



**SEVENTH FRAMEWORK PROGRAMME
THEME**

**FET proactive 1: Concurrent Tera-Device
Computing (ICT-2009.8.1)**



PROJECT NUMBER: 249013



Exploiting dataflow parallelism in Teradevice Computing

**D5.4 – System Integration Analysis, Measurement and Tuning of the
Reliability System**

Due date of deliverable: 31st March 2014
Actual Submission: 19th May 2014

Start date of the project: January 1st, 2010

Duration: 51 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)

Deliverable number: **D5.4**

Deliverable name: **System Integration Analysis, Measurement and Tuning of the Reliability System**

File name: TERAFLUX-D54-v8.doc

Page 1 of 60

Change Control

Version#	Date	Author	Organization	Change History
1	03.02.2014	Sebastian Weis	UAU	Initial document
2	01.03.2013	Arne Garbade, Sebastian Weis	UAU	Integrated core and NoC-level sections
3	03.02.2014	Amit Fuchs, Yaron Weisberg	MSFT	Integrated operating system section
4	31.03.2014	Sebastian Weis	UAU	Revised document after internal review
6	04.05.2014	Roberto Giorgi	UNISI	Review

Release Approval

Name	Role	Date
Sebastian Weis, Arne Garbade	Originator	14.03.2014
Theo Ungerer	WP Leader	19.03.2014
Roberto Giorgi	Project Coordinator for formal deliverable	09.05.2014

TABLE OF CONTENTS

GLOSSARY.....	8
EXECUTIVE SUMMARY.....	9
1 INTRODUCTION.....	10
1.1 FAULT TOLERANCE CONCEPT.....	11
1.1.1 <i>Monitoring of Cores and Interconnect by periodic Heartbeats.....</i>	11
1.1.2 <i>Leveraging Dataflow for Fault Detection and Recovery.....</i>	11
1.2 DOCUMENT STRUCTURE.....	13
1.3 RELATION TO OTHER DELIVERABLES.....	13
1.4 ACTIVITIES REFERRED BY THIS DELIVERABLE.....	14
2 CORE-LEVEL FAULT TOLERANCE IN TERAFLUX.....	15
2.1 FAULT DETECTION.....	15
2.1.1 <i>Sphere of Replication for Double Execution.....</i>	15
2.1.2 <i>Pessimistic Double Execution.....</i>	16
2.1.3 <i>Optimistic Double Execution.....</i>	18
2.2 RECOVERY.....	20
2.2.1 <i>Global Error Recovery.....</i>	20
2.3 QUANTITATIVE RESULTS.....	23
2.3.1 <i>Simulation Methodology.....</i>	23
2.3.2 <i>Fault-free Execution.....</i>	24
2.3.3 <i>Execution under Faults.....</i>	29
2.3.4 <i>Multi Node Behavior.....</i>	30
3 NOC-LEVEL FAULT TOLERANCE IN TERAFLUX.....	33
3.1 IMPACT OF HB MESSAGES ON APP. MESSAGES.....	33
3.1.1 <i>Metrics of interest.....</i>	33
3.1.2 <i>Evaluation Methodology.....</i>	34
3.1.3 <i>Quantification.....</i>	36
3.2 FAULT LOCALIZATION WITH MULTIPLE FAULTS WITHIN THE NOC.....	41
3.2.1 <i>Investigation Methodology.....</i>	41
3.2.2 <i>Applying fault pairs to the NoC.....</i>	42
3.2.3 <i>Quantification.....</i>	47
4 OS-LEVEL FAULT TOLERANCE IN TERAFLUX.....	49
4.1 BASIC ARCHITECTURE.....	49
4.1.1 <i>Clustered Architecture.....</i>	49
4.2 OPERATING SYSTEM GOALS.....	50
4.2.1 <i>Execution Model.....</i>	50
4.2.2 <i>Fault Tolerance.....</i>	50
4.3 RUNTIME ENVIRONMENT.....	50
4.3.1 <i>Memory Arrangement.....</i>	50
4.3.2 <i>Inter-node Communication.....</i>	51

Deliverable number: **D5.4**

Deliverable name: **System Integration Analysis, Measurement and Tuning of the Reliability System**

File name: TERAFLUX-D54-v8.doc

Page 3 of 60

4.3.3	<i>Node Failure Tolerance</i>	52
4.3.4	<i>Thread Execution Procedure</i>	53
4.4	IMPLEMENTATION DETAILS	55
4.4.1	<i>Thread Identifier</i>	55
4.4.2	<i>Thread Binaries</i>	55
4.4.3	<i>Fail Tolerant Synchronization Count</i>	56
4.4.4	<i>Integration into TERAFLUX</i>	56
4.5	RELATED RESEARCH	58
REFERENCES		59

LIST OF FIGURES

FIGURE 1: INPUT REPLICATION FOR TSCHEDULE INSTRUCTION.	12
FIGURE 2: SPHERE OF REPLICATION FOR DOUBLE EXECUTION.....	15
FIGURE 3: PESSIMISTIC DOUBLE EXECUTION.	16
FIGURE 4: OPTIMISTIC DOUBLE EXECUTION WITH REDUCED WAITING TIME.	19
FIGURE 5: ESTABLISH GLOBAL CHECKPOINT FOR 4 NODES.	22
FIGURE 6: EXECUTION OVERHEAD OF PESSIMISTIC DOUBLE EXECUTION COMPARED TO NON-REDUNDANT EXECUTION WITH HALF OF THE CORES PER NODE.	25
FIGURE 7: EXECUTION OVERHEAD OF OPTIMISTIC DOUBLE EXECUTION COMPARED TO NON-REDUNDANT EXECUTION WITH HALF OF THE CORES PER NODE.	26
FIGURE 8: OVERHEAD OF SPECULATIVE OPTIMISTIC DOUBLE EXECUTION WITH NODE CHECKPOINTING AT AN INTERVAL OF 10,000 CYCLES COMPARED TO NON-REDUNDANT EXECUTION.....	28
FIGURE 9: OVERHEAD OF PESSIMISTIC DOUBLE EXECUTION IN THE CASE OF A FAULT RATE OF 0.01 FAULTS PER SECOND COMPARED TO NON-FAULTY PESSIMISTIC DOUBLE EXECUTION.	29
FIGURE 10: OVERHEAD OF OPTIMISTIC DOUBLE EXECUTION IN THE CASE OF A FAULT RATE OF 0.01 FAULTS PER SECOND COMPARED TO NON-REDUNDANT EXECUTION WITHOUT FAULTS.	30
FIGURE 11: SCALABILITY OF PESSIMISTIC AND OPTIMISTIC DOUBLE EXECUTION FOR 1 TO 8 NODES. EACH NODE COMPRISES 16 CORES.	31
FIGURE 12: THROUGHPUT OF APPLICATION MESSAGES UNDER TRAFFIC PATTERN RANDOM.	36
FIGURE 13: THROUGHPUT OF APPLICATION MESSAGES UNDER TRAFFIC PATTERN HOTSPOT.	36
FIGURE 14: APPLICATION MESSAGE LATENCY UNDER TRAFFIC PATTERN RANDOM.....	37
FIGURE 15: APPLICATION MESSAGE LATENCY UNDER TRAFFIC PATTERN HOTSPOT.....	37
FIGURE 16: JITTER AT 0.0001 FOR TRAFFIC PATTERN RANDOM.....	38
FIGURE 17: JITTER AT 0.001 FOR TRAFFIC PATTERN RANDOM.....	38
FIGURE 18: JITTER AT 0.01 FOR TRAFFIC PATTERN RANDOM.....	38
FIGURE 19: JITTER AT 0.00001 FOR TRAFFIC PATTERN HOT-SPOT.	39
FIGURE 20: JITTER AT 0.001 FOR TRAFFIC PATTERN HOT-SPOT.	39
FIGURE 21: JITTER AT 0.01 FOR TRAFFIC PATTERN HOT-SPOT.	39
FIGURE 22: BLIND SPOT DUE TO F ₁	42
FIGURE 23: CORRESPONDING MATRIX OF SUSPICIOUS NETWORK COMPONENTS.	42
FIGURE 24: RESOLVED BLIND SPOT BY MOVING THE FDU TO A DIFFERENT CORE.....	43
FIGURE 25: BLIND SPOTS DUE TO F ₁ AND F ₄	43
FIGURE 26: CORRESPONDING MATRIX OF SUSPICIOUS NETWORK COMPONENTS.	43
FIGURE 27: PHANTOM FAULT DUE TO F ₁ AND F ₂	45
FIGURE 28: CORRESPONDING MATRIX OF SUSPICIOUS NETWORK COMPONENTS.	45
FIGURE 29: EXAMPLE SCENARIO FOR THE IMPLICATION OF SUCCESSIVE OCCURRING MULTIPLE FAULTS.....	46
FIGURE 30: STATUS MATRIX AFTER PHASE #2 FOR <i>SIMULTANEOUSLY</i> OCCURRED FAULTS.	47
FIGURE 31: STATUS MATRIX AFTER PHASE #2 FOR <i>SUCCESSIVELY</i> OCCURRED FAULTS.	47
FIGURE 32: LOGICAL SYSTEM VIEW.	50
FIGURE 33: MEMORY ARRANGEMENT.....	51
FIGURE 34: SIMULATION OVERVIEW.....	57

LIST OF TABLES

TABLE 1: BASELINE NODE CONFIGURATION	24
TABLE 2: TERAFLUX-SPECIFIC PARAMETERS.....	24
TABLE 3: NODE UTILIZATION OF PESSIMISTIC AND OPTIMISTIC DOUBLE EXECUTION (NO-FAULTS).....	27
TABLE 4: SPEEDUP OF OPTIMISTIC DOUBLE EXECUTION COMPARED TO PESSIMISTIC DOUBLE EXECUTION (NO-FAULTS).....	27
TABLE 5: OVERHEAD OF OPTIMISTIC DOUBLE EXECUTION COMPARED TO PESSIMISTIC DOUBLE EXECUTION WITHOUT NODE CHECKPOINTS.	29
TABLE 6: OVERVIEW OF THROUGHPUT, LATENCY, AND MAXIMUM DELAY FOR APPLICATION MESSAGES	40
TABLE 7: DIFFERENT PROBLEMATIC FAULT PATTERN REGARDING LOCATION AND ORIENTATION.....	44
TABLE 8: QUANTIFICATION OF PATTERNS	48

List of contributors to the writing of the document.

Sebastian Weis, Arne Garbade, Theo Ungerer
University of Augsburg

Yaron Weinsberg, Amit Fuchs
Microsoft Research and Development

© 2009-14 TERAFLUX Consortium, All Rights Reserved

Document marked as PU (Public) is published in Italy, for the TERAFLUX Consortium, on the www.teraflux.eu web site and can be distributed to the Public.

The list of author does not imply any claim of ownership on the Intellectual Properties described in this document.

The authors and the publishers make no expressed or implied warranty of any kind and assume no responsibilities for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information contained in this document.

This document is furnished under the terms of the TERAFLUX License Agreement (the "License") and may only be used or copied in accordance with the terms of the License. The information in this document is a work in progress, jointly developed by the members of TERAFLUX Consortium ("TERAFLUX") and is provided for informational use only.

The technology disclosed herein may be protected by one or more patents, copyrights, trademarks and/or trade secrets owned by or licensed to TERAFLUX Partners. The partners reserve all rights with respect to such technology and related materials. Any use of the protected technology and related material beyond the terms of the License without the prior written consent of TERAFLUX is prohibited. This document contains material that is confidential to TERAFLUX and its members and licensors. Until publication, the user should assume that all materials contained and/or referenced in this document are confidential and proprietary unless otherwise indicated or apparent from the nature of such materials (for example, references to publicly available forms or documents).

Disclosure or use of this document or any material contained herein, other than as expressly permitted, is prohibited without the prior written consent of TERAFLUX or such other party that may grant permission to use its proprietary material. The trademarks, logos, and service marks displayed in this document are the registered and unregistered trademarks of TERAFLUX, its members and its licensors. The copyright and trademarks owned by TERAFLUX, whether registered or unregistered, may not be used in connection with any product or service that is not owned, approved or distributed by TERAFLUX, and may not be used in any manner that is likely to cause customer confusion or that disparages TERAFLUX. Nothing contained in this document should be construed as granting by implication, estoppel, or otherwise, any license or right to use any copyright without the express written consent of TERAFLUX, its licensors or a third party owner of any such trademark.

Printed in Siena, Italy, Europe

Part number: *please refer to the File name in the document footer.*

DISCLAIMER

EXCEPT AS OTHERWISE EXPRESSLY PROVIDED, THE TERAFLUX SPECIFICATION IS PROVIDED BY TERAFLUX TO MEMBERS "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS.

TERAFLUX SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES OF ANY KIND OR NATURE WHATSOEVER (INCLUDING, WITHOUT LIMITATION, ANY DAMAGES ARISING FROM LOSS OF USE OR LOST BUSINESS, REVENUE, PROFITS, DATA OR GOODWILL) ARISING IN CONNECTION WITH ANY INFRINGEMENT CLAIMS BY THIRD PARTIES OR THE SPECIFICATION, WHETHER IN AN ACTION IN CONTRACT, TORT, STRICT LIABILITY, NEGLIGENCE, OR ANY OTHER THEORY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Deliverable number: **D5.4**

Deliverable name: **System Integration Analysis, Measurement and Tuning of the Reliability System**

File name: TERAFLUX-D54-v8.doc

Page 7 of 60

Glossary

D-FDU	Distributed Fault Detection Unit
DF-Thread	A dataflow thread
D-TSU	Distributed Thread Scheduler Unit
ECC	Error Correction Code
EDC	Error Correction and Detection
FM	Frame Memory
Leading Thread	Represents the forerunning thread in the Double Execution approach
L-FDU	Local Fault Detection Unit
L-TSU	Local Thread Scheduler Unit
MAPE	Acronym for Monitoring, Analysing, Planning, and Executing
NoC	Network-on-Chip
Node	Group of cores and additional TERAFLUX hardware units
SPSC	Single Producer Single Consumer
TDMA	Time Division Multiple Access
Trailing Thread	Represents the trailing thread in the Double Execution approach
QoS	Quality of Service

Executive Summary

This deliverable reports on the research carried out in the context of DoW Task 5.4 (project months 36 -51) “**System Integration Analysis, Measurement and Tuning of the Reliability System**”:

- We integrated optimistic and pessimistic Double Execution, two redundant thread execution mechanisms, in the TERAFLUX architecture.
- We integrated a global error recovery mechanism into the TERAFLUX architecture.
- We quantified the overhead of pessimistic and optimistic Double Execution and local and global error recovery in the TERAFLUX architecture.
- We quantified the overhead of heartbeat messages in the NoC and developed a fault localization scheme in the interconnection network for multiple link faults.
- MSFT integrated their fault tolerant operating system concept into the architecture.

