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

PROJECT NUMBER: 249013

Exploiting dataflow parallelism in Teradevice Computing

D5.3 – Development of Intra-Cluster Fault-Detection and Recovery Mechanisms and Dynamic Adaption

Due date of deliverable: 31st December 2012
Actual Submission: 20th December 2012

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	**04.10.2011**	**Sebastian Weis**	**UAU**	**Initial document**
0.2	**07.11.2012**	**Arne Garbade, Sebastian Weis**	**UAU**	**Merged sections**
0.3	**14.11.2012**	**Arne Garbade, Sebastian Weis**	**UAU**	**Revised sections**
0.4	**16.11.2012**	**Arne Garbade, Sebastian Weis, Avi Mendelson**	**UAU and MSFT**	**Merged section from Partner MSFT**
1.0	**22.11.2012**	**Arne Garbade, Sebastian Weis, Avi Mendelson, Bernhard Fechner, Theo Ungerer**	**UAU and MSFT**	**First complete version**
1.1	**30.11.2012**	**Pedro Trancoso, Alberto Scionti**	**UCY and UNISI**	**Internal review process**
1.2	**03.12.2012**	**Arne Garbade, Sebastian Weis**	**UAU and MSFT**	**Integrated comments from internal review**

Release Approval

Name	Role	Date
Sebastian Weis, Arne Garbade	**Originator**	**05.12.2012**
Theo Ungerer	**WP Leader**	**05.12.2012**
Roberto Giorgi	**Project Coordinator for formal deliverable**	**08.12.2012**

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, Bernhard Fechner, Theo Ungerer
University of Augsburg

Avi Mendelson
Microsoft Research and Development

Glossary

D-FDU	Distributed Fault Detection Unit
DF-Thread	A dataflow thread
D-TSU	Distributed Thread Scheduler Unit
FM	Frame Memory
L-FDU	Local Fault Detection Unit
L-TSU	Local Thread Scheduler Unit
MAPE	Acronym for Monitoring, Analysing, Planning, and Executing
MCA	Machine Check Architecture
Leading Thread	Represents the main 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
Trailing Thread	Represents the duplicated thread in the double execution approach
QoS	Quality of Service
HTM	Hardware Transactional Memory
ECC	
TREAD	T* instruction for reading from Frame Memory
TWRITE	T* instruction for writing to a Frame Memory
TSCHEDULE	T* instruction for scheduling a new DF-Thread

TDESTROY	T* instruction for destroying the context of the calling DF-Thread
DF-Thread	Data Flow Thread (in TERAFLUX style)
Cluster	Group of nodes
NoS	“nano or pico”-kernel
Code Memory	Memory region containing the thread’s code
Frame Memory	Memory region allocated to each DF-Thread for dataflow-style communication
Thread Local Store	Private memory region of each thread

Executive Summary

This deliverable reports on the research carried out in the context of DoW Task 5.3 (project months 25 -36) "**Development of Intra-Cluster Fault-Detection and Recovery Mechanisms and Dynamic Adaption**":

- We enhanced the TERAFLUX architecture to support Double Execution of Dataflow Threads and implemented it in the TERAFLUX simulator.
- We implemented the thread restart mechanism to support recovery on thread level. Furthermore, we extended thread restart recovery to dataflow threads using Transactional Memory and OWM.
- We developed efficient mechanisms based on heartbeat messages to locate faults within the interconnection network.
- We defined the hierarchical architecture of the TERAFLUX operating system and its implications on the dynamic adaption of its fault tolerance mechanisms.

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

1 Introduction

The tera-scale level transistor integration capacity of future TERAFLUX devices will make them orders of magnitude more vulnerable to faults. Without including mechanisms that dynamically detect and mask faults, such future devices will suffer from uneconomic high failure rates. In Work Package 5, we focus on reliability aspects on four levels within the TERAFLUX architecture, to assemble a reliable system out of unreliable components. These levels are the cores, the nodes, the interconnection network, and the operating system.

In detail, the DoW for Task 5.3 defines three subtasks:

1. "*Development of intra-cluster fault detection mechanisms: We will design mechanisms to monitor the cores by FDUs in cooperation with the core-internal fault detection. Also, we will refine the interfaces between cores and FDU with the previous results from Task 5.1 and Task 5.2. The fault information obtained by the FDUs will be sent as feedback to the system.*"
2. "*Development of fault recovery mechanisms: In this task, we will analyse the monitored information for rating the reliability of supervised cores. We will also develop recovery strategies and trigger recovery actions in this task.*"
3. "*Add dynamic adaption of mechanisms: i.e., extend the model to include different mechanisms and dynamically change between them to allow smooth adaption of the mechanisms to achieve different targets*".

On the intra-cluster (core and node/cluster) level, we focus in this deliverable on the "*development of intra-cluster fault-detection and recovery mechanisms and dynamic adaption*". We refined the Double Execution of dataflow threads in the TERAFLUX architecture and the implications for the architecture. We further specified a thread restart recovery mechanism, leveraging the side-effect free semantic of the dataflow execution model. We also propose solutions to support thread restart recovery for dataflow threads using Transactional Memory and Owner Writable Memory. Finally, we propose a node recovery mechanism to support longer checkpoint intervals.

On the interconnection level, we present efficient methods to localize faults within the network (router and link). The localization technique utilizes the knowledge of the existing Heartbeat messages and extracts inherent information from them.

On the operating system level, we show how to provide the collected fault data of the D-FDUs to the operating system and propose mechanisms to dynamically adapt the system.

1.1 Document structure

Development of intra-cluster fault detection mechanisms is covered in Section 2, where we present a detailed description of Double Execution exploiting the dataflow execution model and the required runtime extensions to enable the TERAFLUX architecture to support redundant execution of dataflow threads. This section further describes the thread restart recovery mechanism incorporating transactional memory and owner writable memory.

Section 3 covers fault localization in the interconnection network by exploiting timing and routing information of heartbeat messages. Additionally, we present a topology consideration, which was necessary to cope with the hierarchically clustered TERAFLUX architecture

Section 4 discusses the hierarchical structured TERAFLUX operating system and the dynamic adaption of the system under faults.

1.2 Relation to other deliverables

- Fault detection and monitoring mechanisms are described in D5.1 and D5.2.
- The TERAFLUX architecture and execution model is described in D6.1, D6.2 and D7.1.
- Fault injection techniques and COTSon implementation issues of proposed techniques are discussed in D7.3 and D7.4.

1.3 Activities referred by this deliverable

This deliverable refers to the research carried out in Task 5.3 – Development of Intra-Cluster Fault-Detection and Recovery Mechanisms and Dynamic Adaption.

2 Advanced Intra-cluster Fault Detection and Recovery Mechanisms

In Deliverable D5.1, we presented a general specification of the D-FDU and described different message types to monitor cores by the D-FDU. This general D-FDU specification was extended in Deliverable D5.2 to also support mutual D-FDU monitoring of FDUs in different nodes. In the current Deliverable D5.3, we focus on our advanced intra-cluster fault detection mechanism, which exploits the dataflow execution model to detect faults within the cores pipelines by using Double Execution of dataflow threads. Furthermore, we leverage the dataflow execution model, in particular its side-effect free and single-assignment semantics, for a local node checkpoint mechanism using thread restarts.

Figure 1 depicts the schematic view of the task distribution in a TERAFLUX node and illustrates the interaction between the different components. In the following, we will describe the main intra-node components from our fault tolerance point of view, namely the *D-TSU*, *D-FDU* and the *operating system*. We will particularly focus on the tasks, which are important for intra-node fault detection and recovery mechanisms.

As we described in Deliverable D5.1, the D-FDU is a small support component[1], which gathers health information of the cores within its node by using heartbeat messages, analyses and aggregates the information and reacts on it. Internally, the D-FDU operates in a Monitor-Analyze-Plan-Execute (MAPE) cycle [1], derived from the field of autonomic computing. For our Double Execution approach, we extended the D-FDU to also support CRC-32 signature comparison. Accordingly, the D-FDU does not only provide aggregated information about the cores health and wear-out state, but also uses event messages to inform the D-TSU about faults, detected by signature comparison in Double Execution Mode.

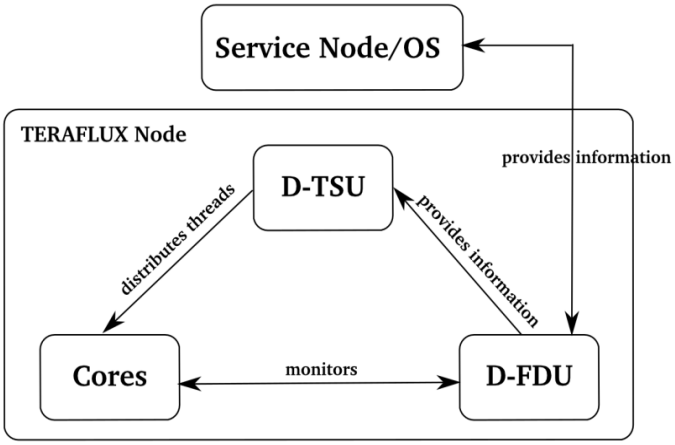

Figure 1 Schematic View of the Task Distribution between the D-TSU, the D-FDU, and the Service Node

[1] The D-FDU functionality could be implemented either as a hardware unit or a software unit that can be dynamically assigned to the cores.

The main task of the D-TSU is to provide hardware support for dataflow thread synchronization and scheduling. In order to double threads at runtime, it was necessary to modify the internal organization of the D-TSU (see Section 2.1.1).

The D-FDUs provide information to the various global Service Nodes, which host the TERAFLUX operating system. Since the TERAFLUX architecture supports a global linear address space, we use shared memory to distribute information between the Service Nodes' operating system and the D-FDUs per node. Similar to the communication with the D-TSU, each node's D-FDU will provide the aggregated node's health information to the Service Nodes. In order to do so, a Service Node reserves a special memory region per node. The D-FDUs will periodically update the node's health status by writing to this memory section. Accordingly, the Service Nodes periodically check those regions to update their global view of the system and dynamically react on changed conditions (see Section 4).

The remainder of this section will provide a detailed description of the required implementation enhancements for Double Execution of dataflow threads, which was initially described in Deliverable D5.2. We present first results, which show the runtime overhead of Double Execution and the thread restart recovery in the fault and the fault-free case.

2.1 Double Execution

The Dataflow Execution Model in TERAFLUX provides side-effect free execution of pure dataflow threads and single-assignment semantics for thread frames[2]. We leverage both inherent features of the execution model to provide a scalable and light-weight fault detection scheme for dataflow threads [2].

As we described in Deliverable D5.2, our Double Execution approach duplicates dataflow threads (DF1-Thread, see D6.1 and D7.1) at runtime. This makes the detection of faults transparent to the programmer and the compiler. In fact, the architecture can execute the same TERAFLUX program using Double Execution or regular dataflow execution. According to the definition given by Rotenberg for redundant execution on an SMT processor, we call the thread that is duplicated *leading thread* and its duplicate *trailing thread* [3].

In the TERAFLUX architecture, Double Execution works as follows:

1. A thread is duplicated when its synchronization count becomes zero, i.e. a thread has received all its inputs and is ready to execute. To indicate the thread's duplication, the L-TSU sends notification messages to the D-TSU and the D-FDU. The D-TSU, which keeps all threads' continuations of its node in a Thread-to-Core List (TCL), creates a copy of the leading thread's continuation, distributes it to another core's L-TSU and stores the new continuation in the TCL. The thread distribution is limited to the cores affiliated to the D-TSU. However, a leading thread and a trailing thread never share the same core. This rule is enforced by sending the specific continuation to different cores within the node. The respective L-TSU proceeds with the execution of the leading thread as usual.

[2] The Double Execution approach described here is based on the T* instruction set extension [14] and the DTA-C [15] flavored execution model of TERAFLUX; however, it can be easily ported to other threaded dataflow execution models like DDM.

