we have used the state-of-the-art WCET analysis tool *aiT* from AbsInt [86, 87]. This choice was a requirement of the European project and we did not have time to test other tools such as OTAWA [85].

However, to allow integration of individual, sequential WCET results into the parallel WCET computation, the analysis of code obtained from the C functions must follow the methodology defined in this section.

4.1.1 Function characterization

Our timing model requires not only the execution time (in isolation) of each dataflow function, but also information about the number of memory accesses it makes and the number of other (library) functions it calls. Furthermore, during the static analysis phase providing this information we also compute other function-related information needed during mapping and code generation. The characterization of each function f therefore includes six different properties, the most important computed using *aiT*:

- An upper bound WCET(f) on the worst-case execution time of the function, in isolation (without external interferences). The resulting value must cover the

end points.

execution time of the function proper, not including the time needed to build the call context (putting arguments on the stack, calling the function, writing outputs at the end of 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. Dataflow communication variables are not accounted here. Since the convention is to have them as arguments, they are part of the stack (even the outputs, under the simplifying assumption).

- Upper bounds WCLC(f, l) the worst case number of calls to another function l. Of particular importance on the test platform is the software implementation of division, used by `gcc` whenever C code uses it. These functions are not inlined in the code of f and can potentially be accessed by several cores at the same time so we need to take them into account during interference analysis.

- An upper-bound WCSS(f) on the worst-case size of the stack during function execution. Given the properties of the dataflow function (in particular, the absence of recursion), stack is statically bounded and likely to be close to the average size need.

- The sizes CS(f) and DS(f) of the generated code and local data sections.

For this characterization (and especially the first 3 terms) to be compatible with the parallel timing analysis defined next and with the mapping method of the next chapter, static analysis by aiT must be performed under specific hypotheses concerning both the compilation of the code to be analyzed, and the configuration of aiT. These hypotheses are defined in the following two sections.

4.1.2 Compilation

The *aiT* tool works on executable binary code. If timing analysis is performed on an already complete implementation, this poses no problem, as the final code and allocation of the dataflow functions is known. However, our objective is different: We want to use the dataflow function characterizations in the synthesis of the parallel implementation. The final allocation is unknown at this point, and yet we need characterizations of dataflow functions that remain safe under implementation-time allocation.

For simplicity, we assume that analysis is performed separately for every dataflow function. To allow analysis, the function must be compiled and linked to produce a statically allocated executable and linkable format (ELF) file. If the function is provided under the form of hierarchic dataflow (Hepatgon) code, we assume that compilation inlines sub-nodes, so that a single C function is generated. This function may call low-level library functions, such as software division, but the use of library functions increases interferences, as library functions are shared between the cores. To follow the principles of dataflow synchronous design, these functions must either be stateless or used at a single place in the code. Their use must therefore be reduced to a minimum (*e.g.* through the use of compiler optimizations such as constant folding). We also

assume, as a simplifying assumption, that the code generated from the dataflow function is grouped into only 2 ELF sections, one for code (usually labelled `text`), and one for data.

To ensure that analysis results in isolation cover the final implementation, we must make sure that the generated assembly code is the same (*i.e.* same opcodes) in both cases. This is done through the use of the same optimization options, the use of annotations prohibiting inlining in the final implementation, and an encoding of data enforcing the use of the same adressing modes in both the analyzed program and the final implementation.

But ensuring that the (assembly) code is identical is not enough to ensure the preservation of WCET analysis guarantees. We also need to ensure that the caches would react in the same way (the same sequences of hits and misses) under the memory allocations of both implementation and analysis code. To this end, we make a number of allocation hypotheses that must be respected in both analysis code and the final implementation, by all functions. Under the cache configuration of the target platform, these hypotheses enforce cache partitioning properties that in turn ensure the preservation of WCET guarantees:

- The code of library functions is organized as a single section whose length is less than one cache way. Allocation is the same in the analysis binary and in the final implementation, and is aligned on instruction cache line size.
- If the function text section is smaller than the size of a cache way (4ko on the test platform), then its start is aligned on instruction cache line size. If it is larger, then its start is aligned on the size of a cache way.
- The location of the stack is fixed (the same in both analysis binary and implementation).
- The data of library functions is organized as a single section whose length is less than one cache way. Allocation is the same in the analysis binary and in the final implementation, and is aligned on data cache line size.
- If the stack and function data accesses can target the same cache lines, or if the function data section is greater than the size of a cache way, then its start is aligned on the size of a cache way. If not, it is aligned on the size of a data cache line.

These hypotheses can be greatly simplified if no library functions are needed. This will be the case in the experiments of Chapter 6. Other sets of hypotheses may ensure the same preservation result on the test platform or on platforms with different cache organizations (*e.g.* cache partitioning and/or prefetching can be used to simplify the mapping rules).achieve the same objective.

4.1.3 Static analysis

For each function f, once the analysis binary is built, multiple calls to aiT are needed to compute the for worst case execution time WCET(f), the worst case number of

accesses WCAT(f, r), where r ranges over the memory areas holding the stack, the function text and data sections, and the pre-allocated data, and the worse case number of library calls WCLC(f, l), for each library function l called by f.

To compute the number of accesses to a specific memory region, the configuration script of *aiT* must contain lines of the form:

```
area 0x120000 to 0x13ffff count accesses;
```

To ensure that the stack is set to the correct value, we allocate it explicitly to the correct value, *e.g.*:

```
routine "f" {
    enter with: reg("r12") = 0x160000;
}
```

Similar directives are used to allocate all reference-passed arguments of the function to addresses to a separate memory area (they are accounted for separately).

When f calls a library function l, we have two ways for obtaining valid WCET and memory access. One is to link the complete function code and perform analysis on the complete binary. The second is to provide *aiT* with information on this function:

```
routine "l" {
    not analyzed;
    takes: 153 cycles;
}
```

This configuration directive states that the library function `l` is not to be analyzed. Instead *aiT* assumes its code will take at most 153 cycles to execute.

4.2 Analysis of thread code fragments before synthesis

Dataflow functions are available before the mapping phase, so it is possible to compile and analyze them using *aiT* (or a similar tool, such as OTAWA). However, the top level of the implementation—the threads—is only synthesized once the mapping is performed and its code contains operations out of the scope of purely sequential code analysis, such as synchronizations. Therefore, we cannot use tools like aiT or OTAWA to perform the analysis of such code. To work around this problem, we derive ourselves upperbounds. Of the threads running in lockstep, described in Section 3.3.1, we only need to analyze the infinite loop body, which is a sequence of dataflow function calls and API primitive calls. For both dataflow function calls and API primitive calls we need to provide *prior to mapping* safe timing and memory access characterizations. To provide these characterizations, we perform a manual analysis, facilitated by the simplicity of the code.

Function calls. In the previous section we explained how to use aiT to derive the WCET of each dataflow function f called by the threads. But to apply the parallel WCET computation method of Section 4.4, we need WCET estimations covering the whole cost of each function call operation:

$$WCET(o) = WCET(f_o) + call\_WCET(f_o) + interf(o)$$

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

To allow the computation of $call\_WCET(f_o)$, the code that calls f_o is analyzed to produce a full non-functional characterization, like that produced using *aiT*. But *aiT* can only be applied on binary code, so it cannot be applied on the exact assembly code sequences for function call and return before mapping and code generation take place. For this reason, we rely on manual reasoning to derive the needed upper bounds for the target systems's application binary interface (ABI). The analysis starts from a description of the calling convention used by the C compiler.[4] The resulting characterization is a function on the number, type, and order of arguments of a dataflow function. This manual analysis is done at compiler construction time, for each ABI/C compiler pair.

In our example of Figure 3.5, calling the C function associated with `f` requires placing on the stack the value of variable i and the pointer to x. It requires read accesses to the thread code and to the location of i, and a single write access to the location of x, at the end of the function execution. For simplicity, we derive the characterization using very conservative assumptions. For instance, we consider that every memory read access results in a cache miss.

According to the Kalray call convention [88], the first arguments of a function are placed on 8 registers and therefore do not increase the stack. Remaining arguments are placed in order on the stack, aligned to their size. This means we must account in the computation for empty spaces in the stack caused by the succession of a word-long argument and a double word-long one. For example, assuming the top of the stack is aligned on the size of a double word, adding to it a word and a double word in this order will fill the stack as much as if we had two double words or two words and a double word.

Primitive calls We also need non-functional characterizations for the API primitives called by the threads. Given that the number of arguments is here fixed, the WCET of a primitive call operation can be computed as:

$$WCET(o) = WCET(p_o) + interf(o)$$

[4] Which in turn depend on the chosen C compiler and optimization choices, which must be fixed.

Here, the first term only depends on the primitive type $p_o \in \{\texttt{lock}, \texttt{unlock}, \texttt{inval}, \texttt{flush}\}$. To derive the characterization of $WCET(\texttt{lock})$, $WCET(\texttt{unlock})$, $WCET(\texttt{inval})$, and $WCET(\texttt{flush})$, we rely on a manual process (as for $call\_WCET(f_o)$). For $WCET(\texttt{lock})$, we assume that the primitive does not need to wait.

4.3 Memory access interferences

Consider a code fragment o (function or API primitive call, or code snippet of one of the threads). Once we have an upper bound on its worst-case duration in isolation, the only ingredient we need to compute its WCET in a parallel context is an upper bound on the interferences on o 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.

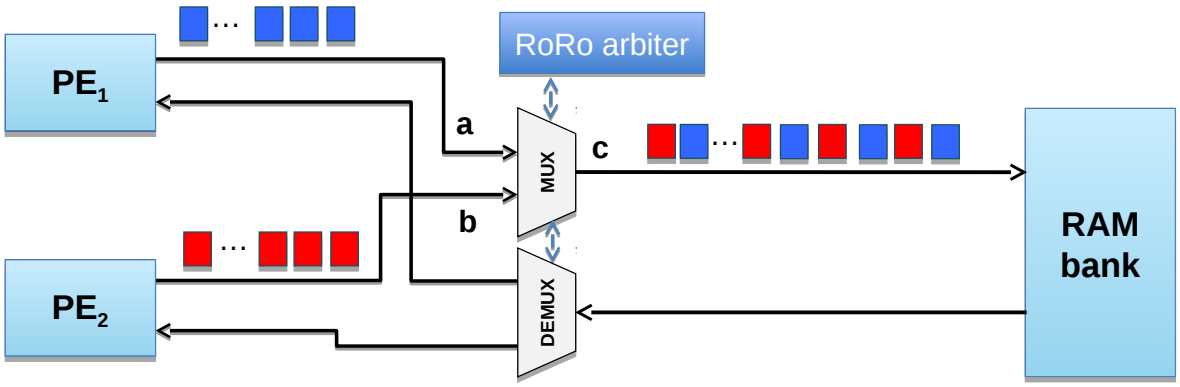


Figure 4.1: Simple memory multiplexer with round-robin arbiter

We illustrate how interferences due to concurrent memory accesses can happen using the simple interconnect in Figure 4.1. This one is much simpler than the interconnect of the Kalray test platform, but is subject to the same basic interference phenomenon. In Figure 4.1, two processing elements (PE) comprising a core and L1 caches are linked to a single memory bank through a multiplexer. Concurrent accesses are subject to fair (Round Robin) arbitration. Both cores issue requests to the memory (the blue and red boxes) to the memory. Assuming the same number of requests comes from both PEs, the worst-case interference scenario is the one of the figure, where each request of PE_2 is delayed by one request of PE_1 arriving at the same cycle but in a favorable arbiter configuration. Note that, as usual in memory arbitration, once a request is accepted, it cannot be interrupted.

Based on this worst-case interference scenario, we can derive a formula for computing a worst-case delay on computations of one PE due to memory accesses of the other. Assume the memory requests are generated by tasks t_1 and t_2 running in parallel on PE_1 and PE_2 respectively. Assume task t_i issues a_i requests to the memory bank, $i = 1, 2$. Then, an upper-bound on the worst-case number of cycles that can be added to the duration of task t_1 by memory interferences from task t_2 is then $min(a_1, a_2) \times d$, where

d is the worst-case duration of a request by the memory.

This formula is correct, but not yet precise enough, because it does not account for the fact that on most architectures (including Kalray), all memory requests do not take the same time. On Kalray, a read request fetches an entire line of cache and keeps the memory interface busy for eight cycles (a cost denoted with rd), whereas a write request concerns only one word and is done in a single clock cycle (a cost denoted with wd). We also consider at this point the fact that there are not one, but sixteen memory banks. To take this into account, we denote with $a_t(B)$ the number of (all) memory accesses of a task t to a memory bank B, with $r_t(B)$ the number of read accesses, and with $w_t(B)$ the number of write accesses. With these new notations, we can refine the previous formula. On our test platform, an upper-bound on the number of cycles that can be added to the duration of a task t_1 due to memory interferences on bank B from task t_2 is:

$$Interf(t_1, t_2, B) = rd \times min(a_1(B), r_2(B)) + wd \times min(a_1(B) - min(a_1(B), r_2(B)), w_2(B))$$

This formula is no longer symmetrical. It is composed of two terms, corresponding to interferences from a different type of request (read or write). As read requests are the most expensive, they are considered with priority in order to produce a worst-case bound. We implicitly assume that we have no information on the distribution of requests inside the execution of t_1 or t_2. For instance, in the first term, we can have at most $min(a_1(B), r_2(B))$ interferences from read requests issued by t_2. For the second term, t_1 requests where read interferences were already accounted for are not considered. Note that we do not need to consider the type of requests issued by t_1.

This formula can be extended to cover all interferences on a task t by cumulating the interferences on all memory banks from all tasks t' whose execution can overlap with that of t:

$$Interf(t) = \sum_{t' \| t} \sum_{i=0}^{15} Interf(t, t', B_i)$$

Cumulating is done under worst-case assumptions, by simply adding them. Note the notation used to identify tasks whose execution can overlap with that of t. We call these *concurrent* tasks of t.

This formula extends previous work on timing analysis of parallel applications [19] by considering the different contributions of read and write accesses to the interference budget.

4.4 Parallel WCET computation

The structure of the generated code, exemplified in Figures 3.5, 3.7 and 3.8, has been chosen to allow the computation of tight bounds on the execution time of one cycle of the `for` loops running in lockstep. Recall that each iteration of these top-level `for` loops implements one hyper-period of the integration program. Each loop body is formed of the global barrier code (in the yellow box in Figure 3.5) 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 [5] implementing the dataflow function. Cache operations placed before and after the function call ensure memory coherency. Lock operations are placed at the beginning and at the end of the snippet. They are always paired, to enforce order relations between specific points of different threads (the red arrows of Figure 1.2). One `unlock` operation marks the dependency start and one `lock` operation on the same lock marks the dependency end. The code generation process ensures that no other lock operation can access the same lock between the points in time where the `unlock` and `lock` calls corresponding to the dependency are executed.

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

The scheduling and code generation process ensure that this graph is acyclic. Figure 4.2 presents the DAG associated with the example in Figure 3.5.

Property 1 (Parallel WCET computation)**.** *Assume that for each operation o of this DAG we can compute an upper bound on its duration WCET(o) (for release operations we consider instead the value of the release annotation), and that* `B` *is an upper bound on the duration of the barrier synchronization (including the call to* `time_wait`*). Then, according to [20], an upper bound on the duration of an iteration of the loops running in lockstep is obtained by adding* `B` *to the critical path of the DAG.*

From the WCET analysis perspective of [20], this DAG is the CFG of the application after adding the communication arcs, assuming that each operation has its own basic block. The fundamental difference w.r.t. [20] is that we cannot build this CFG starting from existing parallel code, because this analysis must be performed *incrementally during* parallelization.

As previous sections explain how WCET(o) can be computed for all o, the definition of the timing model is complete.

4.5 Platform description format

Recall from the introduction (Figure 1.3) that our parallelization method involves multiple tools. Exchange of information between these tools must be formalized. We have

[5] 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.

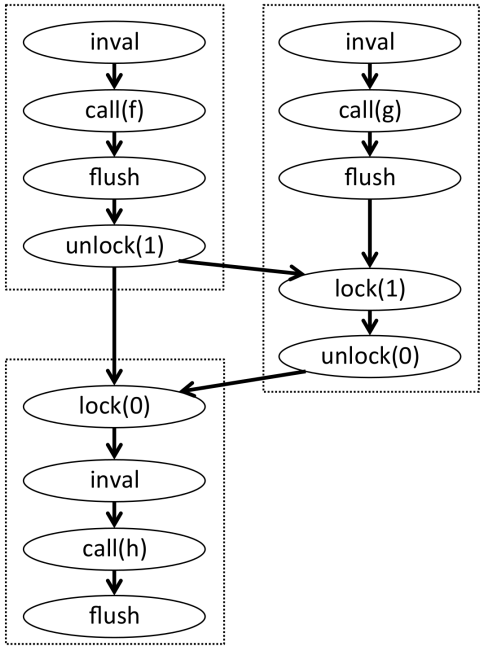


Figure 4.2: The DAG of operations and dependencies for the parallel code in Figure 3.5. Snippets are represented with dotted boxes grouping the operations.

presented in previous chapters the Lustre/Heptagon specification (and its normalized version) and the C and linker script output of the method. One last piece of information in this figure requires formalization—the platform description.

What Figure 1.3 labels as platform description is structured into two separate files:

- The hardware description file, used to define and configure the hardware.
- The analysis result, containing the non-functional characterization computed through static analysis for every dataflow and library function.

The analysis file is an intermediate artefact, obtained by applying the method of Section 4.1. The hardware description file is a primary input of our mapping method. Along with the integration specification and with the code of the dataflow nodes/functions, it fully defines a mapping and code generation problem.

The hardware description file and the analysis file of the example in Figure 3.5 are provided in Figure 4.3. The hardware description specifies the number of cores the implementation can use, the number and size of memory banks, and memory ranges that are reserved and cannot be used during memory allocation. Our example states that parallelization is performed on 2 cores, and memory configuration corresponds to the standard Kalray MPPA256 Bostan cluster—16 banks of 128kBytes each, of which one range is reserved for basic software and one is reserved for ROM (as explained in Section 3.3.2).

The analysis file is application-dependent. It contains the characterization of each dataflow function (no library function is used). For each function, it provides the size of

```
Architecture

Cores:2

Memory Bank Number : 16
Bank Size : 0x20000
Memory Excluded
[Start:0x000000 End:0x060000]
[Start:0x1ff000 End:0x200000]
```

```
Function f :
Text : 200 Data : 16 Stack : 16
WCET : 454
WCAT :
  Text :  [ 4   0   0 ]
  Data :  [ 0   0   2 ]
  Stack : [ 0   4   4 ]

Function g :
Text : 140 Data : 8 Stack : 4
WCET : 368
WCAT :
  Text :  [ 3   0   0 ]
  Data :  [ 0   0   2 ]
  Stack : [ 0   1   1 ]

Function h :
Text : 180 Data : 4 Stack : 8
WCET : 406
WCAT :
  Text :  [ 4   0   0 ]
  Data :  [ 0   0   1 ]
  Stack : [ 0   2   1 ]
```

Figure 4.3: Architecture description (left) and function characterization (right) files for the example in Figure 3.5

its text section, data section, worst-case stack length, WCET in isolation as computed in Section 4.1, and WCAT for each memory region of interest. Each of the WCAT figures differentiates between three types of accesses: code reads, data reads, and data writes.

The characterization of the API primitives and the interference model should ideally be included in the architecture description file, but at this point they are not. More work is needed to allow the definition of a language allowing the definition of these models for multiple hardware platforms and configurations.

Chapter 5

Mapping and code generation

5.1 Real-time systems compilation

In previous chapters we have discussed the functional and non-functional specification of the systems under development, as well as the timing analysis of parallel software. In this chapter, we consider the automatic mapping and code generation allowing the transformation of a full-fledged system specification into parallel implementation code that is both functionally correct and satisfies the non-functional requirements.

In solving this problem, we are subject to the fundamental difficulty of real-time scheduling: mapping and code generation depend on the timing characterization of system components, whereas the timing characterization itself depends on mapping and code generation. This difficulty is particularly acute in the case of safety-critical control systems where certification regulations require a strict respect of real-time requirements. In such cases, an exhaustive search for the optimal solution is not possible, even for the simplest systems, in the absence of simplifying hypotheses.

In solving the mapping and code generation problem, given the complexity of the applications and of the execution platforms we consider, we make the radical choice of always seeking scalability, possibly at the expense of optimality.[1] To this end, we avoid methods that are often used in real-time scheduling, but have high theoretical and empirical complexity:

- Backtracking-based methods, often used in conjunction with contraint programming under one of its forms (integer linear programming, satisfiability modulo theories, *etc.*).
- Fixpoint iteration, to solve the cyclic dependency between timing analysis and mapping, when it is not accompanied by a guarantee of rapid convergence.

Given these choices, our main inspiration in designing mapping algorithms came not from real-time scheduling, but from compiler design. In the proposed implementation flow of Figure 1.3, page 18, mapping and code generation is performed by the *parallel back-end*. This back-end has the same general functions as the back-end of a compiler

[1] Note that classical optimality results in real-time scheduling hold under simplifying hypotheses, meaning that the overall method is not optimal.

for an imperative language. *Starting from the intermediate representation – in our case the normalized platform-independent program of Section 2.3 – it performs memory allocation and scheduling and then generates correct and efficient target-specific code in a scalable way.*

This similarity of objective has lead in our case to significant similarities in the implementation:

- Like a compiler back-end for a sequential imperative language [89, 90], our back-end uses a *static scheduling heuristic based on list scheduling*, presented in Section 5.4. The use of low-complexity static scheduling heuristics is meant to guarantee scalability.

- To represent the output of the mapping phase prior to code generation, we rely on *scheduling tables*, also called *reservation tables* or *timetables* in some contexts. This is similar to classical work on optimized compilation for VLIW and superscalar architectures [38, 91].

Due to these strong similarities, we view our mapping process as a system-level compilation process subject to non-functional, and in particular real-time requirements. We dub this *real-time systems compilation.*

Schedulability vs. optimization While major similarities exist between real-time systems compilation and classical compilation, there are also significant differences. The most obvious is that the output of our parallel back-end is C code (plus `gcc` annotations and linker scripts), whereas classical compilation starts at C level.

But beyond form, a more fundamental difference exists: In real-time systems compilation, the main performance-related goal is not to optimize some metric such as speed (throughput), memory footprint, or energy consumption. Instead, it is to produce an implementation that is functionally correct and which respects the non-functional requirements [1].

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. This amounts to a strong integration of all the transformations of Figure 1.3 (mapping/parallelization, code generation, C compilation of both tasks and threads, timing analysis of sequential code) around the timing model of Chapter 4.

By comparison, classical optimizing compilers provide no worst-case timing guarantees. Instead, their objective is to synthesize code with good *average-case* performance on chosen application types. To this end, they perform an average-case, precise-enough, and unsafe accounting of time and possibly other non-functional properties,

such as power consumption. In turn, this requires comparatively less integration between scheduling, timing analysis, and code generation.

5.2 Reservation tables

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 presented in Figure 5.1.

```
type interval = { starti : int ; endi : int } (* time/RAM interval *)

type reservation_table = {
  length : int ;                       (* size of scheduling table *)
  inst_cpu : cpuid instance_map ;      (* allocation of snippets to CPUs *)
  inst_time : interval instance_map ; (* time reservations of snippets *)
  fun_ram : interval fun_map ;         (* RAM reservations of functions *)
  var_ram : interval var_map ;         (* RAM reservation of variables *)
}

type valid_scheduling_state = {
  ns : instance_set ;                    (* set of yet unmapped dataflow functions *)
  free_ram : free_ram ;                  (* yet unallocated RAM *)
  free_cpu_time : free_cpu_time ;        (* yet unallocated CPU time *)
  inst_current_wcet : int instance_map ; (* current WCET of allocated snippets *)
  inst_wcat : wcat instance_map ;        (* computed WCAT of allocated snippets *)
  rt : reservation_table ;               (* reservation table *)
}

type scheduling_state = NonSchedulable | Schedulable of valid_scheduling_state
```

Figure 5.1: Data structures used for scheduling, in OCaML syntax

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 corresponds 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. Synchronization resources will be allocated during code generation, but enough CPU time and memory must be reserved for them during the mapping phase.

CPU time is allocated to the instances of dataflow functions of the normalized integration program. The reservations made for one function instance must cover the needs of the C code snippet generated for it, as explained in Section 4.4. This snippet consists of one dataflow function call and possibly a number of API cache coherency and synchronization primitive calls.

Allocation of CPU time to snippets is done using the `inst_cpu` and `inst_time` fields of the `reservation_table` structure. The two fields are maps (partial functions) from

function instances to core identifiers and time intervals respectively. When scheduling succeeds, they associate to each instance, *i.e.* to each snippet, exactly one core and one time interval $[s, e]$ with $0 \leq s < e \leq l$, where l is the length of the reservation table.

Memory is allocated to dataflow functions (not to their instances) and to dataflow variables. If scheduling suceeds, each dataflow function `f` is allocated a single memory interval, used to store all its code and local data (the two ELF sections generated by the compilation process of Section 4.1.2). Each dataflow variable is allocated one memory interval.

Note that memory reservations do not have a lifetime. This is natural for embedded code, but is less so for data variables. Implicitly, the lifetime of a variable is the whole period, and no lifetime-based optimization can be used in our systems. This is due to the *observability* non-functional requirement—the value of a dataflow variable must be available to inspection at any point of execution.

A explained in Section 3.3.2, 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.

5.2.1 Safe abstraction issues.

Reservation tables provide a time-triggered execution pattern, whereas the implementations we synthesize are mostly event-driven. The execution synchronizes with real-time only at the level of the global barrier and at the level of dataflow function instances where the respect of the release date is not enforced by event-driven synchronization and sequential control passing. These instances are those with a release date strictly greater than the release date of all dataflow functions on which it depends through control or data (event-driven) dependencies.

