Redundant Dataflow Applications on Clustered Manycore Architectures

Christoph Kühbacher, Theo Ungerer, and Sebastian Altmeyer
University of Augsburg
Augsburg, Germany
{kuehbacher,ungerer,altmeyer}@es-augsburg.de

ABSTRACT

Increasing performance requirements in the embedded systems domain have encouraged a drift from singlecore to multicore processors. Cars are an example for complex embedded systems in which the use of multicores continues to grow. The requirements of software components running in modern cars are diverse. On the one hand there are safety-critical tasks like the airbag control, on the other hand tasks which do not have any safety-related requirements at all, for example those controlling the infotainment system. Trends like autonomous driving lead to tasks which are simultaneously safety-critical and computationally complex. To satisfy the requirements of modern embedded applications we developed a dataflow-based runtime environment (RTE) for clustered manycore architectures. The RTE is able to execute dataflow graphs in various redundancy configurations and with different schedulers. We implemented our RTE design on the Kalray Bostan Massively Parallel Processor Array and evaluated all possible configurations for three common computation tasks. To classify the performance of our RTE, we compared the non-redundant graph executions with OpenCL versions of the three applications. The results show that our RTE can come close or even surpass Kalray's OpenCL framework, although maximum performance was not the primary goal of our design.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; **Redundancy**; **Interconnection architectures**; **Multicore architectures**;

KEYWORDS

dataflow, runtime environment, software redundancy, NoC-based architecture, embedded systems

ACM Reference Format:
Christoph Kühbacher, Theo Ungerer, and Sebastian Altmeyer. 2022. Redundant Dataflow Applications on Clustered Manycore Architectures. In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22), April 25–29, 2022, Virtual Event, .* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3477314.3507272

https://doi.org/10.1145/3477314.3507272

1 INTRODUCTION

The trend towards increasing performance requirements is no longer limited to the domain of high performance computing, and multicore processors are widely used in complex embedded systems today. Cars are an example for a complex embedded system consisting of various electronic control units (ECUs) connected via different kinds of buses. A major reason for the increasing use of multicore processors in the automotive domain is to consolidate different software components on one chip and thus reduce the number of ECUs. Tesla is a pioneer in this field. Back in 2012, the Model S already utilized only 3–4 ECUs [23]. Since the de facto standard in the automotive industry, AUTOSAR (AUTomotive Open System ARchitecture), was originally designed for singlecore processors, an extension of the software stack was required. First support for multicore processors was added in version 4.0 of AUTOSAR. However, the AUTOSAR multicore extensions were only designed for a simple hardware model in which cores access the main memory and other peripherals over a shared bus [4]. To meet the requirements of future automotive applications, there is great interest in extending the existing multicore capabilities and adding support for additional hardware architectures like clustered manycores [4, 12, 22].

The software components running on the ECUs of modern cars are diverse and have different requirements. First, there are safety-critical tasks like the airbag control, anti-lock braking system, electronic stability control and emergency brake assist. While such tasks are not computationally complex, they place high demands on predictability and fault tolerance. The complete opposite would be tasks controlling the car's infotainment system which becomes more performance demanding as technology evolves, but is not safety-critical at all so that a best-effort approach is sufficient. In between lies a broad spectrum of tasks with diverse performance and safety requirements, for example active suspension or head light control tasks.

With the continuous expansion of the assistance systems and the goal of fully autonomous driving, safety-critical software components are becoming more and more complex. A common task in this area is the execution of convolutional neural networks (CNNs) for detecting pedestrians and objects like lane lines, street signs and other vehicles from camera images [15, 19]. CNNs are directed graphs consisting of different types of operation nodes, for example convolution, Rectified Linear Unit (ReLU) and pooling nodes. Multi- and manycore processors are well-suited for the execution of CNNs because of the parallelism within operation nodes and, depending on the graph, the possibility to execute nodes concurrently.

In embedded systems with high safety requirements, the correctness of computed results is mandatory. The only way to ensure the correctness of data is to introduce redundancy to the system.

Without redundant computation and storage, it is impossible to verify whether data is correct or incorrect due to a fault, such as a bitflip. Typically, data is computed two or three times. Higher levels of redundancy are uncommon.

To satisfy the varying requirements of complex embedded applications, we developed a runtime environment (RTE) for clustered manycore architectures based on a network-on-chip (NoC). With modern computational tasks such as CNNs in mind, we based our RTE design on directed acyclic graphs (DAGs) which are executed in a dataflow fashion. Our approach targets systems with varying requirements regarding predictability and fault tolerance as it is able to execute arbitrary DAGs with either offline or online scheduling in different redundancy configurations. Changing the redundancy configuration at runtime is also possible. Our execution model follows a coarse-grain dataflow style, i.e. graph nodes refer to potentially complex functions and data passed between them is often a structure or collection of data elements.

The main contributions of this paper are:

- A coarse-grain dataflow execution model based on directed acyclic graphs with support for redundant execution.
- An RTE design for clustered manycore architectures which is able to execute graphs with either offline or online scheduling in different redundancy configurations.
- A comparison of our bare-metal RTE implementation on the Kalray architecture with Kalray's OpenCL framework regarding their performance on three computing tasks.
- An evaluation of the different redundancy configurations our RTE supports.

2 HARDWARE ARCHITECTURE OVERVIEW

An overview of the hardware architecture we consider is shown in Fig. 1. The central component is a network-on-chip (NoC) which connects an arbitrary number of tiles. Our general RTE design is not limited to a particular NoC topology. The only requirement is that the hardware can exchange data freely between tiles. Each tile consists of a network adapter, a small tile-local memory (TLM) and multiple cores. For our design, tiles do not need to be homogeneous, i.e. they may contain differently sized TLMs and a varying number of cores. Each tiles has its own address space and cores can only access the respective TLM directly. If data from another TLM is required, a transfer between the two TLMs must be initiated. We assume that the hardware provides mechanisms to exchange small messages on the one hand and transfer large portions of data efficiently on the other. Our RTE further requires that there is one tile with direct access to a larger off-chip memory. To differentiate between this special tile and the remaining tiles, we call it the *driver tile* and all other tiles *compute tiles*.

The described hardware model was chosen to resemble a commercially available platform, the Kalray Massively Parallel Processor Array (both the first generation *Andey* [9] and the second generation *Bostan* [17]). The Kalray processor consists of 16 compute tiles. Each of these tiles contains a 2 MB TLM and 16 freely programmable cores running at a frequency between 400 MHz and 800 Mhz. Compute tiles are connected by two NoCs, a control NoC for messages, and a data NoC for larger chunks of data. The latter supports direct memory access (DMA) transfers, in which the network adapter itself successively transmits all bytes. In addition to the compute tiles, the NoCs are also connected to multiple I/O subsystems. In our bare-metal RTE implementation on the Kalray architecture, we use one of the I/O subsystems as the driver tile. I/O subsystems are special tiles with access to an external DDR3 memory and an ethernet controller. In contrast to compute tiles, I/O subsystems have fewer cores but a wider NoC interface for faster concurrent transfers between the DDR memory and TLMs.

