SEVENTH FRAMEWORK PROGRAMME
THEME
FET proactive 1: Concurrent Tera-Device Computing (ICT-2009.8.1)

PROJECT NUMBER: 249013

TERAFLUX

Exploiting dataflow parallelism in Teradevice Computing

D5.2 – Development of Inter-Cluster Fault Detection Mechanisms and Core-Internal SW and HW Protection

Due date of deliverable: 31st December 2011
Actual Submission: 31st December 2011

Start date of the project: January 1st, 2010 Duration: 48 months

Lead contractor for the deliverable: UAU

Revision: See file name in document footer.

Project co-founded by the European Commission within the SEVENTH FRAMEWORK PROGRAMME (2007-2013)	
Dissemination Level: PU	
PU	Public
PP	Restricted to other programs participant (including the Commission Services)
RE	Restricted to a group specified by the consortium (including the Commission Services)
CO	Confidential, only for members of the consortium (including the Commission Services)

Change Control

Version#	Date	Author	Organization	Change History
0.1	**22.08.2011**	**Sebastian Weis**	**UAU**	**Initial release**
0.2	**21.10.2011**	**Sebastian Weis, Arne Garbade, Bernhard Fechner, Julian Wolf, Sebastian Schlingmann**	**UAU**	**Merged sections**
0.3	**27.10.2011**	**Sebastian Weis, Arne Garbade, Bernhard Fechner**	**UAU**	**Revised sections**
0.4	**20.11.2011**	**Avi Mendelson**	**MSFT**	**Integrated OS part**
1.0	**25.11.2011**	**Sebastian Weis, Arne Garbade, Bernhard Fechner, Julian Wolf, Sebastian Schlingmann, and Theo Ungerer**	**UAU**	**First full version**
1.1	**29.11.2011**	**Arne Garbade, Sebastian Weis**	**UAU**	**Minor Changes - Layout**
1.2	**05.12.2011**	**Sebastian Weis, Arne Garbade**	**UAU**	**Integrated review comments from Partner UCY and MAN**
1.4	**19.12.2011**	**Sebastian Weis, Arne Garbade, Theo Ungerer**	**UAU**	**Final version**

Release Approval

Name	Role	Date
Sebastian Weis	**Originator**	**12.12.2011**
Theo Ungerer	**WP Leader**	**19.12.2011**
Roberto Giorgi	**Project Coordinator for formal deliverable**	**30.12.2011**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

The following list of authors will be updated to reflect the list of contributors to the writing of the document.

Sebastian Weis, Arne Garbade, Sebastian Schlingmann,
Julian Wolf, Bernhard Fechner, Theo Ungerer
University of Augsburg

Avi Mendelson, Doron Shamia
Microsoft Research and Development

Glossary

D-FDU	Distributed Fault Detection Unit
DF-Thread	A dataflow thread
D-TSU	Distributed Thread Synchronization Unit
FM	Frame Memory
L-FDU	Local Fault Detection Unit
L-TSU	Local Thread Synchronization Unit
MAPE	Acronym for Monitoring, Analysing, Planning, and Executing
MCA	Machine Check Architecture
Leading Thread	Represents the main executed thread in the double execution approach
NoC	Network-on-Chip
Node	Group of cores and additional TERAFLUX hardware units
OWM	Owner Writable Memory
TCL	Thread-to-Core List (cf. D6.1)
Trailing Thread	Represents the duplicated thread in the double execution approach

Executive Summary

This deliverable reports on the research carried out in the context of DoW Task 5.2 (project months 13 -24) **"Development of Inter-Cluster Fault Detection Mechanisms and Core-Internal SW and HW Protection"**:

- We refined our fault-tolerant threaded dataflow architecture to detect faulty cores, routers, and links, including different types of FDUs on core and node level. We exploited the Machine Check Architecture for core level fault detection and present two techniques to detect soft errors (program flow checking and double execution with voting) by the D-FDU and use the side-effect free execution of DF- threads for an easy fault recovery by thread re-execution.
- We investigated clustering of cores into "nodes" or "cluster" located on a 2D-mesh structured Network on Chip with faulty elements.
- We specified the inter-cluster fault detection mechanism between D-FDUs and present grouping strategies for the mutual D-FDU monitoring.
- We extended the inter-cluster fault detection concept to detect faults in I/O Devices and memory controllers.
- We enhanced the operating system work of year 1 to recover from faults on higher level.

Hence, all goals of WP5 for the second year were achieved.

1 Introduction

In Deliverable D5.1 we have described Fault Detection Units to detect faults in cores, clusters, and the interconnection network. We have presented a detailed concept of the internal FDU behavior as well as different communication protocols and fault detection message types.

This Deliverable extends the hierarchical FDU concept and the monitoring architecture introduced in Deliverable D5.1. It further focuses on the *inter-cluster fault detection mechanisms based on the defined overall architecture (see DoW Task 5.2).*

The DoW Task 5.2 defines three subtasks:

1. "*Development of clustering of cores: We will develop strategies to form clusters which means that a set of cores is assigned to a FDU. The FDU is responsible to monitor all cores within its cluster.*"
2. *"Development of grouping strategies for FDUs: Additionally grouping strategies for FDUs are needed. Such a group only consists of FDUs and all FDUs within the same group monitor each other. This allows for a detection of faults of FDUs or whole clusters. After the detection of faults restructuring of clusters and groups may be necessary."*
3. "*Development of inter-cluster fault detection mechanisms: Based on the structure of the groups of FDUs mechanisms will be developed which allow for a mutual fault detection of FDUs within the same group. It is necessary to analyse different sources of information (e.g. core-internal fault detection, lifesign-messages) to analyse faults. These range from soft-errors of an FDU to a permanent failure of a whole cluster."*

Clustering of cores is covered in Section 3, where we depict the technical boundaries of the core/D-FDU ratio with respect to a 2D-mesh structured interconnect. We first present a static clustering technique without link faults. Then we incorporate link and router faults during operation, which led us to a dynamic re-clustering algorithm to prevent bottlenecks in the interconnection network.

Since the grouping strategies for D-FDUs and the inter-cluster monitoring are strongly coherent, we have aggregated subtasks 2 and 3 in Section 4. Here, we describe the technical details of the inter-node monitoring between D-FDUs, which exploits techniques for the D-FDU-core monitoring. Later on we discuss different inter-cluster grouping strategies to detect D-FDU faults.

Additionally the reviewers of the project suggested:

"Soft-error tolerance and attention to fault detection within FDUs as well as fault tolerance between them (using TSUs) in WP5 plus considerations of including power management in FDUs and TSUs; also the granularity of the fault tolerance substructure (i.e. FDU/TSU to core ratio) should be considered more carefully."

We have paid special attention to the increasing soft error rates in future VLSIs and propose Double Execution of TERAFLUX threads and control flow checking in our enhanced fault-detection architecture to detect soft errors in the cores and the D-FDUs (see Section 2 and Section 4). For the power management we propose to share responsibilities between the D-FDU, which is responsible for dynamic voltage and frequency scaling and ensuring the reliability within its cluster, and the D-TSU,

which has knowledge about thread execution. We will target dynamic voltage and frequency scaling in project year 3.

Furthermore, "Partner MSFT will focus on other parts of the system which are not CPU, such as I/O, peripheral devices etc and examine how to detect errors from these sources and how we can recover from them. Extend the work to include SW and HW protection (not HW only): It looks like that recover from fault which are external to the core may require a new SW/HW interfaces. Partner MSFT will extend its work to examine these aspects of the problem and come out with a holistic approach that aims to allow a full recovery from errors regardless their origin."

Similar to D-FDU to D-FDU monitoring that is performed with the same method like D-FDU-core monitoring; we extended the monitoring technique to I/O devices by the following approach. We assume that I/O controllers are situated at the edges of the chip, attached to the inter-node NoC. Each I/O controller is monitored by an associated D-FDU like a normal core. However, the I/O controller core monitors all the attached I/O devices and sends its health state on the heartbeat messages to the D-FDU.

Moreover, MSFT performed additional work on the OS integration into TERAFLUX, in particular the OS level of fault detection and thread level scheduling to D-TSUs.

1.1 Document structure

In Section 2 we refine the high level fault detection architecture introduced in D5.1. We further introduce two new fault detection mechanisms and an easy recovery technique, based on the TERAFLUX execution model. In Section 3 we depict techniques to derive the best number of cores per node. Section 4 covers inter-cluster fault detection, fault detection in I/O devices and memory controllers, and discusses grouping strategies based on mutual D-FDU monitoring to detect D-FDU and cluster faults. Section 5 extends the Operating System work of MSFT to detect and tolerate faults on system level layer.

1.2 Relation to other deliverables

This Deliverable bears relation to

- D5.2 extends the fault detection architecture of D5.1.
- D5.2 is built upon the TERAFLUX architecture described in D6.1.
- The proposed fault detection and recovery techniques of D5.2 are also shortly described in D6.2.
- Fault injection techniques are part of D7.3.

1.3 Activities referred by this deliverable

This deliverable refers to the research carried out in WP5 in project year 2.

2 Refined Fault Detection and Recovery Architecture

The TERAFLUX threaded dataflow architecture and execution model is described in the Deliverables D6.1 and D6.2. It is designed to fully exploit the Thread-Level Parallelism (TLP) provided by future parallel systems. Furthermore, it addresses scalability by a hierarchical structured execution model.

This section enhances the TERAFLUX architecture to a fault tolerant architecture [17], using our extended Fault Detection Mechanisms and exploiting the side-effect free semantic of the TERAFLUX execution model.

2.1 Enhanced high level TERAFLUX architecture

In the following we describe the enhancements for a fault tolerant architecture integrated into the TERAFLUX architecture (shown in Figure 1), including the newly introduced L-FDU and three techniques to detect faults in such architectures, namely the Machine Check Architecture, Control Flow Checking, and Double Execution of DF-Threads. Note that we will describe the TERAFLUX architecture from our fault detection point of view, focusing on its fault detection and recovery benefits.

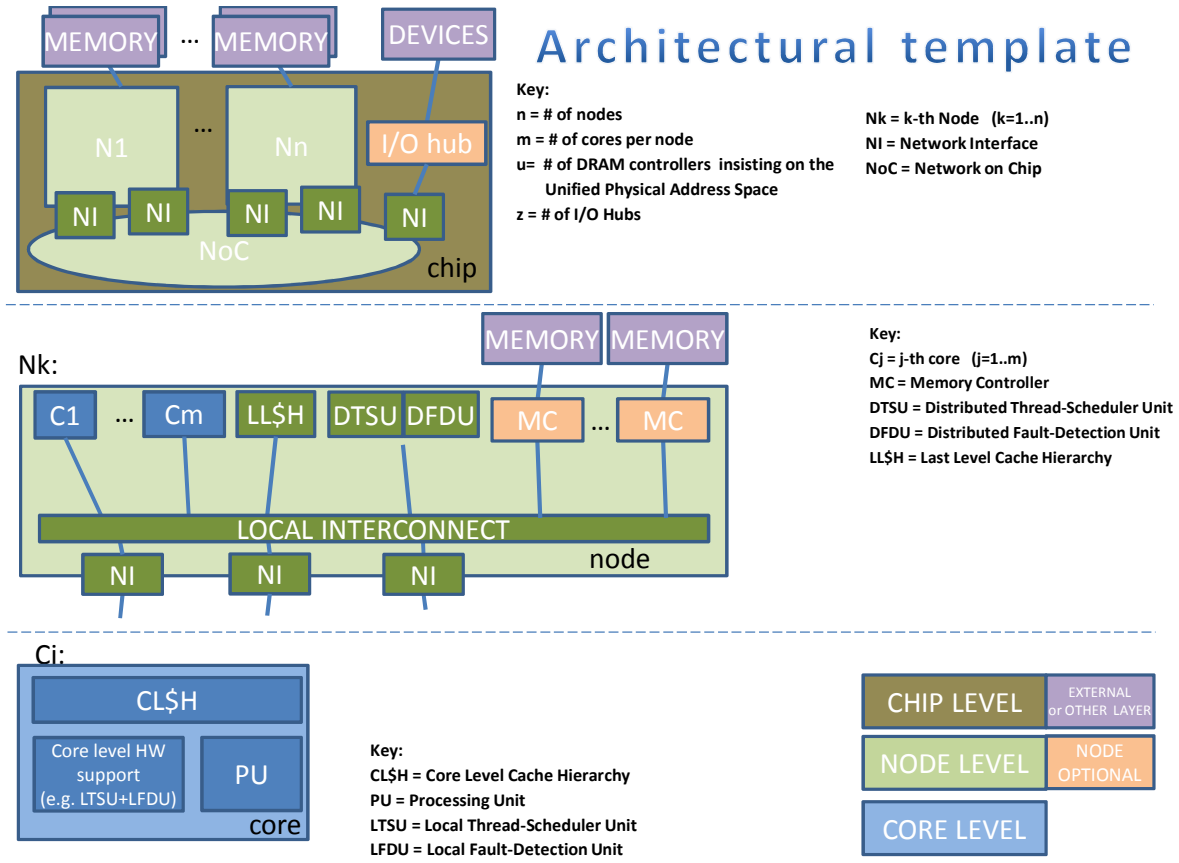

Figure 1: High level TERAFLUX architecture

On core level, the basic elements of the TERAFLUX architecture are single cores containing an x86-64 pipeline (x86-64 ISA with dataflow extensions [10]) along with a core-level cache hierarchy. Each core includes special hardware extensions consisting of two modules:

- The Local Thread Scheduling Unit (L-TSU) is responsible for scheduling threads on its affiliated core and communicating with other L-TSUs or the node's D-TSU.
- The Local Fault Detection Unit (L-FDU) is responsible for the detection of faults and reliability management within a core.