We must therefore define the abstraction properties relating scheduling table and implementation code. We require a scheduling table and the corresponding multi-threaded implementation to satisfy the following well-formedness properties:

Sequential resources Dataflow functions allocated on the same CPU must have non-overlaping time reservations, allowing their implementation as a sequential thread.[2] Note that memory reservations are not subject to sequential resource requirements, as they have no lifetime and isolation is based on spatial arguments alone.

Causal correctness If a dataflow function instance f uses a value produced by another instance g, then:

- **CC1** The time reservation for g ends before the reservation for f starts.
- **CC2** The execution of the function call associated with g ends before the execution of of the function call associated with f starts. Furthermore, if f and g are executed respectively on cores c_f and c_g, with $c_f \neq c_g$, then core c_g

[2] For general Heptagon programs, this must be understood modulo use of conditional execution [92, 27]. However, the conditional execution aspect is already covered in previous literature, and orthogonal to the interference-related aspects we consider in this thesis. As the industrial use cases do not require conditional execution, either, we do not formally cover it in this thesis.

performs a cache flush operation after g, and c_f performs a cache invalidation after the cache flush of c_g and before the execution of f. Ordering can be ensured through either event-driven synchronization, or through timing arguments.

Respect of requirements If a dataflow function instance f has release date r and deadline d, if its time reservation starts at date s and ends at date e, and if its execution starts (in one cycle) at date xs and ends at date xe, then:

RR1 $r \leq s < e \leq d$

RR2 $r \leq xs < xe \leq d$.

Reservation size The time reservation of a function instance must be longer than the sum of the WCET of the snippet associated with the instance, as defined in Section 4.4 (including coherency, synchronization, and interference over-heads).

Preservation of interferences When two snippets access at least one common memory bank and have non-overlapping time reservations, their executions must always be sequenced, by means of either event-driven synchronization, or through timing arguments. In other words, no interferences can occur at execution time that were not considered in the interference analysis at scheduling table level.

Note that some of the properties were already covered in previous literature [92, 34, 36]. Such are (sequential resources), (CC1), (RR1).

Also note that, through these properties, determining the schedulability of the implementation can be decomposed in three problems:

SCH1 Checking that the scheduling table satisfies the well-formed properties (sequential resources), (CC1), and (RR1). Previous work on off-line real-time scheduling [91, 23, 14] already covers (extensively) these aspects.

SCH2 Ensuring that the synchronization protocol enforces the respect of the ordering properties (CC2), (RR2), and (preservation of interferences).

SCH3 Checking that the reservation size satisfies property (reservation size).

Significant previous work covers (SCH1). Ensuring (SCH2) amounts to ensuring the preservation of a DAG structure at execution time by means of mutex operations (it is always possible). Ensuring (SCH3) is technically the most difficult, as it encapsulates all the quantitative aspects of the interface between the scheduling table model and the timing analysis of C code.

5.3 Incremental timing analysis

Recall that the reservation table of the full application is built incrementally, using a list scheduling heuristic applied off-line. The function instances of the integration program are considered one by one, in an order compatible with dataflow dependencies.

When a function instance is considered, CPU time is allocated to its snippet, and all yet unallocated code and data it uses—function code and data, dataflow variables—is mapped to memory. Once the mapping choices are made for an instance, function, or variable, they are never changed (there is no backtracking).

The main difficulty in the definition of our mapping method is allowing the *incremental* proof of timing correctness of the implementation. This amounts to ensuring that (SCH3) is true in an incremental fashion that follows step by step the advance of the list scheduling heuristic. The reservation size of a node must accomodate not only the WCET of the dataflow function call, but also memory coherency costs, synchronization costs, and interferences. In particular, it must include a budget meant to cover all interferences by, and all synchronizations with function instances that were not yet mapped.

5.3.1 Memory coherency protocol

On the chosen execution platform, it is easy to bound costs related to memory coherency: invalidating the entire data cache before execution of the function and flushing all changes to main memory after its execution is possible at a low, fixed cost. For this reason, we perform both operations systematically, one before and one after each function call.

5.3.2 Synchronization protocol

As explained in Section 4.4, the synchronization protocol enforces a DAG structure (and possibly some timing constraints) over the snippets of an execution cycle of the parallel implementation. Inside each cycle, synchronization is performed point-to-point, using paired `unlock` and `lock` primitives. As threads are fully sequential, we can always assume, without loss of generality, that a snippet can be source of at most $N - 1$ dependencies, where N is the number of threads/cores.[3] The same property holds for destinations.

The cost of synchronizations on the Kalray platform is low in terms of CPU cycles. A lock takes at most 10 cycles and an unlock 25 cycles. However, systematically reserving a large number of outgoing synchronizations to account for all possible allocations of dependent nodes is not acceptable in practice. It leads to bloated code, significant runtime penalty and, most important, requires large numbers of hardware locks to be used at the same time. On our largest use case, provided in Section 6.1.1, such an approach was impossible to implement, as it required more than 127 hardware locks (the platform limit), even after the use of various methods (detailed in Section 5.5.1) that drastically reduce the number of needed locks through sharing and through lifetime reduction.

To avoid these problems, we have introduced a new method for synthesizing the synchronization protocol. It potentially introduces undeeded synchronizations, but can be implemented in a low time and code size envelope, with a small, fixed number of

[3] If more dependencies exist, they can be optimized by retaining, for each processor p different from the one executing the snippet, only the first dependency towards p.

resources depending only on the number of cores/threads. Most important, our method provides good practical results on the use cases.

Synchronization synthesis is performed as follows: To each snippet, we associate two synchronization points: one at the beginning (before the cache invalidation operation preceding the function call), and one at the end (after the cache flush operation following the function call). All the synchronization points of all the snippets are fully sequenced using mutex operations, in a way that enforces property (SCH2). In the C code, each synchronization point is translated into two mutex operations – one `lock` waiting for the completion of the previous synchronization point (if any, and if not on the same thread/core), and one `unlock` to give control to the next point (if any, and if not on the same thread/core). The code of Figure 3.5 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 remaining three synchronization points are ordered by two synchronizations: one from the end of `f` to the end of `g` and then another from the end of `g` to the beginning of `h`.

Under this code generation method, the total synchronization overhead of one snippet is up-bounded by 2 × (WCET(`lock`) + WCET(`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 WCET(`lock`) + WCET(`unlock`)).

5.3.3 Interferences

When performing the mapping of a function instance, 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 later in the scheduling process. Yet, without backtracking, we must also provide a budget for these interferences.

We do this by provisioning 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 provided as an input to the scheduling algorithms (the `provision` input in Figure 5.2). We do not define automatically a value for this percentage. For now, the efficiency of the back-end allows us to try different values manually until we find an optimum.

Through the `inst_current_wcet` field of the `scheduling_state` data structure of Figure 5.1, the scheduling routine maintains at all times a safe upper bound on the execution time of all function instances that were already scheduled. These figures include memory access interferences from already scheduled function instances. Whenever a new function instance *fi* is mapped, `inst_current_wcet`(fi) is first computed, and `inst_current_wcet`(gi) is updated to include interferences from operations of *fi* (if any) for all function instance *gi* that has been already scheduled. It is required that, at all times during scheduling, this value stays smaller than the reserved CPU time, for all scheduled instances. Mapping choices not respecting this requirement must be rejected.

5.3.4 Reservation size

Under the design choices of Sections 5.3.1-3, the minimal length of the time reservation made for a snippet associated with an instance of function f is:

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

This formula and the interference model of Section 4.3 are directly employed in the scheduling algorithms of the next section. They are safe under the code generation assumptions detailed above.

5.4 Scheduling algorithm

The scheduling algorithm is structured as a classical list scheduling heuristic. With the notations of the previous sections, two high-level routines are presented:

- The list scheduling driver that makes most mapping decisions.
- The routine that updates the scheduling state based on the decisions of the scheduling driver, and determines if the new state satisfies property (SCH3). Thus, it can be seen as a schedulability test.

5.4.1 Incremental resource allocation

The use of list scheduling in real-time and embedded systems is by no means new [93, 94, 23, 34]. Its main advantage is low complexity—each dataflow node or task is considered only once, which largely reduces the search space, leading to a scalable implementation process. Its main disadvantage is the lack of optimality. However, as both previous work and our results show, fine tuning the lower levels of the heuristics can result in very efficient implementations, close to theoretical limits.

The top-level list scheduling driver is presented in Figure 5.2. It consists in an initialization phase, followed by a 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 dependencies have all been scheduled. Among them, the choice of what instance to schedule next is done with an earliest-deadline-first policy (EDF). For function instances with the same deadline, we choose the one with least laxity. For the example in Figure 1.1, the instance of `f` is chosen first due to its deadline (the choice being between `f` and `g`). Then, `g` is scheduled (no choice exists, because `h` depends on `g`), and finally `h`.

The subroutines `needed_align` and `needed_ram` compute the necessary information for the memory allocation of the objects related to the currently scheduled instance. The alignment is the one of Section 4.1.2, the same that was used for the timing analysis of the function. The RAM reservation size must accomodate the code and local data of the function, as well as yet unallocated dataflow variables (inputs and outputs) of

```
1  procedure ListSchedulingDriver
2  inputs: d:dataflow; /* integration program */
3          provision:float; /* interference provisions (% of function WCET) */
4          cpus:int; /* number of CPUs to use on the architecture (≤ 16) */
5  outputs: s:scheduling_state
6  begin
7    vs := build_init_scheduling_state(d,cpus) /* set vs.st to be a scheduling table of
8       length equal to the period of d and no reservations, vs.ns to contain all
9       function instances of d, vs.free_cpu_time to contain all CPU time,
10      and vs.free_ram to contain all free memory (in blue in Figure 3.6) */
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 */
17     nr := needed_ram(d,vs,fi) /* size and organization of RAM to allocate */
18     res := reservation(get_function(fi,d),provision) /* time reservation size */
19     /* earliest start date depends on release date and predecessors' end */
20     sd := max(d.release[fi],max(vs.rt.fun_time[gi].endi | gi precedes fi in d))
21     /* find the first date where the function can be scheduled (if any) */
22     found := false ;
23     while ((sd+res≤d.deadline[fi]) and not found) do
24       /* attempt allocation on all CPUs and retain all valid scheduling results */
25       sres := ∅
26       /* Reservation start date must allow the scheduling of a synchronization
27          point exclusive in time with all others synchronization points */
28       sd := next_sync_free(vs,sd) /* returns -1 upon failure */
29       if sd >= 0  then
30         for cpu := 0 to cpus-1 do
31           sprime := ScheduleBlockAtDateOnCPU(d,vs,fi,cpu,sd,res,na,nr)
32           if sprime ≠ NonSchedulable then sres := sres ∪ {sprime} end
33         done
34       end
35       if sres = ∅ then
36         /* allocation not possible at date sd, advance date */
37         sd := advance_time(sd,d,fi,vs)
38       else
39         s := choose_optimal(sres,fi,cost_function)
40         found := true
41       end
42     done
43     if not found then return NonSchedulable end /* scheduling of f was not possible */
44   done
45   return s /* application scheduling was successful */
46 end procedure
```

Figure 5.2: List scheduling driver pseudo-code

the function. While the routine determines the needed size and alignment, it does not

choose the exact address, which is performed later.

Scheduling is performed at the earliest date possible after the release date and the end of all predecessors of the dataflow function instance. Starting at this date, scheduling will be attempted on every core (by the `for` loop in lines 30–33). If scheduling at a specific date fails on all cores, time is advanced (line 37) and scheduling attempted again until a solution is found or scheduling is no longer possible given the duration and deadline of the function instance (line 23). If for a given start date multiple cores allow scheduling, one mapping is chosen using a cost function (line 39). For the application of Figure 1.1, the allocation and scheduling of `f` is possible on both core 0 and core 1, at date 0. The two possible schedules having the same real-time properties (and thus the same cost), allocation on core 0 is retained. When mapping `g`, the earliest start date determined by release date requirement and predecessor dependencies is computed (in line 20) as `sd` $= 0$. However, starting `g` at date 0 would mean that the start synchronization points of `f` and `g` overlap in time, which is not permitted according to Section 5.3. The call to `next_sync_free` (in line 28) finds the first date after `sd` allowing the mapping of a synchronization point that is non-overlapping with synchronization points of previously mapped functions. Recall that the length of a synchronization point is $\delta = \text{WCET}(\texttt{lock}) + \text{WCET}(\texttt{unlock})$. Therefore, the mapping of `g` is first attempted at date δ on all processors, through calls to function `ScheduleBlockAtDateOnCPU`.

As mentioned above, when mapping is possible at a given date on several cores, the choice of mapping is performed using a cost function (in line 39). The cost function used in our experiments favors mappings that reduce the end date of the instance.

When all instances have been scheduled, the routine returns the final scheduling state that contains all mapping choices needed for the code generation. At this point, synchronization has not yet been synthesized, even though time has been reserved for synchronization points. Once the scheduling phase is completed, code generation is always possible under the previous assumptions.

5.4.2 Implementation of `fby` equations

The list scheduling algorithm only traverses and maps dataflow function instances. However, our normalized specifications also include `fby` operations.

As explained in Chapter 2, the `fby` construct allows one value to pass from one cycle to the next. Traditional code generation for `fby` equations [52, 23] uses separate variables for the input and output variables. An explicit copy operation is performed at the end of each cycle to update the output variable with the current value of the input variable. In some cases, this implementation is mandatory, as the two variables must be live at the same time, like in the following fragment, where both `x` and `y` are used by `g`:

```
x  = f() ;
y  = 0 fby x ;
() = g(x,y) ;
```

However, while sometimes necessary, this encoding is in most cases unneeded, because the application *scheduling* can ensure that the input and output variables of the `fby`

are never live at the same time. In such cases, only one C variable is needed to encode both Heptagon variables and no copy operation is needed, resulting in a reduction in code size, data size, and duration. Such is the case in the following example:

```
x  = f() ;
y  = 0 fby x ;
() = g(y) ;
```

Here, dataflow variables `x` and `y` can be implemented by a single C variable provided that `g` is scheduled before `f`.

Beyond these small code fragments, all `fby` equations of our industrial case studies have this property, meaning that the optimization potential is considerable. Based on this observation, we decided to make the simplified encoding of `fby` the standard in our code generation scheme. This was particularly beneficial for variables encoding exposed states in normalized integration specifications.

For a given `fby` construct, simplified code generation is possible if all dataflow functions that consume the output of the `fby` can be scheduled before the node producing its input. This constraint is enforced through explicit control dependencies added to the graph provided as input to the scheduling algorithm. The compilaton of Heptagon programs with allocation annotations enforces similar scheduling constraints [50].

While certain programs cannot be implemented under this simplified code generation, all correct programs can be transformed to allow compilation under the simplified rules, through the introduction of explicit copy operators. For instance, our first example of code requiring the two-variable `fby` encoding can be transformed into:

```
x  = f() ;
x1 = x ;
y  = 0 fby x1 ;
() = g(x,y) ;
```

This transformation amounts to dissociating the copy function of the `fby` from that of a semantic indicator of the point where one value crosses the barrier between successive cycles, but without role in code generation. This transformation can be performed automatically at scheduling time, by introducing *on-the-fly* the copy equations and the extra variables, and thus removing the scheduling dependencies when scheduling cannot advance otherwise.

5.4.3 Reservation and schedulability test

The list scheduling driver routine of Figure 5.2 chooses the thread/core on which a dataflow function is mapped, chooses a start date for its code snippet on the chosen processor, and determines the memory allocation needs. The remaining mapping decisions, the schedulability check of property (SCH3) and the update of the scheduling state data structure are performed by routine `ScheduleBlockAtDateOnCPU`, called by the scheduling driver (in line 31) and presented in Figure 5.3.

```
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 has free time interval starting at date start, of length at least
8        time_len and allowing the scheduling of the end synchronization point */
9    f_res := find_free_interval(vs,cpu,start,time_len)
10   if invalid_interval(f_res) then return NonSchedulable end
11   vs.rt.inst_cpu[fi] := cpu;
12   vs.rt.inst_time[fi] := f_res;
13   vs := update_free_time(vs,cpu,f_res);
14   /* Reserve a large-enough, well-aligned free interval (if possible). */
15   vs := reserve_mem(vs,cpu,fi,mem_align,mem_res)
16   if invalid_state(vs) then return NonSchedulable end
17   /* Worst-case accesses of snippet fi to various banks under given allocation
18      (upper bounds over r_fi(b) and w_fi(b), for all b, cf. Section 4.3) */
19   vs.inst_wcat[fi] := wcat(d,vs,mem_res,fi)
20   f := get_function(fi,d) /* get the function of the instance */
21   vs.inst_current_wcet[fi] :=
22     WCET(f)+call_WCET(f)+2*(WCET(lock)+WCET(unlock))+WCET(inval)+WCET(flush)
23   forall gi function instance already scheduled do
24     if intervals_overlap(vs.rt.inst_time[gi],f_res) then
25       vs.inst_current_wcet[gi] :=
26         vs.inst_current_wcet[gi] + interf(vs.inst_wcat[gi],vs.inst_wcat[fi])
27       /* if provisions on gi are not sufficient due to fi */
28       if vs.inst_current_wcet[gi] > length(vs.inst_time[gi]) then
29         return NonSchedulable
30       end
31       vs.inst_current_wcet[fi] :=
32         vs.inst_current_wcet[fi] + interf(vs.inst_wcat[fi],vs.inst_wcat[gi])
33       /* if provisions on fi are not sufficient due to gi */
34       if vs.inst_current_wcet[fi] > length(vs.inst_time[fi]) then
35         return NonSchedulable
36       end
37     end
38   done
39   /* allocation and scheduling are possible */
40   return (Schedulable vs)
41 end procedure
```

Figure 5.3: The routine that checks schedulability and updates the scheduling state

The first mapping decision this routine makes concerns the reservation end date (in line 9). The resulting reservation interval must not overlap with pre-existing reservations on the same processor, be longer than `time_len`, and its end (for a length of δ) must not overlap with synchronization points of previously mapped functions (on any processor). Function `find_free_interval` always chooses the smallest end date with this property.

Ensuring that synchronization points do not overlap means that time reservations are often longer than $reservation(f, provisions)$, to ensure that the end synchronization point does not overlap with others. If in the example of Figure 1.1 we have $reservation(g, provisions) = reservation(f, provisions)$ with f is already scheduled, then the reservation made for g will end at date $reservation(f, provisions) + \delta$, so that its reservation size ends on an interval free of synchronization points.

If a suitable end date cannot be found, the function returns `NonSchedulable`. If a date is found, the scheduling state is updated with the new processor time reservation (in lines 11 through 13).

Memory allocation is performed (and the scheduling state updated) in line 15. For each function instance we reserve one memory interval, possibly of size 0,[4] 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 function instance. Allocation inside this interval, as well as its alignment, is fixed by the calls to `needed_ram` and `needed_align` in lines 16 and 17 of Figure 5.2. The code and data of the function always come first, according to the rules of Section 4.1.3. If memory allocation cannot be performed, the function returns `NonSchedulable`.

The remainder of the routine performs schedulability analysis and updates scheduling state fields that are only used to speed up this analysis (`inst_current_wcet` and `inst_wcat`). Along the execution, field `inst_current_wcet` maintains for each function instance that has already been mapped an over-estimation of its WCET, including the worst-case interferences from other already-mapped function instances, according to the timing model of Chapter 4. When `ScheduleBlockAtDateOnCPU` is called on a function instance fi, the code in lines 19-38 sets up the initial WCET estimate for fi and updates the estimation for all already mapped instances gi, according to the model of Section 4.3. The schedulability test is performed in lines 28 and 34, by checking that the current estimations do not become greater than the reserved time intervals, as explained in Section 5.3.

5.5 Synchronization synthesis

While the principles of synchronization synthesis were defined in Section 5.3, we detail the synthesis process using the more complex example of Figure 5.4. We assume this example is mapped on 3 computing cores, and that the scheduling table is provided in Figure 5.5.

The scheduling table has a length of 1000 time units. Inside each of the 8 function instance reservations we highlighted in red the time intervals (of length δ) dedicated to the beginning and ending synchronization points. The scheduling algorithm ensures

[4] Zero-size reservations are typically used when scheduling multi-period applications. In this case, the normalization phase of Section 2.3, and in particular hyper-period expansion, will produce multiple instances of the same function that share the same input and output variables. While the first instance will require a non-zero memory interval, subsequent ones require no supplementary memory.

```
1  period(1000) node main () returns ()
2  var
3    v1,v2,v3,v4,v7 : int ;
4    v5,v5d : int at v5 ;
5    v8,v8d : int at v8 ;
6    v9,v9d : int at v9 ;
7  let
8    v1      = n1() ;
9    v2      = n2() ;
10   v3      = n3(v5d,v8d) ;
11   v4      = n4(v1,v2,v9d) ;
12   v5      = n5(v1,v2) ;
13   ()      = n6(v4) ;
14   v7      = n7(v2,v3,v4) ;
15   (v8,v9) = n8(v5,v7) ;
16   init<<v5>> v5d     = 0 fby v5 ;
17   init<<v8>> v8d     = 0 fby v8 ;
18   init<<v9>> v9d     = 0 fby v9 ;
19 tel
```

Figure 5.4: Simple integration specification used to exemplify synchronization synthesis

that the time reservations made for synchronization points never overlap, allowing their sequencing.

Code generation takes advantage of this property and enforces the total ordering of the synchronization points using a number of hardware locks/mutexes equal to the number of cores. In our case, we use locks of indices 0, 1, and 2.

The red arrows—both solid and dashed—materialize the point-to-point ordering between synchronization points that results in full sequencing. To optimize code generation, we note that certain dependencies do not need implementation using mutex-based synchronization (the ones corresponding to dashed red arrows). For instance, sequencing the start synchronization points of the first operations on each core is not needed, as they are not source or destination of data or control dependencies. Similarly, sequencing the last 3 synchronization points on core 2 is not needed, as they are naturally sequenced by the flow of sequential execution on the core.

Now that we know which dependencies must be encoded using mutex operations, we make the convention that the hardware lock of index i is used to unlock computations on core i, for $0 \leq i \leq 2$. Then, a solid red line from core i to core j will result in the synthesis of `unlock(j)` on core i, and `lock(j)` on core j. When a synchronization point is both source and destination of solid arrows (dependencies requiring encoding), then during code generation the `lock` operation is placed before the `unlock` operation.

In Figure 5.5 we have pictured above each solid dependency arc the lock index used for synchronization, and inside each synchronization point the lock and unlock operations that are generated. For instance, the end synchronization point of `n5` will

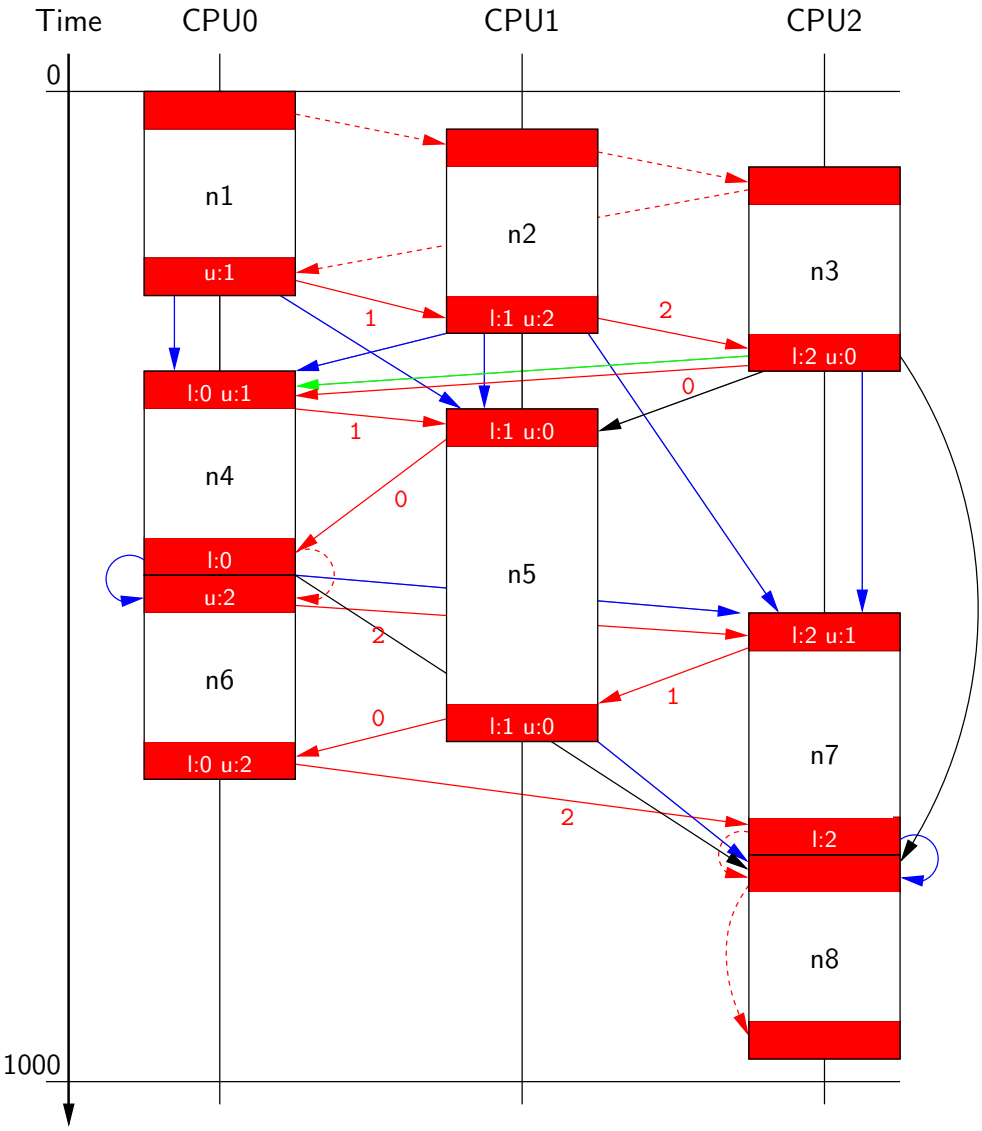


Figure 5.5: Scheduling table and synchronization synthesis for the example in Figure 5.4

generate the code:

```
lock(1,1) ;
unlock(0) ;
```

The red arrows of our figure and the mutex operations implementing them enforce at execution time the DAG ordering between operations, as defined in Section 4.4. Thus, they enforce the respect of:

- Data dependencies imposed by the dataflow semantics (in blue in Figure 5.5).
- Anti-dependencies ensuring that the simplified encoding of `fby` equations is correct (in black in Figure 5.5).