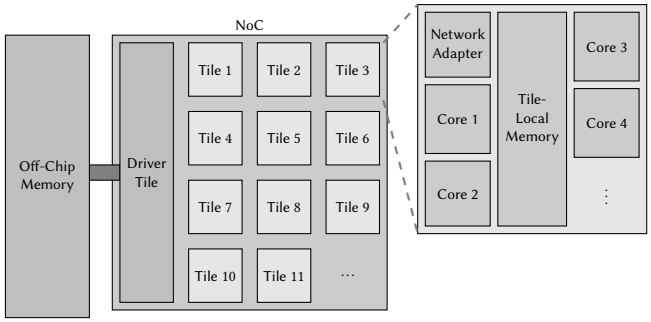

Figure 1: Abstract clustered hardware architecture

3 COARSE-GRAIN DATAFLOW MODEL

Our RTE executes programs modeled as directed acyclic graphs (DAGs). Whenever we use the term "graph" or "dataflow graph", we imply that these graphs are DAGs. In our dataflow model, DAGs are bipartite and consist of actor nodes and data nodes. Actor nodes (for more conciseness we will refer to them as *actors*) contain information about how data is processed, while data nodes are used to model memory requirements. Since our dataflow model is coarse-grain, data nodes usually refer to data collections or structures rather than single values, and actors represent potentially complex functions applied to the data. To match the behavior of functions, actor nodes have exactly one successor. Moreover, it is important that actors only refer to pure functions, i.e. functions which do not change the global state, and do not allocate heap memory. Therefore, all memory requirements (besides runtime stack memory) appear in the dataflow graph as data nodes, and the RTE is able to determine whether there is enough memory available on a tile before an actor is executed. This is one of the reasons for the explicit representation of data in our model. The second reason is that explicit data nodes are useful in identifying data across different TLMs. In shared-memory systems, data can be easily addressed by its memory address. However, in a clustered architecture where each tile has its own address space and the same data may be present in multiple memories, a different way to identify data is required. In our RTE, data node IDs are used for this purpose.

Graphs are supposed to be executed multiple times or repeatedly. To support repeated executions, graphs consist of different types of data nodes, in particular input, output, constant, and inner nodes. Input nodes do not have incoming edges and thus their data has to be specified for each graph execution. Data in constant nodes, on the other hand, cannot be replaced and thus stays the same in all dataflow executions. Data inside inner and output nodes is computed during graph execution. Both types of nodes can have

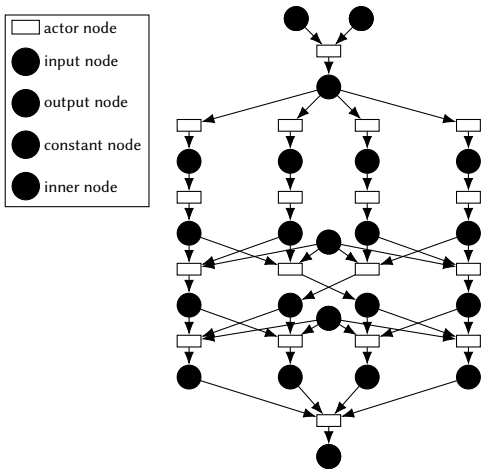

Figure 2: Example dataflow graph

incoming and outgoing edges. The reason for this distinction is that the data inside of output nodes is kept in memory so that it is still available when the graph execution is finished, while the content of an inner node is removed when the data is no longer required. An example graph with the different types of nodes is shown in Fig. 2. Circles represent data nodes, while actor nodes are drawn as white boxes. This graph is actually a DAG representing one of the benchmark applications we used to determine the performance of our RTE (see Section 6), more specifically the fast Fourier transform (FFT) application. It should be noted that we did not use this exact graph in our experiments since in this graph the input data is divided into only four parts which is not enough to utilize the sixteen compute clusters of the evaluation hardware.

Our execution model supports redundancy through actor duplication. Actors can be executed up to three times, and results are verified by special comparison actors. It is not necessary to manually specify redundancy in a graph. The RTE automatically constructs redundant actors, comparison actors, and the required additional data nodes. It is also possible to change the redundancy of graphs between two graph executions. In this case, the RTE rearranges the structure of the graph as it adds or removes redundant nodes for each actor as shown in Fig. 3.

4 RTE DESIGN

This sections describes the design of our RTE in a top-down fashion, beginning with the main procedure. The general steps to execute a graph are shown in Algorithm 1. Our RTE does not require that

Algorithm 1: Steps to execute a graph

```
1 g ← load DAG;
2 compute schedules for g;
3 do once, multiple times or repeatedly
4 |   assign data to g's input nodes;
5 |   execute g;
6 |   extract data from g's output nodes;
7 end
```

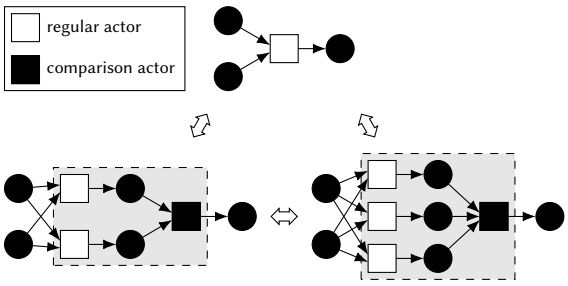

Figure 3: Changing the redundancy of an actor

all graphs are permanently available in the main memory. Instead, graphs can be loaded from an external source in a graph description format. Once a graph g has been imported, the RTE computes a set of schedules in such a manner that the graph can be executed in different redundancy configurations. As soon as each input node of g has been prepared, the graph execution is started. When the execution is finished, result data is extracted from the output nodes. All steps besides the actual graph execution involve only the driver tile. Algorithm 1 only shows one possible sequence. Since the RTE only provides a set of routines and the main procedure is user-defined, it is also possible to import multiple graphs and execute them alternately. Furthermore, result data of a graph execution can either be processed outside the RTE or used as the input of another graph execution (either of the same or a different graph).

A graph execution cannot be started while another graph is executed. Simultaneous graph execution is only possible by specifying a DAG consisting of multiple unconnected smaller DAGs. However, there is no guarantee that the DAGs are actually executed simultaneously. Should each DAG offer enough parallelism to utilize all tiles, the RTE might execute them sequentially.

