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

PROJECT NUMBER: 249013

TERAFLUX

Exploiting dataflow parallelism in Tera-Device Computing

D5.1 - Design Exploration of FDUs and Core-Internal Fault-Detection

Due date of deliverable: 31st December 2010
Actual Submission: 31st December 2010

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

Lead contractor for the deliverable: Universitaet Augsburg (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#	Author	Organization	Change History
0.1	**Sebastian Weis, Arne Garbade, Sebastian Schlingmann and Theo Ungerer**	**UAU**	**Initial release**
0..2	**Avi Mendelson and Doron Shamia**	**Microsoft**	**OS and core-internal fault detection**
1.0	**Theo Ungerer**	**UAU**	
1.1	**Avi Mendelson and Doron Shamia**	**Microsoft**	**Preliminary review**
InternalReview	**Theo Ungerer**	**UAU**	**Proofreading**
InternalReview	**Pedro Trancoso**	**UCY**	**Overall review**
1.2 – Release	**Sebastian Weis, Arne Garbade, Theo Ungerer**	**UAU**	**Document finishing**

Release Approval

Name	Role	Date
Theo Ungerer	**Originator, WP Leader**	
Roberto Giorgi	**Project Coordinator for formal deliverable**	**31.12.2010**

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.

Arne Garbade, Sebastian Weis, Sebastian Schlingmann, Theo Ungerer
Universitaet Augsburg

Avi Mendelson, Doron Shamia
Microsoft Research and Development

Printed in Siena, Italy, Europe.
Part number: *please refer to the File name in the document footer*.

Glossary

Cluster	Group of cores; synonymous of “node”
COTSon	Simulator software provided under the MIT license by HP-Labs
Core	Processing element
FDU	Fault Detection Unit
MCA	Machine Check Architecture
Node	Group of cores; synonymous of “cluster”
TAAL	TERAFLUX Architecture
TBM	TERAFLUX Baseline Machine
TSU	Thread Scheduling Unit
UAU	University of Augsburg
MSFT	Microsoft R&D Israel
Electromigration	Mechanical and matter wear out, often caused by moving electrons in integrated circuits.
L4	Microkernel being used to manage a cluster of computers
Nanos	Alternative microkernel to be considered
Service-OS	A Linux OS that manages the I/O and other resources of the system

Executive Summary

The aim of WP5 is to establish a reliable system out of unreliable components, which can be cores and interconnects. WP5 in DoW Task 5.1 (months 1 - 12) "**Design Exploration of FDUs and Core-Internal Fault-Detection"** states:

> Design exploration of FDUs: UAU will investigate the general specification of the Fault Detection Unit (FDU) as well as the design space exploration of the different variants of FDU implementations. The goal is to develop a specification of an FDU and its interface to a core, to a Thread Synchronization Unit and between FDUs.
> At the early stages of the project, MSFT will focus on the design of the feedback loop between the high level SW manager (e.g., the OS or middle-ware layer) and the core internal fault detection. By doing that, we hope to come with holistic approach that incorporate HW monitors, HW fault tolerant components, SW layer of management and define agreed APIs that could be used in the next stages of the research in order to increase reliability and even to improve performance when faults become more common."

The work of UAU is focusing on inter-core fault detection techniques using Fault Detection Units (FDU). Therefore UAU started with a design space exploration of different FDU variants (push, pull, alert mechanisms for heartbeat messages), FDU implementations, and interfaces.

The design space exploration has finally resulted in a proposal of a functional FDU specification based on the MAPE (Monitoring, Analysis, Planning, and Execution) cycle of Autonomic Computing. Abstract message interfaces of FDU to all communication partners (FDU-core, FDU-TSU, FDU-FDU, FDU-operating system) were specified for push, pull, and alert messages.

UAU and MSFT worked together on a preliminary specification of core-internal fault detection. The result proposes how the performance measurement system and the machine check architecture of current AMD/Intel processor families can be exploited for reliability in the TERAFLUX architecture. Moreover, we propose to extend the heartbeat message format to transport additional temperature and performance measures.

MSFT worked on the system level management of resources, including the loop between the different levels of the systems in order to achieve load balance and fault tolerance to guarantee the performance and the reliability of a massive parallel system. The control of scheduling and resources will be done in hierarchical level, where distributed FDUs are used to guarantee the characteristics of the basic nodes by accessing the different resources of a single cluster such as the cores, local memories, etc. The FDU will be in charge on inquiring the execution state; e.g., faults and performance of each core. The document will also define the bidirectional communication between the "system" and the FDU so that the OS will be able to access the FDU and to get the required information on the health of the cluster. Later on, this information could be used as part of the recovery and load balance mechanisms on the system.

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

1 Introduction

It is expected that within the next ten years a chip will be able to host more than 1000 heterogeneous cores [1]. With an ongoing decrease of the transistor size, the probability of physical flaws on the chip induced by voltage swinging, natural cosmic rays, thermal changes or variations in the manufacturing process will increase [2]. Even though reliability and fault tolerance have always been important issues for mission critical systems, they are now new for the upcoming general-purpose many-core processors, bringing up completely new challenges.

While in mission critical systems reliability has always been essential at all cost, the architecture of general-purpose processors is strongly influenced by economical constraints. This requires reliability solutions which scale with the number of cores and the increasing failure probability on a chip but within a reasonable architectural effort [3].

A computer system is fault tolerant, if it provides safety and liveness even in the presence of faults. Safety means that a system never produces any incorrect results, even if a fault occurs. It hides all faults from the user. Besides that, it is required that the system does something useful. Unplugging the power supply will stop the faulty component from generating computation results, hence ensuring its safety, but is often an unacceptable solution. This is where liveness comes in. Liveness means that the system continues execution, despite of faults. Liveness combined with safety means that the system makes forward progress, while assuring that no wrong results are produced. However, situations exist where liveness and safety of a system cannot be guaranteed. In such scenarios it is at least necessary to maintain the safety of the system. As an example it can be referred to an automatic teller machine (ATM), which disables itself, instead of giving out a wrong amount of money [4].

The increasing transistor density makes errors on the chips in future many-core systems unavoidable. For that reason, even current multicore processors offer the ability to detect particular faults which occurred within the cache, TLB or the core microarchitecture [5]. In addition, the fault tolerance literature documents different fault detection and recovery techniques, which extend the cores' microarchitecture [4][6].

If a fault is not correctable by a core itself, a current processor would raise an exception and the operating system is likely to be shutdown. Projected on future many-core processors, this would mean an increase of catastrophic shutdowns of the complete system due to the increasing failure rate.

Since the TERAFLUX architecture is planned to be comprised of thousands of cores it would be likely to suffer from such catastrophic processor shutdowns as well. Nevertheless, the dataflow architecture offers in the case of fault tolerance certain advantages in comparison to architectures based on von-Neumann's execution model. In particular, the functional semantic of dataflow in conjunction with a Thread Scheduling Unit (TSU - D6.1) supports the ability to restart the execution of a TERAFLUX-thread on a more reliable core in the presence of non-correctable faults.

Those techniques support the concept to distribute TERAFLUX-threads on reliable cores and exclude defective cores dynamically at runtime.

The first objective to provide reliability within the TERAFLUX architecture is the detection of faults. Whereas the distribution of TERAFLUX-threads is provided by a distributed Thread Synchronization Unit (TSU), the fault detection of the cores is handled by the Fault Detection Unit (FDU). The FDU's primary goal is to monitor and detect faults on the cores over an unreliable communication channel which may itself suffer from faults. While the first goal is to detect faults, the second goal of an FDU is to maintain the operability of a cluster, for example by dynamic clock and voltage scaling while incorporating the monitored failure rate, heat, and core utilization.

Since the TSU is in charge of the thread assignment on the cores, it has to keep the information about interdependences between threads, the current assignment of threads to cores, and the state of thread execution. If a core suffers from a fault, which might affect the reliability of the core or the system, the FDU should detect this malfunction and inform the TSU urgently.

This behaviour illustrates the basic requirements for an FDU. It is not only important that faults are reliably detected; in addition, it is also necessary to locate and prevent problems and report such information to other administrative units, like the TSU and the Operating System.