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

1 Introduction

In the Deliverables D5.1, D5.2, and D5.3 we presented concepts to improve the on-chip reliability for a future parallel architecture. These concepts are applied at different levels: the core level, NoC level, and OS level. At core and OS level, we were able to leverage the coarse-grained dataflow semantics in order to implement efficient fault detection (Double Execution) and recovery mechanisms (thread restart recovery). On NoC level we exploited heartbeat monitoring to incorporate a network-on-chip fault localization mechanism.

This Deliverable D5.4 focuses on *System Integration Analysis, Measurement and Tuning of the Reliability System*.

In detail Task 5.4 requests:

“In this Task the developed techniques are integrated more tightly into the overall architecture. To evaluate the benefits and the performance of the proposed methods, the system is measured with specific benchmarks and applications. The developed techniques will be tuned and optimized for fault detection capability and minimal space and runtime overhead. Further verification of the reliability features is performed through pilot studies.”

On the core-level, UAU developed optimistic and pessimistic Double Execution variants to improve the runtime overhead induced by the redundant thread execution of Double Execution. Optimistic Double Execution also demanded for an additional recovery mechanism. Therefore, we extended the node recovery mechanisms proposed in Deliverable D5.3 to support a coordinated global checkpoint mechanism. Finally, we integrated optimistic and pessimistic Double Execution and local and global thread recovery in the TERAFLUX simulator [1] [2], developed in WP7. Based on this integration, we were able to quantify the overhead induced by optimistic and pessimistic Double Execution, as well as the overhead induced by the coordinated global checkpoint mechanism. Finally, we also investigated the recovery overhead for a system that is constantly suffering from a high rate of transient faults.

On NoC-level, UAU deepened the investigation from year two (Deliverable D5.2) of the impact of message based fault tolerance mechanisms on application messages. We could show that under different traffic patterns (applied to application messages) the proposed Staircase routing strategy can significantly relax the impact of Heartbeat messages within the NoC. Furthermore, we examined the capabilities of the fault localization technique developed in year three (Deliverable D5.3) regarding multiple faults within the NoC. We identified problematic fault patterns (distributed in space and time) of multiple faults, which prevent successful fault localizations and give a quantitative result on the portion of these patterns compared with all possible patterns.

On OS-level, MSFT focused on their distributed, reliable operating system to manage the dataflow thread execution. They integrated their work also in the COTSon simulator, which implements a shared memory model for highly-parallel architecture, with acquire/release consistency.

1.1 Fault Tolerance Concept

Since most of the optimizations and evaluations are based on mechanisms reported in earlier deliverables and publications [3] [4], this section includes a short recap of the background work described in the Deliverables D5.1, D5.2 and D5.3. For the reader's convenience, we start with a summary of the previously introduced on-chip fault tolerance concepts. For a more detailed discussion of these concepts please refer to the Deliverables D5.1, D5.2, and D5.3.

1.1.1 Monitoring of Cores and Interconnect by periodic Heartbeats

1.1.1.1 Fault Monitoring by periodic Heartbeats

In order to transmit fault information from the core-local FDUs (L-FDU) to the distributed FDU (D-FDU), we proposed special status messages (heartbeats), carrying health state information of the respective processor core. To gain the core status information, the L-FDU may read the performance counter registers, as they are available in current x86 cores. Once gathered the information, the L-FDU generates a message and stores the gathered information in it. Before sending this status message to the affiliated D-FDU, the L-FDU has to wait until it gets a certain time slot. These time slots are meant to separate heartbeat messages within the interconnection network. The reason for the separation lies in the D-FDU, since a missing heartbeat message indicates the loss of a processor core (without the distinction of a broken core or a disconnected core).

1.1.1.2 Fault Localization on NoC-Level

The fault localization within the interconnection network is also based on the periodic heartbeat messages. Together with a deterministic routing strategy and a prioritization method (Quality of Service, QoS), we are able to calculate the estimated arrival times of all heartbeat messages. The precise arrival time is used to check whether a heartbeat message was transmitted correctly (in terms of its route through the network), or not. When a message is delivered late, we can infer that it was not transmitted via the intended route, which indicates that there is a faulty element somewhere on this route. In this first step, we assume all the network components corresponding to this route to be suspicious. On the other hand, if a heartbeat message is delivered in time, we may rehabilitate former suspected elements corresponding to this route. Doing this for all heartbeat messages and their routes through the network, we are able to localize precisely single faults within the interconnection network. For a detailed presentation we refer the reader to Section 3.1.1 of Deliverable D5.3.

1.1.2 Leveraging Dataflow for Fault Detection and Recovery

As stated before, the fault detection and recovery mechanisms exploit the TERAFLUX dataflow execution model.

1.1.2.1 Fault Detection by Double Execution

The concept of fault detection is implemented by redundant execution of dataflow threads, called Double Execution. The basic Double Execution mechanism is able to support special and temporal redundancy. Double Execution leverages the dataflow execution semantic to provide efficient solutions for *thread duplication*, *input replication*, and *output comparison*.

Thread Duplication. Since dataflow threads are side-effect free and `twrites` are only assigned once before the thread starts (for the T* instructions s. D6.2, D7.4 and [16]), only the *continuation* of a thread needs to be copied for Double Execution within a node. The dataflow execution model thereby eases the dynamic duplication of the threads for Double Execution.

As Double Execution uses the dataflow threads for redundant execution, the input data of the redundant dataflow threads must be consistently replicated.

Input Replication The execution of a dataflow thread only depends on its thread frame. Since the thread frame is immutable after the synchronization count has reached zero, both redundant threads are allowed to read the input data from the same thread frame, because data inconsistencies between redundant threads induced by race conditions of concurrent `tread` and `twrite` operations are impossible.

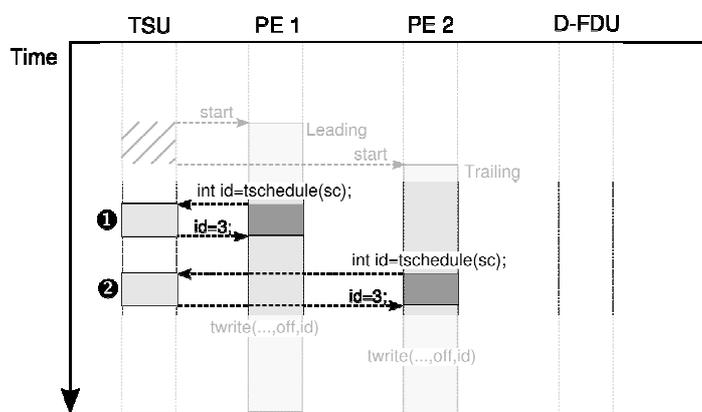


Figure 1: Input replication for `tschedule` instruction.

However, beside the input data from the thread frame, a redundant dataflow thread can issue `tschedule` instructions to dynamically create subsequent threads. In Deliverable D5.3, we have described a technique to guarantee consistent thread IDs for redundant threads (see Figure 1).

Synchronization and Output Comparison. Double Execution threads are synchronized for result comparison on thread level. Therefore, the synchronization frequency depends on the executed application and the length of the dataflow threads. In order to reduce the synchronization overhead, CRC-32 signatures of the redundant threads' write sets are created and compared. Faults can only be detected, when the thread pairs synchronize, which means that the average fault detection latency directly depends on the application's average dataflow thread length.

Optimizing Double Execution Based on the original Double Execution concept, we further developed two different Double Execution variants, which have influence on the recovery mechanism and the containment of errors in the architecture:

- *Pessimistic Double Execution* pessimistically assumes that faults are frequent and recovery is often triggered.
- *Optimistic Double Execution* tries to optimize the fault-free case by an optimistic commit of the dataflow threads.

1.1.2.2 Recovery

In the Deliverables D5.2 and D5.3, we developed a thread restart mechanism and a node recovery mechanism, which both make use of the dataflow execution model.

Thread Restart Recovery The functional semantic of the execution model prevents dataflow threads from accessing the global memory before the `tdestroy` instruction (for the T^* instructions s. D6.2, D7.4 and [16]) is called by a thread. This enables the D-TSU to directly control all memory accesses, which are able to manipulate the global system state. Accordingly, the dataflow thread boundaries can be seen as inherent execution checkpoints.

Based on the side-effect free execution model, which is supported by the TERAFLUX architecture through the core-local write buffers and the speculatively created continuations (see Deliverable D5.3), the D-TSU can restart dataflow threads to recover from faults.

Optimizing Recovery Although the thread restart recovery is transparent to the application level, the programmer, and the compiler, its recovery capability is restricted to the length of the dataflow threads of the application. This means, long latency fault detection mechanisms are not supported by the thread restart recovery. Furthermore, optimistic Double Execution makes it impossible to use thread restart recovery. Therefore, we proposed a node checkpoint mechanism in Deliverable D5.3. In this Deliverable we extended node checkpointing to a coordinated global checkpointing mechanism which supports global checkpointing of the TERAFLUX system across nodes.

1.2 Document structure

In Section 2 we present pessimistic and optimistic versions of Double Execution of dataflow threads. Additionally to the thread restart recovery, we describe a global recovery mechanism for the TERAFLUX system. Section 2 also includes quantitative results regarding the induced overhead of these mechanisms.

Section 3 presents extended work on the fault localization mechanism, including an investigation of the impact of prioritized heartbeat messages upon application messages, an extensive investigation on multiple faults within the interconnection network, and an extension to our previously started network topology consideration from a fault tolerance perspective.

Section 4 describes the operating system fault tolerance mechanism across different TERAFLUX nodes.

1.3 Relation to other deliverables

- The monitoring concept, the heartbeat message protocols and the D-FDU MAPE cycle are part of Deliverable D 5.1.
- The Double Execution principle and fault tolerant hardware extensions are described in the Deliverables D5.2 and D5.3.
- The described recovery mechanisms are also used in the TERAFLUX abstraction layer described in D6.4.
- Deliverable D7.5 describes the example execution of an optimistic and pessimistic Double Execution run for one TERAFLUX node for the Fibonacci benchmark.

1.4 Activities referred by this deliverable

This deliverable refers to the research carried out in *Task 5.4 – System Integration Analysis, Measurement and Tuning of the Reliability System* and also concerns the fulfillment of *Milestone M5.4*.

2 Core-Level Fault Tolerance in TERAFLUX

This section starts with a description of pessimistic and optimistic Double Execution. Furthermore, a global recovery mechanism will be described and finally evaluation results of the execution overhead are given.

2.1 Fault Detection

Compared to prior redundant execution mechanisms, like [5] [6] [7], Double Execution (see Deliverables D5.2 and D5.3) makes extensive use of the TERAFLUX dataflow execution principle. As described, dataflow execution is leveraged for efficient solutions of *input replication*, redundant *thread synchronization*, and *output comparison*. Beside the advantages inherent to loosely-coupled redundant execution mechanisms, Double Execution does not require permanent coupling between redundant execution units, reducing the performance degradation when an odd number of processing elements is permanently broken and provides support for dynamic time and spatial execution.

In the remainder of this section, *pessimistic* and *optimistic* Double Execution mechanisms are introduced to either optimize for fast recovery or fast redundant thread execution. Finally, both Double Execution variants are qualitatively compared with redundant execution schemes for control-flow based multi-cores, regarding their input replication, thread synchronization, and output comparison techniques and the implications for their use in massively-parallel architectures.

2.1.1 Sphere of Replication for Double Execution

The *sphere of replication* defines the hardware region, where faults can be detected by a redundant execution mechanism. Input data that enters the sphere must be replicated in a consistent way for both execution instances. Likewise, data that leaves the sphere of replication must be checked for faults, otherwise faults cannot be detected at a later stage of execution.

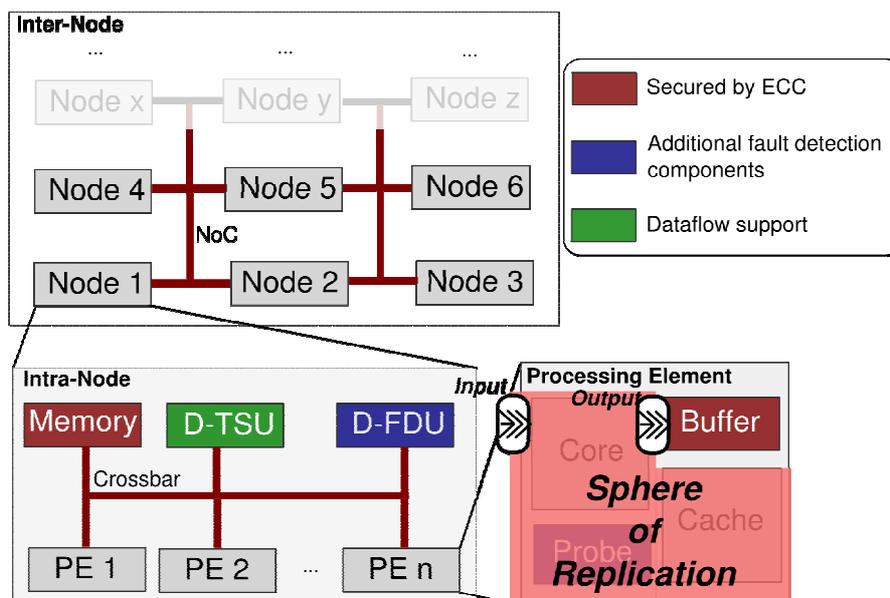


Figure 2: Sphere of Replication for Double Execution.

Figure 2 shows the sphere of replication for transient, intermittent, and permanent faults used for the enhanced fault tolerant TERAFLUX architecture.

Each sphere is restricted to one processor core, incorporating all core-local hardware components, i.e. the core itself, the private cache, the frame memory (FM), and the L-FDU. However, the *write buffer* must be safeguarded by ECC, since the write set of a thread is kept in this buffer after the signature has been created. Hence, a fault in the write buffer after the CRC-32 signature was created cannot be detected by Double Execution.

For the rest of the system, i.e. interconnection networks, off-chip memory, and the memory controller, we assume that efficient information redundancy mechanisms, like ECC or EDC, are employed. Furthermore, we assume that the D-TSU and the D-FDU are safeguarded by special hardware designs.

2.1.2 Pessimistic Double Execution

Pessimistic Double Execution is a variant of *Double Execution*, which pessimistically assumes that faults occur frequent and recovery operations must be triggered often. As a consequence, *Pessimistic Double Execution* tries to reduce recovery costs at the expense of redundant execution performance.