Beside the L-TSU and L-FDU, each core stores the data of a running thread in the Frame Memory (FM) or the Owner Writable Memory (OWM). The FM or the OWM are filled with the thread's data (denoted as thread frame) before execution. Please note that pure dataflow threads (DF1A, DF1B) are not allowed to read from other thread frames (in FM or OWM). However, writes into disjoint locations are permitted to support communication between threads in order to provide the inputs for subsequent threads (c.f. Deliverable D7.1).

On node level, the Distributed Thread Scheduling Unit (D-TSU) coordinates the scheduling of the threads to cores within a node and takes care of inter-node communication with other D-TSUs. Therefore, the D-TSU holds a table for bookkeeping the scheduled continuations to the cores within its node, the Thread-to-Core-List (TCL) (cf. Deliverable D6.1 – Section 5.1.3). This table is crucial for our thread-recovery mechanisms, since it supports thread restarts within a node. The Distributed Fault Detection Unit (D-FDU) is responsible for fault detection, performance monitoring, and reliability management on node-level (see Deliverable D5.1).

For the communication between nodes, the TERAFLUX architecture defines an interconnection network. We assume that the structure of this network is a 2D-mesh. All communication from one node to another will be handled by the interconnection network. Furthermore, we consider memory controllers to access off-chip memory and I/O-controllers on node level. The controllers are connected to the interconnection network as well.

A usual TERAFLUX program is partitioned in coarse-grained dataflow threads (DF1A, DF1B, DF2, or DF2 with Transactions). For the rest of this section we only target DF1A and DF1B threads, which fully obey the side-effect free execution rule.

The execution of a dataflow thread consists of three phases. First, the preload phase loads data from the FM or OWM and stores it into the core registers. The second phase is the thread execution, where the thread executes without any memory access. The third phase is the poststore phase, where the results from the thread execution are written to the consumer thread frames.

Beside the frame, each thread encloses an assigned control structure called continuation. The continuation stores control information about the thread, i.e. the pointer to the thread frame, the program counter, and the synchronization count (number of empty inputs). For a more detailed description see Deliverables D6.1 and D6.2. A dataflow thread will be scheduled for execution if and only if all inputs have been written to its thread's frame and therefore its synchronization count is zero.

The already introduced Fault Detection Units (FDUs) on core and node level are the central hardware support units for our comprehensive fault detection approach. The Distributed FDU (D-FDU) is an observer-controller unit operating on node level. As such, a D-FDU autonomously queries and gathers the health states of all cores within its node over the unreliable local interconnect and the NoC. The D-FDU is supported by the L-FDUs with each node's core. In addition, D-FDUs monitor each other in order to detect faults of other D-FDUs in other nodes. The D-FDU analyses the gathered information and provides the D-TSU with information about the state of the whole node and other D-FDUs.

The L-FDU is a small hardware unit implemented on each core to support fault detection by the D-FDU by extracting information from the Machine Check Architecture (MCA), the Performance Counters, and the Control Flow Checker.

Basically, the L-FDU has two tasks:

- Reading out the fault detection registers of the monitored core, i.e. results of the Machine Check Architecture, the Performance Counters, or the Control Flow Checker.
- Periodic communication with the D-FDU by sending heartbeat messages of the core.

Concerning intra-node fault detection, the D-FDU detects core and link failures and informs the D-TSU about the faulty components, while the D-TSU is responsible for thread recovery and restart.

The internal behaviour of the D-FDU is adapted from an autonomic computing approach, which organizes the operation principle into the four consecutive steps: Monitoring, Analysing, Planning, and Executing (MAPE) (see Deliverable D5.1 and Weis et al. [16]). The MAPE cycle operates on a set of managed elements, comprising intra-node (cores and D-TSU) and inter-node elements (other D-FDUs) in other nodes D-FDUs detect faults and proactively maintain the operability of the node they monitor, for example by dynamically performing clock and voltage scaling while monitoring the cores' error rates, temperatures, and utilization. In this context proactive means the prediction of a core's health state based on monitored information and taking action before the core gets damaged.

The intra-node monitoring of cores, D-TSU, and D-FDU is separated in two categories: time and event-driven.

Time-driven messages are heartbeat messages that include a set of core health information. The D-FDU expects a heartbeat message of a core in a certain time interval. If no heartbeat messages arrive at the D-FDU within the expected interval, the associated core will be suspected as faulty. A permanent fault of a core can be expected when multiple faults are detected in a short period of time. As a consequence, the D-FDU considers the core as completely broken and informs the D-TSU. The D-TSU itself is monitored by the D-FDU with the same techniques as a regular core. Thus, D-TSU faults can be detected as well. The D-FDU communicates with the D-TSU via command messages, i.e. notify, request, and response messages. The D-TSU requests the D-FDU to change the frequency of a core or to reduce the frequency in the case of low workload, while the D-FDU reports the D-TSU on thermal and error conditions. In case of an intermittent or permanent fault, the D-TSU temporarily or permanently stops scheduling any threads to the broken core.

Event-driven messages are alert messages in case of core faults. These messages are triggered by the L-FDU and notify the affiliated D-FDU within the node.

Recent microprocessors are equipped with an architectural subsystem called Machine Check Architecture (MCA) that is able to detect and correct certain faults. For instance for the AMD K10 processor family, the MCA can detect faults in the data and instruction cache, the bus unit, the load-store unit, the northbridge, and the reorder buffers (for more details see Deliverable 5.1).

We exploit this state-of-the-art fault detection technique to explicitly safeguard memories and communication channels in the TERAFLUX architecture. This incorporates explicitly the main memory, Frame Memories, Owner Writable Memories, and point-to-point interconnections. Since frequent occurrences of errors detected by the MCA can be an indicator for intermittent or permanent faults, or a permanent breakdown of the whole core, the L-FDUs transmit this information within its periodic heartbeat messages to the D-FDU. Based on this, the D-FDU can make predictions about the current reliability state of the core.

2.2 Core-level Fault Detection Mechanisms

Beside the Machine Check Architecture, we incorporate two additional techniques to detect faults within the core; the cheaper Control Flow Checker and the more costly Double Execution.

2.2.1 Control Flow Checker

According to the TERAFLUX thread definitions in deliverable D6.1, all thread types except for DF1a can contain internal control flow caused by jump, branch or loop instructions. Fault injection studies [9, 14] show that a high amount of errors occurring in a computer system are control flow errors, i.e. errors which cause timing or logical divergence from the proper control flow. Control flow errors are mainly caused by single event upsets (SEUs) and single event transients (SETs) either in the memory or in processor-internal components modifying e.g. the instruction opcode or the program counter value during execution. We introduce a lightweight mechanism to detect control flow errors during runtime. Compared to Double Execution (as described in Section 2.2.2), which detects faulty behaviour after the execution of threads, the control flow checking technique can speed up fault detection and permits lower detection latencies for transient and permanent errors affecting the control flow, while having small overhead in execution time.

The general concept of such a checking mechanism is to verify that the runtime control flow of a thread corresponds to its expected behaviour. As we focus on errors caused by transient, non-reproducible faults, an on-line error detection mechanism is the only feasible solution to detect such control flow errors. Typical design parameters for this kind of detection mechanism are fault coverage, detection latency, overhead concerning both memory and execution time, and the monitoring hardware complexity [11].

Techniques for a detection of control flow errors can be implemented in hardware or software. Accordingly, these approaches either introduce an additional hardware block, like a watchdog processor performing reliability checks during runtime or they add supplementary code on software-level to perform monitoring operations. However, both alternatives have benefits and drawbacks as well: While hardware-based approaches usually provoke high complexity for the integration into a

system, the advantage is a good average performance due to less overhead. Moreover, most of these techniques do not require changes in the executed application. Software-based approaches on the other side are easy to integrate, but cause significant overhead concerning memory usage and execution time. Also, it is needed to add redundant information to the application source code. A solution for this dilemma can be a *hybrid* detection technique, combining benefits of both hardware and software-based approaches.

The error detection mechanism proposed here is based on a new approach of checking both the timing behavior and the logical control flow of threads during run-time. This combination promises a much better fault coverage compared to a stand-alone temporal or logical check mechanism. Our approach is a hybrid hardware-software technique, which basically consists of two steps:

- an *off-line* phase, in which the safety-critical application is split, analysed, and hardened with additional check points in the program code, and
- a *run-time* phase, in which a connected hardware check unit reacts to the inserted check points during program execution.

Similar to other control flow checking approaches [8, 5] our technique is based upon partitioning the program code into basic blocks [1]. There is no instruction like a jump, branch, or call within a basic block, which could change the control flow, except possibly for the last instruction. Moreover, no instruction in the basic block can be the destination of a jump, branch or call, except potentially the first one.

To provide a high level of clarity, we separate the description of our technique into two parts: Firstly, we explain the instrumentation and checking mechanism only for timing errors occurring in the control flow. Secondly, the additional part focusing on the detection of logical control flow errors is presented.

2.2.1.1 Temporal Control Flow Monitoring

After splitting the code into fine-grained basic blocks, we add check points at the beginning of each basic block containing its maximum execution time (MET) in clock cycles or milliseconds. From a technical point of view, such a checkpoint consists of one or more instructions providing a specific value symbolizing the upper bound of a block's execution time to the hardware check unit (e.g. by writing to a defined register). Since these integrated instructions extend the execution time, a timing analysis of an application has to take the added check points into account. Therefore, it is necessary to temporarily add "empty" check points to enable a correct timing analysis of the overall execution time including these instructions. In the following step, the result of the analysis will be integrated in the check points. Figure 2 shows some details on this instrumentation procedure: Empty checkpoints are added to the basic blocks BB_i, BB_j and BB_k, and then each MET is calculated and inserted.

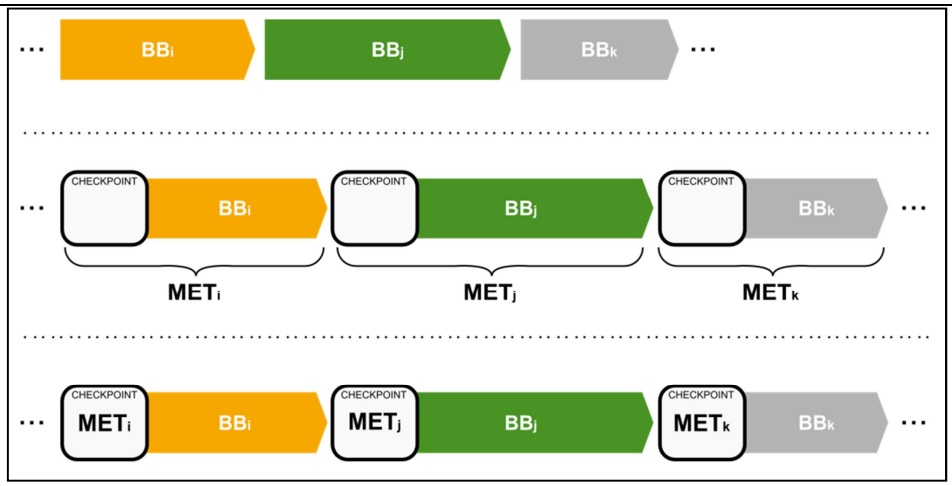

Figure 2: Basic blocks without instrumentation, with empty checkpoints and with completely filled checkpoints.

