Sergey Ostroumov | Pontus Boström | Marina Waldén | Mikko Huova

Deriving Efficient and Dependable Parallel Programs from Simulink Models

Turku Centre *for* Computer Science

TUCS Technical Report
No 1111, May 2014

Deriving Efficient and Dependable Parallel Programs from Simulink Models

Sergey Ostroumov
TUCS – Turku Centre for Computer Science
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5A, 20520, Turku, Finland
Sergey.Ostroumov@abo.fi
Pontus Boström
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5A, 20520, Turku, Finland
Pontus.Bostrom@abo.fi
Marina Waldén
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5A, 20520, Turku, Finland
Marina.Walden@abo.fi
Mikko Huova
Tampere University of Technology,
Department of Intelligent Hydraulics and Automation
Korkeakoulunkatu 10, 33720, Tampere, Finland
Mikko.Huova@tut.fi

TUCS Technical Report
No 1111, May 2014

Abstract

Modern highly-demanding applications require high computational power, on the one hand, and fulfilment of real-time constraints and a high level of resilience, on the other. Model-based development provides means to address these objectives. Simulink is one such widely used tool for model-based development of software. It supports a complete design chain starting from modelling and ending in code generation. However, the programs generated by the built-in code generator cannot fully utilize the potential offered by energy and performance efficient many-core platforms that are likely to be used in various application domains: from home use electronics to complex critical control systems. Furthermore, the failure of a processing core where the derived implementation runs leads to the failure of the application. In contrast, the distribution of tasks among processing cores allows for invoking dynamic reconfiguration procedures, e.g., tasks reallocation, so that the required level of resilience is achieved and efficiency is maintained at an acceptable level. Despite this, however, the application tasks may lose data when they are dynamically reallocated to mask failures of individual cores. This paper addresses the problems described above by proposing: 1) an approach to generation of parallel implementations from Simulink models, 2) relying on 1), a mechanism that prevents data loss when application tasks are reallocated. The paper also presents evaluation results of the proposed approach on an industrial case study using a commercially available NoC-based platform, namely TilePro by Tilera. The evaluation shows only about 1% performance degradation when comparing an FT implementation with a non-FT one.

Keywords: Data Loss Prevention; Dynamic Reconfiguration; Many-Core Platforms; Parallel Programs; Resilience; Simulink

TUCS Laboratory
RITES – Resilient IT Infrastructures
Distributed Systems Laboratory
Integrated Design of Quality Systems group

The work was done within the Digihybrid project in the EFFIMA program coordinated by FIMECC.

1. Introduction

Due to a highly dynamic nature of modern embedded applications, their computational and communicational intensity is very high. On the other hand, they demand a new design paradigm in order to meet real-time constraints and a high level of resilience. To abstract away implementation details and focus on functionality of a system under development, designers typically employ various modelling techniques. One such widely used technique is modelling within the Simulink model-based design environment [1]. Simulink supports a complete design chain starting from modelling and simulation and ending in generation of, for example, C code. However, the generated programs cannot fully utilize computational and communicational power offered by energy-efficient many-core platforms.

A Network-On-Chip (NoC) which represents a communication network of cores has been proposed as a scalable paradigm that can provide high computational power fulfilling timing constraints and low power consumption [2]. There are commercially available platforms such as TilePro by Tilera [3] and Intel Single Cloud Chip **Error! Reference source not found.** that employ NoC. The programs are deployed on a platform using mapping algorithms (e.g., [5][6]) in order to achieve various objectives such as to minimize power consumption while providing high level of performance. However, high level of on-chip integration increases the probability of various faults [7] whilst high computational load may cause creation of hotspots leading to thermal problems [8][9]. Additionally, radiation which is frequent in space, but becomes an issue at the ground level as well can cause transient faults [10]. This can eventually induce a faulty execution of applications. One of the powerful techniques to tolerate these faults is dynamic reconfiguration, namely tasks reallocation [7][11][12]. This technique can be executed by agents that are integrated into the platform and perform efficient management without overloading the platform with monitoring and recovering activities [12][13][14]. However, when tasks are reallocated to non-faulty cores, they may lose data in the process, which can lead to the production of an erroneous output. Consequently, to achieve resilience, application tasks need to adopt a mechanism that provides means to continue execution without losing data when they are reallocated.

This paper proposes: 1) an approach to the derivation of parallel implementations from Simulink models, 2) relying on 1) a resilience mechanism that prevents applications from data loss when the application tasks are dynamically reallocated. To support the proposed approaches, we present performance evaluation results obtained using the TilePro platform [3].

The remainder of the paper is organized as follows. Section 2 reviews the related work. Section 3 provides preliminaries for the proposed approaches. Section 4 introduces the mapping between Simulink models and application characteristic graphs as well as describes generic functionality of tasks. Section 5 discusses the resilience mechanism for the platform and applications management relying on the mapping in Section 4. Section 6 describes a simplified version of an industrial case study modelled within the Simulink design environment. Section 7 illustrates the performance evaluation results for the derived implementations as well as performance assessment of the task reallocation procedure. Section 8 concludes the paper and outlines the directions of our future work. Finally, Appendices A-E demonstrate implementation details in terms of the derived C code.

2. Related Work

A Simulink model is a hierarchical dataflow diagram from which the Simulink design environment can generate sequential or multi-task C code scheduled according to the rate monotonic principle [15]. Natale [16] has proposed an algorithm that allows multi-task implementations of Simulink models to run on a single-core processor and aims at optimization of performance of the applications in terms of space (memory) and time. The algorithm analyzes cases where additional buffering can be avoided. However, the generated code is not aimed at parallel execution on a many-core platform. Additionally, automatic parallelization of the sequential code is still an open issue [17]. On the other hand, if the whole application is running on a single core and that core fails, the application may need to be restarted, i.e., to start execution from the very beginning. This may lead to data loss and missed deadlines.

In contrast to [15][16], we propose to generate a parallel implementation directly from a Simulink model by using application characteristic graphs (ACG) [5] as an intermediate step. The use of ACG allows designers to employ mapping algorithms for many-core platforms considering various optimization objectives, e.g., performance (real-time constraints) [5] and/or power consumption [18], resilience [7][12] etc. The generated concurrent code preserves the semantics of Simulink models. Moreover, the division of the system into parallel tasks enables the application of resilience mechanisms to tasks and, hence, improves the utilization of the platform.

After a parallel implementation is derived, the application should produce the expected result, i.e., satisfy reliability requirements. One technique to fulfil these requirements is to use redundancy, in particular, spare processing resources. There are several works addressing this issue. For example, Bolchini, Carminati and Miele [10] assume forked data parallel programs and propose to replicate a whole application or some of its threads in order to detect and tolerate failures of processors. The authors consider three duplication techniques: 1) duplication with comparison, 2) triplication and 3) duplication with comparison and re-execution FT. The authors propose an adaptation engine that acts according to the evolving environment. They consider several parameters which the adaptation engine needs to take into account. The adaptation engine incorporates observe-decide-act loop in order to achieve adaptability to faults.

Pinello, Carloni and Sangiovanni-Vincentelli have proposed another approach to replicating dataflow actors [19]. The authors consider a fault model, in which components are fail-silent, i.e., they either produce a correct result or produce no result. To effectively detect failures, the authors rely on failure patterns proposed in [20]. These patterns describe a set of vertices of a process graph that may fail within the same iteration. The authors use software replication for critical tasks statically at design time, where each replica is then executed on a separate control unit. Using this technique, the authors describe a fault-tolerant data flow.

An approach to tackle hardware failures in process networks has been proposed by Ceponis, Kazanavicius and Mikuckas [21]. The authors present an extension of Kahn process networks, namely Error-Proof Process Network (EPPN). They give operational semantics of EPPN in the form of labelled transition system, where concurrent nodes communicate via first-in-first-out (FIFO) channels. The nodes can check whether the channels are full or empty and can proceed to blocking write or read, respectively. The authors show a dynamic reconfiguration mechanism where actions of a faulty node are transferred to an adjacent non-faulty functional node and the communication channels are adjusted accordingly using checks on FIFO channels. According to the authors, this mechanism may make the network non-deterministic whilst enabling further execution of the nodes and helping in synchronizing data. When functionality of a failed node is delegated to a non-faulty operating node, data loss occurs. To tackle this problem, the authors introduce the

default value. Although the mechanism appears to fulfil continuous and on-time result delivery, the default value may not preserve semantics of the original application.

Similarly as in [10][19][21], we consider hardware failures of processing units in the underlying many-core platform. However, in contrast to [10][19], we rely on dynamic reconfiguration of the platform, namely tasks reallocation [7][11][12], which can be effectively and efficiently performed by agents integrated into the platform [12][13][14]. The tasks reallocation enables uninterruptable execution of applications [7][11] and avoids resource wasting caused by duplicating applications or threads (actors) in contrast to [10][19]. We adopt a mechanism proposed in [12], where spare processing units are used in case of failures of operating nodes. Similarly to [10], the approach in [12] also allows the reuse of cores from which tasks have been reallocated when no failure of such a core is detected anymore so that the task can be returned to its original location. We also take into account the fact that when tasks are reallocated due to failures of processing units, they may lose data. To address this problem, we propose a resilience mechanism, in which the reallocated tasks operate on the current values instead of the default ones in contrast to [21]. Therefore, the determinism of the application is preserved. Additionally, our approach is not restricted to data parallel applications, but can also be applied to functionally parallel ones in contrast to [10].

3. Preliminaries

3.1. Simulink Models

A Simulink model is a hierarchical dataflow diagram [1]. The model consists of a collection of functional blocks that have in-ports (inputs) and out-ports (outputs) allowing connections between blocks via typed signals. The blocks may have parameters that are initialized at the beginning of the execution and remain constant during the execution. Moreover, the blocks can contain memory. In this case, the value of the outputs depends not only on the inputs, but also on the previously computed value.

The blocks can be grouped into sub-systems. There are two types of sub-systems in Simulink: virtual and atomic [22]. Virtual sub-systems are used for the structural purpose only and do not affect the model execution. They can be seen as containers for functional blocks which the Simulink engine expands in place before execution. Atomic sub-systems are treated as single atomic units.

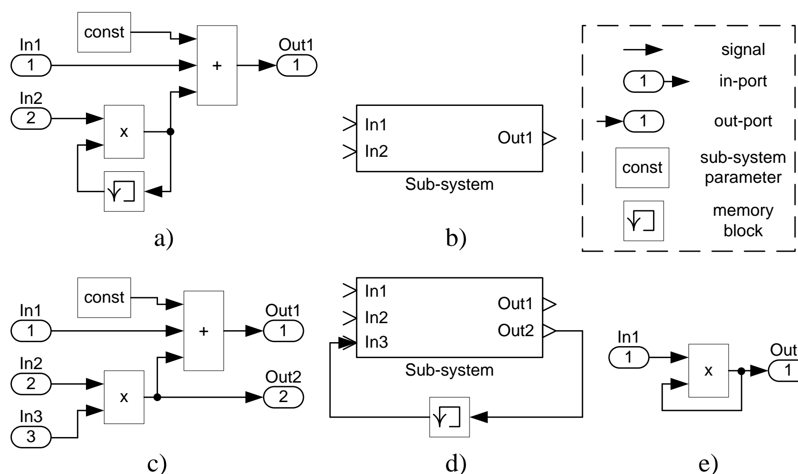


Figure 1: Simulink models:
a) sub-system content with memory, b) sub-system block without memory,
c) sub-system content without memory, d) sub-system block with memory, e) algebraic loop

Fig. 1 illustrates an example of a Simulink model. The model in Fig. 1, a) contains two in-ports and one out-port. It includes a constant parameter as well as a memory block (i.e., the model is stateful). The presented sub-system computes the function

$$\text{Out1} = \text{const} + \text{In1} + (\text{In2}*\text{mem});\ \text{mem}^{+1} = \text{mem} * \text{In2},$$

where mem is the current value stored in the memory and mem^{+1} is the next value to be stored in the memory. This model is grouped into a sub-system presented in Fig. 1, b). The sub-system in Fig. 1, c) represents the same sub-system as in Fig. 1, a), where the memory block is extracted outside of the sub-system as shown in Fig. 1, d). The sub-system in Fig. 1, c) is then stateless.

The models can be continuous or discrete [22]. Each block in a discrete-time model is evaluated at regular intervals with a specified sampling period. Generally, a Simulink block can be represented by the differential algebraic equation (DAE) [23]:

$$\begin{aligned} y(k) &= f(c, x(k), u(k)) \\ x(k+1) &= g(c, x(k), u(k)) \end{aligned} \tag{1}$$

where y is a list of out-ports, u is a list of in-ports, c are the sub-system parameters and x stands for an internal memory (state vector). The function f updates out-ports y at sample k while the function g updates the state x.

