Contract no. 248424
FP7 STREP Project

MADNESS

Methods for predictAble Design of heterogeNeous Embedded System with adaptivity and reliability Support

D5.4: Report on the integration of fault tolerance

Due Date of Deliverable	**8th March, 2013**
Completion Date of Deliverable	**8th March, 2013**
Start Date of Project	**1st January, 2010 - Duration 39 Months**
Lead partner for Deliverable	**USI**

Revision: v1.0

Project co-funded by the European Commission within the 7th Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contents

1. Summary

This document summarizes the activities performed in Work Package 5 during the third year of the project. The involved task is *Task 5.4: Integration of the fault tolerance support into the FPGA-based environment*.

The integration activities are based on previous work done in WP5 (reconfiguration policies and fault tolerant NoC components), WP3 (NoC platform with advanced message passing support) and WP6 (PPN middleware, software-based task migration support).

Section 2 presents the developed fault tolerance techniques based on online remapping and provides the details about the integration of the self-testing module and the task migration hardware module into tile architecture of the MADNESS NoC platform.

Section 3 describes the methodology implemented for the evaluation of the fault tolerant characteristics of the components of a Network-on-Chip, and its integration into a design space exploration flow.

To conclude, Section 4 presents interaction between WP5 and other work-packages of the MADNESS project.

2. Fault-aware Online Task Remapping

In this chapter, we describe the work done to enable continuity of service in the presence of permanent faults in processing elements when running PPN applications on NoC multiprocessors. As a basic assumption, online software-based self-testing is adopted for detecting permanent faults in processors. The fault model comprises stuck-at faults which may represent various types of errors at the processor level such as halting of the processor (crash), wrong computation and execution of arbitrary code in the program memory etc. The NoC components and memories are assumed to be designed so as to grant continuity of service even in the presence of (a predetermined set of) faultssuch that they exhibit a much smaller IP-level failure rate. We assume that one CPU fails at a time and that remapping has been completed before possibly a new fault appears in another CPU.

At the core of the proposed fault recovery techniques lies the idea of online remapping of tasks that previously ran on the faulty core onto fault-free ones. Differently than the spare core placement approaches [1] [2], which map all the tasks from the faulty core to one of the spare cores, our approach remaps the tasks mapped on the faulty tile to possibly different tiles that are already executing other tasks of the application. Moreover, the proposed mechanism is not based on active redundancy and concurrent error detection, thus presenting itself more favorable for embedded platforms with limited resources.

The proposed solution encompasses support for fault detection and fault recovery via online task remapping. Two fault tolerance mechanisms are proposed. First technique is a roll-forward recovery scheme, and the second one is based on rollback and implicit checkpoints.

The general overview of the system is shown in Figure 2.1. The text in blue corresponds to additions or modifications that enable the fault tolerance support. It can be seen that the proposed mechanisms involve hardware and software modifications on top of the MTOS, message-passing support and the NORMA-based NoC platform. The fault tolerant tile is obtained by adding two hardware modules to the tile architecture, namely the *Self Testing Module* (*STM*) and the *Task Migration Hardware* (*TMH*). The fault-aware remapping support is mainly a software module named *Remapping Manager* (*RM*) that is introduced to the run-time environment. STM helps in the detection of faults. TMH helps in notifying a remapping manager, which carries out the rest of the fault recovery process. Depending on the error propagation requirements, the TMH may also help migrating the state of tasks from the faulty tile. The template of the PPN process bodies is also slightly modified along with some changes in the PPN communication primitives.

In the remainder of this chapter, the two fault tolerance mechanisms will be explained in detail by describing hardware and software support (shown in blue in Figure 2.1) required by each mechanism.

2.1 Fault recovery without error propagation (FRWOEP)

The first technique, which we named as *Fault Recovery without Error Propagation* (*FRWOEP*), is a rollback-based technique at the granularity of a single iteration of a process. It relies on executing the self-testing routine at the end of each iteration of a PPN process. If the test is successful, then the results of the current iteration, which are to be written to the output FIFO channels of the process, are guaranteed to be correct. If the test fails, then the recovery mechanism is started with the help of TMH, which is responsible for notifying

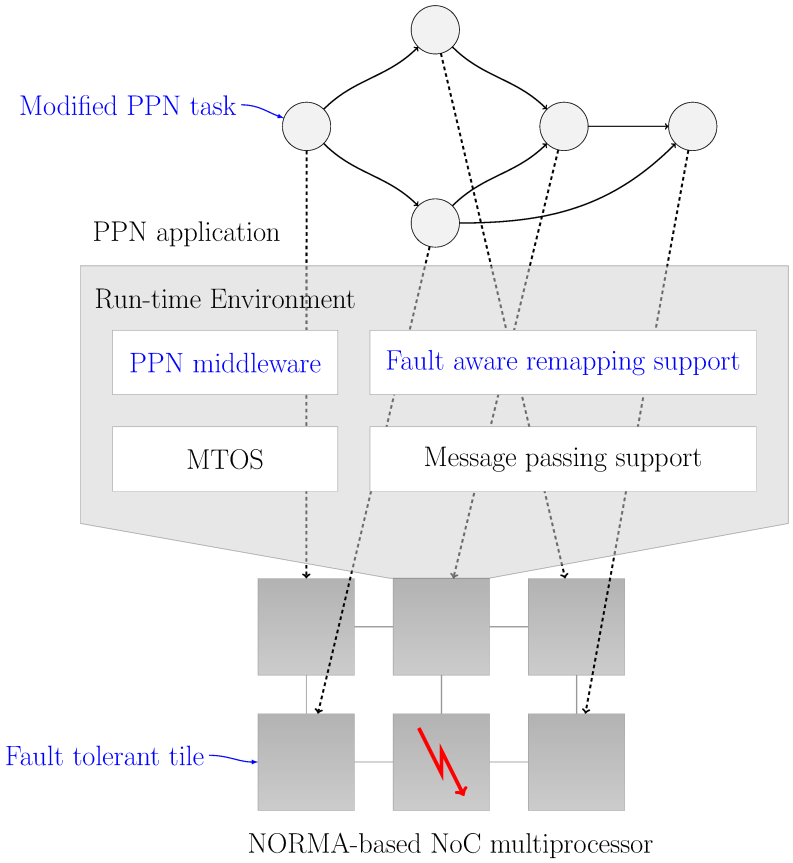


Figure 2.1: System overview with the fault tolerance support

the remapping manager, transfering the state of the tasks which consists of the iterators of the tasks as well as the tokens in the input and output FIFO channels.

2.1.1 Modifications to the PPN processes

As a part of the FRWOEP support, the process bodies are required to be modified. Algorithm 2 shows how the basic process body shown in Algorithm 1 is modified to support the FRWOEP mechanism.

Algorithm 1 A basic PPN process

```
1: for (i=0 ; i<M; i++) do
2:     for (j=0 ; j<N; j++) do
3:         read(in, CH1);
4:         out = f(in);
5:         write(out, CH2);
6:     end for
7: end for
```

All PPN processes have the same code structure (as shown in Algorithm 1). Nested loops iterate, for a given number of times, the body of the process, which is split into three main parts. First, the process reads the input data tokens from (a subset of) the input channels. This is represented by the *read*() statements in the algorithm. Second, the process function (*f*) produces the output tokens by processing the input tokens. Finally, the output tokens are written to (a subset of) the output channels represented by the *write*() statements.

Algorithm 2 The PPN process template in the FRWOEP mechanism

```
 1: if (migration) resumeState;
 2: for (i=i0 ; i<M; i++) do
 3:    for (j=j0 ; j<N; j++) do
 4:       acqData(CH1);
 5:       read(in, CH1);
 6:       setTimer();
 7:       out = f(in);
 8:       selfTest();
 9:       write(out, CH2);
10:       relSpace(CH1);
11:    end for
12:    reset j0;
13: end for
14: return
```

We comment and motivate the PPN process structure shown in Algorithm 2 in the following. When the thread starts, in line 1, it checks if the *migration* flag is set. If the migration flag is false, it means that the process starts from scratch, with empty input and output FIFOs and $i_0 = j_0 = 0$. Otherwise, it means that a migration has been performed, so the process state is reloaded.

Since the PPN model definition requires a stateless process function, for example *f* in Algorithm 1, i.e., a function whose execution does not depend on the previous iterations, the state of a PPN process is represented only by:

- the content of its input and output FIFOs;
- its iterator set, namely the values of the nested loop iterator variables, see (*i, j*) in Algorithm 2, lines 2 and 3;

When a function requires to have a state, it is represented in the PPN model by a stateless function with FIFO self-edges, which represent the function state.

Due to the request-based flow control policy used for implementing the KPN semantics on the NoC platform, in addition to the above-listed items, pending requests on the outgoing channels from the faulty processing element also constitute part of the state to be recovered. All three state components listed above are transferred from the faulty tile to the remapping manager upon fault detection.

Lines 2 and 3 differ from the basic process structure in Algorithm 1 because the iterators inside the for loops do not start from zero in case of migration. Instead, they start from the values i_0 and j_0, which represent the iteration at which the process was interrupted by the fault detection while running on the source tile. After the first complete execution of the inner for loop, starting from j_0 , the value of j_0 is set to zero in line 12 such that the next execution of the inner loop starts correctly with $j = 0$.

The *read* communication primitive is different from the one used in the basic process structure. It is split into three separate operations (see lines 4, 5, 10). First, the input channel (CH1) is tested to verify the presence of an available data token, using the acquireData function (*acqData*(*CH1*) in line 4). Then, the token is actually copied from the software FIFO to the input variable which will be processed by the process function *f*. The copy operation is performed in line 5. However, differently from the normal read primitive, the memory locations occupied by the read token are not released immediately. The actual release, which consumes the data from the FIFO by increasing the read pointer, takes place only in line 10 (*relSpace*(*CH1*)). This way, if a fault is detected before the release instruction, the process can be correctly resumed on the destination tile since it will read again the same input token, because the read pointer is not changed. Note

that, in case of multiple input or output channels, the release operations should be grouped together and placed right after the main body of the process, in order to guarantee a consistent process state.

In order to tolerate crash errors, which result in the processor not executing any instructions, a watchdog timer is set to expire within a time limit (line 6). This time limit is greater or equal to the sum of the worst case execution time of process function (*f*) and the self-test routine. In the case that the program counter reaches the end of the self-test routine, the timer is reset before it times out. Otherwise, the timer signals the TMH module about the fault detection. The faults can be more subtle and may result in computational errors. Such faults are detected by running a self-test routine as shown in line 8. In the case that the self test routine produces a different output than normal, it is detected by the self-testing module, which in turn signals the fault detection to the TMH.

If a crash fault occurs between the end of self-test routine (line 8) and setting of the timer (line 6), it cannot be detected. Moreover, if a fault occurs just after executing a self-test succesfully (line 8), it may result in a corrupt data to be written to the channel while executing line 9. However, we can argue that time window (thus the probability) for such cases is very small as the majority of the time will be spent in executing the process function *f* and the self-test routine.

2.1.2 Fault-aware remapping support

The actors involved in the fault recovery procedure are as follows:

- *processing element* of the source tile (i.e., the tile that experiences the fault);
- *self-testing module* in the source tile;
- *task migration hardware* module in the source tile;
- *remapping manager* which runs on one of the fault-free tiles;
- *predecessor and successor tile*(*s*): the tile(s) which has a producer or a consumer task of any of the tasks on the source tile;
- *new tile*(*s*): the tiles that will run at least one of the tasks on the source tile after fault recovery;
- *other tile*(*s*): the fault free tile(s) that are neither the source tile, a new tile, a predecessor or a successor tile.

Figure 2.2 shows the sequence diagram of the recovery procedure. After executing the self-testing routine, if a fault is detected on the source tile, the STM issues a fault detection signal to the TMH. The TMH isolates the faulty processor. The TMH reports the fault to the RM by sending a fault detection message (the selection of the RM tile is explained in section 2.2.1). The RM calculates the new mapping of the tasks using the remapping heuristics (described in Deliverable 5.2 and 5.3). The RM informs the predecessor/successor tile(s) and the other tiles about the new mapping of the tasks to update their middleware tables. The predecessor/successor tiles send a flush message to the faulty node to make sure that there are no tokens or requests still travelling to the faulty tile. Upon the reception of flush messages from predecessor/successor tiles, the TMH responds with a flush message to make sure that there are no tokens or requests still travelling to predecessors or successors. Then the TMH sends to the RM the state of the tasks, which consists of (i) the iterators of the loops in the case of PPN tasks, (ii) the information of the FIFO channels (pending requests and number of tokens in the FIFO channels), (iii) the tokens in the input and output FIFO channels. After the RM receives the tasks' state from the TMH, the RM sends these data to the new tile(s) according to the

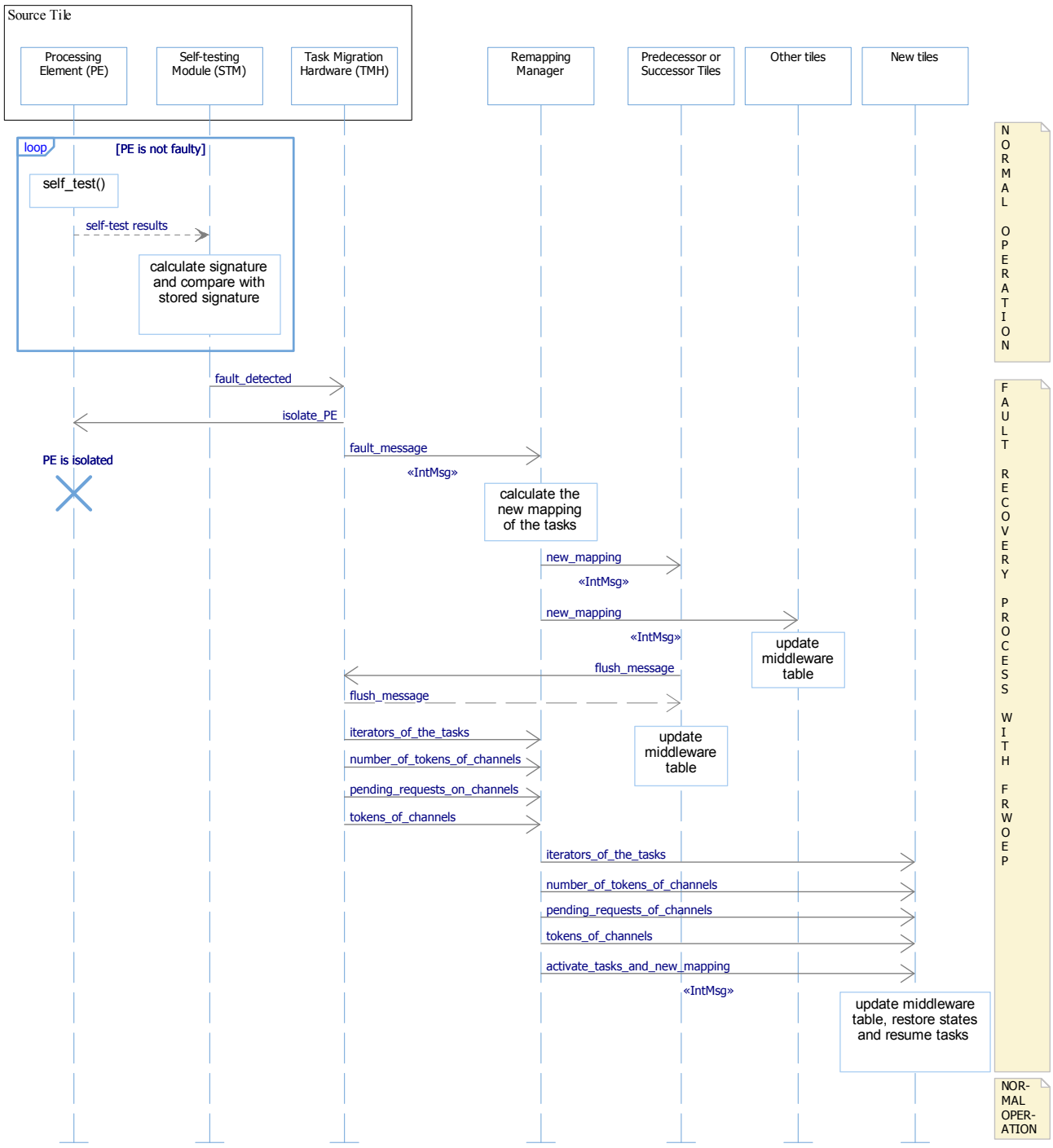


Figure 2.2: FRWOEP sequence diagram

new mapping decision. Then the RM sends to the new tiles a task activation message along with the new mapping information allowing updating of their middleware tables and the activation of migrated tasks. This finalizes the fault recovery procedure.

In order to support the FRWOEP mechanism, the self-test and task migration hardware modules have been integrated to the tile architecture as shown in Figure 2.3. A detailed description of STM and TMH-FRWOEP is given in the two following sections.

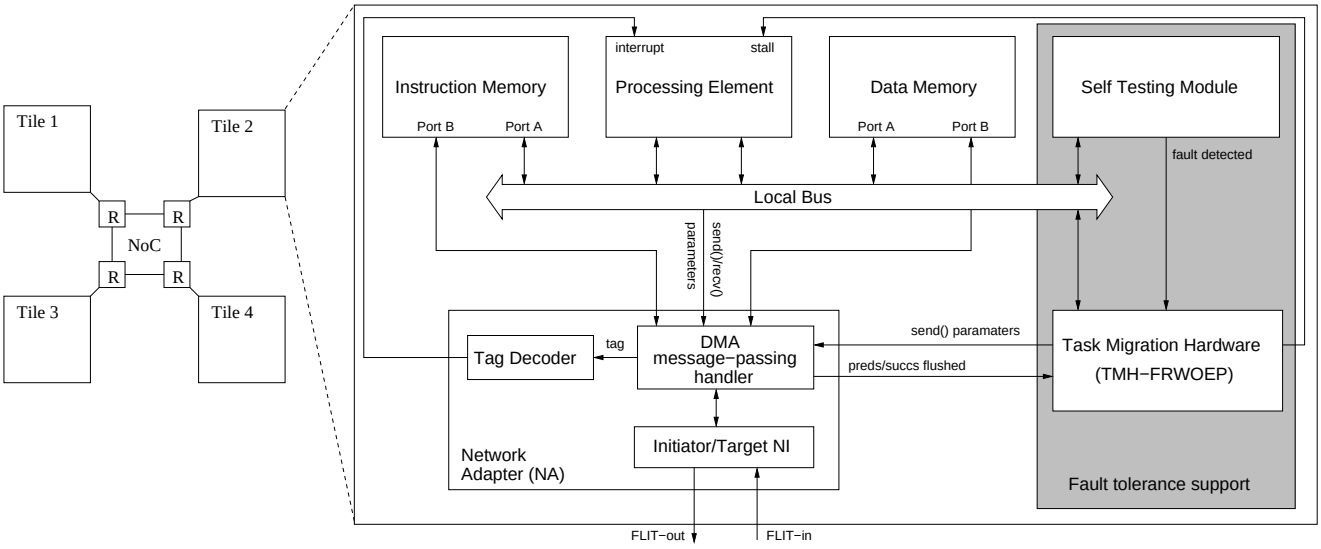


Figure 2.3: Block diagram of the fault tolerant tile

2.1.3 Self Testing Module

The *Self-testing module* (STM) is a hardware module that supports the execution of the testing routines in each tile of the Network-on-Chip (NoC) which includes a processing element. Hardware modules are needed whenever it is not possible to implement techniques such as distributed testing [3], which relies on the availability in the same platform of several instances of general purpose processors. The STM checks the results of the software testing routines and activates the procedures for task remapping and migration, as described in Section 2.1.2 and 2.2.1. The STM is in charge of collecting the outputs of the processor when executing the software routine, of calculating the signature of the outputs, and of checking it against the expected signature, in order to verify the correctness of the routine execution. The testing routine is stored directly in the processor local memory, and it is scheduled by the operating system. Results of its execution are sent directly to the STM.

