Parallel Code Generation of Synchronous Programs for a Many-core Architecture

Amaury Graillat[1][3] Matthieu Moy[2] Pascal Raymond[1] Benoît Dupont de Dinechin[3]
[1]Verimag - Univ. Grenoble Alpes
Saint Martin d'Heres, France
first.name@univ-grenoble-alpes.fr
[2]LIP
69007 Lyon, France
matthieu.moy@univ-lyon1.fr
[3]Kalray
Montbonnot, France
benoit.dinechin@kalray.eu

***Abstract*—Embedded systems tend to require more and more computational power. Many-core architectures are good candidates since they offer power and are considered more time predictable than classical multi-cores.**

Data-flow Synchronous languages such as Lustre or Scade are widely used for avionic critical software. Programs are described by networks of computational nodes. Implementation of such programs on a many-core architecture must ensure a bounded response time and preserve the functional behavior by taking interference into account.

We consider the top-level node of a Lustre application as a software architecture description where each sub-node corresponds to a potential parallel task. Given a mapping (tasks to cores), we automatically generate code suitable for the targeted many-core architecture. This minimizes memory interferences and allows usage of a framework to compute the Worst-Case Response Time.

INTRODUCTION

Lustre is a synchronous data-flow language, whose industrial version, Scade, is widely used for time-critical applications, especially in avionics. These applications are more and more complex and single-core architectures are no longer sufficient. Many-core are a good alternative to multi-core since they offer a distributed memory avoiding the bottleneck of a single memory.

We present a tool to generate parallel code from unmodified Lustre programs to a many-core architecture. Timing non-determinism of the platform is addressed with a state-of-the-art execution model: bare-metal implementation with static scheduling and banked memory mapping to minimize interferences.

The novelties of this paper are: 1) The code is generated for a many-core allowing communication through shared-memory or Network-on-Chip (NoC). 2) The tasks are time-triggered to minimize interference and allow bounding the Worst-Case Traversal Time (WCTT) of the application: we use an existing framework to bound WCTT and compute the release dates of the tasks.

The structure of this paper is the following: Sec. II gives an overview of the parallelization methods. Sec. III shows of how to configure platform and how to generate parallel code minimizing interferences. Sec. IV describes the connection of the generator with the framework of response-time analysis. In Sec. V solution is applied to some use-cases.

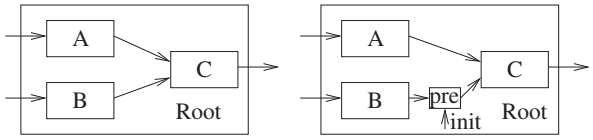

Fig. 1. Synchronous Data-Flow Networks. (a) A and B can be computed in parallel and B and C must be in sequence. (b) B can be computed in parallel with both A and C because of the delay (`pre`).

I. BACKGROUND

A. Lustre and its sequential compilation

A Lustre program, called a *node*, is a data-flow network of sub-nodes (see. Fig. 1). Lustre has the synchronous semantics: a node behavior is a sequence of atomic reactions. Nodes can be combinatorial (e.g. A, B and C in Fig. 1), meaning that their current output depends on their current input. Lustre provides a built-in operator `pre` which is equivalent to a register in a synchronous circuit: in Fig. 1b, the `pre` operator produces the output `init` during the very first reaction, and then the value produced by `B` at the previous reaction.

Classical Lustre compilation consists in producing a purely sequential procedure `step` implementing one atomic reaction. This sequential code must respect the data dependencies. For instance, in Fig. 1a, `A` and `B` can be executed in any order before `C`. In Fig. 1b the dependency between `B` and `C` is broken by a `pre` operator. Hence, `A` must be computed before `C`, but `B` can be computed at any time.

The goal of this work is to exploit the data-(in)dependency to generate parallel code rather that sequential one.

B. Targeted Many-Core Platforms and Time-Predictability

We target many-core platforms such as the Kalray MPPA-256 [1]. They are composed of clusters connected with a NoC. In a cluster, the memory or scratchpad memory of can be shared between the processors of the cluster. As in T-CREST [2], this local memory is accessible in write from any other clusters with a Remote Direct Memory Access (RDMA) working through the NoC. In the Kalray MPPA this memory is multi-banked in a such way that a memory access in one bank does not interfere with access in another bank.