4.1 Graph Import

As mentioned above, the RTE needs to import DAGs in a graph description format from an external source. The overall RTE design does not depend on a specific format. In our RTE implementation on the Kalray Massively Parallel Processor Array (MPPA), we use an extended DOT graph description format [10]. The extensions we introduced are limited to additional node and edge attributes which allow the RTE to distinguish between different kinds of nodes, determine memory requirements, and ensure the correct order of actor inputs. Functions applied by the actors are not imported along with the graph, but come with the binary. Therefore, RTE users either have to include the functions for all graphs in the binary or switch between different binaries at runtime.

The place graphs are imported from depends on the operation area. In our evaluation setting, the Kalray MPPA is embedded in a host system and thus the RTE imports graph from files in the host filesystem. If the Kalray MPPA is not embedded into a host system but is instead used standalone, there are other potential ways to import graphs. One possibility would be through the MPPA's ethernet interface. Further, it would be possible to include the graph description in the application binary. However, since the binary is loaded into the I/O subsystem's TLM, this works only for small graphs.

4.2 Scheduling

Because our execution model is based on the dataflow principle, actor executions are data-driven. However, the RTE has to decide which actor is executed on which tile and in which order the actors are executed. There are two kinds of approaches to schedule graphs depending on the time the scheduling takes place. The actor mapping and order can be either determined before (offline) or during (online) graph execution.

In order to support a broader variety of configurations and to explore the two different techniques, we examined one scheduling procedure for each approach. Because our dataflow model does not restrict the shape of graphs, computing an optimal schedule offline is NP-hard [21]. However, there are many DAG scheduling heuristics that cover a variety of hardware architectures and usage scenarios. The scheduling heuristic we think is most suitable for our dataflow approach is Heterogeneous Earliest Finish Time (HEFT) [20] due to its versatility and efficiency [7]. HEFT belongs to the category of list scheduling heuristics, which compute a schedule in two steps. The actors are first placed in a list whose order depends on the respective scheduling heuristic. HEFT uses a property called *upward rank* for this. Then, the scheduler iterates over the list and chooses a processing element for each actor based on a set of rules which are specific to the respective scheduler. In case of HEFT, the scheduler aims at minimizing the *earliest finish time* (EFT) for each actor. Due to the special driver tile, we had to extend the standard HEFT heuristic slightly. The exact modification is shown in Algorithm 2. Additional lines (compared to standard HEFT) are highlighted. Inside the main loop, a conditional expression checks whether the currently considered actor node has too many inputs or processes too much data due to compute tile memory restrictions. If at least one of the two conditions is true, the actor is mapped to the driver tile. Although the actor assignment does not depend on the earliest finish time (EFT) in this case, it must still be computed since the EFT of an actor depends on the EFT of its predecessors. It is easy to see that, despite of the modifications, the heuristic still produces valid schedules because the scheduling list is created and used like in standard HEFT.

The online scheduling technique we implemented in our RTE is based on work stealing. In this approach, each tile has its own double-ended queue. A queue contains all actors that are expected to be executed on the respective tile. If a queue is not empty, the tile extracts an actor at the front, executes it and inserts actors that became ready also at the front. Otherwise, the tile steals an actor from the back of the queue of another tile. There are multiple possible ways to determine from which other tile the actor is stolen. A very common approach and the one we took for our implementation is to choose the tile randomly [5]. At the beginning of a graph execution all actors that are ready to be executed are inserted to random queues. As with the HEFT heuristic, the work stealing procedure has to be modified so that all actors which have too many inputs or process too much data are placed in the driver tile's queue. Furthermore, compute tiles must not steal from the driver tile (and vice versa) and, whenever the driver tile executed an actor, subsequent actors should be inserted into compute tile queues whenever possible. In such cases, our implementation chooses the compute tile queues randomly. It should be noted that, since TLMs

Algorithm 2: Modified HEFT scheduling heuristic

```
1  function HEFT_WITH_DRIVER(g : graph)
2      compute mean values for all node and edge weights in g;
3      compute the upward rank for all nodes in g;
4      create a sorted list of nodes by nonincreasing order of upward
         ranks;
5      while the scheduling list is not empty do
6          let n be the first node in the list;
7          remove n from the list;
8          if n has too many inputs or n processes too much data then
9              compute the EFT of n on the driver tile;
10             assign n to the driver tile;
11         else
12             for each compute tile t do
13                 compute the EFT of n on tile t;
14             end
15             assign n to the tile with the lowest EFT;
16         end
17     end
18 end
```

are small and graphs are therefore only present in the off-chip memory, the tiles cannot maintain queues themselves. Instead, the driver manages all queues and executes the stealing procedure for all tiles.

4.3 Memory Management

Since all graphs and schedules are too large for TLMs, and therefore stored in the off-chip memory, the driver tile is responsible for the overall graph execution. For each compute tile, it maintains an array of data nodes which are present in the tile's local memory. To simplify the communication between the driver and compute tiles, the RTE uses a statically allocated amount of memory on the compute tiles to store data nodes. When the RTE is started, each compute tiles sends the address of this portion of memory to the driver tile. Furthermore, the data node memory is statically divided into fixed-size slots. This allows the driver to initiate data transfers to (or from) any slot without a complex communication protocol. Although a more complex dynamic memory management would allow the RTE to use the TLMs more efficiently, this would either require more messages or more computation on the driver (if the driver runs the allocation routine for all tiles in order to reduce the number of messages). In our RTE implementation on the Kalray Bostan MPPA, 1.25 MiB is reserved for data nodes. It is divided into 8 slots so that each slot is 160 KiB in size. The remaining 750 KiB contain the RTE code, functions for actor executions, and runtime stacks for the 16 cores on the tile. Since the off-chip memory has to be able to store larger portions of data, a simple static allocation similar to the TLMs is not feasible, and thus the driver tile uses a dynamic memory allocator to manage the off-chip memory.

4.4 Graph Execution, Driver Part

The driver has to fulfill different roles during graph execution. First, since only the driver has access to the graphs and offline schedules, it is the only tile that is able to determine whether an actor is ready,

i.e. all preceding actors are finished. However, before an actor can be sent to a compute tile, the driver has to ensure that the required data nodes are in the corresponding TLM. If this is not the case, the driver must either transfer the data node itself or tell the tile in which TLM the data is availabe so that the tile can start a data transfer between its own and the remote TLM. Besides transferring data and actors to compute tiles, the driver is responsible for the execution of actors that require large amounts of memory. It is possible that the driver has to transfer data from TLMs into the off-chip memory first. Lastly, the driver periodically checks whether a network event happened, i.e. whether a data transfer is finished or there is a message in its message queue. In both cases, the driver reacts by updating its information about the respective TLM.