- Control dependencies ensuring property (preservation of interferences) of Section 5.2.1, *i.e.* that two nodes cannot interfere at execution time if they do not overlap in the scheduling table and access the same memory bank (in green in Figure 5.5). In our example, function `n3` must complete execution before `n4` because they both access the memory bank where the outputs of `n8` are allocated.

. Note that the relation between required control or data dependencies and their implementation using mutex operations (solid red arrows) is not always direct. For instance, the (dataflow) data dependency between `n5` and `n8` (in blue) is enforced by two red arrows: the one connecting the end of `n5` with the end of `n6`, and the one connecting the end of `n6` with the end of `n7`.

5.5.1 Potential trade-offs

The main drawback of our method for communication synthesis is that it enforces unneeded dependencies. In some cases, like in Figure 3.5 (page 58) the unneeded dependency enforced through lock 1 between the snippets associated with `f` and `g` can be optimized out. But the total ordering requirement necessarily results, for non-trivial examples, in a synchronization structure that is more rigid than necessary. In Figure 5.5, the execution of function `n6` requires *a priori* no synchronization with the sequence of `n7` and `n8`.

We have pictured in Figure 5.6 a synchronization pattern that does not over-constrain scheduling, only enforcing mandatory synchronizations. Note that the arcs of the synchronization pattern (in red) are a sub-set of the set of data and control dependencies we are enforcing (the same as in Figure 5.5). It is in fact a minimal sub-set enforcing the desired partial order. Determining such a minimal sub-set is done through the use of vector clocks [95].

As evidenced by our example, a minimal synchronization approach has drawbacks of its own. First, a function instance must potentially wait for $N-1$ mutexes, and unlock as many (whereas in the previous synchronization approach, at most 2 synchronization operations are needed per synchronization point).

The second drawback is that time intervals with dense cross-synchronization patterns appear. Cross-synchronization is not a problem *per se*, but becomes one when assigning hardware locks to implement the dependencies. Indeed, two synchronizations (red arrows in Figure 5.6) can be assigned the same hardware lock only when both source and destination of one are ordered before the source and destination of the other by other synchronizations or by the sequencing of operations inside a thread. In our case, none of the 6 dependencies can be thus ordered, so 6 hardware locks are needed. For instance, the dependency from `n3` to `n4` and the one from `n4` to `n7` cannot use the same hardware lock, because doing so would potentially let the `lock` operation guarding `n7` consume the token meant to unlock `n4`.

For large, very concurrent examples like our first use case of Section 6.1.1, this means that a minimal synchronization approach is not implementable on our target platform, which provides only 127 hardware locks.

Between the synchronization method that sequences all synchronization points (which

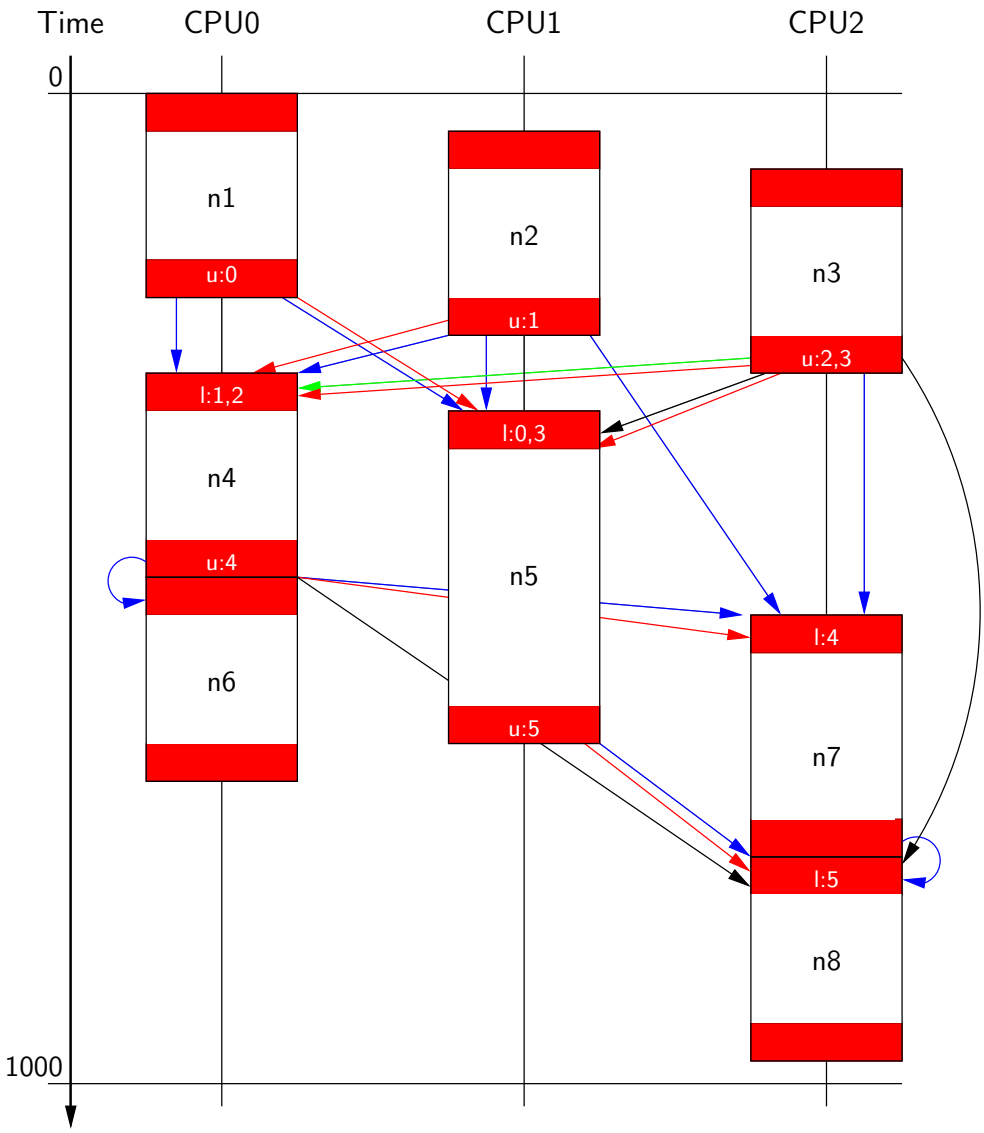


Figure 5.6: Minimally-constraining synchronization pattern for the example in Figures 5.4 and 5.5

we use for most of our experiments) and the minimally constrained synchronization method introduced in this section (which may render some systems non-implementable), a large exploration space is open for the definition of efficient synchronization methods. The trade-off is between supplemental constraints, on one side, and complexity of the generated code (necessary hardware resources, number of synchronization operations) on the other. We have only briefly explored some of them in Section 6.2.5.

Chapter 6

Experimental evaluation

At this point, the definition of our synthesis tool flow, whose overall structure is pictured in Figure 1.3 (page 18), is complete. This chapter is dedicated to its experimental evaluation on two large real-world case-studies provided by our industrial partners in the framework of the ITEA3 ASSUME project.

6.1 Industrial use-case definition

Both use-cases are large-scale avionics applications. We refer to them as UCA and UCS. Both are multi-period applications following a major frame/minor frame (MAF/MIF) execution pattern common in avionics (defined below).

While in both cases the specification is based on the SCADE dialect of Lustre, its form was not fully standardised to match the form described in Chapter 2. Instead, it follows the internal industrial process of the use case provider. A specific import phase (detailed below) was necessary in both cases to put the application in the form we require.

For both UCA and UCS, the use case specification is the one that supported *sequential* code generation for a production *single-core* architecture. One consequence of this is that the two specifications were not developed with parallelization in mind, and include artefacts specifically meant for sequential code generation. One key difficulty in the import process was to strip down the requirements related to sequential code generation so that parallelization gains are not artificially limited, but without changing the function or the fundamental real-time requirements.

6.1.1 Use case UCA

The use-case was initially presented under the form of three artefacts:

Non-deterministic dataflow. A set of 5124 dataflow nodes connected through more than 34189 communication variables. The dataflow node set is flat, with no hierarchy. Each of the nodes is specified in SCADE, but the integration of the nodes into a dataflow is not: it is implicitly defined by the names of input and output arguments of SCADE nodes.

Each node is associated a period of 10, 20, 40, or 120 ms. Each variable is associated a type. Nodes can be stateful or not.

The dataflow dependency system induced by communication variables is cyclic, meaning that this specification part alone cannot be seen as a deterministic functional specification.[1]

Sequencing costraints. For each period $p \in \{10, 20, 40, 120\}$, a partial order between nodes of period p. This definition assumes that execution is divided into sections of length p. Inside each section, nodes of period p must be executed exactly once, in an order compatible with the specified partial order.

In addition, fully ordered subsets of nodes are identified, with the requirement that their execution is performed in sequence without interruption. These are meant to represent chains of computations whose duration must be tightly bounded on the (slower) single-core execution platform

Sequential implementation. The static schedule of the legacy single-core implementation. This implementation follows a classical MIF/MAF execution pattern. Execution is divided into 5 ms intervals (the MIFs). Execution is triggered by the timer at the beginning of each MIF. The code of each MIF is a function containing a sequence of dataflow node calls, whose execution must be completed before the next MIF is started. Execution is non-preemptive. There are 24 MIF functions forming a pattern of 120 ms (the MAF) that is repeated indefinitely. We shall also use the notations MIF and MAF to refer to the durations of the two time intervals. Execution of nodes is periodic—if a node is first executed in MIF 3 (of start offset 3*5 ms with respect to the beginning of the MAF) and has period 40 ms=8*MIF, then it is also executed in MIFs 11=3+8 and 19=3+2*8.

We start by noting that the first two pieces of use case specification, which were the system specification in the internal process of the use case provider, have two undesirable properties:

1. They do not determine a deterministic functional specification.[2] This often happens in the engineering of embedded control systems, where the stability of the system under control can be exploited to facilitate the mapping process. In particular, this can be done by replacing direct node dependencies (within a cycle) with delayed dependencies through the introduction of `fby` equations. Such transformations are avoided whenever possible, as they require validation by the control engineering teams, but are permitted by the design process.

2. They contain artefacts related to the sequential implementation. In particular, the sequencing constraints between nodes of the same period leave only little con-

[1] Some cyclic Lustre/SCADE specifications can be given deterministic semantics. However, breaking a cycle to allow execution requires the use of operators that can produce their output without input from the cycle. The only Lustre/SCADE operator allowing this are `if` and `merge` (conditional control). Our specification does not contain these operators and is thus non-deterministic.

[2] For instance, the communication patterns between nodes of different periods are not specified.

currency. This almost complete sequencing is not justified by data dependencies or other system-level requirements.

Our first objective is therefore be to derive a more standard functional and non-functional specification containing no artefacts specific to the sequential implementation.

In doing so, we take a classical "software parallelization" approach by assuming that:

- The functionality of the sequential implementation must be preserved in the parallel implementation. In doing so, we choose not to exploit the stability of the system and the degrees of freedom visible under the form of specification non-determinism.
- Sequencing requirements not due to data dependencies are discarded.

The sequencer is therefore used to extract the data dependencies and the initial variable values needed to transform the non-deterministic dataflow into a fully deterministic activation and data dependency pattern. This pattern can be encoded under the form of a Heptagon integration program following the guidelines of Section 2.3.2 (page 45).

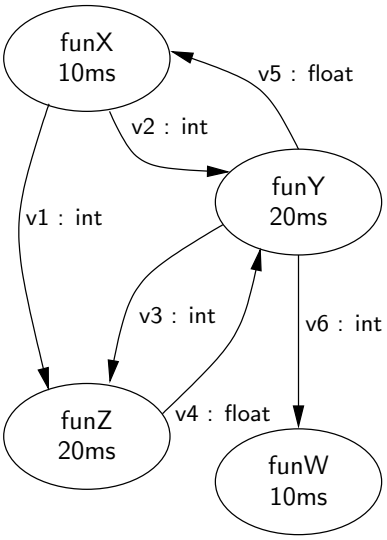
We illustrate the transformation process with the simple example of Figures 6.1 and 6.2, which only considers nodes of two periods (10 ms and 20 ms) and produces a specification with a MIF of 5 ms and a MAF of 20 ms. The simplified input used during importation is provided in Figure 6.1 (left). Note that we only consider the non-deterministic dataflow and the sequential schedule. The partial order specification is ignored. The sequential schedule is first used to determine the activation pattern (the clock) of each node, which is computed in our Heptagon program under the form of Boolean variables (`c_x` through `c_w`). Activation on this condition is enforced through sampling of the node inputs.

Upsampling and downsampling of variables during communication between tasks with different activation is treated in a uniform way by creating a persistent version of each variable, available at each execution cycle (variables `v1-v6`) through the upsampling of the primary outputs of the node instances (variables `v1p-v6p`).

The sequencer is also used to determine the order in which nodes must be executed, *but only when this order influences the result of the computation.* For instance, the order in which `funW` and `funZ` are executed in the last MIF of the MAF is ignored. On the contrary, the order in which `funX` and `funY` are executed in the first MIF determines which of the variables `v2` or `v5` is transmitted within the MIF, and which one is delayed. In our case, the sequential schedule executes `funX` before `funY`, so the input of `funX` is delayed using a `fby`.

The result of the importing process on the full UCA specification, after normalization, has 18672 node instances. Given its size, we considered that several of its sub-sets can be seen as representative, and be used to provide (less complete, yet meaningful) experimental evaluations. These are:

- Each of the 24 MIFs of the application, taken separately, seen as a single-period specification of period MIF. These applications are denoted UCA_i, for $0 \leq i \leq 23$. These applications are already quite large, with 690 node instances on average.



(a) Non-deterministic dataflow

MIF_0 5ms	funX; funY;
MIF_1 5ms	funW;
MIF_2 5ms	funX;
MIF_3 5ms	funW; funZ;

(b) Sequential schedule

```
period(5) node main () returns ()
var
  v1,v2,v3,v4,v5,v6 : int ;
  v1p,v2p,v3p,v4p,v5p,v6p : int ;
  cnt:int ; c_x,c_y,c_z,c_w:bool ;
let
  (* computation of activation clocks *)
  cnt = 0 fby ((cnt+1)%4) ;
  c_x = (cnt%2)=0 ;
  c_y = cnt=0 ;
  c_z = cnt=3 ;
  c_w = (cnt%2)=1 ;

  (* node activation *)
  (v1p,v2p)     = funX(0 fby (v5 when c_x)) ;
  (v3p,v5p,v6p) = funY(v2 when c_y,0 fby (v4 when c_y)) ;
  v4p           = funZ(v1 when c_z,v3 when c_z) ;
  ()            = funW(v6 when c_w) ;

  (* communications *)
  v1 = merge c_x (true->v1p) (false->((0 fby v1) whenot c_x)) ;
  v2 = merge c_x (true->v2p) (false->((0 fby v2) whenot c_x)) ;
  v3 = merge c_y (true->v3p) (false->((0 fby v3) whenot c_y)) ;
  v4 = merge c_z (true->v4p) (false->((0 fby v4) whenot c_z)) ;
  v5 = merge c_y (true->v5p) (false->((0 fby v5) whenot c_y)) ;
  v6 = merge c_y (true->v6p) (false->((0 fby v6) whenot c_y)) ;
tel
```

(c) Integration program

Figure 6.1: Import method for use case UCA. Small sub-set (left) and integration program (right)

- The sub-set of the application containing the nodes of periods 10 or 20 ms. This application has MAF=4*MIF=20 ms. It contains 757 unique nodes and 1488 node instances after normalization (hyper-period expansion). We denote this application UCA-RED (for *reduced*).

As an indication of the quantitative properties of the dataflow nodes, the WCET of their step functions after code generation ranges from 37.5 ns to 60.66 μs, under the analysis methodology of Chapter 4.

```
period(20) node main () returns ()
var
  (* Exposed state variables *)
  sX_0, sX_2, dsX_2 : state_X at sX ;
  sY_0, dsY_0 : state_Y at sY ;
  sZ_3, dsZ_3 : state_Z at sZ ;
  sW_1, sW_3, dsW_3 : state_W at sW ;
  (* Original dataflow variables (possibly replicated) *)
  v1_0,v1_2: int at v1;
  v2_0,v2_2: int at v2;
  v3_0 : int ; v6_0 : int ;
  v4_3, dv4_3 : float at v4 ;
  v5_0, dv5_0 : float at v5 ;
let
  (* MIF 0 *)
              deadline(5)  (sX_0,v1_0,v2_0)      = funX(dsX_2,dv5_0) ;
              deadline(5)  (sY_0,v3_0,v5_0,v6_0) = funY(dsY_0,v2_0,dv4_3) ;
  (* MIF 1 *)
  release(5)  deadline(10) sW_1                  = funW(dsW_3,v6_0) ;
  (* MIF 2 *)
  release(10) deadline(15) (sX_2,v1_2,v2_2)         = funX(sX_0,v5_0) ;
  (* MIF 3 *)
  release(15)              sW_3                  = funW(sW_1,v6_0) ;
  release(15)              (sZ_3,v4_3)           = funZ(dsZ_3,v1_2,v2_2,v3_0) ;

  init<<sX>> dsX_2 = init_funX fby sX_2 ;
  init<<sY>> dsY_0 = init_funY fby sY_0 ;
  init<<sZ>> dsZ_3 = init_funZ fby sZ_3 ;
  init<<sW>> dsW_3 = init_funW fby sW_3 ;
  init<<v4>> dv4_3 = 0 fby v4_3 ;
  init<<v5>> dv5_0 = 0 fby v5_0 ;
tel
```

Figure 6.2: Normalized version of the integration program of Figure 6.1(right)

6.1.2 Use-case UCS

This use-case is quite different from UCA. A traditional SCADE-based process was used here to perform the full functional specification of the application, including the specification of periodic activation and communication patterns. SCADE code is very hierarchical. An adaptation of the Heptagon compiler, realized by G. Iooss from the Inria PARKAS team, allowed the translation of the high level of the application into Heptagon.

To specify periodic activation, computation of activation clocks is performed at multiple hierarchy levels. Computation and communication code surrounding it follows a structure close to that of the examples in Figure 6.1. Thus, UCS also follows a MIF/MAF activation pattern with MIF=15 ms and a MAF formed of 16 MIFs for a total of 240 ms.

As the application is very hierarchical, parallelism must be exposed to our mapping

method using inlining annotations, as proposed in Section 2.2.1.

The computation of conditional activation at several hierarchy levels poses an efficiency problem: if activation is hidden inside a node compiled to sequential code, then its execution time may significantly change between activations in different MIFs. If our method considers only the largest WCET estimation, it may impact the guaranteed performance of generated implementations. To avoid this problem, we assume that inlining exposes all hierarchy levels where periodic clocks are computed.

The normalized specification contains 4792 function instances whose sequential functions have WCET ranging from 1 to 994 μs. Overall, UCS (while quite large) is smaller than UCA but its nodes are of significantly larger grain. Like UCA, UCS is also a specification meant for sequential implementation and, unlike in UCA, there is no rapid method for eliminating specification artefacts hampering parallelization.

6.1.3 General considerations

For both case studies, a significant effort has been needed to build complete specifications for our back-end. This effort was not only of scientific nature but also included an important part of engineering. We had to carry out the normalization of the initial specifications, as presented above, build scripts for the timing analysis of their components and the production of our architecture-dependent input, and tune their compilation to ensure the respect of our timing analysis requirements. The effort is highly dependent on the industrial internal process and coding practices. The cost of this engineering effort should be taken into account for any industrial application of our method. This is not unique to our approach.

6.2 Experimental results

The experiments with UCA and UCS had three main objectives:

E1 Evaluate the scalability of the compilation toolflow.

E2 Evaluate the guaranteed efficiency of the generated parallel code.

E3 Validate the correctness of the generated code through execution on actual hardware, and evaluate its execution efficiency.

The code and data of the full applications UCA and UCS do not fit in the memory of the test platform (1.65 MB available with the system configuration available for our experiments, cf. Section 3.3.2). Executing them on the test platform would require the use of memory paging mechanisms (overlays). Such an abstraction remains challenging to implement in a hard real-time environment and is not yet available in our framework.

However, to allow the evaluation of our parallelization method on the code of the use cases, which we consider as representative for a certain class of critical embedded control systems, we have performed the following:

- To evaluate objectives E1 and E2, we have configured the mapping, code generation, and WCET analysis tools of our tool flow to assume a larger SRAM memory. We have done this by setting in the architecture description file the memory bank size to a large-enough value allowing the allocation of all code and data. We have also evaluated these objectives on the partial specifications UCA_i, $0 \leq i \leq 23$, and UCA-RED.

- We evaluated objective E3 on the UCA-RED application alone.

In order to evaluate the synchronization synthesis methods, we have also applied memory and synchronization optimization techniques UCA_i, $0 \leq i \leq 23$.

6.2.1 Scalability

To evaluate scalability, we focused on the performance bottleneck of the compilation flow – the scheduling and allocation algorithms defined of Section 5.4. This evaluation was done on the full UCA and UCS, as well as the 24 MIF of UCA independently, that we denote UCA_i for $0 \leq i < 24$. On a 4-core Intel Core i7 architecture with 16 Gbytes of RAM, the scheduling and allocation of the largest example (full UCA) on 8 (respectively 16) cores cores took 29.22 s (respectively 42.97 s), the compilation of the UCA_i took 0.23 s (respectively 0.51 s) on average, and that of UCS 3.72 s (respectively 3.51 s).

For UCA_0 we plotted in Figure 6.3 the compilation speed as a function of the number of target cores (note that resulting code is unimplementable on the target platform beyond 16 cores). The graph is quasi-linear, as the scheduling algorithm attempts mapping on each core before chosing the best solution.

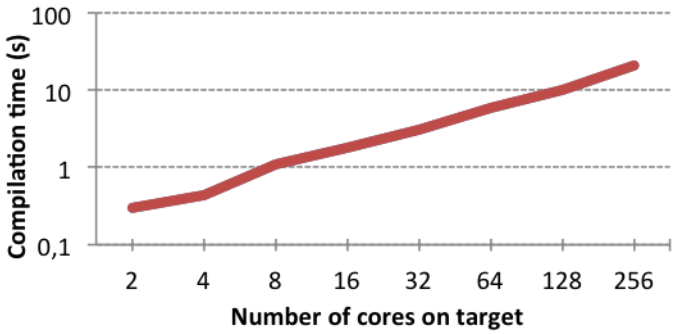


Figure 6.3: Parallel back-end compilaton time of UCA_0 as a function of the number of cores (logarithmic scale on both axes)

6.2.2 Parallelization efficiency

Parallelization of MAF/MIF applications

Our use cases are critical embedded applications, where the main performance-related goal is not to optimize some metric such as speed (throughput), memory footprint, or energy consumption. Instead, it is to produce an implementation that is functionally correct and which respects the real-time requirements specified by the integration program.

However, to evaluate the parallelization efficiency of our method, we need to introduce an efficiency metric. For single-period applications, the metric is natural: the ratio between the minimal period guaranteeing schedulability in the sequential case and the minimal period guaranteeing schedulability for the parallelized code. For MIF/MAF applications, this metric can be generalized into the ratio between the minimal MIF value guaranteeing schedulability in the sequential case and the MIF value guaranteeing schedulability for the parallelized code. This metric can also be generalized to include release date and deadline requirements, by scaling them in proportion to the MIF.

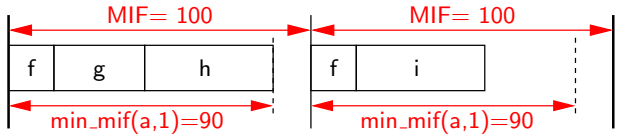
For an application a implemented on c cores with p interference provisions, we make the following notations:

- If $c \geq 2$, implementation is performed using our method and we denote:
  - $\text{min\_mif}(a, c, p)$ is the minimal MIF duration allowing implementation with our tool. If implementation is possible for some larger MIF value, then $\text{min\_mif}(a, c, p)$ can be computed as the maximum among all MIFs of the maximum among all function instances of the MIF of the end offset of the function instance with respect to the MIF start.
  - $\text{op}(a, c)$ is the optimal interference provisions, *i.e.*, the value of p that maximizes $\text{min\_mif}(a, c, p)$ for given a and c.
- If $c = 1$ (the sequential case):
  - $\text{min\_mif}(a, 1)$ is computed as the maximum of the per-MIF sum of block WCETs, including function call overhead, but no synchronization, cache coherency, on interference.
  - $op(a, 1)$ is naturally 0, as no interferences must be provisioned.