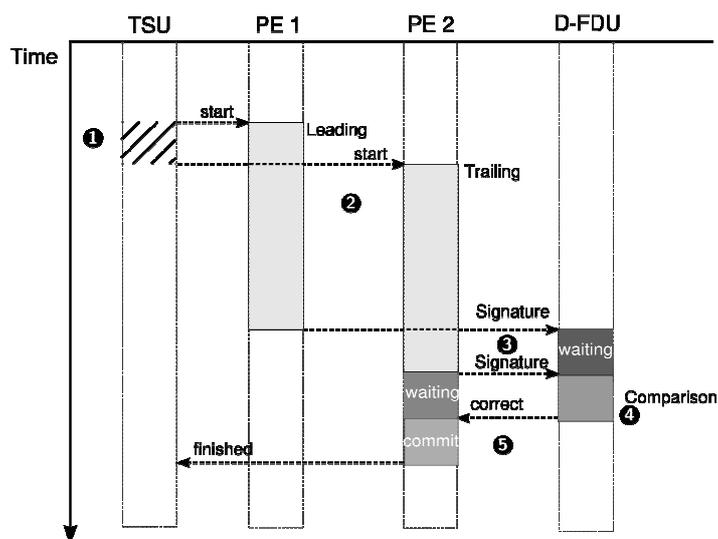


Figure 3: Pessimistic Double Execution.

2.1.2.1 Execution Principle

Figure 3 shows, how *Pessimistic Double Execution* works:

1. **Thread start:** The leading thread is immediately started, when a core is available and ready for execution. The trailing thread is started, when the next core in the node becomes available. The time between the start of the leading thread and the trailing thread is called *start slack*. Since threads are started on their availability, the *start slack* is determined by the utilization of the node.

2. **Thread execution:** During execution, the core of the trailing thread buffers all `twrite` (and `twritep`) instructions in its write buffer. Simultaneously, the L-TSU creates a CRC-32 signature of all `twrites` (and `twriteps`) instructions, incorporating the target thread ID, the target address, and the data. The L-FDU of the leading thread's core also creates a CRC-32 signature of all `twrites` (and `twriteps`); however, after signature creation the `twrites` (`twriteps`) of the leading thread are discarded. Since only the trailing thread buffers `twrites`, the leading thread's core can immediately schedule waiting threads, when the leading thread has finished. Other write operations to the thread local storage (heap or stack) do not need to be buffered or incorporated in the signature, since they will be automatically overwritten after recovery.
3. **Thread end:** 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. Since the results of the leading thread are not buffered by its core, the core is immediately ready to execute the next dataflow thread. In contrast, the trailing thread's core has buffered the results in the write buffer and must wait until result comparison to be able to commit them to global memory.
4. **Result comparison:** The D-FDU, which has been notified by the D-TSU about duplication of the redundant continuation, waits for the signatures of both the leading and the trailing threads, compares them and informs the D-TSU and the waiting trailing thread about the result.
5. **Thread commit or recovery:** In case of a non-faulty execution of both threads, the PE redirects the buffered writes of the trailing thread to the TSU, which commits them to the global memory and reduces the synchronization counts of the succeeding dataflow threads. Finally, the D-TSU subsequently deletes the continuations of the leading and the trailing thread. If a fault was detected, the D-TSU instructs the D-TSU to flush the core-local write buffer with the write set of the leading thread and discards all continuations created by the faulty thread.

2.1.2.2 Performance Overhead

Beside the doubled core utilization inherent to all redundant execution schemes, the overhead of Double Execution compared to a conventional dataflow execution is influenced by two factors:

A *longer thread execution time* (slack and signature verification) and the *idle time* of the trailing thread (in case the leading thread finishes later than the trailing thread).

2.1.2.2.1 Longer Thread Execution Time

For the global progress of the system, *Pessimistic Double Execution* behaves like a conventional dataflow execution, since only the trailing thread is allowed to commit its results to global memory. However, compared to conventional dataflow execution, the time between start (of the leading thread) and commit (of the trailing thread) is longer than for a non-redundant dataflow execution. The longer execution time is affected by two factors: The *start slack* and the *comparison latency* both lead to a deferred commit of the trailing thread, and hence to longer thread execution time.

2.1.2.2.2 Idle Time of the Trailing Thread

While the core of the leading thread is immediately released for subsequent threads after `tdestroy` has been retired, the trailing thread has to buffer the results of the dataflow execution in its write buffer. During result comparison, the core of the trailing thread cannot execute subsequent dataflow threads, since the core's write buffer would be otherwise overwritten. Therefore, the core of the trailing thread is blocked for further dataflow threads until the D-FDU has finished the result comparison and the core's write buffer has been committed.

The blocking time is minimized, when the leading thread always finishes execution before the trailing thread. In this case, the core is only blocked for other leading threads until the D-FDU has compared the results and confirmed a fault free execution to D-TSU, which triggers the commit of the redundant execution.

2.1.2.3 Error Containment

Since pessimistic Double Execution buffers the computational results of the trailing thread in the core-local write buffer, possible errors cannot be distributed to the global system state, i.e. the global memory, which can be accessed by all cores of the system. In other words, error propagation for pessimistic Double Execution is contained by the sphere of replication. As no global memory is written until pessimistic Double Execution ensures the fault-free execution by comparison of the signatures, the global system state does not need to be recovered. Furthermore, since communication between threads is only allowed, when dataflow threads commit their results, no faults are propagated until the fault-free execution is guaranteed. This means that only the core must be recovered in case of a faulty thread execution.

2.1.3 Optimistic Double Execution

To eliminate the performance overhead introduced with pessimistic Double Execution for longer thread execution time and the core's idle time and to increase the parallelism of Double Execution, the D-FDU can dynamically decide to use *Optimistic Double Execution*. Hence, the D-FDU may analyze that faults are very rare events and most threads can be executed without suffering from a fault. Accordingly, there is no need for a fast and simple recovery mechanism, but for an efficient redundant execution. In this case the D-FDU can choose *Optimistic Double Execution* to speed up the fault free-case of Double Execution.

2.1.3.1 Execution Principle

In detail, *Optimistic Double Execution* works as follows:

1. **Thread start:** Similar to Pessimistic Double Execution, the redundant threads are started on core availability.
2. **Thread execution:** During execution, the leading thread's core buffers all `twrite` operations in its core-local write buffer. Simultaneously, the L-TSU creates a CRC-32 signature of all `twrites` (and `twriteps`). The L-TSU of the trailing thread's core also creates a CRC-32 signature of all `twrites`, however, the (`twrites`) of the leading thread are discarded immediately after signature creation.
3. **Thread end:** When the leading thread has finished execution, indicated by `tdestroy`, the L-FDU sends the CRC-32 signature to the D-FDU. Furthermore, the core executing the

leading thread immediately commits the results to global memory without waiting for the D-FDU to compare both signatures and the synchronization counts of the succeeding threads are decremented. Unlike in pessimistic Double Execution, the D-TSU can immediately start threads, when their synchronization count has reached zero without waiting for the redundant threads.

4. **Result comparison:** The D-FDU waits for the signatures of both the leading and the trailing threads and compares them.
5. **Thread commit or recover:** In the case of a non-faulty execution, the D-TSU can proceed as usual. If a fault was detected, the FDU must trigger the global system recovery, described in Section 2.2.

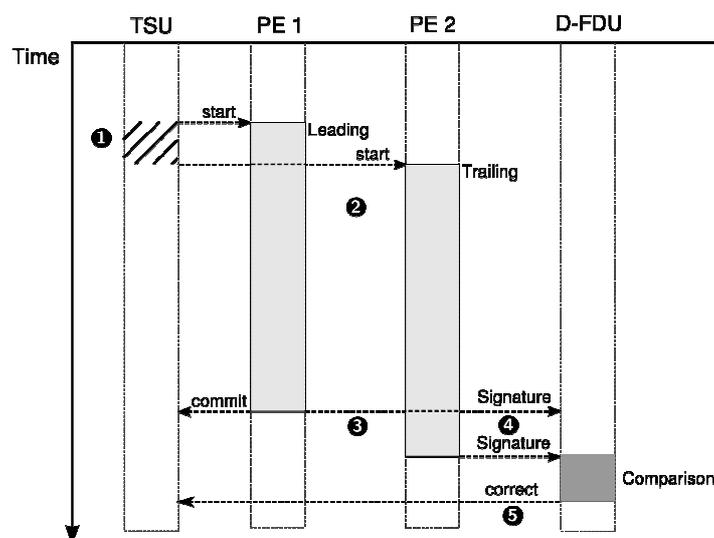


Figure 4: Optimistic Double Execution with reduced waiting time.

2.1.3.2 Performance Overhead

Compared to the pessimistic variant, optimistic Double Execution increases the utilization of additional parallelism, since unchecked threads are allowed to commit their results and to optimistically spawn succeeding threads. Furthermore, result checking can be deferred to a later point in time without reliability implications. The direct commit of the leading thread completely eliminates the core blocking and hence the idle time.

In other words, optimistic Double Execution may reduce the performance overhead induced by *Pessimistic Double Execution*. The increased thread execution time is reduced, since the leading thread is allowed to immediately commit to the global system state without waiting for the trailing thread. As a consequence, the trailing thread's core does not need to be blocked to wait for signature comparison by the FDU, since the results have already been committed by the leading thread.

2.1.3.3 Error Containment

The anticipated commit of the leading thread, makes it possible that erroneous results have already been written to the global system state and may be consumed by subsequent threads. As such, errors may be spread over the whole system until detection.

Since errors can be at any address in the distributed shared memory after detection, the recovery mechanism must recover the complete global memory across DF-thread boundaries. In Section 2.2, we propose an efficient global system recovery mechanism for single-node and multi-node TERAFLUX systems, based on the coarse-grained dataflow execution model, which can be used together with *optimistic Double Execution*.

2.2 Recovery

The TERAFLUX dataflow execution model does not only provide advantages for fault detection, but also for recovery from faults.

In this section, we will describe a global recovery mechanism, which exploits the coarse-grained dataflow execution model. Since we must assume that fault rates in future systems will raise, we propose a hierarchical fault recovery approach. This means, that we expect that some components, like the cores' pipelines and the local caches are more often affected by faults. In this sense, we use a local recovery mechanisms at core level, which exploits the functional semantic of the dataflow execution model to restart threads. This enables all other cores to make forward progress without global synchronization and wasting fault free computations due to recovery actions. On the other side, some faults may have longer detection latencies than the shortest thread runtime. Furthermore, some faults may affect components, which are not covered by the sphere of recovery of the thread restart mechanism. For instance, intermittent faults within the D-TSU, may lead to wrong scheduling decisions or synchronization counts of the system. Furthermore, in some cases longer checkpoint intervals are required, e.g. to store the global system state on a fail-safe storage. As a consequence, we developed coordinated node local checkpointing, which is based on the TERAFLUX dataflow execution.

The optimistic Double Execution mechanism in particular requires a global recovery mechanism, since a dataflow thread is allowed to commit possibly faulty results to the main memory without waiting for the redundant thread to compare the results.

2.2.1 Global Error Recovery

Global Error Recovery in TERAFLUX implements a coordinated node-local backward error recovery scheme for a highly parallel architecture. Hence, the TERAFLUX system periodically creates global checkpoints, which describe a global state of the system. Coordinated means that the node's D-FDUs coordinate the point in time, when a checkpoint will be created, while node-local means that after the nodes have agreed on a checkpoint, each D-FDU will create a checkpoint of its node.

2.2.1.1 Node Checkpoints

The coordinated global checkpointing uses the node checkpointing already described in Deliverable D5.3. This Section presents a recap of node checkpointing.

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 (see next subsections for the evaluation).

After the checkpoint has been established all subsequent `twrites` to the checkpoint's memory region will be logged. This means, the D-TSU maintains a log of all changes to the thread frames within the checkpoint's memory region. Newly allocated thread frames must not be recovered and are created outside of the checkpoint's frame memories.

When a fault is detected, the D-TSU recovers to the last checkpoint by restoring the frame memory log and its backup context.

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, 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 complete main memory.

2.2.1.2 Coordinated Global Checkpoint

Creating a global checkpoint of the whole system is composed of two steps. First, all nodes synchronize to agree that a global checkpoint will be taken.

Therefore, all nodes complete the dataflow write operations currently in flight and forbid subsequent threads to commit their results to the distributed shared memory. Second, when all nodes have agreed on the checkpoint, each node starts to create a node-local checkpoint. After this has been done, the system can proceed with the thread execution.

To create a checkpoint, the system complies with the following checkpoint protocol to coordinate a new global checkpoint (see Figure 5):

- Synchronise
 1. Synchronise with all D-FDUs
 - D-TSUs finish their outstanding commits
 2. ACK from all D-FDUs
- Create Node Checkpoint
 3. Send messages to start checkpoint
 - All nodes create node checkpoints
 - During checkpoint creation local nodes can proceed without committing to main memory
 4. ACK that the new checkpoints have been created
- Resume Execution on all Nodes

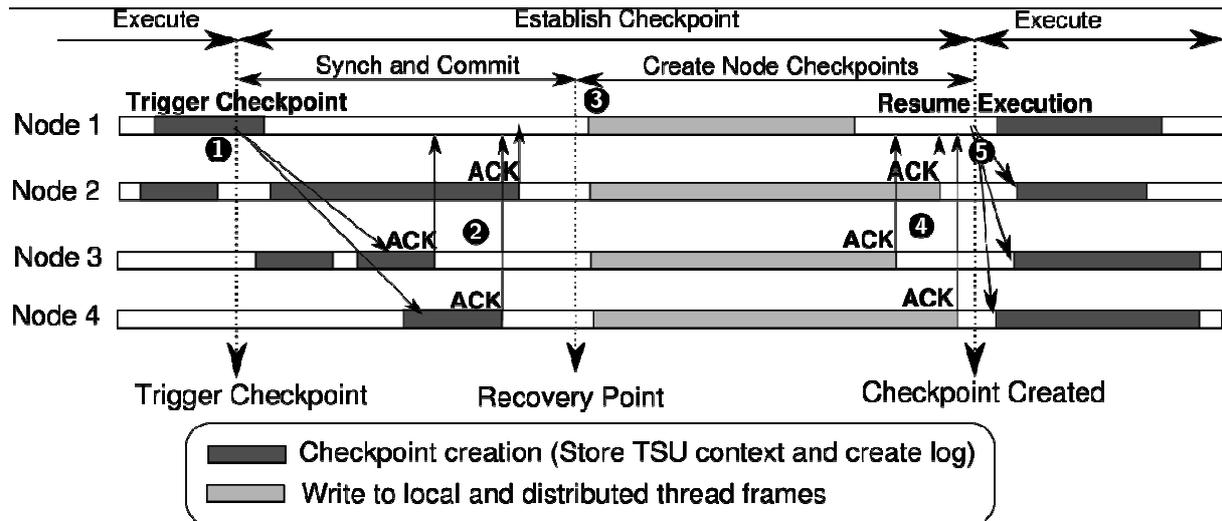


Figure 5: Establish Global Checkpoint for 4 Nodes.