4.5 Graph Execution, Compute Tile Part

The compute tile part of a graph execution is less complex than the driver part. Compute tiles do not act by themselves, but wait for messages from the driver tile. There are four different kinds of messages. A data node message tells the compute tile to update its data node array in which node properties like the data size and node ID are stored. The message also contains the information whether the actual data was already transferred by the driver or has to be fetched from another TLM. Other possible instructions from the driver are to clear a memory slot by removing the corresponding entry in the data node array or to clear the whole data node memory. The last kind of message is an actor message. When a compute tile receives a message of this type, the tile stores the result node's meta information in the node metadata array, executes the actor and sends a notification message to the driver.

4.6 Actor Execution

Since each tile (including the driver tile) contains multiple cores which are able to access the whole tile-local memory or, in case of the driver tile, the off-chip memory, actor functions should contain enough parallelism to utilize all cores on a tile. Parallelism should be specified in an abstract way, i.e. the RTE user should only define which parts of the function can be executed in parallel and not on which exact cores they are executed. Currently, our RTE implementation on the Kalray MPPA provides a parallel loop function which is similar to the parallel for-loop from OpenMP for this purpose. Fig. 4 shows an example. The actor applies a function f which doubles its input to all integers in the data node and the parallel loop function distributes loop iterations as evenly as possible across the three cores. This is of course a rather small example, and real hardware will likely provide more than three cores per tile. In case of the Kalray MPPA, there are 16 cores per compute tile.

4.7 Additional Remarks

To support a broader range of applications efficiently, our RTE provides three special types of actors (besides comparison actors) which are always executed on the driver. There is a special actor to extract a continuous part of data from a node. Actors of this type can be used to split a larger data node into smaller ones which fit into the memory slots on compute tiles. Similarly, there is another type of actor to collect multiple data nodes and store the data continuously to create a larger data node. Our RTE implementation

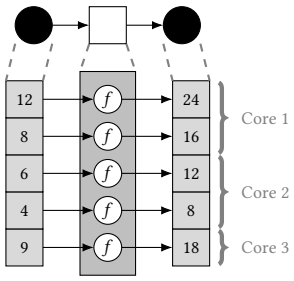

Figure 4: Actor applying $f(x) = 2x$ with three cores

on the Kalray platform optimizes the execution of these special actors as it only sets the respective pointers accordingly whenever possible instead of actually copying the data. A third special actor type is used to reorder the elements in a large data node. We use this actor in our benchmarks to prepare input data before splitting it into smaller chunks. In Fig. 2, a graph from our FFT benchmark, the topmost actor is such an actor. The actor has two inputs, an input node and a constant node. The purpose of the latter is to specify how the data elements are reordered.

5 REDUNDANCY

The previous sections described only standard graph executions, but our RTE also supports redundancy by duplicating actors. Graphs can be executed in five different configurations in total:

(1) **Non-redundant** execution
(2) **Two** redundant actor executions on the **same tile**
(3) **Three** redundant actor executions on the **same tile**
(4) **Two** redundant actor executions on **different tiles**
(5) **Three** redundant actor executions on **different tiles**

Our RTE is able to execute graphs in all five configurations with either offline schedules or work stealing. Both the HEFT scheduling heuristic and work stealing mechanism have to be extended in order to support all five configurations. The reason for this is configuration 4 and 5, i.e. redundancy across different tiles. In both configurations, the number of actors in the graphs differs from the non-redundant case and thus each one requires its own schedule. Redundant actor execution on the same tile does not require a modification in the scheduling process since the combination of two or three consecutive redundant executions and a comparison can be considered as one execution in the scheduling process. Therefore, non-redundant schedules and ordinary work stealing can be used.

5.1 Scheduling with Redundant Actors

We first want to focus on offline scheduling with the HEFT algorithm. As described above, the extension only has to consider redundancy across different tiles. Algorithm 3 shows the general procedure of our extended HEFT scheduling implementation for three redundant actor executions. Differences to standard HEFT are highlighted. The extended variant basically schedules redundant actors consecutively (line 9) and assigns them to different tiles (line 11). Comparison actors are not considered at all in the scheduling process since these actors are treated specially in our RTE (see Section 5.2). To compute a schedule with only two redundant

Algorithm 3: HEFT Scheduling with Redundancy

```
1  function HEFT_REDUNDANT(g : graph)
2      compute mean values for all node and edge weights in g;
3      compute the upward rank for all nodes in g;
4      create a sorted list of nodes by nonincreasing order of upward
       ranks ignoring all redundant and comparison nodes;
5      while the scheduling list is not empty do
6          let n be the first node in the list;
7          remove n from the list;
8          let n' and n'' be the redundant nodes of n;
9          for x in {n, n', n''} do
10             for each compute tile t do
11                 if n, n' or n'' is assigned to t then
12                     proceed with next tile;
13                 end
14                 compute the EFT of x on tile t;
15             end
16             assign x to the tile with the lowest EFT;
17         end
18     end
19 end
```

actor executions, removing n'' from the set in line 9 is sufficient. It should also be noted that Algorithm 3 omits actor executions on the driver tile. Our actual implementation for the Kalray architecture is a combination of Algorithm 2 and 3. Furthermore, our RTE currently does not support redundant actor execution on the driver. Our hardware model currently contains only one driver tile, and this tile would have to execute all redundant actors which is a potential bottleneck for the whole graph execution.

Similar to the HEFT scheduling heuristic, work stealing has to be slightly extended to support redundant actor executions on different tiles. Both the insertion and stealing have to be modified. By default, whenever a tile finishes an actor execution, all actors which became ready are inserted into the tile's queue. For redundant actor executions, this would likely lead to redundant actors being executed on the same tile. We extended the insertion routine so that the first actor is inserted as usual and the second and possibly third actor is inserted into a different queue. To choose suitable tiles, the insertion routine iterates over all data nodes processed by the actor, and creates a list of those tiles which have computed at least one of these nodes. The second and third actor are then assigned to those tiles from this list whose queues have the least amount of elements. Whenever there are multiple possible candidates, the routine chooses a random tile. Similarly, it chooses a random tile out of all available tiles in case the list is empty. As mentioned above, the stealing routine must also be altered. To ensure that a tile does not execute duplicate actors, the extended stealing routine must check whether the tile has not already executed a redundant actor and that no redundant actor is currently in its queue. Since the stealing routine only checks the back element of a queue, there is a chance that the described routine does not find an actor for stealing even though at least one of the queues contains a suitable actor. However, in contrast to standard work stealing, tiles do not only receive work by stealing, but also through the scheduling of redundant actors where queues with fewer elements are prioritized.