With these notations, the *guaranteed speed-up* of the parallel code with respect to the sequential reference is formally defined as

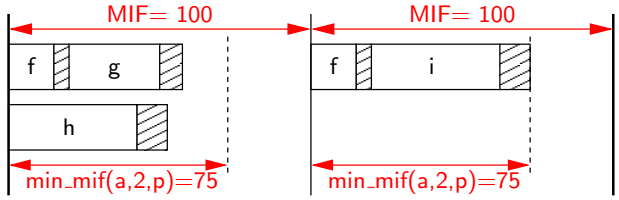
$$\text{gso}(a, c) = \frac{\text{min\_mif}(a, 1)}{\text{min\_mif}(a, c)}$$

An upper limit on the guaranteed speed-up for a given application can be obtained using the critical path method [96]. For each MIF m we determine its critical path, weighted with the WCET of the functions $\text{cp}(a, m)$. The computation of the critical path must consider both dataflow dependencies, and anti-dependencies required to allow



(a) Sequential case, no costs related to parallelization

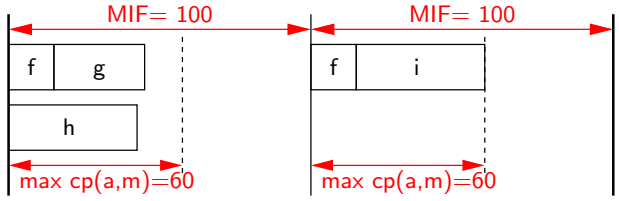
Baseline for schedule length: $2 \times 90 = 180$



(b) Parallel case, including all overheads

Minimum schedule length for 2 cores: $2 \times 75 = 150$

Guarranteed speed-up: $90/75 = 1.2\times$



(c) Theoretical limit, no overheads, infinite number of cores

Ideal schedule length: $2 \times 60 = 120$

Theoretical speed-up limit: $90/60 = 1.5\times$

Figure 6.4: Parallelization efficiency metrics for a simple MAF/MIF application

the simplified implementation of `fby` equations. Then, the theoretical parallelization limit of the application is computed as:

$$\mathrm{pl}(a) = \frac{\mathrm{min\_mif}(a, 1)}{\max_m \mathrm{cp}(a, m)}$$

Figure 6.4 illustrates the computation of these values on a simple example with four nodes `f`, `g`, `h` and `i`. The period of `f` is equal to the MIF, which is 100 time units. The other nodes have period equal to the MAF, which is 200 time units. The WCETs of the nodes are respectively 20, 30, 40, and 40 time units. We assume that the only dependencies to be considered during critical path analysis are from `f` to `g` in the first MIF and from `f` to `i` in the second. Figure 6.4(a) provides a mono-processor implementation of the example. The minimal MIF length guaranteeing schedulability is $\mathrm{min\_mif}(a, 1) = 90$.

Figure 6.4(b) provides the result of the parallelization of the example on two cores, accounting for a given interference provision $p = op(a, 2)$ and including all overheads (the shaded parts). We assume that the minimal schedulable MIF length is $\mathrm{min\_mif}(a, 2, p) = 75$, bound by the second MIF. Therefore, the guaranteed speed-up for this implementation is $\mathrm{gso}(a, 2) = 90/75 = 1.2$.

Figure 6.4(c) applies the critical path method, which amounts to parallelizing while assuming an infinite number of cores and no overheads or interferences. The longest path among the MIFs is the one formed by `f` and `i`, with length 60. Therefore, the theoretical upper-bound on the speed-up is $\text{pl}(a) = 90/60 = 1.5$.

Parallelization evaluation on the use cases

The speed-up figures $\text{gso}(UCA, c)$ and $\text{gso}(UCS, c)$ for $c = \{2, \ldots, 16\}$ are provided (in blue) in Figure 6.5, along with the upper bounds pl(UCA) and pl(UCS) (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.

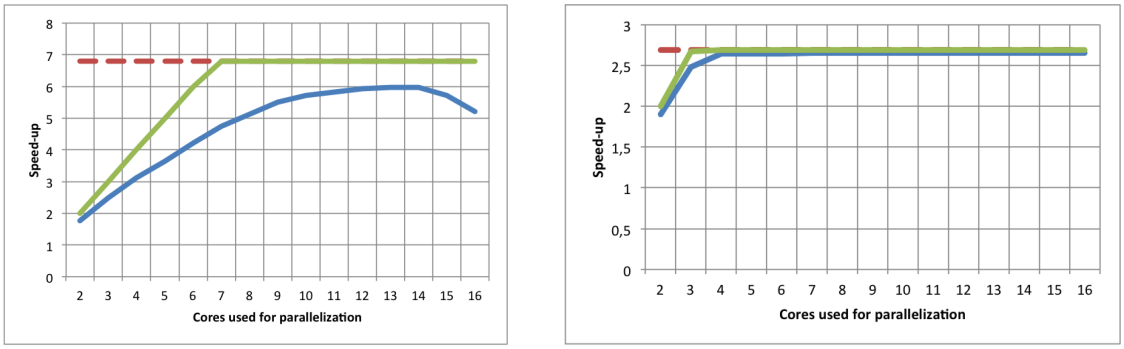


Figure 6.5: Speed-up figures for UCA (left) and UCS (right)

Clearly, the parallelism exposed in UCS is quite limited, with a theoretical upper bound of 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(UCS)). 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. To improve parallelization results, specification changes are needed—exposing finer-grain parallelism than the one currently exposed through inlining annotations, or even changes of the functional specification, as explained in Section 6.1.1.

UCA exposes significantly more parallelism, at a finer grain. The guaranteed speed-up is very good, reaching 5.98× on 14 cores and coming within 12% of the theoretical limit. However, beyond 14 cores, overheads come to dominate parallelization gains. To determine the causes of this behavior, it is interesting to consider the parallelization results on UCA_i, in Figure 6.6(left). Notice that most UCA_i parallelize better than the whole UCA does. This is normal, because the results in Figure 6.5 must consider the worst-case among the MIFs, and because during parallelization of the full UCA the MIFs constrain one another through the memory allocation, resulting in lower overall performance. Also note that for all UCA_i performance is lost beyond a certain number

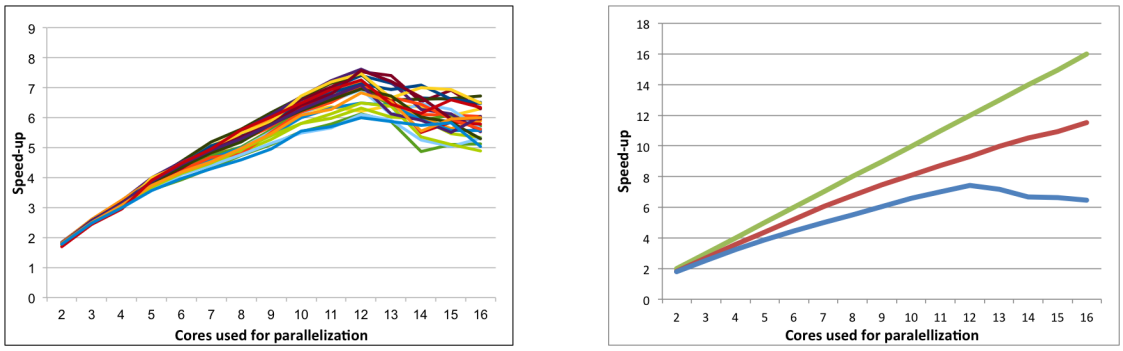


Figure 6.6: Guaranteed parallelization for UCA_i, $0 \leq i < 24$ (left). Contribution of overheads for UCA_0 (right).

of parallelization cores. To provide insight into the contribution of the various overheads in this performance loss, we provide in Figure 6.6 (right) the guaranteed speed-up for UCA_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 Figure 6.5, 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.

We conclude that experimental data seems to indicate that the performance loss is due to memory access interferences, which come to dominate parallelization gains beyond a number of cores. This phenomenon, also known as saturation of the memory bandwidth [97], is well-documented in single- and multi-core scheduling. Our results shows that, beyond providing hard real-time guarantees, our timing model can predict this phenomenon, suggesting that it could be of use to engineers in platform dimensioning.

Measurement-based performance. 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 the guaranteed speed-up we have measured execution duration and compared it to measured duration of a sequential implementation. We have done this for each of the 24 MIFs of UCA, and then taken the average and variance. Results are provided in Figure 6.7 (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 UCA_i.[3] The acceleration of the code with respect to measured sequential execution is lower than its statically predicted counterpart, which is normal, because the code was parallelized based on worst case quantitative data in both cases. However, it remains efficient and exhibits the same pattern of performance loss due to memory bandwidth saturation.

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

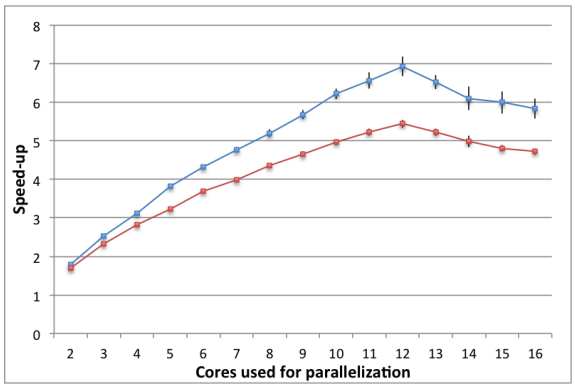
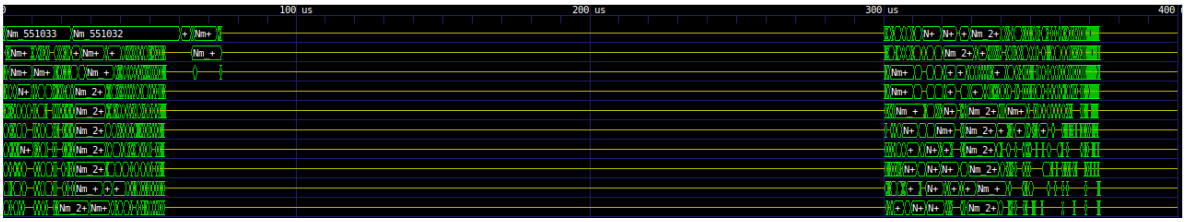


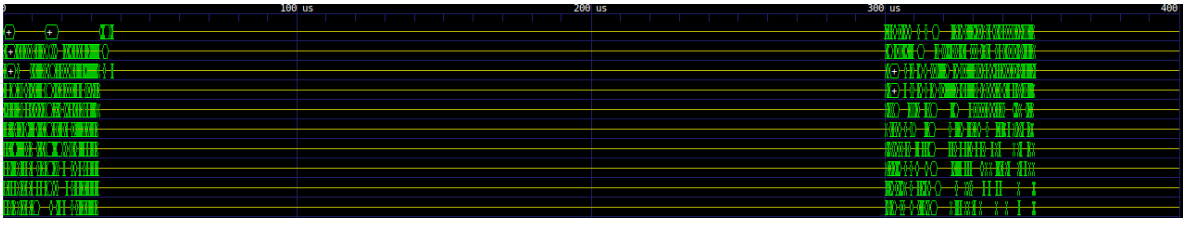
Figure 6.7: Predicted (in blue) vs. measured (in red) performance for UCA_i

6.2.3 Correctness

To validate the correctness of the toolflow output, we have executed the parallel implementations of UCA_i 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. Note that, while our real-time parallelization method is of "correct-by-construction" type, validation is by no means sufficient to guarantee the correctness of the implementations generated for real-life applications. It is done using only one application, and covers only real-time aspects and the absence of execution-time deadlocks. Significantly more validation work is needed to provide evidence that the resulting implementations can be used in real-life critical systems.



(a) Schedule



(b) Execution traces

Figure 6.8: Partial visualization of schedule and execution trace for UCA-RED

We also evaluated the functional correctness of our parallel implementations us-

ing UCA-RED. We compared the execution of the parallelized code produced by our method with the output of a sequential reference code, using provided vector test sets for the input variables. For a same input set, the traces on both single- and multi-core implementations were the same.

6.2.4 The cost of isolation

In Section 3.3.3 we have argued that our mapping and code generation approach, which enforces isolation properties only when absolutely necessary, is best suited for the fine-grain parallelization of applications where isolation between components is not a requirement. In such cases, isolation needlessly limits the implementation space, and impacts efficiency.

We provide here first quantitative results in support of our statements. They were obtained on use case UCA. Figure 6.9 provides the number of shared C variables in the code we generate (bottom line), compared with the number of variables when using per-node replication [81] (top), or per-core replication [19, 35], assuming the same allocation of nodes to cores as for our code. The significantly larger number of variables does not only count as extra data memory footprint, but also as copy operations, which take significant time and space (in code memory).

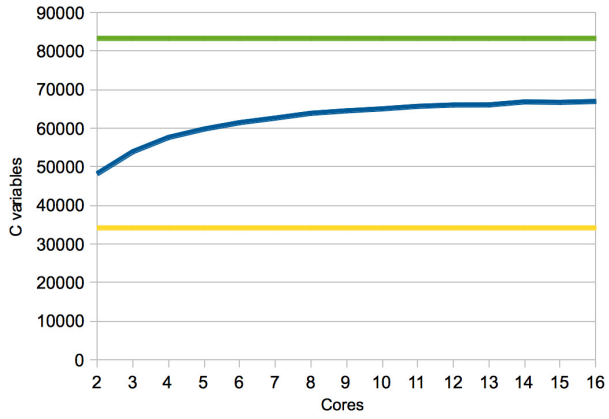


Figure 6.9: Number of C variables in code generated for UCA using our method (bottom), if we assume per-node variable replication (top) and if we assume per-core replication (middle)

Figure 6.10 shows the effect of requiring the absence of interferences in our method that does not replicate variables among cores/nodes. Peak performance is virtually halved.

In both Figures 6.9 and 6.10 the mapping method is our own, and we only vary mapping or code generation options. This poses a problem of representativity of our results, especially given that our mapping heuristics are not optimized to either:

- Reduce the number of per-core variable cores by encouraging joint allocation of nodes using the same variables.
- Parallelize while fully prohibiting memory access interferences for prolonged time periods (as it is done, for instance, in the BSP model [79]).

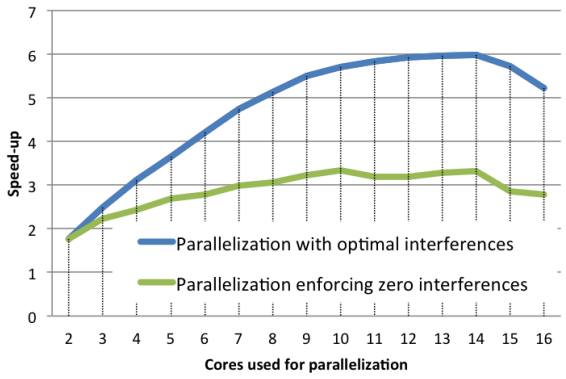


Figure 6.10: Guaranteed speed-up through parallelization. Normal output (blue) vs. no interferences allowed (red)

More experimental evaluation is needed in this direction, and our results should be seen as first results in this direction.

The final argument we make in this section considers the scheduling table and trace visualizations of Figure 6.8. They exhibit the classical difference between worst-case reservation (top) and average-case execution (bottom), with the latter smaller (often significantly smaller) than the former. As explained in Section 3.3.3, our use of event-driven synchronization (as opposed to time-triggered execution of each node) provides a certain robustness to timing errors. A timing estimation error on one or more nodes—nodes exceeding at run-time their alotted time budgets—could still not lead to a deadline miss error if the other nodes can absorb the overhead.

6.2.5 Memory use and synchronization optimizations

Up to this point, our experiments were performed with the main version of our method and tool flow, which respect two fundamental requirements:

- The *observability* requirement imposed by the industrial context: the value of a dataflow variable must be available to inspection at any point of execution. This requirement prohibits all memory reuse optimization allowing two temporary variables with disjoint lifetimes to share the same memory location.

- The *bounded execution time* requirement, imposed by the incremental timing analysis objective: we must be able to compute a tight static bound on the duration of a code snippet when the associated function instance is mapped, *i.e.* without knowledge of the mapping of yet unmapped function instances. In particular, as explained in Section 5.5.1, this requirement virtually imposes the use of synchronization mechanisms that over-constrain the implementation, such as the one of Section 5.3.2.

In this section we slightly extend the scope of our evaluations by including aggressive memory optimizations and hardware lock reuse optimizations on top of the synchro-

nization synthesis method of Section 5.5.1. The objective is to determine the memory and synchronization optimization potential of common embedded applications.

We have modified the memory allocation and synchronization synthesis algorithms as follows: During memory allocation, the scheduling table is used to determine the lifetimes of variables, which allows their optimized allocation onto the locations of the physical memory using register spilling methods. Input variables are excluded from this optimization, as it is more difficult to determine their lifetimes. Then, we used an optimized version of the synchronization synthesis method of Section 5.5.1 to reduce the number of needed hardware locks and needed synchronization operations.

We have applied this method on use cases UCA_i, $0 \leq i \leq 23$, and the results are provided in Table 6.1. Colums (Nodes) and (Vars) respectively provide the number of nodes and non-input variables of the use cases. The optimization only allows reuse between variables of the same type. For this reason, to evaluate the reduction in memory needs, we simply provide in column (Mem) the number of memory locations storing one variable at a time. The reduction in memory allocation needs is an impressive 71.2% on average.

MIF	Specification		Implementation		
	Nodes	Vars*	Mem*	Sync	Locks
UCA_0	778	6077	1951 (32.1%)	1040	96 (9.2%)
UCA_{16}	669	5008	1603 (32%)	958	108 (11%)
UCA_8	653	4813	1537 (31.9%)	907	120 (13%)
UCA_7	868	5239	1351 (25.8%)	1291	115 (8.9%)
UCA_{23}	905	5894	1294 (22%)	1370	118 (8.6%)
UCA_{15}	857	4945	1247 (25.2%)	1296	118 (9.1%)
Avg.			28.8%		10%

*Input variables not included.

Table 6.1: Memory and synchronization optimization figures for UCA_i, $0 \leq i \leq 23$

The number of point-to-point synchronizations (Sync) corresponds to the number of solid red arrows in the examples of Section 5.5.1. As explained in that section, a minimally-constraining synchronization synthesis algorithm does not produce code implementable on our test platform due to the limited number of hardware locks. For this reason, we have allowed the synchronization synthesis algorithm to trade some of the run-time scheduling freedom for a smaller number of hardware locks, allowing implementation on our test platform (as the needed number of locks is smaller than 127 in every UCA_i.

Part II

Back-end correctness formalization

Chapter 7

The Intelus language

In the first part of my thesis the focus was on the construction of efficient parallel implementations of real-time systems. In this second part, we are interested in the correctness of implementations. By correctness we understand:

- The absence of run-time errors, such as a mutex being unlocked while it is already unlocked.
- The preservation of the functional semantics of the Heptagon integration program by the parallel implementation code.
- The respect of non-functional (real-time) requirements.

In this thesis, I only consider on the first two points, which can be summarized as the functional correctness of the implementation. More precisely, my objective is to allow the full formalization of functional correctness.

We have seen in Chapter 2 how a *sub-set* of the Heptagon language is used for the platform-independent specification of our applications. In this chapter, we formalize the definition of this Heptagon sub-set and we build upon it a new dataflow synchronous language allowing the definition of both integration specifications and *implementation models*, *i.e.* faithful models of the parallel implementations we synthesize. We call the new language InteLus, for Integration Lustre.

As explained in Chapter 1, we believe that in the construction of parallel implementations of dataflow specifications, the compiler should not build just the multi-threaded C code but (first and foremost) the richer InteLus model exposing the fundamentally data-flow organization of the computations performed by the multi-threaded implementation. From this model, implementation C code can be extracted through selective pretty-printing, while knowledge of its data-flow organization facilitates analysis.

We could define InteLus as a strict extension of Lustre or Scade with new constructs for our new modeling needs. However, the system-level and mapping-oriented perspective makes some major features of Lustre/Scade unneeded, and including them would only pollute our presentation. For this reason, we stick to the set of constructs presented in Chapter 2.

The syntax of InteLus is provided in Figure 7.1. As the grammar rules show, the language has 3 layers:

```
<prog> ::= <type>* <fun>* var <var>* let <ceq>* <thread>+ tel
<type> ::= type <id> size <int>
<fun> ::= fun <id> (<list(<arg>)>) -> (<list(<arg>)>) <alloc>?
<arg> ::= <tid><lalloc>?
<var> ::= input? <nvl(<id>)> : <tid> <valloc>? ;
<tid> ::= bool | int | float | <id> | event
<ceq> ::= <guard> <wait>? <done>? <api> <eq> ;
<eq> ::= (<nvl(<lval>)>) = (<nvl(<sexpr>)>) | <lval> = <sexpr>
       | (<list(<lval>)>) = <fid>(<list(<sexpr>)>)
       | <lval> = <fid>(<list(<sexpr>)>)
       | <id> = <k> fby <sexpr> | <id> = <sexpr> when <id>
       | <id> = merge <id> <sexpr> <sexpr>
       | <id> = pre <id>                                   (non-primitive)
<sexpr> ::= <lit> | <id>
<lit> ::= true | false | <num> | top
<lval> ::= <id> | _
<list(X)> ::= | <nvl(X)>                                   (list meta-rule)
<nvl(X)> ::= X | <nvl(X)>,X                                (non-void list meta-rule)
<wait> ::= wait(<list(<id>)>)
<done> ::= done(<list(<id>)>)
<guard> ::= top | <id> | <guard> <wait>? <done>? on <id>
<alloc> ::= at <addr>
<lalloc> ::= at <id>
<valloc> ::= <alloc> | <lalloc> <cpuloc>? | at <muxid>
<cpuloc> ::= on <cpuid>
<thread> ::= thread <cpuloc> <alloc> stack <addr> <ceq>*
<api> ::=  [lock:<muxid>]
         | [unlock:<muxid>]
         | [inval:<addr>]
         | [flush:<addr>]
```

Figure 7.1: Lustre language subset (in black). InteLus extensions (blue). Extension with mapping information (red).

- The dataflow core that can be seen as a sub-set of Lustre or Heptagon. Its terminals and non-terminals are in black in Figure 7.1.

- A set of language extensions (in blue) that still follow a dataflow synchronous paradigm. These extensions are needed to allow the natural representation of the

functional part of implementation models.

- Language extensions allowing the representation of non-functional properties in implementation models (in red). These extensions allows the identification of threads, the specification of memory allocation properties, and the specification of machine operations implementing certain dataflow equations.

In this chapter we progressively introduce the language layers and their intuitive semantics. We also emphasize the proximity between implementation model and corresponding threaded C implementation.

7.1 Lustre/Heptagon sub-set

Unlike Lustre and Scade, which also allow the programming of the sequential tasks of an embedded system, InteLus is only designed to allow the system-level integration of these tasks. For this reason, an InteLus specification never needs to include/use another (there is no modularity), so full-fledged interface definitions are not needed. Like in the normalized integration specifications of Section 2.3.3, we assume that all inlining, hyper-period expansion, and node state exposure as dataflow variables has already been performed. Thus, an InteLus specification only instantiates stateless dataflow functions. To facilitate mapping, we identify input variables with the keyword `input`. They intuitively correspond to memory-mapped input devices.

To avoid modularity issues, we assume that an InteLus specification includes the declarations of the dataflow functions it uses. All custom data types must also be declared, but not defined—they are treated as opaque types, with no internal structure.

In InteLus, no complex dataflow expressions can be built. Each equation is either an assignment, or an instance of a function call or dataflow operator (`merge`, `when`, `fby`, or `pre`).

```
fun f:()->(float)
fun g:(int)->(int)
fun h:(float,int)->(int)
var
  input i : int;
  x : float; y : int; z : int; d : int;
let
  x = f(i);
  y = g(d);
  z = h(x,y);
  d = 0 fby z;
tel
```

Figure 7.2: Simple InteLus program

Figure 7.2 provides a simple InteLus program that uses only the core language constructs (the Lustre/Heptagon sub-set). This example is the InteLus version of the

Heptagon integration program of Figure 1.1, page 16. An implementation model (and associated C and linker script code) for this example has already been provided in Figure 1.4, page 19. This implementation model corresponds to the output of our mapping method.

We provide here, in Figure 7.3, a different dual-core implementation and implementation model for the same example. Its main particularity is that it *does not use global barriers, but only relies on mutex operations for thread synchronization.* We consider this exercise useful, as we want our formal modeling approach to cover not only the implementations we synthesize in Part I of this thesis (which use global barriers), but a larger class, and in particular allow for fully asynchronous executions where computations of the next synchronous cycle can start on one core while another core still performs computations of the current cycle (without affecting determinism).

```
fun f:()->(float) at 0x20100
fun g:(int)->(int) at 0x30300
fun h:(float,int)->(int) at 0x20500
var
  i:int at 0x80000 i0:int at i on cpu0
  y:float at 0x32000 y0:float at y on cpu0
  y1:float at y on cpu1
  z:int at 0x22004 d:int at z
  z0:int at z on cpu0 d1:int at z on cpu1
  x:int at 0x22000 x0:int at x on cpu0
  u,u1:event at 1 v:event at 0
  s0,s1,s2,s3,s4,s5,s6,s7,s8,t0,t1,t2,t3,t4,t5:event
let
   d = 0 fby z    // C code needed for state init
  u1 = top fby u  // C code needed for mutex init
  s8 = top fby s7 // thread 0 self-activation, no C code
  t5 = top fby t4 // thread 1 self-activation, no C code


thread on cpu0 at 0x20000 stack 0x30000
  s8          done(s0) [inval:0x80000] i0 = i
  s8 wait(s0) done(s1)                 x0 = f(i0)
  s8 wait(s1) done(s2) [flush:0x22000]  x = x0
  s8 wait(s2) done(s3) [lock:0]         _ = v
  s8 wait(s3) done(s4) [inval:0x32000] y0 = y
  s8 wait(s4) done(s5)                 z0 = h(x0,y0)
  s8 wait(s5) done(s6) [flush:0x22004]  z = z0
  s8 wait(s6) done(s7) [unlock:1]       u = top


thread on cpu1 at 0x30000 stack 0x40000
  t5          done(t0) [lock:1]         _ = u1
  t5 wait(t0) done(t1) [inval:0x22004] d1 = d
  t5 wait(t1) done(t2)                 y1 = g(d1)
  t5 wait(t2) done(t3) [flush:0x32000]  y = y1
  t5 wait(t3) done(t4) [unlock:0]       v = top
tel
```

(a)

```
1
2
3
4
5
6
7
8
9
10
11
12
13  void init(){
14    z = 0;
15    unlock(1);
16
17
18  }
19  __attribute__((section(.text.cpu0)))
20  void thread_cpu0(){ init0(); for(;;){
21      dcache_inval();
22      f(i,&x);
23      dcache_flush();
24      lock(0);
25      dcache_inval();
26      h(x,y,&z);
27      dcache_flush();
28      unlock(1);
29  }}
30  __attribute__((section(.text.cpu1)))
31  void thread_cpu1(){ init1(); for(;;){
32      lock(1);
33      dcache_inval();
34      g(z,&y);
35      dcache_flush();
36      unlock(0);
37  }}
```

(b)

Figure 7.3: Barrier-less implementation model (a) and corresponding C code (b) for the example in Figure 7.2. Line numbering is shared between the two listings.