1.1 Document Structure

Section 2 describes the terminology of fault, error and failure and provides basic fault models for cores and interconnects. Section 3 specifies the internal structure of the Fault Detection Unit (FDU) as running a MAPE cycle of monitoring cores. Section 4 defines the communication protocols and the interfaces of the FDU to the cores, the TSU, the Service Node with operating system, and to other FDUs. Section 5 describes the core-internal fault detection techniques of current cores and how they can be exploited by the FDU. Section 6 presents the first results of the TERAFLUX operating system research.

1.2 Relation to Other Deliverables

Deliverable D6.1 provides the basic TERAFLUX architectural description. D7.1 and D7.2 specify the interfaces within the COTSon and the basic simulation implementation of the TERAFLUX architecture, which also includes details on the scheduling unit. The Deliverable D5.1 is coherent with all previously mentioned documents.

1.3 Activities Referred by this Deliverable

All activities described in this document concern the work of UAU and MSFT done on Task 5.1 "**Design Exploration of FDUs and Core-Internal Fault-Detection"** in WP5 covering months 1 to 12 of the project:

> Design exploration of FDUs: UAU will investigate the general specification of the Fault Detection Unit (FDU) as well as the design space exploration of the different variants of FDU implementations. The goal is to develop a specification of an FDU and its interface to a core, to a Thread Synchronization Unit and between FDUs.
> At the early stages of the project, MSFT will focus on the design of the feedback loop between the high level SW manager (e.g., the OS or middle-ware layer) and the core internal fault detection. By doing that, we hope to come with holistic approach that incorporate HW monitors, HW fault tolerant components, SW layer of management and define agreed APIs that could be used in the next stages of the research in order to increase reliability and even to improve performance when faults become more common."

2 Terminology and Basic Fault Models

This section states the terminology used through this document and introduces then the considered higher level fault models. The fault models are required as basis for fault injection as specified in Task 7.3 of WP7.

2.1 Fault, Error and Failure

In the literature the terms **fault**, **error** and **failure** are mixed up and are partly used synonymously. In the fault tolerance domain, we distinguish between these three terms by their occurrence in different layers of the abstraction hierarchy.

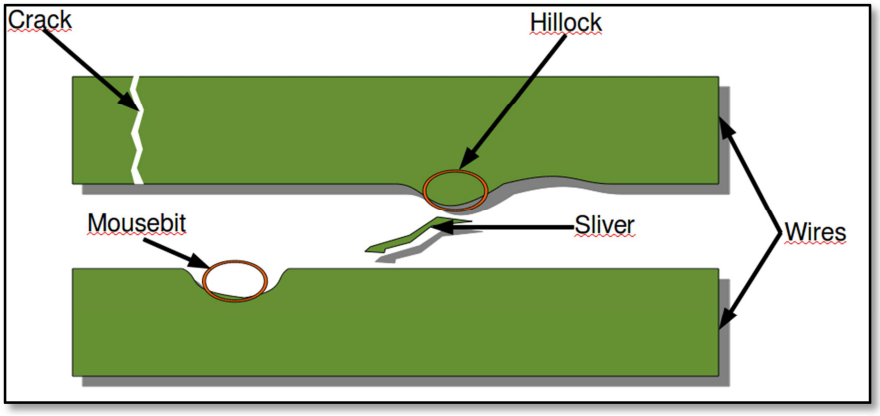

Figure 1: Examples of defective interconnections or CPU internal wires taken from [7]

Low level faults may occur on the lowest physical level of the system, which are caused by **defects** of the hardware. Some kinds of defects are shown in Fig.1. The manifestation of a fault may then cause an **error**, which is shown up in one or more flipped bits.

Such an error may cause a **fault at higher level**, e.g. a core fault or an interconnect fault.

A **failure** is an incorrect computation result showing up in the software layer. The appearance of a failure may finally lead to a system crash or an incorrect output, which is usually visible to the user.

Faults, errors, and failures may differ in terms of their duration. According to that, they can be divided in three subcategories:

- **Transient errors** which are also known as **soft errors** [7] **(**or single-event upsets) appear and disappear in an unpredictable way. Common causes for a transient error are cosmic rays, which may disrupt the charge of a DRAM cell. These errors are non-deterministic and because of their unforeseeable behaviour hard (if not impossible) to locate. Therefore, the key metric for this error type is a statistical number – the **S**oft **E**rror **R**ate. This number describes the statistical occurrence of an error. Transient errors are usually handled in current processors by error correction algorithms (such like backward- or forward-error-correction [8]).

- Longer persistent faults can cause **intermittent errors** [7] that can last from a few milliseconds up to several minutes. They are mostly caused by temporary hot spots on the processor's die. As an example, during the manufacturing process a little metal sliver could be left behind between two interconnection lines. While the temperature heats up the wire will widen and the sliver can cause a short. When the material cools down, the wires shrink back to the normal size and the intermittent error disappears. In some cases the sliver can burn into the material and produce a permanent error. In this case, an intermittent error becomes permanent and must be treated in a different way.

- Permanent physical flaws can cause **permanent errors** [7]**,** which are enduring for the complete life time of the chip and do not disappear after its first occurrence. This type of error arises out of defects in the manufacturing process or hardware aging during runtime due to electromigration.

2.2 Fault Models

Fault models provide an abstraction between the particular fault source and its manifestation in the different layers of the abstraction hierarchy. Because we are mainly interested in higher level faults, we will use the terminology of transient, intermittent, and permanent faults instead. To investigate reliability and fault tolerance solutions for complex many-core processors, it is essential to abstract from single physical flaws or low level fault models, like the often-cited stuck-at fault model [4] at the lowest hardware level.

This document combines two higher level fault models. First, the **core fault model**, which abstracts from low level microarchitectural core fault models like bridging fault model or delay fault model. Second, the **inter-core fault model**, which considers an unreliable communication between the cores, i.e. faulty interconnects and routers.

2.2.1 Core Fault Model

Considering the manifestation of core faults, we distinguish further between total faults, partial faults and retarding faults.

- A core suffers from **a total fault** if it is completely broken down and therefore no longer responsive. This may result from a serious fault in the core (like a stuck-at-fault).
- By contrast, if a core is reachable via the interconnection network and still responding to requests, but the core's internal hardware fault detection has identified an anomalous behaviour, then the core is suffering from a **partial fault**. Depending on the impact of the fault, this may require that results from this core, like stores to memory or newly issued threads, could no longer regarded as functional correct and must be suspected incorrect.
- **Retarding faults** are faults that do not directly impact the functional correct execution, but the performance capability of a core.

The **core fault model of TERAFLUX** covers all faults which are detected by the intra-core fault detection mechanisms and total faults of a core (the core cannot answer anymore). Technically, we are focusing on the **AMD Opteron Machine Check Architecture** [9] which is available in AMD

processors. Current implementations of the AMD Opteron Machine Check Architecture detect faults in the load/store unit, the data cache unit, the instruction cache unit, the bus unit to the Northbridge, and the on-chip Northbridge itself, among others. The Machine Check Architecture itself distinguishes between correctable and not correctable faults. We consider cores suffering from not correctable faults as unreliable. If the core suffers from a specific amount of correctable faults it might be suspected unreliable, too.

In complement, the **AMD Opteron Instruction-Based Sampling** and the **AMD Performance Counter Register** [9][10] are used to identify performance degradations of a core. The Instruction-Based Sampling is generally used for performance profiling on the system and application level. We utilize the Instruction-Based Sampling as well as the Performance Counter Registers of the AMD Opteron architecture to get runtime information of the executed code and thus information about the probability of performance degradation due to a retarding core fault, which did not directly influence the functional correct execution.

2.2.2 Inter-Core Fault Model

As mentioned before, the inter-core fault model encompasses link faults in the interconnection network. The faults in the inter-core fault model are further classified by the intensity of the fault: **Total faults** are complete link faults; **partial faults** are among others a wire defect of links that degrade link performance [11].

It is assumed that all fault types can be detected on a lower level by the connected routers. This means if a serial wire is suffering from timing issues resulting from crosstalk or electromigration, which causes a repeated or deferred packet transmission on the phit layer we assume a retarding link fault. If the wire or a router is completely broken down, we refer to the total link fault.

The simulation model of the interconnection network will be defined within WP7 and will be part of the COTSon extensions.

3 The General Specification of the Fault Detection Unit (FDU)

This section is meant to specify the general FDU tasks. It describes in an informal way the objectives of the FDU and depicts the internal behaviour of an FDU. A detailed description and specification of FDU's interfaces is given in Section 4. We further base our investigations in the following on the TERAFLUX architecture, which is described in D6.1 and D7.2.