978-3-9819263-0-9/DATE18/©2018 EDAA

On single-core architectures, Worst-Case Execution Time (WCET) analysis is sufficient to bound the response time of a program. On multi-core, shared-memory access time depends on memory congestion. The Multicore Response Time Analysis (MRTA) [3] is a generic framework to analyse response time taking into account memory bus, preemption and DRAM interference. It has been extended for the MPPA Compute Cluster memory bus [4], and adapted to an application model in which tasks have release date.

II. Parallel Code Generation: Overview and Constraints

A. Parallelization at the Program Architecture Level

For a data-flow synchronous program, several levels of parallelization are possible. One can use the classical code generation, and try to parallelized the generated C code using classical methods such as OpenMP or fork-join. This *intra-node parallelization* is not automatic and does not not exploit the intrinsic parallelism of the data-flow design.

Another solution is to allow parallel execution of nodes at any level in the data-flow hierarchy, by using, for instance, a "remote" procedure call (send/receive). This method exploits the data-flow structure but requires to modify the compiler.

We have chosen a less general solution where only the direct sub-nodes of the main node are implemented as parallel tasks. For instance, in Fig. 1a, the sub-nodes A, B and C are compiled and embedded into 3 concurrent tasks. The main node does not require any code, but is used to define the communications between the tasks, in such a way that no operating system is needed at runtime. This method is similar to consider the main node as an *Architecture Description Language*, and close to the solution proposed in Prelude [5]. However Prelude requires a real-time operating system (RTOS) at runtime.

B. Method Overview

We consider the top-level node of a Lustre program. Functional code (*step* function) of each sub-node is generated using the unmodified Lustre v6 compiler. Channels of communication between each pair of communicating nodes are extracted by syntax analysis. It gives information about the type and size of the data and delay (`pre`) and initial value. In Fig. 2 the dotted circle are channels (for a pair of node). Channels Root-A and D-Root are communication with the environment. We implement the channels with shared-memory or NoC communication. A task is composed of the functional code of a node and some channels.

We statically schedule tasks on the cores using a variant of Nguyen *et al.* [6]'s proposal, which uses an Integer Linear Programming-based mapping and scheduling algorithm for multi-core. It takes the local cache affinity into account.

C. Task-Triggering

Task execution is Data-Triggered (DT) when computation starts when all the data has been received. It is robust since the behavior is preserved when execution time is variable. Task execution is time-triggered (TT) when computation starts at a release date. Rihani *et al.* [4] provide a framework to compute the release date taking memory interference into account. TT allows a tighter bound of hardware interferences and response-time (see Sec. I). To improve safety, we combine TT and DT: tasks wait for their release dates *and then* wait for the inputs. For prototyping releases can be set to 0.

Delayed Communications: The `init->pre x` expression specifies a delayed communication. In Fig. 3, `A` takes an output of `B` from the previous period and the value `init` for the first period. Both the current and the previous value are stored. The delay can be implemented with a double buffer `[b0,b1]` and a `swap(b0,b1)` function. Access to the buffer are denoted `get(b0)` and `set(b1)`.

Several schedules are possible for `swap`, `get`, `set` and the computation of `A` and `B`. `Swap` can be executed at the beginning of the period (Fig. 4a) or at the end (Fig. 4b). The first schedule is less convenient because the `[b0,b1]` buffer has to prevent modification of `b1` until `swap` has terminated. Also, as the NoC directly writes into the buffer, preventing this modification requires an extra synchronization message. Moreover, this requires scheduling `set(b1)` after `swap`.

Executing `swap` at the end of the period (Fig. 4b), allows to consider the `pre` operator as any other task (read, compute, write). This `PRE` task takes as input the buffer `b1`, and when executed, copies this buffer to the input (`b0`) of the consumer `A`. The task `PRE` must be scheduled after `A` and `B` and a barrier is required to ensure the completion before the next period.