7.2 Synchronous model extensions

The constructs of Lustre/Heptagon do not allow the faithful representation of the function and structure of C multi-threaded implementations like ours. For instance, Lustre does not allow the representation of control dependencies. To allow their representation while preserving the dataflow form of our language we introduce:

- The `event` type representing pure synchronization.
- The `wait` and `done` constructs allowing the representation of control dependencies.

The `event` type has a single value, denoted `top`, representing pure synchronization (*e.g.* mutex synchronization). This type already exists in Signal [64], with the difference that its only value is here `top`, and not `true`[1].

To illustrate the use of variables of type `event` to represent mutex-based synchronization, consider lines 24 and 36 of Figure 7.3. The structural proximity between C code and implementation model means that for each line of C, exactly one equation captures its semantics. In Figure 7.3, we ensured this equation is placed on the same line as the corresponding C. Thus, waiting on a lock in line 24 of the C code is represented in the implementation model (also in line 24) by the waiting for a value `top` on the variable `v` of type `event`. Note the use of `_` in the equation to denote the fact that the output is discarded, as we were only interested in the arrival (the `_` notation allows reducing the number of synchronization variables). The lock in line 24 is unlocked by the C statement in line 36. In the implementation model, this is represented by the assignment of `top` to `v`. The causality of the dataflow model semantics ensures that the dataflow equation on line 24 can only be executed after the equation in line 36.

To represent the sequencing of equations, we need to represent control dependencies. To do this, we rely on two novel constructs—`wait` and `done`—which manipulate signals of type `event`. When placed in front of an equation, `wait`$(s_1, \ldots, s_k)$ will delay the start of the equation until a `top` value (a token) can be read on each of the variables $s_1, \ldots, s_k$. When placed in front of an equation, `done`$(s_1, \ldots, s_k)$ will write `top` on each of the variables $s_1, \ldots, s_k$ after the completion of the execution of the equation. In Figure 7.3(a), we use `wait` and `done` to enforce the sequence of the equations inside each thread. For instance, the equation in line 25 can only start (due to `wait(s3)`) after the equation in line 24 has been executed, as this equation produces `s3` using the `done(s3)`.

Sequencing data-flow equations to build threads running in an asynchronous environment requires a normalization phase. For each equation of a thread, this phase builds a *guard*, which is the (possibly empty) cascade of tests needed to determine, at each execution cycle, if the equation is executed *or not*, thus allowing giving control in sequence[2]. Guards are placed in front of equations. A guard always starts with

[1] One can also see a similarity with the *pure signals* of Esterel. However, there are some significant differences: the semantics on InteLus is not constructive, and there is no single assignment property nor clock calculus meant to balance productions and consumptions in Esterel.

[2] The decision not to execute is particularly valuable in an asynchronous environment where absence of an event cannot *a priori* be detected.

```
fun P:()->(int)
fun C:(int)->()
var x:int
let
  x = P()
  ()= C(x)
tel
```

(a) Specification

```
fun P:()->(int) at 0x20100
fun C:(int)->() at 0x30200
var x0:int at 0x20500
    y:int at 0x30500 y0:int at y on cpu0
    yd:int at y yd1:int at y on cpu1
    c:bool at 0x30504 c1:bool at c on cpu1
    u,u1:event at 0 v,v1:event at 1
    s0,s1,s2,s3,s4,s5,t0,t1,t2,t3,t4,t5:event
let
  c  = false fby c1 ;// C code needed for state init
  yd = pre y ;    // same memory allocation, no C code
  v1 = top fby v ;// C code for mutex initialization
  u1 = u ;        // HW mutex semantics, no C code
  s5 = top fby s4 // thread 0 self-activation, no C code
  t5 = top fby t4 // thread 1 self-activation, no C code


thread on cpu0 at 0x20000 stack 0x30000
  s5          done(s0)                        x0 = P()
  s5 wait(s0) done(s1)      [lock:1]           _ = v1
  s5 wait(s1) done(s2)                        y0 = x0
  s5 wait(s2) done(s3)      [flush:0x30500]    y = y0
  s5 wait(s3) done(s4)      [unlock:0]         u = top


thread on cpu1 at 0x30000 stack 0x40000
  t5          done(t0) on c [lock:0]          _ = u1
  t5 wait(t0) done(t1) on c [inval:0x30500] yd1 = yd
  t5 wait(t1) done(t2) on c                  () = C(yd1)
  t5 wait(t2) done(t3)      [unlock:1]        v = top
  t5 wait(t3) done(t4)                       c1 = true
tel
```

(b) Implementation model

```
1
2
3
4
5
6
7
8
9   void init(){
10    c = 0;
11
12    unlock(1);
13
14
15
16  }
17  __attribute__((section(.text.cpu0)))
18  void thread_cpu0(){ init0(); for(;;){
19      P(&x);
20      lock(1);
21      y = x ;
22      dcache_flush();
23      unlock(0);
24  }}
25  __attribute__((section(.text.cpu1)))
26  void thread_cpu1(){ init1(); for(;;){
27      if(c) lock(0);
28      if(c) dcache_inval();
29      if(c) C(y);
30      unlock(1);
31      c = 1;
32  }}
```

(c) C code

Figure 7.4: Producer/consumer specification (a), dual-core barrier-less pipelined implementation (c), and corresponding implementation model (b)

a variable of type `event` (or the literal `top`) which identifies the *trigger* event of the cascade of tests.[3] Each test is made on a single Boolean variable. Each test variable is preceded by the `on` keyword. The constructs `wait` and `done` allowing the definition of control dependencies can be placed before each `on` keyword, allowing the definition of

[3]The evaluation of the cascade of tests only starts when this event arrives.

complex hierarchies of conditional control. Consider the following equation:

```
u on x wait(a) done(b) on y (z,m) = f(t)
```

The guard is marked in blue. The guard is evaluated during a cycle only after the arrival of a `top` value on `u`. Then, if `x=true`, a `top` value is waited for on variable `a` after the test on `x` was performed and before the test on `y` is executed. A `top` value is written on variable `b` after `f` is executed, if `y=true`, or just after the test on `y`, if `y=false`. Note how, unlike previous uses of guards in synchronous languages [98], our guards focus on synchronization, clearly representing the flow of control from trigger to cascade of tests and synchronizations, until passing control in sequence.

An InteLus program featuring non-trivial guards is provided in Figure 7.4 (b). It is a dual-core, barrier-less implementation model of the simple producer/consumer specification of Figure 7.4(a). The implementation uses a non-trivial optimization—software pipelining [56]—meant to increase the computation throughput by allowing the implementation to produce a value and consume the previously-produced one in parallel.

While in the functional specification of Figure 7.4(a) the production of a value by function `P` and its consumption by `C` are performed in the same cycle, enforcing a dependency between `P` and `C`, in the pipelined implementation the value produced in one cycle is consumed in the next one. Thus, no dependency exists between `P` and `C` inside a cycle. Of course, allowing `P` and `C` to run in parallel requires the creation of two physical copies of variable `x`—`x` and `y` in the C code. A copy operation (in line 21) transfers the data from `x` to `y` inside each cycle, allowing the next consumption to take place.

In the first execution cycle, `C` must not be executed, as `P` has not yet produced a value. This conditional activation is represented with Boolean variable `c`, which is `false` in the first cycle and `true` afterwards. Variable `c` is tested by equation guards in lines 27-29.

Note that `fby` equations do not generate thread code. For data variables (with type different from `event`) this means that implementation models satisfy the rules of Section 5.4.2, allowing the simplified implementation of delays.[4] Code is needed for these equations only in the initialization function `init`.

The `fby` equations on variables of type `event` are either implementing the semantics of HW mutexes (in lines 12 and 13), or the self-reactivation of the thread loop body when one cycle of execution is completed (in lines 14 and 15). When a mutex must be initially unlocked, as in line 12, code is needed in the initialization function.

7.3 Non-functional extensions

The previous two sections have introduced the functional sub-set of InteLus, which corresponds to the black and blue syntax elements in previous figures of this chapter. We now introduce the red syntax elements, which provide non-functional information:

[4] In other terms, the construction of the implementation model has instantiated the necessary copy operations.

- The structuring of equations into threads.
- The mapping of implementation elements (threads, variables) onto hardware resources (cores, memory addresses).
- The implementation of certain equations with API primitives of the execution platform.

This specification part is necessarily dependent of the choice of platform—hardware and primitive library.

7.3.1 Thread structure

Equations are grouped into threads using the `thread` annotation. An equation belongs to the thread identified by the first `thread` annotation above it. The implementation model in Figure 7.4(b) identifies 2 threads, one formed of the equations in lines 19-23, and the other of the equations in lines 27-31.

Equations above the first `thread` annotation are not part of threads. There are 3 types of equations that do not require code generation inside threads:

- Copy equations on data variables, when both source and destination variables have the same allocation.
- `fby` and `pre` equations.
- Equations representing the semantics of platform devices such as mutexes.

In Figure 7.4(b), this is the case for equations in lines 10-15.

We require that all equations of a thread have the same trigger variable, which represents the event triggering iterations of the thread body. In our examples, triggers implement self-activation after completion of one cycle. More generally, they can be used to represent other triggering mechanisms, such as interrupt-driven execution.

7.3.2 Location allocation

We use annotations to allocate:

- Threads on cores of the architecture.
- Thread code, data, and stack to memory.
- Data variables (of type different from `event`) to memory
- Synchronization variables (of type `event`) corresponding to mutex synchronization to hardware mutexes.
- The allocation of the code and data of external functions to memory.

The definition of each thread is accompanied by three annotations defining respectively the core it will be executed on, the base address where all code and local data are allocated, and the stack base. The first thread of Figure 7.4(b) is executed on core `cpu0`, its code and data are allocated at address 0x20000, and its stack base is 0x30000.

All data variables are allocated to memory. Line 3 of Figure 7.4(b) provides the basic allocation annotation, which simply provides the memory address (an integer) preceded by `at`. All data variables of an implementation model can be annotated in this way. However, to facilitate analysis and the understanding of implementation models, we allow these low-level annotations to be substituted in some cases with:

- "`at` *`var`*", when allocation is the same as that of variable *var*.
- "`at` *`var`* `on` *`cpu`*", when the variable represents the local view of core *cpu* (*i.e.* the value returned by its cache) when accessing variable *var*.

To understand the utility of these alternate annotations recall that, on a shared memory platform, accessing *at the same time* a given memory location may return different values (essentially due to the existence of caches). To faithfully represent the semantics of a multi-threaded shared memory implementation, the state of the memory sub-system (including the caches) must be represented. For instance, in Figure 7.3(a) variables `y0` and `y1` respectively represent the value of variable `y` as viewed from cores `cpu0` and `cpu1`, while `y` represents the value stored in main memory.

Knowing exactly what a data variable represents (the value in main memory or in a cache) is not mandatory when generating `C` code (only the address is needed). However, when alternate annotations are used, they:

- Allow the production of more legible code. For instance, in line 22 of Figure 7.3(a), code generation uses for `i0` and `x0` the locations associated with the target of the allocation annotations, thus reducing the number of C variables and making the code easier to read.
- Facilitate the analysis of the specification.

Note that the allocation annotations of InteLus do not currently include the size of memory regions allocated to the various objects. This information is needed to prove the correctness of memory allocation, but we assume it can be obtained through static analysis of the various components of the implementation, so that it is not included in the implementation model, which only includes *mapping and code generation decisions*.

7.3.3 Machine semantics of API call equations

Implementation model equations that represent calls to platform API primitives are annotated with their machine semantics action.[5] In the remainder of this thesis, we shall only cover the API calls used in barrier-less implementations: `lock`, `unlock`, `dcache_flush`, and `dcache_flush`. Extension to cover barrier operations is straightforward.

Each annotation has arguments, according to its type:

[5] The full definition of machine semantics will be done in Chapter 9.

- `dcache_flush` and `dcache_inval` annotations take as argument a memory address. The cache line containing this address is respectively flushed to main memory or invalidated. As a simplification meant to simplify the definition of the semantics, it is assumed that each variable fits inside one cache line.
- `lock` and `unlock` annotations take as argument the identifier of the mutex they act upon (in our case, an integer).

7.4 Implementation model completeness

An InteLus implementation model is considered *complete* when it allows the generation of multi-threaded code that can be compiled and linked by the C compiler. This requires:

- That the functional part of the application (obtained by removing the non-functional annotations) is correct.
- The allocation into threads of all equations save those identified in Section 7.3.1.
- The sequencing of all equations of a thread, using `wait`/`done` dependencies, thus allowing sequential code generation.
- Assigning a machine semantics to all equations part of a thread which are not calls to external function.
- Allocating all code and data to memory.
- Allocating all synchronization variables used to represent thread synchronization onto hardware mutexes.

Completeness does not imply correctness. Even if implementation code can be generated, this code may still produce run-time errors or output incorrect results.

Chapter 8

Platform independent semantics

The InteLus language offers the constructs to define not only traditional Lustre programs, but also their complete implementation on parallel platforms. To lay the ground for formal validation of such implementations, we need to define the semantics of InteLus. As explained in the introduction, we need to define two semantics:

- The synchronous semantics, derived from that of Lustre, which simply discards mapping annotations.
- The machine semantics, which interprets the program and its annotations as a multi-threaded imperative program.

In defining the correctness of an implementation model with respect to the original functional specification, the gap between synchronous semantics of the specification and machine semantics of the implementation is difficult to cross directly. To facilitate the definition of the correctness properties, we introduce a third semantics of InteLus, intermediate between the synchronous semantics and the asynchronous machine semantics:

- The kahnian semantics, which is an asynchrounous dataflow semantics interpreting the (deterministic) InteLus programs as a variant of Kahn Process Networks (KPN). Like the synchronous semantics, the kahnian semantics discards mapping annotations.

This section defines the two dataflow semantics—synchronous and kahnian. It also states a number of correctness properties InteLus programs must respect in order to allow implementation and correct execution on a platform with bounded resources.

8.1 Synchronous semantics

Semantics of Lustre have already been defined elsewhere [13, 99]. However, the InteLus semantics we define here has two elements of originality with respect existing Lustre semantics, which justify its presentation in full:

- It covers non-trivial extensions to Lustre, most notably the guards.

- It is a micro-step operational semantics, which facilitates the link with the operational machine semantics.
- It does not require the pre-computation of clocks, as the original semantics does [13].

To simplify semantics definition, and without affecting generality, we shall assume in this chapter that:

- Constants, such as integer literals, only appear as the first argument of a `fby` statement. In particular, they are never given as arguments to functions.
- No assignment equations "`x=y`" exist.
- No "`_`" lvalues are used.
- No `pre` operators are used.
- Input and output variables of a `fby` statement are different.
- Each equation has (possibly empty) `wait` and `done` constructs.

Any InteLus program can be transformed into one respecting these constraints through simple source-to-source transformations. For instance, the literals can be removed by replacing them with calls to constant functions.

The semantics of an InteLus statement or program p is described through structural operational semantics (SOS) transitions of the form:

$$p' \underset{I}{\overset{O}{\Rightarrow}} p''$$

Transitions describe the behavior of p for one execution cycle. In the transition p' and p'' are terms describing the state of p before and after the execution of the cycle, I is a valuation of all input variables of p, and O is a valuation of all variables that are not inputs.

8.1.1 Variable valuations

We denote with $\mathit{Type}(v)$ the data type of a variable v. In a transition, the value assigned by I or O to a variable v is either of the type $\mathit{Type}(v)$, or can be the special value `nil` representing the absence of a value. If V is a set of variables, we denote $\mathcal{R}_V = \prod_{v \in V} \{\mathit{Type}(v) \cup \{\texttt{nil}\}\}$, the set of all possible assignments of the variables in V. For $r \in \mathcal{R}_V$ and $v \in V$ we denote with $r(v) \in \mathit{Type}(v) \cup \{\texttt{nil}\}$ the value assigned by r to v.

We denote with $\overline{\mathcal{R}}_V$ the set of partial valuations over V. Given $r \in \overline{\mathcal{R}}_\mathcal{V}$ we define its support $\mathit{supp}(r)$ as the set of variables to which r assigns a value (possibly `nil`). We also define $\overline{r}$ as the (partial) valuation that retains of r only the values different from `nil`. Note that $\mathcal{R}_V \subset \overline{\mathcal{R}}_V$ and $\mathit{supp}(\overline{r}) \subseteq \mathit{supp}(r)$.

We denote with $\langle X_i \mapsto x_i \mid i = \overline{1,n} \rangle$ or $\langle X_1 \mapsto x_1, \ldots, X_n \mapsto x_n \rangle$ the element of $\overline{\mathcal{R}}_V$ assigning x_i to variable X_i, $i = \overline{1,n}$. The element of $\overline{\mathcal{R}}_V$ making no assignments is denoted $\langle\rangle$. The notation is ambiguous, as the domain V is not explicit.

If $I \subseteq V$ and $r \in \overline{\mathcal{R}}_V$, then $r \mid_I \in \overline{\mathcal{R}}_I$ is the restriction of r to the variable set I. Note that if $r \in \mathcal{R}_V$, then $r \mid_I \in \mathcal{R}_I$. If $I \subseteq V$, then $\overline{\mathcal{R}}_I$ is naturally injected in $\overline{\mathcal{R}}_V$. By abuse of notation (supported by our ambiguous notation $\langle\rangle$ above), we shall say that $\overline{\mathcal{R}}_I$ is included in $\overline{\mathcal{R}}_V$.

If $r_1, r_2 \in \overline{\mathcal{R}}_V$ such that all the bindings of r_1 are also bindings of r_2, then we write $r_1 \leq r_2$. Relation $\leq$ is a partial order on $\overline{\mathcal{R}}_V$. We denote with $\cup$ the (partially defined) least upper bound operator associated with $\leq$ on $\overline{\mathcal{R}}_V$. The completely defined greatest lower bound operator is denoted $\cap$. If $r \in \overline{\mathcal{R}}_V$ and $I \subseteq V$, then we define $r \setminus I = \langle X \mapsto x \mid r(X) = x \; and \; X \notin I \rangle$.

8.1.2 State terms

Under the simplifying assumptions defined above, the state elements of an InteLus program are its `fby` statements. Each of them stores exactly one value—the last value of its input variable, if one exists, otherwise the initial constant of the `fby` statement.

The current state of a statement or program p can therefore be represented by another program obtained by replacing in p the constants of the `fby` statements with the values currently stored by the `fby` statements. If we consider the code fragment on the left:

```
y = add(x,x)          y = add(x,x)
z = 10 fby y          z = 6 fby y
```

then its initial state is the fragment itself. After one cycle where `x` has value 3, the new state is the fragment on the right.

Given a statement or program p, we denote with $\mathcal{S}_p$ the set of possible state terms over p. Elements of $\mathcal{S}_p$ are all identical upto a constant change in `fby` equations. The initial state of p is p itself.

8.1.3 Transitions and SOS rules

Consider a statement or program p, and assume that $\mathcal{I}$ is the set of variables taken as input by p, and $\mathcal{O}$ is the set of variables assigned by p. Then, transitions of p have the form $p' \overset{O}{\underset{I}{\Rightarrow}} p''$, where $p', p'' \in \mathcal{S}_p$, $I \in \mathcal{R}_{\mathcal{I}}$, and $O \in \mathcal{R}_{\mathcal{O}}$. Causal correctness requires that $\mathcal{I} \cap \mathcal{O} = \emptyset$. State term p' is called the input state, while p'' is the output state.

Such transitions are built using the structural operational semantics (SOS) rules of Figure 8.1 and Figure 8.2. The rules of Figure 8.1 provide the semantics of InteLus' subset without guards, which can be seen as a sub-set of Lustre. The rules of Figure 8.2 add support for equation guards.

The rules of Figure 8.1 are for the most part straightforward. Each operator (*e.g.* `when`, `merge`, function call) has one rule covering the case where it is not executed in a

$$\frac{x \neq \texttt{nil}}{Z = \texttt{merge}\ C\ X\ Y \xRightarrow[\langle C \mapsto \texttt{true}, X \mapsto x, Y \mapsto \texttt{nil}\rangle]{\langle Z \mapsto x\rangle} Z = \texttt{merge}\ C\ X\ Y} \quad \textbf{(merge+)}$$

$$\frac{y \neq \texttt{nil}}{Z = \texttt{merge}\ C\ X\ Y \xRightarrow[\langle C \mapsto \texttt{false}, X \mapsto \texttt{nil}, Y \mapsto y\rangle]{\langle Z \mapsto y\rangle} Z = \texttt{merge}\ C\ X\ Y} \quad \textbf{(merge-)}$$

$$Z = \texttt{merge}\ C\ X\ Y \xRightarrow[\langle C \mapsto \texttt{nil}, X \mapsto \texttt{nil}, Y \mapsto \texttt{nil}\rangle]{\langle Z \mapsto \texttt{nil}\rangle} Z = \texttt{merge}\ C\ X\ Y \quad \textbf{(merge-nil)}$$

$$\frac{x \neq \texttt{nil}}{Y = X\ \texttt{when}\ C \xRightarrow[\langle C \mapsto \texttt{true}, X \mapsto x\rangle]{\langle Y \mapsto x\rangle} Y = X\ \texttt{when}\ C} \quad \textbf{(when+)}$$

$$\frac{x \neq \texttt{nil}}{Y = X\ \texttt{when}\ C \xRightarrow[\langle C \mapsto \texttt{false}, X \mapsto x\rangle]{\langle Y \mapsto \texttt{nil}\rangle} Y = X\ \texttt{when}\ C} \quad \textbf{(when-)}$$

$$Y = X\ \texttt{when}\ C \xRightarrow[\langle C \mapsto \texttt{nil}, X \mapsto \texttt{nil}\rangle]{\langle Y \mapsto \texttt{nil}\rangle} Y = X\ \texttt{when}\ C \quad \textbf{(when-nil)}$$

$$\frac{(y_1, \ldots, y_m) = F_{fid}(x_1, \ldots, x_n)}{(Y_1, \ldots, Y_m) = fid(X_1, \ldots, X_n) \xRightarrow[\langle X_i \mapsto x_i | i = \overline{1,n}\rangle]{\langle Y_i \mapsto y_i | i = \overline{1,m}\rangle} (Y_1, \ldots, Y_m) = fid(X_1, \ldots, X_n)} \quad \textbf{(fun)}$$

$$(Y_1, \ldots, Y_l) = fid(X_1, \ldots, X_k) \xRightarrow[\langle X_i \mapsto \texttt{nil} | i = \overline{1,n}\rangle]{\langle Y_i \mapsto \texttt{nil} | i = \overline{1,m}\rangle} (Y_1, \ldots, Y_l) = fid(X_1, \ldots, X_k) \quad \textbf{(fun-nil)}$$

$$(Y) = k\ \texttt{fby}\ X \xRightarrow[\langle X \mapsto k'\rangle]{\langle Y \mapsto k\rangle} (Y) = k'\ \texttt{fby}\ X \quad \textbf{(fby)} \qquad (Y) = k\ \texttt{fby}\ X \xRightarrow[\langle X \mapsto \texttt{nil}\rangle]{\langle Y \mapsto \texttt{nil}\rangle} (Y) = k\ \texttt{fby}\ X \quad \textbf{(fby-nil)}$$

$$\frac{p_1 \ldots p_n \overset{O_1}{\underset{I_1}{\Rightarrow}} p'_1 \ldots p'_n \qquad p_{n+1} \overset{O_2}{\underset{I_2}{\Rightarrow}} p'_{n+1} \quad supp(O_1) \cap supp(O_2) = \emptyset \quad supp(O_2) \cap supp(I_1) = \emptyset}{p_1 \ldots p_n p_{n+1} \xRightarrow[I_1 \cup (I_2 \setminus supp(O_1))]{O_1 \cup O_2} p'_1 \ldots p'_n p'_{n+1}} \quad \textbf{(compose-acyclic)}$$

$$\frac{p_1 \ldots p_n \xRightarrow[I \cup \langle Y \mapsto v\rangle]{O \cup \langle X \mapsto v'\rangle} p'_1 \ldots p'_n \quad Y{=}k\ \texttt{fby}\ X \xRightarrow[\langle X \mapsto v'\rangle]{\langle Y \mapsto v\rangle} Y{=}k'\ \texttt{fby}\ X}{p_1 \ldots p_n \ \ Y{=}k\ \texttt{fby}\ X \xRightarrow[I]{O \cup \langle Y \mapsto v, X \mapsto v'\rangle} p'_1 \ldots p'_n \ \ Y{=}k'\ \texttt{fby}\ X} \quad \textbf{(compose-fby)}$$

$$\frac{p_1 \ldots p_n \overset{O}{\underset{I}{\Rightarrow}} p'_1 \ldots p'_n \qquad i \text{ permutation of } 1..n}{p_{i_1} \ldots p_{i_n} \overset{O}{\underset{I}{\Rightarrow}} p'_{i_1} \ldots p'_{i_n}} \quad \textbf{(permute)}$$

$$\frac{body(p) \overset{O}{\underset{I}{\Rightarrow}} body(p') \qquad \overline{I \cup O} \text{ maximal in the sense of } \leq}{p \overset{O}{\underset{I}{\Rightarrow}} p'} \quad \textbf{(program-closure)}$$

Figure 8.1: Operational semantics of InteLus' subset without guards

cycle (the **-nil* rules, which simply propagate `nil` values from inputs to outputs), and one or two rules covering the case where the operator is executed. Each rule implicitly requires that input and output variable sets are exclusive.