Figure 2.4 shows the architecture designed for supporting the execution of the software testing routines, which is intended to work as a wrapper around the processor for helping detecting permanent faults in it. With respect to the module introduced in Section 2.2. of Deliverable D5.3, we improve the architecture of the module by reducing to almost 0 the time overhead due to the calculation of the signature, and by reducing the area overhead by removing the FIFO buffer. As discussed in Section 2.1.2 and 2.2.1, the STM module was integrated into the platform.

The STM is memory-mapped on the tile's system bus and it can be directly accessed by the processor. In the *prologue* of the software testing routine, the expected signature is copied in the *slv_signature* registers. Then, the STM is activated, by writing into the *slv_start_stop* register. When active, the STM samples the outputs of the device under test (DUT) (i.e., the processor) and copy them in the *slv_data_in* registers. For

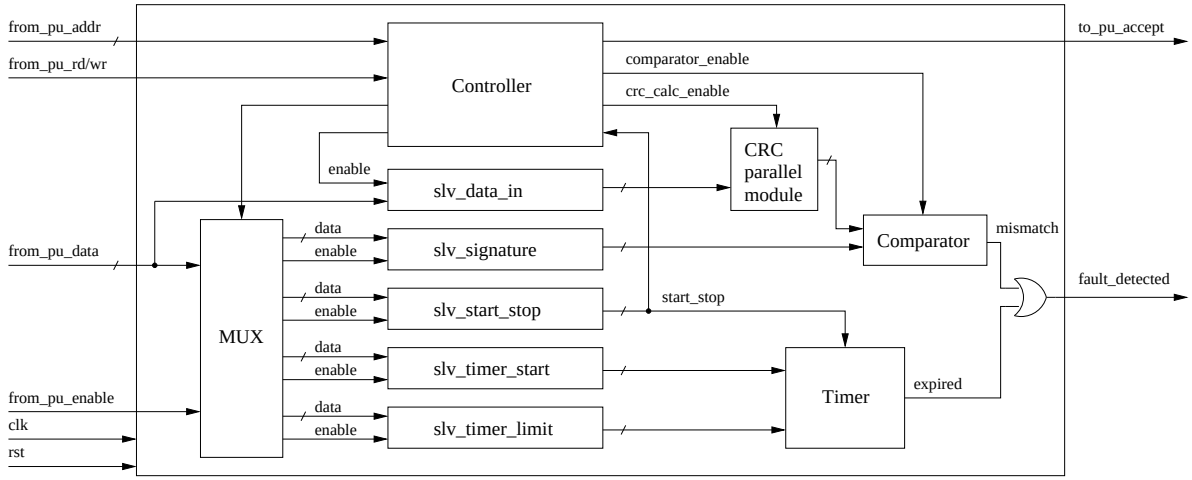


Figure 2.4: Overview of the STM architecture

Table 2.1: Xilinx synthesis results for the area of the STM

FPGA elements	# used
Slice registers	163
Slice LUTs	285
Occupied slices	126

each new data inserted in the *slv_data_in* registers, the CRC parallel module calculates immediately the value of the signature for the samples received up to that moment. At the end of the execution of the software routine, the STM is stopped by writing into the *slv_start_stop* register. The STM compares the value stored into the *slv_signature* registers with the final signature calculated by the CRC parallel module. If the two values does not match, the *fault_detected* signal is set to '1' for a clock cycle.

The STM also supports detecting crash errors. A fault may result in the processor not executing the self-testing routine at all. A timer is introduced inside the STM to detect such errors. In the FRWOEP case, the time limit is written inside the *slv_timer_limit* register at the beginning of each iteration of the PPN process and the timer starts to count down from that value. If the self-testing routine does not complete within the time limit, a crash error is assumed and *fault_detected* signal is raised. Otherwise, if the self-testing routine is completed (inferred by a write into the *slv_start_stop* register), the timer is stopped until the next time it is set and started.

Table 2.1 shows area overhead of the STM in the case of synthesis for FPGA by using Xilinx ISE. The table reports number of slice registers (representative of memory), slice LUTs (representative of combinational logic), and occupied slices (representative of total area). The STM introduce an overhead of around 6% in the number of used slice registers and of 11% in the number of LUTs, with respect to the area occupied by a Microblaze, the standard Xilinx soft-core processor. The Microblaze taken as reference is a three-stage pipelined processor with 2KBytes of I-Cache and 4KBytes D-Cache, and MMU and 32-bit integer multipliers enabled. In this configuration, the Microblaze occupies 2,530 slice registers and 2,536 LUTs.

2.1.4 Task Migration Hardware (TMH-FRWOEP)

Figure 2.5 shows the internal structure of the TMH-FRWOEP. It consists of the following components: TMH register file, TMH controller, TMH shmpi_send registers, multiplexers. Each component will be discussed in details in the following subsections. Firstly, an interface specification is provided.

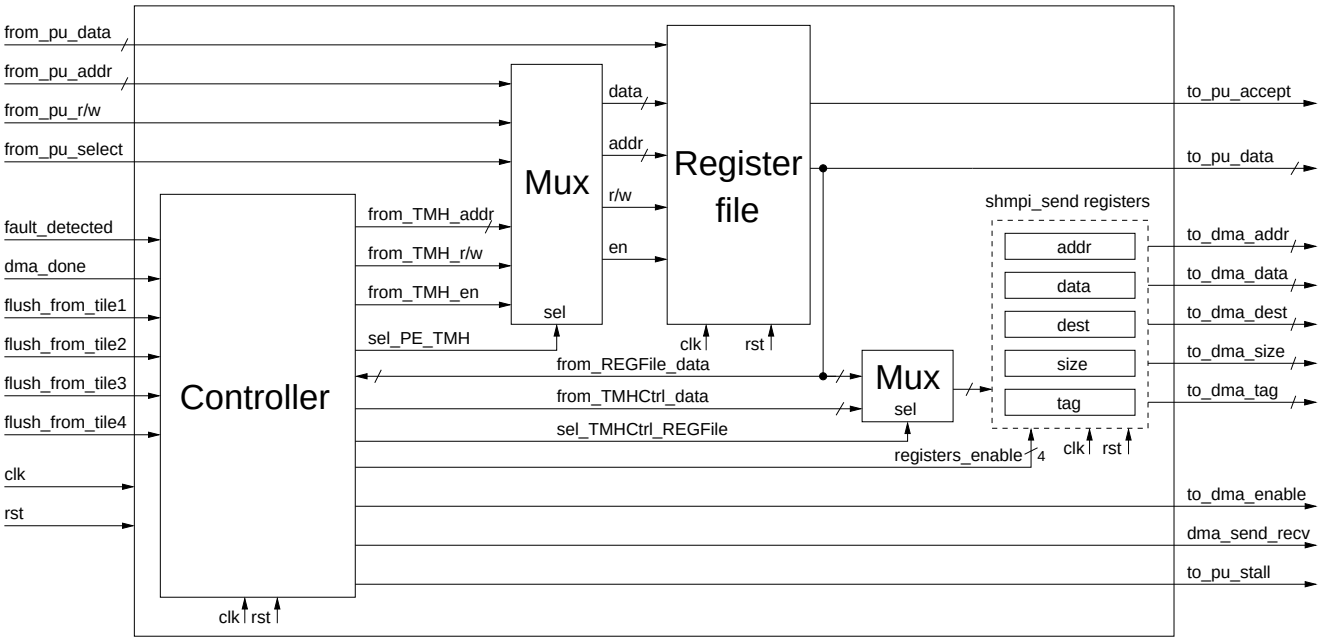


Figure 2.5: TMH-FRWOEP block diagram

The module is synchronous with an asynchronous reset, so the input ports of clock and reset are present:

- *clk:* Active on positive edge.
- *reset:* Asynchronous, active high.

The input port that connects the STM to the TMH:

- *fault_detected:* If sampled high, indicates that the STM has detected a fault and the task migration procedure should be started.

Ports for communicating with the processing element:
Inputs:

- *from_pu_addr:* Address from the processor. This will be used to store the data inside the TMH, the allowed range of addresses is 0x84000000 – 0x84FFFFFF.
- *from_pu_select:* If sampled high, indicates that the processor is trying to write/read data to/from the TMH.
- *from_pu_rnw:* This signal will be sampled only when *from_pu_select* is high. If *from_pu_rnw* is sampled high indicates a read operation from the TMH, else indicates a store operation.
- *from_pu_data:* The data from the processing element to be stored in the TMH, when *from_pu_select* is high and *from_pu_rnw* is low.

Outputs:

- *to_pu_stall:* This signal isolates the processing element. When the TMH sets this signal to high, the processing element is stopped.
- *to_pu_accept:* It indicates to the processor that the performed load/store operation is completed successfully.

- *to_pu_data:* The data sent back to the processor when performing a read from the TMH. The answer is considered valid by the processor only when *to_pu_accept* signal is high.

Signals to communicate with the DMA message-passing handler (the outputs correspond to the parameters of the SHMPI_send() primitive):
Inputs:

- *dma_done:* When high, signals that the DMA operation has been completed and the data has been received successfully.

Outputs:

- *to_dma_addr:* It indicates from which local address to load or store the data.
- *to_dma_data:* When the address is in the range of the TMH (i.e. 0x84000000 – 0x84FFFFFF), the payload of the message is from the TMH.
- *to_dma_size*: It indicates the size of the payload to be transferred in bytes.
- *to_dma_dest:* It indicates the ID of the destination tile of the sent message.
- *to_dma_tag:* It contains the message tag, this tag defines from which class the message is.
- *to_dma_enable:* When high, it indicates that all the *to_dma_*< *x* > signals are valid.
- *dma_send_recv:* When high, it indicates a DMA send operation. A DMA send operation is initiated when both *to_dma_enable* and *dma_send_recv* are high.
- *flush_from_tile_*< *x* >*:* If sampled high, it indicates that the NI has received a flush message from tile < x >.

Internal signals:

- *from_TMH_addr:* Address from the TMH controller. It is used to define an address inside the register file at which a read/write operation will be executed.
- *from_TMH_enable:* If sampled high, it indicates that the TMH controller is trying to store/read data from/to the TMH register file.
- *from_TMH_rnw:* This signal will be sampled only when *from_TMH_enable* is high. If *from_TMH_rnw* is sampled high, it indicates a read operation from the TMH register file, else it indicates a store operation.
- *from_TMH_data:* The data from the TMH controller to be stored in the TMH register file, when *from_TMH_enable* is high and *from_TMH_rnw* is low.
- *sel_PE_TMH:* When sampled high, the register file is driven by the inputs from the TMH controller (TMH_DRIVEN) otherwise it is driven by the inputs from the processor (PU_DRIVEN).
- *sel_TMHCtrl_REGFile:* When sampled high, the input data for the TMH shmpi_send registers is from the TMH controller (TMH_CTRL_UNIT_DRIVEN) while when sampled low, the input data are from the register file (REG_FILE_DRIVEN).

- *registers_enable:* They are the enable signals from the controller to the TMH shmpi_send registers to write the input data on the positive edge of the *clk*.
- *from_TMHCtrl_data:* The data from the TMH controller to be written in the TMH shmpi_send registers, when *sel_TMHCtrl_REGFile* is high.
- *from_REGFile_data:* The data from the register file to be written in the TMH shmpi_send registers, when *sel_TMHCtrl_REGFile* is low or to be written in the TMH controller when a read operation is issued.

The TMH register file

The TMH register file is a 32-bit register file where reading and writing operations takes only one cycle. In the case of a write operation, the *addr* port identifies the address where the *data* should be written when the *enable* signal is high and *rd*/*wr* signal is low. While in the case of a read operation, the *addr* port is set with the address of the register to be read with both *enable* signal and *rd*/*wr* signal are high. The registers are memory-mapped registers with range from 0x84000000 to 0x84FFFFFF. All the needed information during the migration operation is stored inside the register file. During the normal operation of the processor, the content of the register file is kept updated by writing the current status to its registers. Table 2.2 shows the content of the register file.

Table 2.2: The structure of the TMH-FRWOEP register file

address	Register name
0x84000000	these two addresses are reserved for development purposes
0x84000004	
0x84000008	TMH_FAULT_MSG_ADDR
0x8400000C	TMH_FAULT_MSG_DEST
0x84000010	TMH_FAULT_MSG_SIZE
0x84000014	TMH_FAULT_MSG_TAG
0x84000018	TMH_SUCC_PRED_TO_THIS_TILE
0x8400001C	TASKS_MAPPED_TO_THIS_TILE
0x84000020	TMH_ITERATORS_BASE_ADDR
0x84000024	TMH_ITERATORSET_SIZE
0x84000028	TMH_FLUSH_ITERATORS_MSG_TAG_BASE_VALUE
0x8400002C	CHANNELS_MAPPED_TO_THIS_TILE
0x84000030	TMH_CHANNELS_MSG_TAG_BASE_VALUE
0x84000034	TMH_CHANNELS_REQUEST_BASE_ADDR
0x84000038	TMH_CH_0_NBTOKENS
0x8400003C	TMH_CH_0_TOKEN_SIZE
0x84000040	TMH_CH_0_FIFO_BUF_PTR
0x84000044	TMH_CH_1_NBTOKENS
0x84000048	TMH_CH_1_TOKEN_SIZE
0x8400004C	TMH_CH_1_FIFO_BUF_PTR
0x84000050	TMH_CH_2_NBTOKENS
0x84000054	TMH_CH_2_TOKEN_SIZE
0x84000058	TMH_CH_2_FIFO_BUF_PTR
0x8400005C	.....
0x84000060	.....
0x84000064	.....

The description of each register in the TMH-FRWOEP register file:

- *TMH_FAULT_MSG_ADDR:* It stores the starting address of the fault detection message which is stored somewhere in the data memory.
- *TMH_FAULT_MSG_DEST:* It stores the tile ID where the remapping manager is running on.
- *TMH_FAULT_MSG_SIZE:* It stores the total size of the fault detection message in bytes.
- *TMH_FAULT_MSG_TAG:* It stores the tag value for the fault detection message.
- *TMH_SUCC_PRED_TO_THIS_TILE:* It stores which tiles are the predecessors or the successors of this tile such that each tile is represented by one bit. For example, if $Tile_0$ and $Tile_3$ are the predecessors or the successors, then *TMH_SUCC_PRED_TO_THIS_TILE* = 0x00000009 = 1001_2.
- *TASKS_MAPPED_TO_THIS_TILE:* It defines which tasks are running on this tile. One bit is used to represent each task, so up to 32 tasks could be mapped on a single tile. For example, if T_1,T_4,T_5 are mapped to a tile then the value of *TASKS_MAPPED_TO_THIS_TILE* is 0x00000032 = 110010_2.
- *TMH_ITERATORS_BASE_ADDR:* The iterators of each task are stored in the memory. The base address of this structure is saved in *TMH_ITERATORS_BASE_ADDR*.
- *TMH_ITERATORSET_SIZE:* It defines how many bytes are the iterators of the tasks.
- *TMH_FLUSH_ITERATORS_MSG_TAG_BASE_VALUE:* It keeps the tag base value (0x1200) for both flush and iterators messages such that the tag for the flush message is 0x1200 while for the iterators message is (0x1200 + 2 + *task_num*).
- *CHANNELS_MAPPED_TO_THIS_TILE:* It defines the channels that has this tile as a source or a destination in the middleware table. One bit is used to define each channel, so up to 32 channels could be supported by a single tile. For example, if $Tile_1$ is the producer or the consumer for these channels Ch_2, Ch_3 and Ch_4 then the value of *CHANNELS_MAPPED_TO_THIS_TILE* register is 0x0000001C = 11100_2.
- *TMH_CHANNELS_TAG_BASE_VALUE:* It stores the tag base value (0x8000) for the channel messages (request messages, number of tokens messages and tokens messages) while the tag base value for a channel is defined (0x8000 + *ch_num* * 0x100). For Ch_2, the tag base value is 0x8200, so the tag value of the request message is 0x8200, the tag value of the number of tokens message is 0x8201 and the tag value of the tokens message is 0x8202.
- *TMH_CHANNELS_REQUEST_BASE_ADDR:* When a request for tokens arrives from another tile for a given channel, then the request is stored in a data structure saved in the local memory such that one memory location is kept for each channel. The base address of this structure is saved in *TMH_CHANNELS_REQUEST_BASE_ADDR* register.

For each channel, we store three values in the TMH register file:

- *TMH_CH_< x >_NBTOKENS:* the number of tokens in the buffer for channel < x >.
- *TMH_CH_< x >_TOKEN_SIZE:* the token size for channel < x >.
- *TMH_CH_< x >_FIFO_BUF_PTR:* saves the starting address of the FIFO buffer where the tokens of the channel < x > are stored in the memory.

The middleware primitives are modified to keep the TMH register file up-to-date regarding the value of the number of tokens contained in the FIFO channels.

TMH shmpi˙send registers

The communication between the tiles is based on the message passing paradigm. Two versions of the send function is implemented in hardware: one is used when the data to be sent comes from the local memory while the other is used when the data to be sent comes from the TMH itself. To issue a send message by the TMH, firstly; the values of the parameters of the send function should be loaded and stored in the corresponding TMH shmpi_send registers then the *dma_send_recv* and *to_dma_enable* signals are set high for one clock cycle to initiate the send operation. when the sent data is from the TMH itself, then the value of the address register should be from the TMH address range (i.e. from 0x84000000 to 0x84FFFFFF) to indicate that the sent data comes from the TMH not from the local memory. This function is limited to transfer only one register value (i.e., the *data* register) per message so the value of the size register is equal to one.

The TMH controller

Figure 2.6 shows the FSM of the TMH-FRWOEP controller. The FSM of the TMH-FRWOEP controller is a hierarchical state machine diagram where the state with a double circle notation represents a super state. After asynchronous reset signal, the TMH runs in the *idle state* where the register file is kept updated by the processing element by writing the latest information to its registers. When the *fault_detected* signal is set high, the TMH sets the *to_pu_stall* signal to stop the faulty processor. When the TMH succeeds to send a fault detection message to the remapping manager, it waits for flush messages from the predecessor and successor tile(s). when a flush message arrives from a certain tile, it sets one of the *flush_from_tile_< x >* signals and sets one bit in the *Flush_reg* register. When *Flush_reg* is equal to the *TMH_SUCC_PRED_TO_THIS_TILE* register then the TMH is able to respond with flush messages through the *Sending flush MSGs to Preds and Succs* state. The TMH sends to the RM the states of the tasks that were running on this tile through *Sending the state of the tasks to RM* state followed by the channels info and tokens if exists through the states *Sending the pending requests to RM*, *Sending the NB_tokens of the channels to RM* and *Sending the Ch_tokens to RM*, respectively. When all the channels data are transferred successfully to the RM, the TMH reaches to the final state *Migration process done successfully*. The internal FSM diagrams of the super states in Figure 2.6 is provided in Appendix A.

2.1.5 Experimental results

We tested the FRWOEP mechanism with a single fault scenario (initial mapping shown in Figure 2.16(a) and the fault injection on Tile 1 as shown in Figure 2.16(b)). Figure 2.7 shows the finishing time of each phase of the FRWOEP mechanism averaged over 13 different fault injection times. The average recovery time is 38115 cycles. 46% of this time is taken by the phase in which the state of tasks and channels from the TMH is received by the remapping manager. In this particular scenario, the RM is also the new tile where DCT is being remapped to. Therefore, the phase of transfering the state to the new tile does not take as much time.