5.2 Redundant Graph Execution

Graph executions with redundant actor executions on the same tile only differ slightly from non-redundant executions. Compute tiles receive an actor from the driver, execute it either two or three times, and compare the results. In case all results differ, the tile can re-execute the actor on its own without contacting the driver. However, there is one minor caveat the driver tile must observe. Before sending an actor to a compute tile, the driver must ensure that there are enough free slots in the TLM to store the results of all redundant actor executions.

Redundant actor executions on different tiles, on the other hand, mainly affect the driver since it executes all comparison actors and may initiate actor re-executions in this configuration. As described in the previous section, pre-computed schedules do not contain comparison actors. Similarly, comparison actors are ignored in the work stealing process. Instead, whenever redundant actors are finished, the driver immediately executes the comparison actor. In order to reduce the amount of data transferred over the NoC and to reduce the amount of computation on the driver, compute tiles do not send back all the computed data. Instead, they send a checksum of the data along with the notification message about a finished actor. Thus, the driver tile only has to compare two or three checksums and, depending on the result, tell the compute tiles to re-compute the data or that the data is correct. An important aspect of our RTE is that the redundancy mechanism is not speculative. The driver only sends an actor to a tile after the data from preceding actors was verified. This ensures that the rollback after a fault is fast since only one actor has to be re-executed.

It should also be noted that our redundancy mechanism currently focuses mainly on transient faults, i.e. faults that do not occur repeatedly. A faulty core which consistently produces wrong results will cause the system to re-execute actors without progress. The only exception is the fifth configuration, i.e. three redundant executions on different tiles. This configuration can handle one faulty tile since for each wrong result two correct results are available and the incorrect data is discarded.

6 EVALUATION

As described in Section 2, the introduced hardware model was chosen so that we could implement our RTE design on the Kalray platform. Nonetheless, evaluating the performance of our dataflow RTE is rather difficult. To the best of our knowledge, there are only three frameworks for the Kalray MPPA which are similar to our RTE in the sense that they handle transfers between tiles automatically. These three frameworks are Kalray's implementation of the ΣC dataflow language [8], the UpScale SDK originating from the P-SOCRATES project [16], and Kalray's OpenCL implementation. Even though the dataflow concepts of ΣC are similar to our approach, the framework only supports the first generation of the Kalray MPPA and is not available for our second generation Kalray Bostan development board. The Upscale SDK, on the other hand, is available for the Kalray Bostan MPPA. Furthermore, it has some similarities to our approach since it is based on graphs and offline scheduling. However, the project seems to be no longer actively maintained and not compatible with newer versions of Kalray's

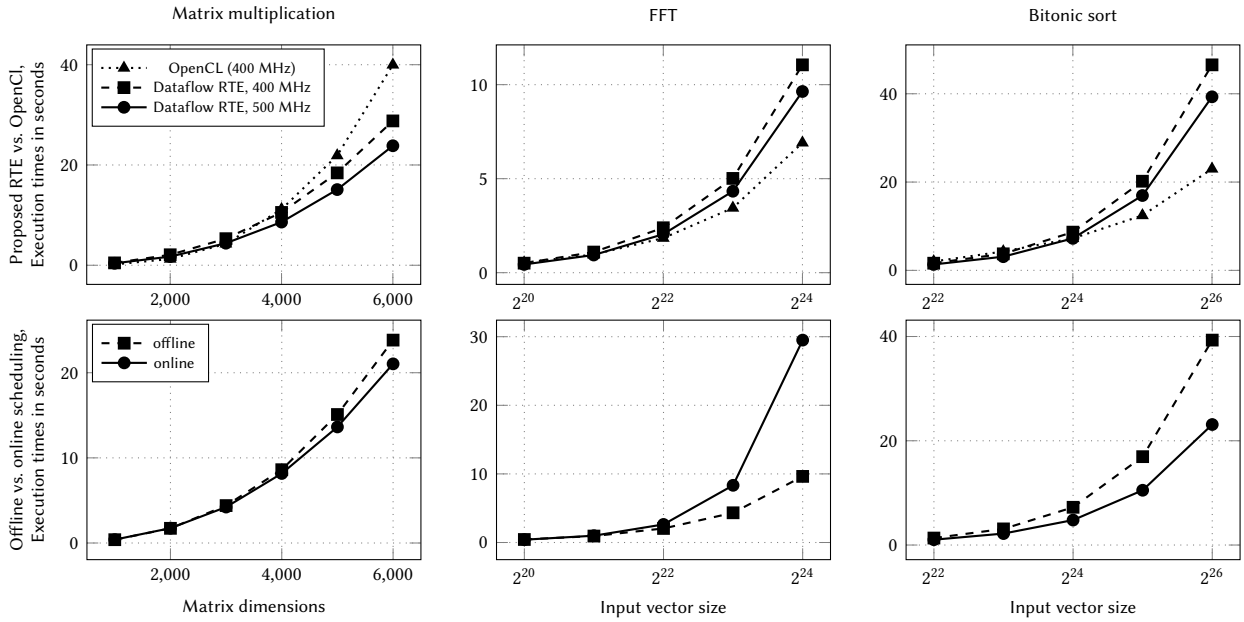

Figure 5: Execution times of dataflow and OpenCL programs

software development kit. Thus, we chose Kalray's OpenCL implementation as a reference for the performance of our RTE, although it differs the most from our approach and has a different focus. While Kalray's OpenCL framework targets high-performance computing, our RTE focuses on the execution of applications with different levels of redundancy.

6.1 Benchmarks and Evaluation Preliminary

Another difficulty in the evaluation is that there is, again to the best of our knowledge, no commonly used benchmark suite for the Kalray Bostan MPPA. OpenCL benchmarks usually target GPUs instead of NoC-based architectures like the Kalray MPPA. Therefore, we implemented three benchmark applications based on algorithms which are commonly used in the embedded systems domain as both dataflow graphs and Kalray OpenCL programs. The three algorithms are integer matrix multiplication, fast Fourier transform (FFT) and bitonic sort. There are multiple reasons why we chose these exact algorithms. First, each algorithm focuses on a different kind of instruction (integer arithmetic, floating point arithmetic and load/store instructions). Second, the three algorithms differ in their computational complexity. And third, for better comparability between the frameworks and configurations, all three algorithms provide enough parallelism to utilize all cores of the MPPA.