When the D-FDU has detected a fault, it triggers the Global Error Recovery. The global recovery works in the same way as the global checkpoint mechanism:

- Synchronise
 1. Synchronise with all D-FDUs
 2. D-TSUs stop their execution
 3. ACK from all D-FDUs
- Restore Node Checkpoint
 1. Send messages to restore prior checkpoint
 2. All nodes rollback their state
 3. ACK from all D-FDUs that node recovery has been finished
- Resume Execution on all Nodes

2.3 Quantitative Results

In this section, we present quantitative results for optimistic and pessimistic Double Execution and local and global recovery mechanisms.

2.3.1 Simulation Methodology

The simulation results were obtained from the TERAFLUX simulator. We extended the TERAFLUX simulator to support optimistic and pessimistic Double Execution and thread restart recovery and global checkpointing. We used 4 Benchmarks, which were compiled with the TERAFLUX OpenStream compiler:

- Cholesky (256x256 Matrix, Block Size: 8x8)
- Fibonacci (31, cut off: 19)
- Seidel (256x256 Matrix, Block Size: 8x8)
- Sparse LU (256x256 Matrix, Block Size: 8x8)

Furthermore, we hand-coded a matrix multiplication (160x160 Matrix, Block Size: 16x16), which also uses the T*-instruction set extension (see also D6.2, 7.4 and [16]).

Finally, we show results that the redundant execution approach is able to scale with the number of nodes in the system.

2.3.1.1 System Configurations

We evaluated different single-node and multi-node configurations of the fault-tolerant TERAFLUX architecture.

2.3.1.2 Baseline Node Configuration Parameters

The baseline node configuration has 1, 2, 4, 8, 16, or 32 cores, respectively. Each core is operating at 1GHz and consists of an out-of-order pipeline with 5 stages, a maximum instruction window size of 128 and a maximum fetch and commit width of 2 instructions per cycle. The private cache hierarchy of each core is comprised of separate 64kB L1 instruction and data caches and a 256kB unified L2 cache. A special frame cache, which is exclusively used to store thread frames, is optional. The assumed memory bus latency is 30 cycles, while the memory latency is 150 cycles. Table 1 depicts the parameters of the baseline machine in more detail.

Table 1: Baseline Node Configuration

Parameters	Values
Cores	1, 2, 4, 8, 16, 32
Core Parameters	Out-of-Order, Pipeline length: 5, Fetch Width: 2, Commit Width: 2, Instruction Window: 128
L1 I- and D-Cache (private per core)	Size: 64kB, Line Size: 64, Sets: 2, Hit Latency: 1 cycle
Unified L2-Cache (private per core)	Size: 256kB, Line Size: 64, Sets: 16, Hit Latency: 13 cycles
Frame Memory (FM) (private per core)	Size: 64kB, Line Size: 64, Sets: 2, Hit Latency: 1 cycle
Memory Bus Latency(L2/FM to memory)	30 cycles
Memory Latency	150 cycles

2.3.1.3 TERAFLUX specific parameter

For the TERAFLUX specific parameters we assume that t_{writes} to write buffer take 3 cycles, while t_{writes} to local main memory take 30 cycles, in average. For the inter-node t_{writes} , we assume 150 cycles.

Table 2: TERAFLUX-specific Parameters

Parameters	Values
t_{write} inter-node Latency	150 cycles
t_{write} Latency (write buffer)	3 cycles
t_{write} Latency (commit to memory)	30 cycles
$t_{schedule}$ Latency	40 cycles
$t_{destroy}$ Latency	40 cycle

We simulated single node configurations with 1, 2, 4, 8, 16, and 32 cores.

We also simulated different multi-node configurations with 16 cores from one to 8 nodes in order to determine the scalability of our fault tolerance solutions.

2.3.2 Fault-free Execution

This Section presents the overhead for pessimistic and optimistic Double Execution and optimistic Double Execution in combination with node checkpointing.

2.3.2.1.1 Execution Overhead for Double Execution

It is inherent to all redundant execution approaches that they consume twice of the resources compared to a non-fault-tolerant execution. Since this overhead is inevitable, we compare Double Execution in relation to a non-fault tolerant execution with half of the cores.

The *Execution Overhead* describes the fraction of increase in execution time (in case of no-faults) compared to a non-fault-tolerant execution with half of the cores. Figure 6 shows the results for single-node execution for pessimistic Double Execution. It can be seen that the overhead for pessimistic Double Execution without global checkpointing induces an overhead between 2% and 23% compared to a non-fault tolerant execution with half of the cores. Furthermore, the results depict that the overhead remains constant even when the cores per node are scaled-up.

As described in Section 2.3, the main overhead for pessimistic Double Execution is induced by the increased idle time for the result comparison of the D-FDU. This is also the reason for the higher overhead for Seidel, Cholesky, and Sparse LU, since these benchmarks have a high number of inter-thread dependencies, which prevent subsequent threads to get started until the trailing thread has committed.

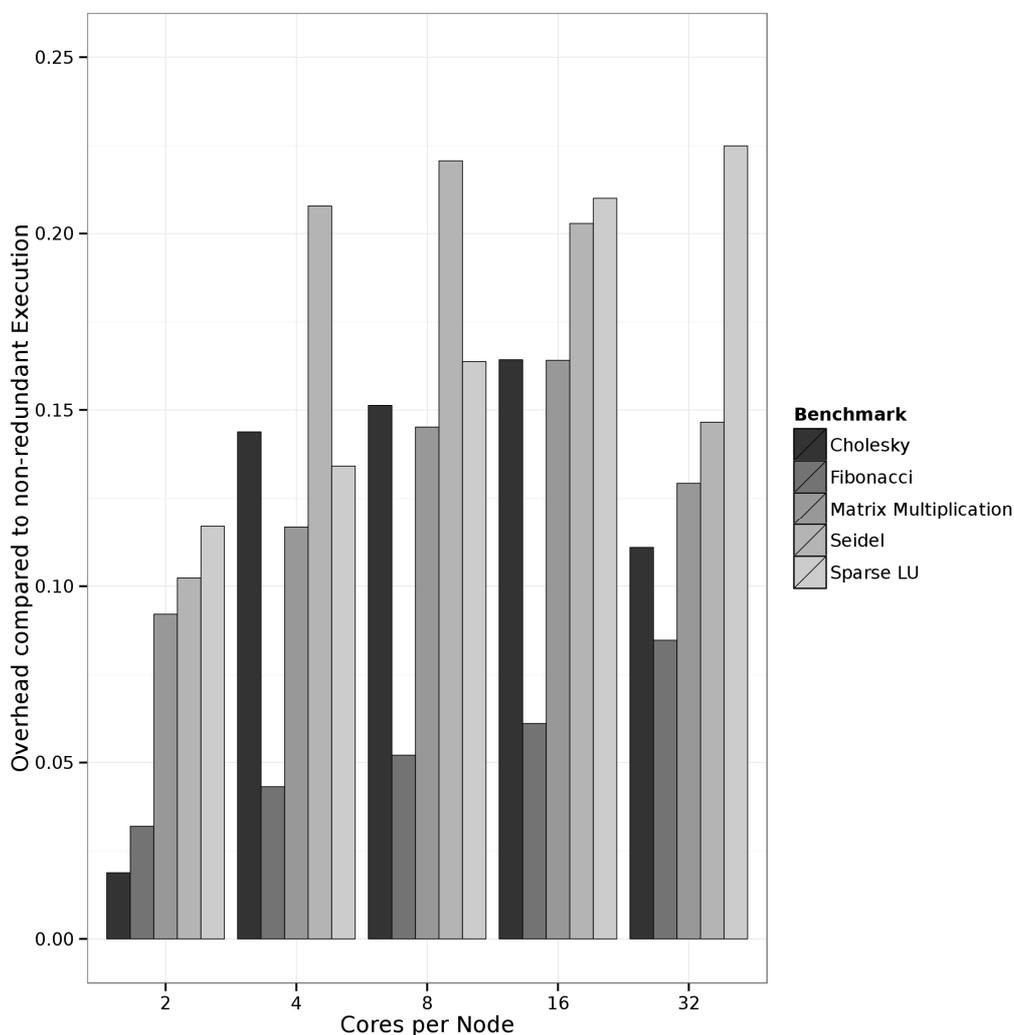


Figure 6: Execution Overhead of pessimistic Double Execution compared to non-redundant execution with half of the cores per node.

Figure 7 depicts the overhead for optimistic Double Execution. Compared to pessimistic Double Execution, the execution overhead can be reduced for all benchmarks. In particular, Cholesky and Seidel can now better utilize the node. This results from the effect that Cholesky and Seidel are not able to fully utilize the cores over the complete execution time (cf. Table 3), due to parts with low parallelism. However, the idle cores can be used by the optimistic Double Execution, since the trailing threads have no data dependencies and must never wait for input results. Instead, they can be immediately scheduled for execution, whenever a core becomes available for execution. The Seidel kernel, by contrast, is not able to fully utilize the node, due to the small workload. As a result, optimistic Double Execution is also not able to fully utilize the nodes with higher core counts. In this case, the speedup of optimistic Double Execution remains low.

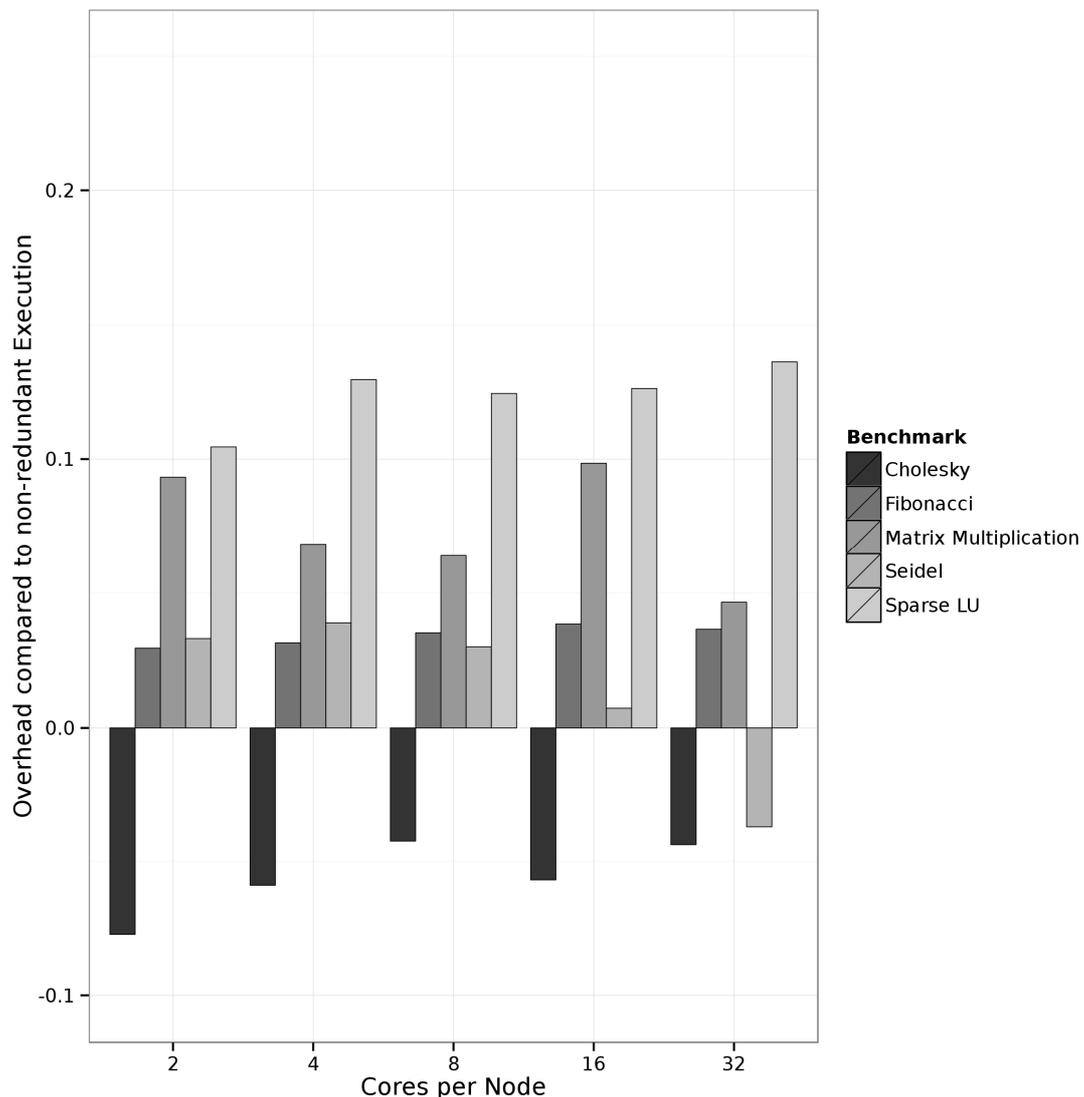


Figure 7: Execution Overhead of optimistic Double Execution compared to non-redundant Execution with half of the cores per node.

Table 3 shows the node utilization for pessimistic and optimistic Double Execution. It can be seen that optimistic Double Execution increases the node utilization for all benchmarks. The increased node utilization comes from the reduced idle time, which is eliminated by Double Execution. Furthermore, the Cholesky and Seidel benchmark are not able to fully utilize the node, which in turn leads to a negative overhead, since Double Execution can use the under-utilized cores for the redundant threads.

Table 3: Node Utilization of pessimistic and optimistic Double Execution (no-faults)

Benchmark/Cores	2		4		8		16		32	
	Pes.	Opt.								
Cholesky	94%	99%	88%	98%	87%	96%	82%	92%	74%	80%
Fibonacci	99%	99%	99%	99%	99%	99%	99%	99%	99%	99%
Matmul	99%	99%	99%	99%	99%	99%	98%	99%	97%	97%
Seidel	97%	99%	93%	99%	93%	99%	92%	98%	90%	96%
Sparse LU	97%	99%	98%	99%	98%	99%	95%	97%	89%	90%

Table 4 depicts the speedup of optimistic Double Execution for all benchmarks compared to pessimistic Double Execution.

Table 4: Speedup of optimistic Double Execution compared to pessimistic Double Execution (no-faults).

Benchmark/Cores	2	4	8	16	32
Cholesky	9%	17%	16%	18%	14%
Fibonacci	0%	1%	1%	2%	4%
Matmul	0%	4%	7%	5%	7%
Seidel	6%	13%	15%	16%	16%
Sparse LU	1%	0%	3%	7%	7%

2.3.2.2 Node Checkpointing Overhead

As mentioned in Section 2.1.3, optimistic Double Execution cannot use thread restart recovery, since possible faulty results may be committed to the main memory before the fault can be detected by the D-FDU. Therefore, we use node checkpointing in the case of optimistic Double Execution to recover from faults. However, node checkpointing induces additional overhead, when a checkpoint is created, since the D-TSU finishes all thread commits and prevent subsequent threads from committing.