A Simulink model may also contain algebraic loops [24]. An algebraic loop stands for the case where the left side of equation (1) contains u. In other words, an algebraic loop describes a connection between an in-port and an out-port of the same block such that the out-port drives the in-port directly (Fig. 1, e)) or via feed-through blocks. A feed-through block is a block whose in-port directly controls out-port. The Simulink solver does not solve DAEs directly, but numerically determines the values at each simulation step. This means that the solver may not be able to always solve an algebraic loop. Consequently, the main problem caused by algebraic loops is that the code cannot be generated from a model containing them [24].

To effectively generate a parallel implementation, we consider discrete-time models with atomic sub-systems that specify periodic real-time systems. We assume that the model is single-rate, i.e., all its sub-systems fire at the same time intervals. Furthermore, we assume that the model from which the code is generated is causal, i.e., it does not contain algebraic loops.

3.2. Communication Platform

The generation of a parallel code requires designers to take into account characteristics of the underlying platform. A 2D mesh NoC-based many-core platform is considered typical for parallel applications [5][6][7][13]. It consists of tiles that include processing units (PUs) and routers (RTs) [2] (see Fig. 2). RTs allow communication between tiles by routing packets. The communication mechanism usually employs FIFO buffers [25][26], which preserves the flow order of data. Moreover, the platform typically supports checks whether the buffers are full or empty. Therefore, the tasks can read packets as soon as they arrive in the input buffers and send processed data when there is an available space.

In this paper, we consider a 2D mesh NoC-based many-core platform as well. We assume the platform to be homogenous at the global level, i.e., all tiles are identical, while their internal structure might be heterogeneous. We further assume that the routers employ deterministic routing with the dead-lock and live-lock free algorithm, which provides low latency and suits real-time control systems [27][28].

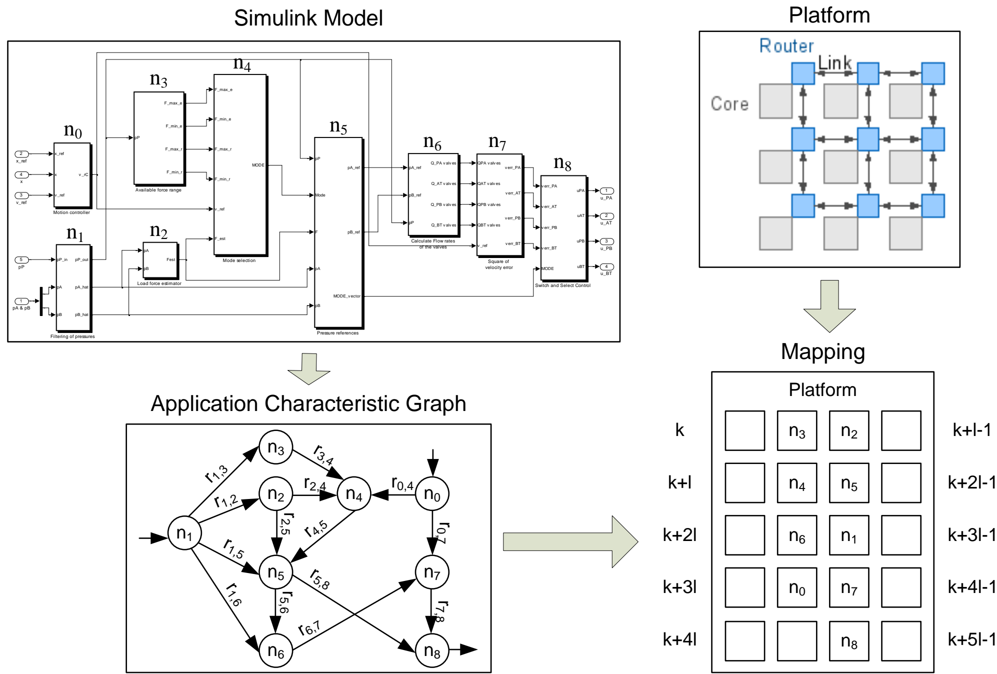


Figure 2: Application characterization graph and mapping example

4. Derivation of Parallel Programs from Simulink Models

We translate a Simulink model into a set of concurrent tasks that are given by the sub-systems and communicate according to the signals in between. This process can be summarized as the following algorithmic steps (we focus on steps 2-4 in the succeeding sections):

1. Flatten the model, where the top-level atomic sub-systems reflect tasks according to the designer choice.
2. Construct the application characteristic graph (ACG) from the flattened model using the mapping proposed in the sub-section 4.1.
3. Generate threads for the tasks and communication according to ACG (Appendices A-D).
4. Generate the main thread that will create the necessary environment (Appendix E).
5. Apply a mapping algorithm (e.g., [5][7][12]) using the ACG.

4.1. Construction of ACG from Simulink

To apply mapping algorithms that enable optimization in terms of, e.g., performance and power consumption [18] or resilience [7][12], we need to construct an Application Characteristic Graph (ACG) from a flattened Simulink model. An ACG consists of tasks and edges, where the edges show communication rates r between tasks via FIFOs. The construction of an ACG from an arbitrary model is illustrated in Fig. 2.

Let us now show the formal mapping between Simulink and ACG. A Simulink model is formally defined as a directed acyclic graph $G_{sm} = (N, E, S, \wp, \partial)$, where:

- the set $N = \{n_i \mid i \in 0..m\text{-}1\}$ contains atomic functional blocks (nodes) numbered from 0 to m-1 and m is the total number of blocks,

- the set $E = \{e_i \mid i \in 0..k-1\}$ represents links between those nodes and k is the total number of links,
- the set $S = \{s_i \mid i \in 0..m-1\}$ includes sampling rates of the atomic blocks,
- the function $\wp : N \rightarrow S$ assigns sampling periods to the nodes,
- the function $\partial : N \times E \rightarrow N$ specifies the next node by the current node and a link between the nodes.

An ACG, in its turn, is a tuple $ACG = (V, T)$ [5], where:

- the set of vertices V specifies clusters of tasks such that each $v \in V$ should run on a separate core,
- the set $T : V \times V \times R$ denotes directed edges showing the communication dependencies and rates $r_{i,j} \in R$ bits per time unit between those tasks.

Similarly as in the approach proposed by Boström [23], we interpret each node of G_{sm} as a vertex of ACG with synchronous dataflow semantics, i.e., each atomic sub-system as a separate execution task that can be run on a single core. However, in contrast to [23], we group the links of G_{sm} into edges of ACG. An edge between an arbitrary pair of nodes n_i, n_j in ACG reflects a group of links between the same nodes in G_{sm}. In essence, the links constitute communication between the nodes. The rates of packets are computed according to the function $\wp$. The input and the output signals of the blocks that interact with the environment do not participate in the construction of ACG. This is because these signals do not affect the application internal structure. The formal definition of the mapping function between G_{sm} and ACG is as follows:

- $V = N$,
- $\forall n_i, n_j \in V \,.\, \neg n_i = n_j \Rightarrow T = \{t \mid \exists E' \,.\, E' = \{e \mid \partial(n_i,e)=n_j\} \wedge t = (n_i, n_j, sizeof(E')/\wp(n_i))\}$,

where $sizeof(E') = \sum sizeof(e_k),\ k \in 1..card(E'),\ e_k \in E'$, denotes the total size of communication in bits determined by the types of the signals.

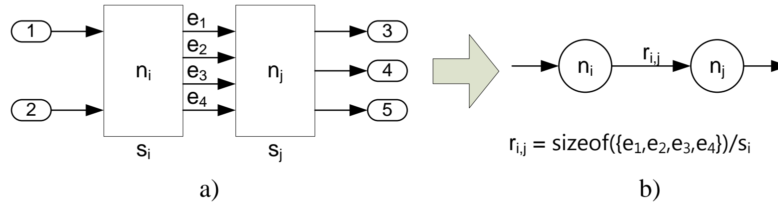


Figure 3: Constructing ACG from G_{sm}: a) Simulink model, b) ACG

Fig. 3 illustrates the mapping by using a small example. There, we have two blocks n_i and n_j without memory and with sampling rates s_i and s_j, respectively (Fig. 3, a)). There are nine signals, four of which connect the blocks. This model is then a tuple $G_{sm} = (N, E, S, \wp, \partial)$, where:

- $N = \{n_i, n_j\}$,
- $E = \{e_1,e_2,e_3,e_4\}$,
- $S = \{s_i, s_j\}$,
- $\wp = \{(n_i,s_i), (n_j,s_j)\}$,
- $\partial = \{(n_i, e_1, n_j), (n_i, e_2, n_j), (n_i, e_3, n_j), (n_i, e_4, n_j)\}$.

Let us construct an $ACG = (V, T)$ from G_{sm} (Fig. 3, b)). We proceed as follows:

- $V = N = \{n_i, n_j\}$,

- $T = \{t | \exists E'.E' = \{e | \partial(n_i,e) = n_j\} \wedge t = (n_i,n_j,sizeof(E')/\wp(n_i)\} =$
  $\{t | \exists E'.E' = \{e_1,e_2,e_3,e_4\} \wedge t = (n_i,n_j,sizeof(E')/\wp(n_i))\} =$
  $\{(n_i,n_j,sizeof(\{e_1,e_2,e_3,e_4\})/\wp(n_i))\} =$
  $\{(n_i,n_j,sizeof(\{e_1,e_2,e_3,e_4\})/s_i)\}$. □

Please notice that the input signals of the block n_i (i.e., in-ports) and the output signals of the block n_j (i.e., out-ports) do not participate in the construction of ACG from G_{sm}. This is because these signals do not influence the application internal structure.

4.2. Task Pattern

Each task $v \in V$ of ACG executes a function and is mapped to a separate PU in the platform. However, independently of the functionality of different tasks, each task operates according to the pattern shown in Fig. 4. A task runs the loop for receiving, processing and sending (RPS) data, where the behaviour of every task is specified as follows:

- a task starts processing data as soon as it has at least one token (i.e., one piece of data) in every input buffer,
- when a task runs, it consumes one token from every input buffer and produces one token for every output buffer, i.e., the task processes the received data according to the function derived from the model and sends processed data further according to the edge of ACG,
- a task without inputs fires every s sampling time.

To preserve behavioural semantics when mapping G_{sm} to ACG, we assume the ideal case where the computation in tasks and communication take no time as does in the Simulink blocks and links.

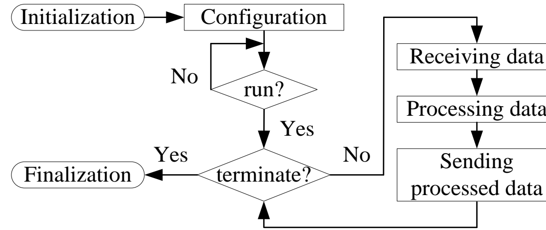


Figure 4: Task pattern

Claim 1. The Simulink model and the ACG are functionally equivalent.

Proof. Consider the mapping between Simulink and ACG. The in-ports and out-ports of the model are respectively inputs and outputs of the ACG graph. The number of tasks in ACG corresponds to the number of atomic sub-systems in the Simulink model. Given the same input data, the tasks and the corresponding sub-systems produce the same result, since each task executes a function of the corresponding sub-system. Due to the assumption on behavioral semantics and the fact that the model is single-rate, the tasks produce data on every output edge every s time units when all input data are available. Hence, an ACG and a Simulink model are equivalent. □

5. Resilience of the Platform and Applications

5.1. Fault Model

We consider the fault model that captures *physical failures* of processing units of the platform. A failure can be caused by transient, intermittent or permanent faults due to high temperature [8], radiation [10], for example. We assume that only one failure of PU can occur at a time independently of the number of faults causing it. In other words, a sufficient amount of time must elapse between two consecutive failures.

For the sake of simplicity, we assume that PUs are fail silent that either produce the correct result or no result at all [19][29]. Fail-silence assumption however can be softened if erroneous results are detected and isolated by using various mechanisms such as model-based diagnosis [30], runtime verification [31], by integrating CRC-like sums into packets and their checks into tasks [29] and others [32][33].

After a task is reallocated from a failed tile, the task starts over from the initialization phase (see Fig. 4); hence, all local variables receive initial values. However, the packets are stored in the buffers of RT which is a separate unit of a tile (see, e.g., [3]) or in the main memory. Therefore, these data remain intact and can also be reallocated along with the task.

We can assume that reading from and writing to a FIFO buffer (queue) are atomic operations, i.e., either the buffer is read or updated, respectively, or not. However, if a task has several input and/or several output buffers, the reading and sending proceed in a buffer-by-buffer manner. In addition, we distinguish between source and regular tasks. The source tasks receive input data from the environment. The regular tasks consume data produced by other tasks and send processed data further or provide an output to the environment. Independently of whether a task is source or regular, it can be stateless (without memory) or stateful (with memory). Consequently, we have 4 cases in total: *stateless regular tasks*, *stateless source tasks*, *stateful regular tasks* and *stateful source tasks*.