3.1 High level TERAFLUX Architecture

An FDU is responsible for the collection of health states of a cluster of cores (see Figure 2) within the TERAFLUX processor. A more detailed instance of the TERAFLUX architecture could look like the block diagram of Figure 3.

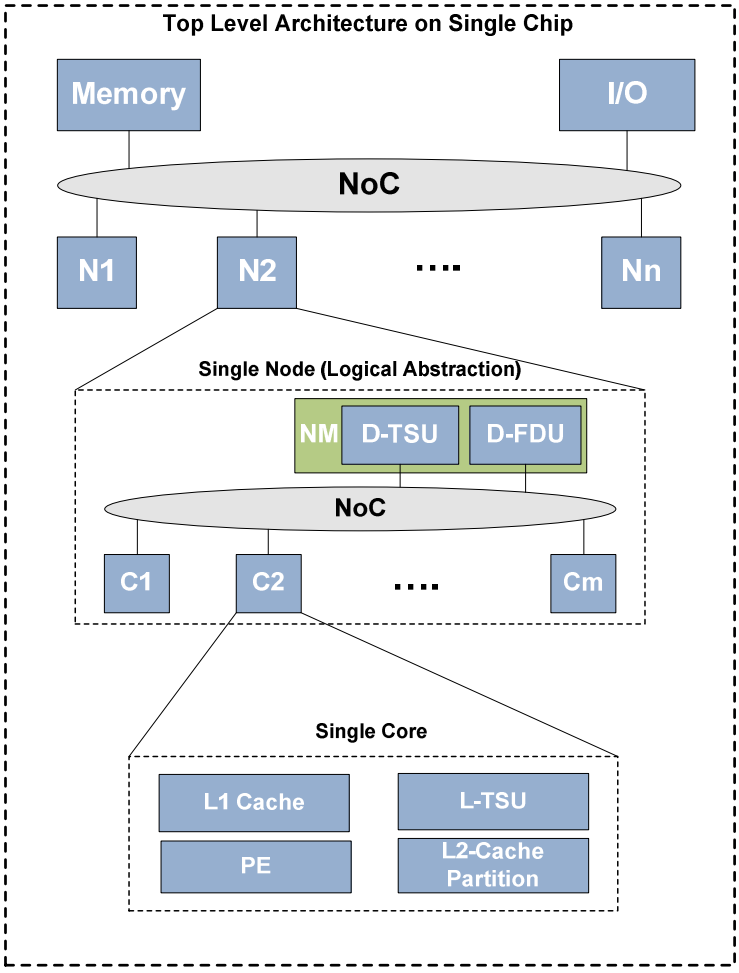

Figure 2: TERAFLUX High level Architecture. Here we distinguish between D-TSU, D-FDU, and L-TSU, but for easy of reading in this document we just call TSU the D-TSU and FDU the D-FDU.

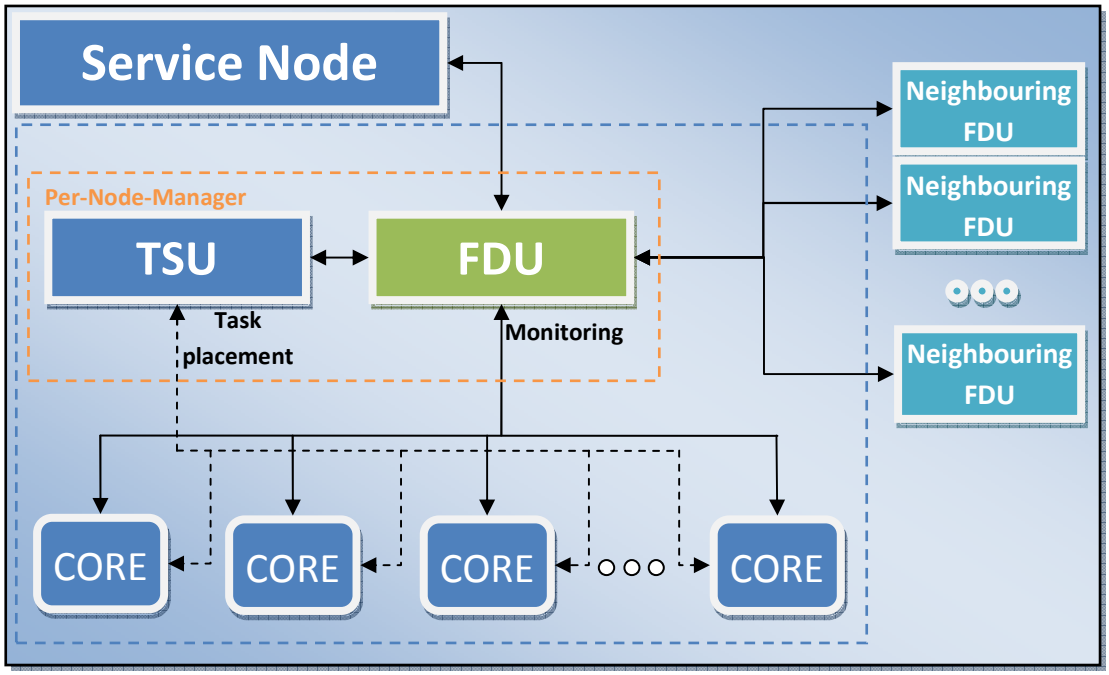

Figure 3: Schematic view on the communication partner

The TERAFLUX-architecture will be divided into two hierarchical layers; at the higher level we will have a **Service Node** hosting the main TERAFLUX operating system (e.g., Linux) and taking care of I/O, Interrupts, and legacy code. At the lower level we will have a set of clusters, which are called Nodes in TERAFLUX terminology (see figures 2 and 3). A cluster is comprised of a number of cores, one TSU and one FDU. It is expected that due to the feature size and other process issues, the basic hardware blocks such as processor, network, logic, etc. may suffer from reliability issues. Thus, the major challenge of the TERAFLUX hardware is how to build a reliable system out of many unreliable components. As explained before, such hardware may suffer from transient faults, from permanent faults and it is also assumed that some parts may become faulty for a limited time and may "recover" to a reliable state later on.

It is also expected that such a system may suffer from process variability and similar effects that may allow different parts of the system to run at different speeds (and with different power consumption). Please note that from the system point of view, we are interested in the effective performance of a core or even of a whole cluster.

Since our system is built in a hierarchical structure, we decided to manage the power, performance, thermal conditions, and faults in a hierarchical manner as well; at the top level, the Service Node that runs the operating system will manage resources at a coarse granularity, while each node will manage the resources it consists of (the node affiliated cores). This structure allows us to manage resources at different scale and at different time granularities as well. The lower level will take care on most of the urgent considerations such as thermal issues that need spontaneous response, while the top level can

handle most of the events at coarse timing constrains. Therefore, at the cluster level we devote special hardware modules, named TSU and FDU to control the resources. The TSU will be in charge of thread scheduling and recovery. The FDU will be in charge of monitoring the actives[1] of all resources within the cluster, handling faults and report to the TSU and the OS manager at higher level. Power management and other core control related activities will be performed by the FDU as well. At the service level, a special OS driver will be defined to monitor the cluster activities. This driver will communicate with the rest of the software in the system through special APIs, defined as part of the TERAFLUX architecture.

Although, at the beginning of the research we will refer to the entire activities of the resource management, power management and fault tolerant in a "purely" hierarchical manager. Later on in the research we will extend the model to allow some of the high level functionalities to be distributed. Those will be implemented in a distributed manner allowing the FDUs to monitor each other, in order to detect malfunction hardware components including routers and links.

At the cluster level the management is mainly done by three components; the core (as discussed before), the FDU and the TSU.

3.2 Scope of the FDU

The tasks of an FDU are to gather, analyse, plan and respectively react on the provided information of its affiliated cores. It is additionally considered that the cores themselves are only reachable over an unreliable communication channel, which ultimately means that interconnections can fail as well. Every core within a TERAFLUX-cluster is monitored by an FDU. The analysed and previously filtered information supplied by the FDU may be used by the TSU to decide how to distribute the threads among the cores within a cluster, and if possible, the FDU may also set the clock rate of each individual core. We may decide to reduce the clock rate of a core that has suffered many faults before deciding to shut it down. The information may also be used for higher level decisions such as load balancing and shutdown of clusters [13].