In favor of an easier implementation, we made some restrictions regarding the inputs of our benchmark applications. The matrix multiplication benchmark requires two square integer matrices, the FFT benchmarks an input vector of 2^n double-precision floating point complex numbers, and the bitonic sort benchmark an input vector of 2^n integers. Furthermore, to achieve the best performance, we specified the dataflow graphs for our RTE so that the size of each data node is as close to the size of memory slots as possible. Similarly, for the OpenCL applications, we arranged the data in a way that is advantageous for Kalray's OpenCL paging mechanism. For the OpenCL matrix multiplication benchmark, we enlarged the matrices slightly (e.g. from 2000×2000 to 2048×2048) and transposed the second matrix so that memory pages contain full rows or columns. The comparison between the matrix multiplication benchmarks is actually not completely fair since the OpenCL version uses the host system for the preparation of input matrices, while the dataflow benchmark prepares the input on the MPPA. In our experiments, the paging mechanism did not handle the matrix transposition very well, and we do not want to compare the dataflow matrix multiplication with a rather slow OpenCL matrix transposition.

When an application binary is transferred to the Kalray MPPA via the JTAG loader, it is possible to specify the clock frequency at which the program is executed through a command line option. It is often stated (for example in [17]) that the Kalray Bostan MPPA usually operates between 400 MHz and 800 MHz. However, the development board we used in the evaluation was not able to execute binaries at frequencies above 550 MHz, and thus we ran our RTE usually at 500 MHz. The Kalray manual does not explicitly state at which frequency OpenCL programs are executed (only that the maximum OpenCL frequency is 600 MHz). We assume that OpenCl programs are executed at the default frequency, i.e. 400 MHz. Therefore, we also executed the three dataflow benchmarks in the reference configuration at 400 MHz.

6.2 Graph Executions with Offline Schedules

Non-redundant graph executions with pre-computed schedules represent the baseline for all further evaluation results. Each benchmark was executed ten times for each input size, and the curves show the respective average values. We did not include variances

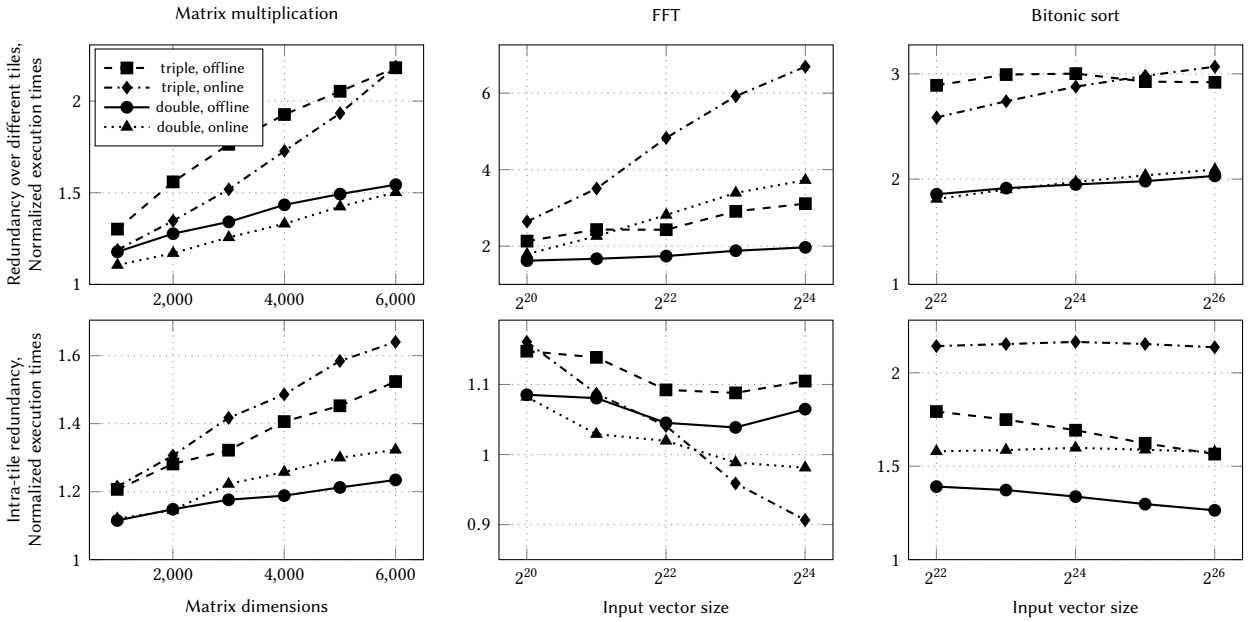

Figure 6: Execution times of redundant dataflow executions normalized to non-redundant executions

in the figure since execution times usually did not vary by a large amount, both for our RTE and Kalray's OpenCL framework. In the vast majority of experiments, the difference between the shortest and longest execution time we measured was below 5%. Execution times are shown in Fig. 5. The OpenCL benchmarks outperforms our dataflow RTE consistently in the FFT benchmark, for small matrices in the matrix multiplication benchmark, and for large input vectors in the bitonic sort benchmarks. There are multiple reasons for these performance differences, often related to our redundancy mechanism. One reason is that data in our execution model is immutable, and actors always create new data. This is an essential aspect of our redundancy mechanism and allows the RTE to re-execute actors in case of a faulty execution. The OpenCL FFT and bitonic sort benchmark applications, however, work in-place. Bitonic sort, for example, benefits from an in-place execution as it compares pairs of elements and only needs to swap them when they are in the wrong order. A second reason is that, in favor of easier rollbacks, dataflow actors in our model are limited to one output. This reduces the performance of the dataflow FFT benchmark since some computation is repeated in order to compute the left and right elements in the butterfly pattern. Another factor lies in the use of fixed schedules. Our HEFT implementation estimates execution times and data transfer times according to user-defined hints and the amount of processed or transferred data. Especially transfer times are difficult to predict accurately because the exact timing of transfers at runtime can vary, and it is thus difficult to determine offline which transfers will occur in parallel. Section 6.3 shows that using work stealing can lead to increased performance. In our opinion the performance drawbacks are not drastic and outweighed by the redundancy mechanisms our RTE offers.

When comparing the curves for the two different frequencies, it is noticeable that increasing the frequency by 25% leads to a reduction in the execution time by roughly 17% for the matrix multiplication, 13% for FFT and 15% for bitonic sort. The matrix multiplication benchmark benefits the most from an increased frequency since it is the most expensive algorithm out of the three with a complexity in $O(n^3)$. This higher overall complexity affects mainly individual actors instead of leading to overly large graphs. Thus, in comparison to the other algorithms the influence of actual computation on the overall execution time is higher.