In the FRWOEP mechanism, the self-test is executed at each iteration of each process. Therefore the duration of the self-test routine influences majorly the overhead of the technique during normal operation. Since we have not implemented real self-test routines for the Microblaze processor, we report analytical results of this overhead with respect to various execution times of the self-test routine ranging from 10k to 100k cycles. The mapping used in the calculations is the one of Figure 2.16(a). As shown in Figure 2.8, the overhead is linear with respect to the self-test duration and changes from 7.7% to 71%. Naturally designing

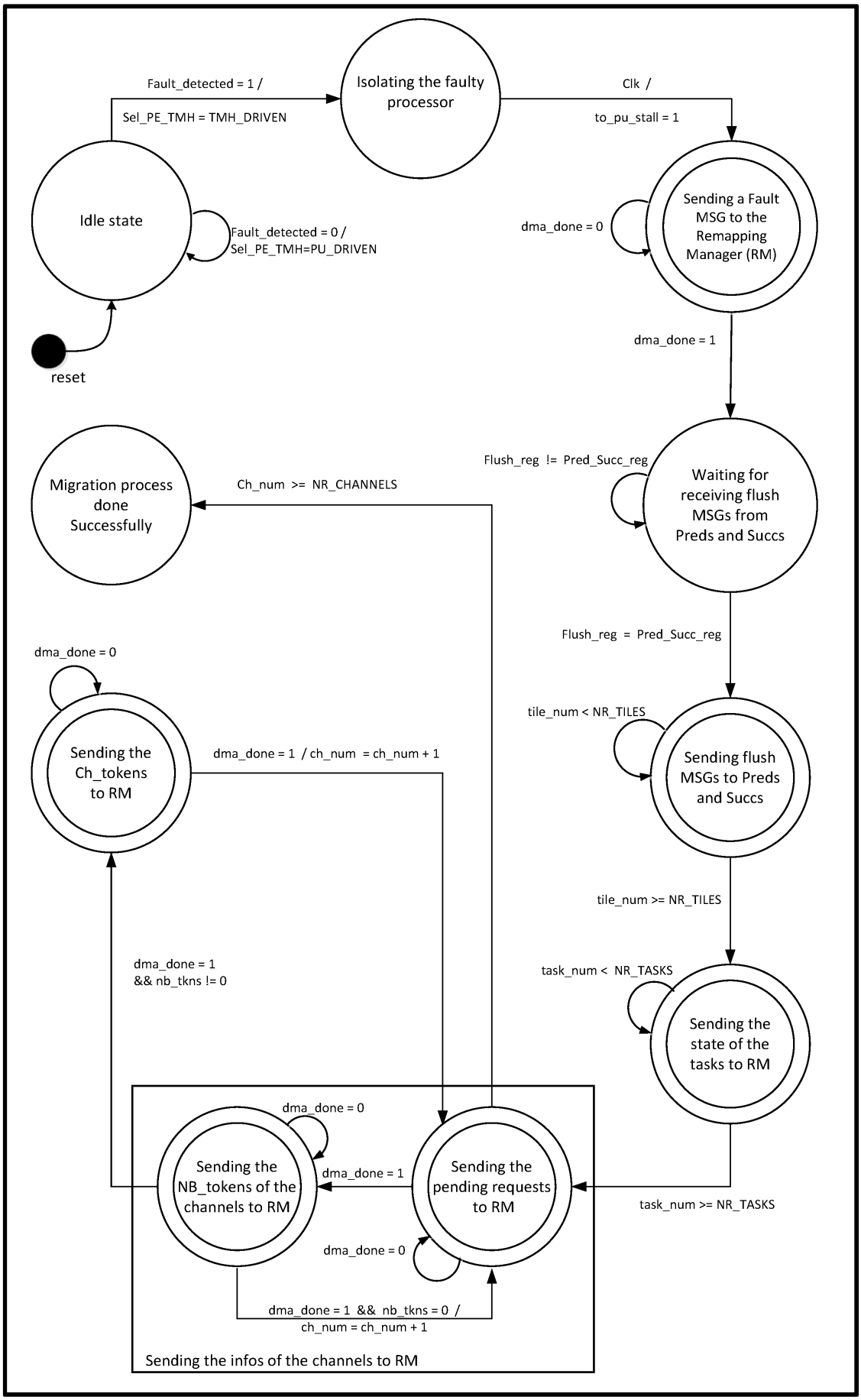


Figure 2.6: The finite state machine of the TMH-FRWOEP controller

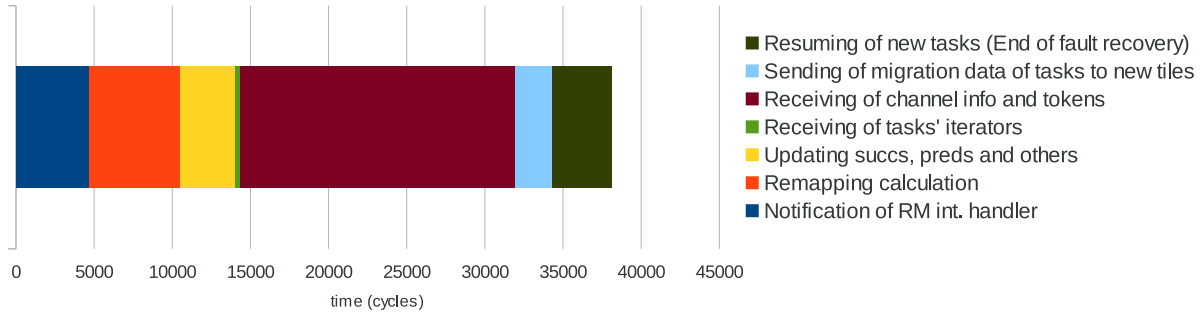


Figure 2.7: The time of fault recovery actions with the FRWOEP mechanism

a self-test routine involves a trade-off between its execution time and fault coverage ratio. Selecting 40k cycles as a typical duration for the self-test (taken from [4] for a processor of supposedly similar complexity), we see that the overhead would be 29%. This overhead is due to additionaly workload inflicted upon the critical node that determines the throughput of the whole application.

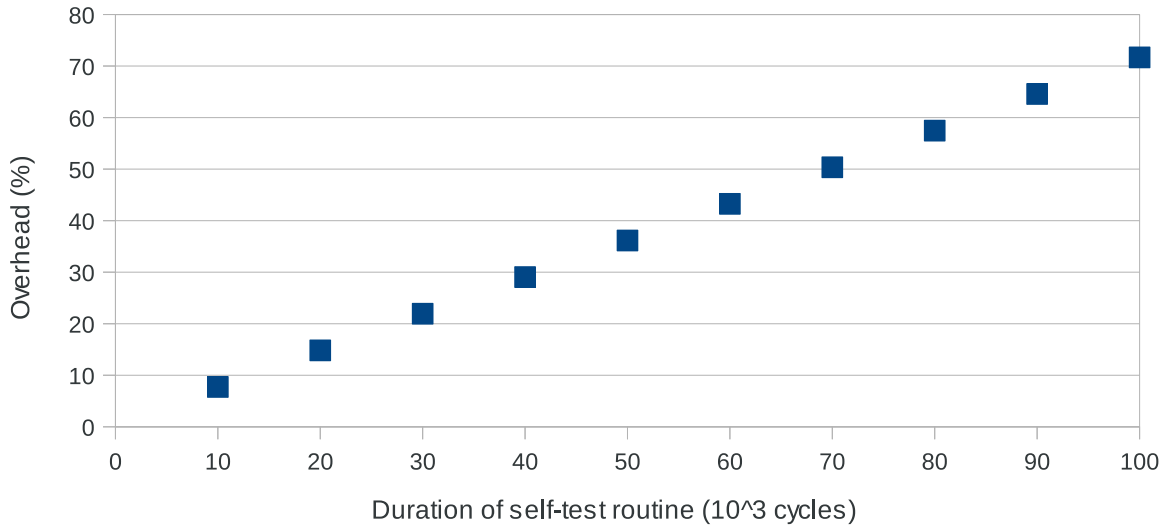


Figure 2.8: Performance overhead with respect to the duration of the self-test routine with the FRWOEP mechanism

As the TMH for FRWOEP requires channel information to be stored in its internal register file, its area varies according to the number of channels available in a given application. In order to see the change in the area, we have synthesized the TMH using Xilinx ISE for different number of channels, i.e., 4, 8, 16, 32. Figure 2.9 reports slice registers (representative of memory), slice LUTs (representative of combinational logic) and occupied slices (representative of total area). The area of TMH-FRWOEP increases almost linearly with increasing number of channels mainly due to the increasing size of the register file.

Table 2.3 shows the synthesis results for area with respect to the base tile, adaptive base tile (i.e., with the Message Passing Handler module) and the fault tolerant tiles of FRWEP and FRWOEP mechanisms with STM and TMH modules. Figure 2.10 shows the area overhead of the two fault tolerant tile types with respect

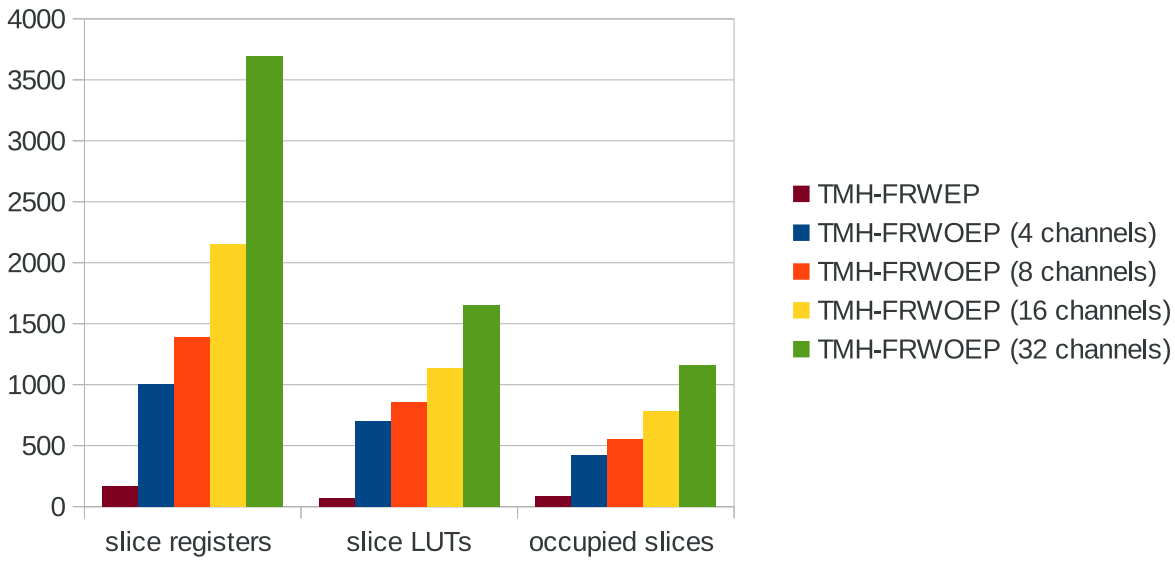


Figure 2.9: Synthesis results for the area of the TMH with both FRWOEP and FRWEP mechanisms

Table 2.3: Area synthesis results of TMH and STM with both FRWOEP and FRWEP mechanisms as well as the base tile architecture

	Occupied Slices	Slice Registers	Slice LUTs
Base tile	1983	4567	6430
Base tile + MPH	2632	6579	8985
Base tile + MPH + TMH-FRWOEP (4 channels)	3052	7581	9687
Base tile + MPH + STM + TMH-FRWOEP (4 channels)	3178	7744	9972
Base tile + MPH + TMH-FRWEP	2717	6749	9051
Base tile + MPH + STM + TMH-FRWEP	2843	6912	9336

to the adaptive base tile. For the FRWOEP mechanism that supports 4 channels (i.e. necessary number of channels for MJPEG), the area overhead compared to the adaptive tile is 20.7%. This overhead would become more relevant if the number of supported channels is increased. The area discussion of the FRWEP mechanism is done in section 2.2.5.

Figure 2.11 reports the maximum clock frequency achievable with changing number of channels. As can be seen, the frequency remains almost constant. Therefore, TMH is scalable with respect to the clock frequency.

2.2 Fault recovery with limited error propagation (FRWEP)

The second fault recovery technique is *Fault Recovery with Error Propagation* (*FRWEP*). In streaming applications, the stream consists of a sequence of data items with increasing indices. The main idea behind this technique is roll-forwarding to the next stream index upon the detection of the fault. Thus the data item will be processed partially and an incomplete output will be produced. However, the output will be as expected starting with the next data item in the stream. The FRWEP mechanism is applicable only if such incomplete output or incorrect output for a limited time is allowed. Unlike FRWOEP, which does self-testing

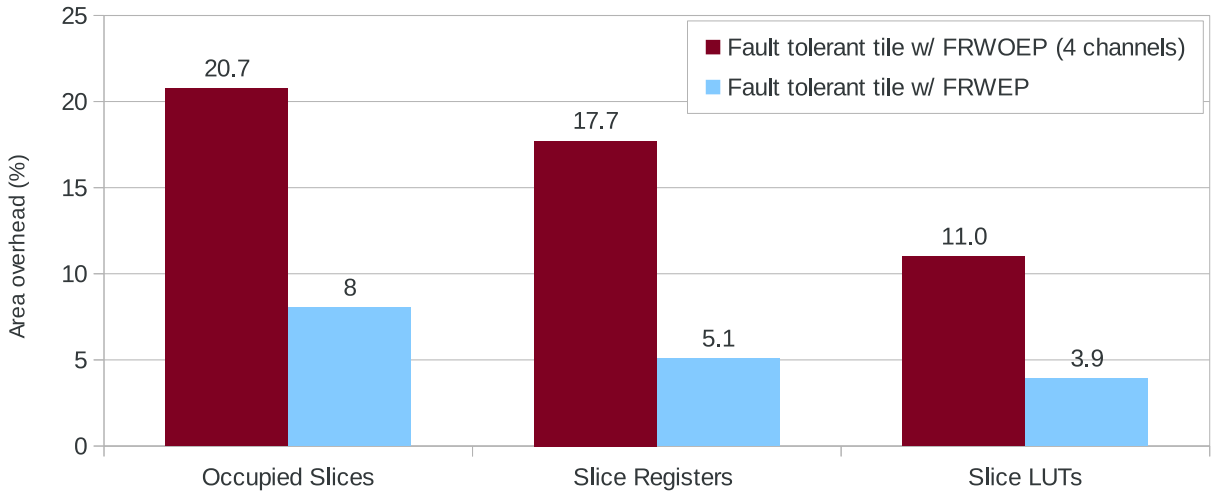


Figure 2.10: Area overhead of TMH and STM with both FRWOEP and FRWEP mechanisms in comparison to the adaptive base tile architecture

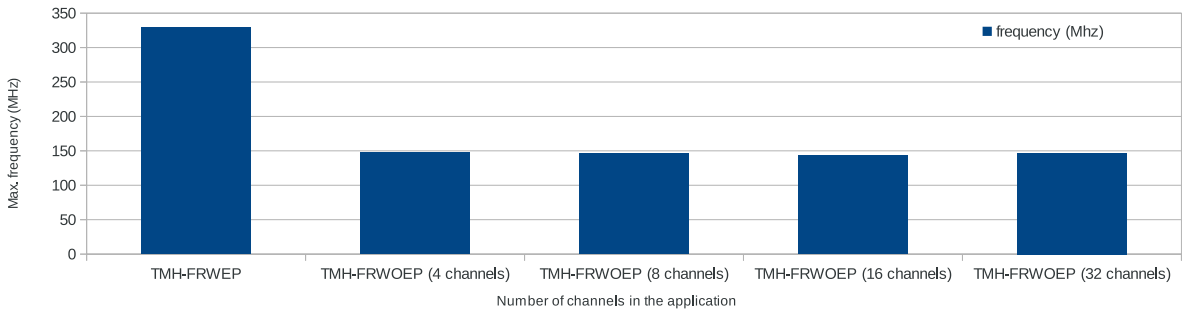


Figure 2.11: Synthesis results for maximum achievable clock frequency of the TMH with both FRWOEP and FRWEP mechanisms

at each iteration of a process, FRWEP relies on online periodical software-based self-testing. Therefore, incorrect results may propagate to the user until the next invocation of the self-testing routine which detects the fault and initiates the recovery.

2.2.1 Fault-aware remapping support

A message sequence diagram involving all the actors of the FRWEP technique is shown in Figure 2.12. A self-test routine is periodically invoked on each tile alongside the nominal processes of the application. Similar to the FRWOEP case, STM checks whether the CRC signature of the results matches the stored signature or not. Alternatively, if the STM is not written with any self-test results within a self-test period, it infers that a crash error has occurred. In either case, TMH is activated and the processing element is isolated. Unlike FRWOEP, TMH is responsible only for the notification of the remapping manager about the fault detection. This is done by TMH by sending an interrupting message to RM.

When RM receives the fault detection message, the tasks on the fault-free PEs would be working normally either executing their processing functions, being blocked on a read or a write. Some tokens of the tasks could be waiting on the software FIFOs to be sent or in the NI buffer to be received or even travelling on the NoC. Similarly pending requests, which are sent from consuming tasks to the producer tasks, might have been already received or might be still travelling along the NoC. Given such a snapshot at the instant of fault detection, RM takes a number of steps to flush the current state from all the tiles and makes the tasks ready to continue executing the next data item.

First, it sends a *FLUSH_TASK* interrupting message to the tile hosting the source task. Upon receiving this interrupt, the interrupt handler of the source tile sets a flag (*isTaskToBeFlushed*) requesting the flushing of the source task. The process bodies and KPN communication primitives are modified as explained in section 2.2.2 to respond to such requests. In the special case of the source task, the source seeks the stream forward to the next data item and responds back to RM with the index of the next data item. RM continues sending the *FLUSH_TASK*(*next_data_item_index*) interrupting messages to all other tiles that run at least one task (except the faulty tile). This interrupt message sets the *isTaskToBeFlushed* flag for all the tasks on the tile. It also marks the iterator values to be updated (*isDataItemIndexToBeUpdated*) when the tasks are resumed after the recovery is finished. The purpose of flushing all tasks is to delete the state of the current frame by removing the tokens in all input and output FIFO channels and reseting pending requests. However serving the flush request is not as simple as that due to the tokens or requests currently travelling on the NoC. Therefore, a *channel flush mechanism* is employed in order to guarantee that all the state (tokens and requests) are received by the tiles and the NoC buffers are free of such state.

The channel flush mechanism is based on the idea that the NoC transmits the packets in order. Therefore, if a special *CHANNEL_FLUSH* token is sent by a source tile and is received by a destination tile, it is guaranteed that any data that has been sent by the source tile before the special token should be already received in the NI buffer of the destination tile. Given a PPN task graph, the channel flush mechanism should guarantee that at least one *CHANNEL_FLUSH* token is sent from the source tiles to the destination tiles for every channel of the PPN task graph (except those channels in/from/to the faulty PE and those that are inside a tile, that is, source and destination tile are the same).