From the FDU point of view, each device (e.g. core or router) is responsible for detecting certain internal faults of its own. If it detects an error, then it is not feasible to report the error in every detail. Especially if an error is correctable by the device itself, there is no immediate need to report it directly to the FDU. Of course, each occurrence of an error should be logged, so the FDU can ask for performance measures, which could consist of the amount of retries a device needed to finish a computation correctly.

To limit the workload of a single FDU, we organise the total number of cores into clusters. Each cluster consists of a number of cores, an FDU and a TSU. The number of cores per cluster, e.g. 16 cores per cluster, will result from the network design space exploration (to be performed by UAU in year 2 of the project).

[1] Sending messages

3.3 FDU Organisation

To achieve the main goal of ensuring the system's stability and performance, we apply an autonomic/organic computing approach [15][16] organising the FDU operation principle into the following four consecutive steps: **monitoring**, **analysing**, **planning**, and **executing** (see Figure 4). This MAPE-cycle is operating on a set of managed elements. These managed elements comprise intra-cluster elements (the affiliated cores and the TSU) and inter-cluster elements (other FDUs) in other clusters. In the following section we describe the four phases of the MAPE-cycle in detail.

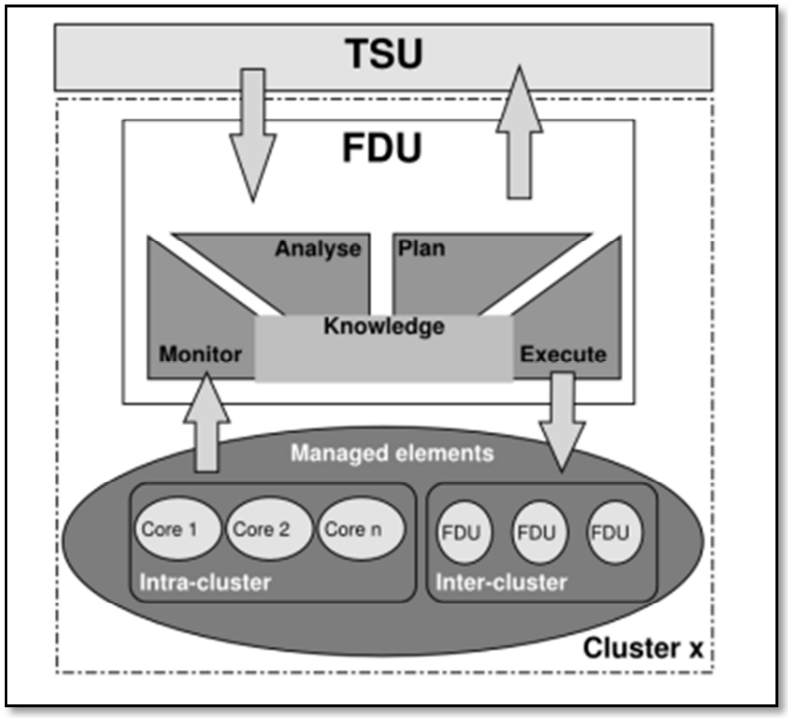

Figure 4: The MAPE-cycle [15] – Adaptation for FDU purposes

3.3.1 Monitoring

As already mentioned, detecting faults is one major task of the FDU and crucial for a reliable TERAFLUX system. Therefore, the FDU has to **monitor** all unreliable processor components within its cluster that may influence the system stability and that may suffer from faults. These components are the cores within a cluster, their routers and links. Additionally the FDU monitors its affiliated TSU and other FDUs from neighbouring clusters (see Figure 4).

The monitoring mechanism could be realized in two different ways: First, the **pull mechanism** can be used as follows. The FDU sends request messages to the monitored core and awaits a response message from that monitored core. Second, the **push mechanism** can be established to ensure that the monitored core periodically sends a message to the FDU. In order to initialise the push mechanism the FDU sends a request message, which contains an interval value to the monitored device. This will

instruct the core to send periodically a message in the defined interval. Those messages are also known as heartbeat messages.

Besides the two mechanisms that are initiated and controlled by the FDU, there is another mechanism that is decoupled from the instructions of the FDU. As already mentioned, the monitored devices are able to detect errors for themselves. This will be used to generate **alert messages**, which are sent asynchronously to the FDU to inform it about a fault.

In later stages of the project, additional mechanisms may be considered. One option would be to attach a core's status information to a normal application message[2].

3.3.2 Analysis

After the FDU has received information from its managed elements in the monitoring-phase, it makes forward progress to the analysis phase, in which the information is analysed.

In the analysis-phase the FDU aims for four different targets:

1. Detecting faulty cores and other FDUs
2. Detecting unresponsive cores and other FDUs
3. Detecting broken interconnections
4. Determining the performance of cores and interconnects

In the following subsections, we describe those four objectives in more detail and derive different tasks from the objectives. We ordered the four tasks regarding to their perceived importance.

3.3.2.1 Detecting faulty cores and other FDUs

At the current stage of the research, we consider two techniques to detect faulty elements.

First, the FDU receives an alert message from a core or an FDU. This means, the device has detected a fault by itself and informs now the FDU. The FDU suspects the device to be faulty and informs the TSU in its cluster.

Second, the FDU analyses the periodically received messages from the cores and FDUs. When a determined threshold of a key metric is exceeded, the FDU suspects the core or FDU to be faulty and informs the TSU.

3.3.2.2 Detecting unresponsive cores

The second most important target is the identification of unresponsive cores, i.e. total faults of cores. Those cores would not send any messages to the FDU and do not respond on requests. But those behaviours are also well known symptoms for a potential faulty interconnection (or a whole NoC's region). To detect such unresponsive cores, we will incorporate process failure detector implementations from the field of distributed systems, which can exploit the periodically received messages to detect unresponsive cores [12][17].

[2] This is also known as piggybacking or lazy monitoring.

Furthermore, the FDU will keep information about the execution state for each core. The state will be preliminary set to the state "suspected" and no new workload should be assigned by the TSU until the FDU was able to figure out, what the problem actually is [12][17].

The first step to identify the error is done by sending a request message on a different route to the core. That may bypass a faulty router or link and the request may reach the core. If so, within this message the FDU orders the core to send its response with the same routing algorithm. As a result of this, the FDU may receive heartbeats from the core again. This is an indicator for a changed interconnection connectivity, which leads to the Planning task.

If the FDU still does not get any life signs from the core, then it is assumed that there is a major problem with the interconnection network or the core is indeed suffering from a total fault. In any of those cases, the core will be marked in the FDU per-core table as faulty. The FDU's last option is now a cluster wide broadcast message. If there is still a connection to the core and the core is still alive the nature of a broadcast will guarantee that the core will receive it. This is a drastic way to reach a core due to its high load on the interconnection network, but it is necessary to rule out a false positive. The response of a core should then also be send back by broadcast.

3.3.2.3 Detecting broken interconnects

As mentioned above, it is necessary to infer the interconnection connectivity of a cluster, in order to distinguish between errors in cores and errors regarding the communication. So it is desirable to have a precise representation of the cluster network. In future work, we may also want to know if a communication path is completely broken or if it is still usable but with some degraded properties like lower bandwidth.

3.3.2.4 Determine the performance of cores and interconnects

This task is meant to maintain the inner cluster performance. Let's say that some cores of the cluster may suffer from electromigration earlier than other or from the increased temperature of that chip region. Those issues can be handled by decreasing the clock rate of the involved cores. While this becomes a permanent solution to a core, its computing capability has to be rated differently in order to support a smarter thread distribution by the TSU.

The new cluster status is a matter of particular interest for the TSU and the higher level thread management, as they decide which device or chip region is appropriate for a given workload. Therefore, the FDU collects the data from the core's heartbeat messages and determines with this information the current performance of each affiliated core. The information may consist of the amount of retries a core needed to execute a thread. This gives a hint on how often a core suffers from single event upsets (only not correctable errors are counted). If this counter hits a threshold, the FDU will try to minimize the effect.

3.3.3 Planning

In addition to the semantic analysis of the received messages where already the first decisions on what actions are taken will be made, we propose a further planning instance. This long-term (strategic) decisions are mainly related to the cluster health management. This means that essentially all of the

information, such as the state of the communication network or the computing capability of all cores is used to infer the state of the cluster. Within this approach appropriate opportunities may arise to reconfigure a cluster in order to minimize congestion and to enable the TSU to do a smarter task placement. To ensure the clusters health we propose the following three planning objectives.