2. During execution, each L-TSU buffers all TWRITEs of the leading thread (for more details see Section 2.1.1) in a core-local write buffer. Simultaneously, the L-FDU creates a CRC-32 signature of all TWRITEs, incorporating the target thread id, the target address, and the data. The L-FDU of the trailing thread's core also creates a CRC-32 signature of all TWRITEs; however, the TWRITEs of the trailing thread are discarded after the signature has been created. Other write operations to the thread local storage (heap or stack) do not need to be buffered, since these writes will be automatically repeated by a thread restart.

3. When a thread has finished execution, indicated by a TDESTROY instruction, the core's L-FDU sends the CRC-32 signature to the D-FDU.

4. The D-FDU waits for the signatures of both the leading and the trailing threads, compares them and informs the D-TSU about the result.

5. In the case of a non-faulty execution of both threads, the L-TSU[3] redirects the buffered writes of the leading thread to the D-TSU, which commits them to the main memory and reduces the synchronization counts of successor threads. Finally, the responsible D-TSU subsequently deletes the continuations of the leading and the trailing thread in its TCL. If a fault was detected, the D-TSU instructs the L-TSU to flush the core-local write buffer and discards all continuations created by the faulty thread.

Redundant execution of dataflow threads promises the following advantages over lock-step architectures [4]:

1. Result comparison is restricted to data, which is consumed by subsequent threads.
2. Result propagation is only required when a thread has finished execution, which inherently supports deferred memory updates.
3. Redundant thread execution is synchronized at thread level. This enables the D-TSU to exploit the scalability of the dataflow model for both the leading and the trailing thread. In particular, the D-TSU scheduler can take advantage of under-utilized cores.

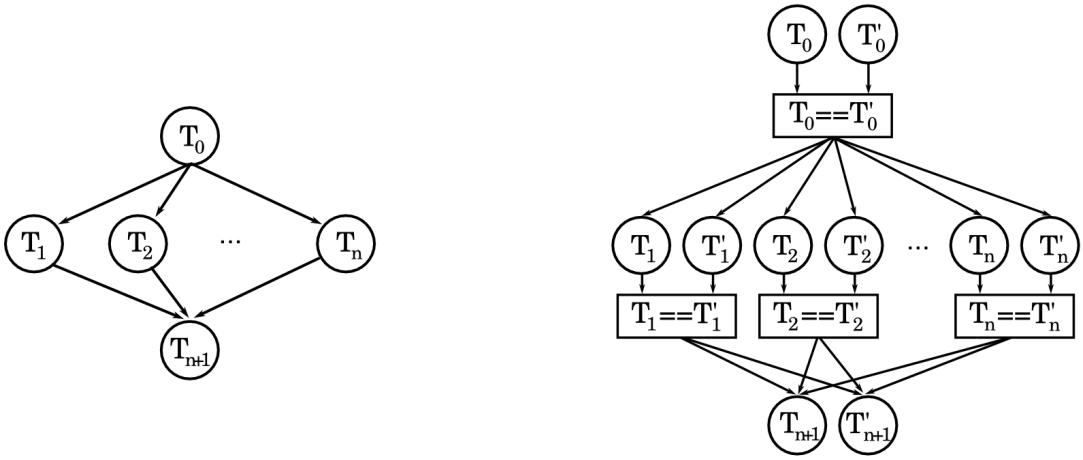

Figure 2 Dependency graph for regular dataflow execution (left graph) and double execution (right graph)

[3] **This is a variant of the T*-TSU**, which is described in WP6, extended for fault tolerant execution.

Figure 2 shows the dependency graph of a T* program (left) and the dynamically created dependency graph of the same program during a Double Execution run (right). It can be seen that the original program first executes T_0. Since this part of the program is sequential, Double Execution may exploit under-utilized cores for spatial redundant execution of T_0', if the system has at least more than one core. After the results of T_0 and T_0' are compared, the synchronization counts of the subsequent threads are decremented, and the subsequent threads (T_1T_1',..., T_n T_n') can be started. In this case, the TERAFLUX thread scheduler will try to execute as many redundant thread pairs (T_n and T_n') as possible in a spatial redundant way (see Figure 3, left). If it is not possible to schedule all waiting threads to idle cores, since the thread-level parallelism of the original program was already able to utilize all cores, the scheduler will execute the threads in a temporal redundant[4] and a spatial redundant way (see Figure 3, right).

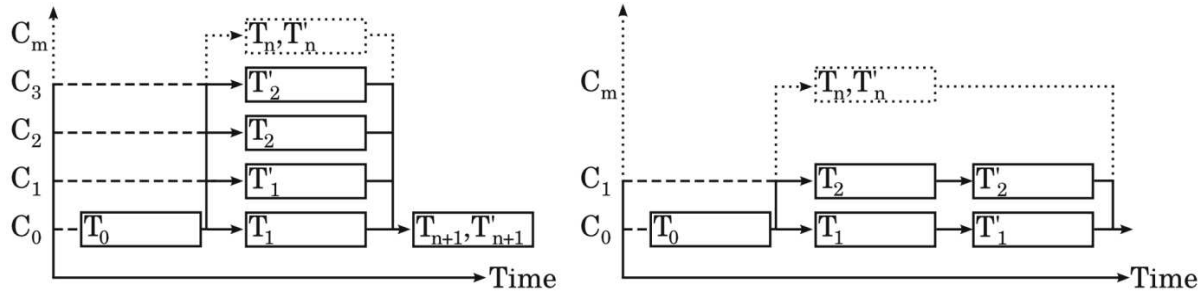

Figure 3 Spatial Redundancy (left) and Temporal Redundancy (right) of Double Execution

2.1.1 Runtime Extensions

Double Execution of dataflow threads required enhancements of the TERAFLUX runtime system. In this section, we will describe the necessary extensions to the D-TSU and the cores.

2.1.1.1 D-TSU enhancements

Node-wide thread management and scheduling are controlled by the D-TSU. The D-TSU maintains continuations, which are per-thread control structures, keeping pointers to the Code Memory, Frame Memory, and the Thread Local Storage (see D6.3, D7.1 and D7.3). Because of the TERAFLUX execution model, the thread execution is side-effect free. Therefore, from a D-TSU point of view, we only need to duplicate the thread continuation in the D-TSU to create a trailing thread. The architecture guarantees that the code memory and the frame memory will never change during thread execution. The Thread Local Storage, however, is unique to each thread, and newly allocated with each continuation. The Double Execution mechanism takes advantage of the execution model, since this way only TWRITEs release thread results to the global system state and hence need to be incorporated in the CRC-32 signature calculation.

Figure 4 shows the original and the doubled continuation, with the additionally allocated thread local storage for the trailing thread. We added the following fields to the original continuation specification:

- RED_CONT (Redundant Continuation Pointer): Stores the thread ID of the redundant continuation.

[4] This assumes that the cores have enough resources to buffer all TWRITEs.

- SPEC (Speculative) bit: Indicates, whether this thread is speculative.
- PARENT_ID: Stores the thread ID of the parent thread. Required to discard the created speculative continuations inside the D-TSU, if the parent thread needs to be restarted.
- TRAIL bit: Indicates, whether this thread is a trailing thread.
- ID: Unique identifier of a thread.

Please note, that the synchronization count of the redundant thread is always zero, since the redundant continuation is only created, when the synchronization count of the original thread became zero.

As mentioned before, the trailing thread is handled by the D-TSU scheduler like a usual thread, with the exception, that redundant threads will never be scheduled to the same core, to enable the detection of both permanent and transient faults. Furthermore, trailing threads are never moved to other nodes to ensure a fast result comparison by the node's D-FDU.

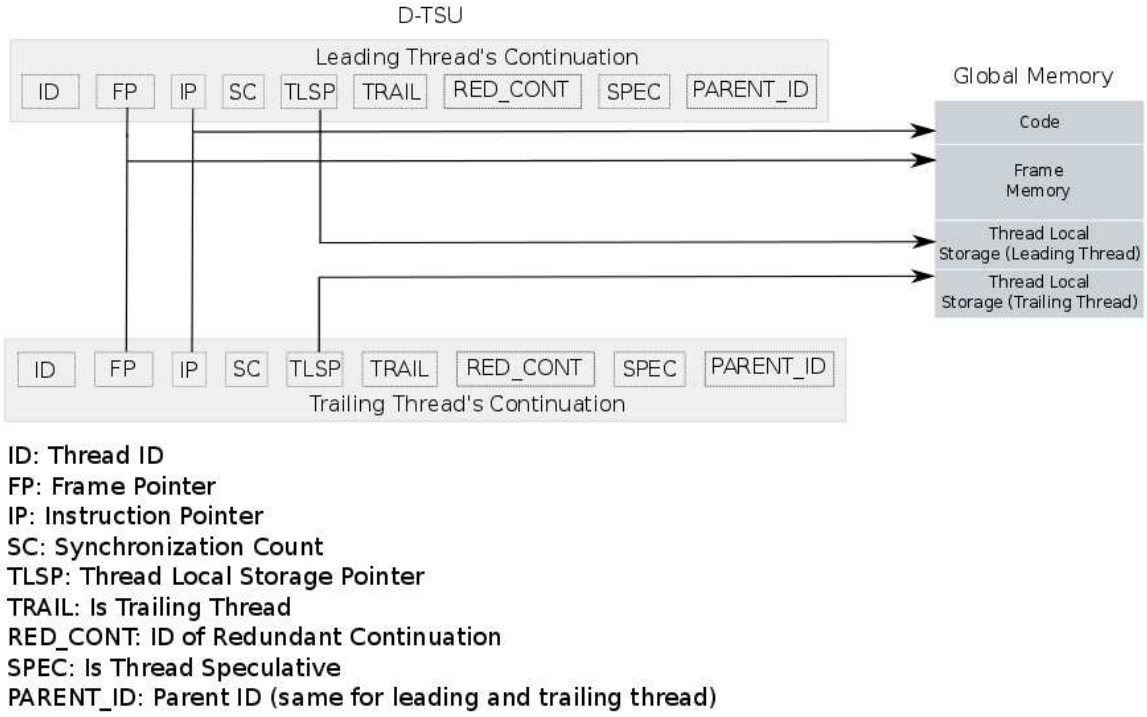

Figure 4 Extended Continuations for Double Execution (Owner Writable Memory – or OWM – and Transactional Memory pointers are omitted for simplicity)

Although, dataflow threads (DF1-Threads) do not need to wait for input data after the synchronization count became zero, they can comprise TSCHEDULE instructions to schedule subsequent dataflow threads. If the D-TSU receives a TSCHEDULE request from an L-TSU, a new continuation is created and a new frame memory, required to store the thread's data, is allocated. The D-TSU finally returns the ID of the newly scheduled thread. However, the returned ID is runtime dependent and may be later passed to subsequent threads by TWRITE instructions. To ensure deterministic write sets of both threads, the returned IDs must be the same for the trailing and the leading thread. Since the thread execution is not synchronized on instruction level, it may happen that the leading thread runs behind the trailing thread or vice versa. In order to prevent stalls induced by TSCHEDULE synchronization

between the redundant threads, we let both issue TSCHEDULEs. To prevent redundant thread scheduling, the D-FDU maintains a counter per continuation, which keeps the number of issued TSCHEDULE instructions. When a thread issues a TSCHEDULE request, the D-TSU compares the TSCHEDULE count in its continuation with the continuation of the redundant thread. However, if the TSCHEDULE count is greater than the TSCHEDULE count of the redundant thread, the D-TSU knows that this thread is running ahead of its redundant copy. In this case, the D-TSU processes the TSCHEDULE request as usual and stores the scheduled thread ID in a table. If the TSCHEDULE count is lower than the TSCHEDULE count in the redundant continuation, the D-TSU has already processed this TSCHEDULE request from the redundant thread, which is running ahead. In this case, the D-TSU proceeds with a table look-up to retrieve the previously stored thread ID, which was created by the TSCHEDULE of the redundant thread.