Algorithm 3 shows the channel flush mechanism carried out by each PPN process as a part of serving the flush request for the task (*serve_flush_request*()). In lines 3–, firstly, each task sends a CHANNEL_FLUSH token to each destination tile of its output channels, given that the destination tile is not the faulty tile and that the destination tile is not the same tile that runs the task. In lines 6–13, each task receives a CHANNEL_FLUSH token from each source tile of its input channels, given that the source tile is not the

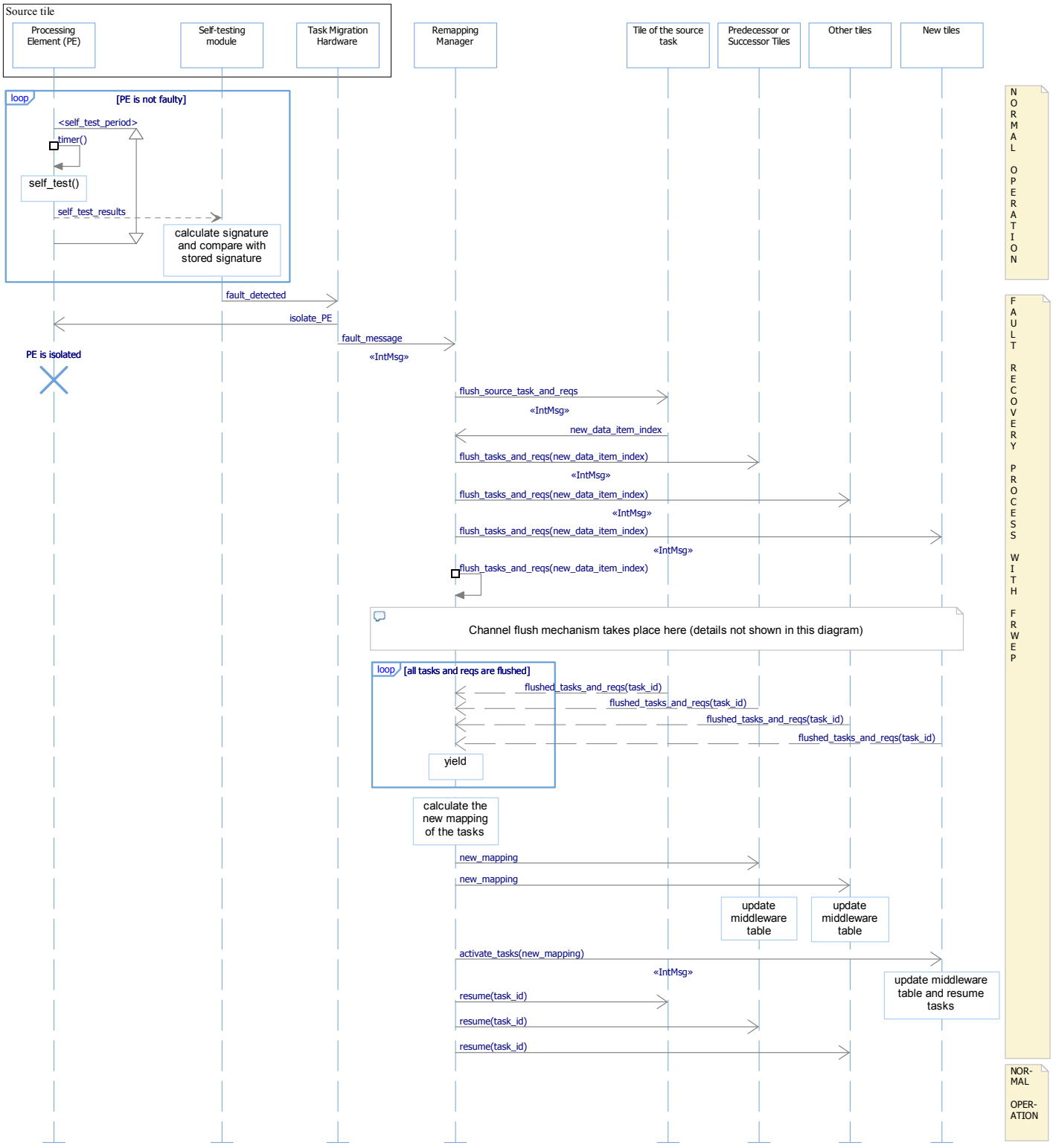


Figure 2.12: FRWEP sequence diagram

faulty tile and that the source tile is different than the tile of the task. In order to let other tasks on the same tile execute their part in the channel flush process, the task yields while polling on the CHANNEL_FLUSH tokens.

The case of requests is a bit different due to the fact that they are sent in the reverse direction of a channel (i.e., from the consuming task to the producing task). The channel flush mechanism should also make sure a *CHANNEL_FLUSH* token is sent in the reverse direction of every channel. This can be achieved by a slight modification to the text of Algorithm 3 by replacing *source* with *destination* and vice versa.

Algorithm 3 The channel flush mechanism inside *serve_flush_request*() of task *t*

```
1: CH_I(t) : input channels of task t
2: CH_O(t) : output channels of task t
3: for all ch ∈ CH_O(t) and destination(ch) is not faulty and source(ch) ≠ destination(ch) do
4:     shmpi_send(token, 1, destination(ch), CHANNEL_FLUSH_TAG);
5: end for
6: while all CHANNEL_FLUSH messages are not received do
7:     for all ch ∈ CH_I(t) and source(ch) is not faulty and source(ch) ≠ destination(ch) and !isFlushed(ch) do
8:         if shmpi_nb_recv(token, 1, source(ch), CHANNEL_FLUSH_TAG) then
9:             isFlushed(ch) = true;
10:        end if
11:    end for
12:    yield();
13: end while
14: return
```

Based on a simple cyclic graph of four tasks, the channel flush mechanism has been formally verified using the NuSMV tool [5] to satisfy the specification that for all possible executions it eventually holds that all tasks are flushed.

By the end of the channel flush mechanism, all the tasks proceed with transfering the tokens in the NI buffer to the FIFO channels and clearing the tokens in the FIFO channels as well as the requests in the NI buffer. As a result, the resources used by the current data item are recovered and no residual state remains which would likely cause negative impacts in the future. Finally each flushed task responds to RM that the flushing of the task is completed and waits for a *resume* message from RM before continuing execution.

Then RM calculates the new mapping of the tasks on the faulty PE and updates the middleware table of all tiles except the faulty tiles and the new tiles. Then it updates the middleware tables on the new tiles and activates the tasks on their new processors. Finally it sends all other tasks *resume* messages to let them continue their execution.

Decentralized Remapping Manager

The centralized techniques represent a single point of failure and thus they should be avoided in fault tolerance mechanisms. In both FRWOEP and FRWEP mechanisms, RM is the main actor coordinating the recovery process. Therefore it is important that RM is not centralized. As a solution to this problem, RM is run as a dormant thread on each processing element. Any tile can assume the role of RM when it receives an interrupting fault detection message. A simple algorithm is used to assign an RM instance to each tile. TMH of a faulty tile sends the fault detection message to the RM instance assigned to its tile. The RM assignment algorithm is as follows: A tile chooses as its RM the tile that has the smallest index that is greater than its own index and that is fault-free. If there is no such tile, it chooses the tile that has the greatest index that is smaller than its own index and that is fault-free. This makes sure that each tile is assigned a fault-free RM tile. Given the single fault assumption, when a fault occurs, every fault-free tile is informed about the

faulty tile and update their knowledge about the fault status of other tiles. If the faulty tile is their currently assigned RM tile, they re-run the RM assignment algorithm.

2.2.2 Modifications to the PPN processes

As a part of the FRWEP support, the process bodies and the KPN communication primitives are required to be modified. Algorithm 4 shows how the basic process body shown in Algorithm 1 is modified to support the FRWEP mechanism.

In the case that a process is activated on the new tile, the checking of migration flag (line 1) will allow the process to start execution from the correct data item index.

In the case that a process is not on the faulty tile, it receives a *FLUSH_TASK* request. This request is eventually checked in the modified *read*() primitive (in Algorithm 5 line 5) or in the modified *write*() primitive (in Algorithm 6 line 2). The *serve_flush_request*() function carries out the channel flush mechanism by clearing the FIFO channels and requests as explained in section 2.2.1. It also sets *isDataItemIndexToBeUpdated* and yields in a loop until *resume* message is received. If a flush request is served inside *serve_flush_request*(), it returns true. Therefore the blocking read (in Alg. 4 line 10) or blocking write (in Alg. 4 line 15) calls return immediately after the flushing of the task, thanks to the modifications in Alg. 5 line 6 and Alg. 6 line 3. The check on *isDataItemIndexToBeUpdated* in Alg. 4 line 11 and 16 allows the process to break execution of the current data item and reach line 5 where it starts the execution of a new data item with the updated data item index.

Algorithm 4 The PPN process template in the FRWEP mechanism

```
 1: if migration then
 2:     i0 = newDataItemIndex;
 3: end if
 4: for (i=i0 ; i<M; i++) do
 5:     if isDataItemIndexToBeUpdated then
 6:         i0 = newDataItemIndex;
 7:         isDataItemIndexToBeUpdated = false;
 8:     end if
 9:     for (j=0 ; j<N; j++) do
10:         read(in, CH1);
11:         if isDataItemIndexToBeUpdated then
12:             break;
13:         end if
14:         out = f(in);
15:         write(out, CH2);
16:         if isDataItemIndexToBeUpdated then
17:             break;
18:         end if
19:     end for
20: end for
21: return
```

In order to support the FRWEP mechanism, the self-test and task migration hardware modules have been integrated to the tile architecture as shown in Figure 2.13. A detailed description of TMH-FRWEP is given in section 2.2.4.

Algorithm 5 The read(token, channelID) primitive in the FRWEP mechanism

```
1:  if fifo[channelID] is empty then
2:      sendRequest(channelID);
3:  end if
4:  while fifo[channelID] is empty do
5:      if serveFlushRequest() then
6:          return
7:      end if
8:      processNIMsgs();
9:  end while
10: fifoGet(token, channelID);
11: return
```

Algorithm 6 The write(token, channelID) primitive in the FRWEP mechanism

```
1:  while fifo[channelID] is full do
2:      if serveFlushRequest() then
3:          return
4:      end if
5:      processNIMsgs();
6:  end while
7:  fifoPut(token, fifo[channelID] );
8:  processNIMsgs();
9:  return
```

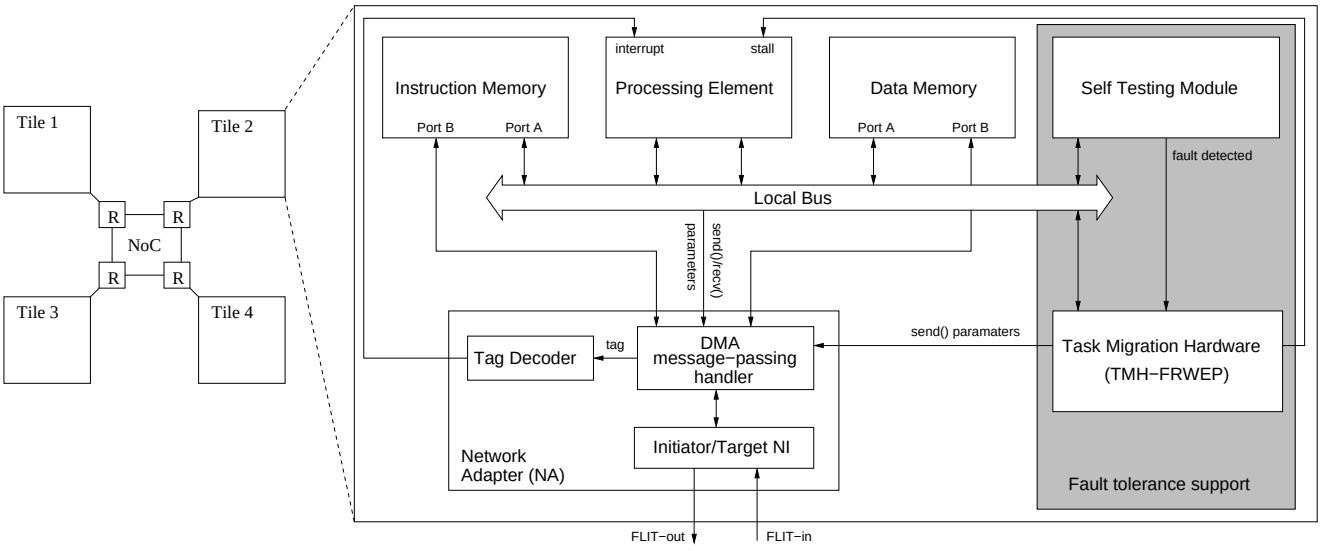


Figure 2.13: Block diagram of the fault tolerant tile

2.2.3 Self Testing Module

The self-testing module is almost the same as the one of FRWOEP (described in section 2.1.3). There is a slight difference only in the timer component. In the FRWEP case, the timer works in a periodical fashion. The period, which is equal to the self-testing period, is written at boot time once into the *slv_timer_limit* register. It is started only once via the *slv_timer_start* register. Then the timer counts down. At the time it reaches zero, a check is done by monitoring the *start_stop* input of the timer whether there has been a self-testing routine execution during the count down. If so, the timer is reset again to the *slv_timer_limit* value and starts counting down again. Otherwise, a crash error is assumed and *fault_detected* signal is raised.

2.2.4 Task Migration Hardware (TMH-FRWEP)

The block diagram

The structure and the function of TMH-FRWEP is much simpler than TMH-FRWOEP because TMH-FRWEP is only responsible for isolating the processor when the fault is detected then reporting the fault to the remapping manager through an interrupting message. Figure 2.14 shows the internal structure of TMH-FRWEP which is similar to that of TMH-FRWOEP but the size of the register file is smaller, there is no *flush_from_tile_< x >* signals since there are no flush messages to receive and there is shmpi_send data register as the fault detection message does not require to send data from the TMH itself.

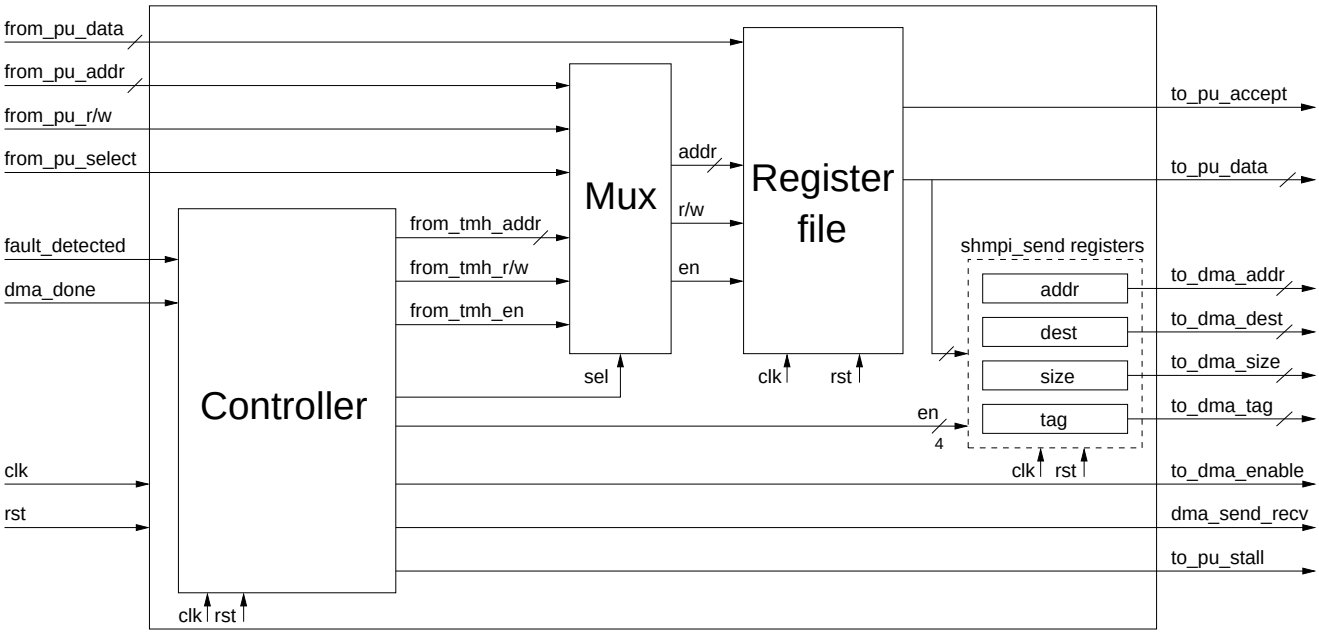


Figure 2.14: The block diagram of the TMH-FRWEP

The TMH register file

The size of the register file is more compact as it only has four registers (address, destination, size and tag) which corresponds to the parameters of the shmpi_send() message. Table 2.4 shows the content of the register file.

Table 2.4: The structure of the TMH-FRWEP register file

address	Register name
0x84000000	these two addresses are reserved for development purposes
0x84000004	
0x84000008	TMH_FAULT_MSG_ADDR
0x8400000C	TMH_FAULT_MSG_DEST
0x84000010	TMH_FAULT_MSG_SIZE
0x84000014	TMH_FAULT_MSG_TAG

TMH shmpi˙send registers

The normal shmpi_send() function with four parameters (address, destination, size, and tag register) is needed to send the *Fault MSG*.

The TMH controller

After asynchronous reset signal, the TMH starts from the *idle state* where the register file is kept updated by writing the latest information to its registers. When the *fault_detected* signal is set high, the TMH sets the *to_pu_stall* signal to stop the faulty processor from working then during four states the parameters of the send message (address, destination, size and tag registers) are set to send a *Fault MSG* to the RM. The *to_dma_enable* and *dma_send_recv* signals are sampled high to initiate the send operation then waiting in *Sending the Fault MSG is successfully done* state until the *dma_done* signal is sampled high to proceed to the *DONE state* as shown in Figure 2.15.

2.2.5 Experimental results

For the MJPEG application mapped on the 2x2 NoC platform with Microblaze processors, we have carried out six different fault scenarios as shown in Figure 2.16, 2.17, 2.18, 2.19, 2.20, 2.21. A fault scenario is identified by the initial mapping and the order of fault injections on the different tiles. Because of the limitations of the platform, we have mapped the source task onto Tile 3 and the sink task (VLE) onto Tile 4. The mapping of DCT and Q is varied by mapping them individually or together onto Tile 1 and/or Tile 2. Moreover, faults are injected to Tile 1 and Tile 2 in different orders, though making sure that the second fault is injected after the recovery from the previous fault is finished (in adherence to the single fault assumption).

Fault injections are done by activating TMH directly by the processor. Figure 2.22 shows the finishing times of each recovery action averaged over 11 different fault injection times for the same fault scenario. Time 0 represents the instant that TMH is activated (i.e., fault detection instant). The average of the total recovery times range from 39k to 98k cycles for different fault scenarios. It can be seen that the flushing of tasks takes the majority of the time and its duration may vary greatly in each experiment with a different fault injection time or a different fault scenario. This is mainly due to the fact that the task may be executing its processing function before it gets to serve the flush request. In the worst case the flush request may arrive just after a read() call which leads to a delay as much as the duration of the processing function until the flush request is served in the next write() call. In the MJPEG case, one iteration of the heaviest task, which is DCT, takes around 132k cycles.