The result of the off-line phase is the application code, in which each basic block is instrumented with its off-line analysed MET value. To use this information for error detection, we integrate a specific hardware check unit connected to the processor. This check unit reads the MET values from the check points during runtime and is able to detect an erroneous timing behaviour as follows: As soon as the program execution reaches a checkpoint, the processor transfers the MET value to the hardware check unit, which writes it to a defined *Checkpoint Value Register (CPVR)*. This CPVR value is decremented by the check unit at each following processor cycle. When the next checkpoint is reached, this MET value is written to the CPVR. This mechanism makes sure that the CPVR will never obtain the value 0, if the program is correctly executed. Whenever the control flow leaves a basic block within the previously calculated MET, the following checkpoint is reached and the CPVR is updated by a (usually) higher value. Therefore, we can assume a timing error, if the CPVR reaches the value 0. In this case, a basic block required more cycles than the timing analysis had computed off-line.

This technique of temporal control flow monitoring can be easily enhanced to support also a detection of timing errors in DF1a threads, which do not contain control flow instructions. Thus, a DF1a thread is similar to a basic block and we can insert the first checkpoint right in front of the first instruction of that thread. Moreover, we integrate a checkpoint either before or immediately after the thread's last instruction. By this, we can detect timing errors analogously to the other thread types.

2.2.1.2 Logical Control Flow Monitoring

Beside the check of the correct timing behaviour, it is essential to detect an erroneous logical control flow with low detection latency, i.e. within a few processor cycles. For this purpose, we can easily extend the included mechanism for temporal control flow monitoring, consisting of off-line software instrumentation and runtime checking. By this, we can provide both temporal and logical control flow checking, having little additional overhead.

In this section, we cover also cases not strictly related to the Data-Flow execution model as outline in most of the other deliverables. In particular, we cover the general case that may be useful for the legacy or system threads (L-Threads, S-Threads as defined in D7.1, D6.1), i.e., Non-DF Threads.

In general, checking a logically correct sequence assumes that single elements can be identified. If an application's code is split into basic blocks, their succession during execution is analysable. We annotate each basic block with a unique identifier (ID) which is added to the checkpoint containing the MET value. Moreover, we enhance the functionality of the hardware check unit to be capable of monitoring the correct sequence of basic blocks and detecting logical control flow errors, too.

It is required to enable a fast and easy check during runtime causing little overhead. In order to satisfy these demands, we develop a technique to explicitly predict successors: IDs can be arbitrarily assigned to basic blocks. Along with each ID, we store the pre-calculated successor ID (also called *predicted successor*) or a list of two possible successor IDs of this basic block. So, the hardware checker compares during the runtime phase, whether an actually executed basic block is an allowed successor. To give a better understanding of our approach, we regard each possible variation of the control flow and describe the instrumentation progress in detail:

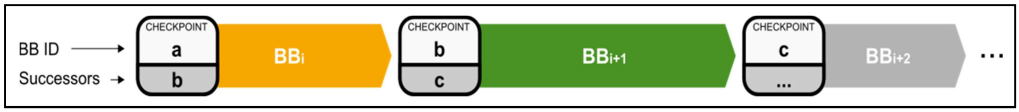

Figure 3: Sequential basic blocks

- In the *sequential* case, the last instruction of a basic block is neither a jump nor a branch instruction. That means the next executed basic block is definitely the following block in the program code. As depicted in Figure 3, we add to each checkpoint the ID of the basic block itself and the ID of its follower.

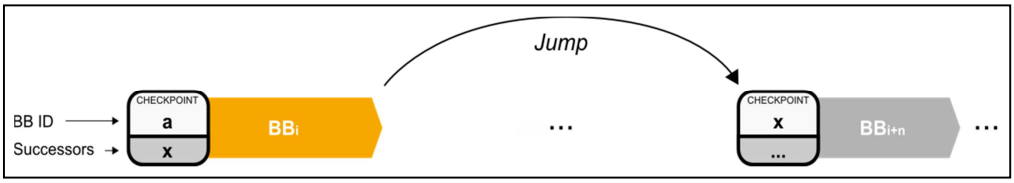

Figure 4: Basic blocks with a jump instruction

- In case of an unconditional (direct) *jump* instruction at the end of a basic block, there is only one allowed successor. Figure 4 shows how the predicted successor of BB_i has to be updated accordingly.

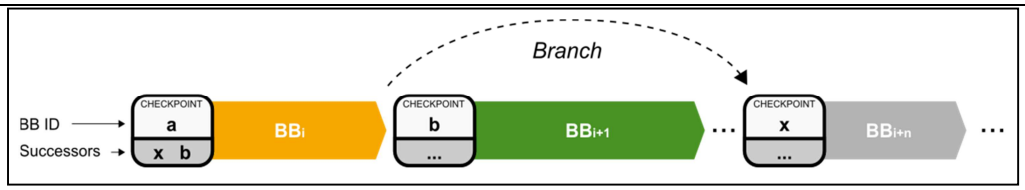

Figure 5: Basic blocks with a branch

- If a basic block ends with a *branch* instruction, there are two possible successor blocks in the control flow, depending if the branch condition is true or false. As we cannot distinguish off-line which path will be taken during execution, we add both possibilities to the checkpoint. So, basic block BB_i in Figure 5 contains the IDs of both basic blocks BB_{i+1} and BB_{i+n} in a list of successors. The instrumentation in case of a *loop* instruction at the end of a basic block is done in the same way.

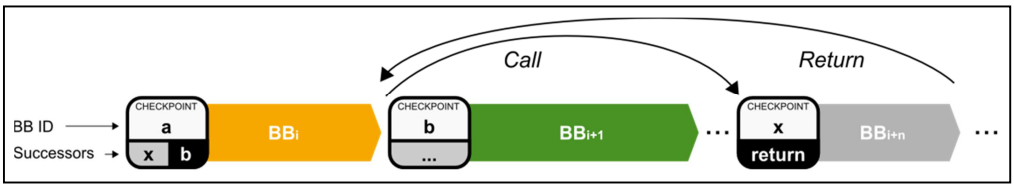

Figure 6: Basic blocks with call and return

- The handling of *calls* and *returns* is more difficult, as a function is usually called from different positions within an application. Inside the function code it is unknown, where the function was called and which will be the target of the return. In order to know which basic block follows the last basic block in a function, we have to temporarily store the position of the function call. This information can be used for a prediction of the function return. Figure 6 shows the instrumentation of calls and returns. In this example, BB_{i+n} is a function, which is called from BB_i and potentially from other basic blocks. First, we add the ID of BB_{i+n} as the only allowed successor in the checkpoint of BB_i. Moreover, we append the basic block which should be executed after the function's return (in this example BB_{i+1}). Within the function, it is now sufficient just to signal the return, because the saved value can be used for a prediction. This instrumentation mechanism also works properly for nested function calls, if we introduce a stack memory to save multiple predictions for the return IDs.

- In case of *indirect jumps*, a successor is hard to determine at compile time, as it depends on the register values during execution. However, as indirect jumps rarely occur in our context, we currently neglect this issue.

So, our instrumentation mechanism allows a handling of every common variation of the control flow, while the memory overhead caused by storing potential successors is obviously limited. Neglecting indirect jumps, the branching factor in the control flow of an application is not higher than two. Also for calls and returns it is sufficient to add two successive IDs inside a checkpoint.

The hardware check unit, which becomes active as soon as a checkpoint is detected during runtime, is enhanced to interpret the instrumented values. The check unit has to compare the current ID with the predicted successor(s). Furthermore, it must save the current prediction values for the checking progress when the application reaches the next checkpoint. According to the control flow, different compare operations have to be performed:

- In case of a *sequential* control flow or a (direct) *jump*, a basic block has only one possible successor. So, the check unit makes sure that the following ID is equal to the predicted one.
- If a *branch* or *loop* occurs, a basic block has two possible successors, depending on the branch condition. Therefore, the check unit tests if the following basic block corresponds to the first or second prediction.
- A function *call* has one allowed successor, similar to a sequential control flow. As described before, the checkpoint also contains the ID of the basic block which is executed after the return from the function. So, the check unit has to ascertain that the following basic block is the one inside the function. Moreover, it pushes the basic block ID for the return on a stack memory in order to be used for a compare operation later.
- In case of a *return* from a function, the check unit will take (and remove) the stack's top entry, in order to compare it to the following basic block ID.

For the prediction we have to provide memory for storing at most two successive ID values and a stack memory for function calls. The stack size depends on the degree of function nesting, which is usually determined by the processor architecture.

2.2.2 Double Execution

For our Double Execution approach, we duplicate DF1a and DF1b during runtime. Therefore, we follow the definitions given by Rotenberg [12] and call the thread that is duplicated *leading thread* and its duplicate *trailing thread*.

Since the execution of DF1a and DF1b threads is side-effect free and writes are only assigned once, we must only duplicate the *continuation* of a thread. This relaxes the complexity for the memory management as well as the management of the trailing thread.

Within the Thread-to-Core List (TCL) in each D-TSU, all continuations scheduled to a core within the node are redundantly stored. Our approach then only duplicates the redundantly stored continuation in the D-FDU and schedules it to another core within the node. This means we can exploit data locality by sharing the thread frame between the leading and the trailing thread.

The D-FDU within a node is finally used as the comparator of the result sets of both of the threads. In the fault free case, the writes of the leading thread are forwarded by the D-TSU, i.e. written to all consumer thread frames. Otherwise, the D-TSU triggers the thread recovery mechanism.

In more detail, double execution works as follows:

1. A thread is duplicated at that moment its synchronization count became zero, i.e. a thread has received all its inputs and is ready to execute. The L-TSU, which is in charge of scheduling the threads to its cores, then proceeds with execution of the leading thread as usual.

2. To indicate the thread duplication, the L-TSU sends notification messages to the D-TSU and the D-FDU. The D-TSU is now responsible for copying the redundantly stored continuation of the thread and distributing it to the same or another core within the node, depending on the type of fault to detect. To detect transient faults only, the D-TSU can schedule the thread to the same core. To detect permanent and intermittent faults as well, the D-TSU schedules the thread on a core within the same node, but on a different core by passing the copied continuation to the L-TSU of the core.

3. When both threads have finished execution, the L-FDU redirects the writes of the threads to the D-TSU and the D-FDU. The D-TSU buffers the writes until the D-FDU, which is in charge of comparing the results, gives a positive feedback.

4. In the case of a fault-free execution of both threads, the responsible D-FDU deletes the continuation in its TCL and forwards the writes of the leading thread to the appropriate consumer threads. In the case of a fault it has to re-execute the thread.

2.3 Fault Recovery of TERAFLUX threads

The chief advantage of the TERAFLUX execution model for fault recovery is the side-effect free thread execution. This inherent functional semantic includes execution checkpoints between pure dataflow threads (DF1a and DF1b threads). In other words, a DF1a or DF1b thread can be restarted, as long as no writes to consumer threads have taken place. This is always the case for DF1a and DF1b threads, since the output frame becomes visible only after finishing the whole execution of the producer thread. Compared to a state-of-the-art many-core systems, these checkpoints promise a smaller memory footprint and simpler semantic for simple rollback-recovery mechanisms.

Figure 7 shows how the recovery mechanism will work. Note that we implicitly assume double execution to detect faults. When the D-FDU determines a fault within a monitored core (between time T2 and T3), it provides the corresponding core ID, together with the fault information to its affiliated D-TSU. Subsequently, the D-FDU tries to determine the cause of the detected fault. Depending on the kind of the fault the D-TSU can either restart the thread (at T4, after the rollback between time T3 and T4) on the same core or re-allocate all threads of the faulty core to reliable cores. In the given case of a transient fault, usually the D-TSU will try to re-execute a thread again on the original core (at T4). The re-execution can easily be done by overwriting the continuation field at the L-TSU with the redundant continuation field hold by the D-TSU. The L-TSU will then schedule the thread again. In our approach restarting threads is assured by the D-TSU, which only forwards writes to the consuming thread frames if and only if the D-FDU signals the fault free execution of the producing thread.