- ***The placement of the FDU***. This will help to minimize the heartbeat induced congestions and reduce response times for more distant cores. Our first approach to this objective is the placement of FDU in the centre of the cluster. This is not a trivial problem, however, since the cluster is not necessarily symmetrical[3], and may have holes in the communication network. Nevertheless, we expect the placement of the FDU on the centroid to be an ideal decision.

- ***The decreasing and increasing of the clock rate of the devices*** (cores, routers, interconnection bandwidth). Using the adjustment of clock rates can solve various problems on the chip, which are associated with a raised heat stress. Since the devices may interact with each other in terms of heat, the planning with respect to changes in clock rates are thus carried out at cluster level. Should the situation in a cluster relax again, the clock rate of individual devices can again be raised in the same way.

- ***Rearrangement of the clusters.*** Emerging permanent errors will be detected and make necessity of rearrangements of the clusters from time to time. This reorganization is also a task that is performed in the planning stage. However, contrary to the above described planning goals, here all the information is needed from all clusters. In addition, there must be exactly one FDU that sets this new order and picks out a new FDU core for each new cluster.

3.3.4 Execution

Depending on its analysis and planning results, the FDU takes action to solve the consequences of a fault in the cluster. At this point we distinguish between reactive behaviour and deliberative behaviour. The **reactive behaviour** is a result directly inferred from the analysis process and does not do throw any planning instances. This should facilitate a fast reaction on critical system states, reported from a core. These fast reactions are needed to minimize the probability that a corrupted core state compromises a running thread, which then spread out any wrong (starting) values for other threads.

The **deliberative behaviour** mentioned before does not have such strict constrains in terms of reaction time. Deliberative behaviour uses the decisions made in the planning phase and effectuates them. Examples we foresee for those decisions are the redeployment of the FDU itself or the rearrangement of cluster structures. Some of those decisions may also be made by a group of FDUs; however, this is out of scope for this document and will follow in the next project year.

No matter which action the FDU performs it also has to inform the other components of the system, which may be interacting with the cores (such as scheduler for task placement). The FDU briefs the other services which might depend on the gathered and pre-processed information. At the current project state there are the following components, which have to be informed:

[3] The shape of a cluster may become convex or concave, which can complicate the (re-)placement of the FDU.

- the affiliated TSU needs to know the core states for thread placement and
- the TERAFLUX OS running on the Service Node needs information about the cluster states.

4 FDU Interface Specification

This section describes the interface specification of the FDU. We start by reviewing the types of messages and the communication partners of the FDU, and then describing how the communication should be implemented from the perspective of the FDU. Note that this is a preliminary specification with respect to the communication. Later in the project it might be necessary to make changes to this specification and adapt it to the needs of all communication partners.

First, the high level interfaces are defined. Those interfaces define the communication partners from the perspective of the FDU. The communication specification is then classified in two categories, *Time-Driven* and *Event-Driven* and briefly explained. The last part of this section describes the individual messages as they are anticipated by the FDU. It is not yet implemented as an API, but a rule of how communication should proceed from the FDU perspective. The communication partners of an FDU are the monitored cores, the TSU, the neighbouring FDU (other cluster), and the Service Node (respectively OS).

4.1 High level communication protocol for fault detection

Since the cores are able to detect internal errors independently and moreover can distinguish whether there is a correctable or not correctable error, we propose a mixture of communication for fault detection. The two communication mechanisms are described below.

4.1.1 Time-driven: Heartbeats

On the one hand, the messages are sent periodically. These messages contain a set of core health information, which are predefined by the FDU. Of particular interest are errors that influence the actual core performance. The performance is measured by the number of attempts a core needed to execute a thread until the execution was finally fault free. Moreover, these time-driven messages are used as so-called Heartbeats, which support the FDU to confirm that the core is still alive. If no messages arrive within the expected duration at the FDU, then the FDU loses the trust to the core. In consequence the FDU considers the core as faulty and informs the TSU. Thus, the TSU takes all necessary actions to isolate the faulty core. Heartbeats can be implemented as push (the preferred one) or as pull mechanism.

Figure 5 shows a core sending periodic heartbeat messages. After the first not received heartbeat, the FDU regards the core as faulty and informs the TSU, which takes care for the correct thread execution.

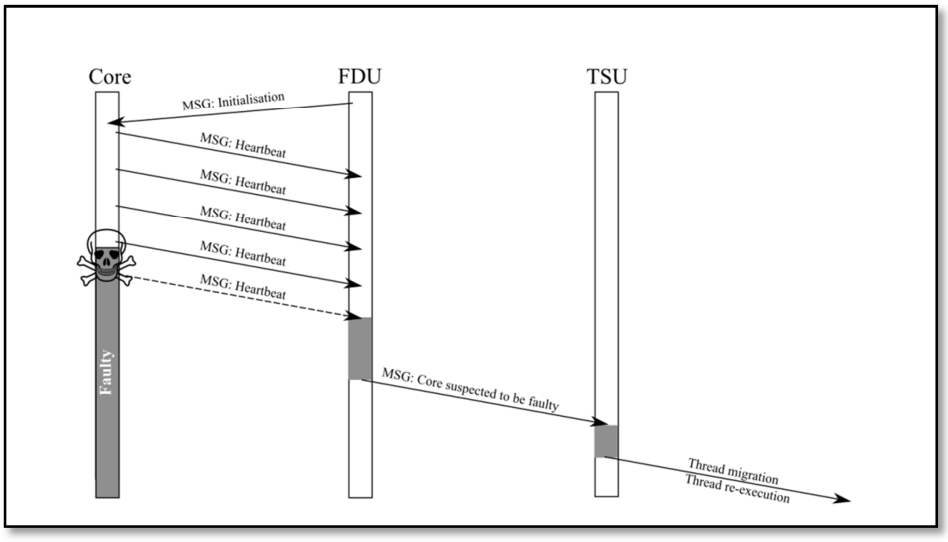

Figure 5: Time-driven fault detection with Heartbeat push messages

4.1.2 Event-driven: Alerts

Another mechanism is when messages are automatically sent by the core. This mechanism is used when the core has detected an error that was not correctable and a re-execution of the thread cannot be done with safety. The message of the core is defined as an alert message and sent with priority on the network. The high priority of this message ensures that it arrives with low latency at the FDU and can be handled immediately.

The FDU will inform the TSU immediately about the affected core. Thereafter, the FDU sends commands (reduce clock speed or turn off the core) or requests (requests for information about the core status) to the core. Figure 6 depicts an error detected by the core itself.

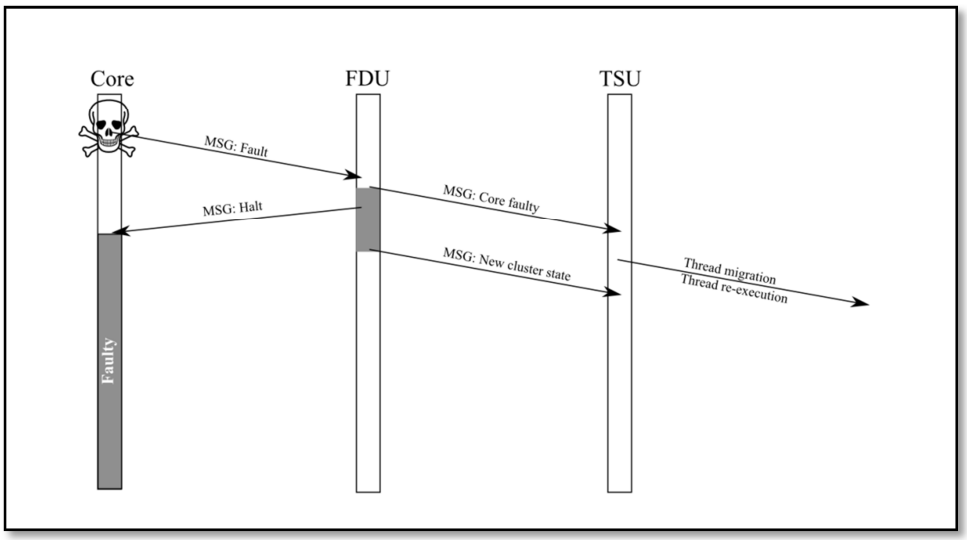

Figure 6: Event-driven fault detection

4.2 High Level FDU Interfaces

The FDU interfaces are highly dependent on the overall TERAFLUX architecture. This makes it essential to define a high level abstraction of the FDU communication specification first, which are independent form the underlying network topology. The following is the complete list of the communication partner:

- A core within a cluster,
- The affiliated TSU,
- The Service Node with the OS (as a higher level management instance)
- The neighbouring FDUs (future work).