If the fault injection is done in the hardware level, then the time from the fault occurence till its detection would be added to the fault recovery time. Since faults are detected via self-testing, the worst case fault detection time would be equal to the period of the self-test (in case the fault occurs immediately after a successful self-test). When FRWEP is used, the shortest self-test period for the MJPEG case would be equal

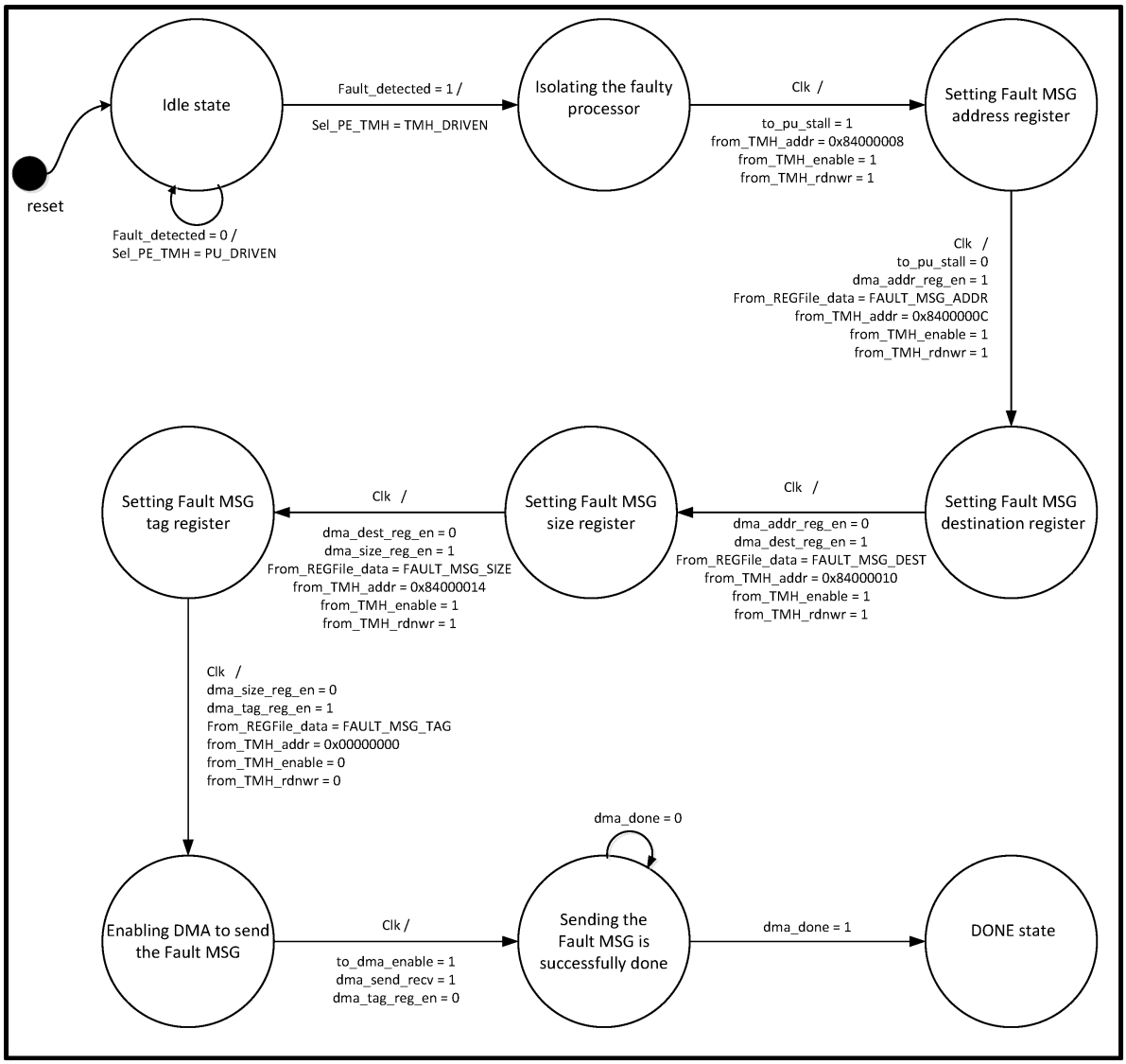
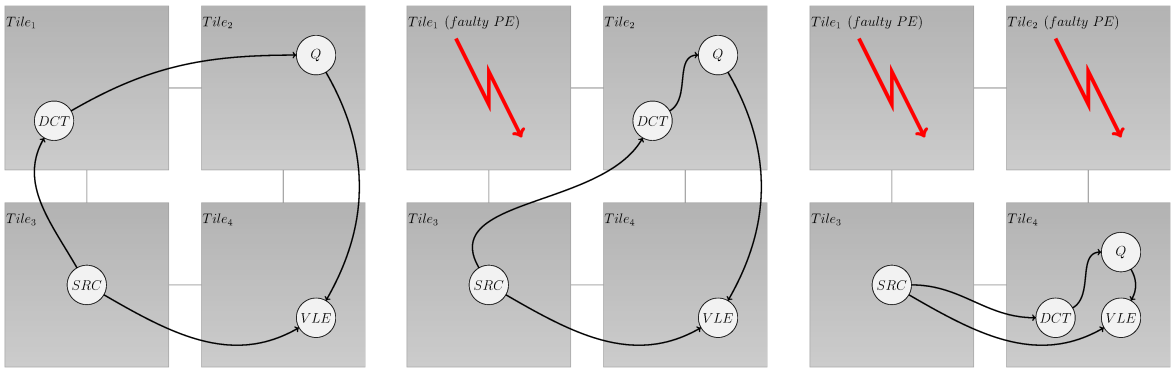
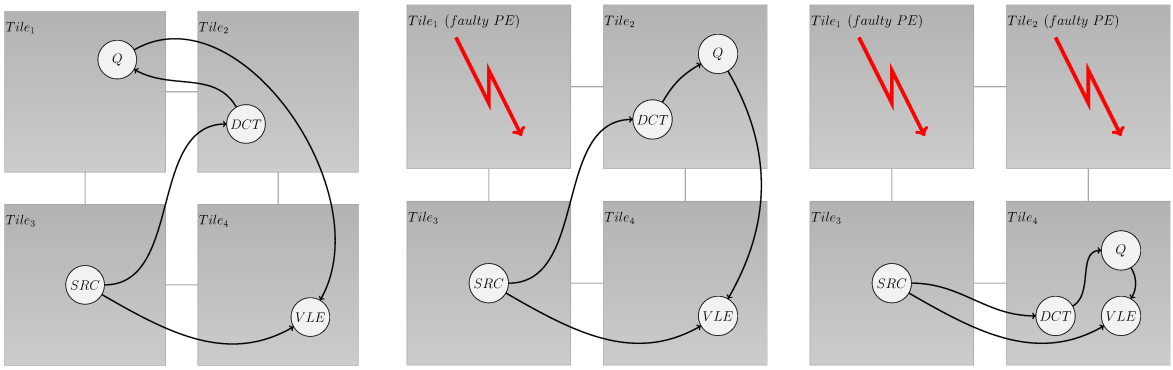


Figure 2.15: The finite state machine of the TMH-FRWEP controller



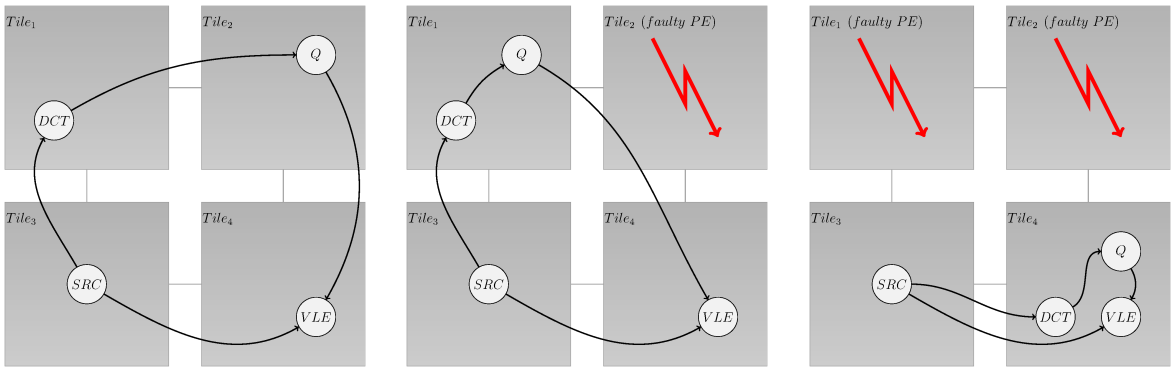
(a) Initial mapping (b) First fault (c) Second fault

Figure 2.16: Scenario 1



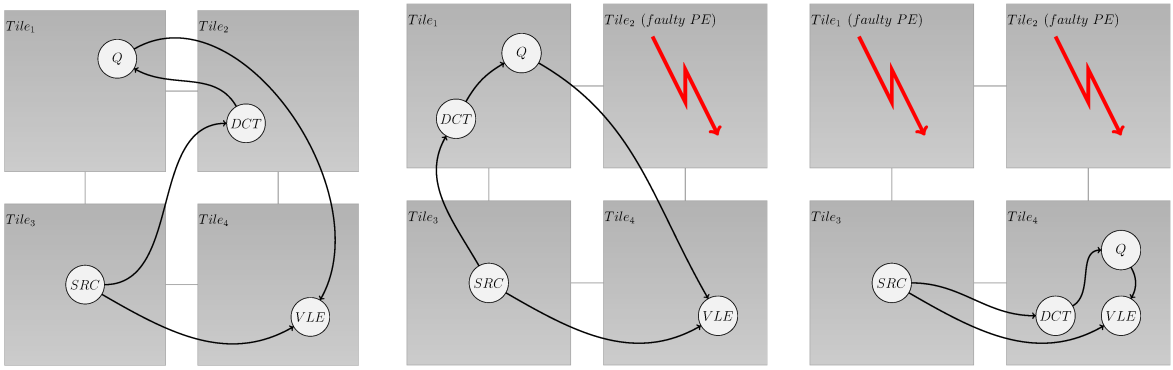
(a) Initial mapping (b) First fault (c) Second fault

Figure 2.17: Scenario 2



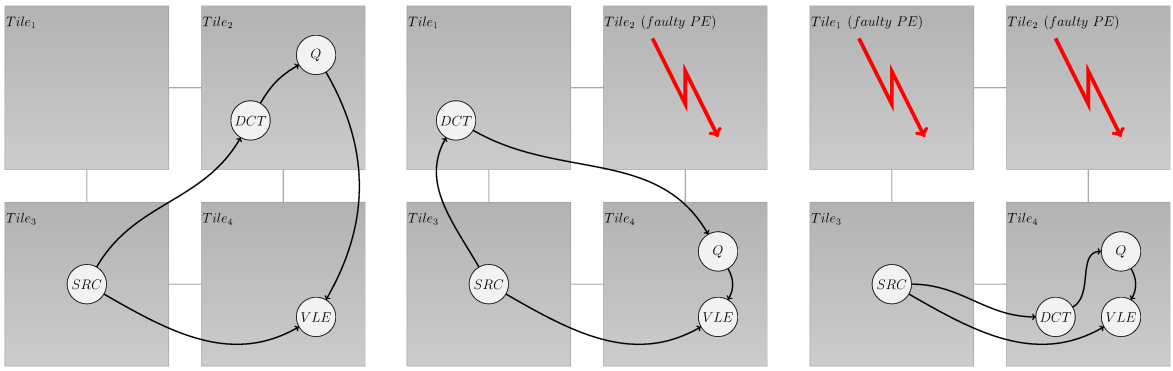
(a) Initial mapping (b) First fault (c) Second fault

Figure 2.18: Scenario 3



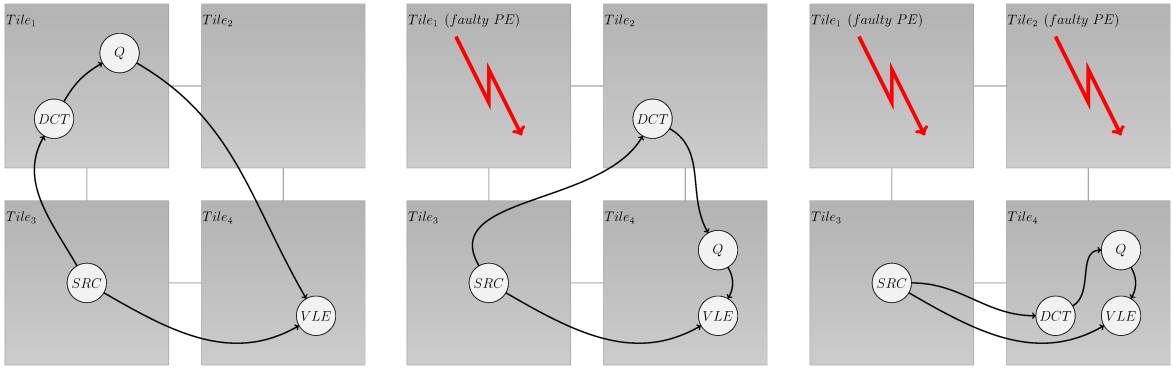
(a) Initial mapping (b) First fault (c) Second fault

Figure 2.19: Scenario 4



(a) Initial mapping (b) First fault (c) Second fault

Figure 2.20: Scenario 5



(a) Initial mapping (b) First fault (c) Second fault

Figure 2.21: Scenario 6

to the processing time of one frame. In case the MJPEG tasks are mapped one-to-one on the 2x2 platform, the execution time of one frame is around $10x10^6$ clock cycles. The observed fault recovery times in Figure 2.22 would constitute only a small fraction of the total recovery time. That is to say, the fault recovery time is fundamentally determined by the self-test period.

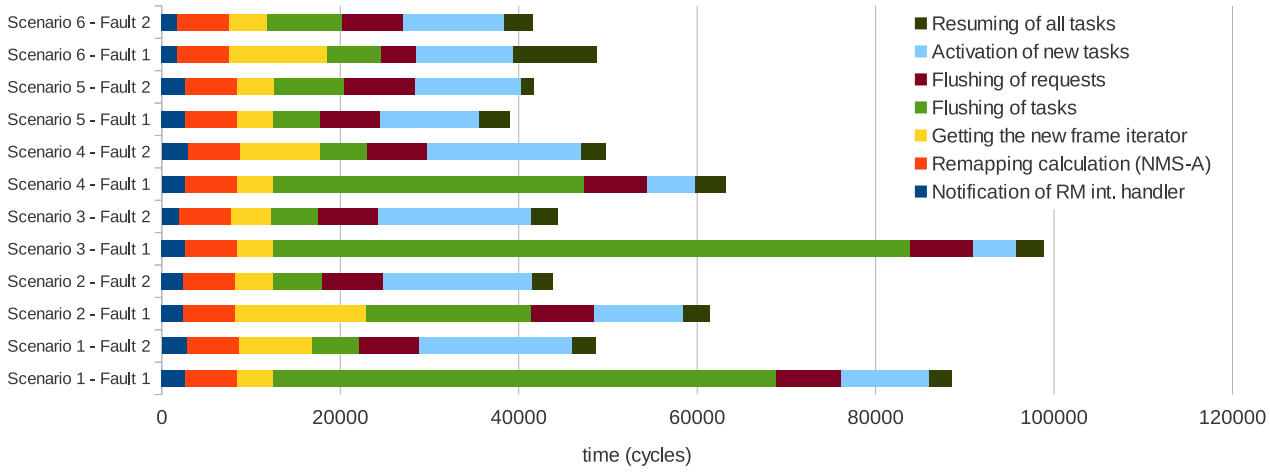


Figure 2.22: The time of fault recovery actions with the FRWEP mechanism

Unlike the FRWOEP mechanism, the self-test can be executed at any desired frequency in the FRWEP mechanism. Obviously this frequency would affect the overhead of the technique during normal operation. Assuming a self-test execution time of 40k cycles, the overhead with respect to varying self-testing period (quantified by the number of frames processed within the period) is shown in Figure 2.23. It can be seen that the overhead due to the self-test routine diminishes completely when the period is greater than 7 frames. The converged value of 1.07% overhead is due to the modifications done in the PPN processes to support the FRWEP mechanism.

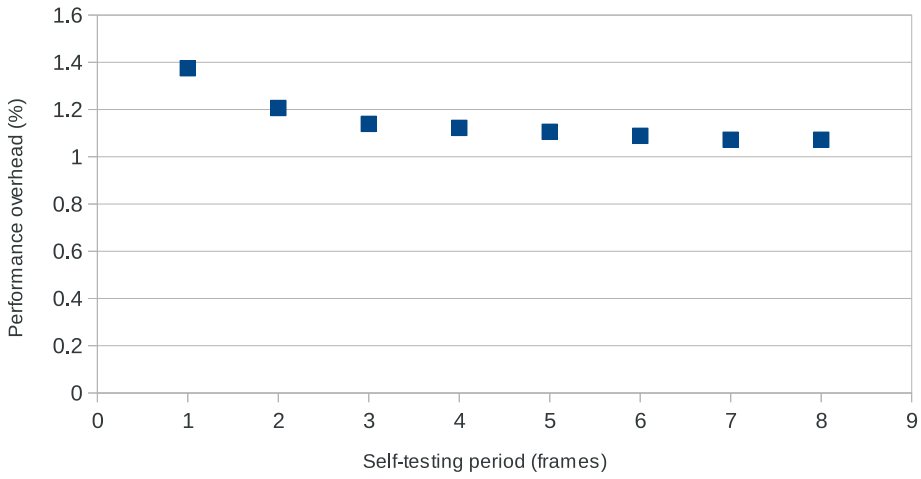


Figure 2.23: Performance overhead with respect to the period of the self-test routine with the FRWEP mechanism

Similarly, one can assess the overhead with respect to the duration of the self-test routine. Assuming that a self-test is executed for every frame, the overhead by varying the self-test duration is shown in Figure 2.24. An order of change in the execution time of the self-test routine increases the overhead from 1.1% to 1.9%, which are much better values compared to the ones of FRWOEP (7.7% to 71.7% – see Figure 2.8).

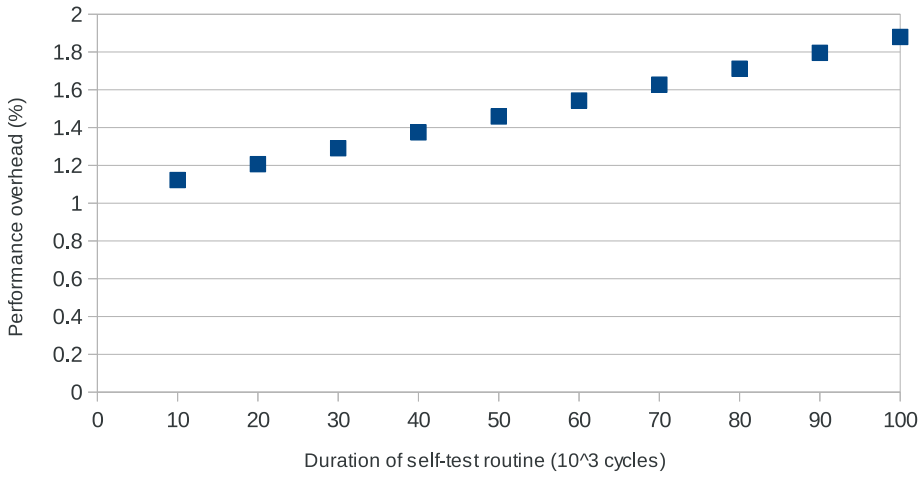


Figure 2.24: Performance overhead with respect to the duration of the self-test routine with the FRWEP mechanism

The area and frequency numbers of the TMH-FRWEP module are reported in Table 2.3, Figure 2.9 and 2.11. Due to its simplicity THM-FRWEP has a much smaller area and achieves a higher clock frequency than TMH-FRWOEP. As shown in Figure 2.10, the area overhead due to STM and TMH in FRWEP is only 8.0% as opposed to 20.7% in FRWOEP (with TMH supporting 4 channels). Moreover, TMH-FRWEP is clearly scalable as these results do not depend on the number of channels in the application.

2.3 Online remapping heuristics

In this section we describe a set of experiments that we performed in addition to the ones that have already been reported in Deliverable D5.3 regarding the evaluation of the online remapping heuristics. The addition is based on the H.264 decoder case study, a streaming application in the multimedia domain, provided by University of Leiden. It is described in section 2.3.1. We map the H.264 decoder application onto a 2x2 mesh of Microblaze processors implemented on an Virtex-6 FPGA board. The execution time of the proposed remapping heuristics as well as the accuracy and optimality of the chosen remappings are evaluated in section 2.3.2 and 2.3.3, respectively.

2.3.1 H.264 decoder

The simplified PPN specification of this case study is shown in Fig. 2.25. In the final implementation, the nodes *get_data*, *parser*, and *cavlc* have been merged into a single process, H_0. In this case study, the size of the exchanged tokens ranges between 1 and 5000 bytes. The execution time of each process of the H.264 decoder application are shown in Table 2.5.

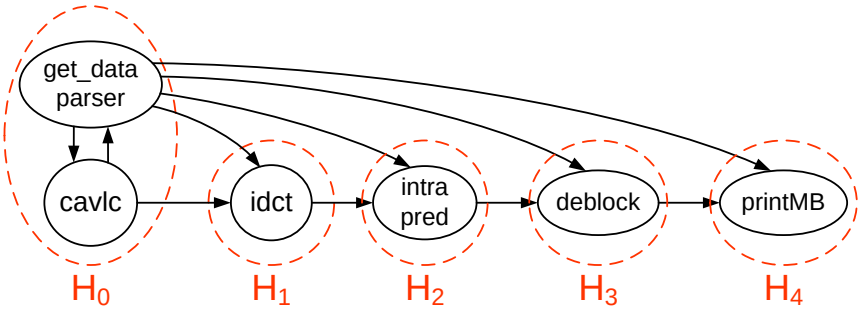


Figure 2.25: Simplified PPN specification of the H.264 decoder.