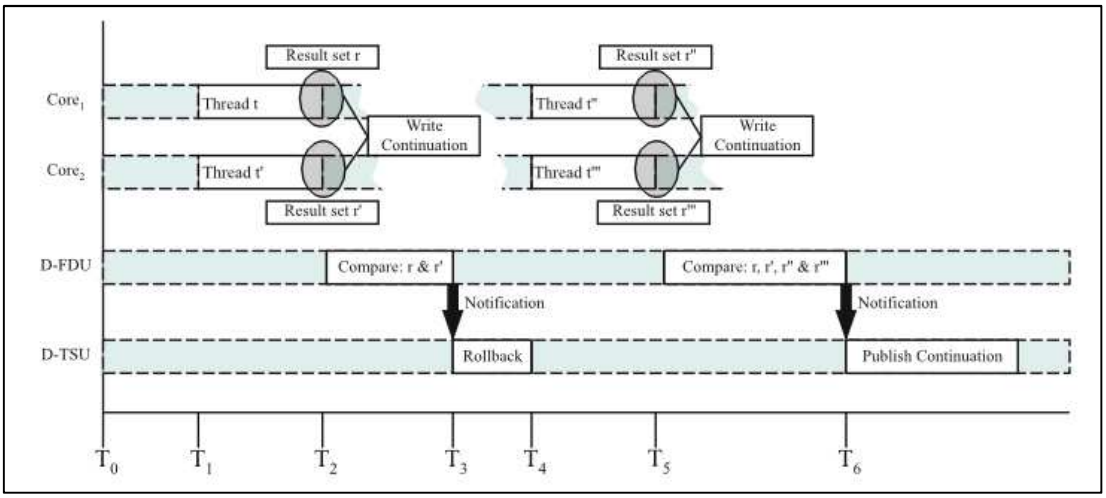

Figure 7: Thread re-execution example

If the D-FDU assumes a permanent or intermittent fault due to many re-execution attempts or information from the L-FDU, it must exclude the faulty core from further workload. This is done by providing the D-TSU with the information, which core is faulty. Consequently, the D-TSU re-schedules all threads of the faulty core on another reliable core. In order to do that, the D-TSU traverses its Thread-to-Core-List and searches for corresponding entries scheduled on the faulty core. If the D-TSU finds an entry that is associated with the faulty core, it re-assigns the entry to a reliable core.

Subsequently, the L-TSU has to allocate a thread frame for the newly assigned thread and fill the frame with the data from the D-TSU.

We will pursue the described fault recovery mechanism in project year 3. In particular, we will incorporate DF2-Threads with Thread Local Storage and DF2-Threads with Transactions.

3 Clustering of Cores within a 2D Mesh-based NoC

The high level Teraflux architecture (see Deliverables D6.1, D6.2, and Section 2) distinguishes a node-external NoC from a node-internal local interconnect. Within this section, we assume that the "node internal interconnect" is not fixed by hardware, but part of a uniform mesh-based NoC. We introduce the notion of a Node Manager that comprises a D-FDU and an associated D-TSU within a "node" or "cluster". We distinguish a hardware- and a software-based Node Manager implementation and evaluate the trade-off of a Node Manager's static placement within a mesh-based NoC. In coherence with DoW Task 5.2 description, we call this "clustering of cores" around a Node Manager (respectively D-FDU) to build up a "node". Section 3.3 describes a re-clustering of cores to D-FDUs in cases of detection of faulty components (interconnections and cores) based on a software implementation of the Node Manager.

We assume that one D-FDU will not be able to monitor all cores of a 1000 core chip. Therefore, we propose to have a certain number of D-FDUs cooperating with each other in order to distribute the fault detection workload. This results in a communication pattern between the D-FDUs and raises the question how the monitored cores are grouped to logical clusters. A logical cluster may only be seen by its affiliated D-FDU and D-TSU and can be built and re-built dynamically.

We paid particular attention on the parameter selection for clustering. One of these crucial parameters is the size of a cluster. The cluster size describes how many cores are included in the cluster. We focused on this parameter, because the cluster size has a direct impact on the chip performance and the accuracy of our fault detection approach.

Additionally we focused on the traffic generated by heartbeat messages. We assume that the D-FDU is designed to be used on a many-core processor with a mesh-based shared interconnection network. For this purpose and having the clustering in mind, it is necessary to investigate the impact of the fault detection messages on the processor's interconnection network. Different cluster sizes also have different implications for the interconnection network and especially the router connected to a D-FDU is a bottleneck. In order to prevent bottlenecks in the interconnect heartbeat messages must be coordinated. This raises the need to

1. balance the load of heavily utilized interconnects and
2. a sending pattern ensuring that at most one heartbeat message arrives at this router per network cycle. This is necessary because we expect a steady monitoring stream from all cores sending heartbeat messages to the D-FDU. This may lead congestions if more than one message arrives at the D-FDU in a network cycle.

This section addresses the impact of fault detection monitoring overhead on a 2D-mesh network and its implication on the application communication. We investigated XY and Staircase Routing and propose a combination of both algorithms to reduce delays and jitters of application messaging. We also implemented a *heartbeat message sending pattern* each core has to obey. After we investigated the size parameter for clustering, we describe our proposed clustering techniques.

3.1 Network Considerations

In our Deliverables D5.1and D5.2 we have proposed techniques that are able to establish a reliable system out of unreliable components by simultaneously exploiting the architectural opportunities of future many-core processors. Our D-FDU focuses on the monitoring of cores and analyses the information gathered from them. If a faulty core is detected the D-FDU plans and executes a reaction in order to recover from this fault. In order to gain the needed information from the monitored cores, the D-FDU awaits heartbeat messages periodically sent from the monitored cores. This results in a communication pattern of fault detection messages towards the D-FDU, which is normally located in the centre of the monitored cores (see Figure 8 and Figure 9). This centralized communication overhead influences the communication capabilities directly, as the heartbeat messages may interfere with other messages within the communication network. Since the D-FDU expects the heartbeat message of a particular core at a certain point in time we can exploit the deterministic semantic of XY Routing [3, 4] and its derivative Staircase Routing [15].

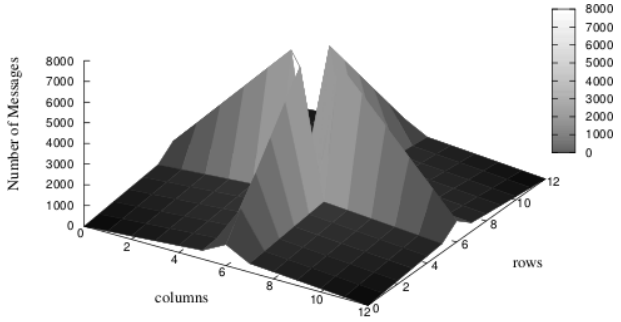

Figure 8: Network load distribution induced by heartbeat messages using XY routing

3.1.1 Baseline Network on Chip

A possible instance of the TERAFLUX Architecture Template (cf. D6.2 and Figure 1) can use a "NoC switch" as "local network". Therefore in the following discussion, we assumed such scenario in order to investigate clustering and monitoring for NoC. Our baseline Network on Chip (NoC) encompasses a standard 2D-mesh topology with point-to-point flit-based wormhole routers. Each router in the communication network has five output ports and five input ports (which are north, east, south, west, and local). The local port is connected to a processing element. The remaining ports are connected to its neighbouring router, if any. Each input port has one input buffer, where a router stores incoming flits.

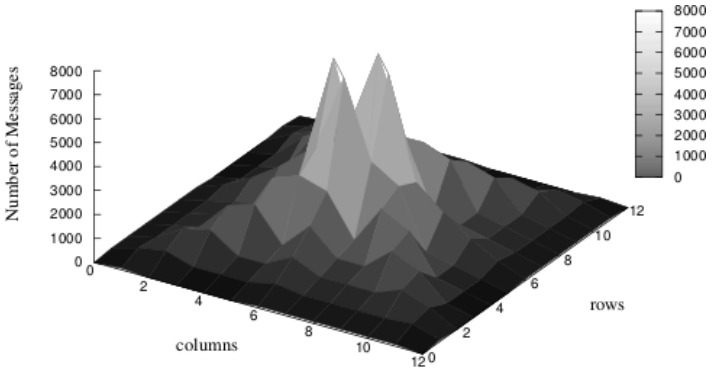

Figure 9: Network load distribution induced by heartbeat messages using Staircase routing

On flit level one packet is divided into several flits composed of one header flit, several body flits and one tail flit. The number of body flits depends on the packet size. We assume that the size of a heartbeat message may vary due to the amount of information that is transmitted from a monitored core. A pure heartbeat message without any further core state information, however, may be composed only of a header flit enclosing a *still alive* bit. If this bit is set the router will not expect a tail flit for this message and immediately release the output port for subsequent messages.

3.1.2 Routing Considerations

Since the routers in Network on Chip architectures require a concise use of the spare chip size, we decided to consider "lean" routing algorithms. In this manner, we determined XY Routing and Staircase Routing as appropriate routing algorithms, since they are well known lightweight routing algorithms. Additionally, these algorithms have the advantage that they are deterministic in case of a fault free interconnection network. However, the limitations of these static algorithms are faulty elements in the interconnection network. In the following, we describe these algorithms and their limitations in more detail. We close this subsection with an explanation on how to solve blocking problems induced by faulty elements.

3.1.2.1 The Staircase Routing Algorithm

As a derivate of the XY Routing, the Staircase Routing is a dimension ordered routing algorithm with minimal path routing. The major difference between both algorithms lies in the way they route a packet through an interconnection network. The XY Routing algorithm routes a packet in the x-dimension until the packet reaches the destination column and proceed the routing in y-dimension. The Staircase Routing algorithm alternates the dimension after each hop the packet takes. The alternating routing results in a staircase-shape looking path which is depicted in Figure 10 ($S1$ to FDU). If the packet reaches its destination row or column, the routing will then proceed without alternating the dimension ($S2$ to FDU). We will show in Section 3.2.4 that a combination of both algorithms has a relaxing effect on the network traffic.

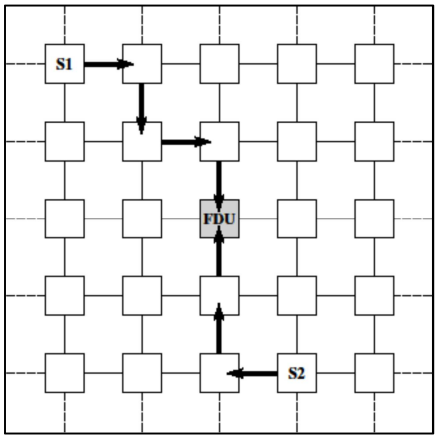

Figure 10: Staircase Routing: A Dimension ordered routing algorithm, which alternates its routing dimension after each hop. Packets from S1 and S2 are transmitted to the FDU.

Simulations running with the static XY Routing and Staircase Routing showed that it is not trivial to determine better static routing algorithm. Figure 8 and Figure 9 show the results for a simulation of

heartbeat messages using XY Routing and Staircase Routing for 100,000 cycles. The XY Routing strategy results in a clear division of the group of cores into four quadrants and represents the first significant effect of the dimension ordered routing. On the central axes of the group, the load on the connections steadily increases, while the load of the “enclosed surfaces" between these axes increases minimally. A problem that may arise in the different loads is the increased risk of congestion at these central axes. Application based messages will be delayed or jittered more often, when they try to traverse the network over these links. However, messages not traversing these links may not suffer from delays and jitter at all.

The Staircase Routing is used to reduce the delay and the jitter induced by the heartbeat messages using XY Routing. In Figure 9 we show the same simulation for heartbeat messages, using Staircase Routing. The network load is, compared to XY Routing, more widely distributed. It can be seen that more different paths to the D-FDU are used for the transmission. One can easily see the diagonal axes relieve the central axes of the group and the messages meet only in the middle of the group (near the D-FDU). In contrast to XY Routing the Staircase routing in Figure 9 shows that more messages may suffer from delays and jitter. However, in most cases the effective delay will be smaller.

3.1.3 Prioritization of Packet Switching

To keep the deterministic characteristic for fault detection messages even for a fully occupied network with application messages, we propose a priority based arbitration for packet switching. The high priority class is dedicated for fault detection messages, which are preferably processed by the routers. The low priority class is used for application messages and is stalled whenever a fault detection messages is available. More elaborated prioritization techniques, however, are already discussed in [2, 6] and will not be further discussed in this work.

3.1.4 Case of Faulty Elements

We assume an interconnection network element as faulty, when the element suffers from permanent or intermittent faults. Although we do not specifically consider the faultiness of routers and network interfaces, we do this without any loss of generality since a faulty component can be modelled by a set of faulty links. To detect a faulty link we follow the approach of Grecu et al. [7] using self-checking mechanisms. As this is an already widely researched area and the self-checking mechanisms are well known, we do not discuss those mechanisms further.

In cases of faults within the interconnection network the static routing algorithms XY and Staircase Routing fail to solve this situation adequately. Figure 11 illustrates the blocking situation for packets sent from Core $S1$ and $S2$. $S1$ sends a heartbeat message to the D-FDU, while $S2$ sends an application message to $S3$. Depending on the router implementation the blocked packets could either create a deadlock or they will be dropped by the holding router after a certain period of time.