5.2. Fault scenarios

According to the pattern shown in Fig. 4 and the described fault model, there are several possible *scenarios* of a fault occurrence within the RPS loop:

- (FS1) *a fault occurs before a task reads any packet from the input buffers*. In this case, the task can still read the input data after reallocation as the data remain in the FIFO buffer that does not belong to the PU.
- (FS2) *a fault occurs while a task reads input data*. That is, a task reads packets from some first queues, but fails to read the necessary data from other queues. Hence, some pieces of data may be lost.
- (FS3) *a fault occurs before the task sends the processed data*. The task has read all the required input packets, but has not finished processing them or has not been able to send the processed data. In this case, the task loses data of one iteration. This can also lead to desynchronized reception of data by the successor tasks.
- (FS4) *a fault occurs while a task sends data*. In this case, some successor tasks may receive packets with new data while others may not.

5.3. Packet Sending and Handling

To prevent tasks from losing data when addressing physical failures of PUs according to the described scenarios, we propose the following mechanism. Firstly, the packets used for communication between tasks incorporate a sequence number (packet id). The source tasks provide initial values for this number starting from zero and increase it every time when a new input is read (Fig. 5, a)). The regular tasks do not change this number, which allows tasks to synchronize packets received from different queues as explained later in this section.

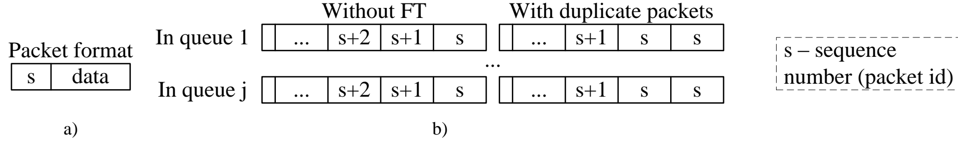


Figure 5: Packets: a) packet format, b) state of buffers with and without FT

Secondly, instead of sending one packet, each task, except for the tasks producing output to the environment, sends the same packet a number of times: the main packet and its duplicates. The number of duplicates (`PD`) depends on the number of faults occurring in a row (`FR`) in a linear manner: `PD = FR + 1`. That is, the number of duplicates depends on the number of reallocations required to tolerate failures of PUs. Here, we assume that a fault cannot occur immediately after one reallocation of a particular task, since we utilize spare cores as in [12]. In this case the sufficient number of duplicates equals to two (Fig. 5, b)): `PD = 1 + 1 = 2`.

Consequently, the tasks in ACG now receive and send two packets with the same data. The packets incorporate a sequence number, where the main packet and its duplicate have the same sequence number. However, the tasks need an intelligent reading procedure that uses duplicates when needed and filters them when there is no failure, i.e., tackles (FS2)-(FS4).

Stateless regular tasks proceed according to the algorithm presented in Fig. 6. When a regular task without a memory block receives a packet from an input buffer, the task compares the id of this packet with the local id. Since initially the local id equals -1 and the id of packets start form 0, the task will use the packets received at the first iteration. After all packets from all input buffers have been read, the task updates the local id with the id of the packets just read. At the following iterations, the task checks whether it has read a duplicate by repeatedly comparing the id of the packets read with the local id. The comparison allows the task to detect duplicates since the main packet and its duplicates have the same sequence number. If no fault has occurred, the value of the local copy of the sequence number is less (previous packet) or equal (duplicate) to the sequence number of the packets read. Hence, the task will simply discard duplicates and reread the buffer for packets with a greater sequence number (condition $pkt_q_j.id <= lsn$ in Fig. 6).

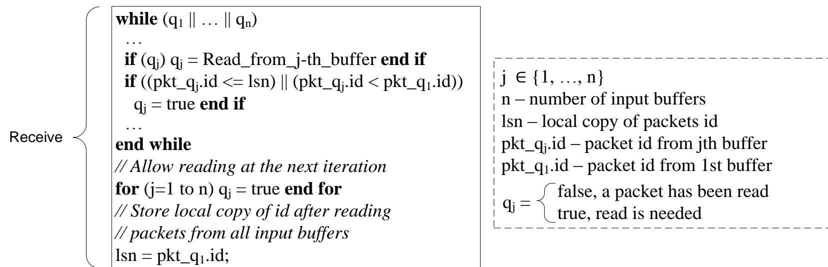


Figure 6: Intelligent reading in regular tasks without memory: the algorithm

In case a fault occurs, the local copy of the sequence number is initialized with -1. Depending on the FS, there are several possible outcomes. In (FS1), the task proceeds normally after reallocation as the main packets remain intact in the input buffers (see Fig. 5, b)). The effect of the other FSs is shown in Fig. 7, which captures states of the input buffers of task n_i considering (FS2)-(FS4).

If (FS2) takes place, there are two possible cases. In the first one, a fault occurs while the task reads main packets from buffers (Fig. 7, FS2, Case 1). In this case, the task can proceed normally after reallocation since there are duplicates in the buffers. In the second case, a fault occurs when the task has read duplicate packets from some queues but failed to read duplicates from other queues (Fig. 7, FS2, Case 2). This may lead to desynchronized packet receiving as the task reads data in a buffer-by-buffer manner. To avoid this, the task compares packet id received from the first queue with ids of the packets read from other queues. If the id of a packet from another queue is less than the id of a packet from the first queue, the task needs to reread this queue (Fig. 6, condition $pkt_q_j.id < pkt_q_1.id$). This enables synchronization of packets read from different queues as only source tasks provide sequence numbers for packets and regular tasks do not modify them.

In (FS3), where a fault occurs before the task starts sending the processed data, the task will use duplicates residing in the buffers after reallocation (Fig. 7, FS3).

Finally, the algorithm also covers (FS4) if, e.g., task n_l is reallocated due to a failure of PU (Fig. 7, FS4), as at least one copy of a packet always resides in the buffers. Please notice that a task can send more than two duplicates in case of (FS4). However, they will be filtered by the proposed algorithm.

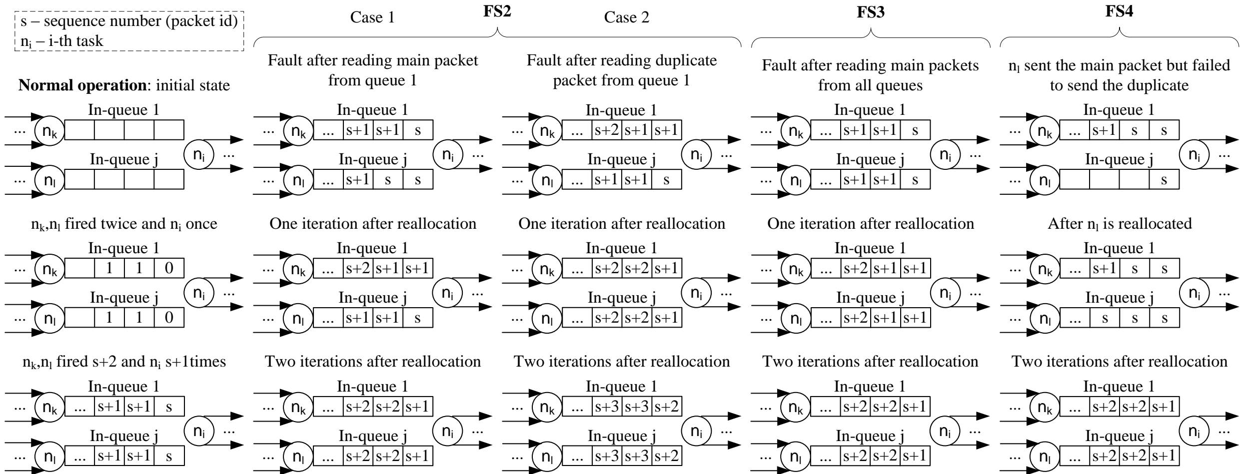


Figure 7: Intelligent reading in regular tasks without memory: buffer states

Stateless source tasks proceed according to the algorithm shown in Fig. 8. Since the source tasks provide sequence numbers for packets, they need to synchronize the ids in case of a failure. To achieve this, we provide such a task with a self-buffer, i.e., the buffer that is read and updated by the task itself. At each iteration step, the source task without memory reads one service packet from the service buffer. If the sequence number has already been used (condition $pkt_q_{srv}.id <= lsn$ in Fig. 8), the task rereads the buffer. Then, it reads the inputs and processes them. After that, the source task sends the processed data to other tasks and, then, sends the service packets with updated sequence number to the self-buffer.

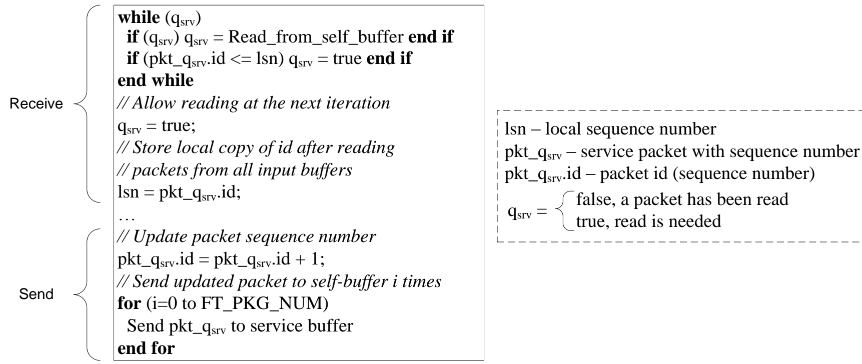


Figure 8: Synchronization of sequence number in source tasks: algorithm

If a failure occurs after the task has read one service packet, but before it has updated the buffer (FS3), the task will synchronize the sequence number with the current value as at least one packet resides in the buffer (Fig. 9, FS3). Similarly, the task synchronizes the sequence number if a failure occurs while service buffer update as the task needs to send duplicate packets (Fig. 9, FS4). Please notice that (FS2) cannot occur because the task either reads the service buffer or not.

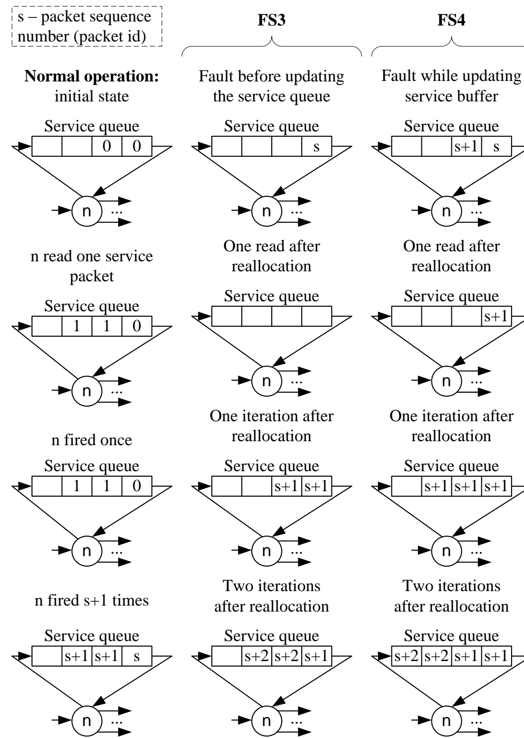


Figure 9: Synchronization of sequence number in source tasks: buffer states

Stateful regular tasks with memory blocks proceed according to the algorithm presented in Fig. 10 similarly to the previously described type of tasks. A stateful regular task also needs a dedicated self-buffer to store a memory value and retrieve it when a failure occurs. The task reads the input packets according to the algorithm presented in Fig. 6. Consequently, (FS2) and (FS3) are already covered. Here, we only show the mechanism for handling a memory value and a corresponding self-buffer.

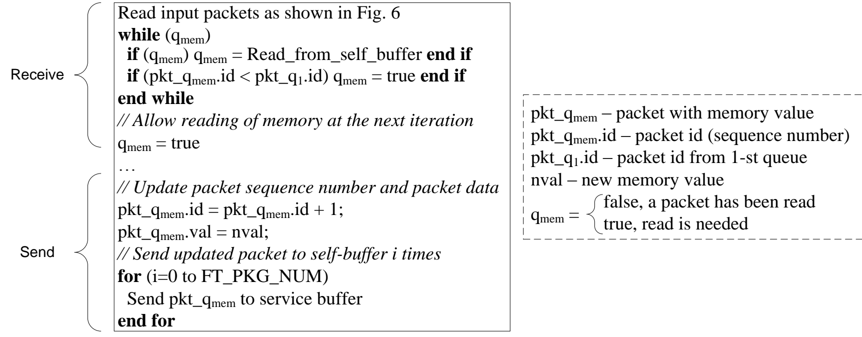


Figure 10: Regular tasks with memory: algorithm

The dedicated self-buffers used for storing a memory value are initialized according to the model when an application is mapped to the platform. After reading all the input packets as shown in Fig. 6, a stateful regular task reads a memory value from the self-buffer. Then, it compares the id of the packet just read with the id of the packet read from the first input buffer. If the id of the memory packet is less than the id of the packet read from the first queue (condition $pkt_q_{mem}.id < pkt_q_1.id$ in Fig. 10), the task rereads the memory buffer. After processing the data read, the task sends processed data to other tasks according to ACG. It also updates the dedicated self-buffer with a new memory value and a new id whose value equals to $pkt_q_1.id + 1$. This is because new packets in the input buffers will have an id whose value equals to $pkt_q_1.id + 1$ as well. Therefore, there is at least one memory value in the buffer (Fig. 11).

