Automatic Parallelization from Lustre Models in Avionics

Jean Souyris, Keryan Didier, Dumitru Potop-Butucaru, Guillaume Iooss, Timothy Bourke, Albert Cohen, Marc Pouzet

Automatic Parallelization from Lustre Models in Avionics

(Short paper)

Jean Souyris[(1)], Keryan Didier[(2)], Dumitru Potop[(2)], Guillaume Iooss[(3)], Albert Cohen[(2)(3)], Timothy Bourke[(2)(3)], Marc Pouzet[(4)(3)(2)]

(1) : Airbus; (2) : Inria Paris; (3) : ENS; (4) : UPMC

Abstract: This poster presents ongoing research on automatic generation and execution of embedded parallel C code. We target safety-critical avionics programs specified in the synchronous language Lustre. The work described is part of the ITEA 3 project ASSUME (September 2015 - August 2018). ASSUME focuses mainly on embedded software engineering for multi-/many-core platforms. Both synthesis, e.g., automatic code generation, and verification, e.g., static analysis, of programs are addressed in the project. ASSUME is driven by the use cases of its industrial partners. One of these use cases consists in the parallelization of an avionics application comprising about 5500 Lustre nodes. After an overview of the ASSUME project, both parallel code generation and execution on a many-core platform will be presented and demonstrated.

Keywords: ITEA3, ASSUME project, Avionics software, multicores and many-cores, Lustre, synchronous, automatic code generation.

1. Introduction – Position of the poster

ASSUME (ITEA 3, September 2015 - August 2018)

This project proposes a seamless engineering methodology for delivering trustworthy new mobility assistance functions on multi-core and many-core architectures. The problem is addressed on the constructive and on the analysis side. For efficient construction and synthesis of embedded systems, the project provides new tools, standards and a methodology to cover most of the challenges by design. In addition, ASSUME provides a well-integrated, sound static analysis solution that allows proving the absence of problems even in a multi-core environment. New algorithms will be integrated in exploitable tools. New interoperability standards and requirements formalization standards will facilitate tool and market player cooperation. The ASSUME consortium includes leading European industry partners for mobility solutions, tool and service providers for embedded system development, as well as leading research institutes for static analysis and for model-driven and traditional embedded systems development.

The solutions for automatic code generation from Lustre models are developed in **ASSUME's work package entitled "WP4 Synthesis of Predictable Concurrent Systems"**. This WP defines solutions for constructing correct and efficient concurrent systems. By correctness we mean both functional correctness, and the respect of non-functional requirements such as timing predictability. The developments of WP4 cover the whole spectrum of embedded systems components: application software, basic software such as communication and synchronization protocols, and hardware. They also cover all the steps of the implementation process, from hardware development to compilation and real-time scheduling. The technical focus of the work package is on formal compiler verification and the correct-by-construction real-time implementation of parallel applications. The common denominators of the technologies developed in WP4 are formalization and full automation. The objective of WP4 is to contribute to the development and transfer of each technology taken separately and to start integrating the various technological bricks into coherent design flows shaped by the needs expressed in the industrial case studies. For instance, one objective of the project is to enable the fully automatic generation of real-time implementations directly from high-level SCADE avionics specifications.

Airbus' use case in ASSUME: today, avionics applications are developed using synchronous languages that are compiled in order to produce an executable for a single-core processor. For an application that needs more calculation speed, one solution would be to increase the core frequency. But this has undesirable effects on avionics computers (e.g., on power consumption). A better solution for increasing the calculation speed is to use more calculation units (multi-/many-cores). For that, a parallel software engineering shall be developed. The development tools and methods that comprise it shall:

- Ensure a significant execution speed-up;
- Preserve the semantics of the model during the transformation of the Lustre models into executable object code.

The main objective of this use case is to explore and develop methods and tools in order to be able to transform a LUSTRE model of a command/control application into a parallelized executable with a significant gain in calculation speed. Initially, this gain will be evaluated using measurements, and in a later step, it will be demonstrated using formal methods.

2. State-of-the-art

Synthesis of critical real-time software for multi-processor architectures: much of the classical work on real-time scheduling (both in research and

industry) relies on a process whereby the implementation is derived by manual transformations. Implementation is followed by verification and validation phases where timing analysis and schedulability analysis guarantee the respect of non-functional requirements. But the complexity of multi-processor execution targets and also of functional and non-functional specifications is increasing rapidly, which makes it difficult to preserve a manual process due to cost, time-to-market pressures, and confidence issues related to the number of errors introduced by human coders.