6.3 Graph Executions with Online Scheduling

As described in previous sections, our RTE also supports graph executions with work stealing. To compare the performance of the two approaches, all three benchmark applications were additionally executed in online scheduling mode at a clock frequency of 500 MHz. Results are shown in Fig. 5. For matrix multiplication and bitonic sort, the execution times are lower when the RTE operates in online scheduling mode. In case of bitonic sort, the performance of our RTE is now much closer to the OpenCL benchmark. The FFT algorithm, however, shows that there are cases where a statically computed schedule leads to a better performance. Work stealing does not consider data dependencies and transfers which can have a large impact on the graph execution time, especially when actors are quite short like in the FFT benchmark. Furthermore, the number of stolen actors can also have an influence on the performance because it is very likely that additional data transfers are required when an actor is stolen from a random tile. Table 1 shows how many actors are stolen on average for the three benchmark applications with various input sizes. The table also puts them in relation with the respective total number of actors in the graph (excluding the redundant actors created by the RTE). It is noticeable that the percentage of stolen actors decreases for larger (wider and deeper) graphs. For the matrix

Table 1: Number of stolen actors for the three benchmarks

	Input Size	Actors in Total	Average Stolen	Percent
MatMul	1000^2	178	10.4	5.8%
	2000^2	1203	31.9	2.7%
	3000^2	3828	48.8	1.3%
	4000^2	8803	65.5	0.7%
	5000^2	16878	82.7	0.5%
	6000^2	28803	91.6	0.3%
FFT	2^{20}	1154	331.8	28.8%
	2^{21}	2562	459.0	17.9%
	2^{22}	5634	674.2	12.0%
	2^{23}	12290	1050.1	8.5%
	2^{24}	26626	1382.5	5.2%
Bitonic	2^{22}	4737	1339.5	28.3%
	2^{23}	11777	1812.3	15.4%
	2^{24}	28673	2317.6	8.1%
	2^{25}	68609	2987.7	4.4%
	2^{26}	161793	4166.8	2.6%

multiplication benchmark, the relative number is the lowest, with only 0.3% of all actors being stolen for the largest graph, while for the FFT benchmark it is highest, at 5.2% for the largest graph.

6.4 Redundant Execution on Different Tiles

Execution times for redundant graph executions on different tiles are shown in Fig. 6. All benchmarks were run at 500 MHz, and we normalized the measurements to the respective non-redundant execution. As described in Section 5.1, due to fact that actors on the driver are not executed redundantly, the execution times can be lower than twice or three times the baseline execution time. This can be observed for the matrix multiplication (and FFT with offline schedules for small input sizes) since the driver tile rearranges the data in this benchmark in order to be able to transfer contiguous portions of memory via DMAs. For larger input matrices, the normalized execution times are increasing. The reason for this is that the actions performed by compute tiles have a higher time complexity on the large scale and thus the influence of driver actor executions is less significant for larger inputs. Multiplying two matrices has a cubic time complexity, while rearranging the data has a linear time complexity. Bitonic sort does not require any memory rearrangement, and thus it is not surprising that executing the graph redundantly takes about twice (or three times for triple execution) the baseline execution time. For FFT with online scheduling, execution times are excessively long, most likely because the work stealing routine performs poorly for actors with short execution times as it does not try minimize the number of transfers.

6.5 Redundant Execution on the Same Tile

Redundant actor execution on the tile is beneficial for the overall execution time since the number of transfers is lower compared to redundancy across different tiles. However, in comparison to the non-redundant execution, the number of transfers might be higher in some cases. The reason for this is that a redundant actor execution on a tile requires multiple memory slots for the results, and hence more data node displacements are required. The results shown in Fig. 6 reveal that a significant portion of the execution time comes from data transfers because execution times are very low compared to standard graph executions. FFT is an extreme case where redundant execution increases the execution time only by 15% at most when offline schedules are used. Because almost all transfers involve the I/O subsystem, it is likely that the preparation of data and transfers on the I/O cores is a bottleneck. Another possibility is that the I/O subsystem's NoC interface or the connection to the DDR memory limits the performance. It seems that transfers do not only have a large influence on the execution times in our RTE, but also in OpenCL since otherwise the performance difference between the two frameworks would be higher. In case of FFT with online scheduling, the situation is even more extreme as redundant executions take less time than non-redundant executions for larger input sizes. In analogy to the results from the previous section, this hints that the work stealing routine performs poorly for actors with a short execution time because redundant actors are executed consecutively.

7 RELATED WORK

Graph-based program representations and concepts similar to our approach are used in a variety of frameworks, for example in dataflow system like the Delaware Adaptive Run-Time System (DARTS) [18], the Concurrent Collections (CnC) language family [6], and ΣC for the Kalray MPPA [8, 13]. Big data and machine learning frameworks like Apache Spark [25], Apache Flink which originated from the Stratosphere project [2] or TensorFlow [1] are based on similar concepts. There is also literature about fault-tolerant dataflow systems, for example [24], [3] and [14]. Although it is conceptually similar to our approach and targets a very similar hardware architecture, the redundancy mechanism described in [24] is more of a hardware redundancy approach since it relies on special hardware units on the chip. The methods from [3] and [14] on the other hand mainly target large high-performance architectures, i.e. workstations or server clusters. However, since the redundancy mechanisms are software based and similar to our approach, we briefly highlight them in the following.

The authors of [3] describe a software-based approach for fault-tolerant dataflow called *Dataflow Error Recovery* (DFER). Like in many dataflow fault tolerance approaches, redundancy is established by adding redundant nodes to the dataflow graph. Additional commit nodes are responsible for comparing the results of redundant executions. If the system detects that a computation produced wrong results, the respective graph node has to be re-executed. The execution of nodes in DFER is speculative, i.e. subsequent nodes are executed with possibly uncommitted values. In case of a fault, this can lead to a domino effect that makes the recovery more difficult. Since the latency caused by this domino effect might be undesirable in some areas, DFER provides the possibility to insert additional edges between a commit node and all nodes that process the corresponding data. These additional dependencies cause subsequent nodes to wait until the required data has been compared.

In [14], two different fault tolerance approaches for the *Kernel for Adaptive, Asynchronous Parallel Interface* (KAAPI) [11] are proposed. Dataflow graphs in KAAPI are not static, but instead built dynamically during their execution. Consequently, KAAPI uses online scheduling in form of work stealing. One of the fault tolerance

techniques described in [14] is *Systematic Event Logging* (SEL). The idea behind SEL is to log all modifications of the dataflow graph during runtime, i.e. all additions and deletions of graph nodes. When a processor fails, the log is used to rebuild the associated subgraph from the checkpoint file. Since checkpoints are created on each graph modification, this approach allows the system to re-execute single dataflow tasks.