D. Tasks Communication

Data and code of a core are assigned to a memory bank. Two communication policies exist: the *remote read policy* consists in reading input in the producer's memory when it is needed. The main advantage of this solution is that data is not duplicated. For the *remote write policy*, the producer dispatches the data into the consumers memory. Remote write is more suited for the RDMA-capable NoC and ensures time locality of the memory accesses.

III. Implementation on a Many-core Architecture

We recall the principles chosen for this implementation: static schedule, TT secured by DT (II-C), spacial isolation of code and data on private banks (II-D), remote write (II-D).

Top-Level Node: Root node is implemented with two tasks to dispatch the inputs (Root-A in Fig. 2) and gather the outputs (D-Root).

Scheduling and Communication: Each core executes a code without preemption *e.g.*, a core executing tasks A and B in sequence (see Fig. 2). `wait_for_data_A` is a blocking function ensuring that data is present in the context `ctx_A` when it returns. `send_data_A` remotely writes data and notifies the consumer. The corresponding code is:

```
A_ctx_reset(&ctx_A);
B_ctx_reset(&ctx_B);
while(true) {
    wait_data_A();
    A_step(&ctx_A);
    send_data_A();
```

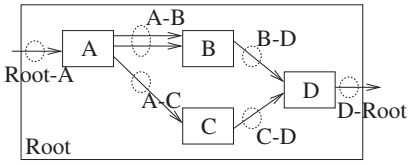

Fig. 2. Extraction of channels from a program.

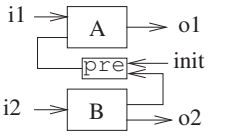

Fig. 3. Lustre program with a `pre`.

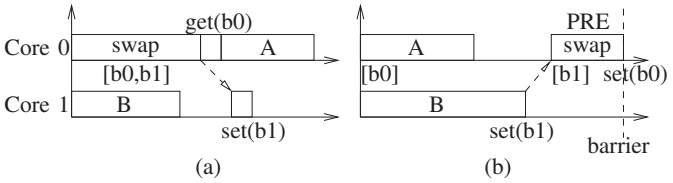

Fig. 4. Implementation of the `pre` operator: double-buffer swap can be done at the beginning (a) or the end (b) of the period.

```
    wait_data_B();
    B_step(&ctx_B);
    send_data_B();
}
```

Implementation of `send_data_*` depends on the nature of the communication. Through NoC, the data is written in a register of the DMA. Through the memory, the outputs are copied in the context of the destination. Implementation of `wait_data_*` is time-triggered, a local timer is used to wait until the release date. Of course, the timers need to be synchronized. In case of deadline miss or for prototyping, a token per channel ensures reception of the data and functional correctness. Hardware event can be used to implement tokens.

IV. Parallelization and Real-Time

In time-critical applications such as avionics, implementation must ensure the testability and traceability of the code. Dynamic scheduling and memory allocation are prohibited. The response time analysis of the program is required.

The framework of [4] takes an execution instance of a data-flow program which is represented as a Directed Acyclic Graph (DAG). Given a schedule, the framework computes release dates for each task such that all the data dependencies are satisfied. Computation of the response times takes into account the interference on data transfers. It requires the classical WCET (in isolation) of each task, the assigned memory bank and the amount of communications.

Usage of this framework in our tool is in four steps: First, we generate a binary executable with approximate release dates. Second, the WCET in isolation of tasks are computed on this binary. Third, the framework computes the safe release dates. Then, safe release dates are then integrated in the binary.

V. Evaluation

A. Sensors Processing on 8 flows

The sensor processing program reads a matrix of $8 * 512$ floats from the input/output cluster of the MPPA, transposes it and dispatches the flows to 8 FFTs blocks. Results are gathered in a single matrix. In the pipelined version (see Fig. 5), stage 1, 3, and the 8 FFTs are concurrently executed on 10 cores. In the non-pipelined version the FFT are executed in parallel on 8 cores. On data-triggered generated code, we obtain:

	Throughput @400 Mhz	Exec. Speedup	Latency	Latency Speedup
Sequential	100.9/s	1.00	9.90ms (1 stage)	1.00
Parallel (8 cores)	396.3/s	3.93	2.52ms (1 stage)	3.93
Pipelined (10 cores)	947.0/s	9.38	3.16ms (3 stages)	2.34

The pipelined version provides a better throughput than the non-pipelined version on 8 cores, but its latency is 1.67 times worse, although still better than the sequential version.