Furthermore, all continuations created by TSCHEDULE instructions are initially marked as speculative. This is necessary for the thread restart mechanism, because the D-TSU must ensure that all continuations created by a faulty thread can be discarded. In order to do so, the parent thread IDs are stored with each continuation. In the case of a thread restart, the D-TSU traverses all continuations and deletes all continuations created by this thread. Furthermore, the D-TSU releases the allocated frame memories of these threads. Please note, that the D-TSU will not schedule a thread to a core, when its continuation is marked as speculative. If the D-FDU confirms the fault free execution of the thread, the D-TSU will mark all successor threads as non-speculative.

2.1.1.2 Core-level enhancements

Although, the TERAFLUX execution model provides side-effect free execution, it must be preserved by the underlying architecture. From a fault tolerance point of view, we must pessimistically assume that TWRITEs issued by a core may contain errors in the target thread ID, the target memory address, and the data. In order to avoid that errors can modify the global state of the system, i.e. manipulating the synchronization count of a wrong thread or over-writing data at the wrong address, we propose to use a core-local ECC protected write buffer, similar to the write buffers used in [5] for hardware transactional memory with pessimistic version management.

The write buffer is managed by the L-TSU. When the system runs in fault tolerance mode, the L-TSU buffers all TWRITEs issued by a core in its core-local write buffer. The write buffer keeps all TWRITEs until the L-TSU receives a message from the D-TSU to commit the buffered TWRITEs. The L-TSU then forwards all TWRITEs to the D-TSU, which processes them in the usual way, i.e. stores the TWRITE data in the target threads' frame memories.

As mentioned, the L-TSU permits only to the leading thread to issue persistent memory accesses (TWRITEs). In particular, the write buffer and the speculatively created continuations in the D-TSU ensure the side-effect free execution of the dataflow threads and prevent subsequent threads to consume wrong results. Furthermore, the L-FDU attached to each core creates a CRC-32 signature, incorporating the target thread IDs, the addresses and data of all TWRITEs, similar to the fingerprint technique proposed by Smolens et al. [6].

The L-FDU also creates a CRC-32 signature of the trailing thread's TWRITES. Finally, the signatures of both threads are sent to the D-FDU, which compares them.

2.2 Recovery Mechanisms

In Deliverable D5.2, we have described a first thread restart recovery mechanism, based on the single-assignment and side-effect free semantics of the TERAFLUX execution model. In this section, we will extend this thread-restart mechanism to dataflow threads using transactional and owner writable memory. Furthermore, we will present a recovery mechanism with longer checkpoint intervals in order to recover from core and node failures.

For the TERAFLUX architecture, we assume that all on-chip memory arrays (frame memory cache, local cache hierarchy, last level cache, and main memory) and all communication interconnects are protected by ECCs, which is already the case in most contemporary multi-core architectures. Accordingly, we consider the cores as potentially vulnerable to transient, intermittent and permanent faults.

2.2.1 Thread Restart of Pure Dataflow Threads

The advantage of the TERAFLUX execution model for fast fault recovery scheme is the side-effect free thread execution. Here, we exploit the dataflow execution model as well, which provides inherent execution checkpoints between pure dataflow threads (DF1-Thread). Hence, DF1-Thread can be restarted as long as their TWRITEs are not visible to the global system state and the effect of all previously issued TSCHEDULE instructions can be reverted. Since the side-effect free execution is supported by the TERAFLUX architecture through the core-local write buffers and the speculatively created continuations, the D-TSU can restart dataflow threads to recover from faults.

The cost for the thread restart mechanism during fault free execution depends on the size of the written data. The overhead is mainly induced by the deferred TWRITEs introduced with the core-local write buffers. Therefore, a thread accesses the global memory only when a TDESTROY instruction was called and the local-write buffer is committed. This means, during thread execution, there are no TWRITE accesses to the global memory. We assume that writes to the core-local TWRITE buffer have low latency, similar to first level cache accesses. However, after TDESTROY has been reached and the D-FDU confirms the fault free execution, the entire local-write buffer must be committed to subsequent thread frames and the threads' synchronization counts are automatically decreased.

The overhead when a fault has been detected is mainly determined by the wasted execution time of the recovered thread. Since we only compare thread results when TDESTROY has been called, the wasted execution time in turn mainly depends on the length of the thread. This means, although the recovery mechanism is transparent to the programmer and the compiler, its recovery capability is constrained by the length of the dataflow threads.

2.2.2 Thread Restart of Dataflow Threads with Transactional Memory

A pure dataflow execution model can provide high scalability and efficient recovery for certain programs, but also exhibits weaknesses in dealing with complex data structures like trees and graphs. To remedy these weaknesses, the TERAFLUX execution model provides Hardware Transactional Memory (HTM) to support shared memory communication between dataflow threads. Unfortunately, such shared memory communication can violate the side-effect free semantics of dataflow threads, since data committed by a transaction can be consumed by other transactions, even if the dataflow

threads have not finished their execution. This behavior is reasonable from a performance point of view, but breaks the thread restart recovery mechanism, discussed in the section before.

To preserve the inherent checkpoints of dataflow threads, even for dataflow threads with transactional memory, transactions are only allowed to commit, when the enclosing dataflow thread has finished execution. To check whether the thread has suffered from a fault, the D-FDU generates CRC-32 signatures of the TWRITES in the write buffer and of the write sets of the transactions, which are embedded in this thread. Only if the write buffer and the write sets of the transactions are proven to be fault free, the transactional memory controller commits the transactional write sets and subsequently the D-TSU commits the core local write buffer.

This restriction may result in larger critical regions and hence higher conflict rates and in the worst case serialized thread execution, but still guarantees fault containment on thread-level granularity. We will investigate the runtime overhead introduced by this restriction in project year 4.

2.2.3 Thread Restart of Dataflow Threads with Owner Writable Memory

From a recovery point of view, we consider the Owner Writable Memory (OWM) implemented on top of the Hardware Transactional Memory. Therefore, the HTM system is leveraged to support rollback of in place updates of the OWM. When a dataflow thread has finished execution, the transactional write set of the OWM transaction is checked, similar to the recovery for dataflow threads with transactional memory and the thread can be either committed or recovered using the TM version management.

2.2.4 Node Recovery Mechanism

Although, the thread restart recovery mechanism provides a way for low-overhead fault recovery, its checkpoint time is restricted to the length of a thread. When a thread's write buffer is committed to the main memory, we can no longer recover from a fault that happened before. To also establish thread-length independent checkpoint intervals, we developed a node global checkpoint mechanism on top of the thread restart recovery. This node checkpoint scheme also uses the core-local write buffers and the side-effect free execution.

The node global checkpoint mechanism works as follows:

1. The D-TSU can establish a checkpoint of its node's state after each thread's commit. To create the checkpoint, the D-TSU determines the start and end addresses of the current frame memory region in the node memory (We assume that the D-TSU can access the global address space, where the frame memory regions are mapped). Furthermore, the D-TSU creates a backup of its current context and stores it in the stable external main memory.
2. After the checkpoint has been established all subsequent TWRITES going to the checkpoint's memory region will be optimistically logged. This means, the D-TSU maintains a log of all changes to the thread frames within the checkpoint's memory region. Please note that newly allocated thread frames must not be recovered and are created outside of the checkpoint's frame memories.
3. When a fault is detected, the D-TSU recovers to the last checkpoint by restoring the frame memory log and its backup context.

4. Maintaining a new global checkpoint is done by updating the start and the end addresses of the current frame memory region and storing the current D-TSU context in the stable main memory. Finally, the log of the previous checkpoint is discarded.

Compared to global checkpoint mechanisms [5], [6], with this mechanism, we do not explicitly have to track the communication between the cores. Additionally, we only need to keep a backup of the current frame memory region, instead of maintaining a log of the whole main memory.

2.3 Preliminary Results

We evaluated our Double Execution and thread recovery techniques using the TERAFLUX dataflow simulator [16], which is based on HP's COTSon multi-core simulator (see D7.1, D7.2, D7.3 and D7.4). Therefore, we extended the baseline D-TSU implementation of the simulator to provide runtime support for Double Execution and thread restart recovery within the D-TSU. The goal of this evaluation was to investigate the overhead introduced by Double Execution in comparison with normal dataflow execution. Finally, we explored the overhead of the thread restart recovery without Double Execution, assuming a core-internal fault detection mechanism, under different core failure rates.

2.3.1 Simulation Methodology

All experiments were limited to one TERAFLUX node. The baseline machine assembles a contemporary multi-core processor with 4, 8, 16, or 32 cores, respectively. Each core consists of an out-of-order pipeline with 5 stages and a maximal fetch and commit width of 2 instructions per cycle. The private cache hierarchy of each core is comprised of separate 32kB L1 instruction- and data caches and a 256kB unified L2 cache. All cores have access to a 16MB L3 cache, exclusively used to store frame memories. The assumed memory bus latency is 25 cycles, while the memory latency is 100 cycles. Table 1 depicts the parameters of the baseline machine in more detail.

Table 1: Hardware parameters for the baseline machine

Parameters	Values
Cores	4, 8, 16, 32
Core Parameters	Out-of-Order, Pipeline length: 5, Fetch width: 2, Commit width: 2,
L1 I- and D-cache (private per core)	Size: 32kB, Line size: 64, Sets: 2, Hit Latency: 1 cycle, Write Through
Unified L2 cache (private per core)	Size: 256kB, Line size: 64, Sets: 8, Hit Latency: 6 cycles, Write Back
L3-cache (shared)	Size: 16MB, Line size: 64, Sets: 16, Hit Latency: 10 cycles, Write Back
Memory Bus Latency(L3-cache to memory)	25 cycles
Memory Latency	100 cycles
TWRITE Latency (write buffer)	3 cycles
TWRITE Latency(commit to memory)	30 cycles
TSCHEDULE Latency	40 cycles
TDESTROY Latency	40 cycles
D-FDU signature comparison Latency	30 cycles

We further assume that writing to the core-private write buffer and generating a CRC-32 signature takes 3 cycles per TWRITE. Committing one TWRITE instruction in the write buffer to global memory is supposed to take 30 cycles. This is assumed to be faster than a regular memory access, since committing the write buffer may take advantage of the DRAM's burst mode. Finally, for the TSCHEDULE instruction, we assume a latency of 40 cycles.

The simulated node was stressed with two T* benchmarks (available in the COTSon repository [16]): a parallel version of Fibonacci (fib), which recursively calculates the Fibonacci numbers, and a block-wise matrix multiply algorithm (matmul), which was manually ported from the StarSS matmul kernel.

fib is a modified version of recursive Fibonacci, which spawns three new threads in each recursive step. Nonetheless, fib recursively spawns new threads for (n-1) and (n-2) until n is equal or less than 28. If n is equal or less than 28, Fibonacci of n is calculated locally.

matmul is a standard block-wise matrix multiplication, in which the maximum thread-level parallelism is restricted by the number of blocks per matrix, while the length of each dataflow thread is determined by the block size.

Both benchmarks fully support the T* instruction set extension [14] and therefore guarantee a side-effect free execution. This enables the TERAFLUX simulator to restart and recover from all faults detected by Double Execution.

2.3.2 Double Execution

We simulated fib(36) and fib(40) with the goal to measure the impact of core utilization on the runtime overhead introduced by Double Execution. Here, fib(36) serves as a benchmark, which cannot completely utilize the node's cores. To show the increasing execution time with Double Execution, we determined the performance degradation, which is the Double Execution runtime normalized to the regular dataflow execution on the same node.