From the compounds of the communication partners now result the interfaces of the FDU. As far as all the communication partners of the FDU are hosted on one of the cores of the system, the communication can be carried out to generic and packet-based messaging. The message structures of the interfaces are described below.

4.2.1 Interface: Request

The pseudo function call `Request` defines a query message from an FDU to a core within a cluster (or to another FDU) in case of the pull mechanism. The definition of the parameter passed to the pseudo function is as the following. The push mechanism is used to initiate the heartbeat messages.

Message format:

`Request( priority, core_id, timestamp, req_msg_id, req_code, data )`

Field Descriptions

The `priority` field is used to access the interconnection network with one of four different privilege levels. The usual value for this parameter is "1" because most of the requests should be answered in low latency. (0 = highest priority, ..., 3 = lowest priority)

The `core_id` field identifies the message destination. Each `core_id` is unique and belongs to one core on the TERAFLUX processor, which can be accessed over the interconnection network. The `core_id` consists of two coordinates of the 2D mesh and represents the place where the core is located. Each coordinate is a positive 5 bit integer value. Together they build the `core_id` with the x-coordinate in the lower 5 bit range and the y-coordinate in the higher 5 bit range.[4]

The field `timestamp` holds the FDU's local time at the moment of message creation. It may be used to calculate the response time of the target core. Furthermore, it may be used to gain a nearly synchronous cluster time, which can be achieved by overwriting the local core time with the local

[4] The sender core_id is omitted from the parameters to shorten the messages. Since we use a relative addressing, the target core determines the message source by calculating the inverse of the `core_id`. This approach is operational in all types of messages in which a response is expected.

FDU time plus an approximation of the duration, the message traversed through the communication network.

The `req_msg_id` field is filled with a unique identification number, which is used to identify the corresponding response message sent from a core. Therefore the requesting device will store the `req_msg_id` in its "Request Table", which is correlated with the target core and a timestamp.

The field `req_code` defines what a core is supposed to do, in order to satisfy the FDU's request. Since all FDU requests are handled by the core's probe, the thread execution will not be interrupted by this kind of message. However, the shutdown message will terminate the thread execution, due to shutting down the core by the probe.

The field `data` is a container, which holds the request information and may be used application specific. From FDU point of view, this container is used to hold the desired registers' entries that the FDU wants to read. To support maximal flexibility we defined three different coding schemes, which are shown in Table 1.

Payload section of data flits			**Description**
0	1	2 ... 31	
0	0	[register address value]	Request for one register value
0	1	[first register address value] [last register address value]	Defines a range of register addresses
1	0	[register address value #1] , ... , [register address value #n]	Defines several register addresses

Table 1: Definition of the payload structure: Request Message

4.2.2 Interface: Response

Response messages are the result of incoming request messages. Response messages should therefore be sent only if a request exists. Such requests can be the request messages of the pull mechanism or the initialising request message of the push mechanism resulting in periodically sent response messages.

Message format:

```
Response( priority, core_id, timestamp, req_msg_id, data )
```

Field Descriptions

The `priority` (0 = highest priority, ..., 3 = lowest priority) field is used to access the interconnection network with one of four different privilege levels. The usual value for this parameter is "1" because most of the requests should be answered in low latency. We propose to copy the priority from the request message, because at the point of request message creation, we defined already which priority the communication should have.

The `core_id` field identifies the message destination. Each `core_id` is unique and belongs to one core on the chip, which can be accessed over the interconnection network. The `core_id` consists of two coordinates of the 2D mesh and represent the place where the core is located. For a response

message a look up of the target `core_id` is not necessary, because it is copied directly from the request message field.

The `timestamp` is copied from the corresponding field of the *request message.*This will allow the FDU to estimate how long this communication took. A special case is the heartbeat message. Those are implicit responses derived from the massage type **heartbeat notification** and can also contain a timestamp. For all other messages types this field is optional.

The `req_msg_id` field is a copy of the corresponding field from the origin request message. It may be used as a control field for the FDU to determine, which request this response message answers (see also `req_msg_id` description in the section above).

The `data` field contains the requested core state information in form of register entries. From the FDU point of view, that means depending on the FDU request message, this field holds either a single register value or a set of register values. The order of this set is defined by the request message from the FDU. The requested data has to be written in the same order to the data field.

4.2.3 Interface: Alert

An alert message is used to send urgent messages from a core to the FDU. The alert message contains just a targeted ID (`core_id`) of the monitoring FDU and the alert code (`error_type`).

Message format:

```
Alert( core_id, error_type )
```

Field Descriptions

The `core_id` field holds the unique id of the FDU, which is awaiting a response of the responding core. Likewise the request message the `core_id` is used for routing the message and determining the source of the message.

The `error_type` encodes which kind of problem the core is suffering from.

Depending on the router configuration and its routing decisions, this message may either be directly routed to the FDU or indirectly via broadcasting. If the communication network is still intact, direct routing may be reasonable. While the communication network suffers from broken links it may make sense to broadcast the Alert to ensure minimal latency.

Unlike the request messages, response messages, and notification messages the alert message has the priority 0 and will be handled with maximum priority.

4.2.4 Interface: Notification

The pseudo function call `Notify` defines a notification message from an FDU to the affiliated TSU within a cluster.

Message format:

```
Notify( priority, core_id, timestamp, data )
```

Field Descriptions

`Priority` is used to access the interconnection network or bus system with one of four different priority privileges. The usual value for this parameter is "0" (highest priority) because most of the requests should be answered with minimal latency. However, there might be some cases where a high priority is not necessary. Therefore we keep this option open and define the priority by hand.

The field `core_id` identifies the message destination. Each `core_id` belongs to one device on the chip, which can be accessed over the communication network. For a mesh based interconnection network `core_id` is used by the router to route the message to the correct device. If a bus system is used for the communication, the `core_id` is directly snooped by the devices connected to the bus.

The `timestamp` holds the FDU's local time at the moment of the message creation. It can be used to infer the response time of the target core. Furthermore, it may be used to gain a synchronous cluster time. Synchronous cluster time can be achieved by overwriting the local core time with the local FDU time plus an approximation of the time, the message traversed through the net communication network.

The `data` field contains the information that is sent from a cluster internal core, a cluster external FDU, or a TSU. The content of the information depends on the target device. A core may send core internal health information (provided by model specific registers, i.e. performance counter registers or machine check registers) to the FDU while the FDU uses this message type to send cluster health information (how many cores are still active and what are their performance capabilities) to the TSU or to the Service Node.

4.3 FDU Interfaces to Communication Partners

4.3.1 FDU to Core

The communication between the FDU and the cores concerns heartbeat messages enhanced with core-internal states (temperature etc.) and alert messages that can be done either with the pull or with the push mechanisms. Request with following response message pairs are used to implement the pull mechanism. For the push mechanism a single initialising request message is followed by periodic response messages.

Response messages contain core-internal states in its data field. Moreover alert messages containing error conditions can be sent asynchronously from the core to its associated FDU in urgent cases.

The core-internal techniques to read a request message and to generate responses and alert messages highly depend on the core to be used. In this stage of the project we assume a message handler activated by a request, reading the performance counter and generating response messages, running on the core.

4.3.2 FDU to TSU

The functionalities between the FDU and TSU are partitioned such that the FDU is responsible for the core and link level and the TSU is responsible for thread handling. Concerning fault tolerance that means, the FDU detects node and link failures and informs the TSU, whereas the TSU is responsible for recovery and thread restart. Moreover, thermal indications will be collected by the FDU and so the decision to reduce the core frequency to meet the thermal conditions will be implemented as part of the FDU, but the load balance that results from this decision will be implemented by the TSU.

The FDU communicates with the TSU via command messages i.e. request, response, notify, and alert messages. Please note that the communication between the TSU and the FDU can be bi-directional since the TSU may request the FDU to change the frequency of a core, or to shut it down in case of low workload, while the FDU needs to report the TSU on thermal and error conditions.