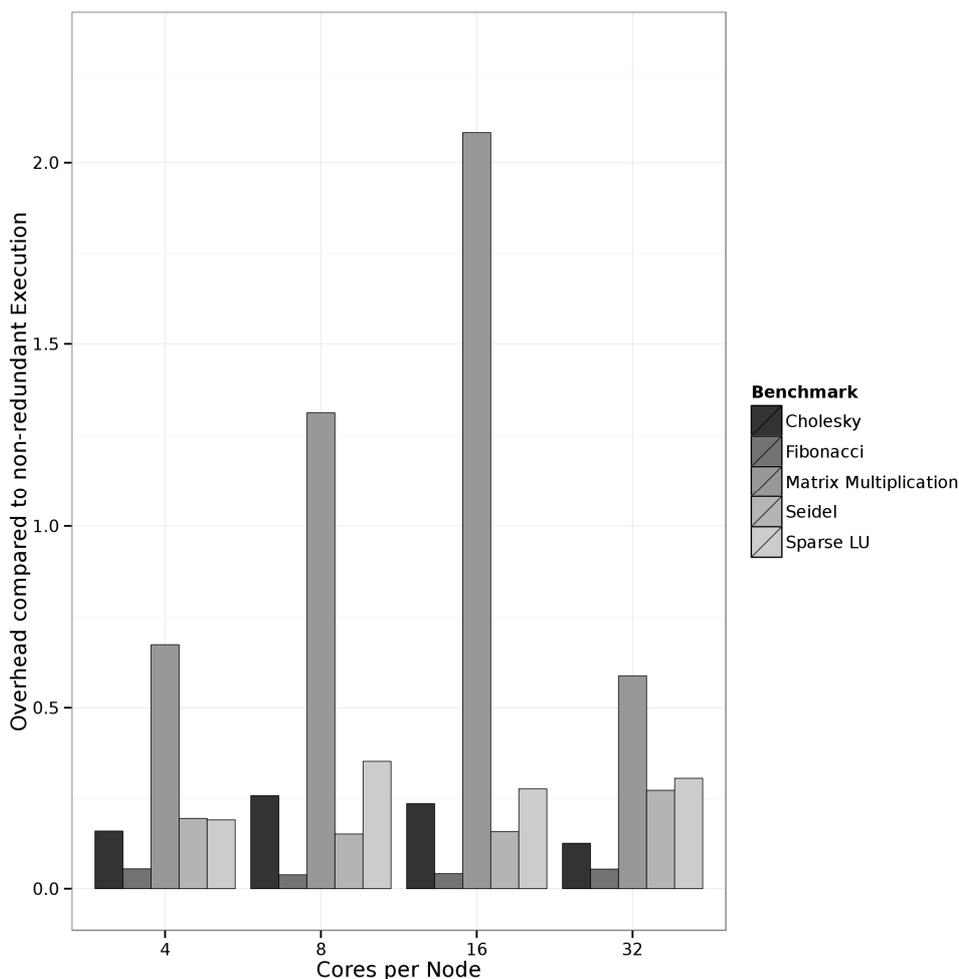


Figure 8: Overhead of speculative optimistic Double Execution with node checkpointing at an interval of 10,000 cycles compared to non-redundant Execution.

Figure 8 shows the overhead for speculative optimistic Double Execution in combination with node checkpointing at an interval of 10,000 cycles. Compared to the non-redundant execution, the overhead is between 10% for Fibonacci and over 200% for Matrix Multiplication. The main reason for high overhead in the case of Matrix Multiplication is the high number of `twrites` in the benchmark. This increases the possibility of core idle times due to checkpoint creation. By contrast, Fibonacci, which has a low communication to computation ratio, shows low overhead for node checkpointing. The reason for the lower overhead in the 32-cores node comes from the low utilization in the regular dataflow case.

Table 5 shows the overhead of optimistic Double Execution and node checkpointing at an interval of 10,000 cycles compared to pessimistic Double Execution without node checkpointing. Based on these results, it can be seen that node checkpointing can lead to high performance degradation for most dataflow benchmarks in case of checkpoint intervals of 10,000 cycles or below.

We therefore conclude that optimistic execution may introduce a high overhead for short node checkpoint intervals.

Table 5: Overhead of optimistic Double Execution compared to pessimistic Double Execution without node checkpoints.

Benchmark/Cores	4	8	16	32
Cholesky	1.2%	9.2%	6.2%	1.2%
Fibonacci	1%	1.3%	1.8%	-2.8%
Matmul	50%	101.8%	165%	40.5%
Seidel	-0.9%	-5.7%	-3.8%	11%
Sparse LU	5%	16.2%	5.5%	6.6%

2.3.3 Execution under Faults

Since we must assume increasing fault rates in future parallel architectures, we also simulated the overhead induced by a very high transient fault rate of 0.01 faults per second.

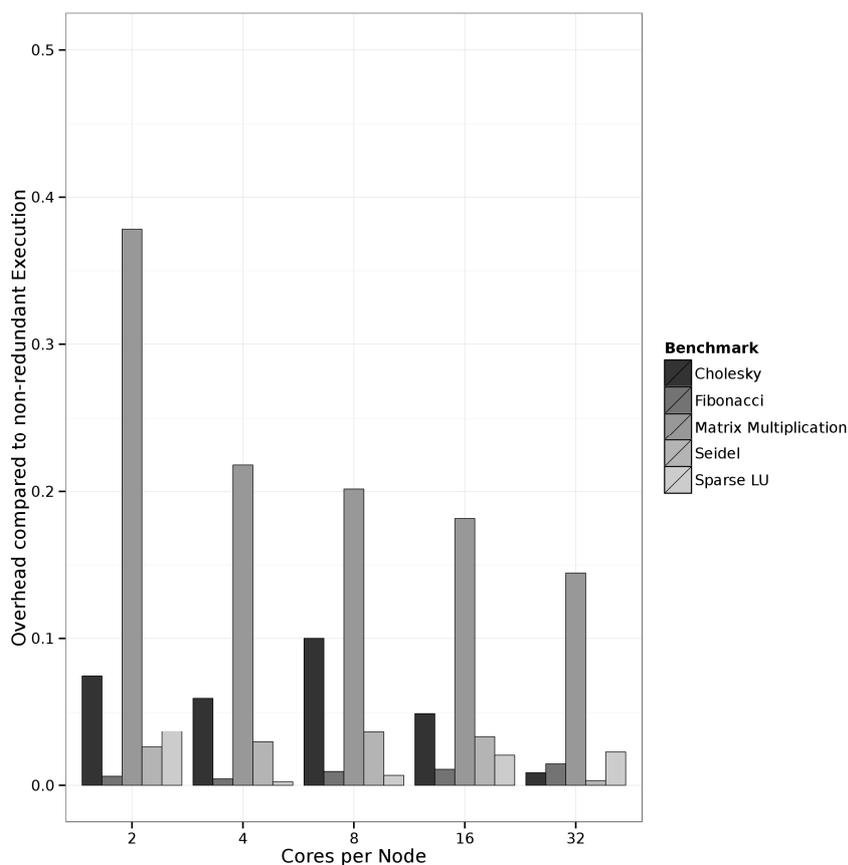


Figure 9: Overhead of pessimistic Double Execution in the case of a fault rate of 0.01 faults per second compared to non-faulty pessimistic Double Execution.

Figure 9 shows the execution overhead for pessimistic Double Execution using the thread restart recovery. It can be seen that faults lead to an overhead of between 1% and 38%. The high overhead of Matrix Multiplication comes from the long uniform dataflow threads of Matrix Multiplication, which lead to wasted execution time in the case of a thread rollback. Additionally, the results show that faults have lower performance impact in nodes with more cores. This comes from the fact that parallel architectures can exploit more possible idle resources in the case of a thread restart.

Figure 10 depicts the overhead of optimistic Double Execution. Due to side-effects in some benchmarks, we were only able to simulate Matrix Multiplication and Sparse LU for node checkpointing and optimistic Double Execution. As expected, the recovery of Matrix Multiplication leads to high execution overhead of over 20%. This overhead is mainly induced by the long dataflow threads of Matrix Multiplication in combination with the node recovery leading to more wasted execution cycles by also recovering fault free threads.

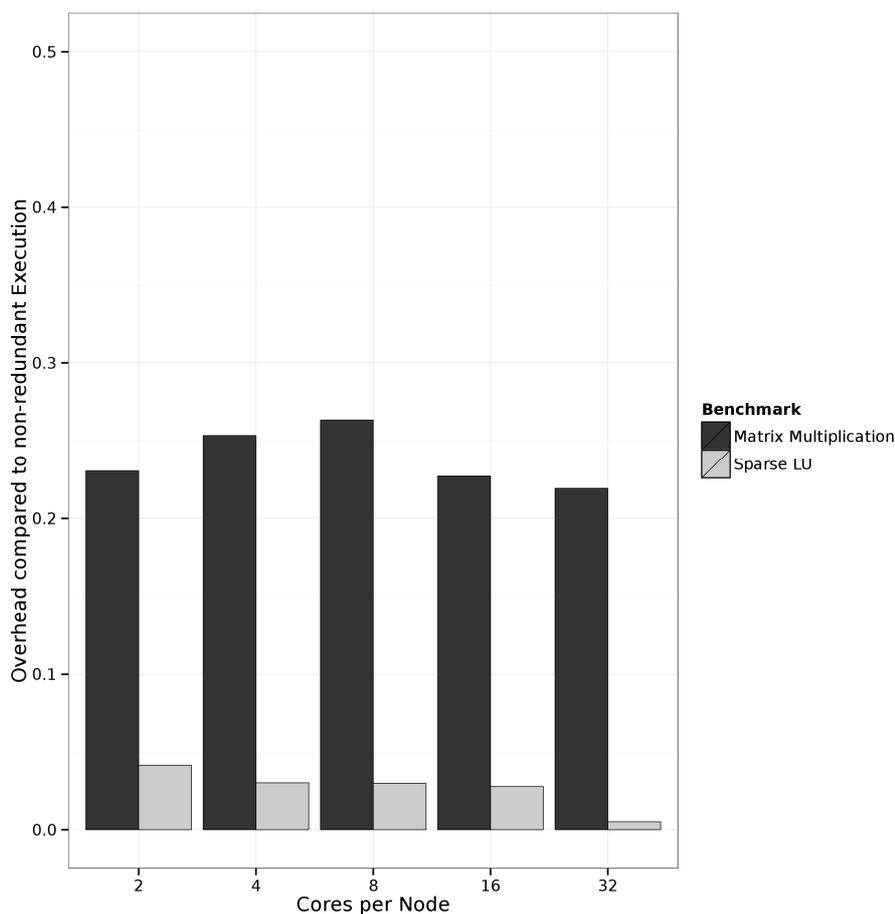


Figure 10: Overhead of optimistic Double Execution in the case of a fault rate of 0.01 faults per second compared to non-redundant execution without faults.

2.3.4 Multi Node Behavior

Fault Detection Mechanisms for parallel architectures must be also efficiently scalable with the number of cores in the system. Since both Double Execution variants restrict redundant execution to

one node and only the trailing or the leading thread is allowed to commit its results, the inter-node communication is not influenced by the Double Execution mechanisms. However, the high inter-node commit latency may result in under-utilized cores, since cores are blocked as long as the write buffer is written to thread frames of waiting threads in different nodes. This may lead to long idle times, in the case of numerous inter-node `twrites`. Please note that we assume inter-node write latency in our simulations of 150 cycles, as shown in Table 2. We simulated optimistic and pessimistic Double Execution on 1 to 8 nodes, where each node has 16 cores. This means, we simulated Double Execution on systems from 16 cores to 128 cores.

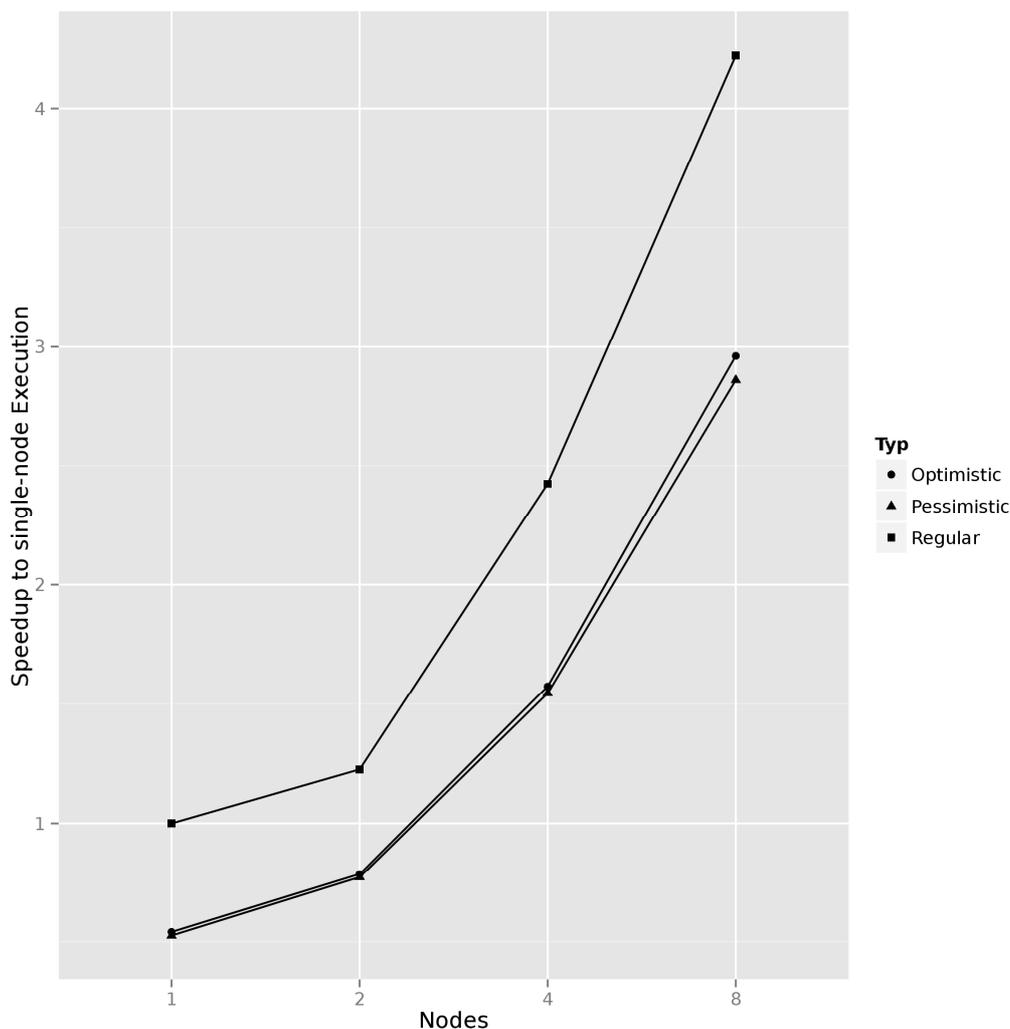


Figure 11: Scalability of pessimistic and optimistic Double Execution for 1 to 8 nodes. Each node comprises 16 cores.