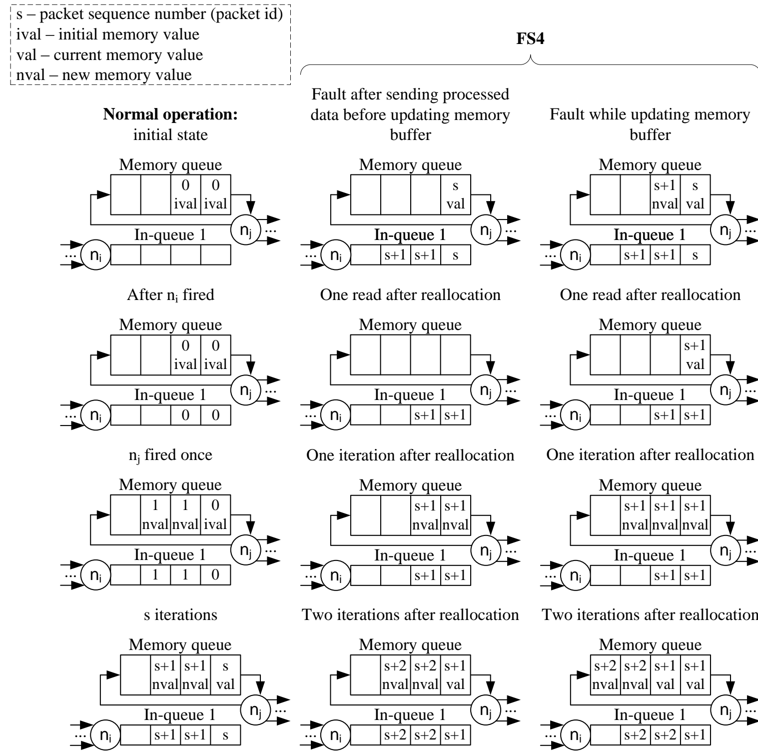


Figure 11: A regular task with a memory block: a) buffer states, b) algorithm

Finally, in case **a source task integrates a memory block**, it combines the algorithms depicted in Fig. 8 and in Fig. 10. The order in which operations take place is the following (Fig. 12). First, the task reads the service queue and stores a local copy of the sequence number. Then, it reads the memory buffer, after which it reads inputs. Then, the task processes the data read and send the processed data further. When the data have been sent to other tasks, the task updates the service

buffer with a new sequence number as well as refreshes the memory buffer with the same new sequence number and a new memory value (Fig. 12).

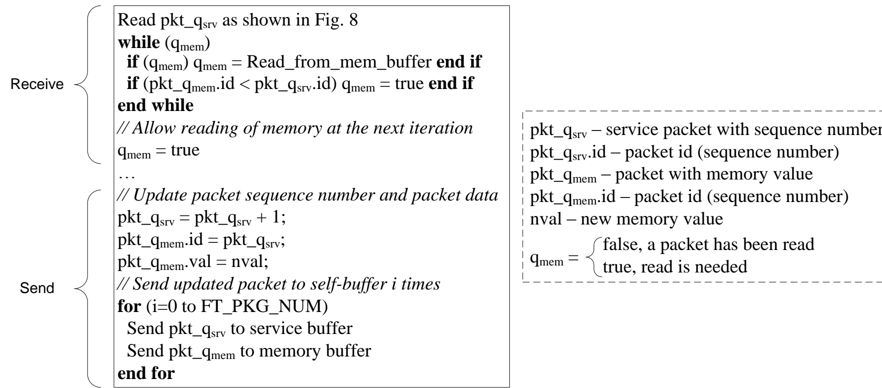


Figure 12: Source task with memory block and synchronization of sequence number: algorithm

Fig. 13 illustrates the buffer states that can occur specifically for this kind of tasks according to the described fault scenarios. The other cases are already handled by the algorithms in Fig. 8 and Fig. 10 and are omitted.

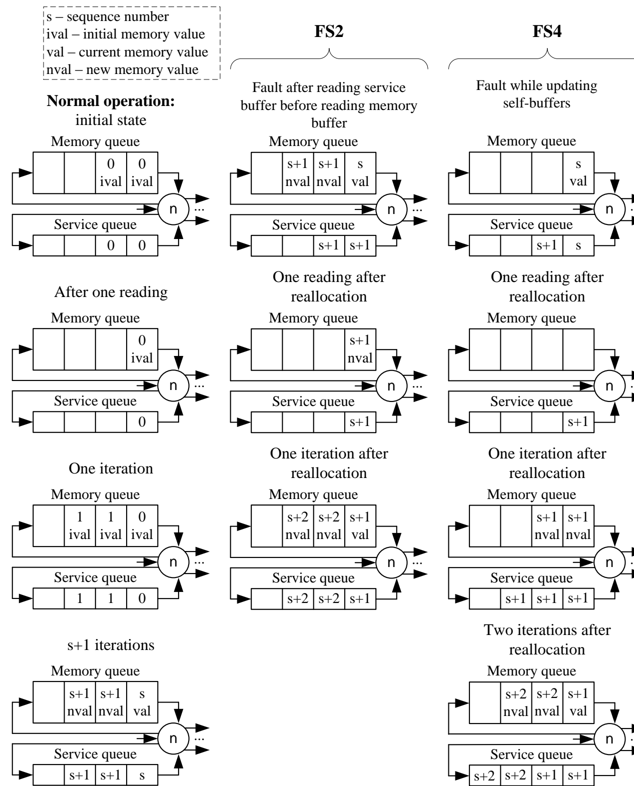


Figure 13: Source task with memory block and synchronization of sequence number: buffer states

Initially, the communication buffers are empty, except for the ones that store sequence numbers and memory values. When the platform agent maps an application to the platform, it provides initial values for these buffers according to the model. The sequence numbers of packets start with zero.

Claim 2. The application tasks operate with the same sequence number within one iteration independently of the number of the source tasks.

Proof. Let us recall that the models we consider in this paper are single-rate, i.e., the tasks fire at regular time intervals. The source tasks start the sequence number from 0 and monotonically increase it when a new input is read. The regular tasks, on the other hand, fire when at least one token is available on every input queue. Furthermore, regular tasks do not change the sequence number, but redistribute it throughout ACG. In addition, the tasks with memory blocks update memory buffer with a new sequence number so that it correlates to the sequence number of input packets. Therefore, every task operates with the same sequence number within one iteration. □

Claim 3. FT packet processing preserves functionality of the application.

Proof. The proof proceeds following the inductive approach on the sequence numbers of packets. Consider the fact that the service and memory buffers are initialized when an application is mapped to the platform. The initial values serve as the base case. The states of the buffers reflect the inductive case. According to Claim 2, the FT tasks operate with the same sequence number as the non-FT tasks within one iteration. Hence, the proposed technique preserves functionality and provides the necessary level of resiliency. □

In summary, the tasks do not lose packets (data) regardless of which fault scenario would happen. Moreover, using fast fault detection, the approach provides low latency as tasks do not need to restart computation from the very beginning, but to reprocess a single piece of data.

6. Case study

This work has been inspired by the controller of digital hydraulics described in details in [34] developed in the IHA laboratory at Tampere University of Technology. The controller represents a real-time periodic system which should react on changes in the environment within specified hard deadlines. The fault occurrence of a processing unit while the controller is running may lead to dramatic effects, e.g., dangerous pressure peaks and wear out of hydraulic components. Therefore, the system has to be efficient and resilient.

Let us now illustrate our approach using case study tasks that represent different cases: a regular task without memory (Fig. 14, task n_3 in Fig. 2), a source task without a memory block (Fig. 15, task n_0 in Fig. 2), a regular task with a memory block (Fig. 16, task n_8 in Fig. 2) and a source task with a memory block (Fig. 17, task n_1 in Fig. 2). For simplicity, the source tasks read input data from the files. The task that produces an output to the environment stores results in a file as well. All tasks instantiate the pattern presented in Fig. 4 and implement a corresponding FT algorithm.

To specify the number of duplicates, we introduce a global constant, namely `FT_PKG_NUM`. If it equals to 1, the tasks send only main packets without duplicates, i.e., the application is non-FT. If it equals to 2, the tasks send one main packet and one duplicate, that is, the tasks can tolerate one failure. Failures are simulated using the statement `if (fault[th_id]) goto finalize;`, where `th_id` is the id of a task. Hence, when the condition holds, the task finalizes its execution.

Regular task without memory. The first example is a sub-system that reflects a regular task without a memory block (Fig. 14). The task reads one input and produces four outputs. The constants `Constant1`, `Constant3` and `Constant5` (Fig. 14, a)) are external to the sub-system, i.e., these are global parameters of the controller. Although this sub-system includes other sub-systems (Fig. 14, b)), we chose it to be a single task as these sub-systems are rather simple and used for the structural purpose only.

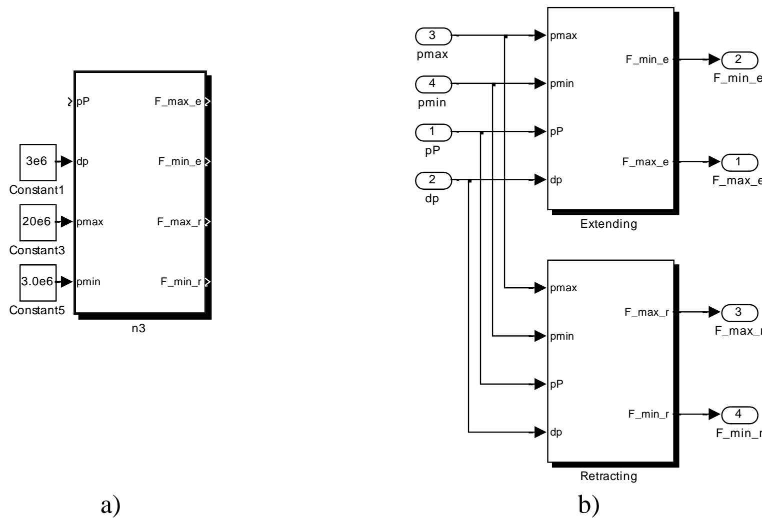


Figure 14: A regular task without memory, n_3: a) sub-system, b) sub-system contents

The excerpt of the derived C code is presented in Appendix A. The code implements the algorithm depicted in Fig. 6. To receive and send data, we use functions Queue_<TYPE>_dequeue and Queue_<TYPE>_enqueue for non-blocking reading and writing, respectively. The former checks whether an input FIFO buffer is empty or not. The latter tests whether a receiver FIFO is full or not. The functions return -1 (true) if the buffer is empty or full. Otherwise, they return 0 (false).

Since the functions for reading from and writing to a buffer are non-blocking, we use the **while** loop to achieve blocking reading and/or writing. For instance, in Appendix A, the task reads a new packet while result of reading stored in the variable tn1q equals to -1, i.e., true. The **if** (tn1q) statement does not allow the task to reread the queue, especially if a task has multiple input queues. The task sends data while the condition qcnt < FT_PKG_NUM holds. That is, when the task sends the main packet to all successor tasks it increments qcnt and proceeds with sending duplicates. Therefore, each successor task first receives the main packet and then FT_PKG_NUM-1 number of duplicates.

Source task without memory. The next example is a source sub-system without a memory block (Fig. 15, a)). We can observe that the functionality of this sub-system is rather simple and the computation is straightforward. First, the sub-system acquires three inputs. Then, two of them are summed up and the result is multiplied by one of the constants. The other input and constant are directly multiplied. Finally, both multiplication results are summed up to compute the output result (Fig. 15, b)). The constants are the local parameters of the sub-system and therefore available to the sub-system only.

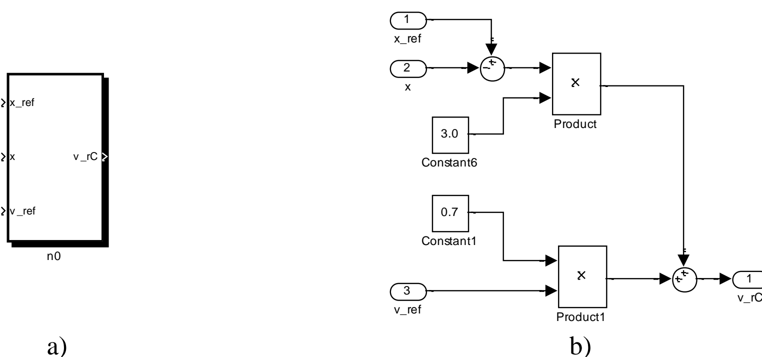


Figure 15: A source task without memory, n_0: a) sub-system, b) sub-system contents

The excerpt of the derived C code is shown in Appendix B. The code implements the algorithm depicted in Fig. 8. Please notice the mechanism for the sequence number synchronization. The task first reads the service packet (the **`while`** `(srvq)` loop). Then, stores the local copy (`lcl_seq_num = itd.pkg.pkg_id`). Finally, after the task has processed data and has sent them to other tasks, it sends the service packets with an updated sequence number to the service buffer (the **`for`** `(`**`short`** `i = 0;...)` loop).