One of the most complicated events in the system is how the core indicates to the TSU that a not correctable fault happened. The TSU scheduled a thread to be executed on a core. In return, when the thread completes its execution, the core needs to send an indication that the execution was completed, and that it is ready to execute the next thread to its associated TSU. When an error occurs, we cannot trust the core to be able to send the indication of the failure to the TSU, but we assume that the error detection hardware on the core will trigger an alert signal indicating that something wrong happened. In order to avoid race conditions, as soon as the FDU receives the alert message on the fault, it sends the indication to the TSU that in return first sends an abort message to the core and second reschedules the thread to be executed either on the same core or on another core.

Please note that the current design of the partition of functionality between the TSU and the FDU may change while we start to further explore recovery mechanisms in the next phases of the project.

4.3.3 FDU to Service Node with Operating System

Similar to the communication between the FDU and the cores, the communication between the Service Node (OS) and the FDU can be done either with the pull or with the push mechanisms. If the push mechanism will be used, the FDU will submit its aggregated information about its cluster (associated cores, TSU and the FDU itself) to the OS, which is executed on the TERAFLUX Service Node. If the pull mechanism will be implemented (the preferred option) the OS will monitor the state of all FDUs periodically and save their statistics in managed tables. At a lower level the OS and the FDU may communicate by using a shared memory similar to the way PCI configuration area works. Regardless the exact mechanism being used (push or pull) we can assume that the OS has a table that represents the health state of each cluster. Please note that the OS do not have to care about the health state of each core, from its point of view, the granularity it deals with is clusters.

4.3.4 FDU to FDU

Even whole clusters can fail. The fault of a whole cluster can be detected by monitoring the FDUs, because a cluster consists of an FDU, a TSU and a number of cores. If the whole cluster fails, also it associated FDU has failed. Therefore, we devise techniques how FDUs are automatically organised in groups that monitor each other (future work in the second project year). The messages between FDUs and monitoring techniques will be similar or the same as between core and FDU. That will allow self-organised FDUs and cluster over the whole NoC.

Currently we assume that the OS on the Service Node periodically monitors the FDUs and detects faults of clusters.

5 Core-Internal Fault Detection

The TERAFLUX architecture is supposed to build upon commodity x86-cores, which even today incorporate limited hardware fault detection and performance measurement mechanisms.

With the aim to guarantee certain reliability at core level and to exploit common and well-tested built-in fault detection features, we encompass such build-in reliability features in our fault detection approach. In this manner, information provided by a commodity x86-core about its current state, which can be internally detected faults as well as key metrics about the current performance, are fundamental measures for our Fault Detection Units.

The rest of this section will focus on features and naming schemes of the AMD 11h processor family, but similar features can be found in processors of the Intel P6 family as well [18].

5.1 Machine Check Architecture

The AMD processor family 10 is equipped with an architectural subsystem called **Machine Check Architecture (MCA)** [9] that is able to detect and correct certain faults in the processor logic. The model specific registers used for the MCA are also readable and writeable via the CPU debugger interface within the SimNow simulator and therefore also provided by COTSon [19]. For AMD family 10 processors the machine check architecture can recognize faults in the data cache, the instruction cache, the bus unit, the load-store unit, the North-Bridge, and the reorder buffers.

Since most space on current chips is occupied by large memory arrays, it is likely that in the case of an error a memory cell will be affected, finally resulting in a detected fault.

In the case of a fault the MCA distinguishes between two different types of faults:

1. Correctable faults can be repaired and hence prevent the core suffering from data and processor state corruption.
2. Non-correctable faults let the processor remain in a corrupted state. Data corruption and wrong processor states are likely. This may makes a rollback necessary.

The number and the source of corrected faults are logged in model specific registers and the core can make progress during logging. By contrast, non-correctable faults can also be logged in a model specific register but at the end the processor cannot make further progress and usually generates a machine check exception.

Since frequent occurrences of correctable and non-correctable faults may be a direct indicator for intermittent or permanent faults, or a permanent breakdown of the whole core, the FDU can use this information to make predictions about the current reliability of a core.

5.2 Performance Sampling

Not all intermittent or permanent errors can be detected by the build-in machine check architecture encompassed in current processors, in particular errors that impair the data paths of a core's pipeline. On the other hand, such undetected errors may not affect per se the functional behaviour of the core.

They may only lead to performance degradation, if they affect complex performance accelerating components like branch prediction circuits.

To enable the TERAFLUX FDU to periodically test for intermittent or permanent faults, the model specific performance counters are exploited. Such performance sampling can be used in two different manners [10].

1. **Caliper mode**, in which the performance counter registers are read before and after an execution.
2. **Sampling mode**, in which an event counter register is preload with a threshold value. If the threshold is reached, an interrupt occurs and the performance counter registers are read by the interrupt service routine.

5.3 Low Level Interface of AMD Cores to FDU

We base our mechanism on the current fault detection and reporting mechanisms existing in AMD x86 cores but if needed, we will extend their interfaces.

Current AMD X86-based cores use different mechanisms to detect and report performance and errors, such as:

- At hardware level, most of the internal buses, memories and some of the logic is protected by error detection and correction mechanisms [17][9][5]. In most of the cases, the system can detect up to N simultaneously errors in a unit, but correct only up to M out of them (N≥M). When an error is detected and corrected, a special counter is increased to allow the system to track its health, but no other operation is triggered. When an error is detected without the ability to correct it, an exception is thrown to allow the exception handler (SW) to decide how to handle it.

- AMD cores (similar to Intel cores as well) define a set of performance, power, and thermal counters [10]. These counters can be operated in two modes of operations, (1) a polling mode, where the software (part of the FDU) can access these counters and read their values (request-response messages), and (2) in a triggered mode (push mode) [10]. Under this mode, the FDU may defines an interval for each counter. When the quota is exceeded, an exception or interrupt is issued to transfer the control to the software. We decided to base the FDU capabilities on these counters and extend them if needed.

We plan to define a set of “side-band” of alert signals that can be driven from the cores to the FDU so that the indication could be received even when the core is faulty and cannot issue the message for itself. Such signals include:

- Live signal: no forward progress, while the core needs to be active.
- Thermal limit exceeded: the core is too hot.
- Internal link liveness information.

Core-local Fault Detection Probe: Both, the MCA and the performance sampling, make use of interrupts in order to read the model specific registers [9]. Since frequent interrupts would degrade the dataflow performance of the core, we consider a lean hardware probe per core, which is directly connected to the model specific registers and the local interrupt controller. This allows the FDU to read (via request and response messages) model specific registers and receive fault related interrupts, which not impair the functional behaviour without interrupting the current dataflow thread. The fault detection probe may later also be used to calculate code signatures to enable the FDU to detect faults, which remain undetected by the core itself. Those faults would then be reported via alert messages to the FDU.

In the current stage of TERAFLUX project we see no reason to go deeper into the core-internal implementation.

6 Operating System Investigations

In order to support massive number of cores, organized into clusters, we assume that the system should provide, on one hand a uniform service to the programmer but on the other hand, it is not intended to implement a full-blown operating system such as Linux at each node. Thus, we assume that the system is divided into service node and TERAFLUX clusters. Heterogeneous HW will fit most to this assumption, but since our research does not focus on HW perspectives, we assume that all clusters are the same, where one (or more) cluster is running a full Linux kernel (service cluster), while the other clusters run a microkernel or picokernel such as L4 or NANOS. From the user point of view, all virtual shared memory of the application being executed can be accessed by all cores (with different access time for each) but there are services such as I/O system calls and more that can be executed only by the Service core. Thus the main focus of the OS related research is how this structure impacts the entire design of the system. For example, how the system boots, how work is scheduled, what is the minimal functionality the microkernel should have in order to fulfil the system requirements, etc., how the Service cluster communicates with all other clusters (and between the TERAFLUX cluster themselves and how to handle resources in the case of imbalance due to workload and/or faults.

In order to achieve that, we started implementing a prototype of a system that can run on the same chip both Linux and the L4 micro-kernel (in parallel we are also looking at the use of NANOS to replace the L4. The current system can demo the configuration then each microkernel emulates the TSU + FDU per cluster and the Linux demo the Service OS.

So far we were successful to

1. boot the system with different OS on each cluster under virtualization layer,
2. use a virtual shared memory to communicate between the different clusters (even if the physical memory is not physically shared), and
3. develop a communication layer between the Linux OS and the L4 micro-kernels.