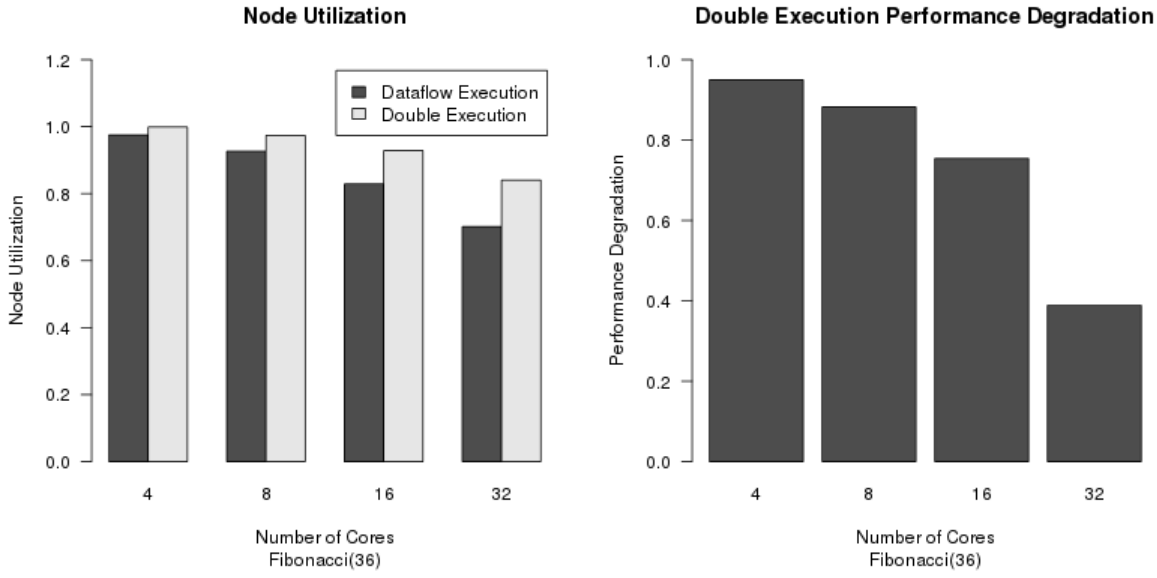

Figure 5 Node utilization (left) and performance degradation for Double Execution of Fibonacci(36)

The plot on the left side of Figure 5 shows the core utilization of regular dataflow execution and of Double Execution. It can be seen that fib(36) is able to nearly utilize 4 cores. However, with an increasing node size, fib(36) is no longer able to utilize all cores. The plot on the right side of Figure 5 depicts the performance degradation for the different node configurations. We can see that Double Execution can exploit underutilized cores to speed up Double Execution runtime. While the performance degradation introduced by Double Execution of fib(36) for 4 cores is 94.4%, the overhead for 32 cores is only 38.8%.

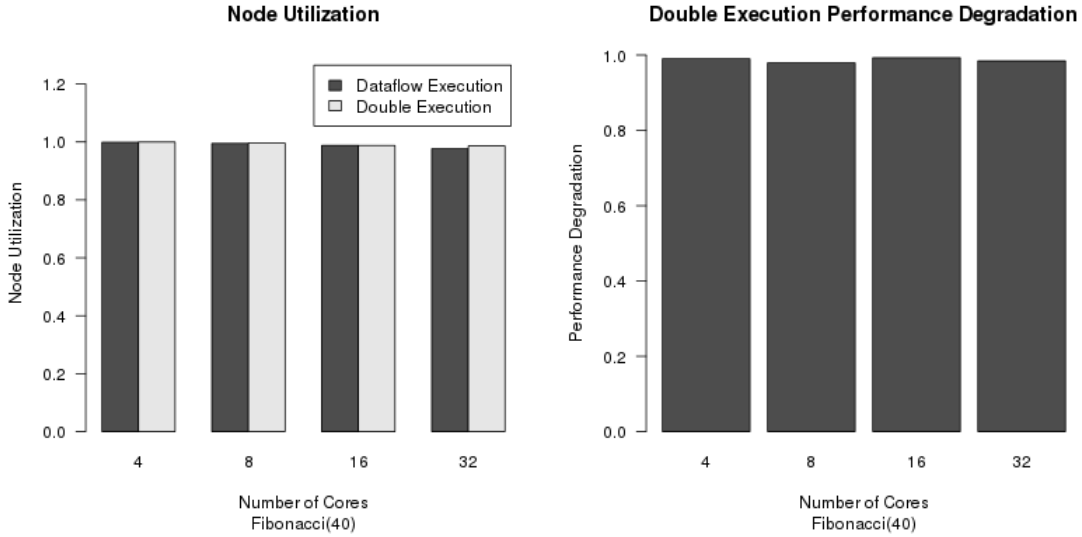

Figure 6 Node utilization (left) and performance degradation for Double Execution of Fibonacci(40)

The plot of fib(40) (see Figure 6) in turn shows that even if Double Execution cannot use underutilized cores (all configurations are utilized by 0.99), the performance degradation never exceeds 100%.

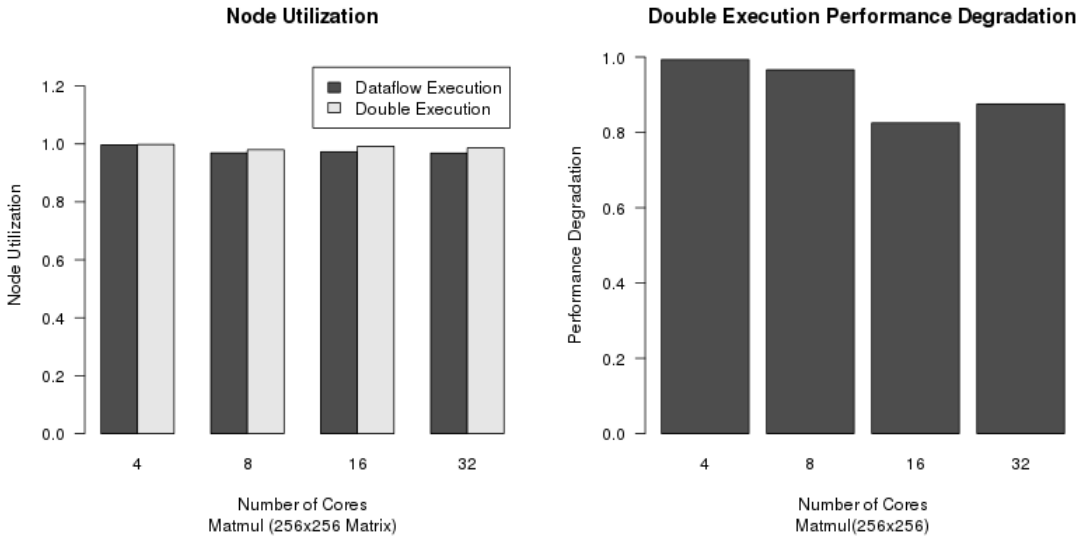

Figure 7 Node utilization (left) and performance degradation for Double Execution of Matmul (256x256 matrices)

Matrix multiply of two 256x256 matrices (shown in Figure 7) revealed that Double Execution may also benefit from the locality in the node's caches. Even though the benchmark is able to fully utilize all node configurations from 4 to 32 cores in dataflow execution, the performance degradation of Double Execution decreases with an increasing number of cores per node.

2.3.3 Thread Restart Recovery

We simulated all benchmarks also with a Mean Time to Failure (MTTF) of 0.1s and 0.01s per core. Since we assume a constant failure rate per core, MTTFs of 0.1s and 0.01 result in average failure rates of 10 failures and 100 failures per second per core. Please note that we do not use Double Execution here in order to measure the pure overhead introduced with thread restart recovery. We normalized the execution times to a failure free run on the same node.

For an MTTF of 0.1s (see Figure 8, left) it can be seen that Fibonacci suffers from a performance degradation of 10 to 20 percent. Furthermore, the performance degradation stays quite constant with increasing node sizes. The overall overhead for both Fibonacci benchmarks was 17.9%.

The performance degradation of matmul, however, decreases until a node size of 16. However, the performance degradation for 4 cores is considerably higher than for Fibonacci. The average overhead of all node configurations for the matmul benchmark was 27.9%.

By reducing the MTTF to 0.01s (see Figure 8, right), we can see a performance degradation between 150% and 220% for Fibonacci. The average overhead for Fibonacci was 187.4%. The average overhead for matmul was 185.5%.

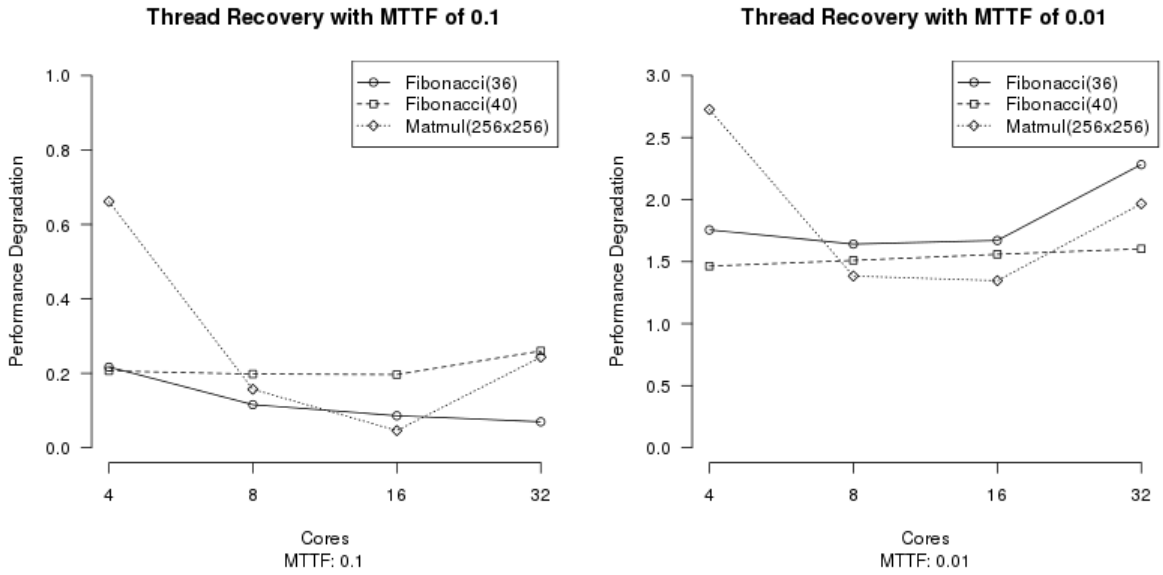

Figure 8 Performance Degradation due to Thread Recovery with MTTFs of 0.1s (left) and 0.01s (right)

3 Fault tolerance for inter- and intra-cluster interconnects

The focus of Sect. 2 was on fault-tolerance within a TERAFLUX node independent of interconnection considerations. Sect. 3 investigates fault tolerance aspects of inter- and intra-node interconnects. In Section 3.1, we proceed with the investigation of 2D mesh-based interconnects. In Sect. 3.2 we discuss several intra-node interconnection topologies and show that a 2D mesh-based intra-node network is preferable from a fault tolerance point of view.

Fault tolerance in the TERAFLUX architecture is not limited to its computing cores, memory controllers and I/O devices. Fault tolerance also expands over all interconnects of the TERAFLUX architecture and the respective controllers. The former fault tolerant tasks in WP5 regarding the Network on Chip (NoC) (D5.2 Section 3 and [9]) studied the implications for the network, when heartbeat messages are used to monitor the D-FDU's affiliated cores in a 2D mesh-based Network On Chip. We have shown how those messages increase the network message delays for application messages and how a mixture of routing policies can potentially reduce this effect.

However, there are some more methods available to raise the quality of heartbeat messages. One of those tweaks is to extract more insights out of the messages, by comprising information given by some D-FDU dictated restrictions. Those extracted information are used to additionally monitor the health state of the network without adding any data to the heartbeat messages itself. The health state can be used to enrich scheduling information and is furthermore needed in order to form logical clusters around a D-FDU. In the Section 3.1 of this chapter, we will describe in detail how we manage this NoC monitoring.

Furthermore, we investigated several interconnection network topologies in order to provide a reliable overall system. Section 3.2 discusses different hierarchical clustered interconnection network topologies. The hierarchical clustered architecture itself is specified in the TERAFLUX Architectural Template and was not yet considered from a reliable interconnect point of view. Section 3.2 concludes with a proposal of a hierarchical clustered architecture comprised of 2D mesh-based interconnects for both the inter-cluster communication and the intra-cluster communication.