Regular task with memory. The sub-system in Fig. 16, a) illustrates an example of a regular task with a memory block. The content of the sub-system is shown in Fig. 16, b). Although the sub-system includes other sub-systems, we chose it to be a single task. The excerpt of the derived C code is presented in Appendix C. This task receives data from two other tasks as well as reads and updates the memory self-buffer. Hence, it implements the algorithm depicted in Fig. 10. The statement **`if`** `(r_data_tn7.pkg_id <= r_data_tn5_copy || r_data_tn7.pkg_id < r_data_tn5.pkg_id)` in the **`while`** `(tn5q || tn7q)` loop makes the task to synchronize the reception of packets from different queues. Since the memory stores a vector of values, we use the function **`memcpy`** to copy vectors.

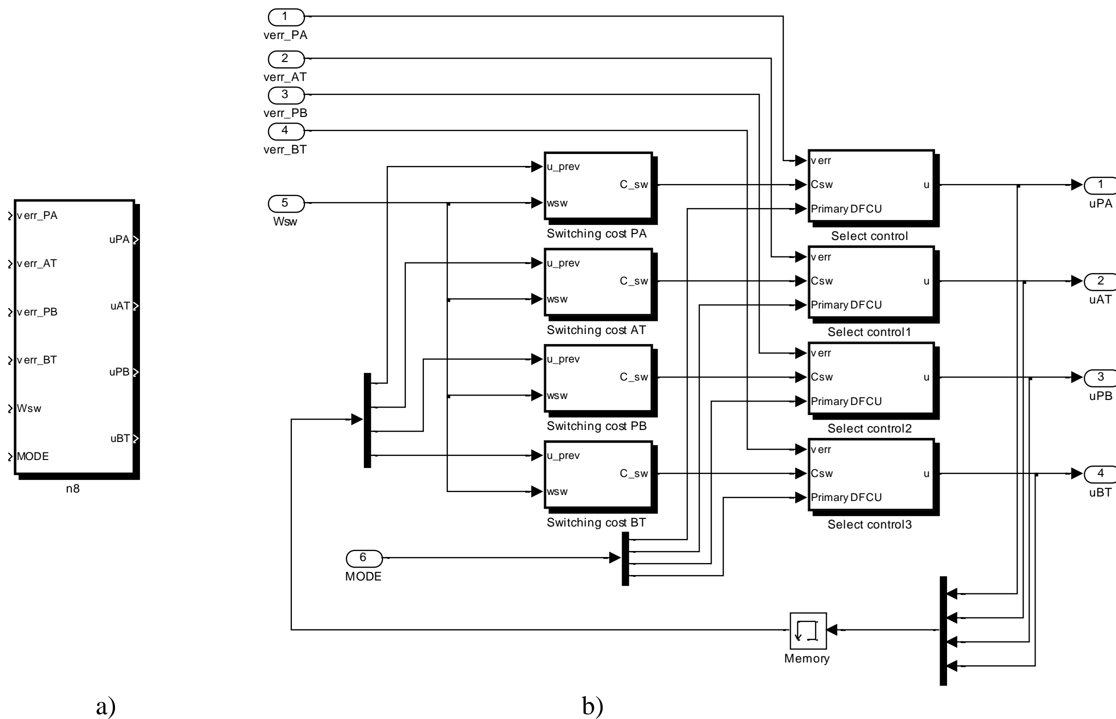
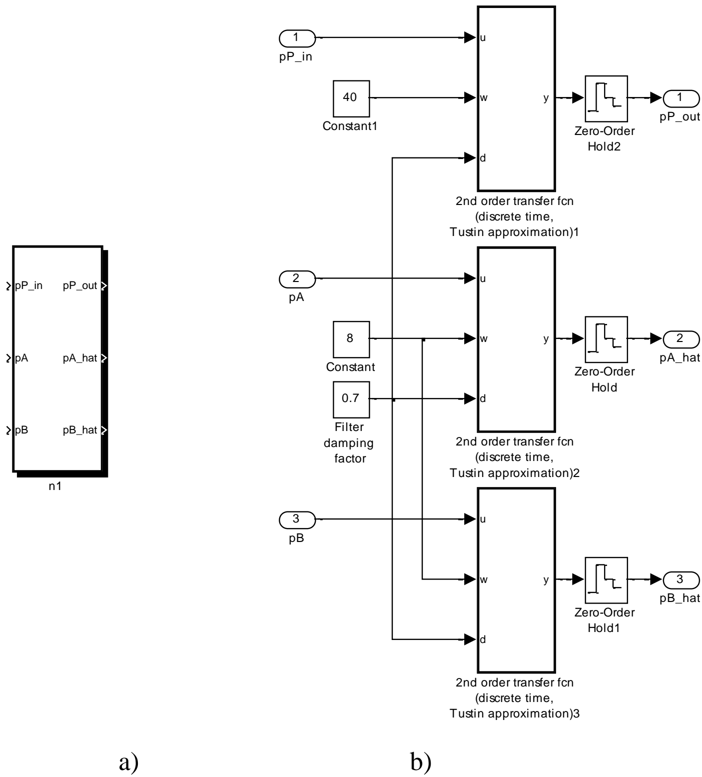


Figure 16: Regular task with memory, n_8: a) sub-system, b) sub-system contents

Source task with memory. Finally, the sub-system in Fig. 17, a) represents a source task with memory. Although this sub-system also contains sub-systems that integrate memory (Fig. 17, b)), we chose it to be a single task. In comparison with other tasks, this task runs six times faster and reads six times more input data. However, it sends packets with the same rate as the other source task. This is achieved by using zero-order hold blocks whose sampling time is the same as the sampling time of the other tasks. Hence, it preserves the value of the sequence number within one iteration, but requires two counters: one for local iteration and the other one for sequence numbers of the packets. Despite this, the task operates according to the algorithm shown in Fig. 12.

The derived C code is shown in Appendix D. Please notice the order in which the instructions take place. First, the task reads the service queue (the **`while`** `(srvq)` loop). Then, the task reads memory buffer (the **`while`** `(memq)` loop). After that, it reads the input data from a file and processes them. At every iteration, the task updates the service and memory queues with new values. If the

condition `cnt % RATIO == 0`, where `RATIO` equals to six, holds, the task sends the computed data to other tasks.



a) b)

Figure 17: Source task with memory, n_8: a) sub-system, b) sub-system contents

Cluster agent task. To enable dynamic reconfiguration of the platform (i.e., tasks reallocation), we adopt the approach proposed in [12] and implement the algorithm of the cluster agent in the form a specific task. This task manages the application and reallocates the case study tasks when simulating failures. The reader is referred to [12] for details on the reallocation algorithm and agent-based management.

Main task. The main task initializes the environment and creates threads out of the ACG tasks. Additionally, it creates the agent task and maps all the tasks to the cores. The derived C code is presented in Appendix E. Please notice that when the main task initializes environment, it also provides initial values for the self-buffers, i.e., for the service and memory buffers.

7. Evaluation results

The proposed approach has been evaluated on a case study described in the previous section and implemented on the TilePro platform [3] without running other applications than OS (Linux Santiago 6.0, Kernel 2.6.36-4). The platform integrates 64 tiles forming an 8x8 square mesh with a network-based communication between the tiles. The network connections are 32-bit full-duplex, there is single cycle latency between adjacent tiles and packet length is up to 128 32-bit words. Bisection bandwidth equals 2660 Gbps. Due to the platform architecture, the size of FIFO buffers is limited to the power of 2. To tolerate faults, the proposed approach requires buffers of size 3. Hence, we provide communication buffers of size 4 for storing 3 packets in total: one current duplicate packet, one new main packet and one new duplicate. The platform runs at the frequency of 862.5 MHz so that one execution cycle approximately takes 1.1594 ns. The platform employs

deterministic XY routing [26] with the dead-lock and live-lock free algorithm suitable for real-time systems [27][28].

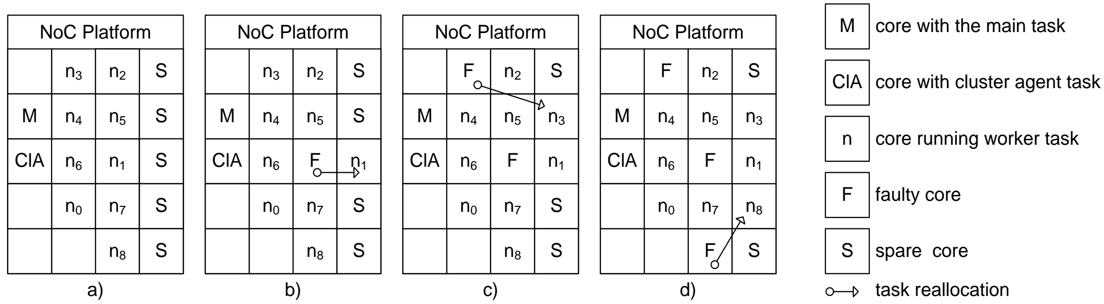


Figure 18: Parallel implementation of safe controller:
a) initial mapping, b) reallocation of task id 1, c) reallocation of task id 3, d) reallocation of task id 8

The initial mapping of the case study is shown in Fig. 18, a). Then, multiple tasks reallocation take place as if faults occurred. Particularly, the tasks with ids 1 (Fig. 18, b)), 3 (Fig. 18, c)) and 8 (Fig. 18, d)) are reallocated. The locations of these tasks reflect middle, topmost and bottommost tiles of the application region, respectively. Additionally, task 1 is a source task with a memory block, task 3 is a regular task without a memory block while task 8 is a regular task with a memory block, which provides output to the environment.

We have first evaluated performance of the sequential and parallel C programs derived from the case study Simulink model without tasks reallocation. The test case consists of 2085 pieces of input data. Tab. 1 summarizes the evaluation results for the whole set of input data, where each number is of the order 1E+09.

Table 1: Performance evaluation of sequential and parallel programs

	Sequential, cycles	Parallel, cycles	Parallel FT, cycles	Ratio P/S, %	Ratio FT P/S, %
Min	2.3359	1.7104	1.7163	73.22	73.58
Max	2.3418	1.7316	1.7526	73.94	74.84
Avg	2.3388	1.7210	1.7345	73.58	74.16

From Tab. 1, we can observe that the parallel program without fault-tolerance technique performs more than 26% more efficiently than its sequential counterpart. The use of the proposed fault-tolerance technique reduces performance by about 1% while allowing the application to produce the expected result without interruption and loss of data when invoking dynamic tasks reallocation. Although we observe 1% reduction in performance, the fault-tolerant parallel implementation is still more efficient (more than 25%) than the sequential implementation without fault-tolerance.

Consequently, we can see two main advantages of generating a parallel program. On the one hand, the reduction of execution time enables the application to fulfil real-time requirements. On the other hand, the saved time can be utilized to achieve system resilience by using dynamic reconfiguration while maintaining performance efficiency.

Furthermore, we have evaluated the effect of the tasks reallocation on the tasks performance. Tab. 2 summarizes the evaluation results, where all numbers are of the order 1E+09. The term *original* stands for the core where a task has been mapped initially, the term *reallocated* refers to a spare core allocated in the rightmost column of the application region according the reallocation algorithm in [12]. The task ids corresponds to the tasks that have initially been allocated to the top (task id 3), middle (task id 1) and bottom (task id 8) cores of the region (see Fig. 18, c), b), d), respectively).

Table 2: Tasks performance

Performance		Task id		
		1	3	8
Min	*original, cycles*	1.6981	1.7064	1.7253
	reallocated, cycles	1.7079	1.7149	1.7343
Max	*original, cycles*	1.7287	1.7406	1.7618
	reallocated, cycles	1.7242	1.7403	1.7514
Average	*original, cycles*	1.7186	1.7300	1.7507
	reallocated, cycles	1.7183	1.7283	1.7452
Max	*r/o ratio, %*	99.74	99.98	99.41

The table shows that the task performance reallocated to a spare core deviates from the task performance allocated initially by at most 0.6%, which is a marginal overhead. In some cases, performance of the reallocated task is better than the performance of the same task at the original location (e.g., task with id 8, minimum). This can be explained by the fact that there is lighter traffic to spare cores when routing packets.

Table 3: Algorithm performance

Performance, *cycles*	Task id		
	1	3	8
Min	1.3094	1.0515	1.0711
Max	1.3886	1.3855	1.1035
Average	1.3444	1.2178	1.0865

Additionally, we have evaluated the performance of the reallocation algorithm that determines spare cores to be used as substitutions [12]. From Tab. 3, where the numbers are of the order 1E+04, we observe that the algorithm determines an appropriate spare core using at most 14000 cycles, which is approximately 16 μS. Considering the fact that even the parallel implementation of the controller requires about 1 mS to process a single piece of data from the input tasks to the output task, the algorithm latency is negligible.