The FFTs take 85.9% of the sequential execution. If the 8 FFTs are in parallel the Amdahl law gives a speedup of 4.03. By measurement the speedup is 3.93 for 8 cores which is close to this optimum. The difference is due to the cost of the synchronizations mechanisms.

B. ROSACE: Flight-Control Use-case

ROSACE [7] is a case-study inspired from true avionic control applications. It controls aircraft altitude and speed. The structure of this case-study is representative of industrial applications, nevertheless it does not have heavy computation thus we added extra computation in each controller nodes: “ROSACE+0”, “+100” or “+200” cycles. “No Compute” contains only the communications. Fig. 6 shows an environment simulation and a controller. We execute the environment on an I/O Cluster and the controller on a Compute Cluster.

We look for the highest frequency F such that the controller is still schedulable. Fig. 6 shows the relative execution frequencies of the nodes. Fig. 8 gives the schedule on 5 cores with the release date (R) of each task for time-triggering.

Durations of this period are given in Fig. 7 for each version of the use case (each amount of computation in the nodes).

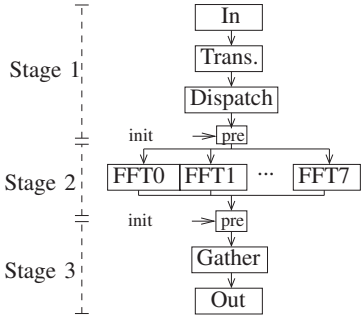

Fig. 5. Processing on 8 sensors pipelined on 3 stages.

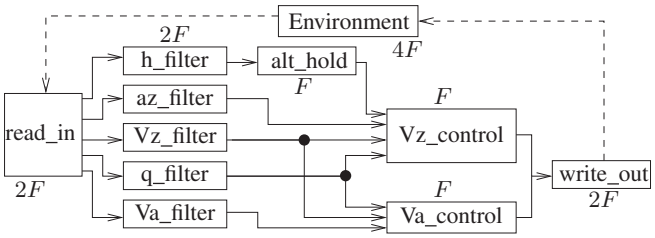

Fig. 6. ROSACE Controller: Lustre implementation

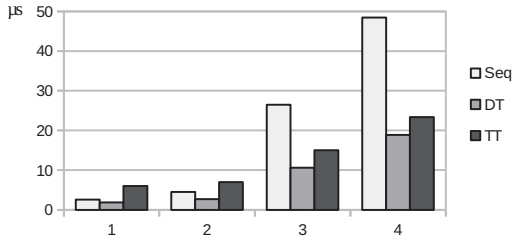

Fig. 7. ROSACE: (1) Communication only, (2) ROSACE+0, (3) ROSACE+100 and (4) ROSACE+200 cycles in each node are compared for Sequential, Data-Triggered and Time-Triggered methods.

DT is always faster than the sequential version and shows a speedup between 1.36 and 2.57. DT is always better than TT with a speedup from 3.18 to 1.24 but this difference seems to be constant (about 4 μs) and due to the over-approximation of the WCET of each tasks and communications compared to the actual execution. However, DT does not provide any guarantee of WCRT.

VI. Related Work

Our work extracts tasks from the top-level node as [5] for a synchrounous language and [8] for Logical Execution Time (LET). For AUTOSAR, tasks are extracted from the whole model in [9] which removes some precedence between tasks to enable more parallelism. Determinism of behavior is enabled by adding timestamps to communications. A synchronous program can be automatically split into tasks communicating through FIFO [10], [11] by analysing the data-dependencies or tasks can be manually expressed with a fork/join paradigm as in [12].