3.1 Fault Information Feedback to the System (TSU and OS)

Thread management and fault tolerance for a one thousand core machine like our target system is a non-trivial duty. Task placement algorithms such as the ***Connectivity Sense Algorithm*** (described earlier in *D5.2, Section 3.3.3.2*) needs the pre-computed Task-Graph information and information about the system properties, which form the core graph. Such properties range from the current workload of the respective cores over the energy budget up to the health state of the monitored cores and the communication system. The latter has a special meaning for the optimal chip usage, since communication between the cores and the management units (such as the D-FDU and D-TSU) is a regular event. Also the access to the off-chip main memory, represents a not to underestimate part of the communication burden. Therefore, this property is an important measure in the management of tasks.

Two circumstances can influence the behavior of the communication network in terms of data throughput:

1. The workload to be handled by the network (including solving congestions).
2. The connectivity of the network.

The workload can be estimated by profiling the application's memory access behavior and by identifying cores communicating with other cores. This can be done offline at compile time by code analysis and also online by monitoring the actual task behavior. From the fault tolerance point of view, the network connectivity is the more challenging aspect of this property.

The heartbeat message-based monitoring provides a good opportunity for monitoring the NoC [10]. The health status of the NoC can be extracted from the heartbeat messages, since these messages inherently contain information about how they traversed through the net. However, there are series of slight adaptations needed to achieve 100% NoC monitoring coverage and different component faults.

We split this section into three parts. First we explain the monitoring technique based on a simplified example. The second part of the section considers the adaptations in order to gain 100% NoC monitoring coverage for our target system. The third part will discuss several adaptations to efficiently locate faults in the NoC with a minimal effort in hardware costs.

3.1.1 Basic Concept: Localization of faulty NoC-Components

While we described the health state monitoring of the cores in previous deliverables such as *D5.1* and *D5.2*, we now present two efficient methods for monitoring the communication system (Network on Chip - NoC). We will show how we are able to monitor the NoC by leveraging the heartbeat messages that we already use to monitor the D-FDU`s affiliated cores. For this, we combine the knowledge of the **timing information** about the message sending pattern (see: *D5.2 Section 3.2.2: Heartbeat Timing Pattern*) with the **message path information** derived from the routing strategy.

The timing pattern was originally designed to avoid collisions between heartbeat messages. In conjunction with the Quality of Service (QoS) it supports a precise estimation about the arrival time of all heartbeat messages. Supposing that a message was delivered with a delay, which is an indicator for an increased hop count, we can conclude that the message was transmitted using a bypass possibly because of a faulty network element. Figure 9 shows an example of how a broken link (representative for all NoC components) affects the timing behavior of a heartbeat message. In this case, the broken link forces the message from the C_2 to be bypassed and results in a delayed delivery at the D-FDU. The delay is then rated as a suspicious *heartbeat message timing behavior* and incidents of a faulty NoC component (assuming, the core sent the heartbeat message in head of time).

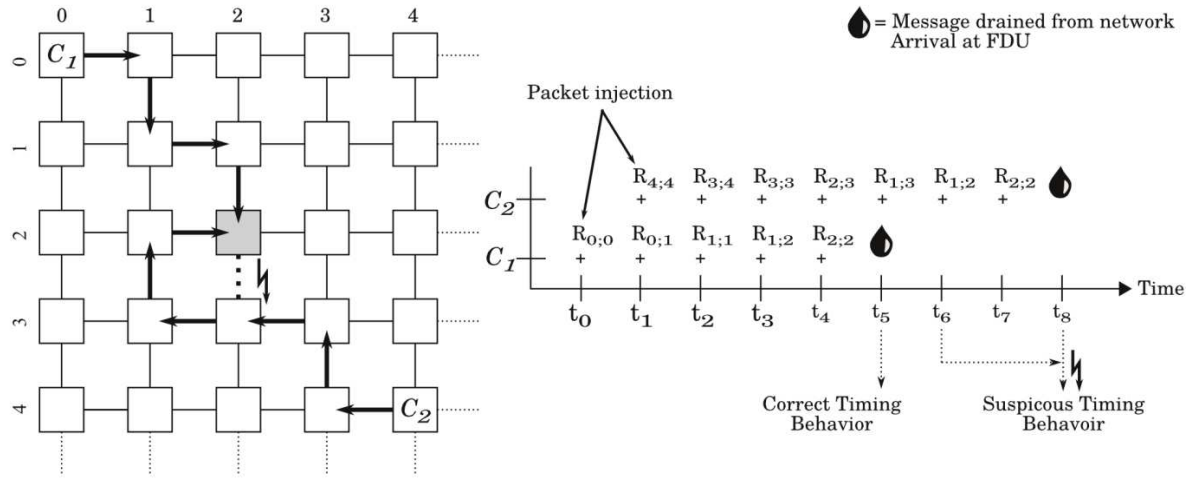

Figure 9 A link between two routers is broken (left), which results in a suspicious heartbeat timing behavior (right)

The routing information is used to determine where the faulty network component is located. Since the D-FDU investigates an unusual timing behavior of one or more heartbeat messages, the path of these messages (provided by the information from the routing strategy) is marked as suspicious. Therefore, the D-FDU is holding a network status matrix, which encodes the fault information of the NoC. The suspicious paths in the matrix are set by incrementing each entry in that matrix by "1" for each suspicious component. Each heartbeat message arriving in time at the D-FDU ensures that the values corresponding to the message´s path are decremented within the matrix. In that manner, a value of "0" within the matrix stands for a fault-free component and does not need to be further decremented.

If enough path information was gathered, the D-FDU will be able to identify the fault location within the network status matrix. The D-FDU then updates its own knowledge base (regarding the changed timing and routing information of the specific core) and forward this to the Distributed Thread Scheduler Unit (D-TSU). The update of the own knowledge base is an important step in order to detect further faulty components incorporating the bias of already known faults.

Many faults within the NoC, however, will not affect the delay of a heartbeat message transmission. This is due to the adaptive routing algorithm, which provides always an alternative link in case of faults or congestions. This alternative link does not necessarily increase or create a transmission delay. Indeed, only the utilization of a few faulty links will create a delay. Following the previous example, only those links in row 2 (horizontal) and column 2 (vertical) will have this effect.

This pitfall can be encountered by creating an artificial delay with a bypass of the faulty and the fault-free alternative link. That means the re-routed messages will be transmitted over a link that increases the distance to the D-FDU in the first place. The routing algorithm simultaneously avoids cycling messages and will route the heartbeat message again on the minimal path to the D-FDU.

3.1.2 Simplified Example

For simplification reasons within this example we consider a simplified NoC model that, in addition to routers, consists of simple bidirectional links between the routers. A link is pairwise shared among

two adjacent routers and allows communication in half-duplex mode. The current direction of transmission is continuously negotiated by the routers on a request basis. As we will show later, this simplification has no direct effect to the fault localization. The localization will work with pairs of unidirectional links as well. However, the simplification hides some gaps within the monitoring procedure, which will be addressed within this subsection.

Figure 10 depicts the two ***views*** of interest. On the left hand side we show how the system looks like in reality and on the right hand side we show the D-FDU`s knowledgebase about the health state of the NoC. Please note, that the sub-matrix $F_{x,y}$ encodes the reliability values for a link at a given router, where ***N*** stands for North, ***E*** for East, and so on. In the following we access a particular *suspicious value* of a link by $F^{D}_{x,y}$, where D is a set of directions $\{N, E, S, W\}$ and defines the direction of the link for the given router at *x* and *y*, respectively. In that manner the D-FDU will update the network status matrix pairwise, since for example the link $F^{S}_{0,0}$ and $F^{N}_{0,1}$ are identical.

We step into the example scenario right after the D-FDU collected the first delayed heartbeat message from C_2. As a response to that, the D-FDU starts marking all the components that were (normally) involved in transmitting this message. Hence, the network status matrix ***S*** of the D-FDU has the state as shown in Figure 10.

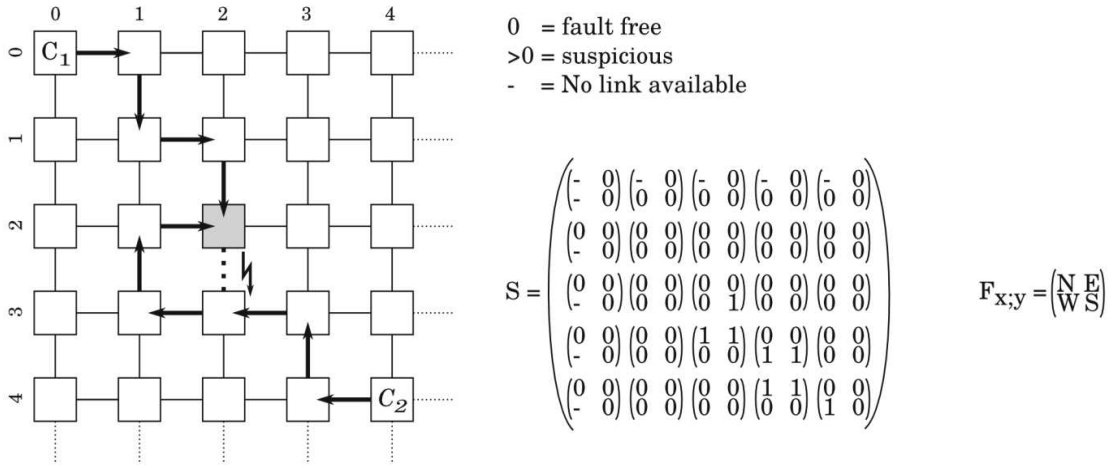

Figure 10 On the left: View on the physical system. On the right: The D-FDU's network status matrix after the arrival of the delayed heartbeat message from C2

Short time later, the D-FDU also collected heartbeat messages from the neighboring cores of C_2. This is depicted in Figure 11. As one can observe, the D-FDU incremented all components that are involved for the transmitting of delayed heartbeat messages and the network status matrix shows a first candidate for a faulty component, which is the link $F^{S}_{2,2}$ and $F^{N}_{2,3}$, respectively[5].

[5] For the sake of clarity we changed the heartbeat timing pattern in **Error! Reference source not found.** and **Error! Reference source not found.** compared to our former described timing pattern in D5.2 in order to demonstrate the localization procedure.

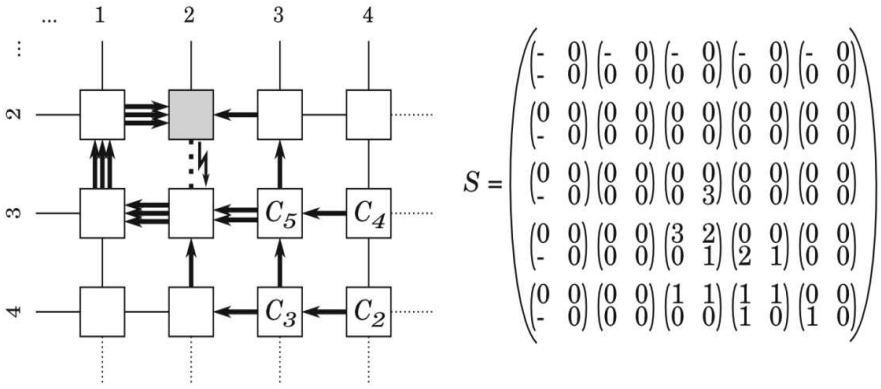

Figure 11 Internal view of the D-FDU's network status matrix S after five heartbeat messages had arrived at the D-FDU

After a complete round of heartbeat messages sent from all D-FDU affiliated cores the network status matrix is filled out with all suspicious paths and as shown in Figure 12. As one can observe, the former faulty candidate is most likely the one in the real world.