Recent advances in this direction have largely automated the construction of task code and even the generation of full real-time implementations, like in the industrial tools such as Simulink Real Time™ or ASTERIOS® Developer.

A few approaches have gone even further, providing real-time schedulability guarantees for the generated multi-threaded code. Among them, we only mention here previous work on SynDEx [GLS99], ΨC[LLAD11], BIP[BBB+11], SchedMCore [SCH17] or the time-driven mapping of Lustre [TPB+08].

For a more complete related work presentation we defer the reader to [CPSL15].

The originality of the work presented here comes from taking into account memory access interferences in the real-time analysis, and from considering the avionics non-functional requirements specific to the case study.

3. Software Characteristics of the use case

The program to be parallelized is an avionics application specified in Lustre. Each of the approximately 5500 nodes in the specification is periodically activated according to one of four harmonic periods: C1 (10 ms), C2 (20 ms), C3 (40 ms) and C4 (120 ms). This program has not been specified as a single LUSTRE program with one base clock, but rather as multiple LUSTRE programs on different base clocks. Importantly, this results in a very flat model. Indeed, contrary to a standard Lustre model in which there is a "main" node at the root of a deep call graph, almost all Lustre nodes in this model are themselves isolated main nodes, and their call graphs are between 0 and 2 nodes deep. Since there are very few calls, most data exchanges between nodes are performed via global shared variables (about 36000). These variables constitute the inner state of the application. Under these conditions, the scheduling of the nodes cannot be generated from the Lustre models alone, since the data dependencies between nodes, e.g., node2 shall use data1 computed by node1 in the same period (C2 for instance), are not in the model. This is why, together with the Lustre nodes, the authors of this kind of Lustre specification also specify the order in which nodes are to be executed. This is done per period C1, C2, C3 and C4.

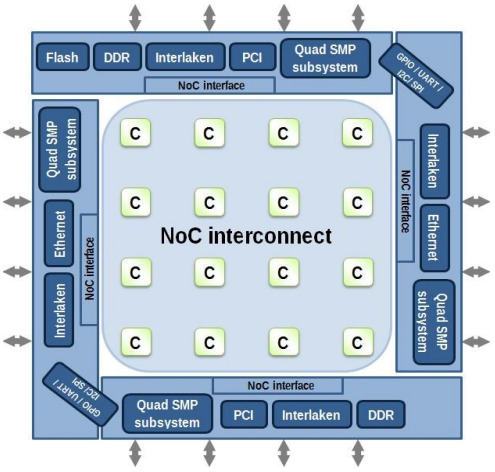
In the current mono-core implementation:

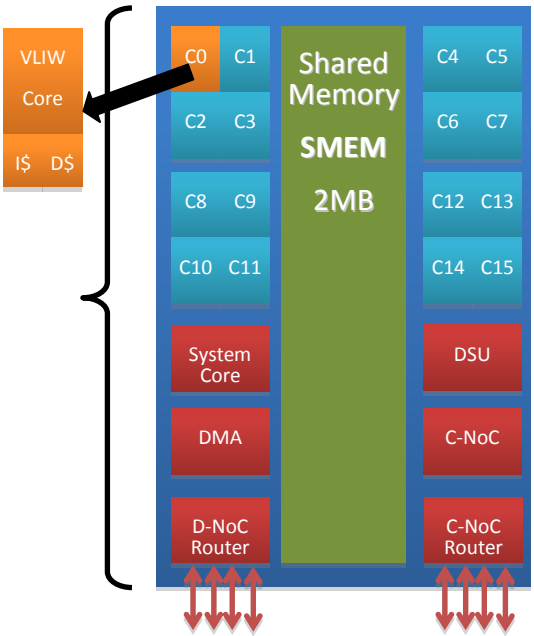
- The Lustre nodes are compiled into code by an automatic code generator;
- The compiled nodes are scheduled by a deterministic sequencer generated at build-time. The inputs of the tool that produces the actual scheduler are:
  - The above mentioned ordered lists of nodes;
  - The WCET of each node.

4. Parallel Execution Platform of the Use Case

The first picture below is the high-level architectural view of Kalray's many-core MPPA. In the central part of the picture, one can see 16 clusters (C) that communicate via a "NoC interconnect". The four borders of the view show interfaces with the external world, e.g., DDR, Flash, and Ethernet.