Figure 11 shows the scalability of the regular dataflow execution, pessimistic and optimistic Double Execution of Matrix Multiplication normalized to the execution on one node. Since Matrix Multiplication uses numerous `twrite` operations for the write-back of the submatrices into the result matrix, Matrix Multiply suffers from higher idle times in this case.

Nevertheless, it can be seen that both Double Execution variants are able to scale in the same way as the regular dataflow execution.

3 NoC-Level Fault Tolerance in TERAFLUX

In this section we will show improvements and extensions to the work previously reported in the Deliverables D5.2 and D5.3 and in [8]. In the previous work, we already discussed by means of an analytical approximation the impact of a fault detection mechanism based status message with respect to application messages (done in D5.2). In addition to that, we designed and implemented a fault localization mechanism, which only uses the timing behavior of the status messages from the message based fault detection as an indicator for faults within the interconnection network (done in D5.3). Furthermore we started in the Deliverable D5.3 a brief consideration of different network topologies and on how they are suitable from a fault tolerance perspective.

Given this previous work, we extended our investigations by a quantitative evaluation by means of network throughput, delay, and jitter regarding the impact of the status messages for the fault detection/fault localization techniques. We present and discuss the results from that evaluation in Section 3.1. Although, we already demonstrated the functionality of the fault localization technique for the interconnection network, we extended the fault model from single faults to multiple faults within the interconnection network. This includes an extensive investigation on fault patterns, which may prohibit a precise fault localization of any given faults within the network. Therefore, we include spatial and temporal distributions for the fault patterns (Section 3.2). The section concludes with a discussion on an extension of the assumed network on chip topology towards a toroidal topology in Section 3.3.

3.1 Impact of HB Messages on App. Messages

The maximum load that can be handled by a network has a strong influence on the overall performance of a processor. An undersized interconnection network can quickly lead to long periods in which a processor is idle while waiting for a response message that is stuck in the network. Such waiting periods for application messages can be amplified by the prioritized processing of heartbeat messages. Since the heartbeat messages have the highest priority in the network, it is expected that it takes more time on average for an application messages to travel the path through the network. These waiting periods are – if the response time of the communication partner itself is ignored – basically dependent on the metrics throughput, latency and jitter. The definitions and explanations of the evaluation metrics are based, unless otherwise indicated, on [9] [10].

3.1.1 Metrics of interest

The **throughput** describes the maximum amount of data that a network can receive and process. Emitting more data than the maximum throughput to the network will effectively lead to network saturation. That is, more messages are injected to the network, than messages that drain from it. If one keeps the injection rate at this level, the message delivery will suffer very high latencies and can also lead to deadlocks.

The **latency** is defined as the period of time that is needed to transmit a full message. The period starts at the creation of the first part of the message at the sending core¹ and ends with the full reception of the last part of the message. At low network loads, the data can be transmitted almost without delay

¹ Even before actually injecting the message to the network

and therefore latency is dominated only by the router's processing pipeline and the delay of the interconnections between the routers. If the load of the network, however, reaches the maximum throughput, it is possible that the latency in the network greatly increases².

Both metrics depend essentially on two factors. The first factor is the injection rate at which the messages are generated during a simulation. The injection rate is thus the temporal distribution of message generation in the simulation. The second factor is the actual used traffic pattern. It determines the communication pairs of the processor cores. The traffic patterns are thus the spatial distribution of the messages in the simulation. Both factors have a great impact on both, the maximum throughput and the latency. Some traffic patterns allow higher injection rates before the network saturates and consequentially a higher maximum throughput than others.

Jitter is an important indicator to determine the uniform rate of message processing by the network. Applications that expect a constant and uniform data stream may be stalled from high jitter values, as parts of the expected messages do not arrive after a specific period of time. Especially execution models similar to a pipelined execution are prone to high jitter values. The effects are noticeable, as the high jitter values may stall the entire progress of a program in a certain pipeline stage. An example, in which the user experiences a fairly quick high jitter, is the video decoding. High jitter values, can easily result to the famous "picture judder", which stalls the motion picture playback temporarily.

3.1.2 Evaluation Methodology

All NoC results were obtained from an extended version of the NOXIM simulator³. The simulations cover different experimental setups regarding the use of heartbeat messages and using different routing strategies. The setup ranges from:

- Baseline measurements without heartbeat messages.
- With heartbeat messages
 - XY routing strategy
 - Staircase routing strategy

The application messages are always routed with the XY routing strategy through the network.

The simulated scenario was the communication structure of a processor with 25 cores. The cores themselves only act as a sender and a receiver module. For application messages, we applied two different traffic patterns, which are Hot-Spot and Random [11] [9]. Additionally several injection rates ranging from 0.00001Flits/Core/Cycle to the throughput saturation point have been used. The actual execution of a program is not supported and is not part of the evaluation. However, as part of the evaluation the processor cores take different roles which are partly defined by the traffic pattern:

- Acting as FDU: One core takes the role of the FDU and sends out heartbeat messages implementing the poll-method to gather the status information from the cores. Furthermore,

² For simulation purposes we sized the injection buffer of a processor core to infinite. That theoretically leads at extreme high injection rates to an infinite latency value.

³ <http://noxim.sourceforge.net/>

the FDU core is configured with a proper TDMA-Scheme. No other communication is allowed for this type of core.

- Asynchronous communication: The processor cores consume application messages, but they do not answer them directly. This can be used to simulate loads for different execution models, such as pipelining and fork join execution models.
- Synchronous communication: The processor cores consume application messages and answer them. This simulates the behavior of a client-server execution model or the access to other devices on the chip, such as memory controller and I/O controller.

In addition, all processor cores answer the heartbeat messages from the FDU with their own heartbeat messages.

From the interconnection point of view, each processor core is connected to one router and uses it to communicate with other components on the chip. The cores and routers are coupled by a bi-directional link with the router, which allows a full-duplex communication between the core and router. The same coupling has also been established for the link between the routers, wherein the topology of the connected router corresponds to a homogeneous two dimensional mesh.

A router internally connects each link in a demux unit which redirects the incoming messages based on a packet flag into two input buffers. The reason for the two input buffers is the spatial isolation among application messages and heartbeat messages as required by the fault detection techniques. In addition, the routing logic is applied at the arrival of a message. Since the routing logic of the XY strategy and the Staircase strategy is relatively lightweight, it can be assumed that it returns the result of the message routing function within the same network clock cycle.

The flow control is based on a two staged wormhole switching, which has been extended for the usage of Quality of Service (QoS). The arbiter unit firstly checks the contents of the buffer for high priority messages (heartbeat messages) and reserves at the same time the I/O interface of the router's internal crossbar. In the second stage, this process is repeated for the lower priority messages. Again, it can be assumed that the channel arbitration and the actual traversal of messages are completed in one network clock cycle. Thus, for the simulation of a router pipeline, the router needs two network clock cycles to process a heartbeat message. Application messages need, due to their variable size of 2-4 flits, 4-8 network clock cycles. However, this assumes that the message is not blocked by another message due to concurrent access to the same output link.

The traffic pattern for the application messages were carried out with a number of different injection rates as a simulation parameter. This injection rate controls the network load varying from 0.00001Flits/Core/Cycle to the theoretical maximum network throughput for a given traffic pattern. The aim of the different injection rates is to determine the saturation point of the network. To determine this point of the network configuration the average throughput was observed. A stagnating throughput by simultaneously increasing injection rates signals that more flits are generated than the network is capable to deliver and thus the network saturates. Beyond the saturation point, increasing injection rates also generate sharply rising message latencies. At this point the messages are loaded in the upload buffer of the processor core straight after creation and stay there until the connected router can process them from there.

As indicated before, a full buffer would cause the sender not to create new messages and thus changing actual the traffic pattern. Therefore an infinitely large buffer for the sender was implemented to ensure the consistency of the traffic pattern. This solution may be unrealistic in the real world; however, it is a viable solution for simulation purposes [9] [10].

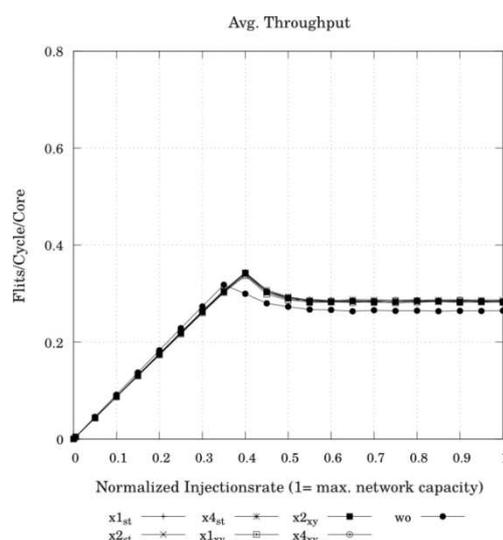
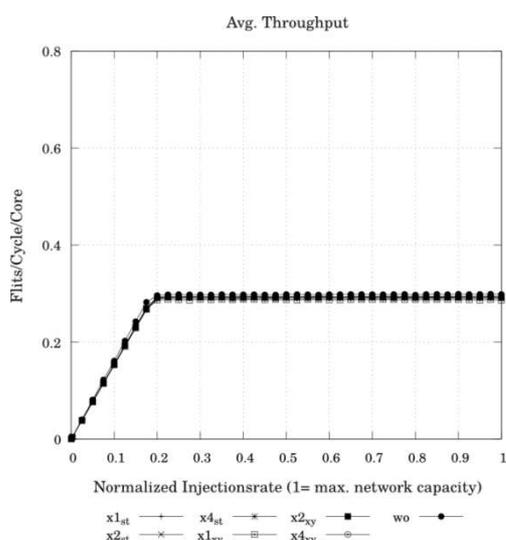
In addition to the traffic patterns for application messages, a closely staggered TDMA-Scheme was used to generate different network loads with the prioritized heartbeat messages. The scheme x1 produces the narrowest heartbeat pattern, thus ensuring the highest load on the links nearby the FDU. In addition, the schemes x2 (load halved) and x4 (load quartered) have been used as less closely staggered patterns.

3.1.3 Quantification

3.1.3.1 Throughput of Application messages

As already described, the throughput of a specific traffic pattern is an important indicator of the effectiveness of a communication network. The throughput shows with stagnant values that the saturation point of the network has been reached, and therefore the upper limits for the subsequent analysis of latency and jitter. A total of three series of experiments were performed for each traffic pattern, where two of them include heartbeat messages routed with XY (R_{XY}) or the staircase strategy (R_{ST}), respectively. The third series of experiments was carried out without heartbeat messages and serves as a baseline for comparison.

In order to put the different traffic patterns among themselves in relation, the injection rates were normalized in relation to the respective network capacity⁴. Figures 12 and 13 summarize the simulation results in terms of throughput for the traffic patterns Random and Hot-Spot. For better illustration and comparison, the injection rates were (X-axis) normalized to the respective traffic pattern. On the Y-axis the amount of draining flits per network clock cycle and processor core is



⁴ Figure 12: Throughput of application messages throughput Figure 13: Throughput of application messages under traffic pattern Hotspot.

Deliverable number: **D5.4**

Deliverable name: **System Integration Analysis, Measurement and Tuning of the Reliability System**

shown.

The traffic pattern Random shows a constant stagnant throughput after reaching the saturation point. This is also referred to as a stable network regarding saturation. Looking at the throughput performance under Hot-Spot traffic pattern, one can observe a clear drop in the throughput of 0.08 to 0.07, but stabilizing again with rising injection rates. The drop in throughput is due to the fact that about 60% of the total communication must be processed directly on the connecting lines, which are already heavily burdened by the heartbeat messages. Additionally, there is no fairness in the treatment between the heartbeats and the application messages due to the prioritization of heartbeat messages. Considering both facts explain the loss of throughput performance. Since for the traffic pattern Random all communication pairs are determined by a uniformly distributed probability, also the network load is uniformly distributed. This ensures the network stability after reaching the saturation point.

3.1.3.2 Latency of Application messages

In addition to the ability to transmit as many messages as possible in parallel, the speed at which a single flit is handled by the network is also critical to the performance of the network. In the absence of heartbeat messages the expected average latency for this network configuration is ideally at 12 cycles⁵ [10]. Figure 14 and Figure 15 show the results of the simulation with respect to the latency of application messages. In addition, the x-axis has a logarithmic scale for the sake of better representation. Even with the use of closely staggered heartbeat messages ($x1_{xy}$ and $x1_{st}$); there is no significant difference between the baseline simulation and those with heartbeat messages. While this observation was mainly expected at lower injection rates, the uniform development of latency close to the saturation point of the network is unusual. At these injection rates significant backlogs of application messages is expected and heartbeat messages should have an obvious impact on

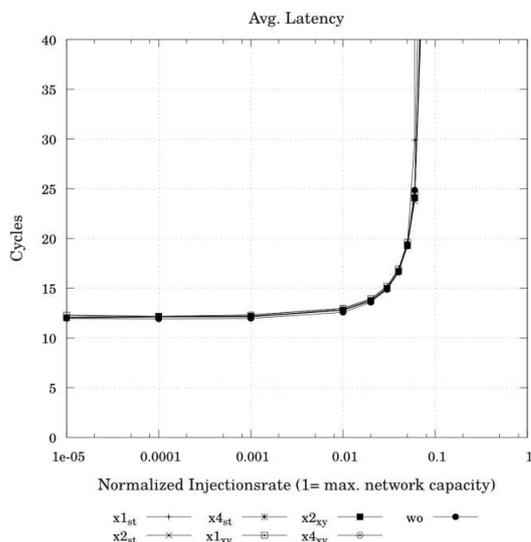


Figure 14: Application message latency under traffic pattern Random.

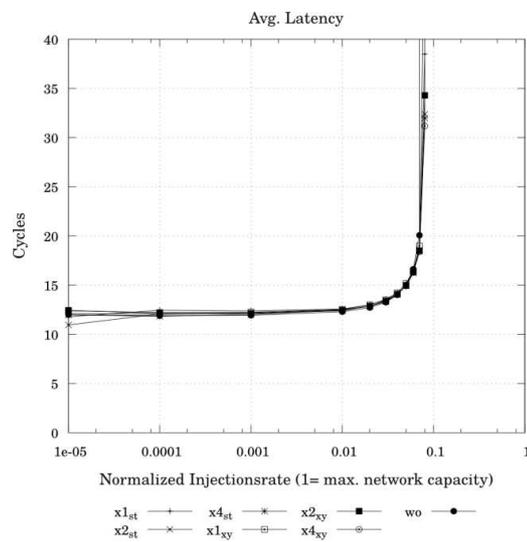


Figure 15: Application message latency under traffic pattern Hotspot.

⁵ $\Delta = \text{Pipeline length} \cdot (H_{avg} + \text{Drain}) + 1$, with H_{avg} as average path length in Hops

application messages.

