Implementation of Real-Time Data-Flow Synchronous Programs on a Many-Core Architecture

Matheus Schuh∗,†,1

∗Univ. Grenoble Alpes, CNRS, Grenoble INP2, VERIMAG, 38000 Grenoble, France
†Kalray S.A., 38330 Montbonnot-Saint-Martin, France

ABSTRACT
Real-Time systems are in the core of our modern lives. They are extensively used in the avion- ics, automotive and medical industry, programmed usually by systems engineers with data-flow synchronous languages, such as Lustre or SCADE. Historically, single-core processors have been used as their execution platform due to their simplicity and predictability. The rise of multi and many-core architectures have matched up with an increase in demand for computational power and integrity of tasks. Code generation, mapping and scheduling of data-flow applications in multiple processors remain as a challenge for their market-wide adoption.

KEYWORDS: Real-Time Systems; Time-Critical; Many-Core Architectures; Data-Flow Synchronous Applications; Worst-Case Execution Time

1 Introduction
Applications that must respect temporal deadlines, producing deterministic and correct outputs are what we call real-time systems. Timing constraints are, in this case, part of the specification and the failure to meet them can have drastic consequences. A common example is a flight controller mechanism composed of the pilot stick and altitude sensors as inputs, a processor for calculation and a wing actuator as output that also feed backs its current state. The delay between an input arrival and an output generation must be known and guaranteed before runtime. In order to achieve this, the Worst-Case Execution Time (WCET) of the task is calculated either statically or through measurement.

Synchronous data-flow languages abstract time through logical periodic clocks and focus on functionality, being a perfect design tool for these complex systems. Simply put, a synchronous program is an interconnected network of nodes, each one performing a function and possibly containing sub-nodes. The connections expose dependencies and thus computation order requirements. In a typical tool-chain, these languages are used to generate

1E-mail: matheus.schuh@univ-grenoble-alpes.fr
2Institute of Engineering Univ. Grenoble Alpes
platform-agnostic C code and auxiliary files leveraging the structure between nodes.

This document will mainly address the problems related to the scheduling and the timing analysis of this generated code, counting with the background of two previous works: [Rih17] and [Gra18]. Single-core and many-core architectures will be considered as platform targets, highlighting the implementation particularities when parallel execution is required. Lastly, an overview of the author’s future work will be given, covering the addition of a Real-Time Operating System (RTOS) and high-level communication layers to the platform.

2 Code generation of synchronous programs

Synchronous data-flow languages ensure formal semantics, determinism and correctness of programs by construction. An essential aspect of the generated C code is the inheritance of these properties. This allows the certification only of the generation tools, which are then capable of transferring these characteristics to the code.

**Single-core code generation.** Data-flow programs have intrinsic order requirements, providing a straightforward static scheduling between the tasks. An WCET analysis is then performed, with prior knowledge of the architecture, checking the schedulability. Once this is verified, a sequential time-triggered or event-triggered execution can be performed on the platform.

It is important to point out that, as there is only one core, the code from each task is isolated and will execute uninterruptedly until its completion. Moreover, the communication, usually achieved using in-memory structures, happens without interference as only one processor can access the data at a given time.

**Many-core code generation.** The parallel execution in different processors of the possibly dependant nodes composing the data-flow is the real challenge in this domain. This execution paradigm introduces concurrency, data exchange between cores and interference when using on-chip resources. These problems have to be carefully addressed within a time-critical context. First of all, the parallelism must be extracted from the program. We assume the hypothesis that one node is one runnable candidate, i.e. an atomic piece of code to be scheduled and mapped to a core. Sequential code is then generated for each runnable and its execution environment is decided afterwards.

The communication must now take into account possible interference when accessing shared resources, in this case the memory. In synchronous languages, communication is either instantaneous or delayed to the next period (pre operator, c.f. figure 1). The former implies in a copy of the output of one node to the input of another one, followed by the emission of a notification token on a channel. The latter is done using a double buffer, creation of an auxiliary task and enforcing scheduling order. Lastly, the synchronization is implemented through blocking waits on the aforementioned tokens.