The second picture below shows the internal architecture of a cluster, i.e., 16 Very Long Instruction Word (VLIW) cores each with their own instruction and data caches, 2 Mb of shared SMEM, and devices for transferring data to and from the interconnect.





6. Automatic Parallel Code Generation from Lustre

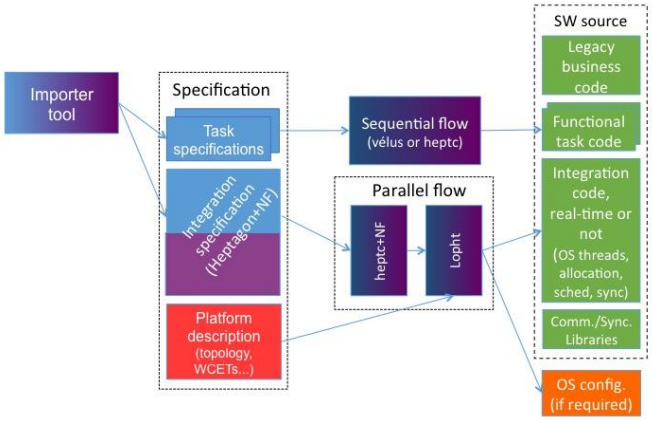


Figure 1. Parallel code generation flow

Following long-standing practice in the avionics industry, the design methodology we propose is based on the use of synchronous languages for the high-level functional specification of applications. More specifically, we shall be using various dialects of the LUSTRE synchronous data-flow language, whose industrial version SCADE is commercialized by ANSYS/ESTEREL TECHNOLOGIES. The standardization of the functional specification level is essential to the cost-effective construction of tools. When the industrial process requires the use of a functional specification formalism different from LUSTRE, process-specific automatic translation tools are needed to ensure seamless integration by translation into a LUSTRE dialect (for instance, we use the dialect provided by the Heptagon compiler [Heptagon]). The functional specification is formed of two parts: the specification of each sequential task as a synchronous program (nodes), and the integration specification. Each task specification is compiled into sequential C code using a classical LUSTRE/Heptagon compiler. The integration specification describes how tasks communicate and synchronize. It is taken as input by the parallelization tool.

```
period(0xf0000) node main () returns ()
var
  x : int ;
let
   x = f( ) ;
  () = g(x) ;
  () = h(x) ;
tel
```

Figure 2. Simple Heptagon specification with non-functional annotations

For parallel and/or real-time implementation, we also need a non-functional (NF) specification. It comprises a description of the execution platform and a set of non-functional requirements of various types. The content of the NF specification depends on the code generation objectives. It can range from a few parallelization annotations to complex descriptions of both the execution platform (e.g., topology, arbitration, and timing characterizations) and non-functional require-ments (e.g., real-time, partitioning, and allocation).

In our case, there are two pieces of information. The Heptagon integration specification (a node) is annotated with all the NF requirements (hence the mixed blue+purple color in Fig. 1). In our example (Fig. 2), an annotation only specifies the expected period of the application. We provide a separate architecture description file that defines, among other details, the number of cores to exploit inside a shared memory cluster and, for each sequential task, various characteristics needed during real-time scheduling (WCET, worst-case number of memory accesses, etc.).

Starting from the high-level functional and non-functional specifications, the tool flow of Fig. 1 produces C code, which is compiled, along with legacy business code and with platform libraries, to produce the executable code of the implementation.

The result is a seamless flow of automatic transformations going all the way from a high-level specification to a running implementation. Such a flow of transformations ensures the correctness of the resulting implementation with respect to the high-level specification provided that:

- the platform description faithfully describes the behaviour of the execution platform (HW and libraries);
- the high-level compiler and C compiler are correct, and;
- the (optional) process-specific importer tool is correct.

6.1. Details of the compilation flow

Heptagon is both a LUSTRE dialect and an open-source compiler for this language. For the scope of this section, we shall disambiguate between the two by calling the compiler “heptc”. One use of this compiler is to generate sequential C code from synchronous programs (nodes) written in a Lustre dialect (much like SCADE). Accepted LUSTRE dialects include Heptagon and Scade v4. The Scade front-end has been added during ASSUME to allow the handling of case studies.