The next step will be to start implementing the fault tolerant and resource allocation algorithms into the basic infrastructure. This step also depends on a few architectural decisions that need to be taken by the WP6, such as the overall memory model of the system and in particular the support we need to provide for transactional memory. Independent of these important decisions we will

1) continue to examine the best microkernel for the project (Currently we compare the NANOS and the L4.),
2) run the system w/o virtualization, and
3) start implementing load balance algorithms based on the health of different clusters.

As soon as we decide upon the open architectural issues we will start implementing the fault tolerance algorithms:

1) We will start with centralized algorithms.
2) Later we will distribute the algorithms (3rd year).

6.1 OS and Core/TSU Interface

The TSU is in charge for maintaining the execution of the TERAFLUX threads on the different cores belonging to the same cluster. At this point we assume that the main OS assigns the threads to be executed to the TSU and the TSU distribute the tasks among the cluster's cores. In order to maintain that, the TSU keeps ready queues to hold all threads yet to be executed. The TSU needs to take into account the information provided by the FDU in order to load balance between the different cores and in order to react when a fault is detected during the execution of a core. We examine the possibility that the TSU will be implemented as a hardware component or as a local micro-kernel like L4.

For the simplicity (and we may decide to change it later on), let assume that (1) we are using two phase commit; i.e. task has no side effects, it either complete its operation or can be reschedule for execution at any time, (2) a tasks are not pre-emptive; it is removed from the ready queue only when it finishes executing the task or if it requests a system call such as I/O, page fault, etc., and thus when an not correctable fault is detected, the system (HW) needs to flush all its internal states, and indicate to the TSU that the task execution terminated unsuccessfully. The TSU will try to re-execute the task on the same core or on a different core within the same cluster. Failing to execute the task on a different core may cause the TSU to ask the Service node, to execute the task elsewhere (most like at the service node itself.)

When tasks end its operation, the control goes back to the TSU with a successful indication, and the TSU needs to decide if to put the task at the wait queue, waiting for the execution of the operation at the service node or to terminate its execution.

7 Conclusion

This Deliverable started with a general specification of a Fault Detection Unit. The internal behaviour was derived from the field of autonomic computing encompassing monitoring, analysing, planning, and execution tasks. The then following design space exploration of different FDU variants resulted in a functional FDU specification using pull and push messages to detect core internal faults. The FDU specification also includes the abstract interface definitions between the FDU and its affiliated cores and TSU. Furthermore, we described the way the FDU will access the different cores to inquire their execution state and the way the OS will access the FDU in order to get the required information on the health of the cluster.

The Operating System was designed as Linux OS running on the Service Node of TERAFLUX and being connected to L4 kernels running on cluster nodes. FDU interfaces are defined that allow the OS on the Service Node to access the cluster information collected by the FDUs.

Thus all objectives of Task 5.1 were achieved.

As enhancements to the original objective of fault detection, we found out that the messages (response and alert) from cores to FDUs could carry additionally information on the temperature and internal fault rates provided by the Machine Check Architecture of AMD/Intel cores. Such information can be used to enhance the FDU functionality such that the FDU can react on such messages to proactively reduce voltage and frequency of the cores or to notify the TSU of partially faulty cores.

The work in the second project year will focus on the inter-cluster fault detection mechanisms based on the defined overall TERAFLUX architecture (as stated in Task 5.2). This concerns the three subtasks:

- Development of clustering of cores: We will develop strategies to form cluster which means that a set of cores is assigned to an FDU. The FDU is responsible to monitor all cores within its cluster.
- 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 the detection of faults of FDUs or whole clusters. After the detection of faults restructuring of clusters and groups may be necessary.
- 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.

We will start with a NoC design space exploration targeting to minimise the overhead of heartbeat messages within the NoC by applying different routing algorithms. We have to enhance the routing techniques to be more adaptive in case of faulty links or routers such that the cores can still be reached.

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.

It will 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.

Although the FDU will be optimized for a fast detection of faults, cases may occur in which even a rapid response of the FDU and TSU cannot prevent faults affecting other threads. This makes rollback and recovery mechanisms necessary incorporating the TSUs, which are in charge of the correct and consistent thread execution. For the future we plan to investigate two alternative recovery techniques[5] to handle such situations.

- One mechanism is based on "reliability by construction". This means, a thread is committed using a two-phase algorithm. In the first phase, the system can terminate the execution at any point of execution, since it keeps its state, before the execution is started and all writes are written to local buffers at that phase. In the second phase, the system guarantees to complete the update operation as it was executed in an atomic manner.
- The second mechanism is based on an "optimistic approach" that assumes that the system rarely fails. Thus, this mechanism relies on the periodic stored system states. A rollback brings the system, or parts of it, back in a stable condition. On the one hand the rollback may affect the overall performance. On the other hand, since the low probability for an occurrence of a fault is assumed, in most cases a rollback might be cheaper compared to start over the whole execution of the involved threads.

Additionally, we will consider for the future the evaluation of dynamic temperature and voltage variations. Hence, we will investigate how dynamic voltage and frequency scaling (DVFS) managed by the FDU influences the reliability and the performance of the system.

[5] Recovery is partly objective of Task 5.3 in year three.

8 References

[1] S. Borkar, "Thousand core chips," *Proceedings of the 44th annual conference on Design automation - DAC '07*, San Diego, California: 2007, p. 746.

[2] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers, "The Impact of Technology Scaling on Lifetime Reliability," *IN DSN (2004), IEEE, IEEE COMPUTER SOCIETY*, 2004, pp. 177--186.

[3] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, 2005, pp. 10-16.

[4] D.J. Sorin, "Fault Tolerant Computer Architecture," *Synthesis Lectures on Computer Architecture*, vol. 4, 2009, pp. 1-104.

[5] R. Iyer, N. Nakka, Z. Kalbarczyk, and S. Mitra, "Recent advances and new avenues in hardware-level reliability support," *Micro, IEEE*, vol. 25, 2005, pp. 18-29.

[6] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, "Ultra low-cost defect protection for microprocessor pipelines," *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems - ASPLOS-XII*, San Jose, California, USA: 2006, p. 73.

[7] T. Lehtonen, J. Plosila, and J. Isoaho, "On fault tolerance techniques towards nanoscale circuits and systems," Turku Center for CS (TUCS," *DEPT. OF INF. TECHNOL., UNIV. OF TURKU*, 2005.

[8] A. Mendelson, N. Suri, and O. Zimmerman, "Roll-forward recovery: the bidirectional cache approach," *Fault-Tolerant Parallel and Distributed Systems, 1994., Proceedings of IEEE Workshop on*, 1994, pp. 59-68.

[9] AMD, "Bios and Kernel developer guide," 2006.

[10] P.J. Drongowski, "Basic Performance Measurements for AMD Athlon™ 64, AMD Opteron™ and AMD Phenom™ Processors," Sep. 2008.

[11] M. Palesi, S. Kumar, and V. Catania, "Leveraging Partially Faulty Links Usage for Enhancing Yield and Performance in Networks-on-Chip," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, 2010, pp. 426-440.

[12] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer, "A new adaptive accrual failure detector for dependable distributed systems," *Proceedings of the 2007 ACM symposium on Applied computing*, Seoul, Korea: ACM, 2007, pp. 551-555.

[13] S. Weis, A. Garbade, S. Schlingmann, and T. Ungerer, "Towards Fault Detection Units as an Autonomous Fault Detection Approach for Future Many-Cores," *ARCS 2011 Workshop Proceedings*, VDE, 2011, pp. 20 - 23.

[14] Ching-Tien Ho and L. Stockmeyer, "A new approach to fault-tolerant wormhole routing for mesh-connected parallel computers," *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, 2002, pp. 48-56.

[15] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, 2003, pp. 41-50.

[16] H. Schmeck, "Organic Computing - A New Vision for Distributed Embedded Systems," *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, IEEE Computer Society, 2005, pp. 201-203.

[17] W. Chen, S. Toueg, and M.K. Aguilera, "On the Quality of Service of Failure Detectors," *IEEE TRANSACTIONS ON COMPUTERS*, vol. 51, 2000, pp. 561--580.

[18] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2."
[19] AMD SimNow™ Simulator 4.4.1 User's Manual, 2007, Rev. 1.83