Table 4: Reallocation performance

Performance, *cycles*	Task id		
	1	3	8
Min	8.3672	8.5377	8.8798
Max	9.9380	10.494	9.7322
Average	9.1065	9.2018	9.3080

We have also evaluated the performance of the reallocation procedure. From Tab. 4, we observe that the worst case is the reallocation of task with id 3, which requires a little more than 100 000 cycles, i.e., approximately 120 μS. Hence, the total time required to reallocate a single task is less than 140 μS.

Therefore, from the above tables, we can conclude that the proposed approach provides a high level of efficiency and enables the applications to produce the expected result. The high level of efficiency is obtained because the network connections are 32-bit full-duplex, there is single cycle latency between adjacent tiles and the packet length is up to 128 32-bit words [11][26]. This provides an adequate bandwidth for duplicating packets.

8. Conclusion and future work

We have shown an approach to deriving parallel programs from arbitrary discrete single-rate Simulink models. Relying on the behaviour of the resulting ACG, we have introduced a scalable fault-tolerance (FT) mechanism that prevents data loss when application tasks are relocated due to failures of PUs. We have evaluated performance of the derived programs as well as of the proposed FT mechanism. The results show only about 1% performance decrease when comparing non-FT and FT versions. Therefore, the proposed approach maintains efficiency and provides resilience to faults allowing applications to produce the expected result. The proposed FT can also be used separately from Simulink but requires the aforementioned assumptions. Furthermore, it is not restricted to data parallel applications and can be applied to functionally parallel ones. Our approach can be integrated into FT dataflow proposed in [21] or in [35].

Due to the systematic nature of the proposed approach, it can be relatively easily incorporated into the Simulink environment in order to allow designers to generate parallel code in an automated manner. Hence, one direction of our future work is to develop a tool support for the proposed approach.

In the current work, we considered single-rate Simulink models where tasks fire at regular time intervals. However, one can specify a multi-rate model where the rate of a sub-system is computed as a greatest common divisor of the rates of its sub-systems. Hence, another future direction of our work is to extend the presented approach to multi-rate Simulink models.

In addition, when mapping Simulink models to ACG, we assumed that the computation in tasks and communication in ACG took no time since Simulink blocks and links take no time either. However, in the real world environment, the tasks take time to compute functions. Hence, another direction of our future work is to explore approaches to semantic equality between G_{sm} and ACG in terms of the described issue. Moreover, the proposed task pattern can be interpreted as a timed automaton. A system is then a collection of communicating timed automata. Therefore, performance analysis using timed automata and their tool support Uppaal [36] is also of particular interest and will be explored in the future work.

Acknowledgment

The authors would like to thank Adjunct Professor Juha Plosila for fruitful discussions. The authors would also like to thank the IHA laboratory at Tampere University of Technology for providing the case study.

References

[1] Simulink, Simulation and Model-Based Design, 2013. Available: http://www.mathworks.se/products/simulink/

[2] L. Benini, G. De Micheli, Networks on chips: a new SoC paradigm, Computer: IEEE, Vol. 35, Issue 1, pp. 70 – 78, 2002.

[3] Tilera, TilePro processor family, 2013. Available: http://www.tilera.com/products/processors/TILEPro_Family

[4] T. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl and S. Dighe, “The 48-core SCC Processor: the Programmer’s view”, in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1-11.

[5] C.-L. Chou, R. Marculescu, User-Aware Dynamic Task Allocation in Networks-on-Chip, Design, Automation and Test in Europe DATE, Munich: IEEE, pp. 1232-1237, 2008.

[6] B. Yang, T. Xu, T. Säntti, J. Plosila, Tree-Model Based Mapping for Energy-Efficient and Low-Latency Network-on-Chip, Design and Diagnostics of Electronic Circuits and Systems (DDECS), Vienna: IEEE, pp. 189-192, 2010.

[7] F. Khalili, H. R. Zarandi, A Fault-Tolerant Low-Energy Multi-Application Mapping onto NoC-based Multiprocessors, International Conference on Computational Science and Engineering, Nicosia: IEEE, pp. 421-428, 2012.

[8] G. Link, N. Vijaykrishnan, Hotspot Prevention Through Runtime Reconfiguration in Networks-on-Chip, Design, Automation and Test in Europe DATE, IEEE, pp. 648-649, 2005.

[9] C. Addo-Quaye, Thermal-aware Mapping and Placement for 3-D NoC Designs, SOC Conference, IEEE, pp. 25-28, 2005.

[10] C. Bolchini, M. Carminati, A. Miele, Self-Adaptive Fault-Tolerance in Multi-/Many-Core Systems, Journal of Electronic Testing: Theory and Applications, Vol. 29, Issue 2, Springer US, pp. 159-175, 2013.

[11] C.-L. Chou, R. Marculescu, FARM: Fault-Aware Resource Management in NoC-based Multiprocessor Platforms, Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble: IEEE, pp. 1-6, 2011.

[12] S. Ostroumov, L. Tsiopoulos, J. Plosila, K. Sere, Formal Approach to Agent-Based Dynamic Reconfiguration in Networks-On-Chip, Journal of Systems Architecture, 59(9), Elsevier, pp. 709-728, 2013.

[13] L. Guang, J. Plosila, J. Isoaho, H. Tenhunen, Hierarchical Agent Monitored Parallel On-Chip System: A Novel Design Paradigm and its Formal Specification, International Journal of Embedded and Real-Time Communication Systems (IJERTCS), Vol. 1, Issue 2, IGI, pp. 86-105, 2010.

[14] P. Rantala, J. Isoaho, H. Tenhunen, Novel Agent-Based Management for Fault-Tolerance in Network-on-Chip, Euromicro Conference on Digital System Design Architectures, Methods and Tools, Lubeck: IEEE pp. 551-555, 2007.

[15] MathWorks, Simulink coder, 2013.
Available: http://www.mathworks.se/products/simulink-coder/

[16] M. Di Natale, Optimizing the multitask implementation of multirate Simulink models, Real-time and Embedded Technology and Applications Symposium (RTAS), IEEE, pp. 335-346, 2006.

[17] N. Vranic, V. Marinkovic, M. Djukic, M. Popovic, An approach to parallelization of sequential C code, Eastern European Regional Conference on the Engineering of Computer Based Systems, IEEE, pp. 143-146, 2011.

[18] M. Noraziz Sham Mohd Sayuti, L. Soares Indrusiak, Real-Time Low-Power Task Mapping in Networks-on-Chip, Computer Society Annual Symposium on VLSI, IEEE, pp. 14-19, 2013.

[19] C. Pinello, L. Carloni, A. Sangiovanni-Vincentelli, Fault-Tolerant Deployment of Embedded Software for Cost-Sensitive Real-Time Feedback-Control Applications, International Conference on Design Automation and Test in Europe, IEEE, pp. 1164-1169, 2004.

[20] C. Dima, A. Girault, C. Lavarenne, Y. Sorel, Off-line real-time fault-tolerant scheduling, Euromicro, IEEE, pp. 410-417, 2001.

[21] J. Ceponis, E. Kazanavicius, A. Mikuckas, Fault Tolerant Process Networks, Information Technology and Control, Vol. 35, No. 2, pp. 124-130, 2006.

[22] MathWorks, Modeling Dynamic Systems, 2014.
Available: http://www.mathworks.se/help/simulink/ug/modeling-dynamic-systems.html

[23] P. Boström, Contract-based verification of Simulink models, International Conference on Formal Engineering Methods (ICFEM), Durham, Springer-Verlag Berlin Heidelberg, pp. 291-306, 2011.

[24] MathWorks, Simulating Dynamic Systems, 2014.
Available: http://www.mathworks.se/help/simulink/ug/simulating-dynamic-systems.html

[25] M. Ebrahimi, D. Masoud, P. Liljeberg, J. Plosila, H. Tenhunen, Efficient Congestion-Aware Selection Method for On-Chip Networks, International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), IEEE, pp. 1-4, 2011.

[26] Tile Processor User Architecture Manual, Tilera, 2011.
Available: http://www.tilera.com/scm/docs/UG101-User-Architecture-Reference.pdf

[27] M. Dehyadgari, M. Nickray, A. Afzali-kusha, Z. Navabi, Evaluation of Pseudo Adaptive XY Routing Using an Object Oriented Model for NOC, International Conference on Microelectronics, IEEE, pp. 204-208, 2005.

[28] V. Rantala, T. Lehtonen, J. Plosila, Network on Chip Routing Algorithms, TUCS Technical Report No 779, TUCS – Turku Centre for Computer Science, pp. 10-16, 2006.

[29] F. Brasileiro, P. Ezhilchelvan, S. Shrivastava, N. Speirs and S. Tao, Implementing fail-silent nodes for distributed systems, IEEE Transactions on Computers, Vol. 45(11), pp. 1226–1238, 1996.

[30] R. Isermann, Model-based fault-detection and diagnosis – status and applications, Annual Reviews in Control 29(1), Elsevier, Vol. 29, Issue 1, pp. 71-85, 2005.

[31] L. Pike, S. Niller, N. Wegmann, Runtime Verification for Ultra-Critical Systems, In Proceedings of International Conference on Runtime Verification, Springer, pp. 310-324, 2012.

[32] C. Villalpando, D. Rennels, R. Some, M. Cabanas-Holmen, Reliable Multicore Processors for NASA Space Missions, Aerospace conference, IEEE, pp. 1-12, 2011.

[33] R. Hamming, Error Detecting and Error Correcting Codes, The Bell System Technical Journal, Vol. XXIX, No. 2, pp. 147-160, 1950.

[34] M. Huova, M. Ketonen, P. Alexeev, P. Boström, M. Linjama, M. Waldén and K. Sere, Simulations with fault-tolerant controller software of a digital valve, Workshop on Digital Fluid Power, Tampere, Finland, pp. 223 – 242, 2012.

[35] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, S. Pezzini: Fault-tolerant platforms for automotive safety-critical applications, International Conference

on Compilers, Architectures and Synthesis of Embedded Syststems, ACM, pp. 170-177, 2003.

[36] G. Behrmann, A. David and K. G. Larsen, A Tutorial on UPPAAL, In Formal Methods for the Design of Real-Time Systems, volume 3185/ 2004, Springer Berlin / Heidelberg, pp. 200–236, 2004.

Appendix A

A regular task without memory (n_3)

	`// A regular task (id = 3) without memory`
	`// tnN is shorter version of task_nN, where N is the task number`
	`void *Task_n3(void *threadargs) {`

Configure	`// Configuration. Obtaining task pid used for (re)allocation`
	`int th_id = (int) threadargs;`
	`tpid[th_id] = tmc_task_gettid();`
	
	`// Declaring and initializing sub-system parameters (constants)`
	`rtP_tn3 rtp_afr;`
	`Task_n3_Init_Structure(&rtp_afr);`
	
	`// Declaring necessary variables`
	`// for receiving packets from task n1`
	`tn1_data r_data_tn1;`
	
	`// for storing local copy of the sequence number`
	`short r_data_tn1_copy = -1;`
	
	`// for storing computed result`
	`tn3_data result;`
	
	`// Declaring variables to process FIFO buffers`
	`// for Receiving`
	`short tn1q = -1;`
	`// flags that packets have been sent. -1 stands for true`
	`short tn4q[FT_PKG_NUM];`
	`for (short i = 0;i < FT_PKG_NUM; i++) tn4q[i] = -1;`
	`// counter for the number of packets sent`
	`short qcnt = 0;`
	
	`// Declaring other variables, if needed`
	`…`

Run?	`// Wait until everything is ready` `while (!run[th_id]) {}`
RPS loop	`// Loop over receiving, processing and sending` `while (!terminate[th_id])   {`

Receive

```
    // Receiving data to process according to ACG in Fig. 2
    while (tn1q) {
      // If no packet has been read
      if (tn1q) {
        // Read a new packet and store the result of operation in the
        // tn1q variable. If reading is successful, tn1q = 0 (false).
        // Otherwise, it is -1 (true)
        tn1q = Queue_tn1_dequeue(tn1_queues[1], &r_data_task_n1);
        // If reading is successful and
        // the task has read a duplicate, reread
        if ((!fopq) && (r_data_tn1.pkg_id <= r_data_tn1_copy))
          tn1q = -1;
      } // end if (tn1q)
    } // end while (tn1q)

    // Store local copy of the sequence number and
    // reset the reading flag
    r_data_tn1_copy = r_data_tn1.pkg_id;
    tn1q = -1;
```

Compute

```
    // Computing a function according to the model
    Task_n3_Func(P_Constant5_Value, P_Constant3_Value,
      P_Constant1_Value, r_data_tn1.pP,
      &rtp_afr, &result);
```