Dropping the packet releases one slot in the input buffer of the router and other messages in this slot may proceed. $S2$, however, may await a response from $S3$ regarding its dropped packet. This could lead to a thread execution deadlock (if no timer signals a timeout). Also, since the D-FDU awaits heartbeat messages in order to feed the internal analyse phase, the D-FDU may mispredict that a core $S1$ is broken. This could also count for every core that takes the same path (or parts of it) to send their heartbeat messages.

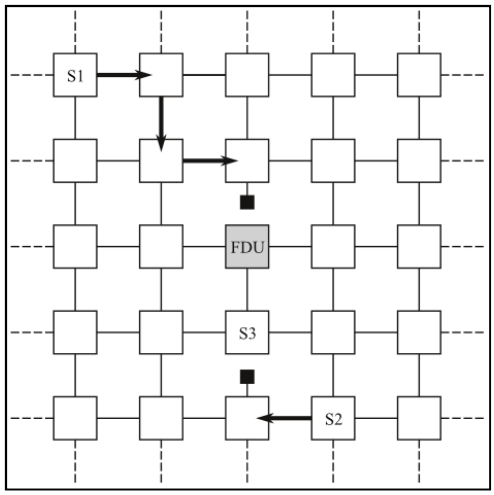

Figure 11: An interconnection network with faulty elements.

To overcome this issue we decided to extend the routing algorithms XY and Staircase to tolerate faulty elements along a packet's path by taking an alternative route to the destination element. However, choosing an alternative route to the destination may violate the turn restrictions of static routing algorithms. For that reason, we have extended the routing algorithms by the turn-model west-first. This model restricts less turns than XY and Staircase do, while supporting deadlock freedom [4]. In a fault free interconnection network, the extension is never triggered and the routing algorithms behave just as static routing algorithms. In the case of faulty elements along a packet's path the extension will be triggered. Figure 12 illustrates this behaviour for the two examples mentioned above. Both messages were originally blocked, but the extensions of the static routing algorithms take now another route for the packet destination.

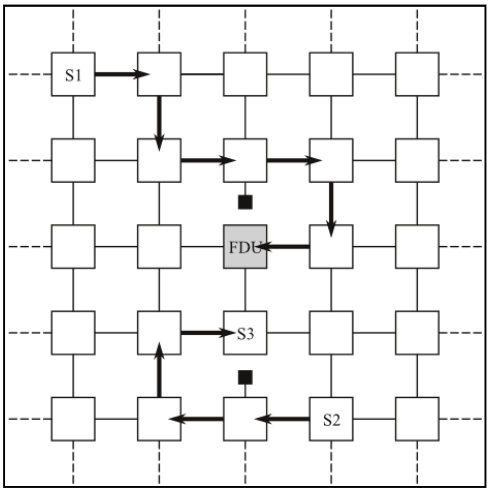

Figure 12: Partially adaptive routing extension for Staircase Routing

The path length of a packet may increase as a result of the way around the blocking element. However, there are situations where no additional delay is created.

3.2 Heartbeat Message Impact

From the D-FDU point of view receiving heartbeat messages in every network cycle is the ideal situation for the maximum fault detection accuracy, unfortunately at the expense of the application message throughput of the communication network. We assume optimal fault detection accuracy when we transmit a maximum of information from the core to the D-FDU. But a fully utilized network leads to message congestion particularly affecting the application messages in terms of delay and jitter. Therefore, it is necessary to investigate the message overhead the D-FDU generates in order to balance fault detection accuracy and its impact on the application communication.

In this subsection we describe how the heartbeat messages influences the application messages and how we ensure that the heartbeat messages do not interfere with each other.

3.2.1 Density of Heartbeat Messages

Unfortunately, it is not enough just to maximize the fault detection accuracy, which would result either in massively sent heartbeat messages or in a disproportional high number of D-FDUs on the chip. The number of D-FDUs on a chip is a trade-off between fault detection accuracy and area overhead by preventing cores to be used for application execution. We have determined that a D-FDU amount occupying $\approx 4\%$ of the available cores (which is the area overhead) is powerful enough to run D-FDUs on a 1000 core processor. Of course, smaller overhead is desirable, but comes with the before mentioned reduced fault detection accuracy. This accuracy suffers from bigger cluster sizes, since we are using the heartbeat timing pattern, which regulates the amount of heartbeat messages send from a core during a certain interval of time. We explain the problem with the fault detection accuracy and the timing pattern in the following subsection in more detail.

An area overhead of $\approx 4\%$ means each group of monitored cores is composed of about 24 cores. The upper bound for the heartbeat message density involves a trade-off between the processor's performance capabilities and the accuracy of the information received by the D-FDU. A high upper bound means good results regarding the processor performance, but reduces also the accuracy of fault detection due to long waiting intervals. The waiting intervals in turn have a direct influence on the interconnection network, meaning a shorter waiting interval for cores induces a higher load on the network. We calculate the message overhead and give an estimation how this overhead leads to delays and jitter for application traffic.

Table 1: Area overhead for different cluster sizes

Cluster size	3x3	5x5	7x7	9x9	11x11	13x13
Area overhead	12.5%	4.2%	2.1%	1.3%	0.8%	0.6%

3.2.2 Heartbeat Timing Pattern

The D-FDU independently monitors an entire group of cores, detects faulty elements, and initiates actions for problem treatment. In a large NoC several D-FDUs monitor each other in order to detect

malfunctioning D-FDUs or failed nodes. One of the key metrics for the D-FDU is the latency of the heartbeat messages. Since all heartbeat messages are sent periodically and routed with highest priority the arrival time of a certain message is deterministic. This determinism is used by the D-FDU to check the accessibility of a core (including the core itself and the interconnect integrity). The loss of determinism is equivalent to the loss of the D-FDU's accuracy and therefore the minimal latency has to be ensured.

To avoid congestion induced by heartbeat messages interleaving each other, we developed a heartbeat message timing pattern, each monitored core has to obey. In order to establish this timing pattern the D-FDU sends configuration messages to its affiliated cores. Such a configuration message contains the precise message sending timing interval for a particular core. To ensure that these timing values are not corrupted by delayed message delivery, we include the configuration messages in the high priority class of the arbitration for packet switching.

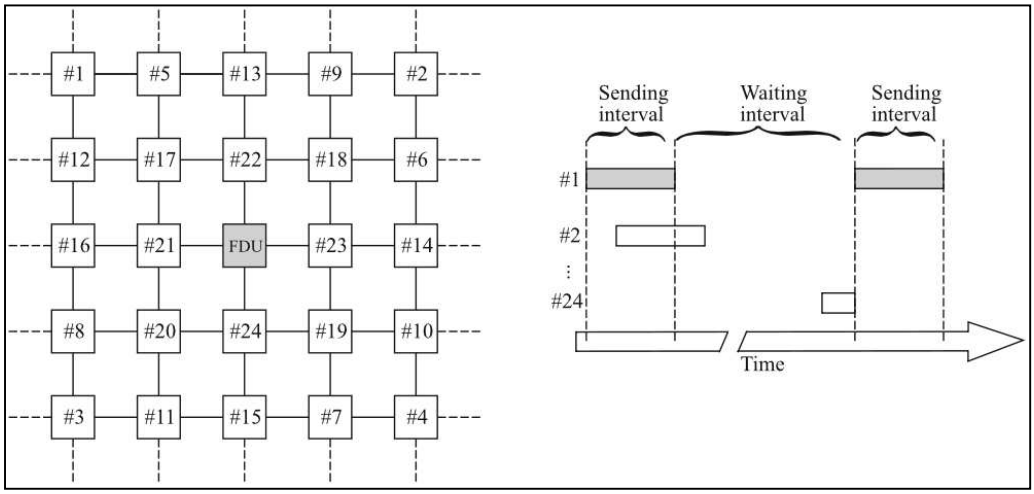

Figure 13: Message timing pattern to avoid interleaving heartbeat messages.

Combined with the high priority of a message, this pattern ensures, that no heartbeat message will interleave with another one as long as there are no faulty elements. As an example Figure 13 shows the timing pattern for a 5x5 sized group of cores. The core in the centre of the group is the D-FDU, where all heartbeat messages are sent to. The $\#n$ in each core defines when a core is allowed to send a heartbeat message. In this example, the sending pattern is based on the Manhattan distance between the cores and the D-FDU. First, the cores with a Manhattan distance of 4 (such as $\#1, \ldots, \#4$) are allowed to send their heartbeat messages consecutively. The cores with the next lower distance follow afterward. On each step from one Manhattan distance to the next lower one, it is necessary to apply a waiting phase until the next core sends its message. Otherwise, the last message of a core with the Manhattan distance d will interleave with the first message of a core with the Manhattan distance $d - 1$. The first core in that group (#1) will send its next heartbeat, when the last core (#24) has sent its Heartbeat. Following this procedure, we can guarantee a minimal latency for each heartbeat message and therefore the necessary determinism for the accurate D-FDU fault detection technique.

As stated before, an increasing cluster size decreases the fault detection accuracy, because the increasing number of cores per cluster also enlarges the messages timing interval. That means in turn,

that the cores have a longer waiting phase until they are allowed to send a heartbeat message, which results in a

1. larger amount of collected information by the L-FDU or
2. (if the L-FDU collects the information just before the heartbeat message is send) that some of the information may be already overwritten by the core internal mechanisms.

In any case we assume that the accuracy of the fault detection suffers from long waiting intervals. For that reason, it is important to determine not only the minimal but also the maximal size of a cluster.

3.2.3 Evaluation Methodology

In the following subsection, we describe the environment of our investigation, the implication of heartbeat messages, and how these implications are related to routing decisions. The subsection closes with a method for calculating the overhead of our heartbeat message based fault detection.

We defined the Accumulated Average Delay ($AAD(d)$) as the key metric calculating the message overhead induced by the heartbeat messages. The AAD determines the delay cycles an application message suffers from while traversing through the network. We split the calculation into three steps:

1. Determining the bandwidth costs for heartbeat messages for each router interconnect
2. Determining the delay for application messages induced by the heartbeat messages for each possible communication path
3. Accumulating each delay value for a certain path length and calculating the arithmetic mean.

$C(n,p)$ is the function returning the bandwidth costs for the transmission (Step 1) from node n to the next node along the path p by following the routing rules. Costs are rising, when more heartbeat messages are sent over a certain interconnection (see Figure 8). The values of $C(n,p)$ vary between 0 and 1, while 0 means no heartbeat message will pass the interconnection and 1 means in every network cycle a heartbeat message will pass the interconnection (100% utilization). Since the heartbeat messages are prioritized the bandwidth costs will be equal to the expected delay an application message will encounter on this interconnection.

The function $\mathrm{W}(p)$ sums up the delays expected for a given communication path p (Step 2):

$$\mathrm{W}(p) = \sum_{\forall n \in p} C(n,p)$$

with n as the current node on path p. Since we assume that all application messages are routed via XY Routing, all paths were generated by using this routing strategy. The function $\mathrm{ADD}(d)$ sums up all accumulated delays for paths with length d and divides the sum by the number of the possible paths (Step 3):

$$\mathrm{ADD}(d) = \frac{1}{|\mathrm{P_d}|} \sum_{\forall \mathrm{p} \in \mathrm{P}} \mathrm{W}(p)$$

with P_d as the set of paths with the length d. XY and Staircase Routing use shortest paths and prevent a circular formation of the messages, d can simply be calculated by determining the number of hops from the source node to the destination node ($\mathrm{d} = |p|$), or Manhattan Distance). Calculating the AAD for all possible paths in a network hosting a D-FDU and its affiliated cores shows the induced overhead and the influence on the delays of application messages.

3.2.4 Application Message Delay Calculation

We applied our metrics to XY Routing and Staircase Routing. The bandwidth cost function $C(n,p)$ for heartbeat messages were separately calculated for XY and Staircase Routing, resulting in two cost sets. We used both cost sets and applied the $AAD(d)$ function to each set. Figure 14 and Figure 15 show the results for different sized groups of cores with respect to the calculated AAD value. The values are grouped by the number of monitored cores. The bars describe the delay an application message suffers from along its communication path. The light grey bars of both Figure 14 and Figure 15 shows the average minimal delay that a packet has to expect for a given group size. Figure 15 represents the combination of XY Routing (for application messages) and Staircase routing (for heartbeat messages) and shows that the average minimum delay decreases with the increasing number of cores within a group. The reason for this decreasing delay is the broader traffic distribution gained from using the Staircase Routing for heartbeat messages. The dark grey bar in the middle of each group represents the overall average delays a packet has to expect. This value remains almost constant, because the timing pattern is always adapted to the group sizes, respectively. Using Staircase Routing for heartbeat messages also lowers the average maximum delay for application messages. Comparing the average maximum delay of both algorithms, we can show that the delays induced by Staircase Routing grow less strongly with the group sizes than XY Routing would. Especially for larger groups of monitored cores, the combination of both routing algorithms results in smaller average maximal delays.