The following three rules (*(compose-acyclic)*, *(compose-fby)*, and *(permute)*) define the semantics of sequences of equations. The two rules *(compose-*)* allow adding one more equation to a sequence for which transitions have already been determined. To simplify the rules, the new equation is always added at the end, and composition must satisfy structural constraints (the new equation is either a `fby` equation, or one whose output is not used in the first term). These two equations allow defining the semantics of sequences of equations that have been topologically ordered to satisfy intra-cycle data dependencies. To allow the definition of the semantics when equation ordering does not satisfy this condition, rule *(permute)*) has been introduced.

Rule *(program-closure)* defines the semantics of programs from that of the program body, which is the sequence of statements of the program. Transitions of the program are maximal transitions of the body. To understand the importance of this rule, often overlooked in existing semantics, consider the following input-less program, which encodes a counter that doubles `x` at each cycle:

```
fun add:(int,int)->(int)
var x:int y:int
let
  x = 1 fby y
  y = add(x,x)
tel
```

Semantic rules *(fun-nil)* and *(compose-fby)* allow the construction of a program body transition where both variables are assigned value `nil`. However, rule *(program-closure)* forbids it, forcing the counter to advance at each execution cycle, which is the semantics implemented by existing Lustre compilers such as Heptagon.

8.1.4 Determinism, traces and correctness

The semantics is deterministic. For given initial state of a statement or program p and for given input valuation I, at most one transition can be built using the rules of Figure 8.1 and Figure 8.2. This is due to the fact that each variable is assigned by exactly one equation, and each equation is deterministic.

Given a program p, an execution trace is any sequence of transitions starting in the initial state:

$$p \xRightarrow[I_1]{O_1} p_1 \xRightarrow[I_2]{O_2} \dots \xRightarrow[I_n]{O_n} p_n$$

We denote the set of all traces of a statement or program p with $Traces(p)$.

We say that a program is correct if a trace exists for every sequence $I_1, \dots, I_n$ assigning values different from `nil` to every input variable. In this thesis, we study the implementation of correct programs. We are not interested in checking the correctness of synchronous programs, as extensive previous work exists on the subject [100, 101].

Given the determinism of the synchronous semantics, a trace is fully identified by its sequence of input events. Thus, $Traces(p)$ can be seen as a sub-set of $\mathcal{R}_V^*$, where V is the set of variables of p.

Definition 1 (Program traces)**.** *Let p be a program over variable set $\mathcal{V}$ and input variable set $\mathcal{I} \subseteq \mathcal{V}$. The set $Traces(p)$ of traces of p is the subset of $\mathcal{R}_{\mathcal{V}}{}^*$ formed of sequences $r_1 \dots r_k$ such that there exist $p_1, \dots, p_k$ with*

$$p \xrightarrow[r_1|_{\mathcal{I}}]{r_1 \backslash \mathcal{I}} p_1 \xrightarrow[r_2|_{\mathcal{I}}]{r_2 \backslash \mathcal{I}} p_2 \dots \xrightarrow[r_k|_{\mathcal{I}}]{r_k \backslash \mathcal{I}} p_k$$

8.1.5 Equation guards

All synchronous languages introduce the notion of *logical clock*. Clocks represent sets of execution cycles, and can be seen as *predicates over the state and inputs of the program*, largely used for analysis purposes.

By comparison, guards are a representation of the operational mechanisms used to represent the control flow (triggering, sequencing, synchronization) in our implementation models. The semantics of the full InteLus language (with guards) is provided as an extension of that of Figure 8.1 with the new rules of Figure 8.2. Existing rules remain unchanged.

Under the extended semantics, if an equation has a guard, and if the trigger is present in a cycle, then the decision to execute the equation is taken by the evaluation of (present) guard variables (if any), and not by the occurrence (or not) of a present/absent variable configuration (as it was the case in the semantics without guards). In particular, when the trigger is `top`, no further absence decision is needed in the execution of the equation.

Guards allow the formalization *at dataflow level* of a process performed by all synchronous language compilers – that of assigning triggering conditions to all equations that produce executable code. In particular, when all equations of a program have a guard, rule **(program-closure)** is no longer needed, meaning that no operational mechanism must be synthesized by the compiler to detect the maximality of program transitions (which can be difficult in multi-threaded contexts).

It is now clear why we required that all equations of a thread share the same guard trigger. Indeed, having two equations with different triggers sequenced inside a thread would require making an absence decision (to give control to the second equation when the first is not executed). This is not possible in general in an asynchronous environment.

Note that, according to the extended semantics, different guards may represent the same logical activation condition (the same clock), but a different causality. Consider the following code fragment:

```
top            b = merge a b1 b2
top            c = and(a,b)
top on a on b1 x = f(y) //does not depend on b2
top on c       z = f(y) //depends on b2
```

$$\frac{p \overset{O}{\underset{I}{\Rightarrow}} p' \qquad \overline{I \cup O} \neq \langle\rangle \qquad \forall i,j : S_i, T_j \notin supp(I \cup O) \qquad \forall i,j : S_i \neq T_j}{\texttt{wait}(S_1,\ldots,S_k)\ \texttt{done}(T_1,\ldots,T_l)\ p \xrightarrow[I \cup \langle S_i \mapsto \texttt{top} | i=\overline{1,k}\rangle]{O \cup \langle T_i \mapsto \texttt{top} | i=\overline{1,l}\rangle} \texttt{wait}(S_1,\ldots,S_k)\ \texttt{done}(T_1,\ldots,T_l)\ p'} \textbf{(wait-done)}$$

$$\frac{p \overset{O}{\underset{I}{\Rightarrow}} p \qquad \overline{I \cup O} = \langle\rangle \qquad \forall i,j : S_i, T_j \notin supp(I \cup O) \qquad \forall i,j : S_i \neq T_j}{\texttt{wait}(S_1,\ldots,S_k)\ \texttt{done}(T_1,\ldots,T_l)\ p \xrightarrow[I \cup \langle S_i \mapsto \texttt{nil} | i=\overline{1,k}\rangle]{O \cup \langle T_i \mapsto \texttt{nil} | i=\overline{1,l}\rangle} \texttt{wait}(S_1,\ldots,S_k)\ \texttt{done}(T_1,\ldots,T_l)\ p} \textbf{(wait-done-nil)}$$

$$\frac{p \overset{O}{\underset{I}{\Rightarrow}} p' \quad \overline{I \cup O} \neq \langle\rangle \quad \forall i,j : C, s_i, t_j \notin supp(I \cup O) \quad \forall i,j : C \neq s_i \neq t_j \neq C}{r(p) \xrightarrow[I \cup \langle C \mapsto \texttt{true}\rangle \cup \langle t_i \mapsto \texttt{top} | i=\overline{1,l}\rangle]{O \cup \langle s_i \mapsto \texttt{top} | i=\overline{1,k}\rangle} r(p')} \textbf{(on+)}$$

$$\frac{p \overset{O}{\underset{I}{\Rightarrow}} p \quad \overline{I \cup O} = \langle\rangle \quad \forall i,j : C, s_i, t_j \notin supp(I \cup O) \quad \forall i,j : C \neq s_i \neq t_j \neq C}{r(p) \xrightarrow[I \cup \langle C \mapsto \texttt{false}\rangle \cup \langle t_i \mapsto \texttt{top} | i=\overline{1,l}\rangle]{O \cup \langle s_i \mapsto \texttt{top} | i=\overline{1,k}\rangle} r(p)} \textbf{(on-)}$$

$$\frac{p \overset{O}{\underset{I}{\Rightarrow}} p \quad \overline{I \cup O} = \langle\rangle \quad \forall i,j : C, s_i, t_j \notin supp(I \cup O) \quad \forall i,j : C \neq s_i \neq t_j \neq C}{r(p) \xrightarrow[I \cup \langle C \mapsto \texttt{nil}\rangle \cup \langle t_i \mapsto \texttt{nil} | i=\overline{1,l}\rangle]{O \cup \langle s_i \mapsto \texttt{nil} | i=\overline{1,k}\rangle} r(p)} \textbf{(on-nil)}$$

$$\frac{p \overset{O}{\underset{I}{\Rightarrow}} p' \quad \overline{I \cup O} \neq \langle\rangle}{s\ p \xrightarrow[I \cup \langle s \mapsto \texttt{top}\rangle]{O} s\ p'} \textbf{(trig-comp)} \qquad \frac{p \overset{O}{\underset{I}{\Rightarrow}} p \quad \overline{I \cup O} = \langle\rangle}{s\ p \xrightarrow[I \cup \langle s \mapsto \texttt{nil}\rangle]{O} s\ p} \textbf{(trig-comp-nil)} \qquad \frac{p \overset{O}{\underset{I}{\Rightarrow}} p' \quad \overline{I \cup O} \neq \langle\rangle}{\texttt{top}\ p \overset{O}{\underset{I}{\Rightarrow}} \texttt{top}\ p'} \textbf{(trig-top)}$$

Figure 8.2: Operational semantics of guards. To facilitate rule writing, we denote $r(p)$="$\texttt{wait}(t_1,\ldots,t_l)\ \texttt{done}(s_1,\ldots,s_k)\ \texttt{on}\ C\ p$"

Here, `x` and `z` have the same values at the same cycles, but their computations do not carry the same data dependencies due to the way the guards are computed.

8.2 Kahnian asynchronous interpretation

The determinism of the InteLus data-flow equations means that we can endow InteLus programs with *asynchronous* Kahn process networks (KPN) [102] semantics. This allows reasoning about synchronization in an asynchronous execution environment, but without having to consider resource allocation. The asynchronous interpretation allows reasoning about asynchronous implementability and allows us to define (in Section 8.3) a set of properties allowing implementation on the target execution platforms. A strong semantics preservation property, defined in Section 8.2.3, links the asynchronous interpretation of an InteLus program to its synchronous interpretation. Furthermore, the small-step operational form of the kahnian semantics facilitates the link with the platform semantics of Chapter 9, facilitating the definition of implementation correctness.

8.2.1 Notations

In our semantics, program equations are deterministic Kahn processes communicating through infinite lossless FIFO channels corresponding to the variables. The semantics is asynchronous. *Absence of a variable value cannot be reacted upon (only presence).* Execution traces are represented with *histories* assigning sequences of values different from `nil` to each variable identifier. Given a set of variables V, we denote with $Hist(V)$ the set of finite histories h assigning to each $v \in V$ a sequence of values $h(v) \in Type(v)^*$. On $Hist(V)$ we introduce the concatenation operation, defined pointwise by $(h_1 h_2)(v) = h_1(v)h_2(v)$ for all $v \in V$. Operator $len(x)$ provides the length of a word $x \in Type(v)$.

The asynchronous observation of a synchronous trace discards synchronization, retaining for each variable the sequence of values different from `nil`. It is defined as $\delta : {\mathcal{R}_V}^* \to Hist(V)$ where $\delta(t_1 t_2) = \delta(t_1)\delta(t_2)$ for all $t_1, t_2 \in {\mathcal{R}_V}^*$ and $\delta(r)(v) = r(v)$ if $r \in \mathcal{R}_V$ with $r(v) \neq$ `nil` and $\delta(r)(v) = \epsilon$ (the empty word), if $r(v) =$ `nil`.

8.2.2 Program state

In the classical definition of Kahn process networks [102], system state is fully determined by the state of the processes and the state of the communication channels. The state of each communication channel can be represented with the sequence of messages produced, but not yet consumed on that channel.

In InteLus, under asynchronous semantics, equations are stateless. This is true even for the `fby` equation, whose internal state is naturally conflated with that of its output variable. Another difference is that point-to-point channels are replaced in InteLus with variables that can be read by multiple equations. To accomodate these changes, we represent states of a program or statement p as pairs $\ll p', h \gg$ where:

- $h \in Hist(V)$, where V is the set of variables of p. It gives the finite (possibly empty) sequence of values different from `nil` that were assigned to each variable since the beginning of the execution. If v is the output of a `fby` equation, $h(v)$ also contains, on the last position, the value currently stored by the `fby` (the synchronous state).
- p' is an annotated term over p. The terms are based on those of the synchronous semantics. In addition, for each equation eq that reads a variable v, the use of v inside the equation is annotated with a non-negative integer defining the position in $h(v)$ that the equation will read at its next execution. Indices in $h(v)$ start at 0.

Note that there is some redundancy between the `fby` constants in the term and the same values stored in the history of the `fby` output variables. We keep both for consistency with the synchronous semantics. The initial state of a program/statement p is $\ll p_0, h_0 \gg$, where the p_0 is obtained from p by annotating each variable read point with 0. If v is defined by equation `v = k fby y` then $h_0(v) =$ `k`. For all other variables $h_0(v) = \epsilon$.

We say that a state $\ll p, h \gg$ is complete if for every variable v defined by `v = k fby y` all read pointers are equal to $len(h(v)) - 1$, and for all other variable w the read pointers

are equal to $len(h(w))$. Complete states can be put in direct relation with states of the synchronous semantics. We denote with $strip(\ll p, h \gg)$ the synchronous state term obtained from $\ll p, h \gg$ by removing the history h and the read point annotations.

8.2.3 Semantic rules and semantics preservation

Semantics is presented in Figure 8.3, under SOS form. Concurrency is by interleaving. At each transition, exactly one equation is executed among those that can be executed (cf. rule *(interleave)*). The rules for guards *(on*)* and *(trig-*)* are optional. Without them, the semantics works for programs without guards. With the rules, equations can have guards. Input variables are not considered in the semantics. It is assumed that each input is read by exactly one equation, and that a fresh value is provided at each execution of that equation.

$$\frac{\forall i \in \overline{1,m} : \boldsymbol{adv(h, Y_i, k)} \qquad (x_1, \ldots, x_n) = F_{fid}(h_k(Y_1), \ldots, h_k(Y_m))}{\ll (X_1, \ldots, X_n) = fid(Y_1^k, \ldots, Y_m^k), h \gg \rightarrow \ll (X_1, \ldots, X_n) = fid(Y_1^{k+1}, \ldots, Y_m^{k+1}), h\delta(\langle X_i \mapsto x_i \mid i = \overline{1,n} \rangle) \gg} \textbf{(fcall)}$$

$$\frac{\boldsymbol{adv(h, Y, n)}}{\ll X = k \texttt{ fby } Y^n, h \gg \rightarrow \ll X = h_n(Y) \texttt{ fby } Y^{n+1}, h\delta(\langle X \mapsto h_n(Y) \rangle) \gg} \textbf{(fby)}$$

$$\frac{\boldsymbol{adv(h, C, m)} \qquad h_m(C) = \texttt{false} \qquad \boldsymbol{adv(h, X, m)}}{\ll Y = X^m \texttt{ when } C^m, h \gg \rightarrow \ll Y = X^{m+1} \texttt{ when } C^{m+1}, h \gg} \textbf{(when-)}$$

$$\frac{\boldsymbol{adv(h, C, m)} \qquad h_m(C) = \texttt{true} \qquad \boldsymbol{adv(h, X, m)}}{\ll Y = X^m \texttt{ when } C^m, h \gg \rightarrow \ll Y = X^{m+1} \texttt{ when } C^{m+1}, h\delta(\langle Y \mapsto h_m(X) \rangle) \gg} \textbf{(when+)}$$

$$\frac{\boldsymbol{adv(h, C, m)} \qquad h_m(C) = \texttt{true} \qquad \boldsymbol{adv(h, X, n)}}{\ll Z = \texttt{merge } C^m \ X^n \ Y^p, h \gg \rightarrow \ll Z = \texttt{merge } C^{m+1} \ X^{n+1} \ Y^p, h\delta(\langle Z \mapsto h_n(X) \rangle) \gg} \textbf{(merge+)}$$

$$\frac{\boldsymbol{adv(h, C, m)} \qquad h_m(C) = \texttt{false} \qquad \boldsymbol{adv(h, Y, p)}}{\ll Z = \texttt{merge } C^m \ X^n \ Y^p, h \gg \rightarrow \ll Z = \texttt{merge } C^{m+1} \ X^n \ Y^{p+1}, h\delta(\langle Z \mapsto h_p(Y) \rangle) \gg} \textbf{(merge-)}$$

$$\frac{\ll p, h \gg \rightarrow \ll p', h\delta(x) \gg \qquad \forall i : \boldsymbol{adv(h, S_i, k)} \qquad \forall i, j : (x(S_i) = x(T_j) = \texttt{nil}) \wedge (S_i \neq T_j)}{\ll r(p, k), h \gg \rightarrow \ll r(p', k+1), h\delta(x \cup \langle T_i \mapsto \texttt{top} \mid i = \overline{1,n} \rangle) \gg} \textbf{(wait-done)}$$

$$\frac{\boldsymbol{adv(h, C, k)} \quad h_k(C) = \texttt{true} \quad \ll eq, h \gg \rightarrow \ll eq', h\delta(r) \gg}{\ll \texttt{on } C^k \ eq, h \gg \rightarrow \ll \texttt{on } C^{k+1} \ eq', h\delta(r) \gg} \textbf{(on+)} \qquad \frac{\ll eq, h \gg \rightarrow \ll eq', h' \gg}{\ll \texttt{top } eq, h \gg \rightarrow \ll \texttt{top } eq', h' \gg} \textbf{(trig-top)}$$

$$\frac{\boldsymbol{adv(h, C, k)} \quad h_k(C) = \texttt{false}}{\ll \texttt{on } C^k \ eq, h \gg \rightarrow \ll \texttt{on } C^{k+1} \ eq, h \gg} \textbf{(on-)} \qquad \frac{\boldsymbol{adv(h, S, k)} \quad \ll eq, h \gg \rightarrow \ll eq', h' \gg}{\ll S^k \ eq, h \gg \rightarrow \ll S^{k+1} \ eq', h' \gg} \textbf{(trig-comp)}$$

$$\frac{\ll eq_i, h \gg \rightarrow \ll eq_i', h' \gg}{\ll eq_1, \ldots, eq_i, \ldots, eq_n, h \gg \rightarrow \ll eq_1, \ldots, eq_i', \ldots, eq_n, h' \gg} \textbf{(interleave)}$$

Figure 8.3: Kahn process network semantics. Predicate $\boldsymbol{adv(h, X, k)}$ is defined as $len(h(X)) > k$. Term $r(p, k)$ is defined as $\texttt{wait}(S_1^k, \ldots, S_m^k)$ $\texttt{done}(T_1, \ldots, T_n)$ p

Predicate $adv(h, v, n)$ determines whether we can read the n^{th} value of variable (v) in history h. It is used to determine if enough input is available to enable the execution of a transition.

The synchronous and Kahnian asynchronous semantics of InteLus are tightly related:

Theorem 1 (Semantics preservation)**.** *Let p be an InteLus program that is correct under synchronous semantics. Then:*

- *For any $r_1 \ldots r_k \in Traces(p)$ there exists an asynchronous trace $\ll p_0, h_0 \gg \rightarrow \ldots \rightarrow$*

$\ll p_n, h_n \gg$ ending in a complete state $\ll p_n, h_n \gg$ such that:

$$h_n = \delta(r_1 \dots r_k)\delta(r)$$

where $r(v) = \epsilon$ if v is not the output of a `fby`*, and $r(v) = k$ if k is the state of the* `fby` *defining v in p_n. When this happens, the final state of the synchronous trace is strip($\ll p_n, h_n \gg$).*

- *For any asynchronous trace $\ll p_1, h_1 \gg \rightarrow \dots \rightarrow \ll p_m, h_m \gg$ of p there exists an extension $\ll p_1, h_1 \gg \rightarrow \dots \rightarrow \ll p_n, h_n \gg$ ($n \geq m$) such that $\ll p_n, h_n \gg$ is a complete state, and there exists $r_1 \dots r_k \in Traces(p)$ satisfying property (1) above.*

While we do not have a full proof for this theorem, we sketch one relying on previous work.

Proof sketch: Direct application of Theorem 4.4 of [103]. The tedious part of the proof would consist of interpreting equations of p as micro-step state transition systems (μSTS) and proving that the synchronous (resp. asynchronous) composition of the μSTSs associated with equations faithfully represents the synchronous (resp. asynchronous) semantics of p. Once this is done, Theorem 4.4 can be applied, after noting that the synchronous correctness of the program ensures that the asynchronous composition of μSTSs is non-blocking.□

8.3 Properties ensuring implementability

For an implementation model to support a correct C implementation under the assumptions defined above (execution in bounded memory, code generation through "pretty printing", without further synthesis of synchronization) its dataflow part must satisfy a number of properties that are naturally expressed under Kahnian semantics. These properties are also amenable to low-complexity verification and synthesis based on sufficient structural properties of the program.

8.3.1 Boundedness

When the program of Figure 7.2 is executed under Kahnian semantics, function `f` may be executed an indefinite number of times before `h` is executed once. Such a behavior requires infinite storage for values of `x`, and therefore is not amenable to static resource allocation.

The synchronization of the implementation, enforced using `event` type variables, must prohibit such unbounded behaviors. We require that our implementation models (like those in Figures 7.3 and 7.4) ensure that no variable requires the storage of more than one value at a time. Formally:

Definition 2 (1-boundedness)**.** *An InteLus program p is called 1-bounded if in any reachable state $\ll p', h' \gg$ of the Kahnian semantics, for every variable v that is not an input and for every read point annotation x of v in p, we have: $len(h'(v)) - x \leq 1$.*

Ensuring 1-boundedness can be done using various sufficient properties. We present here one that has the advantage of simplicity. We say that equation eq_2 depends on eq_1 through variable v, written $eq_1 \overset{v}{\dashrightarrow} eq_2$, whenever eq_1 produces v and eq_2 consumes it.[1] Then, a program p is 1-bounded if whenever $eq_1 \overset{v_1}{\dashrightarrow} eq_2$, there exists a dependency cycle $eq_1 \overset{v_1}{\dashrightarrow} \ldots \overset{v_{k-1}}{\dashrightarrow} eq_k \overset{v_k}{\dashrightarrow} eq_1$ such that $v_2 \ldots v_k$ are always present in cycles where v_1 is present and such that exactly one of the equations $eq_1 \ldots eq_k$ is a `fby` or `pre` equation.

In the implementation models of Figure 7.3 and Figure 7.4, such dependency cycles exist for every equation.

8.3.2 Implementation of `fby` with no internal memory

One key advantage of the implementation models of Figure 7.3 and Figure 7.4 is that data memory allocation can be fully described through annotations of the dataflow variables. This is not possible if we follow the traditional compilation of Lustre to sequential code, where each `fby` equation requires one C variable (different from those associated to input and output) to represent its state. If we require that delays are always scheduled at the end of the cycle, the internal variable is no longer required, but as we saw in Section 5.4.2, the input and output variables still must be different, which reduces the memory optimization potential.

To preserve the simplicity of the allocation annotations and support memory allocation optimizations similar to those of [52], we require that scheduling constraints (through `wait/done` dependencies) ensure that no extra C variable is needed for `fby` states, and that the input and output variables of a `fby` can share the same C variable. When this property holds, no code is needed in the C threads for `fby` equations.

A simple sufficient condition ensuring this property is the following: For every `fby` equation of input variable v and output variable v', if eq is the equation producing v and eq' an equation reading v', then eq' and eq are ordered by a dependency chain containing no `fby` equations $eq' \overset{v_1}{\dashrightarrow} \ldots \overset{v_{k-1}}{\dashrightarrow} eq_k \overset{v_k}{\dashrightarrow} eq$. This sufficient condition is powerful enough to ensure that `fby` equations in Figure 7.3 and Figure 7.4 can use the simplified code generation defined in Section 5.4.2.

8.3.3 Explicit synchronization

Under data-flow semantics (synchronous or Kahnian), data accesses are synchronizing. Under machine semantics (defined in Chapter 9) they are not, like regular memory accesses. Assume that a variable v of type different from `event` is used for communication between two equations eq_1 and eq_2 that will be mapped to different threads. Then, variables of type `event`, later mapped to semaphore operations, must be used to represent the synchronization associated with these data accesses.

The completeness of the synchronization can be checked on the data-flow part of an implementation model, which is an InteLus program. This is done by checking that the Kahnian semantic transitions of eq_2 do not need to perform the synchronization

[1] Through either the equations themselves, or through their guards.

tests on v (the *adv* conditions in Figure 8.3) for any variable v with type different from `event`.

In Figure 7.3, for instance, the production of `y` on `cpu1` (in line 34) and its consumption on `cpu0` (in line 26) are synchronized using variables `t2`, `t3`, `v`, `s3`, and `s4`, corresponding to semaphore operations and to sequencing of equations inside threads.

Chapter 9

Machine semantics and Correctness formalization

The two semantics defined in the previous chapter provide a purely functional, dataflow view of an InteLus program. In this chapter, we introduce the machine semantics of the language, which provides an operational description of the execution of the application on the execution platform, potentially subject to functional interferences between cores due to resource allocation, and only synchronized by mutex operations and by the sequencing of operations inside threads.