Send

```
    // Providing packet id
    result.pkg_id = r_data_tn1.pkg_id;

    // Sending computed data according to ACG in Fig. 2
    while (qcnt < FT_PKG_NUM) {

      // Send a packet to task with id 4 and store the result of
      // sending in the tn4q variable. If sending is successful,
      // tn4q = 0 (false). Otherwise, -1 (true).
      if (tn4q[qcnt])
        tn4q[qcnt] = Queue_tn3_enqueue(tn3_queue, result);
      // Increase packet counter in order to send duplicates
      if (!(tn4q[qcnt])) qcnt++;
    } // end while (qcnt < FT_PKG_NUM)

    // Reset all the sending flags
    for (short i = 0;i < FT_PKG_NUM; i++) tn4q[i] = -1;
    qcnt = 0;
  } // end while (!terminate[th_id])
```

Finalize

```
  // Finalizing the thread
  finalize:
  …
  return NULL;
} // end void *Task_n3(void *threadargs)
```

Appendix B

A source task without memory and with synchronization of the sequence number (n_0)

```
// A source task (id = 0) without memory
// tnN is a shorter form of task_nN, where N is the task number
void *Task_n0(void *threadargs) {
```

Configure

```
  // Configuration. Obtaining task pid used for (re)allocation
  int th_id = (int) threadargs;
  tpid[th_id] = tmc_task_gettid();

  // Initializing sub-system parameters (constants)
  rtP_tn0 rtp_mc;
  Task_n0_Init_Structure(&rtp_mc);

  // Local variables for inputs,
  real_T rtu_x_ref;
  real_T rtu_x;
  real_T rtu_v_ref;
  // result
  tn0_data result;
  // and local copy of the sequence number
  int lcl_seq_num = -1;

  // Local variables for processing FIFO buffers:
  // for processing service buffer (queue)
  ITD_data itd_pkg;
  short srvq = -1;

  // for sending processed data to other tasks
  // flags that packets have been sent
  short tn4q[FT_PKG_NUM], tn7q[FT_PKG_NUM];
  for (short i = 0; i < FT_PKG_NUM; i++) tn4q[i] = tn7q[i] = -1;
  // counter for the number of packets sent
  short qcnt = 0;

  // Declaring other variables, if needed
  …
```

Run?

```
  // Wait until everything is ready
  while (!run[th_id]) {}
```

RPS loop

```
  // Loop over receiving, processing and sending
  while (t0inp_read > 0) {
```

Receive

```
    // Reading the service queue in order to synchronize
    // sequence number. srvq = 0 (false), if reading is successful
    while (srvq) {
      // Reading a packet if nothing has been read yet
      if (srvq) {
        srvq = Queue_ITD_dequeue(ITD_queues[0], &itd_pkg);
        // If a packet is read and its sequence number has
        // already been used, reread the buffer
        if ((!srvq) && (itd_pkg.pkg_id <= lcl_seq_num)) srvq = -1;
      } // end if (srvq)
    } // end while (srvq)

    // Store local copy of the sequence number and
    // reset the reading flag
    lcl_seq_num = itd_pkg.pkg_id;
    srvq = -1;

    // Reading inputs from a file
    …
```

Compute

```
    // Computing a function according to the model
    Task_n0_Func(rtu_x_ref, rtu_x, rtu_v_ref,
      &rtp_mc, &result);
```

Send

```
    // Providing packet id
    result.pkg_id = lcl_seq_num;

    // Sending computed data to other tasks according to ACG, Fig. 2
    while (qcnt < FT_PKG_NUM) {
     // Sending a packet to task with id 4. tn4q = 0, if successful
     if (tn4q[qcnt])
       tn4q[qcnt] = Queue_tn0_enqueue(tn0_queues[0], result);
     // Sending a packet to task with id 7. tn7q = 0, if successful
     if (tn7q[qcnt])
       tn7q[qcnt] = Queue_tn0_enqueue(tn0_queues[1], result);
     // Increase packet counter in order to send duplicates,
     // if the main packets have been sent successfully
     if (!(tn4q[qcnt] || tn7q[qcnt])) qcnt++;
    } // end while (qcnt < FT_PKG_NUM)

    // Reset all the sending flags
    qcnt = 0;
    for (short i = 0; i < FT_PKG_NUM; i++) tn4q[i] = tn7q[i] = -1;

    // Updating sequence number to be sent to the service buffer
    itd_pkg.pkg_id++;

    // Sending service packet with new sequence
    // number FT_PKG_NUM times
    for (short i = 0; i < FT_PKG_NUM; i++) {
      Queue_ITD_enqueue(ITD_queues[0], itd_pkg);
    } // end for
  } // end while (t0inp_read > 0)
```

Finalize

```
  // Finalizing the thread
  finalize:
  …
  return NULL;
} // end void *Task_n0(void *threadargs)
```

Appendix C

A regular task with memory and producing result to the environment (n_8)

```
// A regular task (id = 8) with memory
// tnN is shorter version of task_nN, where N is the task number
void *Task_n8(void *threadargs) {
```

Configure

```
  // Configuration. Obtaining task pid used for (re)allocation
  int th_id = (int) threadargs;
  tpid[th_id] = tmc_task_gettid();

  // Declaring and initializing sub-system parameters (constants)
  rtP_tn8 rtp_sas;
  Task_n8_Init_Structure(&rtp_sas);

  // Local variables for storing memory value
  tn8_memory tn8_memory;
  real_T Memory_PreviousInput[20];

  // Declaring necessary variables
  // for receiving packets
  tn5_data r_data_tn5;
  tn7_data r_data_tn7;

  // for storing local copy of the sequence number
  short r_data_tn5_copy = -1;

  // for storing computed result
  rtB_tn8 result;

  // Declaring variables to process FIFO buffers
  // for receiving only, since the task produces
  // the result to the environment
  short tn5q, tn7q, memq;
  tn5q = tn7q = memq = -1;

  // Declaring other variables, if needed
  …
```

Run?

```
  // Wait until everything is ready
  while (!run) {}
```

RPS loop

```
  // Loop over receiving, processing and sending
  while (!terminate[th_id]) {
```

Receive

```
    // Receiving data to process according to ACG in Fig. 2
    while (tn5q || tn7q) {
      // from task with id 5. tn5q = 0 (false), if successful
      if (tn5q) {
        tn5q = Queue_tn5_dequeue(tn5_queues[1], &r_data_tn5);
        // If reading is successful and the task has
        // read a duplicate, reread
        if ((!tn5q) &&
            (r_data_tn5.pkg_id <= r_data_tn5_copy)) tn5q = -1;
      } // end if (prq)

      // from task with id 7. tn7q = 0 (false), if successful
      if (tn7q) {
        tn7q = Queue_tn7_dequeue(tn7_queue, &r_data_tn7);
        if (!tn7q) {
          // If of the packet just read is a duplicate, reread
          if ((r_data_tn7.pkg_id <= r_data_tn5_copy)||
              (r_data_tn7.pkg_id < r_data_tn5.pkg_id)) tn7q = -1;
        }// end if (!tn7q)
      } // end if (tn7q)
    } // end while (tn5q || tn7q)

    // Store local copy of the sequence number and
    // reset the reading flag
    r_data_tn5_copy = r_data_tn5.pkg_id;
    tn5q = tn7q = -1;

    // Reading the memory value from the memory self-buffer
....// memq = 0 (false), if reading from the buffer is successful
    while (memq) {
      if (memq) {
       memq = Queue_tn8_memory_dequeue(tn8_memory_queue,&tn8_memory);
       if ((!memq) &&
           (tn8_memory.pkg_id < r_data_tn5.pkg_id)) memq = -1;
      } // end if (memq)
    } // end while (memq)

    // Resetting the reading flag and
    // storing local copy of the memory value
    memq = -1;
    memcpy(&Memory_PreviousInput[0],
      &tn8_memory.Memory_PreviousInput[0],20U * sizeof(real_T));
```

Compute

```
    // Computing a function according to the model
    Task_n8_Func(r_data_tn7.verr_PA,
      r_data_tn5.MODE_vector[0], r_data_tn7.verr_AT,
      r_data_tn5.MODE_vector[1], r_data_tn7.verr_PB,
      r_data_tn5.MODE_vector[2], r_data_tn7.verr_BT,
      r_data_tn5.MODE_vector[3], P_Controller_W_sw, &result,
      Memory_PreviousInput, &rtp_sas);

    // Updating local memory variable
    Task_n8_Update(&result, Memory_PreviousInput);
```

Send	<pre> // Updating memory variable to be sent to the buffer
	    tn8_memory.pkg_id = r_data_tn5.pkg_id + 1;
	    memcpy(&tn8_memory.Memory_PreviousInput[0],
	      &Memory_PreviousInput[0], 20U * sizeof(real_T));
	
	    // Sending new memory value to the self-buffer
	    for (short i = 0; i < FT_PKG_NUM; i++) {
	      Queue_tn8_memory_enqueue(tn8_memory_queue, tn8_memory);
	    } // end for (short i = 0; i < FT_PKG_NUM; i++)
	
	    // Storing results in a file
	    …
	  } // end while (!terminate[th_id])</pre>

Finalize	<pre> // Finalizing the thread
	  finalize:
	  …
	  return NULL;
	} // end void *Controller_SwitchAndSelectControl(void *threadargs)</pre>

Appendix D

A source task with memory and with synchronization of the sequence number (n_1)

```
// A source task (id = 1) with memory and sequence number
// synchronization.
// tnN is a short form of task_nN, where N is the task number
void *Task_n1(void *threadargs) {
```

Configure

```
  // Configuration. Obtaining task pid used for (re)allocation
  int th_id = (int) threadargs;
  tpid[th_id] = tmc_task_gettid();

  // Declaring and initializing sub-system parameters (constants)
  rtP_tn1 rtp_fop;
  Task_n1_Init_Structure(&rtp_fop);

  // Declaring the necessary variables
  // Inputs
  real_T pP;
  real_T pA;
  real_T pB;
  // Local
  rtB_tn1 rtB_tn1;
  // Memory
  rtDW_tn1 rtDW_tn1;
  // Result
  tn1_data result;

  // This task reads and processes inputs 6 times faster than
  // other tasks, but sends packets with the same rate as
  // the other source task. Hence, it requires two counters:
  // one for local iteration and the other one for
  // sequence numbers of the packets
  int iter_cnt, lcl_seq_num;
  iter_cnt = lcl_seq_num = -1;

  // Declaring variables to process FIFO buffers
  // for receiving packets from the service self-buffer
  ITD_data itd_pkg;
  // and from the memory self-buffer
  tn1_memory tn1_memory;

  // reading flags
  short srvq, memq;
  memq = srvq = -1;

  // For sending processed data to other tasks
  // flags that the packets have been sent
  short tn2q[FT_PKG_NUM], tn3q[FT_PKG_NUM],
    tn5q[FT_PKG_NUM], tn6q[FT_PKG_NUM];
  for (short i = 0; i < FT_PKG_NUM; i++)
    tn2q[i] = tn3q[i] = tn5q[i] = tn6q[i] = -1;

  // Counter for the number of packets sent
  short qcnt = 0;
```

Run?

```
  // Wait until everything is ready
  while (!run[th_id]) {}
```

RPS loop

```
// Loop over receiving, processing and sending
while (t1inp_read > 0) {
```

Receive

```
  // Reading the service queue in order to synchronize sequence
  // number. If reading is successful, srvq = 0 (false). Otherwise,
  // it equals to -1 (true).
  while (srvq) {
    if (srvq) {
      srvq = Queue_ITD_dequeue(ITD_queues[1], &itd_pkg);
      if ((!srvq) && (itd_pkg.iter_cnt <= iter_cnt)) srvq = -1;
    } // end if (srvq)
  } // end while (srvq)

  // Store local copy of the counters and reset the reading flag
  iter_cnt = itd_pkg.iter_cnt;
  lcl_seq_num = itd_pkg.pkg_id;
  srvq = -1;

  // Reading memory value from the memory buffer. If reading is
  // successful, memq = 0 (false). Otherwise, it equals to -1
  // (true).
  while (memq) {
    if (memq) {
     memq = Queue_tn1_memory_dequeue(tn1_memory_queue, &tn1_memory);
     if ((!memq)&&(tn1_memory.pkg_id < itd_pkg.iter_cnt)) memq = -1;
    } // if (memq)
  } // while (memq)
  // Resetting the reading flag and
  // storing local copy of the memory value
  memq = -1;
  rtDW_tn1 = tn1_memory.rtDW_tn1;

  // Reading inputs from a file
  …
```

Compute

```
  // Computing a function according to the model
  Task_n1_Func(pP, pA, pB, &rtB_tn1, &rtDW_tn1, &rtp_tn1);

  // Updating local memory variable
  Task_n1_Update(pP, pA, pB, &rtB_tn1, &rtDW_tn1);
```