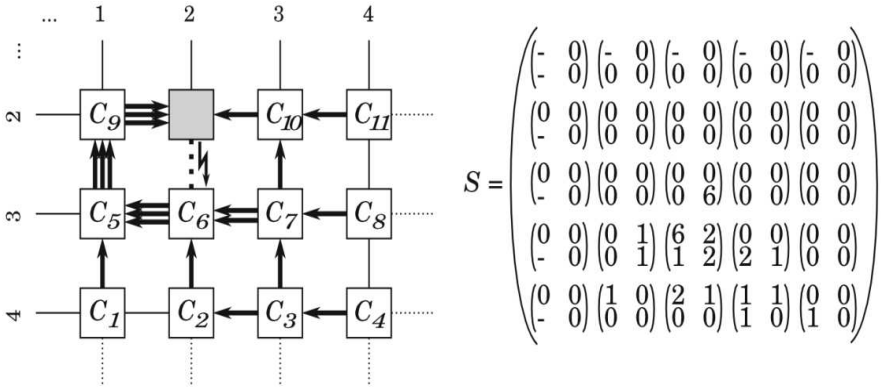

Figure 12 The D-FDU`s network status matrix S after a complete round of heartbeat messages.

3.1.3 Monitoring Gap

However, having a closer look on Figure 12 reveals an issue, which was not addressed yet and refers to what we call a monitoring gap. The link between $\mathbf{C_4}$ and $\mathbf{C_8}$ shows that this link has never been used by any heartbeat message. That means, this link has never been "monitored" and a link failure would not be recognized by the D-FDU. This example (also shown in Figure 13) represents a series of links within the cluster, which are not utilized by heartbeat messages. Additionally, there are links lying in between of two clusters. Those links have also never been monitored and need some additional attention.

The following two subsections discuss how we manage to fill both gaps by slightly expanding the heartbeat message based monitoring mechanisms of the D-FDU. At first, we will handle the gaps within a cluster and continue with handling the gaps in between of clusters.

3.1.3.1 Inside the cluster

To fill the monitoring gap within the cluster, we propose to simply alternate the routing for heartbeat messages triggered by a one bit flag in the head flit. This flag is set only by the sending core and follows a simple rule: The flag is set when the previous heartbeat message was sent without the flag (and vice versa).

At a given router, the flag is read and feed to the routing stage. If the flag was found, the routing logic alternates the direction and the alternate routing is set. That means for staircase routing (the underling routing policy for heartbeat messages) the routers route the first heartbeat message of a specific core following the staircase policy. The second heartbeat message of that core is then treated with the alternate version of staircase. Figure 13 and Figure 14 shows both the staircase policy and its alternate version. One can easily see that the alternate version shows in principle the same behavior as its "normal" counterpart but uses different links at the cluster borders. Of course, the flag needs to be kept until the D-FDU reads the messages, in order to know which routing algorithm was applied to transmit the heartbeat message. Alternatively, the D-FDU can assume the alternating routing algorithms for each heartbeat message. That means the first message of $\mathbf{C_2}$ would be sent using staircase routing and the second one would be transmitted following the alternate staircase version. Doing so, the flag does not need to be forwarded to the D-FDU. With this mechanism we are able to close the monitoring gap within the cluster.

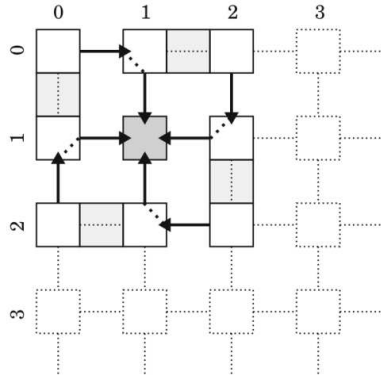

Figure 14: Revealing monitoring gaps using the original Staircase routing

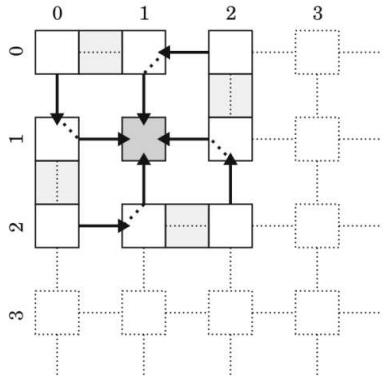

Figure 13: Revealing monitoring gaps using the alternate Staircase routing

The expected costs of realizing the alternating routing can be stripped down to the single bit flag within the heartbeat message and the extended routing logic. The latter one is the more expansive cost, since the additional decision rule needs to be implanted along with the XY-Routing and the original Staircase routing.

Since the header flits of heartbeat messages differ from the application´s header flits, the costs for the header flag arises only for heartbeat messages and is therefore negligible.

3.1.3.2 Between clusters

The other monitoring gap detects the uncovered links in between of clusters. At those points of the network no heartbeat messages utilize ever any links. In Figure 15 we highlight these links as gray boxes at the border of the two clusters α and β. Since the used routing policies do not allow paths that violate the minimum path rule, additional heartbeat messages are needed in order to fill this gap.

The main idea here is to overlap the cluster borders and thereby filling the monitoring gap. Therefore, clusters should be assembled with overlapping borders. In Figure 15 we show clusters how they were originally assembled. One can clearly observe the separation of cluster α and β. Aside others (gray boxes), we marked the not monitored links in between both clusters as *a*, *b*, and *c*. The D-FDU placement algorithm (described in D5.2 Section 3.3.3.2: Connectivity Sense Algorithm) now needs to place both D-FDUs α and β so that the cores α_a, α_b, and α_c will be affiliated to both D-FDUs and hence send their messages to both D-FDUs.

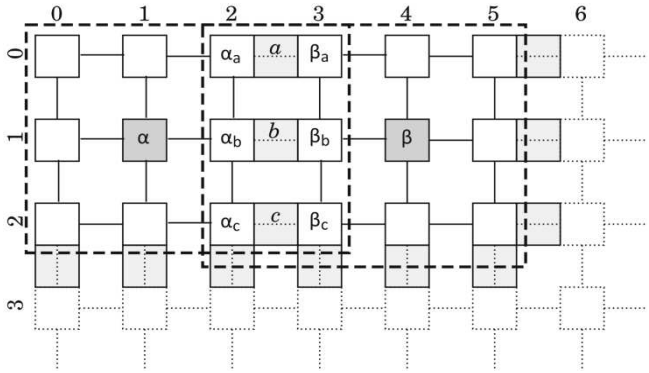

Figure 15: Overlapping D-FDU Clusters

The additional heartbeat messages are the only additional measureable costs we have to expect in order to fill this monitoring gap. The amount of costs for a given cluster can be easily calculated by $no_n + mo_m - f$ $n{\cdot}o_n + m{\cdot}o_m - f$, where *n* and *m* are defined as the node dimensions assuming a rectangular shape of a node. Assuming that *n* is the width of a node and *m* the height (from a 2 dimensional point of view), than *n* represents the northern and southern borders of a given node and *m* represents the eastern and western borders, respectively. The value *o* is a multiplier, which is used to "count" the number of cores, which are located at a certain border (distinguished by the dimension *n* or *m*). Since *n* and *m* represents respectively two borders of a node, o_n and o_m are defined as an integer value with $0 \leq o \leq 2$, where

- the value 0 means no core lies in an overlapped area for the given dimension,
- the value 1 means *n* (or *m*, respectively) cores lie within an overlapping area for the given dimension *n* (or *m*), and
- the value 2 means, all cores of a given dimension lie in an overlapping area.

The symbol *f* represents the amount of failed cores, which not send heartbeat messages anymore.

Taking Figure 15 as an example, the additional messages after expanding the node from FDU_α would be calculated by *(3*0) + (3*1) + 0* for the node with the FDU_α (respectively FDU_β).

At this point, we have a 100% coverage of all network components for the given network model. As we mentioned before, this example bases on a simplified NoC model for the sake of simplicity. If one extends the NoC model to a more actual one, one will see that shared bidirectional links are not the common case in today´s NoC implementation. The most NoC implementations found in open

literature consist of separate input interfaces and output interfaces, which raises another and the last monitoring gap.

3.1.3.3 Unidirectional Links

Re-considering the given example above, we pointed out that this example contains a simplified network model for explanation purposes. The actual router design of the most open literature implementation incorporates a pair of unidirectional links, connecting two adjacent routers or a router and its local core. One link of the pair is the transmitting port and the other the receiving port.

Now, in order to bring the network model to a more actual design, we need to address a final monitoring gap. As illustrated in Figure 16, all heartbeat messages sent from any affiliated core to the D-FDU results in the utilization of links "pointing" to the D-FDU only. Those links directed to the opposite direction are not affected from these messages, and are thus not monitored. Here again raises the need for an extension of our message based monitoring.

An effective and efficient solution is to switch from pushing heartbeat messages to the D-FDU, to polling them from the D-FDU side. In this case, the D-FDU sends heartbeat request messages to its affiliated cores, which in turn immediately answer the requests with heartbeat response messages. Additionally, all heartbeat messages (requests as well as responses) are handled by the routers with the alternating routing policy. The observation coverage of the network´s components are now again at 100%.

However, along with the increase number of heartbeat messages, a small pit-fall exists with polling heartbeat messages. A delayed response message of a core is now causing two paths within the D-FDU´s network state matrix S as suspicious. This is because the D-FDU cannot directly differentiate whether the request message was delayed and experiences the overall delay, or the response message of the core was delayed. According to the former given example, in Figure 16 we again step into the scenario right after the delayed response message of $\mathbf{C_2}$ has arrived at the D-FDU.

This time, the D-FDU incorporates the poll mechanism in order to alter the network state matrix ***S***. Please note that in this example unidirectional links are used. Therefore, the D-FDU updates the entries of network status matrix on a per link basis. In this case the sub-matrix $F^{D}_{x,y}$ returns the *suspicious-value* for the output link *D* of the router at *x* and *y*, respectively.

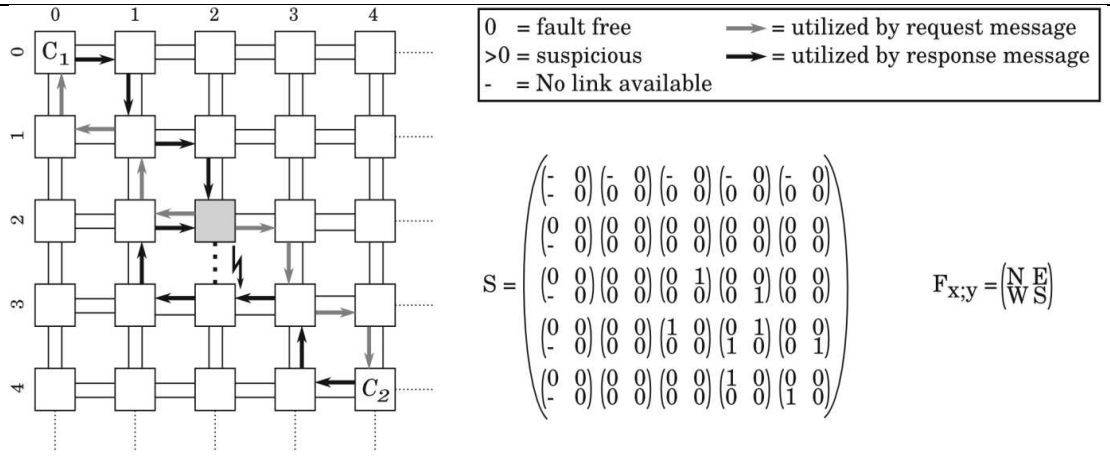

Figure 16: Link utilization separated in request messages (gray arrows) and response messages (black arrows)

Since both request messages and response messages use completely different network components, there is no chance to reduce this first uncertainty shown in Figure 16. Nevertheless, the localization mechanism will resolve this uncertainty and find the faulty component after collecting enough messages from cluster area. It may just need a little more time to sharpen the picture of the health state of the cluster.