The other fault tolerance technique described in [14] is *Theft-Induced Checkpointing* (TIC). In TIC, checkpoints are created periodically and when a task is stolen. The latter are called forced checkpoints. Only the (possibly virtual) processor from which a task was stolen creates such a checkpoint. Normal checkpoints, on the other hand, are created by all processors periodically. Recovery in TIC is similar to SEL, but since not every graph modification is logged there may be more task re-execution required than in SEL.

8 CONCLUSION

This paper presented a dataflow model and matching RTE for NoC-based manycore architectures. The RTE executes applications in form of DAGs either following pre-computed schedules or with work stealing. Regardless of the scheduling method, the graphs can be executed in five different configurations with varying degrees of redundancy. Changing the redundancy configuration between graph executions is also possible. We implemented our RTE design on the Kalray Bostan MPPA and evaluated all supported redundancy configurations and scheduling methods for three common computing tasks. Since there is, to the best of our knowledge, no other framework for the Kalray MPPA following a similar approach and providing a similar degree of flexibility with regards to redundancy, we compared the performance of our RTE to Kalray's OpenCL framework despite of the differences. The experiments show that, even though the main focus of our design was not on high performance, our RTE is able to outperform or at least reach Kalray's OpenCL framework in two of the three benchmarks.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal* 23, 6 (Dec. 2014), 939–964. https://doi.org/10.1007/s00778-014-0357-y

[3] Tiago A. O. Alves, Sandip Kundu, Leandro A. J. Marzulo, and Felipe M. G. França. 2014. Online Error Detection and Recovery in Dataflow Execution. In *20th International On-Line Testing Symposium (IOLTS)*. IEEE Computer Society, 99–104. https://doi.org/10.1109/IOLTS.2014.6873679

[4] Matthias Becker, Dakshina Dasari, Vincent Nélis, Moris Behnam, Luís Miguel Pinho, and Thomas Nolte. 2015. Investigation on AUTOSAR-Compliant Solutions for Many-Core Architectures. In *2015 Euromicro Conference on Digital System Design*. 95–103. https://doi.org/10.1109/DSD.2015.63

[5] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. https://doi.org/10.1145/324133.324234

[6] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. 2010. Concurrent Collections. *Scientific Programmming* 18, 3–4 (Aug. 2010), 203–217. https://doi.org/10.3233/SPR-2011-0305

[7] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou, and Wei Zheng. 2008. *Comparative Evaluation Of The Robustness Of DAG Scheduling Heuristics*. Springer US, Boston, MA, 73–84. https://doi.org/10.1007/978-0-387-09457-1_7

[8] Benôit Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benôit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel. 2013. A clustered manycore processor architecture for embedded and accelerated applications. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2013.6670342

[9] Benoît D. de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. 2014. Time-critical computing on a single-chip massively parallel processor. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–6. https://doi.org/10.7873/DATE.2014.110

[10] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. 2015. Drawing graphs with dot. https://www.graphviz.org/pdf/dotguide.pdf

[11] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. 2007. KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multiprocessors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation (PASCO '07)*. Association for Computing Machinery, New York, NY, USA, 15–23. https://doi.org/10.1145/1278177.1278182

[12] André Göbel and Denis Claraz. 2018. A Multi-Core Basic Software as Key Enabler of Application Software Distribution. In *ERTS 2018 (9th European Congress on Embedded Real Time Software and Systems)*. Toulouse, France. https://hal.archives-ouvertes.fr/hal-02156255

[13] Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. 2011. ΣC: A Programming Model and Language for Embedded Manycores. In *Algorithms and Architectures for Parallel Processing*. Springer, Berlin, Heidelberg, 385–394.

[14] Samir Jafar, Thierry Gautier, Axel Krings, and Jean-Louis Roch. 2005. A Checkpoint/Recovery Model for Heterogeneous Dataflow Computations Using Work-Stealing. In *Euro-Par 2005 Parallel Processing*, José C. Cunha and Pedro D. Medeiros (Eds.). Springer Berlin Heidelberg, 675–684. https://doi.org/10.1007/11549468_74

[15] Andre Luckow, Matthew Cook, Nathan Ashcraft, Edwin Weill, Emil Djerekarov, and Bennie Vorster. 2016. Deep learning in the automotive industry: Applications and tools. In *IEEE International Conference on Big Data (Big Data)*. 3759–3768. https://doi.org/10.1109/BigData.2016.7841045

[16] Luis Miguel Pinho, Eduardo Quiñones, Marko Bertogna, Andrea Marongiu, Jorge Pereira Carlos, Claudio Scordino, and Michele Ramponi. 2014. P-SOCRATES: A Parallel Software Framework for Time-Critical Many-Core Systems. In *17th Euromicro Conference on Digital System Design*. 214–221. https://doi.org/10.1109/DSD.2014.94

[17] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît D. de Dinechin. 2015. The shift to multicores in real-time and safety-critical systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 220–229. https://doi.org/10.1109/CODESISSS.2015.7331385

[18] Joshua Suettlerlein, Stéphane Zuckerman, and Guang R. Gao. 2013. An Implementation of the Codelet Model. In *Euro-Par 2013 Parallel Processing*, Felix Wolf, Bernd Mohr, and Dieter an Mey (Eds.). Springer Berlin Heidelberg, 633–644. https://doi.org/10.1007/978-3-642-40047-6_63

[19] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, and Gagandeep S. Sachdev. 2020. Compute Solution for Tesla's Full Self-Driving Computer. *IEEE Micro* 40, 2 (2020), 25–35. https://doi.org/10.1109/MM.2020.2975764

[20] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (2002), 260–274. https://doi.org/10.1109/71.993206

[21] Jeffrey David Ullman. 1975. NP-complete scheduling problems. *J. Comput. System Sci.* 10, 3 (1975), 384–393. https://doi.org/10.1016/S0022-0000(75)80008-0

[22] Moisés Urbina and Roman Obermaisser. 2017. Efficient Multi-core AUTOSAR-Platform Based on an Input/Output Gateway Core. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 157–166. https://doi.org/10.1109/PDP.2017.85

[23] Hrvoje Vdovic, Jurica Babic, and Vedran Podobnik. 2019. Automotive Software in Connected and Autonomous Electric Vehicles: A Review. *IEEE Access* 7 (2019), 166365–166379. https://doi.org/10.1109/ACCESS.2019.2953568

[24] Sebastian Weis, Arne Garbade, Bernhard Fechner, Avi Mendelson, Roberto Giorgi, and Theo Ungerer. 2016. Architectural Support for Fault Tolerance in a Teradevice Dataflow System. *International Journal of Parallel Programming* 44, 2 (1 April 2016), 208–232. https://doi.org/10.1007/s10766-014-0312-y

[25] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, USA, 15–28.