The machine semantics only covers the cyclic execution of threads once the initialization code has been executed. It is not meant to be a semantics for general multi-threaded implementations. Instead, it only covers the very restricted control structures of our implementation models.

Furthermore, when compared to classical cycle-accurate simulation semantics, it already includes elements of abstraction meant to:

1. Hide timing aspects. This form of abstraction was also employed in the concrete semantics of [10].

2. Semantically isolate the well-formed code described by the implementation models from the potentially more general C code of the external functions (called by implementation models), by means of a clearly-defined interface.

This isolation means that we cannot know the exact position of control (the program counter) during execution of an external function, nor the exact way the function manipulates data during computations.

The machine semantics presented here covers only data-flow programs where the threads self-activate at the end of each cycle.

9.1 State representation

In the machine semantics, states provide an abstract view of the state of the execution platform components, including CPUs, memory hierarchy, and lock status. They have

the structure presented in Figure 9.1. For clarity we use an OCaML-like syntax for the definition of the state.

```
type mach_state = Err | Norm of ctrl_state * mem_state * lock_state
type mem_state = addr -> mem_loc_state
type mem_loc_state = W | R of posint * loc_value
type loc_value = { ram : value; cache : processor -> value; }
type value = D of word | U
type lock_state = lock -> bool
```

Figure 9.1: Representation of the machine state (in OCaML syntax)

The remainder of this section details the various components of the machine state and defines state manipulation functions used by the semantics. It is important to remember that the form of the machine state depends on the hardware architecture of the platform, but also on the chosen application structure and execution model. Indeed, we are defining a execution model for applications with specific structure, not a general execution model for multi-threaded running on the architecture.

A platform can be in normal execution state `Norm`, where its three internal components are visible, or in an error state `Err`. In our semantics, no error recovery mechanism exists. Once reached, the error state cannot be exited.

9.1.1 Control flow state

The control flow state is represented through annotations of the implementation program. Each thread/core has exactly one program counter, represented with a red bullet (•) placed in its body. To allow the identification of all possible memory access or synchronization configurations (under the abstraction defined above) bullets can take the following positions:

- Between guarded equations of the thread, to represent the state where the execution of one equation is finished, but the next (including possible guard tests) has not yet started. When control loops back after the execution of the last equation of a thread, the bullet is first placed after the last equation, and then before the first equation of the thread.

- Before a guard test "`on C`", to represent the state where the execution has reached, but not executed, the associated variable test.

- After all guard and synchronization annotations, to represent the state where the guard has been successfully traversed, but the equation execution has not started.

- Inside the equation, to represent the state where its execution has started, but is not completed. When inside an equation, the token is annotated with the memory locations corresponding to variables in use by the equation execution (read or

written). These values are annotated by the transition entering the equation, and removed by the equation exit transition.

In the initial state, in each thread, the bullet is placed before the first equation. The set of all possible such state representations obtained by bullet annotations is denoted `ctrl_state`.

9.1.2 Memory system state

The memory state includes the state of the RAM and that of the L1 data caches of the CPUs. We denote with `addr` the set of memory addresses that can be used by data-flow variables. At each instant, a memory location can be either written by one equation (as an output variable), or read by zero or more equations, as an input variable. Performing a read access on a variable that is currently written by another equation, or a write access on a variable that is currently written or read by another equation is an error (leads to the `Err` state). In the definition of type `mem_loc_state`, variant `W` corresponds to the write state, and variant `R` corresponds to the read state.

The read variant has two arguments:

- A positive integer giving the non-negative number of readers.
- A data structure storing the values that can be obtained when reading the location.

The value of a memory location can be undefined (`U`) or defined (`D`) with a `word` value. In a given state, read accesses from different processors to the same memory location may produce different values. To allow reasoning on this memory consistency issue, we need to store, for each memory location, not one, but $proc+1$ values, where $proc$ is the number of CPU cores. Type `loc_value` structures the storage of these $proc+1$ values. In a value `l` of type `loc_value`, the value stored in `l.ram` is that stored in RAM. The value stored in `l.cache(p)` is the one that an access from CPU core p would return.

The initial memory state is determined by the initial value of `fby` equations of the program: All locations are in state `(R (0,v))`, where `v.cache(p)=U` for all `p`. Memory locations not corresponding to `fby` output variables also have have `v.ram=U`. For the output of a `fby` equation initialized with `d`, the initial state sets `v.ram=(D d)`.

Semantics rules update the memory state using four functions whose signatures are given below. The first two correspond to the `dcache_inval` and `dcache_flush` API calls. The remaining two implement the semantic actions taken upon entering and exiting an equation.

```
dinval: mem_state*processor*addr -> mem_state
dflush: mem_state*processor*addr -> mem_state
enter : mem_state*processor*(addr list)*(addr list) -> mem_state
exit  : mem_state*processor*((addr*word) list)*(addr list) -> mem_state
```

We call `inval`$(M, p, addr)$, respectively `flush`$(M, p, addr)$, upon execution of the corresponding API primitive in memory state M, on processor p, and with argument $addr$. The functions can only be applied when $M(addr) =$ (`R` (n, v)). Their action

changes v as follows: $\texttt{inval}(m, p, addr)$ sets $v.cache(p)$ to $v.ram$; $\texttt{flush}(M, p, addr)$ sets $v.ram$ to $v.cache(p)$ and $v.cache(q)$ to $\texttt{U}$ for all $\texttt{q}{\neq}\texttt{p}$.

We denote with $[X]$ the memory location of variable X.
Function $\texttt{enter}(M, p, \{X_1, \ldots, X_n\}, \{Y_1, \ldots, Y_m\})$ is called when starting the execution of an equation that reads variables $Y_1, \ldots, Y_m$ and writes variables $X_1, \ldots, X_n$. The call sets the memory state of the locations $[X_i]$ to $\texttt{W}$, and increments the read counter of the locations $[Y_i]$ whenever $[Y_i]$ is not also the location of an output variable X_j.

Function $\texttt{exit}(M, p, \{(X_1, x_1), \ldots, (X_n, x_n)\}, \{Y_1, \ldots, Y_m\})$ is called when completing the execution of an equation that reads variables $Y_1, \ldots, Y_m$ and writes variables $X_1, \ldots, X_n$, when the final values for the written variables are respectively $x_i, i = \overline{1, n}$. The call decrements the read counter of the locations $[Y_i]$ whenever $[Y_i]$ is not also the location of an output variable. For each of the locations $[X_i]$, it sets its state to $(\texttt{R}\ (0, v))$ where $v.cache(p) = (\texttt{D}\ x_i)$, $v.cache(q) = \texttt{U}$ for all $q \neq p$, and $v.ram = \texttt{U}$.

Platform semantics assumes that input variables satisfy some simplifying assumptions. Each input variable is given a memory location different from all other variables, which is read by only one equation, and its value is assumed ready upon reading.

9.1.3 Lock state

The set of all locks is denoted with `lock`. Their state is represented with Boolean values. We update `lock_state` objects using two functions corresponding to the `lock` and `unlock` API calls:

```
unlock: lock_state*lock -> lock_state
lock  : lock_state*lock -> lock_state
```

Function $\texttt{unlock}(L, l)$ can be called only when $L(l) = \texttt{false}$. It sets $L(l)$ to `true`. Conversely, $\texttt{lock}(L, l)$ can be called only when $L(l) = \texttt{true}$. It sets $L(l)$ to `false`.

9.2 Semantic rules

Platform semantics is an interleaving semantics provided under SOS form. Transitions have the form $s_1 \underset{pr}{\longrightarrow} s_2$, where s_1 and s_2 are objects of type `mach_state` and pr is the processor performing the state transition. Figure 9.2 provides the SOS rules grouped in four categories: the API calls, dataflow equations, guards, and sequential and parallel composition. States $s_i = \texttt{Norm}(ctrl, mem, locks)$ are represented here as $ctrl, mem, locks$. When the program counter ($\bullet$) does not directly appear in the rules, we color in red the term containing it.

9.3 Correctness formalization

Assume that p is a functional specification—an InteLus program without non-functional annotations—and that q is a full-fledged implementation model. Stating that q is an

$$\frac{L([s]) = \texttt{true}}{\bullet[lock : [s]]eq, M, L \underset{p}{\to} [lock : [s]]eq\bullet, M, lock(L, [s])}\ \textbf{(lock)} \qquad \frac{L([s]) = \texttt{false}}{\bullet[unlock : [s]]eq, M, L \underset{p}{\to} [unlock : [s]]eq\bullet, M, unlock(L, [s])}\ \textbf{(unlock)}$$

$$\frac{L([s]) = \texttt{true}}{\bullet[unlock : [s]]eq, M, L \underset{p}{\to} Err}\ \textbf{(unlock-err)} \qquad \bullet[inval : adr]\ eq, M, L \underset{p}{\to} [inval : adr]\ eq\bullet, inval(M, p, adr), L\ \textbf{(inval)}$$

$$\bullet[flush : adr]\ eq, M, L \underset{p}{\to} [flush : adr]\ eq\bullet, flush(M, p, adr), L\ \textbf{(flush)}$$

$$\frac{M([C]) = (R\ (\_\ , (D\ \texttt{true}))) \quad M([Y]) = (R\ (\_\ , (D\ y))) \quad M([X]) = (R\ (0, \_))}{\bullet\ (X) = Y\ \texttt{when}\ C, M, L \underset{p}{\to} (X) = \bullet^{X=y}_{Y,C}\ Y\ \texttt{when}\ C, enter(M, p, \{X\}, \{C, Y\}), L}\ \textbf{(when+)}$$

$$\frac{M([C]) = (R\ (\_\ , (D\ \texttt{false})))}{\bullet\ (X) = Y\ \texttt{when}\ C, M, L \underset{p}{\to} (X) = \bullet_C\ Y\ \texttt{when}\ C\ , enter(M, p, \{\}, \{C\}), L}\ \textbf{(when-)}$$

$$\frac{\textit{conditions of}\ \textbf{(when+)}\ \textit{or}\ \textbf{(when-)}\ \textit{not satisfied}}{\bullet\ (X) = Y\ \texttt{when}\ C, M, L \underset{p}{\to} Err}\ \textbf{(when-err)}$$

$$\frac{M([C]) = (R\ (\_\ , (D\ \texttt{true}))) \quad M([Y_1]) = (R\ (\_\ , (D\ y))) \quad M([X]) = (R\ (0, \_))}{\bullet\ (X) = \texttt{merge}\ C\ Y_1\ Y_2, M, L \underset{p}{\to} (X) = \bullet^{X=y}_{C,Y_1}\ \texttt{merge}\ C\ Y_1\ Y_2, enter(M, p, \{X\}, \{C, Y_1\}), L}\ \textbf{(merge+)}$$

$$\frac{M([C]) = (R\ (\_\ , (D\ \texttt{false}))) \quad M([Y_2]) = (R\ (\_\ , (D\ y))) \quad M([X]) = (R\ (0, \_))}{\bullet\ (X) = \texttt{merge}\ C\ Y_1\ Y_2, M, L \underset{p}{\to} (X) = \bullet^{X=y}_{C,Y_2}\ \texttt{merge}\ C\ Y_1\ Y_2, enter(M, p, \{X\}, \{C, Y_2\}), L}\ \textbf{(merge-)}$$

$$\frac{\textit{conditions of}\ \textbf{(merge+)}\ \textit{or}\ \textbf{(merge-)}\ \textit{not satisfied}}{\bullet\ (X) = \texttt{merge}\ C\ Y_1\ Y_2, M, L \underset{p}{\to} Err}\ \textbf{(merge-err)} \qquad \frac{\textit{conditions of}\ \textbf{(fcall)}\ \textit{not satisfied}}{\bullet\ (X_1, \ldots, X_n) = fid(Y_1, \ldots, Y_m), M, L \underset{p}{\to} Err}\ \textbf{(fcall-err)}$$

$$\frac{\forall i : M([Y_i]) = (R\ (\_\ , (D\ y_i))) \quad \forall i : M([X_i]) = (R\ (0, \_)) \quad (x_1, \ldots, x_n) = fid(y_1, \ldots, y_m)}{\bullet\ (X_1, \ldots, X_n) = fid(Y_1, \ldots, Y_m), M, L \underset{p}{\to} (X_1, \ldots, X_n) = \bullet^{X_1=x_1, \ldots, X_n=x_n}_{Y_1, \ldots, Y_m}, enter(M, p, \{X_1, \ldots, X_n\}, \{Y_1, \ldots, Y_m\}), L}\ \textbf{(fcall)}$$

$$\frac{M([Y]) = (R\ (\_\ , (D\ y))) \quad M([X]) = (R\ (0, \_)) \quad [X] \neq [Y]}{\bullet\ (X) = k\ \texttt{fby}\ Y, M, L \underset{p}{\to} (X) = \bullet^{X=y}_{Y}\ y\ \texttt{fby}\ Y, enter(M, p, \{Y\}, \{X\}), L}\ \textbf{(fby)} \qquad \frac{\textit{conditions of}\ \textbf{(fby)}\ \textit{not satisfied}}{\bullet\ (X) = k\ \texttt{fby}\ Y, M, L \underset{p}{\to} Err}\ \textbf{(fby-err)}$$

$$(X_1, \ldots, X_n) = \bullet^{X_1=x_1, \ldots, X_n=x_n}_{Y_1, \ldots, Y_m}\ expr, M, L \underset{p}{\to} (X_1, \ldots, X_n) = expr\ \bullet, exit(M, p, \{X_1, \ldots, X_n\}, \{Y_1, \ldots, Y_m\}), L\ \textbf{(eq-exit)}$$

$$\frac{M([C]) = (R\ (\_\ , l)) \quad with \quad l.cache(p) = (\texttt{D}\ \texttt{true})}{\bullet\texttt{on}\ C\ aeq, M, L \underset{p}{\to} \texttt{on}\ C\ \bullet aeq, M, L}\ \textbf{(on+)} \qquad \frac{M([C]) = (R\ (\_\ , l)) \quad with \quad l.cache(p) = (\texttt{D}\ \texttt{false})}{\bullet\texttt{on}\ C\ aeq, M, L \underset{p}{\to} \texttt{on}\ C\ aeq\bullet, M, L}\ \textbf{(on-)}$$

$$\frac{\textit{conditions of}\ \textbf{(on+)}, \textbf{(on-)}\ \textit{not satisfied}}{\bullet\texttt{on}\ C\ aeq, M, L \underset{p}{\to} Err}\ \textbf{(on-err)} \qquad \frac{aeq, M, L \underset{p}{\to} aeq', M', L'}{\texttt{on}\ C\ aeq, M, L \underset{p}{\to} \texttt{on}\ C\ aeq', M', L'}\ \textbf{(on-comp)}$$

$$\frac{aeq, M, L \underset{p}{\to} Err}{\texttt{on}\ C\ aeq, M, L \underset{p}{\to} Err}\ \textbf{(on-comp-err)} \qquad \frac{aeq, M, L \underset{p}{\to} Err}{\texttt{top}\ aeq, M, L \underset{p}{\to} Err}\ \textbf{(grd-comp-err)}$$

$$\frac{aeq, M, L \underset{p}{\to} aeq', M', L'}{\texttt{top}\ aeq, M, L \underset{p}{\to} \texttt{top}\ aeq', M', L'}\ \textbf{(grd-comp)} \qquad \bullet\texttt{top}\ aeq, M, L \underset{p}{\to} \texttt{top}\ \bullet aeq, M, L\ \textbf{(grd)}$$

$$\texttt{thread}\ eq_1; \ldots; eq_k\bullet, M, L \underset{p}{\to} \texttt{thread}\ \bullet eq_1; \ldots; eq_k, M, L\ \textbf{(seq-loop)}$$

$$\frac{eq_i, M, L \underset{p}{\to} eq_i', M', L'}{\texttt{thread}\ eq_1; \ldots; eq_i; \ldots; eq_k, M, L \underset{p}{\to} \texttt{thread}\ eq_1; \ldots; eq_i'; \ldots; eq_k, M', L'}\ \textbf{(seq)} \qquad \frac{eq_i, M, L \underset{p}{\to} Err}{\texttt{thread}\ eq_1; \ldots; eq_i; \ldots; eq_k, M, L \underset{p}{\to} Err}\ \textbf{(seq-err)}$$

$$\frac{t_i, M, L \underset{p}{\to} t_i', M', L'}{t_1; \ldots; t_i; \ldots; t_k, M, L \underset{p}{\to} t_1; \ldots; t_i'; \ldots; t_k, M', L'}\ \textbf{(interleave)} \qquad \frac{t_i, M, L \underset{p}{\to} Err}{t_1; \ldots; t_i; \ldots; t_k, M, L \underset{p}{\to} Err}\ \textbf{(interleave-err)}$$

Figure 9.2: Platform semantics. From top to bottom: rules for API call equations, rules for dataflow equations, rules for guards, rules for sequential and parallel composition.

implementation of p amounts to making multiple correctness and semantics preservation statements:

Dataflow correctness p and q are correct under synchronous semantics, as defined in Section 8.1.4.

Non-functional completeness q is a complete implementation model, as defined in Section 7.4, allowing the synthesis of compilable and linkable implementation code.

Error-free execution The execution of the implementation does not result in an execution error.

Semantics preservation The dataflow semantics of p is preserved by the execution of the implementation obtained from q.

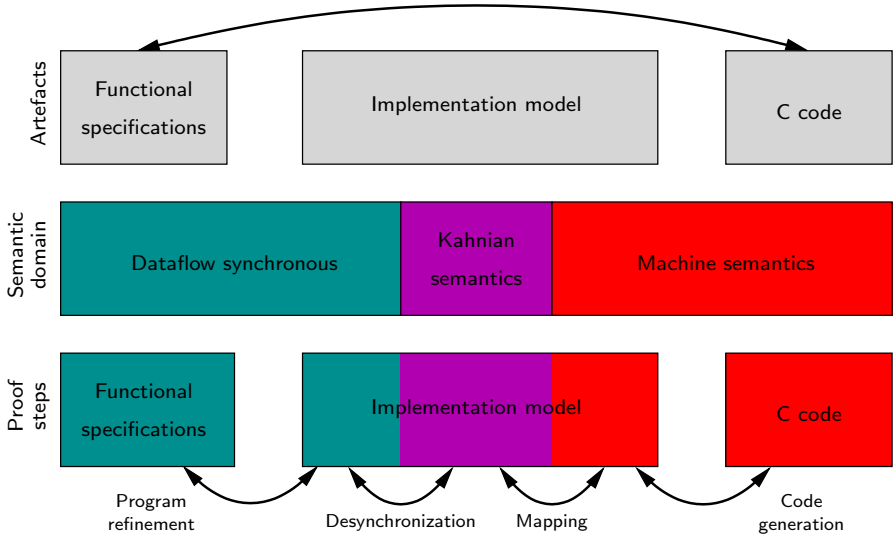


Figure 9.3: Proposed proof flow for semantics preservation

We have already covered in previous sections the first two properties. The definition of the machine semantics allows us to consider here the last two. We focus here on the last two properties, and in particular on semantics preservation, assuming that both implementation model and C code can be synthesized. The remainder of the section is dedicated to *formally defining semantics preservation* and suggesting techniques for ensuring it.

Given the important syntactic and semantic distance between functional specification and C implementation, we propose structuring the definition of semantics preservation (as well as its proof) in several steps, taking advantage of the artefact introduced in this chapter—the implementation model. As pictured in Figure 9.3, the flow spans 3 artefacts—functional specification, implementation model, and multi-threaded C code—and 3 semantic domains—dataflow synchronous semantics, machine semantics,

and the intermediate kahnian asynchronous semantics. Given that functional specifications are only interpreted under synchronous semantics, and that C code is only interpreted under concurrent C semantics, which is a form of machine semantics, the proposed flow involves defining semantics preservation in 4 steps:

Program refinement Between the functional specification p and the implementation model q. This amounts to defining when a synchronous program implements the semantics of another.

Desynchronization Between the implementation model q under synchronous semantics and the same implementation model under kahnian semantics.

Mapping Between the impementation model q under kahnian semantics and the same model under machine semantics.

Code generation Between the implementation model under machine semantics and the C code.

For the (Desynchronization) step, both the definition of the semantics preservation property and the proof that it holds for all correct synchronous programs is provided in Theorem 1 (in page 135).

We do not cover in this thesis the (CodeGeneration) step for two reasons:

- The InteLus machine semantics can also be viewed as an operational semantics of the subset of the concurrent C language to which synthesized code belongs. In this case, it is no longer necessary to define or prove the correctness of this step.
- If the use of a more standard concurrent semantics of C11 [44, 11] is required, the definition of the obligation is complex, as the code we synthesize extends the C11 dialect to allow formal reasoning about optimizations of the memory allocation and of the cache coherency protocol.

9.3.1 Program refinement

Defining the correctness of an implementation process (*e.g.* compilation) is often difficult, due to complex transformations applied to the source code [104]. However, in the system-level implementation of embedded systems, the set of transformations is often reduced to (real-time) scheduling, resource allocation, and the synthesis of glue code enforcing the scheduling and allocation choices. The data-flow synchronous model of InteLus allows the representation of such transformations, even when scheduling uses advanced features such as software pipelining [56].

To define the relation between functional specification and implementation model in such contexts, we introduce the notion of program refinement. Let p and p' be two correct InteLus programs. We say that p' is a refinement of p if p' is obtained from p by a combination of the following transformations:

Pipelining. Inserting or removing `pre` operators along existing data-flow dependencies.

Communication synthesis. Inserting copy equations along existing data-flow dependencies. Some of them are required by pipelining, and some other allow the representation of the memory coherency protocol.

Synchronization. Introduction of `event` variables, equations defining them, and `wait` and `done` constructs in the guards to represent the scheduling choices.

Adding guards or updating them to account for pipelining and communication.

Program refinement does not change the number of program inputs. It also preserves the sets of traces, modulo a pipelining relation (a fixed delay on non-input variables).

We can illustrate this refinement by using these transformations incrementally on the specification example of Figure 7.4.

```
fun P:()->(int)
fun C:(int)->()
var x:int
let
  x  = P()
  () = C(x)
tel
```

```
fun P:()->(int)
fun C:(int)->()
var x,y:int
    c:bool
let
       c  = false fby true
       y  = pre x
       x  = P()
  on c () = C(y)
tel
```

Figure 9.4: Initial functional specification (left) and pipelined specification (right)

In Figure 9.4, we see the result (right) of the pipelining of the initial Lustre specification (left). It introduces the variable `y`, a delayed copy of the original production `x` destined to the consumer function. It also introduces the boolean `c` that serves as an activation condition for the consumer so that it activates only at cycles where `y` is available.

Next in Figure 9.5 (left), we add identity equations to dissociate the semantics of values crossing the synchronous cycles of `pre` and `fby` operators (which will generate no code in the C implementation) from explicit assignment operations that are implemented with C assignments. To this end, we introduce the variables `c1` and `yd` that correspond to the delay equations that appeared with the pipelining tranformation.

The transformation from Figure 9.5 (left) to Figure 9.5 (right) adds a representation of memory hierarchy, assuming a certain split of the program into two threads executed on different cores. Variables `y0`, `y1`, and `y` on the right-side program respectively represent the value of the original variable `y`, as seen, respectively, in the cache of core 0, core 1, and in the main memory. Cache operations performing copies between caches and the main memory are represented with new copy operations. Note that variable `x` is not replicated—it becomes `x0`—because it will only be used one core 0. Note that

```
fun P:()->(int)
fun C:(int)->()
var x,y,yd:int
    c,c1:bool
let
       c  = false fby c1
       yd = pre y
       x  = P()
       y  = x
  on c () = C(yd)
       c1 = true
tel
```

```
fun P:()->(int)
fun C:(int)->()
var x0,y,y0,yd,yd1:int
    c,c1:bool
let
       c   = false fby c1
       yd  = pre y
       x0  = P()
       y0  = x0
       y   = y0
  on c yd1 = yd
  on c ()  = C(yd1)
       c1  = true
tel
```

Figure 9.5: Explicit affectations needed by pipelining (left) and to represent memory hierarchy (right) of pipelined specification of Figure 9.4 (right)

the variable replication and the new copy operations do not change the semantics of the program—the values produced by `P` are all consumed by `C` in the same order and during the same cycles as the pipelined specification of Figure 9.4 (right).

Finally in Figure 9.6, we add equations and guards that manipulate the pure synchronization type `event`. We add what corresponds to mutex operations. Variables `u` and `u1` correspond to the mutex that ensure that the consumer wait for its input, and `v` and `v1` the one ensuring that the production is not copied before the consumer is done with the previous value. We also add the variables `s`* and `t`* that allow the representation of the threads sequences decided by the scheduling through the `wait` and `done` constructs. Once again, these additions do not interfere with the initial communication pattern of the specification and the program stays well-formed, preserving the initial semantics.

At this point, we can also check for the properties ensuring implementability, as defined in Section 8.3.

9.3.2 Mapping

Semantics preservation is defined here as the preservation (between the kahnian interpretation of q and its interpretation under machine semantics) of the sequences of values taken as input (as guards or actual equation inputs) and produced as output by the execution of each equation of the implementation model. These sequences are represented with histories, introduced in Section 8.2.2.