Table 2.5: Execution times of H.264 processes

Process	Avg execution time (c.c.)
H_0	95643
H_1	55775
H_2	33645
H_3	9724
H_4	4075

2.3.2 Execution time of the remapping heuristics

To evaluate the calculation time of the remapping decision, the remapping scenarios given in Fig. 2.26 are used for the H.264 application. The NMS-A/B/C heuristics from [6], which aim at minimizing the throughput degradation, are implemented on the platform. Their calculation time are displayed in Table 2.6. The results reveal that their execution time constitutes a relatively small portion of the fault recovery time reported in section 2.1.5 and 2.2.5.

Table 2.6: Calculation times of remapping heuristics

Heuristic	Avg execution time (c.c.)
	H.264
NMS-A	8172
NMS-B	19603
NMS-C	6664

2.3.3 Evaluation of the remapping strategy

In this section, the quality of the heuristic is evaluated using the H.264 case study by comparing the remapping obtained by the NMS-A/B/C heuristics with actual measurements.

H.264 remappings

Given a 2x2 NoC-based platform with processing elements ($tile_1 = n_1, tile_2 = n_2, tile_3 = n_3, tile_4 = n_4$) and an initial mapping of H.264 tasks $I : H_0 \rightarrow n_3$, $H_1 \rightarrow n_1$, $H_2 \rightarrow n_2$, $H_3 \rightarrow n_4$, $H_4 \rightarrow n_4$ as shown in Fig. 2.26(a), we consider two single fault scenarios for n_1 and n_2. As shown in Fig. 2.26(b), for the case of n_1 faulty, all possible remappings are R_1 ($H_1 \rightarrow n_2$), R_2 ($H_1 \rightarrow n_3$) and R_3 ($H_1 \rightarrow n_4$). Similarly, Fig. 2.26(c) shows the case of n_2 faulty for which all possible remappings are R_1 ($H_2 \rightarrow n_1$), R_2 ($H_2 \rightarrow n_3$)

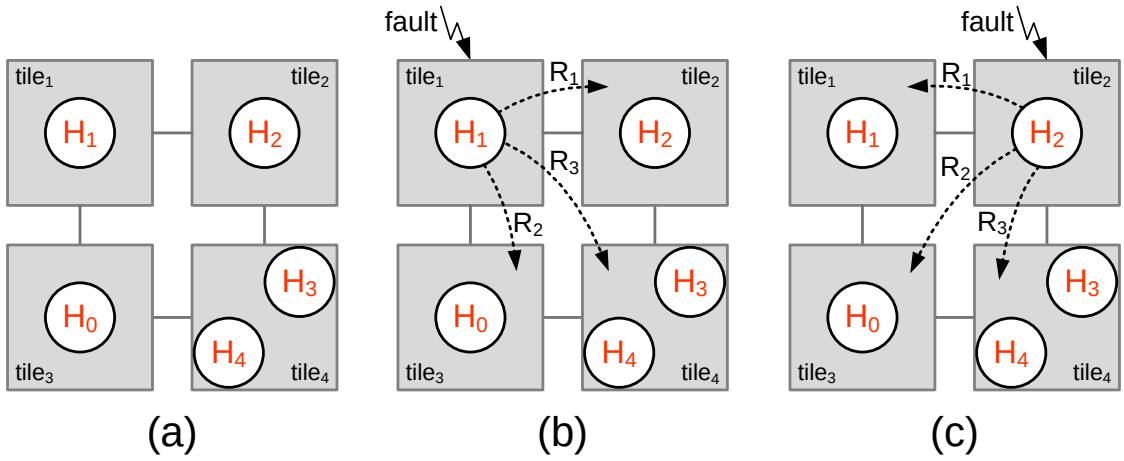
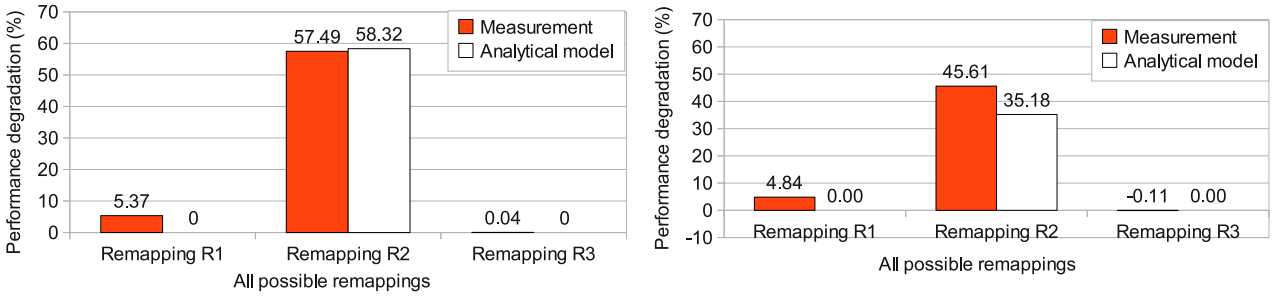


Figure 2.26: Initial mapping and the two single fault scenarios showing all possible remappings.



(a) when n_1 is faulty as shown in Fig. 2.26(b)

(b) when n_2 is faulty as shown in Fig. 2.26(c)

Figure 2.27: Comparison of measured and calculated performance degradation of all possible remappings.

and R_3 ($H_2 \rightarrow n_4$). The total execution times of the H.264 application for all possible remappings, T_{R_i}, are measured on the platform using the RD-int flow control and also calculated by the analytical model.

The performance degradation with respect to the execution time of the initial mapping, T_I, is calculated according to Equation 2.1.

$$Performance\ degradation(R_i) = \frac{T_{R_i} - T_I}{T_I} \tag{2.1}$$

Measured and calculated values are used in Equation 2.1 for calculating the *measured* and *analytical model* degradation results shown in Fig. 2.27(a) and 2.27(b) for faulty n_1 and faulty n_2 cases, respectively. Note that in some cases, for instance *R3* in Fig. 2.27(b), the remapping can lead to a slight performance speedup. In *R3*, this is because the reduction of the communication time over the NoC overcompensates the increased computational workload on n_3.

The optimal remapping is the one which yields to the smallest performance degradation. Despite the differences up to 10% in the estimated and measured degradation values as shown in Figure 2.27(a) and 2.27(b), the heuristics take the optimal decisions. In case of a fault occurring on n_1, all of the NMS-A/B/C heuristics yield to remapping R_3, which is the optimal one as shown in Fig. 2.27(a). In the other considered case, faulty n_2, all the heuristics suggest remapping R_3. Also in this case, the suggested remapping represents the optimal one, as can be deduced by Fig. 2.27(b). The inaccuracy of the analytical model is due to the blockings in the communication and the unaccounted context switching times when several tasks are running on a processor.

3. Design space exploration of fault tolerant NoC components

This section presents the methodology developed for the early exploration of the design space of fault tolerant Networks-on-Chip. By employing a library of alternative implementations for the NoC components, and ad-hoc high-level NoC simulation, fault simulation, and synthesis tools, the methodology is able to select the architecture that satisfies the desired fault tolerance requirements, and that minimizes implementation costs. Section 3.1 presents the considered NoC fault model, as also presented in Deliverable D5.2. Section 3.2 briefly describes the components of the exploration flow. Section 3.3 describes the use of the exploration framework for the selection of the optimal network-on-chip architecture.

3.1 Fault model of baseline NoC components

This section presents the fault models considered for the NoC components. In general, it is possible to distinguish between *soft errors*, caused for instance by the interaction of the system with radiations such as neutrons from cosmic rays and alpha particles from packaging material, and *hard errors*, caused for instance by wear-out mechanisms. A single fault in the elements of the NoC can cause packets to be corrupted or misrouted, leading to possible unrecoverable situations in the NoC, such as deadlock or livelock conditions, or to a system crash [7, 8].

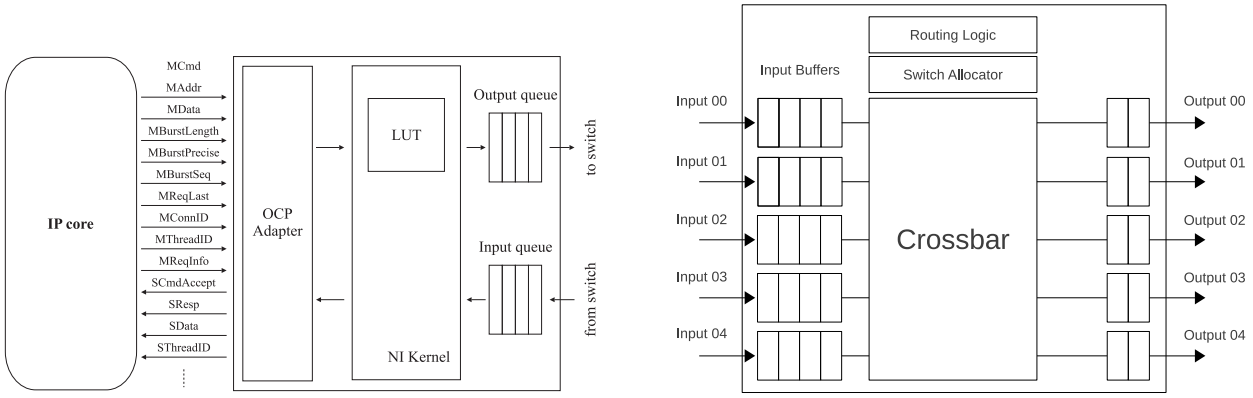
In this work, we target in particular permanent faults. We focus on functional-level error models, obtained by abstracting technology-related low-level fault models, and by collapsing in “fault classes" faults creating the same effect on the system. We comply to the single fault assumption. Consecutive faults are considered mutually independent. Moreover, we assume that the time between two consecutive faults hitting the same component is sufficiently long to apply at run-time the proposed online reconfiguration fault tolerant techniques.

As already introduced in Deliverable D5.1, from the point of view of the error model to be adopted the NoC can be considered as a combination of several components, i.e., links, routers, and network interfaces (NIs). Every component is characterized by a different error model which depends on the specific modules composing it.

3.1.1 Error model characterization

In order to derive the characterize a high level error model for the components of the NoC, we performed a fault injection campaign on a baseline implementation of the NoC, by considering in the evaluation both network interfaces and routers.

As model for the NI, we consider the baseline NI architecture considered in the MADNESS platform, and shown in figure 3.1(a). For the router, we employ the input buffered architecture supporting source-based routing adopted in the platform. A high level view of the architecture of a 5-port router is shown in figure 3.1(b). In the experiments, we considered a tile-based NoC in a 5x5 mesh topology. We considered a 35-bit data-path (32 bits for data and 3 bits for control information) and a depth of 4 for the input buffers of routers,



(a) NI (b) Router

Figure 3.1: Baseline architecture of NIs and routers

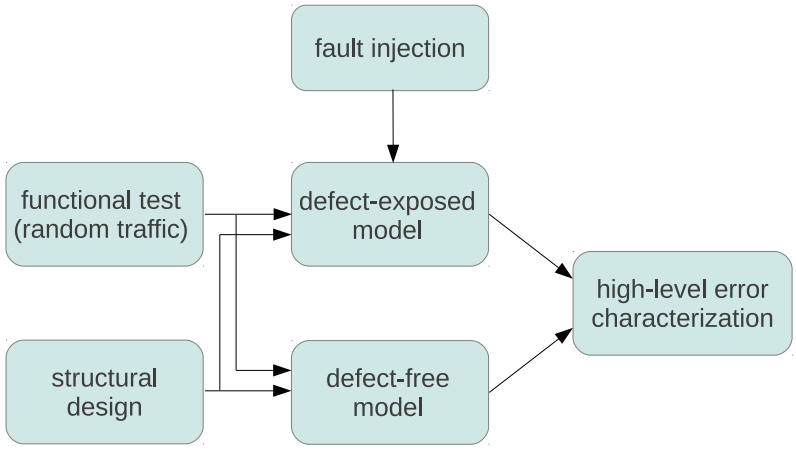


Figure 3.2: Simulation methodology employed for high-level error characterization of NoC components

2 for routers' output buffers, and 8 for input and output buffers of NIs. Both network interfaces and switches were implemented in VHDL, and synthesized with Synopsys Design Compiler, by employing the Nangate 45nm CSS typical open cell technology library [9]. In our synthesis, we targeted a frequency of 500 MHz. In the case of a 5x5 mesh topology, the area obtained for the NI and the router is respectively $0.0096mm^2$ and $0.0097mm^2$.

In order to evaluate the impact of silicon defects in NIs and routers, we employed a simulation infrastructure similar to the one presented in [10]. As shown in figure 3.2, a simulator simulates in parallel two copies of the gate-level description of the design. While one of the two copies is kept defect-free, the second one is subject to fault injection. A full coverage functional test is provided as input of the design, and the output of the simulation of the defect-exposed model is compared against the output of the defect-free model for identifying the high-level error caused by the injection of each fault affecting the system. With this methodology we are able to identify the components of NIs and routers that are most sensitive to (permanent) faults, and the high level error generated by them. In the case of a 5x5 mesh topology, the number of faults injected by the evaluation system into the NIs is 763'800, while the faults injected in the routers is equal to 825'300.

Table 3.1 reports the percentage of faults measured for each component of the NI and the router, calculated with respect to the total number of fault injected into the node. In both cases, *Other* components include glue logic and some additional registers needed for the controlling system's operations. For the 5x5 mesh topology, the number of faults concerning NIs and routers are approximately similar: faults affecting

Table 3.1: Fault injection results for the components of routers and NI in the case of a 5x5 mesh

Component	Fault location	Fault percentage (%)
NIs	LUT	23.64
	Buffers	21.78
	FSMs	1.39
	Other	1.26
Routers	Buffers	32.98
	Switch	10.59
	Allocator	8.23
	Other	0.13

Table 3.2: Percentage of errors (with respect to total NoC errors) measured in the NI during the fault injection campaign

NI Error Type (%)					
Fault location	**Corr. Data**	**Corr. Prot. Conv.**	**Routing Path**	**Control Flow**	**Total**
LUT	0	0.04	23.60	0	23.64
Buffers	17.88	0	0	3.90	21.78
FSMs	1.04	0.35	0	0	1.39
Other	0.53	0.01	0.62	0.10	1.26
Total	19.45	0.40	24.22	4.00	48.07

routers represent around the 52% of the total faults in the NoC, while the faults affecting NIs are the 48%.

By analysing the output of the defect-exposed model, we identified the following types of functional errors for the NI:

1. **Corrupt Data Error:** data are corrupted during the operations of the NI and wrong data are sent through the communication channel. This type of error can happen due to hard and soft faults in the protocol adapters and in the FIFOs;

2. **Corrupt Protocol Conversion Error:** on the side of the node initiating the transaction, faults in the NI lead to control signals received from the core being corrupted, causing the NI kernel to generate wrong routing and control information for the packet header. On the target node side, a fault affecting the protocol conversion will cause a wrong implementation of the core communication protocol, invalidating or disrupting the operation performed. This type of error is due to faults in the NI's protocol adapters;

3. **Routing Path Error:** routing paths inserted in packets' headers are calculated looking up the addresses of requested operations. Faults in the lookup table cause erroneous routing and control information to be inserted in the packet headers, leading to possible communication errors, such as misdirection, deadlock, or livelock. Faults in the LUT and FIFOs can cause this type of error;

4. **Control Flow Error:** faults in registers storing control information in FIFOs and protocol adapters cause errors in the control flow of the FIFOs, by communicating corrupted information about the flits in the buffers. For instance, multiple copies of an outgoing or incoming packet could be sent to the input or output port throughout the time.

Table 3.2 shows errors generated in the NI by the injected faults, according to the high-level error model just described. The table shows the measured number of errors and the related percentage with respect

Table 3.3: Percentage of errors (with respect to total errors) measured in the router during the fault injection campaign

Router Error Type (%)					
Fault location	**Corr. Data**	**Routing Path**	**Switching**	**Control Flow**	**Total**
Buffers	27.4	0	0	5.58	32.98
Switch	8.86	0	0.90	0.83	10.59
Allocator	0.43	0.49	6.04	1.27	8.23
Other	0	0	0	0.13	0.13
Total	36.69	0.49	6.94	7.81	51.93

to the total errors generated in the node. However, it has to be noted that the numbers in the table refer to the potential high level effect of the fault on the system. In the case of a running system, the real error distribution would depend on the working condition (e.g., the packet injection rate and the traffic patterns).

As it is possible to note from the table, a significant percentage of the faults (23.64%) will affect the LUT, leading to the corruption of the routing information stored into it, and potentially to a significant number of run-time routing errors due to packets whose header contain the corrupted information stored in the faulty registers of the LUT. The second most significant type of errors (17.82%) is due to faults affecting the buffers of the NI, and therefore the data (included headers and control information) temporarily stored in it. Faults in FSMs mainly affect logic related to the data transfer during the protocol translation.

Table 3.3 shows high-level errors generated into the routers by the injected faults. The table shows the measured number of errors and the related percentage with respect to the total errors in the node. In the case of the router, we adapt to our implementation the system level fault models defined in [11] and [7]. The following types of error can be identified for the router:

1. **Corrupt Data Error:** transported data are corrupted during its passage through the router;
2. **Routing Path Error:** due to corrupted routing information, the data packet is routed to a direction different than the one implied by the routing information originally inserted in the header of the packet;
3. **Switching Error:** packets are sent to wrong ports or duplicated;
4. **Control Flow Error:** the control flow of the FIFOs is corrupted, due to faults in registers storing control information in the router;

Errors in routers are mainly caused by faults hitting input and output buffers (32.53%), and therefore causing corruption of data, routing, and control information in packets. The second source of errors is the switch, causing in particular corruption of data (8.86%). Faults in the allocator cause mainly errors in the switching of the router, potentially creating problems such as the missed delivery of flits and the wrong output port selection.

By applying the same methodology to the several architectural alternatives for the components of the NoC, we created a library of customizable high-level models that simulates the effects (the high level error) caused by the injection of faults on the component. The models take into account the relative influence and occurrence of each specific type of high level error, and simulate the behavior of the component when hit by the fault (basically providing an indication about the status of the component, i.e., correctly working or not). The use of the library for estimating the *survivability* of the overall system, defined as the probability of producing a correct behavior in the presence of a defined number of consecutive faults [8], is described in Section 3.2.

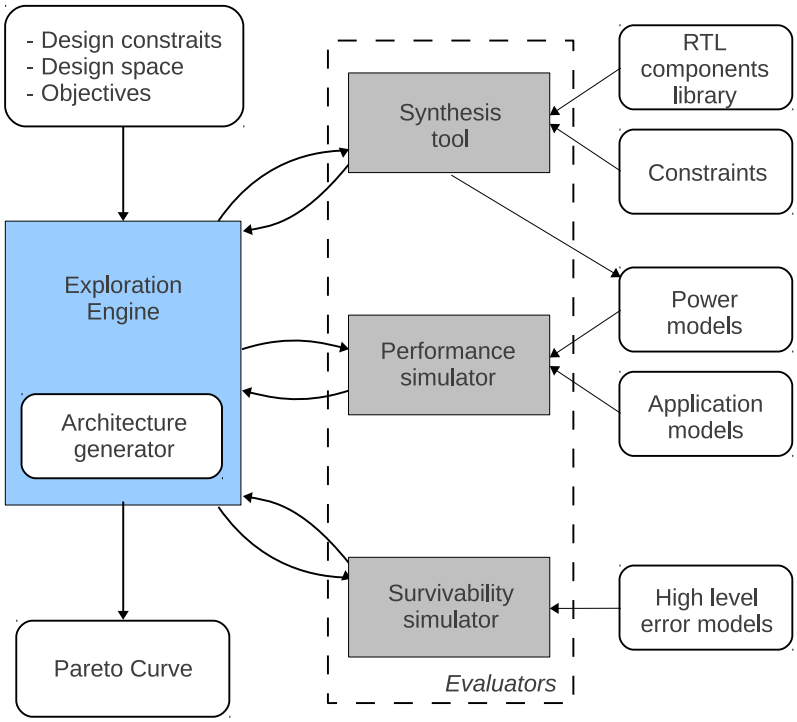


Figure 3.3: Overview of the exploration flow.