The flow starts with an importer tool that translates the industrial specification into functionally equivalent Heptagon code. This tool is needed since industrial use case specifications do not fit directly into the synchronous model (both in terms of format and of the underlying computational model). Internally, this tool builds a dependency graph between the different tasks of the application. Because of the multi-periodic nature of the application, we have to determine which instance of a node produces data read by another instance of a node. We rely on the sequential schedule in order to fit exactly with its semantic.

Further optimization and transformation can be done at this representation level, such as the retiming transformation, in order to distribute the computation of the slow periods across the multiple iterations of the fast period.

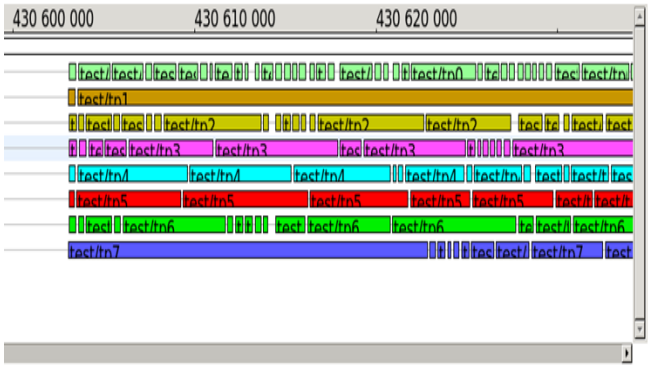
Then, we use the heptc compiler to produce the non-hierarchic data-flow language taken as input by the Lopht tool. In this case, heptc takes as input an extension of the Heptagon language, called Heptagon+NF, where program annotations allow the specification of non-functional requirements. We name this extension heptc+NF.

The Lopht tool [CPSL15] takes as input the non-hierarchic data-flow and the non-functional requirements output by heptc+NP, as well as the platform description described above. It either produces parallel implementation C code that is both functionally correct and respects the non-functional requirements, or reports why it was not able to produce such an implementation.

7. Demonstrator

In order to produce the actual binary file to be loaded onto the MPPA platform, the per-core tasks (up to 16) that have been generated are compiled (C compiler) and linked with platform software (initialization, and observation).

The trace capture capabilities of the Kalray's platform and software tools enable one to produce the following:



Each line corresponds to a core (8 cores in the picture). Time is displayed on the top of the picture and each coloured rectangle on a line represents the execution of one node.

Next to the poster, the demonstration of both the automatic generation of the parallel code and the execution of it on a Kalray machine will be performed.

Furthermore, the speed-up between the parallel implementation and the mono-core one will be shown by also running the latter during the demo.

8. Verified LUSTRE Compilation

In parallel with the Heptagon prototype, we have developed a machine-verified compiler for a subset of Lustre [Velus]. After some simplifications, namely the removal of constant tables and assembly code, the case-study can be compiled in under two minutes by code extracted from the Coq proof assistant. The correctness of this compiler has been formally specified and proved, but it only generates single-body sequential code. The extension to parallel code remains a research challenge.

9. References

[Heptagon] The Heptagon language and compiler. http://heptagon.gforge.inria.fr/ Accessed on June 17, 2017

[CPSL15] T. Carle, D. Potop, Y. Sorel, D. Lesens. From Dataflow Specification to Multiprocessor Partitioned Time-triggered Real-time Implementation. In LITES 2(2) 2015.

[VELUS17] T. Bourke, L. Brun, P.-E. Dagand, X. Leroy, M. Pouzet, and L. Rieg. A Formally Verified Compiler for Lustre. In PLDI 2017.

[BBB+11] Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J. Rigorous component-based system design using the BIP framework.. In IEEE Software 28, 3 (2011)

[GLS99] T. Grandpierre, C. Lavarenne, Y. Sorel. Optimized rapid proto-typing for real-time embedded heterogeneous multipro-cessors. In CODES'99, Rome, Italy.

[TPB+08] Tripakis, S., Pinello, C., Benveniste, A., Sangiovanni-Vincentelli, A., Caspi, P., and Natale, M. D. Implementing synchronous models on loosely time- triggered architectures. IEEE Transactions on Computers 57, 10 (2008).

[LLAD11] S. Louise, M. Lemerre, C. Aussaguès, V. David. The OASIS kernel: A framework for high dependability realtime systems. In HASE 2011, doi: 10.1109/HASE.2011.38.

[SCH17] SchedMCore toolset. http://sites.onera.fr/schedmcore/. Accessed 20 Jun. 2017.

1. Glossary

WCET	Worst Case Execution Time.
WCAT	Worst Case Access Time.