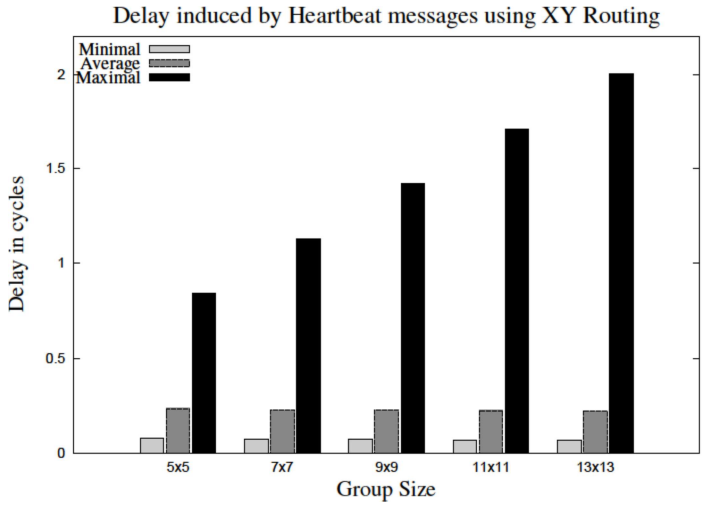

Figure 14: AAD using XY routing algorithm for both heartbeat and application messages.

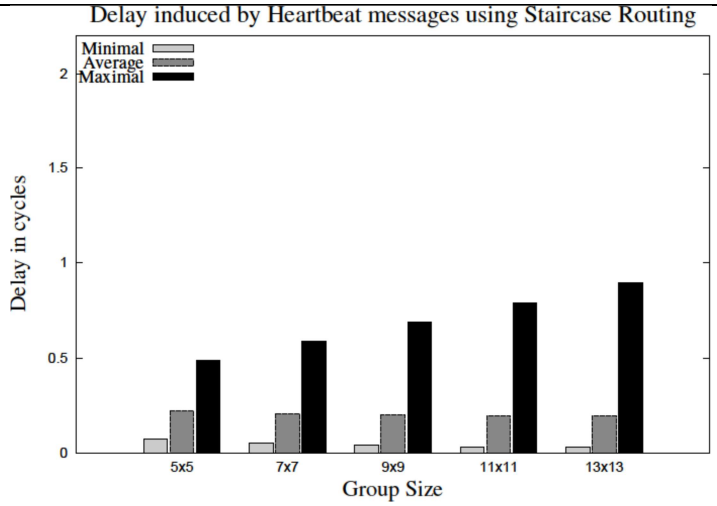

Figure 15: AAD using Staircase Routing for heartbeat messages combined with XY routing for application messages.

3.2.5 Adequate Cluster Sizes

As a result of the density of heartbeat messages (see Section 3.2.1) and the calculated delays for application messages (see Section 3.2.4) the minimal cluster size is constrained by the maximum performance degradation we are willing to pay for fault tolerance. We determined that a degradation of $\approx 4\%$ is a sensible price, since fault tolerance comes not for free. Expressed in numbers, this means we propose a minimum size of 5x5 cores per cluster.

The maximum number of cores depends on the routing algorithm used to route the heartbeat messages. In the case of XY Routing, we see in Figure 14 that the maximum delay crosses the 1 cycle mark already at a cluster size of 7x7. In order to keep the delay to a minimum, we propose to the set the maximum cluster size to 5x5 while using XY Routing for both message types – heartbeat messages and application messages. This might result in a more complex re-clustering, because the clustering algorithm cannot assign several cores to a neighbouring cluster that is already composed of the maximum number of cores. For a worst case scenario, this could lead to a system wide re-clustering.

If the heartbeat messages are routed by Staircase Routing, we can see that we have a little more freedom of choice, which cluster size is sensible. The average delay is even with a cluster size of 13x13 slightly higher compared to the 5x5 version of XY Routing. This gives the clustering algorithm more flexibility to assign cores to a neighbouring cluster by costs of slightly higher average delays. Nevertheless, we propose also for the combined routing a cluster size of 5x5, since we keep the number of cycles – for both; additional average delay for application messages and waiting phases for cores to send their heartbeat messages – low.

3.3 Clustering Mechanisms

As stated in the introduction of this section we propose to arrange the cores into (logical) clusters for fault tolerance purposes. A cluster is composed of a number of cores monitored by the D-FDU. The D-FDU itself is also part of each cluster and serves beside the fault detection as the host for the

clustering algorithm, which is responsible for creating, altering, and breaking up clusters. In this Section, we assume that the D-FDU is implemented in software and executed on a core of a cluster.

For the clustering we use the configuration messages sent from the D-FDU to its affiliated cores. These configuration messages are already used to configure the heartbeat message sending pattern mechanism. This configuration message contains a specific command flag and several command parameters that update the cores heartbeat behaviour. In the case of building a cluster the D-FDU sends a configuration message with the following content to each cluster affiliated core:

- Instruction command (which similar to the configuration command of the heartbeat message sending pattern)
- The number of cycles representing the waiting interval
- The address to the monitoring D-FDU-core

Defined cluster borders are static in the start-up phase. From the location of the D-FDU and the size of its affiliated cluster we can derive all cores a D-FDU has to configure.

In the following subsection we describe the mechanisms we use to configure and establish static clustering used in the start-up phase of the processor chip. In the second part of this subsection we describe the dynamic re-clustering approach, which we developed to react on faulty elements during runtime.

3.3.1 Initial FDU Placement and Entering into Service

We propose the derived the functionality for the initial D-FDU placement from the BIOS of IBM PC compatible computers. Those BIOSs provide, among others, basic CPU configuration mechanisms such as clock settings, memory timings, and safety settings (e.g. temperature shutdown). We further propose to extend the BIOS mechanism to the effect that it configures one specific core of the chip as a **Root D-FDU-Core**. This core loads its code from a specific memory location and starts execution. This code spawns additional D-FDUs at specific locations on the whole chip. In order to do that the Root D-FDU-Core sends initialization messages to the designated cores and these cores load the standard D-FDU-Code from the memory.

As an alternative approach, we can assume that the D-FDU initialization messages are sent to each node's D-TSU, which then spawns a D-FDU-Thread on the designated core.

This approach is very similar to bootstrapping techniques in today's personal computer. The Root FDU-Core is temporally in charge of all subsequent D-FDUs in order to spawn and configure them.

3.3.2 Initial Clustering

As stated in the introduction of this section we do not assume any faulty components on the chip during the system's start-up phase. However, if there are faulty elements on the chip the re-clustering algorithm will solve bottlenecks for heartbeat messages.

At the beginning of clustering, we logically decompose the entire chip in equal parts, which forms the actual clusters. The cluster size of 5x5 is derived from the previously conducted investigation. Since

the detailed TERAFLUX architecture is still evolving, we assume a 32x32 mesh-based arrangement of the cores. This arrangement combined with a cluster size of 5x5 leads to clusters with irregular size. Some clusters will be slightly bigger than 5x5. Following our previously mentioned assumption we have then

- 25 clusters with the size of 5x5,
- six cluster with the size of 7x5, and
- one of the size 7x7.

Our investigations showed that a cluster size of 7x7 is also feasible. If the layout of the die changes (e.g. if the chip has a rectangular topology and not a squared), we can easily adapt this clustering to gain a feasible cluster arrangement.

3.3.3 Re-Clustering of Cores

In the presence of faulty cores or network elements the optimal placement of the D-FDU is no longer a trivial task. The centre of a cluster might not be the ideal position for a D-FDU anymore. For this case we propose to determine the position of the D-FDU with a task-placement algorithm originally developed to place communicating tasks on a Network on Chip. The Connectivity-Sensitive algorithm [13] features a low complexity and tries to minimize the communication overhead by keeping intensely communicating tasks close to each other.

3.3.3.1 Fault Model for Faulty NoC-Elements

We assume the following faults on component level for every component in a cluster and one fault in a single component at a time if not stated otherwise. Since nodes, routers, and D-FDUs are connected by a regular 2D mesh-based interconnection network, the fault model includes the possibility of multiple core faults within a cluster and different clusters as well as multiple D-FDU faults leading to a graceful degradation of a cluster and thus reduced fault detection capabilities.

Furthermore, we assume faults in the following components:

Link: Complete permanent faults and permanent bridging faults across multiple wires, furthermore transient faults.

Core: Complete permanent faults and (within cores) transient faults.

FDUs: Complete permanent (D- and L-) FDU faults and transient faults.

TSU: Complete (D- and L-)TSU faults and transient faults.

Router: Complete permanent Faults.

3.3.3.2 Connectivity-Sensitive Algorithm

The original Connectivity-Sensitive algorithm =[13]=[13] maps an arbitrary task-graph to a core-graph. A task-graph depicts single components of the executing application as nodes which are linked by their communication paths; a core-graph describes the underlying interconnection topology by

representing each core as a node and the interconnecting links as the edges of the graph. Both graph types can have weighted edges to accurately model different communication intensities of tasks and differing bandwidth in the interconnection network.

Rate Tasks *T(E,V)* (by number of connections and combined weight)

Rate Cores *C(E,V)* (by number of working connections)

Choose task *t* with highest rating

Place task *t* on core *c* with highest rating

Put neighbour t_N of *t* in placement list *P* as a tuple (t_N, c)

for each element *(t', c')* in *P* **do**

    if *t'* is not placed yet **then**

        Place *t'* on c_{new} close to *c'*

        Put neighbours of *t'* into *P* as a tuple (t_N, c_{new})

    end if

end for

Figure 16: Connectivity-Sensitive Algorithm

In the first step of the algorithm each core is rated by the out-degree of working connections to working cores. If a connection or core is permanently faulty it cannot be used and thus does not count towards the rating. Similarly the tasks are rated by the number of connections to other tasks and the combined weight of these connections. Initially one of the tasks with the highest rating is chosen and placed on one of the cores with the highest rating. After that all neighbours (t_N) of the placed task are put into the placement list *P* which contains tuples of all the tasks that still have to be placed and the location of the neighbour that put them into the list. Now each element of this placement list *P* is examined. The element *(t', c')* is removed from the list and if it still has to be placed, the algorithm places it on a core as close to c' as possible. In the ideal case this would be a neighbour of *c'*, but if that is not possible, as all neighbours are already taken or failed, the distance is increased. After t' is placed each neighbour task of t' is also put into the placement list *P* as a tuple of the task t_N and the currently used core c_{new}. When *P* is empty the algorithm terminates.

To illustrate the algorithm, Figure 17depicts an example mapping. The algorithm selects task “A” as a first placement candidate and places it on the centre core. As a result of this, tasks “B” through “D” are added to the placement list. After that “B” is placed near to “A” and causes “E” to be added to the placement list. Now “C” and “D” are also placed near “A”, but they are not able to expand the

placement list, as all tasks are already processed or in the list. “E” is finally positioned near “B” which inserted “E” to the list.

The algorithm can handle permanently faulty elements on the chip if a fault tolerant routing strategy is used. Failed links and links to failed routers are not counted in the summation of the out-degree and failed cores are removed from the core-graph. Their links and routers however can still be used. In case of the situation that a whole area is disconnected from the rest of the cluster, the affected area is not used for placement.

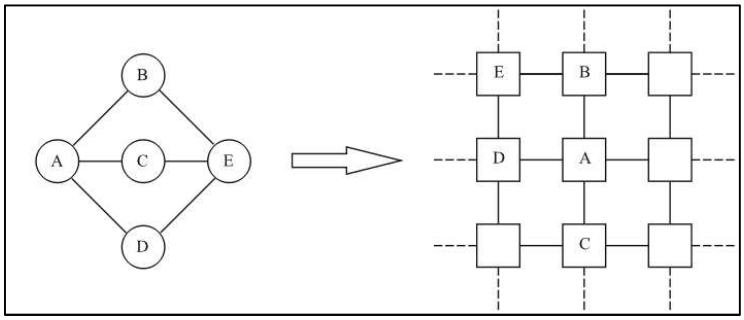

Figure 17: An example of a mapping produced by the Connectivity Sensitive algorithm

For the special case of D-FDU placement the algorithm can be used as it is. The task-graph is generated as a star topology with the D-FDU in the centre (see Figure 18) and each monitored core directly connected to it with a link of equal weight. The number of monitored cores is determined by the remaining working and reachable cores in the current node. These nodes are determined with the clustering method for chips without faults. If it becomes apparent that a node is no longer viable for computations because the number of working and reachable monitored cores is very low, a whole cluster is deemed lost. In that case it is possible to break up the logical cluster and assign working parts of it to the neighbouring clusters.