A welcome side effect of using polling instead of pushing for heartbeat messages is the absence of costly configuration messages sent from the D-FDU to all affiliated cores. This configuration messages originally contains explicit instructions about the message sending timing pattern of the core (see *D5.2 Section 3.2.2: heartbeat Timing Pattern*). These messages need to be sent every time the system is started (e.g. at "power on") and in addition every time the D-FDU needs to change its position or re-arrange the cluster to react on faults. If polling is used instead, a single head-flit-only message is necessary per core and round. Where a round is defined as the amount of time needed to collect heartbeat messages of all affiliated cores.

3.1.4 Single Bit Utilization

As an alternative to the artificial increasing message delay for localizing faults within the NoC, there is another promising approach. The basic idea is using one additional bit as a flag to mark a re-routed heartbeat message. Contrary to the previous proposed approach, with this technique there is no need for an indirect way. As discussed above, the former approach creates an artificial indirection to produce a message delay, which indicates a faulty component of the message`s path. This artificial indirection results in at least in two additional hops of the message, even when a shorter path is available. However, since faulty links in row 2 (horizontal) and column 2 (vertical) induce unavoidable delays, even this method is not free from delays.

The functional procedure of this technique is quite simple. When the built-in self-tests within a router detect a faulty link of the router, the router´s switching arbiter is instructed to cancel out the afflicted link from its switching table. Incoming messages that are supposed to utilize this link now get one of the alternative links provided by default from the routing logic. The final arbitration is done by the switch arbiter, which also set the re-routed bit flag within the heartbeat message header. When the D-FDU receives such a re-routed message, it behaves in the same way as we already described it above.

The beauty of the approach is its light weight nature. Beside the mandatory fault detection ability of the router itself (which is needed anyway for message based fault localization), the only hardware overhead is the following:

- One bit in the heartbeat message header, which implies one bit of additional link bandwidth.
- Within the router, there is just the need for an additional header manipulator at the switch arbiter. The manipulator alters the re-routing flag, if a message needs to be re-routed.

However, the bandwidth consumption resulting from the additional bit flag within the header is negligible, since this wire can also be utilized from any other message. That means any other message type can utilize this wire for payload transmission.

3.1.5 Localization Costs

In this subsection we will show the estimated costs caused by our proposed fault localization technique. The basis of this calculation are the assumptions for the original proposed heartbeat message based monitoring, extended with the poll mechanism and overlapping clusters. Both, request messages and response messages are head-flit-only messages.

In Table 2: Estimated costs of our fault localization technique we distinguish the costs between the usage of the artificial indirection method and the re-routing bit method.

Table 2: Estimated costs of our fault localization technique

	Cost	Localization method	
		Pure timing and routing information	Re-routing bit
Artificial indirection	2 × FL Additional: Special HW support for case differentiation	×	
Alternating Routing	1 Bit in heartbeat message header flit	×	×
Overlapping Cluster Borders	$(no_m + no_m - f)$ × FL	×	×
Polling messages	FL × CN + OCB	×	×
Re-routing Bit	1 Bit in heartbeat message header flit Additional: HW Support to manipulate the Head flit		×

FL: Flit Length in Bits
CN: Number of monitored cores
OCB: Overlapping cluster border (formula description in section 3.1.3.2)

3.2 Topology Consideration and Fault Tolerance Implication

The investigation of Section 3.1 assumes nodes within a 2D mesh-based NoC. In this section we compare the topology with architectural template of TERAFLUX (see Figure 17 of D6.3). Therefore, we broaden the fault tolerance view for other NoC topologies differently from that we already used in the previous deliverables D5.1 and D5.2. While our investigations regarding the communication fabric so far was based on a two dimensional mesh (2D Mesh), the project decided to use a clustered architecture because of scaling reasons. A clustered architecture leads to a hierarchical structured

interconnection network. This network consists of at least two hierarchy stages. The first stage is the global interconnection network and connects all nodes, which are the second hierarchy stage. While the first stage can still be a 2D Mesh topology applying our technique described in Section 3.1, we will consider possible network topologies for the second stage.

The following subsection will therefore discuss which topologies are suitable as second stage interconnect from a fault tolerance point of view. At the end of this subsection we will propose a favorite topology meeting both the performance demands and the fault tolerance constraints.

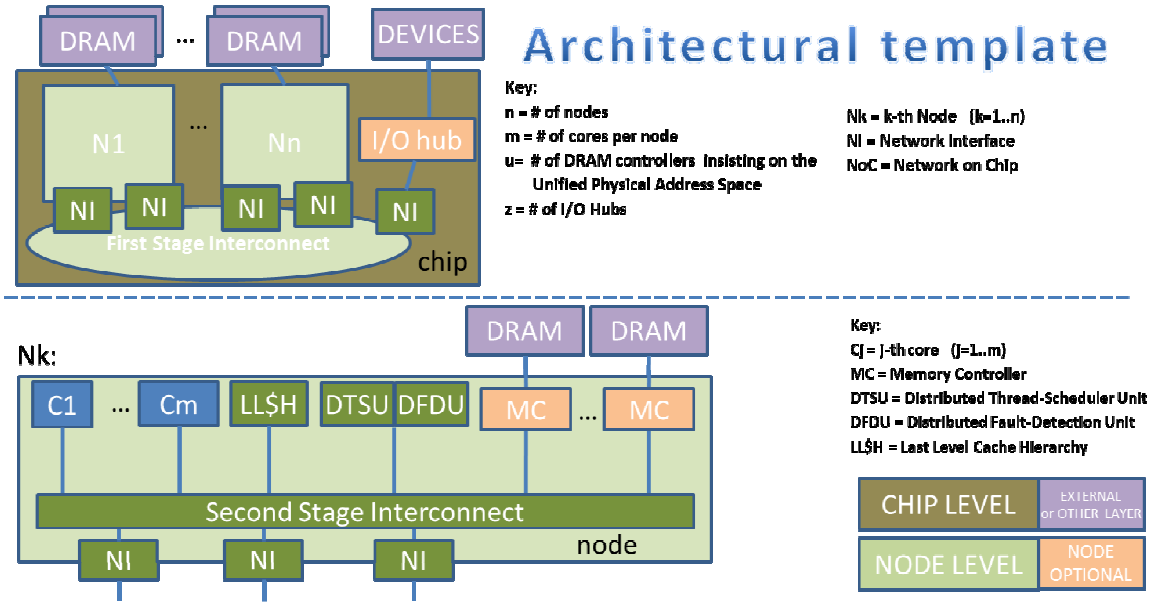

Figure 17: TERAFLUX Architectural template – Chip and Node Level

3.2.1 Clustered Architectures

Four of the most commonly used topologies for core interconnection are bus-based, tree-based, crossbar-based, and 2D Mesh [12]. With exception of the bus-based topology, we assume all topologies as switched networks, which means all cores communicate over the local interconnect via switches (see Figure 18). Bus based connected cores access the bus usually directly without the use of switches or routers.

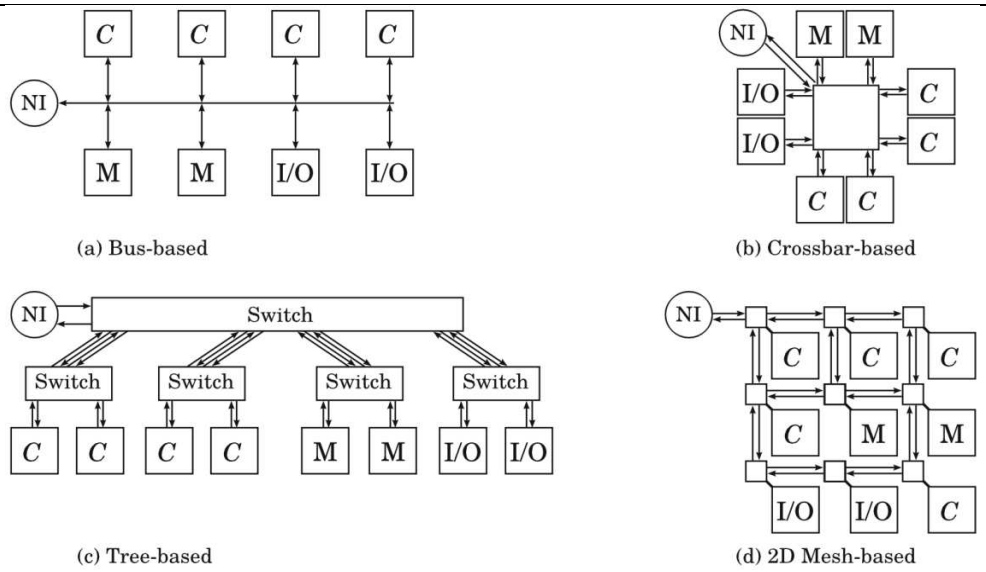

Figure 18: Potential local interconnection networks for the TERAFLUX architecture. C blocks represent cores, M are the memories, NI and I/O are network interface and Input-output devices, respectively.

Bus-based topologies are the most common way to interconnect a small number of cores. While a bus-based topology can support good performance for a small number of cores, it lacks enough capacity to efficiently serve more than 16 cores. Figure 18(a) shows a typical structure of bus-based interconnects. As one can easily see, if the central lane is broken, it disconnects at least two cores, memory controllers, or I/O controllers, respectively. This phenomenon is also referred as to single point of failure and is one of the major reasons, why a topology is not suitable from a fault tolerance point of view.

A crossbar-based topology (Figure 18(b)) has similar fault tolerance properties as the bus-based topology. Here the crossbar itself is the single point of failure and has the potential to disconnect the complete cluster from the rest of the communication network. From the performance point of view, the crossbar poses due its point to point connections better performance compared to the bus-based topology. However, the number of connected components to the crossbar is strongly related to its operational frequency. That means the higher the number of connected cores, the lower the operation frequency and the higher the area costs. Connecting 10 Cores yields max ~85% Chip-Area Utilization, which is the minimum utilization of the nonofficial industry design rule [11]. Connecting more than 10 Cores worsen this utilization factor.

In contrast to the bus-based topology and the crossbar topology the tree-based topology (Figure 18 (c)) provides one of the best performance results regarding highest available clock rates and very low latencies due to a small number of average hops [12]. The maximum number of cores is variable and depends on the integrated bandwidth of links connecting the intermediate switches with the central switch. However, this topology has poor redundancy capacities. Although, communication paths are now separated from each other, the central switch at the top represents a single point of failure. Additionally, the typical tree-based topology uses hard wired communication paths, which makes it hard to take advantages of potential redundant links.

As we already described in our deliverables D5.1 and D5.2 2D meshes are the best choice from the fault tolerance point of view. Its naturally redundant structure provides excellent opportunities in order to react on broken links and even broken routers. From the performance point of view, a 2D mesh-based topology provides good bandwidth capabilities [11]. Given a proper routing algorithm even congested areas can be bypassed and make use of the redundant links surrounding the congestion. However, due to a higher average hop count the performance in terms of latency is not as good as those compared to the tree-based topology.

All these topologies have a common weak agency. The NIC represents a particular failure point. In the worst case, one broken link at the NIC disconnects the whole cluster from the rest of the chip. A disconnected cluster/node may result in a complete system failure, since affiliated resources, such as memory controller or I/O controller may hold data, which is not accessible anymore. The consequence from such a situation can be a deadlocked thread or system.

In order to prevent the system from potential crashes, redundant NICs should be used at each cluster. Either these redundant NICs can be used as spare components, which step in for a broken NIC, or all NICs can be run simultaneously. The latter one has the advantage that the Inter-node communication may be split to different NICs and thus broaden the overall bandwidth for Inter-node communication. However, such a traffic split comes with the expense of additional hardware costs.