3.2 Exploration flow

Figure 3.3 shows an overview of the way the presented methodology can be integrated into the exploration flow. It takes as input a description of the NoC architectural parameters to be explored and of the range of possible values that they can take. Another input is represented by the definition of the constraints to be considered in the exploration (for instance, the limitation to a specific topology, etc.). An exploration of the possible architectural parameters is performed in order to obtain a list of optimal points which maximize the fault tolerance capabilities of the architecture as well as the performance for a specific application, while minimizing design cost.

As shown in the figure, the main components of the flow are:

- **Exploration Engine:** The exploration engine is in charge of exploring different architectural configurations for the NoC, while varying the parameters of the configurable components. Possible parameters that can be explored are for instance the length of internal NoC buffers, or the NoC topology, as well as alternative implementations of NoC components, that provide different degrees of fault tolerance and different behaviors in the presence of faults;

- **Synthesis Tool:** The synthesis tool evaluates costs (in terms of area) associated to the different implementation of the architectural solutions;

- **Performance Simulator:** It simulates the NoC behavior in case of a specific application or set of applications running on the top of the platform, by providing an estimation of network performance such as latency or power consumption;

- **Survivability Simulator:** It simulates the injection of faults into the NoC, while evaluating the errors generated by it. As evaluation metric, we employed the Silicon Protection Factor (SPF) [10], which gives a more representative notion about the amount of protection that a specific fault tolerant techniques

can offer to the system. The SPF is computed by dividing the *survivability* of the system with the area of the protection technique. The *survivability* is defined as the probability of producing a correct behavior in the presence of a defined number of consecutive faults [8]. The higher the SPF, the more resilient is the system to defects. Since the number of defects in a design is proportional in general to its area, the use of this metric for assessing the effectiveness of the protection techniques provides a fairer comparison than the survivability alone.

The survivability simulator is a high level C++ model of the NoC and of its components. The model simulates the behavior of each component at the occurrence of a new fault, by evaluating whether it is able to survive the fault or it produces an error. By performing the fault injection at high level, we significantly reduce the need for evaluating with standard hardware or software techniques the behavior of the system when struck by the faults [12].

Each component of the NoC is modeled by considering synthesis results obtained in the synthesis phase, as well as the error models derived in the error characterization phase for NIs and routers (Section 3.1.1). By employing these models, we inject a defined number of faults in the component. Each single fault is injected in random position over the area of the NoC. Injected faults are mutually independent, and enough time is left to the system to recover from the effects of the fault, for instance in the case of fault tolerant solutions employing reconfiguration methodologies. For each fault, we simulate the behavior of the NoC architecture and determine if with the injected sequence of faults can be still considered error-free. For each fault, we simulate the behavior of the component and determine if with the injected sequence of faults it can be still considered error-free. For each amount of injected faults, we repeated the experiment 10'000 times, for achieving statistical confidence. Therefore, we count the number of times over the total experiments in which after the defined number of injected faults the errors generated in the component was non correctable or non detectable.

3.3 Experimental results

In this section, we describe the design of a fault tolerant NoC by applying the methodology described in Section 3.3. We apply the methodology with goal of minimizing the area while maximizing the SPF and, therefore, the survivability.

To focus the experiments on the fault tolerant aspect of the exploration, we fixed some architectural parameters of the NIs and routers: we considered a 35-bit data-path (32 bits for data and 3 bits for control information) and a depth of 4 for the input buffers of routers, 2 for routers' output buffers, and 8 for input and output buffers of NIs. We consider an NoC composed of 9 nodes.

Table 3.4 presents the several parameters, architectures, and fault tolerant solutions considered in our exploration. Table 3.5 briefly describes the meaning of the terms used in table 3.4. In the case of the elements of the NI, we mainly refer to the work presented in Deliverable D5.3, in which several architectural alternatives were described for the building blocks of the NI, i.e., the lookup table, FIFOs, and finite state machines (FSMs).

In the case of the router, in a similar way to what was proposed in the Deliverable D5.3, we explore architectural alternatives for its building blocks, i.e., input and output buffers, crossbar, switch allocators, and glue logic. We also consider in the design space the possibility of having multiple alternative paths for connecting two nodes, exploring therefore the fault tolerant characteristics of different topologies together with the exploration of different architectural solutions for the NoC's components.

Figure 3.4 shows the Pareto points obtained in the case of the three topologies explored. Between the

Table 3.4: Design space considered for the NoC

Design parameter	**Explored values**
Topology parameters	
Topology	Mesh, Ring, Fat Binary Tree (FBT)
NI's component architecture	
LUT	BAS, TMR, SECDED, FT(n), FT(n/2), FT(n/4) (with *n* number of NoC nodes and LUT entries)
Input buffers	BAS, TMR, SECDED, FT(m-1), FT(m/2), FT(m/4) (with *m* number of FIFO's slots)
Output buffers	BAS, TMR, SECDED, FT(m-1), FT(m/2), FT(m/4) (with *m* number of FIFO's slots)
FSM	BAS, TMR, SECDED, FT
Other	BAS, TMR
Router's component architecture	
Input buffers	BAS, TMR, SECDED, FT(m-1), FT(m/2), FT(m/4)
Output buffers	BAS, TMR, SECDED, FT(m-1), FT(m/2), FT(m/4)
Crossbar	BAS, TMR, SECDED
Switch allocator	BAS, TMR
Other	BAS, TMR

Table 3.5: Design space parameters glossary

Parameter	**Description**
BAS	Baseline architeture, as shown in figure 3.1.
SECDED	Signals are encoded by using a Single Error Correcting and Double Error Detecting (SECDED) Hsiao code [13] able to correct up to one error and detect up to two errors [8].
TMR	Triple modular redundancy implementation. Three copies of the same component perform the same operation, and the single output result is obtained by a voting system [14].
FT(x)	Fault tolerant implementation for LUT and FIFOs, based on the the use of SECDED for encoding stored data, and of a variable level of architectural redundancy (given by *x*) for dealing with permanent faults in the nominal elements [8].

three topologies, the *ring* seems to provide for this configuration better fault tolerance. This is due to the fact that the area of the ring NoC is smaller than the other two topologies, being the number of routers in general smaller and with a lower number of ports. For this reason, the probability of being hit by a fault is proportionally smaller than in the other two topologies. Moreover, with respect for instance to the *FBT*, the fact of having for each source at least two possible paths for reaching the destination nodes make the *ring* topology more resilient. Due the high number of redundant paths, the mesh topology provides in general a good survivability of the system. However, in the calculation of the SPF the values of the survivability is balanced by the higher number of ports in the routers, and therefore, by a bigger area. With respect to the architectures of the components of NIs and routers, it should be noticed that the higher SPF (and therefore the higher resilience) is obtained when employing, in particular for the NIs, the fault tolerant techniques developed during the project.

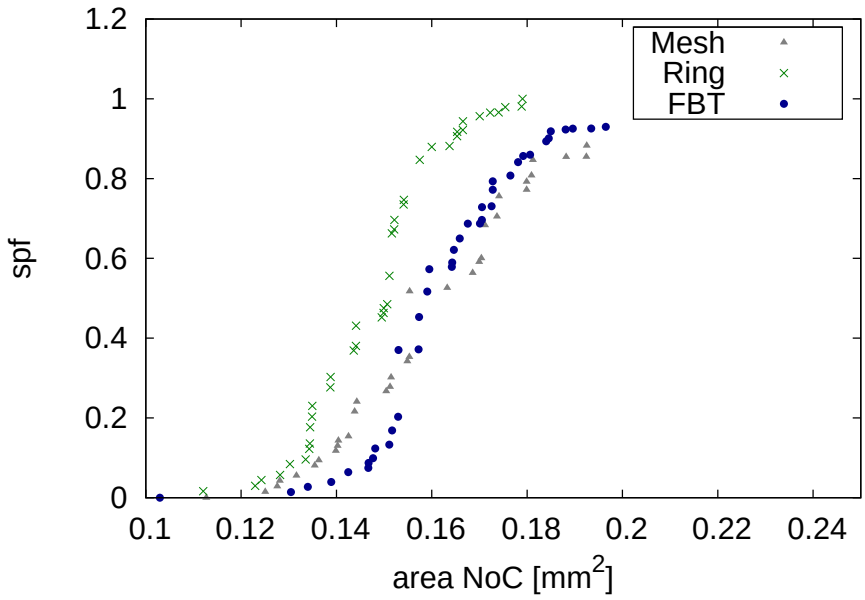


Figure 3.4: Pareto points of the design space performed for the NoC

4. Interactions with other work packages

In this third year of the project, there was an intensive collaboration between USI, UNICA, and UL for the development and integration of the task migration support on the MADNESS platform (WP6). Moreover, there was interaction between USI and UNICA for the definition of the exploration flow for fault tolerant Networks-on-Chip. Furthermore, WP5 activities have been complemented by the work performed by UvA in WP6 for dealing, by applying task-level redundancy and checkpointing techniquesat the KPN level, with permanent and transient failures in multimedia applications that allow a certain level of error propagation (i.e., missing frames).

Appendices

A. The details of the FSM of the TMH-FRWOEP controller

The FSM of sending a Fault detection MSG to the RM

Four states are used to set the parameters of the send message (address, destination, size and tag registers) to send a Fault MSG to the RM. *to_dma_enable* and *dma_send_recv* signals are sampled high to initiate the send operation in *Enabling DMA to send the Fault MSG* state then waiting in *Sending the Fault MSG is successfully done* state until the *dma_done* signal is sampled high to proceed to the next state as shown in Figure A.1.

The FSM of sending Flush MSGs to the predecessor and successor tile(s)

Succ_Pred_reg register is read in the *Reading Succ_Pred_reg* state to know which tile nodes are the successors and the predecessors to this tile node. A Flush MSG is sent to each tile node that its corresponding bit in *Succ_Pred_reg* register is equal to one. The FSM of Sending Flush MSGs to the predecessor and successor tile(s) is shown in Figure A.2.

The FSM of sending the state of the tasks (Iterators MSG) to the RM

The *TASKS_MAPPED_TO_THIS_TILE* and *TMH_ITERATORS_BASE_ADDR* registers are read at the *Reading Tasks_mapped register* and *Reading Iterators_Base_Addr* states respectively. The parameters to send a message are set in the *Setting Iterators MSG destination register*, *Setting Iterators MSG size register*, *Setting Iterators MSG address register* and *Setting Iterators MSG tag registe*r states respectively then the DMA is enabled to send the message by setting *to_dma_enable* and *dma_send_recv* signal high. *dma_done* is sampled high to indicate a successful send operation for the Iterators MSG so that we can send the iterators of the next mapped task to the RM if exists in the same manner as shown in Figure A.3.

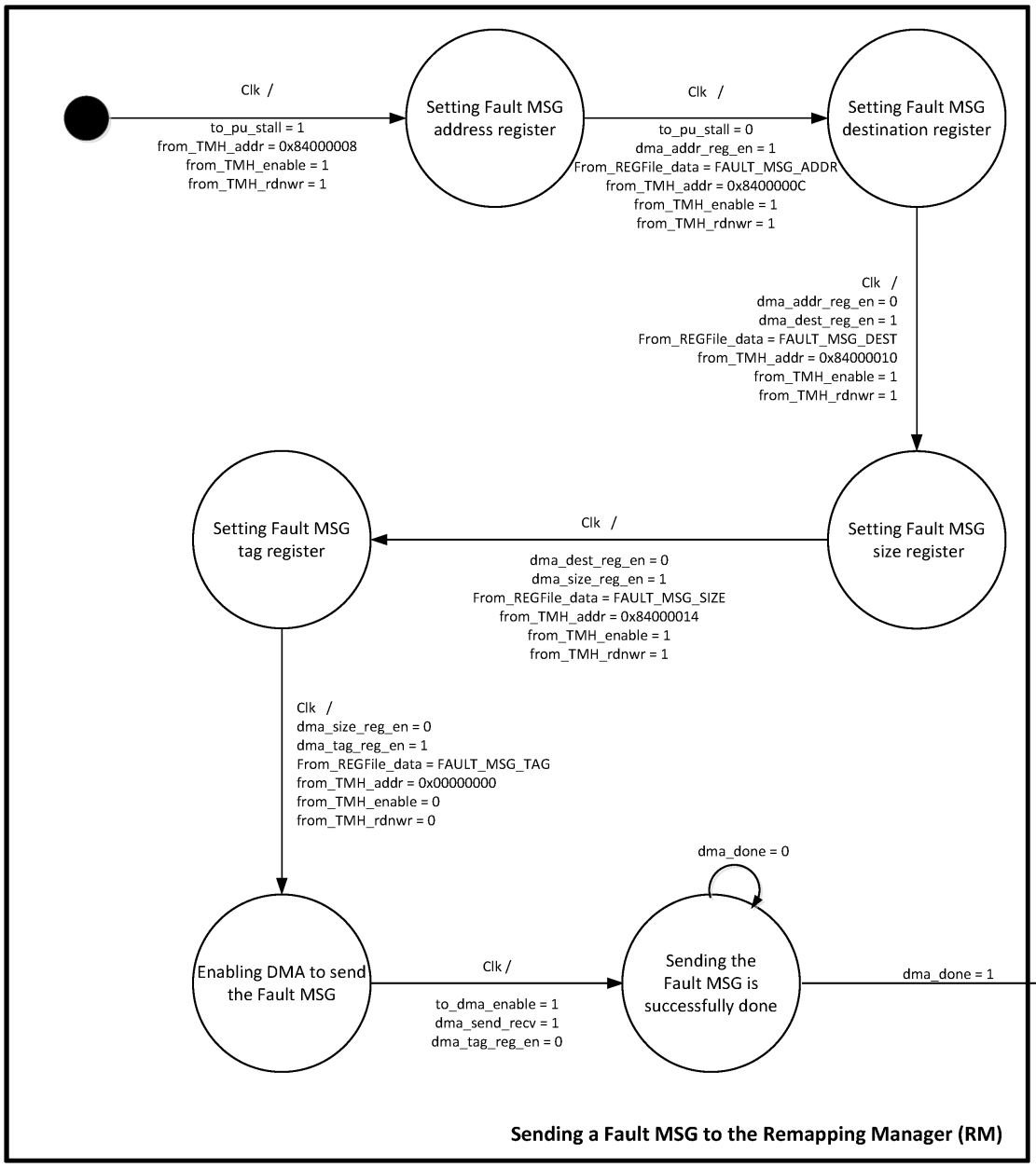


Figure A.1: The FSM of sending a Fault detection MSG to the RM

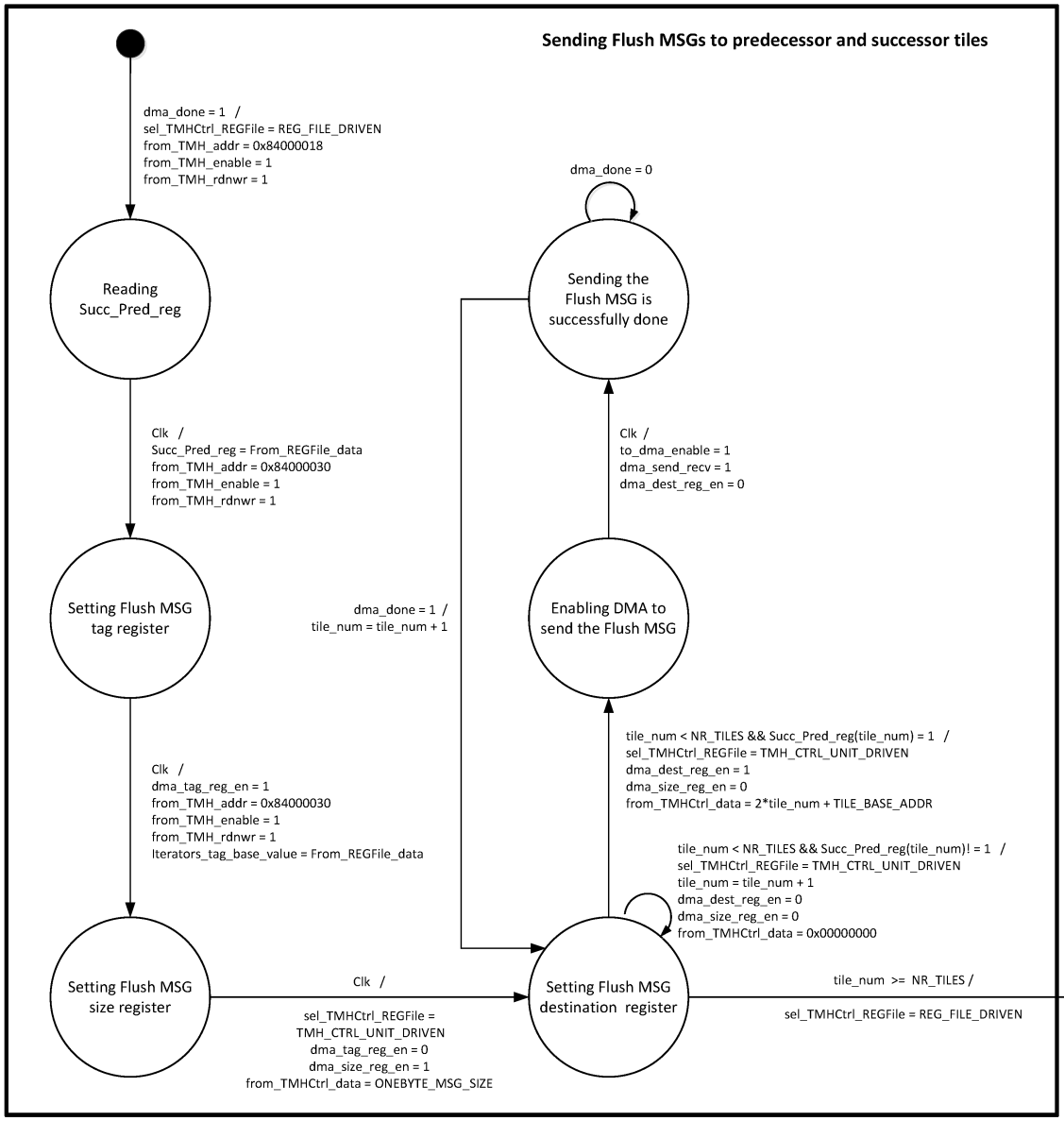


Figure A.2: The FSM of sending Flush MSGs to the predecessor and successor tile(s)