3 Many-core response time analysis

WCET is sufficient to verify the schedulability of generated code on single-core architectures. For many-core, this is not true. There is an overhead due to the interference when
accessing shared resources. This is why the notion of Worst-Case Response Time (WCRT), which accounts for the local WCET and possible interference, has been introduced.

An extremely precise approach to the WCRT is too complex and not scalable, as it is necessary to analyse all running code on all cores at all time. A naive approach would be to assume that there will always be interference when accessing resources, which is pessimistic and leads to overestimation. Therefore, to have a realistic and bounded WCRT, the timing analysis must be done with knowledge of the hardware (in this work the Kalray MPPA2), the software (synchronous data-flow) and the hypothesis that time-triggered execution is possible (avoiding preemption overhead).

Multi-Core Interference Analysis (MIA) Tool. Developed by Hamza Rihani [Rih17] during his thesis, this tool is capable of calculating the WCRT for a given set of tasks on a many-core architecture through a mathematical model of the memory bus arbiter. The algorithm implemented takes as input the tasks isolated WCET and the worst number of memory accesses. This input is supplied by an external tool such as Otawa [BCRS10] or Heptane [HRP17].

The first step is to take an initial scheduling and mapping and estimate the current interference due to other running tasks, adding this delay to the original WCET. If the WCET was modified and tasks now overlap, their release dates should be adjusted. This is repeated until there is no more WCET modification, which leads to the WCRT of the task within this scenario. A final step is to verify that the deadlines are still respected after these modifications.

Model of computation. A complete framework for data-flow parallel code generation, mapping and scheduling on many-core platforms was developed by Amaury Graillat [Gra18] during his thesis. Its target is the MPPA2 architecture relying on some of its features, e.g. banked local memory, and software mechanisms to minimize interference. An overview of the model can be seen in figure 1.

One memory bank is reserved for each core containing code, input buffer and local variables. The generated runnable is executed in a local bank and writes its output to a remote bank. Interference is then possible only during communication. As previously stated, the tasks activation is time-triggered and the MIA tool calculates release dates respecting data dependencies and timing constraints. Finally, the platform is bare-metal, using a non-preemptive static schedule. The mapping between runnables and cores is done through an external tool.

---

**Figure 1: Complete framework for data-flow parallel code generation**
4 Conclusion

The work presented so far has showed how it is possible to tightly bound the WCRT of a synchronous data-flow generated code and deploy it on a many-core architecture with real-time guarantees. A strong motivation for the author’s future work is the new Kalray architecture, MPPA3, and the goal to deliver a more generic and reusable framework.

MPPA3 has a larger local memory (4MB instead of 2MB), allowing to port a RTOS with an Asymmetric Multiprocessing (AMP) scheme. The memory bus arbiter is also modified, requiring a new modeling of access timing in the MIA tool. The reduced number of clusters (5 instead of 16) simplified the Network on Chip (NoC) routing, providing a more predictable inter-cluster communication that can be exploited if tasks run on different clusters. This reduction also facilitates the work of implementing high-level communication layers such as Data Distribution Service (DDS).

To conclude, having the demand for integrity of tasks on the same chip, we aim at building a more versatile model of computation, capable of hosting Lustre/SCADE generated code, as well as Simulink and Logical Execution Time (LET) paradigms such as Giotto, building a mixed criticality capable platform.

Glossary

| AMP | Asymmetric Multiprocessing | NoC | Network on Chip |
| DDS | Data Distribution Service | RTOS | Real-Time Operating System |
| LET | Logical Execution Time | WCET | Worst-Case Execution Time |
| MIA | Multi-Core Interference Analysis | WCRT | Worst-Case Response Time |

References