To take advantage of the multi-bank (or distributed) memories, we enforce shared-memory communication coherency by both data-flow and remote write. When data are shared and global there are different scheme. In *explicit communication*, memory is directly accessed by the task. In *implicit communication* (IC), tasks work on a local copy of the inputs. IC and LET ensures data consistency but increase latency [13].

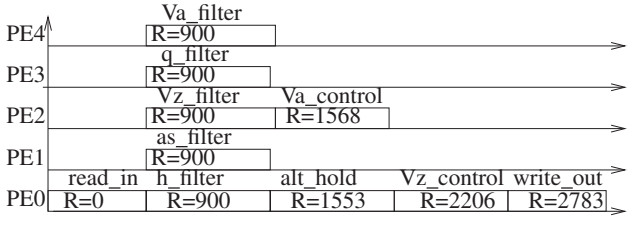

Fig. 8. Static schedule the ROSACE controller tasks on 5 cores. First period is represented. For each task, the release is provided.

Conclusion

This paper presents a tool to generate parallel code from a Lustre program. The solution takes advantage of the data-flow paradigm: tasks and communication channels are extracted from each sub-node of the top-level node. The delay (`pre`) operator is implemented as a particular task.

Generated code adopts a classical execution model to enable determinism of the many-core platform and minimizes interferences: tasks are statically scheduled, interruptions are disabled, code and data are mapped in memory-banks and a protocol synchronizes all the chip clocks. Tasks are time-triggered and the solution uses a framework to bound the Worst-Case Response Time of the application. Remote-write policy allows both NoC and shared-memory communications.

We applied our tool to a sensor processing application and a multi-periodic academic flight control case study. We show both good speedups and worst-case guarantees.

For future work, memory usage can be reduced by decreasing the number of buffers used for communication. The method could be adapted to support features such as Scade automata, and to extract parallelism at any level of the program hierarchy.

References

[1] B. D. De Dinechin, D. Van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *DATE'14*. IEEE, 2014, pp. 1–6.

[2] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann *et al.*, "T-crest: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[3] S. Altmeyer, R. I. Davis, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "A generic and compositional framework for multicore response time analysis," in *RTNS'15*. ACM, 2015, pp. 129–138.

[4] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer, "Response time analysis of synchronous data flow programs on a many-core processor," in *RTNS'16*. ACM, 2016, pp. 67–76.

[5] M. Cordovilla, F. Boniol, J. Forget, E. Noulard, and C. Pagetti, "Developing critical embedded systems on multicore architectures: the prelude-schedmcore toolset," in *RTNS'11*, 2011.

[6] V. A. Nguyen, D. Hardy, and I. Puaut, "Scheduling of parallel applications on many-core architectures with caches: bridging the gap between WCET analysis and schedulability analysis," in *JRWRTC 2015*, Nov. 2015.

[7] C. Pagetti, D. Saussi, R. Gratia, E. Noulard, and P. Siron, "The rosace case study: From simulink specification to multi/many-core execution," in *IEEE RTAS'14*, April 2014, pp. 309–318.

[8] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Embedded software*. Springer, 2001, pp. 166–184.

[9] S. Kehr, E. Quiñones, B. Böddeker, and G. Schäfer, "Parallel execution of autosar legacy applications on multicore ecus with timed implicit communication," in *DAC'15*. ACM, 2015, p. 42.

[10] P. Caspi, A. Girault, and D. Pilaud, "Automatic distribution of reactive systems for asynchronous networks of processors," *IEEE T Software Engineering*, vol. 25, no. 3, pp. 416–427, 1999.

[11] A. Girault, X. Nicollin, and M. Pouzet, "Automatic rate desynchronization of embedded reactive programs," *TECS*, vol. 5, no. 3, pp. 687–717, 2006.

[12] E. Yip, A. Girault, P. Roop, and M. Biglari-Abhari, "The ForeC synchronous deterministic parallel programming language for multicores," in *MCSoC'16*. Lyon, France: IEEE, Oct. 2016.

[13] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst, "Communication centric design in complex automotive embedded systems," in *LIPIcs*, vol. 76. Schloss Dagstuhl-LZI, 2017.