We conjecture that semantics preservation is implied by the respect by q of the properties of Section 8.3 (1-boundedness, implementation of `fby` with no internal memory, explicit synchronization) and by the consistency between the dataflow part of API call

```
fun P:()->(int)
fun C:(int)->()
var x0,y,y0,yd,yd1:int
    c,c1:bool
    u,u1,v,v1:event
    s0,s1,s2,s3,s4,s5,t0,t1,t2,t3,t4,t5:event
let
  c   = false fby c1
  yd  = pre y
  v1  = top fby v
  u1  = u
  s5  = top fby s4
  t5  = top fby t4

  s5          done(s0)       x0  = P()
  s5 wait(s0) done(s1)       _   = v1
  s5 wait(s1) done(s2)       y0  = x0
  s5 wait(s2) done(s3)       y   = y0
  s5 wait(s3) done(s4)       u   = top

  t5          done(t0) on c  _   = u1
  t5 wait(t0) done(t1) on c  yd1 = yd
  t5 wait(t1) done(t2) on c  ()  = C(yd1)
  t5 wait(t2) done(t3)       v   = top
  t5 wait(t3) done(t4)       c1  = true
tel
```

Figure 9.6: Addition of synchronization equations and guards representing scheduling decisions of specification of Figure 9.5 (right)

equations and their API call annotation with respect to resource allocation.

By adding mapping annotations to the program of Figure 9.6, we reach the complete implementation model of Figure 7.4 (b). From there, we still need to check that resource allocation (memory, locks) ensures that:

- The error state is unreachable under machine semantics.
- Execution under machine semantics results in the same states as that under Kahnian semantics.

Chapter 10

Conclusion and perspectives

10.1 Conclusion

This thesis is the result of my work toward the safe and efficient parallelization of hard real-time systems. The problem was tackled on two fronts with :

- The development of an end-to-end compilation method from functional specifications with non-functional requirements to parallel real-time executable code with static guarantees of its correctness.
- The formal modeling of such implementations and the formal definition of their correctness with respect to the specifications.

10.1.1 Efficient parallelization of real-time applications

For the first and main point, I have designed and validated the first fully automated code generation flow capable of compiling a real-world control application into a parallel implementation that is both functionally correct and respects non-functional real-time requirements *without* making simplifying hypotheses concerning the overheads related to parallel/concurrent execution on off-the-shelf hardware. In particular, our flow does *not* require adding experience-based margins to computed 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 analysis into the scheduling loop, and maintaining precise memory mapping and interference information throughout the compilation and analysis flow. This avoids the pitfalls of methods that first build an implementation and only then perform schedulability analysis. Achieving this requires a tight integration of analysis and synthesis steps: the normalization phase that produces an SSA-like intermediate representation, the code generation steps of the dataflow synchronous program, the back-end C compiler and binary utilities (linker and loader), all the way to the parallel real-time scheduling and timing analysis. Integration ensures global consistency with respect to the timing model of the execution platform. The method provides good results on real-world applications, and is scalable. These results may percolate into industrial processes in the future, yet this may involve evolutions of the certification procedures and will certainly

require major efforts in qualifying parallel-capable versions of the leading tools (*e.g.*, Scade Suite and its KCG compiler).

Around this main contribution gravitate a few smaller ones in the fields touching the subject. I presented a way to describe real-time system specification using synchronous languages, focusing on a system-level with non-functional requirements and a way to expose its potential parallelism. I built an extension (albeit minimal) of an existing [77] timing model, of which I integrated an *incremental* interference analysis performed during the mapping. I explored different code generation strategies for parallel embedded code with not only worst-case guarantees as objective but also efficiency and robustness. I also provided a methodology to evaluate the performance of parallel implementations of real-time multi-period applications.

10.1.2 Back-end correctness formalization

For the second point, I proposed a way to model multi-threaded implementations of dataflow specifications using an extension to the Lustre language with additional constructs representing synchronization and annotations that define the actual implementation, allocation, and organization of the elements of the model on the hardware platform. With this model, I proposed a novel approach for the formal validation of the functional correctness of these implementations with respect to their specifications. This approach consists of proving the semantic preservation between the different interpretations of the model with the definition of three formal operational semantics: dataflow synchronous, kahnian, and machine.

While work on this subject is not completed—I did not yet cover the real-time aspects, nor completed the proofs—I believe that I have provided solid proof of my initial claims:

- Data-flow specifications can be given efficient multi-threaded implementations that preserve an internal data-flow structure.

- The data-flow structure of such implementations can be exposed using new language constructs that build upon existing data-flow synchronous modeling practice.

Exposing the dataflow structure of such implementations facilitates formulating the correctness of implementations. If the conjecture of Section 9.3.2 is correct, then the formalization we provided allows us to state a set of correctness properties that is complete, guaranteeing the correctness of all aspects of a concurrent implementation (for systems of a certain type). This is a significant advance with respect to the state of the art. Furthermore, validating the correctness of an implementation with respect to the set of properties we identified can be done using sufficient criteria requiring low-complexity analysis inside the synchronous data-flow semantic domain.

This work lays the ground for the formal and automated validation of multi-threaded implementations that expose their dataflow structure. That includes in particular implementations generated through our compilation method.

10.2 Challenges and perspectives

Looking back to when I began this work, the initial goal was closest to the second part of this thesis with the end-to-end formal validation of parallel real-time systems implementations. We quickly came to realize that state-of-the-art implementation methods at the time lacked a key element to enable their formalization: the cohesion between the different steps of the whole implementation process. Interfaces between timing analysis, real-time scheduling and code generation relied on unsafe hypotheses (arbitrary margins added to WCETs, cost of the OS/glue code, etc.) that were not suitable for formal reasoning. Thus, the goal shifted from the formal validation of end-to-end implementation flows to the development of one that streamline its different steps in a way that would eliminate the need for such hypotheses.

From that point, we faced a difficult exercise. Streamlining an implementation flow to a compilation method that provide safety guarantees required the very tight, formal integration of the whole flow around a precise timing model. This led to a lot of engineering work to interface the different steps of the flow (specification normalization, conventions on code generation, hypotheses on resource allocations, timing analysis through external tool, and their automation!). Furthermore, maximizing the efficiency of implementations revealed itself to be more important than expected. From a scientific perspective, efficiency makes a strong argument when proposing a compilation method. And it is also relevant in an industrial context, even required.

Given the novelty of the approach, in order to prove its feasibility we needed to go all the way and develop the tools allowing use to produce practical results. This implies an in-depth work with a very restricted target rather than an in-breadth exploration of the possibilities offered by the current landscape, both scientific and material.

From this point onward, many challenges remain. We can refine our compilation method on several aspects:

- The experimental evaluation emphasized the importance of evenly distributing the application load among its minor frames. This optimization problem is related to I/O latency requirements specific to each application and industrial workflow, which are generally not exposed in the source model, but rather at more abstract system design levels.

- Interference provisioning has been done in a very simple way that use little to none of the information available to the compiler to optimize their bounds. This can be improved with our knowledge of the memory behavior of each dataflow function and the current state of the scheduling to obtain tighter bounds making the implementation more efficient than what we obtain today by exploring manually different values for the percentage of WCET of each function.

In the longer term, our success on one specific shared-memory architecture motivates an exploration in-breadth towards more complex or different hardware platforms:

- Extension to the full Kalray MPPA many-core (and to its next generation, Coolidge) and other time-predictable architectures [105, 106] requires taking into account

network-on-chip (NoC) interconnects and/or shared caches (*e.g.* by cache partitioning).

- Extension to other timing predictable architectures, such as TC27x [107] that requires taking into account the NUMA model.
- On more classical multi-cores (ARM, POWER, x86) featuring speculative out-of-order execution or hardware cache coherence, it does not seem realistic to provide static hard real-time guarantees, and different approaches with lower safety expectations should be considered.

We can also explore other implementation forms. Now, we generate linear code corresponding to the expansion of specification to the hyper-period. This leads to a consequent amount of code for large applications that must be allocated in memory. To reduce memory footprint, traditional code generation method using conditional execution can be used. In particular, using continuation-passing style for the code snippets would go hand-in-hand with the use of NoC interconnect or communication with external memory.

For the work on formalization and validation, we will continue along two axes. First, we will enrich the implementation modeling language to incorporate new features. They consist mainly of considering new elements of architecture, such as DMAs, network, or external memory.

Another major extension to the current approach, and to my eyes the point that this thesis regrettably lacks to be really complete, is the extension of implementation models with real-time. This is work that can build upon our results on InteLus. It requires the addition of constructs for real-time constraints and synchronization to the InteLus language, and the notion of time to its different semantics.

The full formalization of implementation models should facilitate the building of mapping tools like the one defined in the first part of this thesis.

The second objective is to develop a formally proven translation validation tool covering the transformation of functional specifications into implementation models. This requires proving the conjecture of Section 9.3.2 and then development and proof of tools for the validation of the hypotheses of the conjecture (properties such as 1-boundedness). To be complete, the validation tool needs to be complemented later with the validation of the translation from implementation models to code running on the platform with a property of semantic preservation between our machine semantics and C11 semantics, and with the validation of the translation from Lustre to InteLus, using the program refinement property. Such tool could join and interact with other formally verified compilation tools such as Velus [46] and CompCert [104].

Ultimately, such tool is bound to be extended to the formal validation of real-time properties of the implementations. Although, as we have seen with this work, formally integrating real-time aspects to compilation is not an easy task, and will be even more so in the context of formal methods.

Bibliography

[1] S. Baruah, M. Bertogna, and G. Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer, 2015.

[2] N. Scaife et al. "Defining and Translating a "Safe" Subset of Simulink/Stateflow into Lustre". In: *Proceedings EMSOFT*. Pisa, Italy, 2004.

[3] B. Pagano et al. "A model based safety critical flow for the AURIX^{TM}multi-core platform". In: *Proceedings ERTS2*. Toulouse, France, Jan. 2018.

[4] *LabView*. `ni.com/labview`. Accessed 20 March 2018.

[5] *SysML*. `http://www.omgsysml.org/`. Accessed 20 March 2018.

[6] F. Mallet and R. de Simone. "MARTE: A Profile for RT/E Systems Modeling, Analysis". In: *Proceedings Simutools*. 2008.

[7] *ARINC 653: Avionics Application Software Standard Interface. Part 1 – Required Services. Revision 3*. 2010.

[8] *AUTOSAR*. `autosar.org`. Accessed 20 March 2018.

[9] Edward A. Lee. "The Problem with Threads". In: *IEEE Computer* 39.5 (May 2006), pp. 33–42.

[10] A. Miné. "Static Analysis of Run-Time Errors in Embedded Real-Time Parallel C Programs". In: *Logical Methods in Computer Science* 8.1 (Mar. 2012). `https://arxiv.org/abs/1203.3724`.

[11] Jeehoon Kang et al. "A Promising Semantics for Relaxed-Memory Concurrency". In: *POPL*. 2017.

[12] P. Koopman. *A Case Study of Toyota Unintended Acceleration and Software Safety*. `https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf`. Sept. 2014.

[13] P. Caspi et al. "LUSTRE: A declarative language for programming synchronous systems". In: *POPL*. 1987.

[14] M. Chetto, ed. *Real-time systems scheduling, volumes 1 and 2*. ISTE Ltd. and John Wiley & Sons, Inc., 2014.

[15] Peng Deng et al. "A model-based synthesis flow for automotive CPS". In: *Proccedings ICCPS*. 2015.

[16] E. Yip et al. "The ForeC Synchronous Deterministic Parallel Programming Language for Multicores". In: *Proceedings MCSoC*. 2016.

[17] R. Pellizzoni et al. “PREM A Predictable Execution Model for COTS-Based Embedded Systems”. In: *Proceedings RTAS*. 2011.

[18] A. Graillat et al. “Parallel Code Generation of Synchronous Programs for a Many-core Architecture”. In: *Proceedings DATE*. 2018.

[19] H. Rihani et al. “Response Time Analysis of Synchronous Data Flow Programs on a Many-Core Processor”. In: *Proceedings RTNS*. 2016.

[20] D. Potop-Butucaru and I. Puaut. “Integrated Worst-Case Execution Time Estimation of Multicore Applications”. In: *Proceedings of the WCET workshop, Paris, France*. 2013. DOI: 10.4230/OASIcs.WCET.2013.21.

[21] H. Ozaktas, C. Rochange, and P. Sainrat. “Automatic WCET Analysis of Real-Time Parallel Applications”. In: *Proceedings of the WCET workshop*. https://hal.archives-ouvertes.fr/hal-01239727. Paris, France, 2013.

[22] S. Louise et al. “The OASIS Kernel: A Framework for High Dependability Real-Time Systems”. In: *Proceedings HASE*. 2011.

[23] *SynDEx mapping tool*. www.syndex.org. Accessed 06/2017.

[24] A. Basu et al. “Rigorous Component-Based System Design Using the BIP Framework”. In: *IEEE Software* 28.3 (2011).

[25] *The SchedMCore software framework*. http://sites.onera.fr/schedmcore/.

[26] C. Pagetti et al. “Multi-task Implementation of Multi-periodic Synchronous Programs”. In: *Discrete Event Dynamic Systems* 21.3 (2011), pp. 307–338.

[27] T. Carle et al. “From Dataflow Specification to Multiprocessor Partitioned Time-triggered Real-time Implementation”. In: *Leibniz Transactions on Embedded Systems* 2.2 (2015).

[28] A. Cohen et al. “Hard Real Time and Mixed Time Criticality on Off-The-Shelf Embedded Multi-Cores”. In: *Proceedings ERTS2, Toulouse, France*. 2016.

[29] P. Aubry et al. “Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor”. In: *Proceedings ICCS*. 2013.

[30] B. Bodin, A. Munier Kordon, and B. Dupont de Dinechin. “K-Periodic schedules for evaluating the maximum throughput of a Synchronous Dataflow graph”. In: *SAMOS XII*. 2012.

[31] P. Caspi et al. “From Simulink to SCADE/Lustre to TTA: A Layered Approach for Distributed Embedded Applications”. In: *Proceedings LCTES*. 2003.

[32] V. Brocal et al. “Xoncrete: a scheduling tool for partitioned real-time systems”. In: *Proceedings ERTS*. 2010.

[33] C. Deutschbein et al. “Multi-core Cyclic Executives for Safety-Critical Systems”. In: *Proceedings SETTA*. LNCS 10606. 2017.

[34] T. Carle et al. “Static Mapping of Real-Time Applications onto Massively Parallel Processor Arrays”. In: *Proceedings ACSD, Tunis, Tunisia*. 2014. DOI: 10.1109/ACSD.2014.19.

[35] C. Pagetti et al. “Automated generation of time-predictable executables on multi-core”. In: *Proceedings RTNS*. POITIERS, France, 2018.

[36] S. Skalistis and A. Simalatsar. “Near-optimal Deployment of Dataflow Applications on Many-core Platforms with Real-time Guarantees”. In: *Proceedings DATE*. 2017.

[37] B. Rouxel, S. Derrien, and I. Puaut. “Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.5s (Oct. 2017), pp. 1–20.

[38] T. Carle and D. Potop-Butucaru. “Predicate-aware, Makespan-preserving Software Pipelining of Scheduling Tables”. In: *ACM Trans. Archit. Code Optim.* 11.1 (Feb. 2014).

[39] B.R. Rau and C.D. Glaeser. “Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing”. In: *Proceedings of the 14th annual workshop on Microprogramming, IEEE*. 1981.

[40] V. H. Allan et al. “Software Pipelining”. In: *ACM Computing Surveys* 27.3 (1995).

[41] S. Craciunas and R. Serna Oliver. “Combined Task- and Network-level Scheduling for Distributed Time-triggered Networked Systems”. In: *Real-Time Systems* 52(2) (2016).

[42] Q. Perret et al. “Mapping hard real-time applications on many-core processors”. In: *Real-Time Networks and Systems (RTNS2016)*. Brest, France, Oct. 2016.

[43] R. Gorcitz et al. “On the Scalability of Constraint Solving for Static/Off-Line Real-Time Scheduling”. In: *Proceedings FORMATS)*. Madrid, Spain, 2015.

[44] M.J. Batty. “The C11 and C++11 Concurrency Model”. `https://www.cs.kent.ac.uk/people/staff/mjb211/docs/toc.pdf`. PhD thesis. Wolfson College, University of Cambridge, 2014.

[45] S. Tripakis et al. “Implementing Synchronous Models on Loosely Time-Triggered Architectures”. In: *IEEE Transactions on Computers* 57.10 (2008).

[46] T. Bourke et al. “A Formally Verified Compiler for Lustre”. In: *Proceedings PLDI*. 2017.

[47] Van-Chan Ngo et al. “Modular translation validation of a full-sized synchronous compiler using off-the-shelf verification tools”. In: *Proceedings SCOPES*. 2015.

[48] N. Halbwachs. “A synchronous language at work: the story of Lustre”. In: *Proceedings MEMOCODE*. 2005.

[49] *The Heptagon/BZR distribution*. `http://heptagon.gforge.inria.fr/`, accessed on 05/19/2018.

[50] L. Gérard et al. “A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler”. In: *Proceedings LCTES, Beijing, China*. 2012.

[51] R. Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Transactions on Programming Languages and Systems* 13.4 (1991), pp. 451–490.

[52] D. Biernacki et al. “Clock-directed Modular Code Generation for Synchronous Data-flow Languages”. In: *Proceedings LCTES*. 2008.

[53] S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[54] R. Wilhelm et al. "The worst-case execution-time problem - overview of methods and survey of tools". In: *ACM Trans. Embedded Comput. Syst.* 7.3 (2008), 36:1–36:53.

[55] T. Carle et al. "Reconciling performance and predictability on a many-core through off-line mapping". In: *Proceedings ReCoSoC, Montpellier, France*. 2014. DOI: 10.1109/ReCoSoC.2014.6861367.

[56] T. Carle and D. Potop-Butucaru. "Predicate-aware, Makespan-preserving Software Pipelining of Scheduling Tables". In: *ACM TACO* 11.1 (Feb. 2014).

[57] G. Berry, S. Moisan, and J.-P. Rigault. "Esterel : Towards a synchronous and semanticaly sound high-level language for real-time applications". In: *Proceedings RTSS*. 1983.

[58] Jean-Louis Bergerand et al. "Outline of a Real Time Data Flow Language". In: *Proceedings of the 6th IEEE Real-Time Systems Symposium (RTSS '85), December 3-6, 1985, San Diego, California, USA*. 1985, pp. 33–42.

[59] David Harel and Amir Pnueli. "On the development of reactive systems". In: *Logics and models of concurrent systems*. Springer, 1985, pp. 477–498.

[60] D. Potop-Butucaru, R. de Simone, and J.-P. Talpin. "Embedded systems handbook". In: Taylor&Francis, 2005. Chap. The synchronous hypothesis and synchronous languages.

[61] Reinhard Von Hanxleden et al. "SCCharts: sequentially constructive statecharts for safety-critical applications: HW/SW-synthesis for a conservative extension of synchronous statecharts". In: *ACM SIGPLAN Notices*. Vol. 49. 6. ACM. 2014, pp. 372–383.

[62] Gérard Berry and Georges Gonthier. "The Esterel synchronous programming language: design, semantics, implementation". In: *Science of Computer Programming* 19.2 (1992), pp. 87–152. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(92)90005-V. URL: http://www.sciencedirect.com/science/article/pii/016764239290005V.

[63] *SCADE Suite*. https://www.ansys.com/products/embedded-software/ansys-scade-suite.

[64] P. LeGuernic et al. "Programming real-time applications with SIGNAL". In: *Proceedings of the IEEE* 79.9 (1991).

[65] Edward A Lee and David G Messerschmitt. "Synchronous data flow". In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245.

[66] Greet Bilsen et al. "Cycle-static dataflow". In: *IEEE Transactions on signal processing* 44.2 (1996), pp. 397–408.

[67] Joseph Tobin Buck and Edward A Lee. "Scheduling dynamic dataflow graphs with bounded memory using the token flow model". In: *1993 IEEE international conference on acoustics, speech, and signal processing*. Vol. 1. IEEE. 1993, pp. 429–432.

[68] Sanjoy K Baruah. "Dynamic-and static-priority scheduling of recurring real-time tasks". In: *Real-Time Systems* 24.1 (2003), pp. 93–128.

[69] Chung Laung Liu and James W Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment". In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.

[70] Thomas A Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. "Giotto: A time-triggered language for embedded programming". In: *International Workshop on Embedded Software*. Springer. 2001, pp. 166–184.

[71] Christoph M Kirsch and Ana Sokolova. "The logical execution time paradigm". In: *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120.

[72] Olivier Bouissou and Alexandre Chapoutot. "An operational semantics for Simulink's simulation engine". In: *ACM SIGPLAN Notices*. Vol. 47. 5. ACM. 2012, pp. 129–138.

[73] Paul Caspi, Grégoire Hamon, and Marc Pouzet. "Synchronous functional programming: The lucid synchrone experiment". In: *Real-Time Systems: Description and Verification Techniques: Theory and Tools. Hermes* (2008).

[74] R. Wilhelm et al. "Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems". In: *IEEE TCAD* 28.7 (2009).

[75] S. Abbaspour, F. Brandner, and M. Schoeberl. "A time-predictable stack cache". In: *Proceedings ISORC*. 2013.

[76] *Kalray*. `https://www.kalrayinc.com/`. Accessed 11 July 2019.

[77] Hamza Rihani. "Many-Core Timing Analysis of Real-Time Systems". 2017GREAM074. PhD thesis. 2017. URL: `http://www.theses.fr/2017GREAM074/document`.

[78] Jan Reineke et al. "Timing predictability of cache replacement policies". In: *Real-Time Systems* 37.2 (2007), pp. 99–122.

[79] L. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (Aug. 1990).

[80] *ARINC664. Aircraft Data Network.*

[81] Jean-Louis Colaço et al. "Scade 6: from a Kahn Semantics to a Kahn Implementation for Multicore". In: *2018 Forum on Specification & Design Languages (FDL)*. IEEE. 2018, pp. 5–16.

[82] Damien Chabrol et al. "Deterministic Distributed Safety-Critical Real-Time Systems within the Oasis Approach." In: *IASTED PDCS*. 2005, pp. 260–268.

[83] René LC Eveleens. "Integrated modular avionics development guidance and certification considerations". In: *Mission Systems Engineering* 2 (2006), pp. 1120–1132.

[84] Certification Authorities Software Team. *Position Paper CAST-32A on Multicore Processors.* 2016.

[85] Clément Ballabriga et al. "OTAWA: an open toolbox for adaptive WCET analysis". In: *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer. 2010, pp. 35–46.

[86] Christian Ferdinand and Reinhold Heckmann. "ait: Worst-case execution time prediction by static program analysis". In: *Building the Information Society*. Springer, 2004, pp. 377–383.

[87] *AbsInt*. https://www.absint.com/ait/index.htm Accessed on 05/19/2018.

[88] *Kalray k1b VLIW Core Architecture Reference Manual (KETD-62 W41)*. Kalray S.A. 2015.

[89] M. Smotherman et al. "Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling". In: *Proceedings of the 24th Annual International Symposium on Microarchitecture*. MICRO 24. 1991. URL: http://doi.acm.org/10.1145/123465.123482.

[90] S. Larin and A. Trick. *Instruction scheduling for Superscalar and VLIW platforms. Temporal perspective*. https://llvm.org/devmtg/2012-11/Larin-Trick-Scheduling.pdf. 2012 LLVM Developers' Meeting presentation. 2012.

[91] Thomas Carle. "Compilation efficace de spécifications de contrôle embarqué avec prise en compte de propriétés fonctionnelles et non-fonctionnelles complexes". PhD thesis. Paris 6, 2014.

[92] D. Potop-Butucaru et al. "Clock-driven distributed real-time implementation of endochronous synchronous programs". In: *Proceedings EMSOFT*. 2009.

[93] F. Balarin et al. "Scheduling for Embedded Real-Time Systems". In: *IEEE Des. Test* 15.1 (Jan. 1998), pp. 71–82.

[94] G. Liao et al. "A comparative study of multiprocessor list scheduling heuristics". In: *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*. Vol. 1. Jan. 1994, pp. 68–77. DOI: 10.1109/HICSS.1994.323184.

[95] Colin Fidge. "Timestamps in Message-Passing Systems That Preserve the Partial Ordering". In: *Australian Computer Science Communications* 10.1 (1988).

[96] J.K. Hollingsworth. "Critical Path Profiling of Message Passing and Shared-Memory Programs". In: *IEEE Transactions on Parallel and Distributed Systems* 9.10 (1998), pp. 1029–1040.

[97] *Detecting Memory Bandwidth Saturation in Threaded Applications*. https://software.intel.com/en-us/articles/detecting-memory-bandwidth-saturation-in-threaded-applications/ Retrieved on 05/22/2018. Intel. 2010.

[98] J.P. Talpin et al. "Constructive polychronous systems". In: *Sci. Comput. Program.* 96 (2014), pp. 377–394.

[99] P. Caspi and M. Pouzet. "Synchronous Kahn Networks". In: *Proceedings ICFP*. 1996.

[100] George Hagen and Cesare Tinelli. "Scaling up the formal verification of Lustre programs with SMT-based techniques". In: *2008 Formal Methods in Computer-Aided Design*. IEEE. 2008, pp. 1–9.

[101] Amar Bouali. "XEVE, an ESTEREL verification environment". In: *International Conference on Computer Aided Verification*. Springer. 1998, pp. 500–504.

[102] G. Kahn. "The Semantics of Simple Language for Parallel Programming". In: *IFIP Congress*. 1974.

[103] D. Potop-Butucaru and B. Caillaud. "Correct-by-construction asynchronous implementation of modular synchronous specifications". In: *Fundamenta Informaticae* CXXVII (2006), pp. 1–27.

[104] X. Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (2009), pp. 107–115.

[105] Michael Zimmer et al. "FlexPRET: A processor platform for mixed-criticality systems". In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2014, pp. 101–110.

[106] Andreas Hansson et al. "CoMPSoC: A template for composable and predictable multi-processor system on chips". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 14.1 (2009), p. 2.

[107] *TC27x User's Manual*. Infineon Technologies AG. 2014.