Send

```
    // Each 6th sample is sent for further processing
    if (iter_cnt % RATIO == 0) {

      // Updating data to be send
      result.pP = rtB_tn1.ZeroOrderHold2;
      result.pA = rtB_tn1.ZeroOrderHold;
      result.pB = rtB_tn1.ZeroOrderHold1;
      // Providing packet id
      result.pkg_id = lcl_seq_num;

      // Sending computed data according to ACG in Fig. 2. tnNq equals
      // to 0 (false), if successful and -1 (true), otherwise
      while (qcnt < FT_PKG_NUM) {
        // to task with id = 2
        if (tn2q[qcnt])
          tn2q[qcnt] = Queue_tn0_enqueue(tn0_queues[0], result);
        // to task with id = 3
        if (tn3q[qcnt])
          tn3q[qcnt] = Queue_tn0_enqueue(tn0_queues[1], result);
        // to task with id = 5
        if (tn5q[qcnt])
          tn5q[qcnt] = Queue_tn0_enqueue(tn0_queues[2], result);
        // to task with id = 6
        if (tn6q[qcnt])
          tn6q[qcnt] = Queue_tn0_enqueue(tn0_queues[3], result);
        // Increase packet counter in order to send duplicates,
        // if the main packets have been sent successfully
        if (!(tn2q[qcnt] || tn3q[qcnt] || tn5q[qcnt] || tn6q[qcnt]))
          qcnt++;
      } // end while (qcnt < FT_PKG_NUM)

      // Reset all the sending flags
      qcnt = 0;
      for (short i = 0; i < FT_PKG_NUM; i++)
        tn2q[i] = tn3q[i] = tn5q[i] = tn6q[i] = -1;

      // Updating sequence number to be sent to the service buffer
      itd_pkg.seq_num++;
    } // end if (iter_cnt % RATIO == 0)

    // Updating iteration counter to be sent to the buffer
    itd_pkg.iter_cnt++;
    // Updating memory variable to be sent to the buffer
    tn0_memory.pkg_id = itd_pkg.iter_cnt;
    tn0_memory.rtDW_tn0 = rtDW_tn0;
    // Running loop for updating self-buffers
    for (int i = 0; i < FT_PKG_NUM; i++) {
      // Sending service packet with new sequence number
      Queue_ITD_enqueue(ITD_queues[1], itd_pkg);

      // Sending new memory value to the buffer
      Queue_tn0_memory_enqueue(tn0_memory_queue, tn0_memory);
    } // end for (int i = 0; i < FT_PKG_NUM; i++)
  } // while (t1inp_read > 0)
```

Finalize

```
  // Finalizing the thread
  finalize:
  …
  return NULL;
}
```

Appendix E

The main task that initializes the environment and maps tasks to cores

```
/**
 * The main task that sets up the environment, creates threads and maps them to
 * the platform cores. When the job is complete the main task releases resources
*/

// Defining global variables
// Simulating faults
bool fault[TASKS_NUM];
// Condition to terminate all the tasks
bool terminate[TASKS_NUM+1];
// Condition to run the worker tasks and the agent task
bool run[TASKS_NUM+1];
// Thread ID assigned by OS
pthread_t tid[TASKS_NUM+1];
// Tasks IDs in terms of PID
pid_t tpid[TASKS_NUM+1];
// Addresses of threads
unsigned long th_funcs[TASKS_NUM];

/**
 * Initialize CPUs to be used by the application
 *
 * @param A set of available CPUs
 * @return A set of CPUs to be used by application
 */
cpu_set_t initializeCPUset(cpu_set_t* cpus) {

  // The returned result
  cpu_set_t result = *cpus;

  // Checking if there are CPUs available
  if (tmc_cpus_get_my_affinity(&result) != 0)
    tmc_task_die("THREAD MAIN: 'tmc_cpus_get_my_affinity' has failed.\n");

  // Determining the number of available CPUs
  size_t count = tmc_cpus_count(&result);

  // If the number is not enough, terminate application
  if (count < CPUS_NUM)
    tmc_task_die("THREAD MAIN: Insufficient CPUs available\n");

  // Initializing CPUs to be used by the application
  // Clearing structures
  cpu_set_t u_cpus;
  tmc_cpus_clear(&u_cpus);

  // Filtering CPUs to be used by the application from all available CPUs
  for (unsigned int i = 0; i < CPUS_NUM;i++)    {
    unsigned int cpu = tmc_cpus_find_nth_cpu(&result, i);
    tmc_cpus_add_cpu(&u_cpus, cpu);
  } // end for (unsigned int i = 0; i < CPUS_NUM;i++)

  // Preparing the result
  tmc_cpus_clear(&result);
  tmc_cpus_add_cpus(&result,&u_cpus);
```

```
  // Returning the set of the CPUs to be used by the application
  return result;
} // end cpu_set_t initializeCPUset(cpu_set_t* cpus)

/**
 * The function returns locations of tasks according to the mapping function
 *
 * @param id - Task id
 * @return Location of the task according to the mapping.
 *   Otherwise, returns -1
 */
int returnCoreByTask(int id) {
  switch (id) {
    case 0: return 25;
    case 1: return 18;
    case 2: return 2;
    case 3: return 1;
    case 4: return 9;
    case 5: return 10;
    case 6: return 17;
    case 7: return 26;
    case 8: return 34;
  } // end switch (id)
  return -1;
} // end int returnCoreByTask(int id)

// Initialize working environment
void initEnv() {

  // Initializing tids and tpids
  for (int i = 0; i < TASKS_NUM+1; i++) tid[i] = tpid[i] = 0;

  // Initializing running, termination and fault simulation signals
  for (int i = 0; i < TASKS_NUM+1; i++)
    run[i] = terminate[i] = fault[i] = false;

  // Initialize array of functions
  ...

  /* Creating and initializing memory queues */
  // for task with id = 1
  // Creating memory FIFO buffer
  Task_n1_memory_queue = memalign(MEM_ALIGN, sizeof(*Task_n1_memory_queue));
  assert (Task_n1_memory_queue != NULL);
  Queue_Task_n1_memory_init(Task_n1_memory_queue);

  // Declaring and initializing memory value to be sent to the buffer
  Task_n1_memory Task_n1_memory;
  rtP_Task_n1 rtp_n1;
  Init_Task_n1 (&rtp_n1);
  Task_n1_Init(&Task_n1_memory.rtDW_n1, &rtp_n1);

  // Providing initial packet id
  Task_n1_memory.pkg_id = 0;

  // Sending initial packets to the memory buffer
  for (int i = 0; i < FT_PKG_NUM; i++)
    Queue_Task_n1_memory_enqueue(Task_n1_memory_queue, Task_n1_memory);

  // similarly, for task with id = 4
```

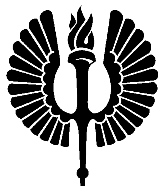
```
// Creating memory FIFO buffer
Task_n4_memory_queue = memalign(MEM_ALIGN, sizeof(*Task_n4_memory_queue));
assert (Task_n4_memory_queue != NULL);
Queue_Task_n4_memory_init(Task_n4_memory_queue);

// Declaring and initializing memory value to be sent to the buffer
Task_n4_memory Task_n4_memory;
rtP_Task_n4 rtP_n4;
Task_n4_Init_Structure(&rtP_n4);
Task_n4_Init(Task_n4_memory.Memory_PreviousInput,&rtP_n4);

// Providing initial packet id
Task_n4_memory.pkg_id = 0;

// Sending initial packets to the memory buffer
for (int i = 0; i < FT_PKG_NUM; i++)
  Queue_Task_n4_memory_enqueue(Task_n4_memory_queue, Task_n4_memory);

// finally, for task with id = 8
// Creating memory FIFO buffer
Task_n8_memory_queue = memalign(MEM_ALIGN, sizeof(*Task_n8_memory_queue));
assert (Task_n8_memory_queue != NULL);
Queue_Task_n8_memory_init(Task_n8_memory_queue);

// Declaring and initializing memory value to be sent to the buffer
Task_n8_memory Task_n8_memory;
rtP_Task_n8 rtp_n8;
Task_n8_Init_Structure(&rtp_n8);
Task_n8_Init(Task_n8_memory.Memory_PreviousInput,&rtp_n8);

// Providing initial packet id
Task_n8_memory.pkg_id = 0;

// Sending initial packets to the memory buffer
for (int i = 0; i < FT_PKG_NUM; i++)
  Queue_Task_n8_memory_enqueue(Task_n8_memory_queue, Task_n8_memory);

// Creating FIFO service buffers for synchronizing sequence numbers provided
// by the source tasks
// Providing initial values
ITD_data itd_pkg;
itd_pkg.pkg_id = itd_pkg.iter_cnt = 0;
for (int i = 0; i < INIT_TASKS_NUM; i++) {
    // Creating FIFO service buffer
  ITD_queues[i] = memalign(MEM_ALIGN, sizeof(*ITD_queues[i]));
  assert (ITD_queues[i] != NULL);
  Queue_ITD_init(ITD_queues[i]);

  // Sending initial packets to the service buffer
  for (int j = 0; j < FT_PKG_NUM; j++)
    Queue_ITD_enqueue(ITD_queues[i], itd_pkg);
} // end for (int i = 0; i < INIT_TASKS_NUM; i++)

// Creating communication buffers
Task_n2_Init_FIFOs();
Task_n3_Init_FIFOs();
Task_n4_Init_FIFOs();
Task_n5_Init_FIFOs();
Task_n6_Init_FIFOs();
Task_n7_Init_FIFOs();
Task_n8_Init_FIFOs();
```

```
} // end void initEnv()

// Create the agent and worker threads
void createThreads() {
  // Creating the cluster agent thread
  // Specifying thread parameters
  cla_data cla_args;
  cla_args.th_id = TASKS_NUM;
  // Number of tasks in the task graph
  cla_args.task_num = TASKS_NUM;
  // Number of columns in the platform
  cla_args.col_num = tmc_cpus_grid_width();
  // Creating the agent thread
  if (pthread_create(&tid[TASKS_NUM], NULL, Cluster_agent, (void *)&cla_args))
    tmc_task_die("THREAD MAIN: creating cluster agent has failed\n");

  // Attributes of threads
  pthread_attr_t thread_attr [TASKS_NUM];

  // Creating worker threads
  for (int i = 0; i < TASKS_NUM; i++) {
    if (pthread_create(&tid[i],
          &thread_attr[i], (void *)th_funcs[i], (void *)i))
      tmc_task_die("THREAD MAIN: creating thread #%d has failed\n", i);
  } // end for (int i = 0; i < TASKS_NUM; i++)
} // end void createThreads()

// Mapping threads to specific tiles according to the mapping function
void mapThreads() {

  // Mapping worker threads
  for (int i = 0; i < TASKS_NUM; i++) {

    // Waiting for a valid thread pid
    while (tpid[i] == 0){}

    // Mapping a corresponding task to a core
    if (tmc_cpus_set_task_cpu(returnCoreByTask(i), tpid[i]) < 0)
      tmc_task_die("THREAD MAIN: mapping of task %d has failed\n",i);
  } // end for (int i = 0; i < TASKS_NUM; i++)

  // Mapping the cluster agent thread
  if (tmc_cpus_set_task_cpu(16, tpid[TASKS_NUM]) < 0)
    tmc_task_die("THREAD MAIN: mapping of the cluster agent has failed");
} // end void mapThreads()

// Waiting for the threads to complete their tasks
void waitForThreads() {

  // Waiting for the source tasks to finish
  for (int i = 0; i < INIT_TASKS_NUM; i++) pthread_join(tid[i], NULL);

  // Terminating other than the source tasks
  for (int i = TASKS_NUM; i > -1; i--) terminate[i] = true;

  // Waiting for the regular tasks finalize
  for (int i = INIT_TASKS_NUM; i < TASKS_NUM; i++) pthread_join(tid[i], NULL);
} // end void waitForThreads()

/* The main task */
int main (int argc, char* argv[]) {
```

```
  // Initializing resources (CPUs)
  cpu_set_t cpus;
  cpus = initializeCPUset(&cpus);

  // Opening files containing input data
  ...

  // Initializing working environment
  initEnv();

  // Creating the cluster agent and worker threads
  createThreads();

  // Mapping threads to specific cores according to the mapping function
  mapThreads();

  // Mapping the main task to a specific CPU
  if (tmc_cpus_set_my_cpu(8) < 0)
    tmc_task_die("THREAD MAIN: mapping has failed");

  // When the configuration phase is complete, run the threads
  for (int i = TASKS_NUM; i >= 0; i--) run[i] = true;
  // and run the agent thread
  run[TASKS_NUM] = true;

  // Waiting for the threads to complete their tasks
  waitForThreads();

  // Finalizing the application
  return 0;
} // end int main (int argc, char* argv[])
```

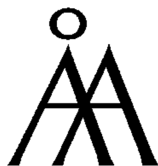
Turku Centre *for* Computer Science

Joukahaisenkatu 3-5 A, 20520, Turku, Finalnd | www.tucs.fi



University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics

Turku School of Economics
- Institute of Information Systems Sciences



Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

ISBN 978-952-12-3066-0
ISSN 1239-1891