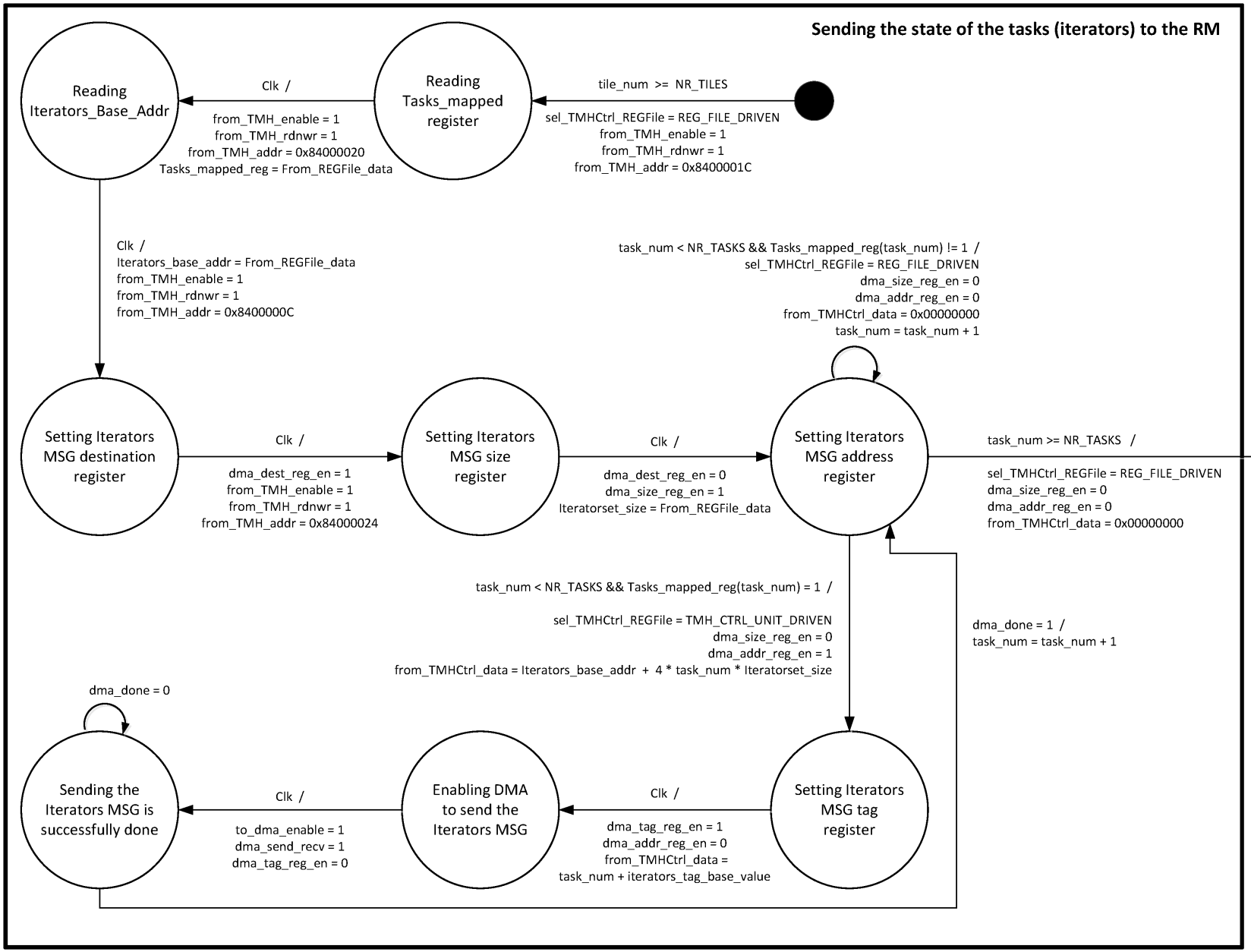


Figure A.3: The FSM of sending the state of the tasks (Iterators MSG) to the RM

The FSM of sending the presence of pending request (Request MSG) to the RM

Figure A.4 shows the state diagram to send the pending requests of the channels to the RM. firstly, the values of *CHANNELS_MAPPED_TO_THIS_TILE*, *TMH_CHANNELS_TAG_BASE_VALUE* and *TMH_CHANNELS_REQUEST_BASE_AD* registers are loaded from the register file respectively. In the next state, the current number of tokens of the channel is read from the saved channel info in the TMH register file. The parameters of the send operation are loaded in the TMH shmpi_send registers (address, tag and size) then when *dma_done* is sampled high, the Request MSG is sent successfully,

The FSM of sending the number of the tokens (NB_TKNs MSG) to the RM

The values of the data, tag, address and size are loaded sequentially in the TMH shmpi_send registers then the DMA is enabled to send the NB_TKNs MSG. When the send operation is done correctly, the number of tokens (*nb_tkns*) of the channel is checked. If the *nb_tkns* is zero then the next state is to send the pending requests of the next channel otherwise the next state is to send the tokens of that channel as shown in Figure A.5.

The FSM of sending the tokens of the channel (Ch_tokens MSG) to the RM

In the same way as shown in Figure A.6, the values of the size, address and tag are loaded in the TMH shmpi_send registers then the DMA is enabled to send the Ch_tokens MSG to the RM. when the operation is successfully done, the next state is to send the channel info (pending request, nb_tkns and tokens) of the next channel if exists.

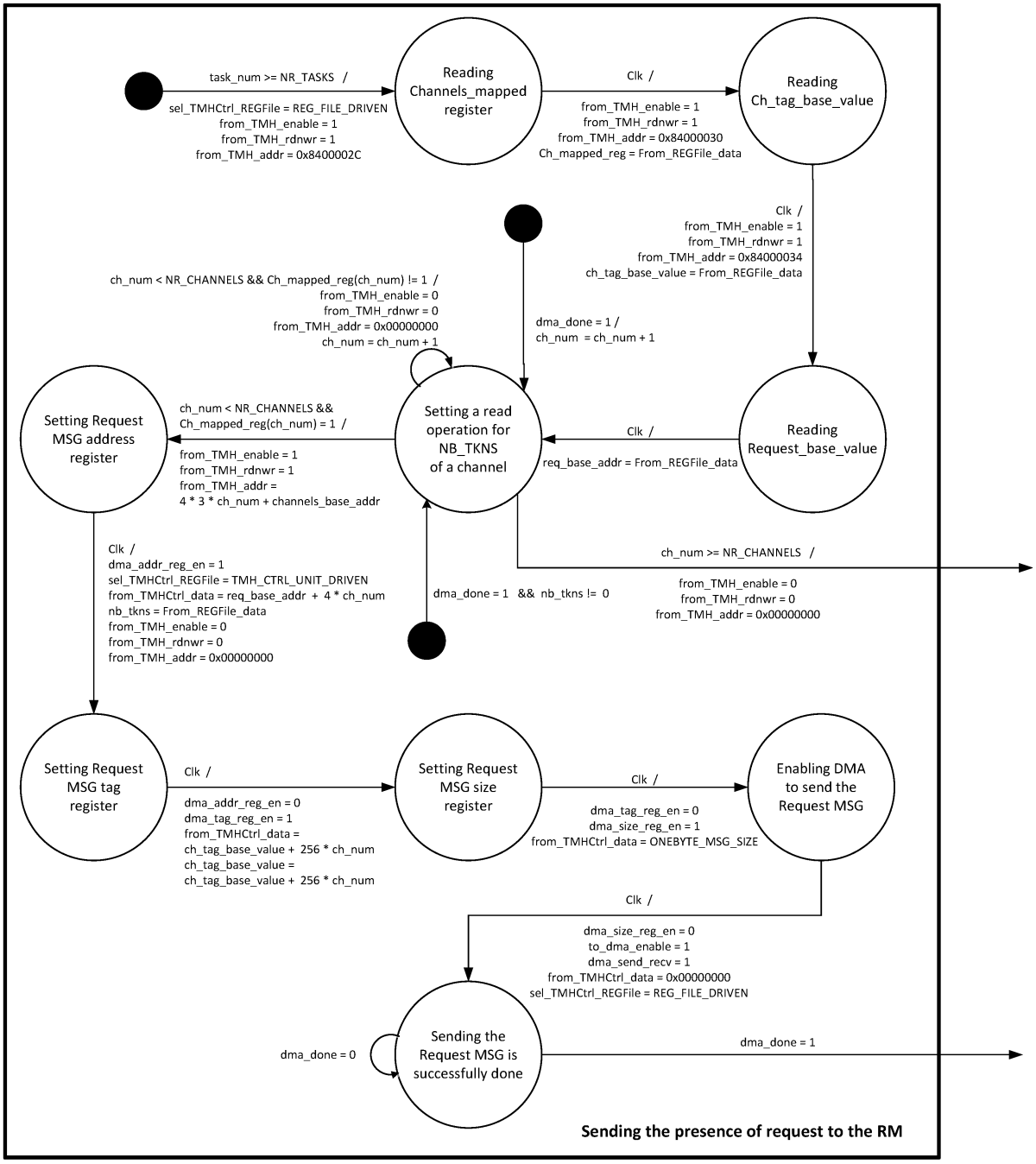


Figure A.4: The FSM of sending the presence of pending request (Request MSG) to the RM

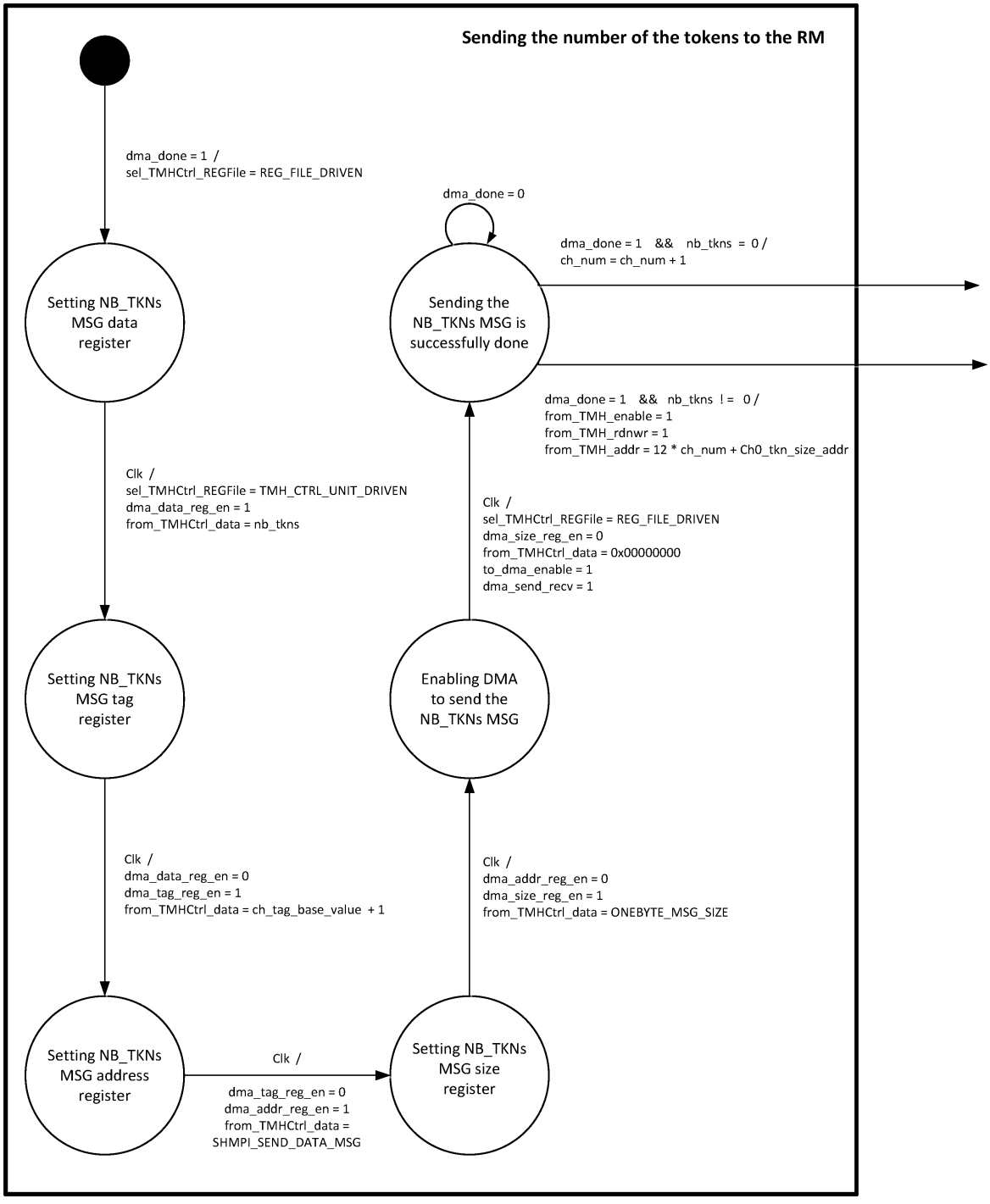


Figure A.5: The FSM of sending the (NB_TKNs MSG) to the RM

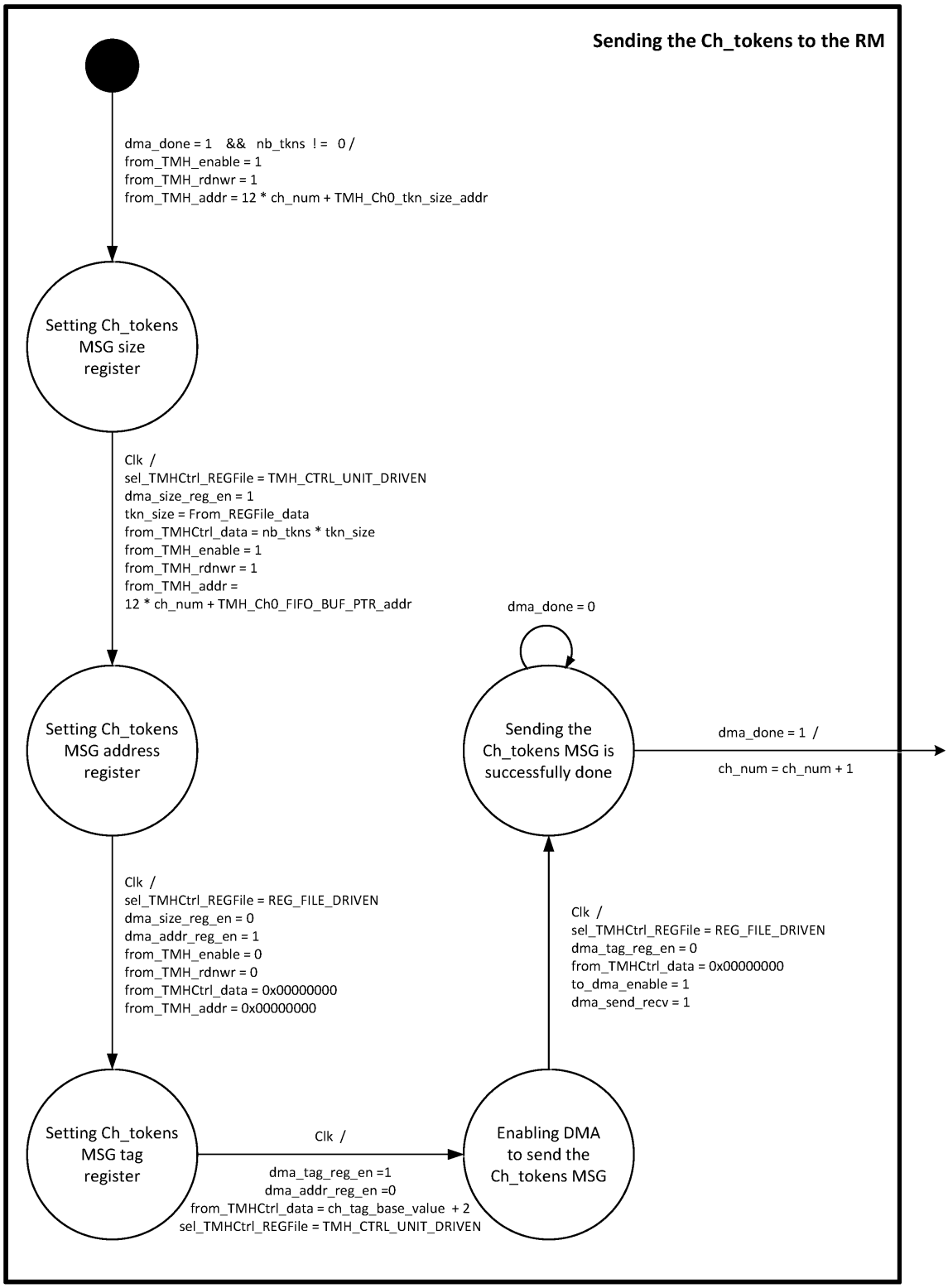


Figure A.6: The FSM of sending the tokens of the channel (Ch_tokens MSG) to the RM

Acronyms

BAS Baseline architecture.

CRC Cyclic Redundancy Code.

DCT Discrete Cosine Transform.

DUT Device Under Test.

FIFO First In First Out.

FPGA Field Programmable Gate Array.

FRWEP Fault Recovery with Error Propagation.

FRWOEP Fault Recovery without Error Propagation.

FSM Finite State Machine.

IP Intellectual Property.

KPN Kahn Process Network.

LUT Look Up Table.

MJPEG Motion JPEG.

MMU Memory Management Unit.

MTOS Multi-Threaded Operating System.

MUX Multiplexer.

NI Network Interface.

NoC Network on Chip.

NORMA No Remote Memory Access.

OCP Open Core Protocol.

PE Processing Element.

PPN Polyhedral Process Network.

RM Remapping Manager.

SECDED Single Error Correcting and Double Error Detecting.

SPF Silicon Protection Factor.

STM Self Testing Module.

TMH Task Migration Hardware.

TMR Triple Modular Redundancy.

VHDL Very High-Speed Integrated Circuit Hardware Design Language.

Bibliography

[1] C.-L. Chou and R. Marculescu, "Farm: Fault-aware resource management in noc-based multiprocessor platforms," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, march 2011, pp. 1 –6.

[2] C. Ababei and R. Katti, "Achieving network on chip fault tolerance by adaptive remapping," *Int. Parallel and Distributed Processing Symposium*, vol. 0, pp. 1–4, 2009.

[3] M. Malek, "A comparison connection assignment for diagnosis of multiprocessor systems," in *Proceedings of the 7th annual symposium on Computer Architecture*, ser. ISCA '80. New York, NY, USA: ACM, 1980, pp. 31–36. [Online]. Available: http://doi.acm.org/10.1145/800053.801906

[4] D. Gizopoulos, "Online periodic self-test scheduling for real-time processor-based systems dependability enhancement," *Dependable and Secure Computing, IEEE Transactions on*, vol. 6, no. 2, pp. 152–158, April-June 2009.

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking," in *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, ser. LNCS, vol. 2404. Copenhagen, Denmark: Springer, July 2002.

[6] O. Derin, D. Kabakci, and L. Fiorin, "Online task remapping strategies for fault-tolerant network-on-chip multiprocessors," in *Proc. of the 5th ACM/IEEE Int. Sym. on Networks-on-Chip*, 2011, pp. 129–136.

[7] A. P. Frantz, M. Cassel, F. L. Kastensmidt, E. Cota, and L. Carro, "Crosstalk- and seu-aware networks on chips," *IEEE Design and Test of Computers*, vol. 24, pp. 340–350, 2007.

[8] L. Fiorin, L. Micconi, and M. Sami, "Design of fault tolerant network interfaces for nocs," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*, 31 2011-sept. 2 2011, pp. 393 –400.

[9] "Nangate 45nm open cell library." [Online]. Available: http://www.nangate.com/

[10] K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky, "Bulletproof: a defect-tolerant cmp switch architecture," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, feb. 2006, pp. 5 – 16.

[11] T. Bengtsson, S. Kumar, and Z. Peng, "Application area specific system level fault models: A case study with a simple noc switch," in *Third IEEE International Workshop on Electronic Design, Test and Applications : proceedings, 17-19 January 2006, Kuala Lumpur, Malaysia*. IEEE Computer Society, 2006.

[12] A. Benso and P. Prinetto, Eds., *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, ser. Frontiers in Electronic Testing. Springer, 2003, vol. 23.

[13] M. Y. Hsiao, "A class of optimal minimum odd-weight-column sec-ded codes," *IBM J. Res. Dev.*, vol. 14, pp. 395–401, July 1970. [Online]. Available: http://dx.doi.org/10.1147/rd.144.0395

[14] I. Koren and C. M. Krishna, *Fault Tolerant Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.