The reason for this uniform development of latency can be answered by the router internal input buffers. The size of the input buffer equals the maximum length of the application messages. Thus, this means that a blocked message can be held entirely by the input buffer of a router and thus the message occupies only the resources of a single router. Thereby, the overall likelihood for backlogs decreases, even for injection rates close to the saturation point. For the sake of completeness it should be mentioned that further (equally prioritized) virtual channels may reduce the effect of backlog for application messages even with larger message lengths.

3.1.3.3 Jitter of Application messages

The study of jitter is carried out differently. Along with the traffic pattern and the injection rate, we also put the path length into consideration. The results of the investigation are based on the measurement of the maximum delay, which an application message was exposed during the simulation. Therefore, the measured transmission latency and the path length of a message were determined. With the help of the path length $|p|$, the measured latency can be adjusted and thus gives the duration of the delay in which the message could not make any progress in the network. Finally, the maximum delay was determined for all path lengths.

Referring to the results we mention beforehand that messages with a mean path length of 3-6 hops are more affected by the delays than the messages with path lengths of 1-2 hops and 7-8 hops. That statement holds also for all traffic patterns. This effect has two reasons. The first aspect of this effect is the time a message spent in the network. Messages with short paths (1-2 hops) stay a short time in the network. Hence, on average, the probability of multiple delays through heartbeat messages is lower for application messages with short paths than with longer paths. The second aspect is the route through the network. Application messages with a path length of 7-8 hops are indeed a longer time within the network. However, the XY routing strategy generates paths that run along the edge of the network. In this area, the presence of Heartbeat messages is fairly low.

The Figure 16, Figure 17, and Figure 18 plot the respective values of maximum delay for the traffic pattern Random with different injection rates and ascending sorted by the path lengths. The x-axis divides the results in the respective path lengths. On the y-axis the maximum delay is shown, wherein

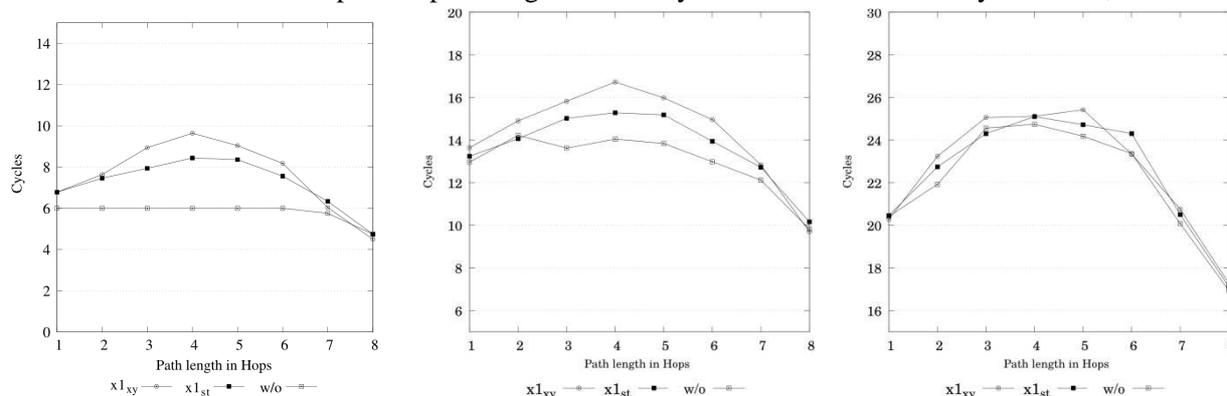


Figure 16: Jitter at 0.0001 for traffic pattern Random.

Figure 17: Jitter at 0.001 for traffic pattern Random.

Figure 18: Jitter at 0.01 for traffic pattern Random.

a delay having a value = 0 corresponds to the ideal latency without any delays for the given path length.

At low injection rates, a constant value of the maximum delay is observable for simulations without heartbeat messages. Hence, it is obvious, that the network is evenly loaded with application messages. The slight reduction of latencies starting from a path length of 7 hops can be explained by the above-mentioned routes messages with longer path lengths.

Significantly longer delays, however, are observable ($x1_{xy}$ and $x1_{st}$) from the results of the simulations with heartbeat messages. If the XY routing strategy is used for heartbeat messages, the maximum delay for application messages is about 60% higher than in the reference simulation without heartbeat messages. By using the Staircase routing strategy, the delays increase at most to about 40%.

With an increasing injection rate also the amount of application messages in the network increases. This results in frequent collisions of application messages with each other, whereby the maximum delay also in the reference simulation increases. This can be easily observed on the basis of Figure 17. The maximum delay increases from 6 clock cycles to 14 clock cycles. Furthermore, it can be seen that the values for simulations with heartbeat messages, do not scale to the same extend as it is the case of the simulation without heartbeat messages. The maximum delay recorded from the simulation with heartbeat messages is 20% (XY strategy) and 13% (Staircase strategy) higher compared to the reference simulation. This relative degradation of the maximum delay is due to the mutual collisions of the application messages. That means the delays become more dominated by the increasing rate of mutual collisions of application messages.

Just before the saturation point of the network, there is neither a significant difference between the simulation including heartbeat messages, nor for those without heartbeat messages (Figure 18). This means the proportion of collisions between application messages and heartbeat messages has decreased to a minimum and thus does not contribute to the apparent maximum delays.

As mentioned above, the traffic pattern Random is indeed a good measure for simulating networks with uniform load distribution, but a purely random load distribution rarely corresponds to a real application-driven communication pattern. In order to take this fact into account, the synthetic pattern *Hot-Spot* was added, and used in the simulation under the same conditions as previously Random. The

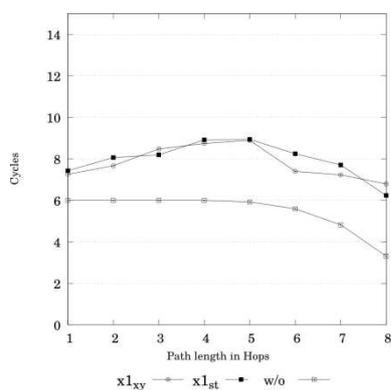


Figure 19: Jitter at 0.00001 for traffic pattern Hot-Spot.

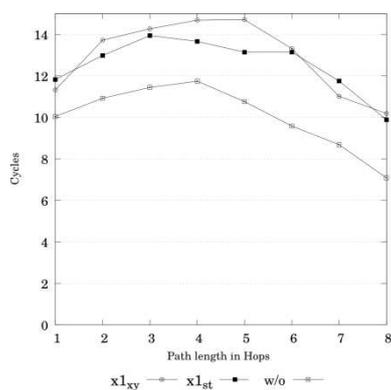


Figure 20: Jitter at 0.001 for traffic pattern Hot-Spot.

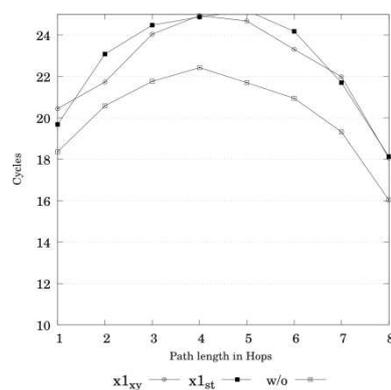


Figure 21: Jitter at 0.01 for traffic pattern Hot-Spot.

result with respect to the maximum delay is plotted in Figure 15.

The Hot-Spot traffic pattern used for application messages presents a notably scenario and produces different results compared to the former measurements regarding the convergence of values at higher injection rates. As we already pointed out, that the mutual collisions of application messages at higher injection rates become a dominant factor for the recorded maximum delays, we now observe different results. The duration of the delays for the application messages due to collisions with heartbeat messages grows stronger with the increasing injection rates, than previously observed. The reasons for that different behavior are the locations of the different hot spots and the respective 60% probability that an arbitrary processor core selects one of the hot spot cores as a destination. These locations were chosen so that they lie exactly on the axis of the FDU. Therefore, application messages using the XY routing strategy are forced to move through the center of the network. Since the probability is very high to be blocked by a heartbeat message, the influence of heartbeat messages to application messages is preserved even at high injection rates. This scenario also shows no significant differences for the delays using the different routing strategies for heartbeat messages. This was, however, expected in this test case, as the application messages were deliberately routed through the center of the network and thus the delays for both routing strategies could be predicted to be quite similar.

3.1.3.4 Quantification Summary

The results of the evaluation regarding the impact of heartbeat messages on application messages by applying the different TDMA schemes show that, on average, no significant impact is produced by the heartbeat messages to application messages (see Table 6: Overview of throughput, latency, and maximum delay for application messages). The average throughput and latency of the application messages are also in the presence of heartbeat messages in the network at the level of the simulations without heartbeat messages.

A closer inspection of application messages with respect to their maximum delays, however, shows that these delays can be reduced by the use of the Staircase routing strategy. Table 1 summarizes all relevant results regarding the maximum delays of application messages. Using the Staircase routing strategy for heartbeat messages and given the traffic pattern decreases the maximum delays up to 25-30% compared to the XY routing strategy. Almost identical are the maximum delays for the traffic pattern Hot-Spot. Here the difference between the two routing strategies is below one percent, which is due to the fact that around 60% of all traffic is routed through the FDU near routers.

The thesis and the analytical approximation from Deliverable 5.2, which states that a more uniform distributed load of heartbeat messages through the Staircase routing strategy can have a relaxing effect on the maximum delays of application messages was underpinned.

Table 6: Overview of throughput, latency, and maximum delay for application messages

Traffic Pattern	Avg. Throughput ⁶			Avg. Latency			Max. Delay ⁷		
	w/o	R _{XY}	R _{ST}	w/o	R _{XY}	R _{ST}	w/o	R _{XY}	R _{ST}
Random	20%			12 cycles			0%	63,3%	38,3%
Hot-Spot	35%	40%		13 cycles			0%	21,3%	21,6%

⁶ Percentage values of the theoretical maximum of throughput for this traffic pattern.

⁷ Percentage values of the maximum delays based on the baseline measurement without heartbeat messages.

3.2 Fault Localization with Multiple Faults within the NoC

As shown in the Deliverable D5.3 and [12] our investigation regarding the fault localization includes single faults within the 2D mesh based interconnection network. However, several studies show that multiple faults in the hardware are not seldom [13] [14] [15]. Therefore, we stressed the localization technique with multiple faults within the interconnection network and determined if this condition has an effect on the localization accuracy. The stress tests reveal that there are some specific fault patterns that are able to either mask other faults or create a *phantom* fault. In the last case, a specific fault pattern produces some sort of blind spot, which let a fault appear, although the fault is actually not present at this point in the interconnection network.

In this section, we will firstly describe the investigation methodology and next to that we group the problematic fault patterns, which have a negative impact to the localization accuracy. The section closes with a quantification of the amount of the problematic fault patterns based on different node sizes.

3.2.1 Investigation Methodology

For this investigation we combined fault patterns with temporal and spatial properties. For this purpose we firstly differentiate between the simultaneous and successive appearance of faults. The need to differentiate the temporal property of a fault comes from the fact that the FDU starts its search for local maxima within the status matrix of suspicious network components after a complete TDMA round has finished. In combination with the spatial properties the temporal patterns deliver different results regarding the localization accuracy.

To have a clear definition for the temporal patterns, we define two or more faults to be simultaneous, if both/all faults appear in the same TDMA round⁸. Successive faults are defined for any other temporal appearance.

The spatial patterns consist of two faults that have been applied to the interconnection network. We applied the faults in a systematic manner by permute all possible fault pairs. After each applied fault pair, we analyzed the resulting status matrix of faulty network components from the FDU and compared it with a hypothetical estimated matrix. If both matrices match, the fault pair is unproblematic. If they differ from each other, we found a problematic fault pair. In order to analyze the reason for the inaccuracy, we also investigated the status matrix of suspicious network components⁹. During the investigation of the status matrix of suspicious network components, we were able to identify recurrent fault placement patterns. Each pattern leads to its own inaccuracy for the localization method. We then categorized the faults and grouped them according to their spatial property.

From the problematic patterns with two faults, complex patterns can be easily generated by combining the different patterns to new ones. The more complex patterns are then also problematic for our localization method. For that reason we omitted permutations with a higher number of faults, since this does not change the results in the localization accuracy.

⁸ That is, after all cores have sent their Heartbeat-Messages including all different routing strategies as described in D5.3.

⁹ This matrix holds the "suspicious values" for each network component after a complete monitoring round.

3.2.2 Applying fault pairs to the NoC

As described above, the following discussion distinguishes the temporal pattern between simultaneously and successively. The analysis therefore initially begins with the application of simultaneous faults and discusses the spatial patterns, which have been identified as problematic. Subsequently, the discussion is repeated with respect to successive faults and the results are discussed.

3.2.2.1 Simultaneous faults in the NoC

The search for problematic patterns with simultaneously occurring faults yielded three groups, in which the spatial patterns could be classified:

- 1) *Faults in the immediate vicinity of the FDU.*
- 2) *Faults with common partial paths.*
- 3) *Closely spaced faults.*

For fault patterns of the group “*faults in the immediate vicinity of the FDU*” there are fault masking effects, in which a fault at a specific location in the network prevents the localization of other faults. The problem here is the proximity of a faulty link to the FDU. The closer the faulty component is next to the FDU, the more heartbeat messages are delayed by this component. This relatively large number of affected heartbeat messages creates a blind spot (gray triangle in Figure 22), which hides any other faults within this area.

To illustrate this problematic pattern, we show in Figure 22 a part of the network, in which two links are assumed to be faulty (marked as f_1 and f_2). In Figure 23 we show the corresponding status matrix of suspicious components. The black shadowed values within the matrix indicate the suspicious components gathered after a complete monitoring round¹⁰.

The following search for all local maxima within this matrix would reveal f_1 as a faulty component. But f_2 would not be identified as a faulty component. Additionally, as one can observe, the black shadowed values span a triangle shaped area over the net. Any fault within that area will be masked by the fault f_1 .

Another factor is that all heartbeat messages sent from within the blind spot are updated in the TDMA scheme and the expected arrival times are thereby adjusted according to the fault f_1 . It is therefore also

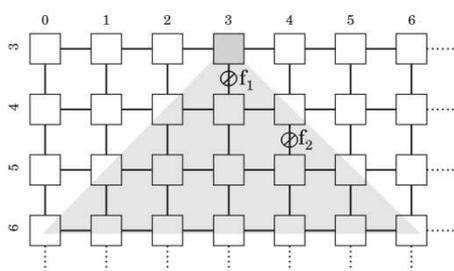


Figure 22: Blind spot due to f_1 .



Figure 23: Corresponding matrix of suspicious network components.

¹⁰ Please note that the gray rectangles in Figure 22 are for illustration purposes only and do not indicate that the faults have been already localized. The rectangles only try to assist the orientation.

not possible by subsequent monitoring rounds to detect additional faults from within the blind spot. This makes this fault pattern critical.