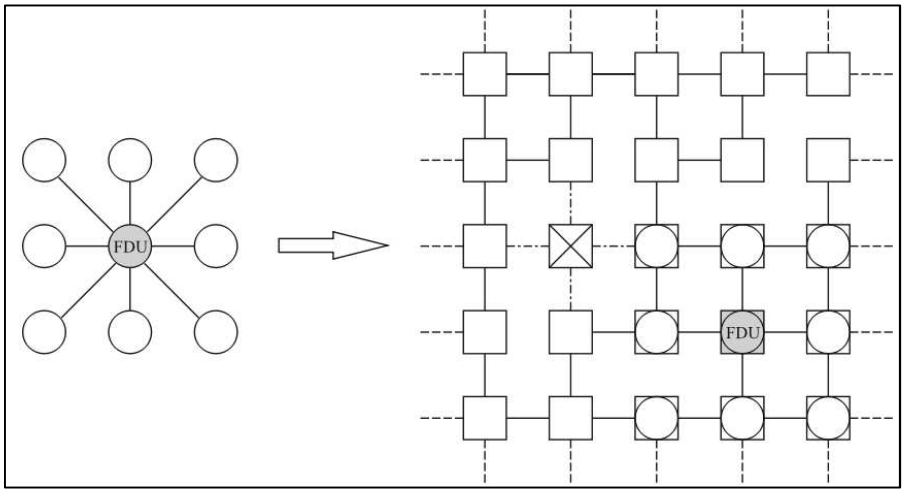

Figure 18: Example task-graph (left) mapping on a logical cluster with faulty interconnection elements.

Figure 18 depicts an example mapping of a task-graph composed of monitored cores (white cycles) connected to the D-FDU (grey cycle). In this example, we omit the remaining monitored cores, for

reasons of clarity. The resulting mapping is shown on the right side, where the D-FDU is the grey square and the monitored cores are represented as white squares with a cycle in them. As mentioned before the D-FDU was placed to a core with best fitting out-degree. Faulty elements are represented as missing links between the monitored cores or by a cross placed in a square as faulty router.

4 Inter-Cluster Fault Detection Mechanisms, Grouping Strategies, and Device Controller Monitoring

As stated in Section 3, it is supposed that one fault occurs in one component at a time. Note, that this includes the possibility of multiple faults in one but not in different components.

Hence, we must not only incorporate faults on intra-node level but also on inter-node level. This means beside core and link faults within a node, D-FDUs and links between nodes can suffer from transient, intermittent, and permanent faults, too. Therefore, we have already described in Deliverable D5.1 that D-FDUs monitor not only **intra-core** elements (D-FDU monitors it affiliated cores) but also **inter-core** elements such as other D-FDUs in other nodes (see Figure 20). This inter-node monitoring is motivated by two observations.

1. In the case of a D-FDU fault all results and events generated by the components within the respective node must be considered incorrect, since the node's D-FDU can no longer ensure reliability and fault detection.

2. The information about the reliability state of a node and its affiliated cores must be distributed to other D-FDUs to ensure a reliable inter-node management.

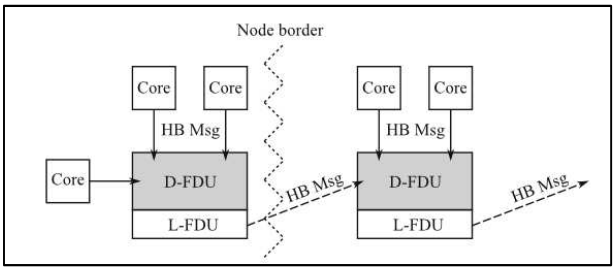

Figure 19: Schematic view on the D-FDU to D-FDU monitoring. The L-FDU sends heartbeat messages to a monitoring D-FDU.

Cases in which the D-FDU may be faulty can lead to serious problems such as committing results from a corrupted thread execution or erroneously labelling a core as faulty even though the core is working correct. In order to prevent this behaviour we incorporate – in contrast to the $(1, n)$ relationship of the intra-node monitoring – an (n, m)- inter-node monitoring relationship.

4.1 Inter-Cluster Monitoring Mechanisms

We consider a D-FDU as software, running on a dedicated core. This dedicated core can be either a specialized embedded controller or one of the general-purpose cores within a node, which is not considered for dataflow thread execution. This means that the inter-node monitoring exploits techniques already described for intra-node monitoring in Deliverable D5.1.

In particular, each D-FDU core has also attached an L-FDU sending both periodic heartbeat messages and event messages to the observing D-FDUs in other nodes. Furthermore, every observed D-FDU exploits existing state-of-the-art Machine Check Architecture techniques and uses the control flow

checker, described in Section 2.2.1. The code running on each D-FDU core must be therefore instrumented for the control flow checker, too.

Additionally to the monitoring of the observed D-FDU core's state, the monitored D-FDU tells the monitoring D-FDUs the state of all monitored intra-node elements of the respective node. The state of a single component within a node, e.g. a core, exchanged between nodes is defined as $\pi \in [0,1]$.

Monitored D-FDUs send the states of the node's components in form of a vector $p = \begin{pmatrix} \pi_1 \\ \cdots \\ \pi_n \end{pmatrix}$ to the observing D-FDU.

4.2 Grouping Strategies for Inter-Cluster Monitoring

We exploit different strategies for grouping the monitoring relations between D-FDUs. The amount of D-FDUs another D-FDU is able to monitor is restricted by two things. First, the additional overhead for gathering the information, which is basically the induced network overhead and second the overhead in memory and execution time to analyse the gathered information. The network overhead explicitly incorporates the distance between the D-FDUs.

The easiest solution for D-FDU grouping is that each D-FDU monitors one neighbouring D-FDU and each D-FDU is monitored by a neighbouring D-FDU. This Ring grouping is depicted in Figure 20.

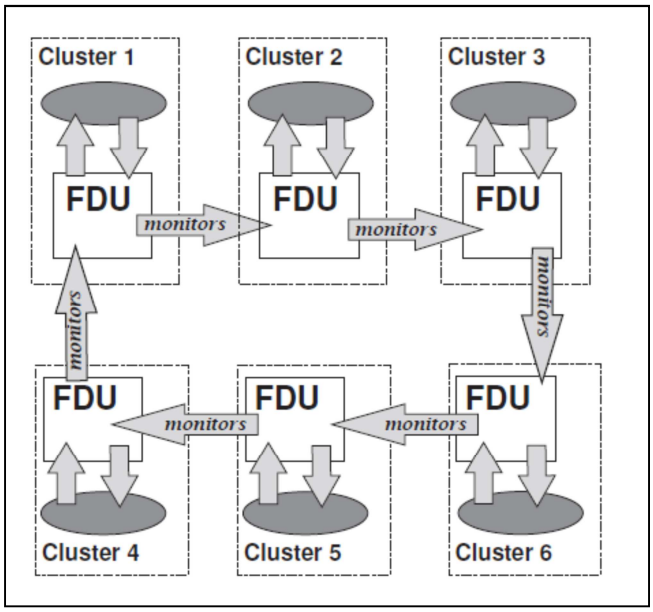

Figure 20: Ring monitoring of D-FDUs

As an extension to the very naive grouping strategy above, we propose a second technique. The motivation of this approach is to minimize the possibility of false-positives. If one D-FDU monitors another one, there are situations where we are not able to distinguish which D-FDU is actually faulty. Additionally, as mentioned, a faulty D-FDU can affect a whole cluster of monitored cores, such as shutting down cores, scaling down cores, etc.

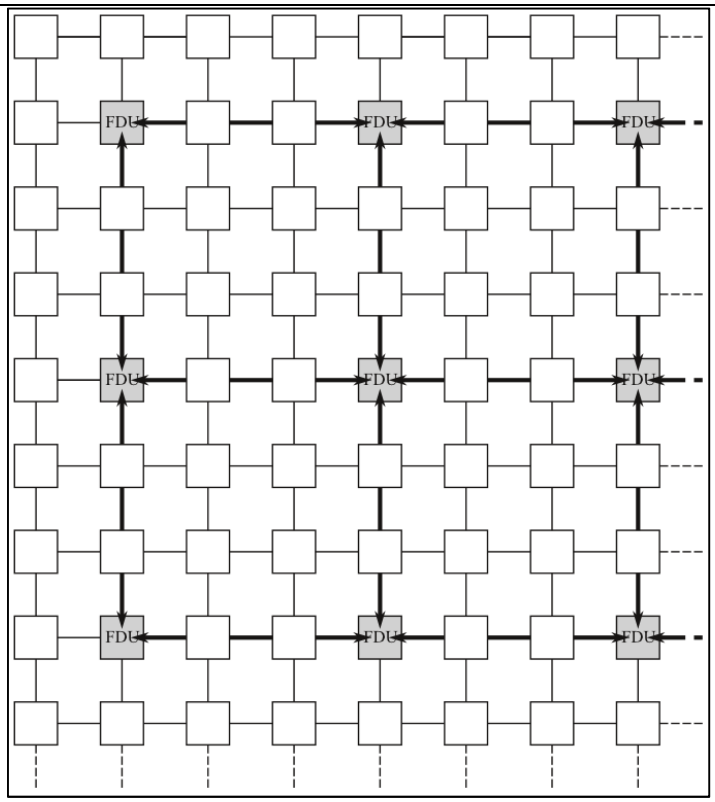

Figure 21: An alternative D-FDU grouping strategy. D-FDUs have at minimum two other D-FDUs monitoring it.

To tackle this problem, we propose to increase the redundancy of the D-FDU monitoring. We propose that at minimum two D-FDUs monitor each other. The amount of D-FDU to D-FDU monitoring can differ depending on the location of a D-FDU; in the corners of a 2D-mesh a D-FDU is monitored by its direct neighbours and at the edges a D-FDU has three other D-FDUs monitoring it. Hence, the D-FDUs near the centre of the processor chip may have at most four other D-FDUs monitoring them. We will investigate both described variants, in particular incorporating the D-FDU's MAPE cycle execution time, in the TERAFLUX integration platform in project year 3.

4.3 I/O and Memory Device Controller Monitoring

I/O and Memory Controllers are located at the borders of the NoC and connected with an external interconnect to peripheral devices or memory units, respectively (see Figure 23). Both, I/O and memory controllers are enhanced by an L-FDU. The L-FDU monitors the devices attached to and transfers collected state information to the D-FDU by sending heartbeat messages. These messages will be sent by the L-FDU similarly to the D-FDU-core monitoring.

State information is mainly composed of availability data regarding devices such as printer, keyboard, etc. If a device is not available (or not plugged in) any more than this information is sent to the D-FDU, which propagates this information to the OS.

We propose two different implementation variants of such a device controller. The first option is to implement the controller as a dedicated hardware unit on chip (see Figure 22). This unit needs to have

an L-FDU attached similar to the other cores within the node, since the state information is sent by heartbeat messages.

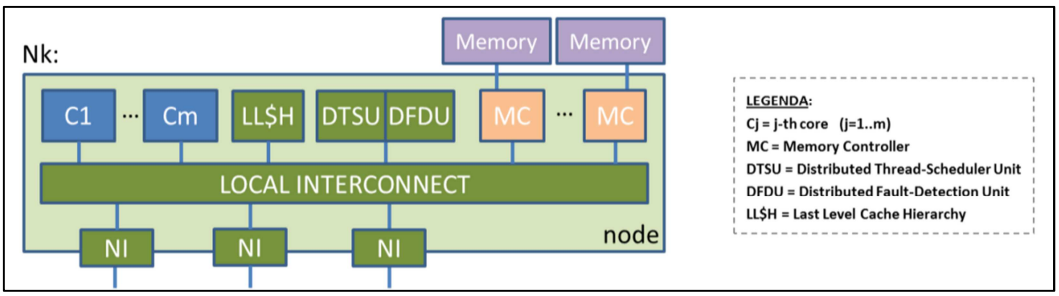

Figure 22: An implementation variant for a Device or Memory Controller monitored by a D-FDU.