3.2.2 Proposed Topology

A cluster derived from the TERAFLUX architecture can be seen as a homogeneous tile. Their distribution over the chip area supports equal link lengths for both horizontal and vertical Switch-to-Switch interconnects. This link symmetry allows the employment of a simple global 2D Mesh network to interconnect the node's local interconnects. Both, the local interconnect and the inter cluster network are then from the same type with nearly equal properties from the fault tolerance point of view. Due to its redundant hardware it is flexible enough to match with the applied fault tolerance mechanisms and is affordable regarding the usage of hardware. 2D Meshes provide feasible performance and the current fault tolerance techniques are already applied to 2D Meshes, so no other investigation regarding different topologies is necessary. Therefore, we propose a 2D Mesh based local interconnect as shown in Figure 18(d), plus the possibility of passing NoC packet around without the need of additional bridges.

3.2.3 Conclusion

From a fault tolerance point of view, a flat NoC topology as investigated in Section 3.1 is superior to hardware-fixed clusters/nodes as proposed by the TERAFLUX architectural template, because there is no single point of failure except for the FDU/TSU nodes itself. Clusters/Nodes with bus, crossbar, or tree interconnects suffer from structural single point of failures. Fixed clusters also have fixed interconnects (NICs) to the global network, which also provide single point of failure that could make the whole cluster/node inaccessible. However, such interconnects could be doubled.

A flat topology with dynamically assigned cores to FDU/TSU would recover the cluster internal single point of failures (namely local interconnect or NIC). It allows reassigning cores to clusters in order to rebalance uneven cluster sizes, which occur due to faulty links, routers, and nodes.

If FDU/TSU functionalities would be implemented in software, such FDU/TSU nodes could also be assigned to any cores, eliminating also the hardware implementation of FDU/TSU. However, this represents a tradeoff between flexibility and performance. In our current design, when the components fails (independently of hardware or software), the system might collapse. However, this circumstance can't be avoided: a software based approach allows to keep the change to re-establish the system stability by moving FDU/TSU to another (non-faulty) core.

4 Dynamic Adaption

Managing the reliability and the resources of large scale systems such as assumed by the TERAFLUX project can be a non-trivial task. Thus, we decided to ease the complexity of the problem by using the following assumptions according to the proposed TERAFLUX architecture (c.f. D6.3).

1. The general system is divided into nodes. Each node has a node management unit; A Distributed Thread Scheduler Unit (D-TSU) and a Distributed Fault Detection Unit (D-FDU).
2. We assume "streaming" like applications, where new threads are dynamically created. Thus, in order to enforce load balancing, we can avoid moving threads from one node to another, but can do it by deciding where to create the new born threads.

All operating system related operations such as resource allocation, performance, and power optimization, fault detection and correction (resilient mechanisms) are implemented in a hierarchical manner:

At the top level, the TERAFLUX operating system sees the system as a collection of nodes. The nodes are connected by a virtual or physical interconnect (NoC). In this case, the considered NoC model is an extension to that described above. Here, we further assume that each router is configurable such that changes regarding the routing policy can be applied via configuration messages. The routing calculation is done by the *Tunable Disjoint Spanning Tree* (TDST) algorithm (developed by partner MSFT), which produces a spanning tree, covering the remaining healthy interconnects.

We are looking for end to end protection schemes in order to tolerate temporary faults in links. A common approach is the usage of fully edge disjoint paths, which are defined as primary and the second as backup. Given the congestion of each link some optimal disjoint path solutions can be found. Accordingly, we are searching for the optimal solution with the maximum number of intermediate nodes. For example, in a regular 2D mesh-based interconnection network with no congestion, the route between two opposite corner nodes should be a staircase for the primary path and its alternate staircase version for the backup path. We call this algorithmic scheme *MaxHop Disjoint Path Algorithm*.

The TERAFLUX operating system interacts with the DTS (all L-TSUs and D-TSUs) in order to perform thread management (i.e., scheduling, …). Hence, the thread management policies resulting from the communication between the TERAFLUX operating system and the DTS are as follows:

- General policies for load balancing of the execution decide where to create new threads. This schedule algorithm aims to load balance the execution between nodes and will take into consideration the nodes' fault rates, frequencies, and power consumption between nodes and power and performance optimizations.
- General policies for thermal and power management of the entire system. As part of that process, the TERAFLUX operating system will allocate a power and thermal budget to each node. In return the node reports to the operating system how well it uses the budged allocated to it. The operating system will change the allocation to the nodes dynamically over the runtime.

- Handle I/O for the entire system.

At the auxiliary nodes (c.f. D6.3 and D7.1), the system is maintained by a "nano or pico" kernel (NoS) (c.f. D7.1) that aims to handle all the resources of the cluster in an optimal way. The NoS will interact with D-TSU for a proper initialization of thread execution. Each thread will be in a dataflow style, meaning, it maintains memory accesses in a local memory, so that no side effect can occur before the thread is completed. The execution has an "all or nothing" notation, meaning we can kill the execution at any point before committing its results back to the global memory and the system can re-execute a thread if needed.

Along with the thread scheduling, the NoS additionally maintains the I/O operations by asking the Teraflux OS to execute it on behalf of the local thread. Error and exception handling in conjunction with gathered statistics on the performance and power of each core so enables global optimizations of voltage and frequency for the cluster (c.f. D7.3).

At the core level, we can assume that all management algorithms are done either in hardware or in microcode of the core. The system can assume a single threaded core. In this case, the role of the core is to

- execute the threads assigned to it,
- detect errors and report them,
- report on power and performance parameters (including heartbeat signals), and
- adjust the voltage and frequency of the core.

The system can assume multi-threaded cores. In this case, we assume a simple switch of event mechanism between the threads that where assigned to the core. In this case, on the top of all the activities a single threaded core needs to

- schedule the thread and
- check if a wait condition is fulfilled.

4.1 Implementation status and issues

The TERAFLUX OS is implemented as part of Linux (c.f. D7.1). We assume that the Linux OS has a special driver that aims to communicate between the TERAFLUX clusters and the Linux OS, as the following Figure 19 shows. A shared memory region is created between the Linux and the NoS together with a system of message queues. That allows an efficient communication between the NoS and the Linux OS. Furthermore, we assume that each cluster is controlled by an independent NoS. Currently, we examine the use of the Kitten nano-kernel [13] for that purpose.

The overall memory hierarchy is divided into local memory for each cluster and observable memory, which is the aggregation of all global memory portions of all nodes, also called Unified Address Space (c.f. D7.1). The local memory of each cluster can be used by its local NoS for maintaining its local data structures. All the shared memory can be accessed by each core, by using a universal addressing scheme (UAS) but no HW coherency is guaranteed. Please note that a core accesses its own global memory by the UAS or by using the direct access more. The system will have a Global Memory Management to handle the UAS.

1. At the node level, we are extending the Kitten nano kernel by adding a message passing layer to manage operations between the nodes and between the Linux and the nodes.
2. We are working on adding a layer of fault tolerant I/O. This is layer is based on one of the two algorithms (1) send all I/O in two disjoint virtual paths or (2) send in a primary path and define a secondary disjoin path to the case of failure.

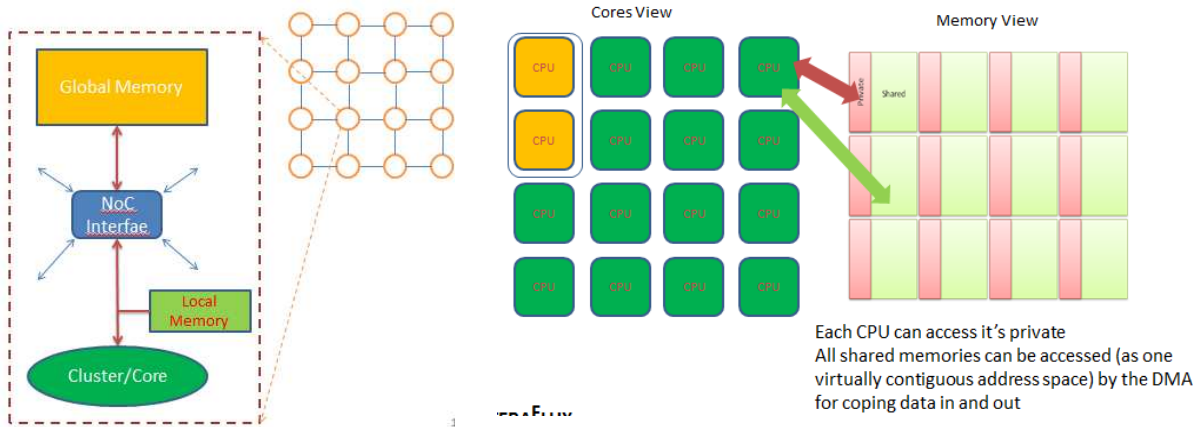

Figure 19: (a) cluster memory hierarchy (b) System-wide memory hierarchies

After building the current generation of dynamic control, we are planning to distribute most of the functionalities of the TERAFLUX OS.

Under this implementation, most of the top-level management that currently is implemented as a Linux device driver will be distributed over the Kitten nano kernel. The main benefit of such implementation is to avoid a single point of failure and potential performance bottleneck.

References

[1] S. Weis, A. Garbade, S. Schlingmann und T. Ungerer, „Towards Fault Detection Units as an Autonomous Fault Detection Approach for Future Many-Cores," in *1st Workshop on Software-Controlled, Adaptive Fault-Tolerance in Microprocessors (SCAFT 2011)*, 2011.

[2] S. Weis, A. Garbade, J. Wolf, B. Fechner, A. Mendelson, R. Giorgi und T. Ungerer, „A Fault Detection and Recovery Architecture for a Teradevice Dataflow System," in *First Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2011)*, Galveston Island, Texas, USA, 2011.

[3] E. Rotenberg, „AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," *International Symposium on Fault-Tolerant Computing*, pp. 84-91, 1999.

[4] D. Bernick, B. Bruckert, P. Vigna, D. Garcia, R. Jardine, J. Klecka und J. Smullen, „NonStop advanced architecture," Bd. Proceedings of the International Conference on Dependable Systems and Networks (DSN), pp. 12-21, 2005.

[5] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis und K. Olukotun, „Transactional Memory Coherence and Consistency," *Proceedings of the International Symposium on Computer Architecture* , pp. 102-113, 2004.

[6] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe und A. G. Nowatzyk, „Fingerprinting: bounding soft-error detection latency and bandwidth," in *Proceedings of the 11th International Conference on Architectural support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[7] M. Prvulovic, Z. Zhang und J. Torrellas, „Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, Washington, DC, USA, 2002.

[8] D. J. Sorin, M. M. M. K. Matrin, M. D. Hill und D. A. Wood, „Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Proceeding of the 29th Annual International Symposium on Computer Architecture (ISCA)*, Washington, DC, USA, 2002.

[9] A. Garbade, S. Weis, S. Schlingmann, B. Fechner and T. Ungerer, "Impact of Message-Based Fault Detectors on a Network on Chip," in *21th International Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, Belfast, 2013 (accepted paper).

[10] B. Fechner, A. Garbade, S. Weis and T. Ungerer, "Fault localizaton in NoCs by Timed Heartbeats," in *Workshop on Dependability and Fault Tolerance (ARCS/VERFE Workshop)*, 2012.

[11] A. Pullini, F. Angiolini, S. Murali, D. Atienza, G. De Micheli und L. Benini, „Bringing NoCs to 65nm,“ *IEEE MICRO*, Nr. 5, pp. 75-85, 2007.

[12] J. Flich und D. Bertozzi, Designing Network On-Chip Architectures in the Nanoscale Era, Boca Raton, FL: Chapman and Hall/CRC, 2011.

[13] Kitten nano-kernel: https://software.sandia.gov/trac/kitten

[14] Roberto Giorgi, "TERAFLUX: Exploiting Dataflow Parallelism in Teradevices", *ACM Computing Frontiers*, Cagliari, Italy, May 2012, pp. 303-304.

[15] Roberto Giorgi, Zdravko Popovic, Nikola Puzovic, "DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems", *Proc. IEEE SBAC-PAD*, Gramado, Brasil, Oct. 2007, pp. 263-270

[16] COTSon repository: cotson.sourceforge.net/