The second implementation variant is the usage of a core extended with additional capabilities. This core has a direct connection to the device it is attached to. Figure 23 illustrate this variant. All I/O requests are sent to this core, which then translates the requests according to the device communication.

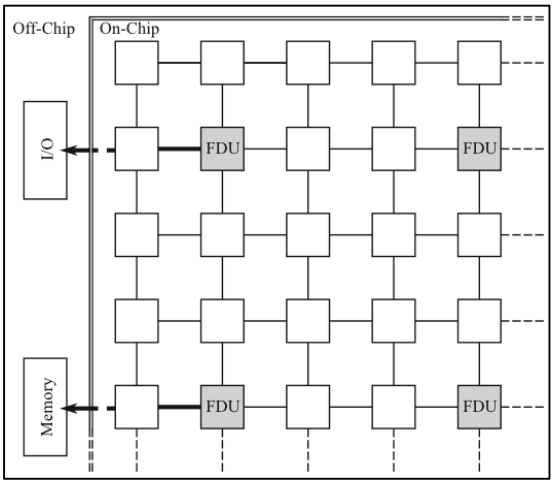

Figure 23: Schematic view on a device monitoring, including an off-chip I/O controller and an off-chip memory controller.

5 Operating System Management

In this section we combine our system level resource management approach introduced in Deliverable D5.1 with fault-tolerance techniques.

5.1 The general structure of the system

In order to manage a large and complex Teraflux system, we believe that we must impose a hierarchy among the cores. Based on the structure of Figure 1 on page 10, the system's physical memory can be structured as follows:

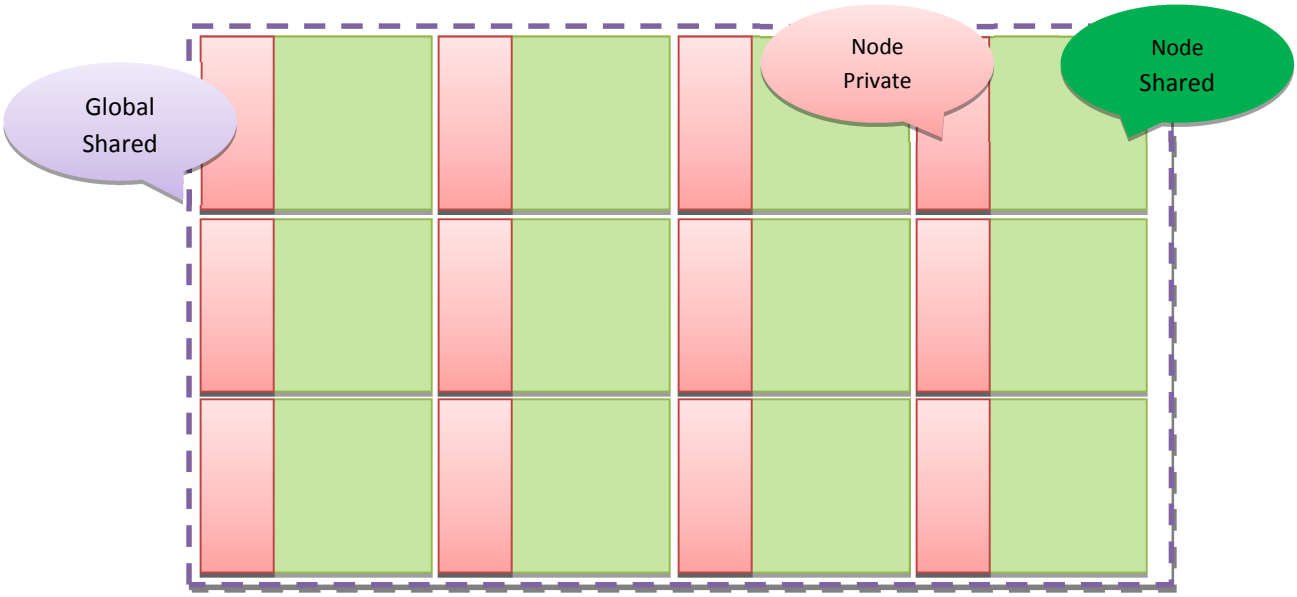

Figure 24: General structure of the memory.

The system is divided into nodes. All cores within a node share the same resources and may maintain hardware coherence within its node. The memory of each node can be divided into local memory that cannot be accessed by other nodes and global memory which can be accessed (but does not require hardware coherence) by all other cores in the system. From an application point of view, the aggregation of all shared pools of memory creates a linear virtual address space.

In order to guarantee an efficient resource management, we assume that the system is logically divided into two parts: the service part (nodes that run a "full operating system") such as a Linux or Windows kernel, and the nodes, which provide support for the Teraflux execution model. Every node is controlled by a microkernel, which is responsible for scheduling and local resource allocation.

For a better understanding the system works as follows:

1. The Compiler generates DF-threads out of sequential code (e.g., C)
2. The execution always starts on the service nodes that generate DF-threads and send them to the different nodes.

3. All DF-threads distributed to the nodes are kept in a "safe memory" queue and scheduled to the node's cores by the D-TSU.

4. After finishing the execution and assuming no fault occurred, the results are written to the node's "safe memory" and the D-TSU writes the results back to global memory. After successful update of the global memory, the thread is removed from the node's queue.

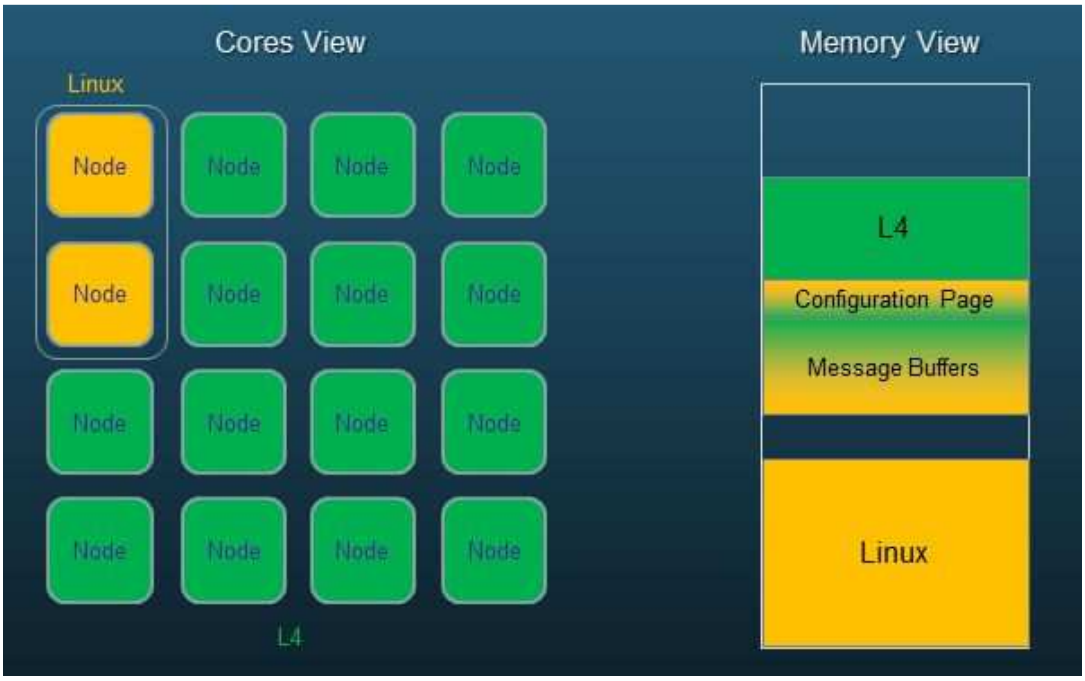

Figure 25: Memory map of the operating system.

5. If a fault is detected by the D-FDU during the thread's execution, the thread is killed (no side effect for DF1 threads) and rescheduled on another core within the node.

6. If a fault occurs while reading or writing data from or to the main memory, we assume a retransmission mechanism to guarantee the completion (at that point we assume that the operation must complete. We may weaken this assumption in the future.)

7. Threads are generated dynamically. The operating system on the service nodes generate the dataflow threads and schedule them on the nodes (at that point the algorithm is centralized, but the next step we will incorporate a distributed version).

Health information gathering and load balancing at the OS level (in the service node):

- Cores send health information (e.g., speed, temperature, number tasks completers, etc.) to the D-FDU.
- The D-FDU sends the information to the service-node.
- The service node takes the health conditions of the nodes into account in order to load balance new dataflow threads.

We believe that the hierarchical execution model described above is a key for managing large and complex systems in the future.

This approach is well suited for handling soft errors as well. Here, we made an important classification of the problems into few categories depending on the HW support we can get. At that point we are focusing on the simplest model, which is based on the following points:

- All memory structures and buses are shielded by error correcting codes or at least parity bits.
- The underlying dataflow execution model provides a side-effect free execution.
- We assume that if the "update global memory" phase begins, it will terminate.

Since we assume that all memory structures are protected by error detection and correction mechanism, we mainly will next focus on faults in the logic. This can be done by space redundancy or time redundancy.

- Space redundancy executes the code on 2 cores (3 are needed for recovery but only 2 for detection), compare the observable outputs and raise a flag if found not to match
- Time redundancy: execute the code twice on the same core and compare results.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Harlow, UK, 1986.

[2] P. Bhojwani, R. Mahapatra, Eun Jung Kim, and T. Chen. A heuristic for peak power constrained design of network-on-chip (noc) based multimode systems. In *VLSI Design, 2005. 18th International Conference on*, pages 124 – 129, jan. 2005.

[3] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[4] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[5] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Soft-Error Detection Using Control Flow Assertions. In *Proceedings of the 18th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*, pages 581–588, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[6] Kees Goossens, John Dielissen, Jef van Meerbergen, Peter Poplavko, Andrei Rădulescu, Edwin Rijpkema, Erwin Waterlander, and Paul Wielage. *Guaranteeing the quality of services in networks on chip*, pages 61–82. Kluwer Academic Publishers, Hingham, MA, USA, 2003.

[7] C. Grecu, A. Ivanov, R. Saleh, E.S. Sogomonyan, and Partha Pratim Pande. On-line fault detection and location for noc interconnects. In *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*, page 6 pp., 2006.

[8] N. Oh, P.P. Shirvani, and E.J. McCluskey. Control-flow Checking by Software Signatures. *IEEE Transactions on Reliability*, 51(1):111–122, 2002.

[9] J. Ohlsson, M. Rimen, and U. Gunneflo. A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog. In *Proceedings of the* 22nd *International Symposium on Fault-Tolerant Computing (FTCS)*, pages 316–325, Los Alamitos, CA, USA, 1992. IEEE Computer Society.

[10] Antoni Portero, Zhibin Yu, and Roberto Giorgi. T-Star (T*): An x86-64 ISA Extension to support thread execution on many cores. pages pp. 277–280. HiPEAC ACACES-2011, July 2011.

[11] R.G. Ragel and S. Parameswaran. A Hybrid Hardware–Software Technique to Improve Reliability in Embedded Processors. *ACM Transactions on Embedded Computing Systems*, 10(3):36:1–36:16, 2011.

[12] Eric Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. *International Symposium on Fault-Tolerant Computing*, 0:84–91, 1999.

[13] S. Schlingmann, A. Garbade, S. Weis, and T. Ungerer. Connectivity-sensitive algorithm for task placement on a many-core considering faulty regions. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 417 –422, feb. 2011.

[14] M.A. Schuette and J.P. Shen. Processor Control Flow Monitoring Using Signatured Instruction Streams. *IEEE Transactions on Computers*, 36(3):264–276, 1987.

[15] Sebastian Weis, Arne Garbade, Faruk Bagci, and Theo Ungerer. Fault detection and reliability techniques for future many-cores. 6th international summer school on advanced computer architecture and compilation for high-performance and embedded systems (ACACES 2010), pages 175–178, 2010.

[16] Sebastian Weis, Arne Garbade, Sebastian Schlingmann, and Theo Ungerer. Towards Fault Detection Units as an Autonomous Fault Detection Approach for Future Many-Cores. In *ARCS 2011 Workshop Proceedings*, pages 20–23. VDE Verlag, February 2011.

[17] Sebastian Weis, Arne Garbade, Julian Wolf, Bernhard Fechner, Avi Mendelson, Roberto Giorgi, and Theo Ungerer. A Fault Detection and Recovery Architecture for a Teradevice Dataflow System. In *Data-Flow Execution Models for Extreme Scale Computing (DFM) 2011 Workshop Proceedings*. IEEE Computer Society, 2011.