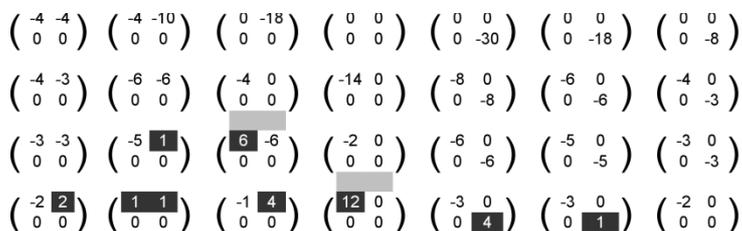


Figure 24: Resolved blind spot by moving the FDU to a different core.

A cost-effective solution to this problem is to let the FDU migrate from their current processor core to another core, if the FDU consists of software. But it should be noted, however, that the monitoring of processor cores has to be stopped in the first place in order to prevent a deadlocked FDU. After the last heartbeat messages of the current monitoring round has arrived at the FDU, the state of the FDU is frozen and transferred to another core. In this case, the contents of the status matrices are discarded because by migrating the FDU to another core also changes the distances of the processor cores to the FDU and thus the arrival times.

How fast this problematic fault pattern can be resolved is shown in Figure 24, which shows the resulting matrix after a complete round of monitoring. The applied fault pattern is the same as previously described. This time, however, the FDU has been shifted to a place in the network to the top and right. The fault f_1 is thus no longer in close proximity to the FDU and the effect of the blind spot has already vanished down enough that both faults are easily locatable by searching for the local maxima in the matrix.

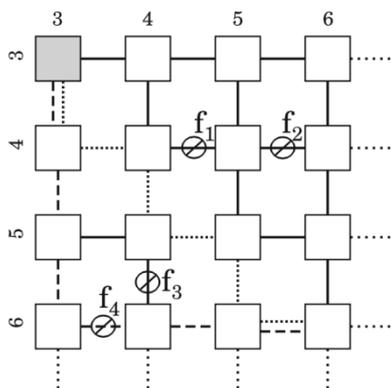


Figure 25: Blind spots due to f_1 and f_4 .

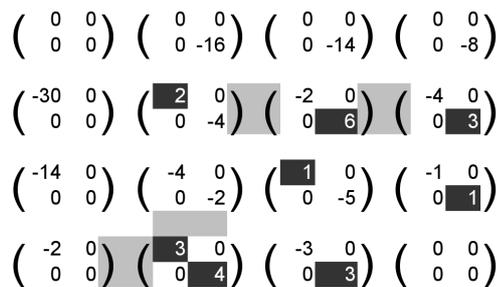


Figure 26: Corresponding matrix of suspicious network components.

Similar to the pattern above, the "Closely spaced faults" can also create blind spots. Again, faulty components produce a, albeit very small, blind spot. The effect can be observed in two different flavors:

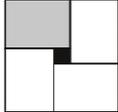
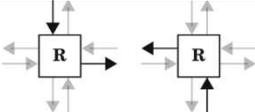
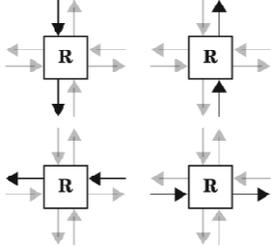
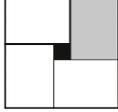
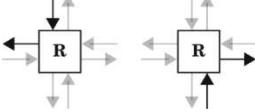
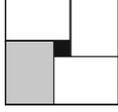
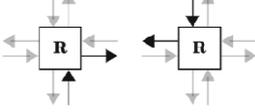
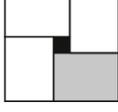
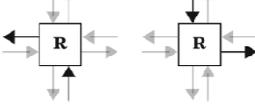
- 1) The faulty components are to each other *in a straight line*.
- 2) The faulty components are located in a particular cluster quadrant *mutually orthogonal*.

Figure 25 shows the two flavors of the fault pattern on the basis of two fault pattern examples: $f_1 \leftrightarrow f_2$ and $f_3 \leftrightarrow f_4$. In Figure 26, the resulting state of the suspicious matrix is illustrated. Again, this matrix does not contain all local maxima corresponding to the faults, which prevents a precise localization of the two faults f_2 and f_3 .

While the blind spot occurs in all patterns whose faulty components are "*in straight line*", the *orthogonally oriented* fault patterns are only problematic if the pattern has a specific orientation to the FDU. The orientation of this problematic pattern is additionally depending on a particular quadrant of a Cluster. Which orientation in each quadrant is problematic, is summarized in **Error! Reference source not found.** The table contains the four quadrants of one FDU-cluster (gray rectangles in the first column), each with the FDU (black square) in the center. For the orthogonal patterns the transmission directions of the heartbeat messages are distinguished additionally. This shows that only those links of the routers are affected that will shorten the path of a heartbeat message its destination. The position within a quadrant, however, is irrelevant. The effect always occurs when one of these *orthogonal* patterns is applied according to the network. Are the faults of the flavor "*in a straight line*" there is no distinction of the quadrants. This effect is independent with respect to the position and orientation of the cluster and preventing in any case the localization of the fault with the higher distance to the FDU.

Basically, this localization issue has the same criticality as the pattern "*in the immediate vicinity of the FDU*". Here, a fault creates a blind spot and prevents the successful search for any faults in that blind spot, too. As faults are masked again, this group is also classified as critical.

Table 7: Different problematic fault pattern regarding location and orientation

Quadrant	Orthogonal	In a straight line
		
		
		
		

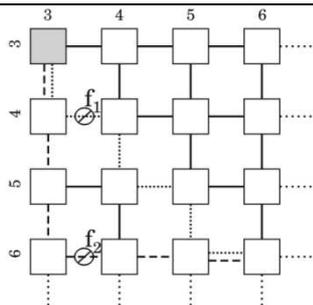


Figure 27: Phantom fault due to f_1 and f_2 .

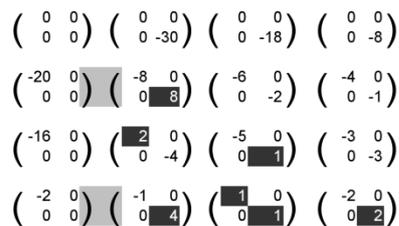


Figure 28: Corresponding matrix of suspicious network components.

“Faults on common partial paths” create illusions and cause that some network components are wrongly classified as faulty. That is what we call a “phantom fault”. It is the exact opposite of fault masking by the previous pattern. As described in the Deliverable D5.3, the normal operation of the localization technique rehabilitates falsely suspected network components by using different routing strategies for heartbeat messages. However, this fails with multiple faults if all of the following conditions are met:

- There are two heartbeat messages from the same processor core that use despite different routing strategies some common partial paths.
- In the network there are two faults that lie on one of the two actually disjoint paths of the heartbeat messages.
- One common partial path is located between the sending processor core and the faults.

If these properties are satisfied, the partial path with the higher distance to the FDU cannot be rehabilitated and creates a *phantom fault*. Figure 27 illustrates this in an example scenario. In the given network, two links at the point f_1 and f_2 are assumed to be faulty. The processor core at the router $R_{(6,6)}$ sent two heartbeat messages with different routing strategies.

The figure shows the resulting paths of the messages as dashed lines. In this example, the routing strategies XY (coarse dashed line) and Staircase routing provide (fine dashed lines) on the links $R_{(6,6)}^W \rightarrow R_{(5,6)}$ and $R_{(4,3)}^N \rightarrow R_{(3,3)}$ the two common partial paths of the corresponding heartbeat messages. In Figure 28 the suspicious matrix is illustrated after a complete monitoring round. After searching for the local maxima, additionally to the faults f_1 and f_2 , another fault on link $R_{(6,6)}^W \rightarrow R_{(5,6)}$ is determined (The value “2” in the bottom-right corner of the matrix is a local maximum). The latter fault is thus a *phantom fault*, which actually does not exist.

The group “common partial paths” has, however, a weak impact on the localization. Although the results shown in the example lead to a wrong classification of a network component, this has no impact on the actual performance of the network. The link $R_{(6,6)}^W \rightarrow R_{(5,6)}$ will still be used by the router to transmit messages of all types. Only the FDU internal representation of the network is in this case not 100% accurate and includes a pessimistic assumption about the state of the interconnection network. There is also the possibility that a migration of the FDU to another processor core changes the paths of the affected heartbeat messages, and thus corrects the misdiagnosis.

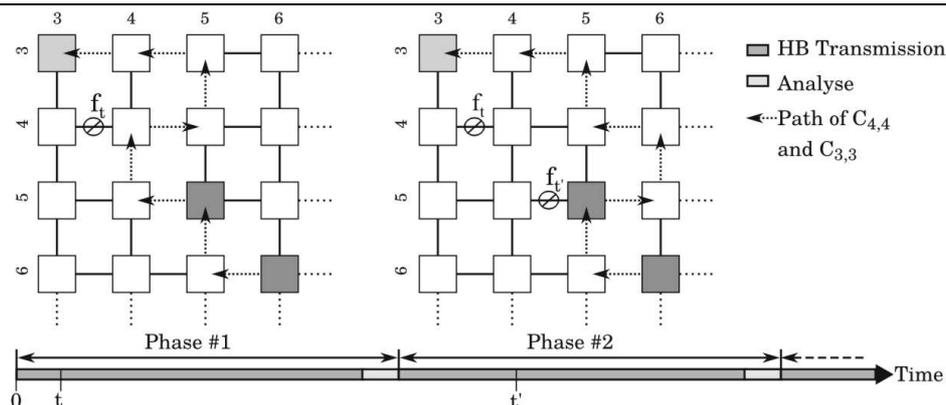


Figure 29: Example scenario for the implication of successive occurring multiple faults

3.2.2.2 Successive faults in the NoC

If the faults occur successively in the network, the effect of a blind spot or phantom fault is partly vanished. The effect of successive faults is illustrated below. The discussion on how this temporal pattern affects the spatial pattern is quantified in the next section.

The example scenario for two successive faults occurring in an FDU cluster is illustrated in Figure 29. From a timing perspective the scenario is divided into two phases. Each phase consist of a complete round of monitoring including the transfer of all of heartbeat messages, the analysis of the arrival times, and the adjustment of the TDMA scheme. In this example, the paths of the heartbeat messages result from the Staircase routing strategy. Additionally the protocol of the artificial delay is applied for blocked heartbeat messages and the final path is indicated by the dashed arrows.

In phase #1 the fault f_e is applied to the network. The dashed arrows show how the protocol of the artificial delay affects the arrival of the heartbeat messages of the processor cores $C_{(4,4)}$, $C_{(5,5)}$ and $C_{(6,6)}$ by a detour.

In the analysis of the first phase, the different arrival times of these messages are evaluated and the fault f_e is localized¹¹. Since the paths of the affected heartbeat messages will follow this route to the FDU in the long run, the TDMA scheme is adjusted accordingly to ensure the mutual isolation of the heartbeat messages¹². This resets also the matrix of suspicions components and phase # 2 can start over. In the following phase # 2, a further fault ($f_{e'}$) is applied to the network. Here again, the protocol of the artificial delay is triggered and detour the heartbeat messages. But this time at the Router $R_{(5,5)}$. Although the heartbeat messages are delayed, in this case, however, the FDU expects this delay by the formerly performed adaptation of the TDMA scheme. The resulting status matrix is

¹¹ As shown in Deliverable D5.3 single faults can be localized precisely.

¹² The mutual isolation requirement is also part of the Deliverable D5.3 and ensures that heartbeat message do not interfere with each other.

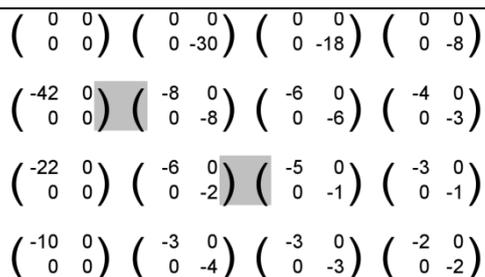


Figure 30: Status matrix after phase #2 for simultaneously occurred faults.

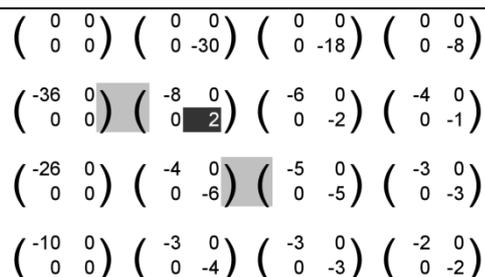


Figure 31: Status matrix after phase #2 for successively occurred faults.

shown in Figure 30. Since all values of the matrix are ≤ 0 , there is no evidence for the FDU that the messages were delayed by the fault $f_{e'}^r$.

Critical to the occurrence of a dead spot like this, however, is the order in which the faults occur. Consider a reversed order of occurrence for both faults from the example above. This reordering results in a status matrix at the end of phase #2, shown in Figure 31. The black shadowed positive value indicates the position of the fault $f_{e'}^r$ after the search for local maxima. The reason for the successful localization in this case is that with the help of the heartbeat messages of $C_{(5,5)}$ and $C_{(6,6)}$, the fault $f_{e'}^r$ could be initially localized in phase # 1. The subsequent occurrence of $f_{e'}^r$ in phase #2 now affects only on the delay of heartbeat messages from $C_{(4,4)}$. Since the heartbeat messages of $C_{(4,4)}$ arrived without a delay beforehand, the fault $f_{e'}^r$ can now also be located.

Although the paths of the heartbeat message given in the scenario above were formed by the staircase strategy, the masking effect is not limited to this routing algorithm. In fact, similar patterns can be generated with the same result for all four of the routing algorithms used here.

3.2.3 Quantification

At the conclusion of the study of multiple faults, it will be shown, how many of the fault patterns are considered to be problematic regarding the fault localization method. Thus, the performance of the localization for multiple faults is shown quantitatively. The basis of quantification consists on counting the respective problematic fault patterns. The results presented here correspond to the percentage of the three problematic spatial fault groups discussed above. In addition, three different cluster sizes (5×5 , 7×5 and 7×7) are applied in the quantification and summarized in Table 8.

One positive point is that the identified critical fault pattern “*in the immediate vicinity of the FDU*” occurs in about 1.5% of all fault patterns, in the worst case. Taking into account the order in which the faults occur, this value decreases further to 0.725%. Significantly higher is the percentage of the fault pattern “*closely spaced faults*” in the network. The compact shape of this spatial pattern, and the two different forms “*orthogonal*” and “*in a straight line*” result in 6.5% (simultaneous) and 1.6% (successive) of all patterns in a problematic pattern.

The spatial pattern “*faults with common partial paths*” stands out with two special properties. Firstly, since this pattern requires a minimum distance between the two faults when placing them, there is a case of small cluster sizes, without a chance to place the pattern. If the minimum distance is not

satisfied, the pattern will turn into "*closely spaced faults*". Secondly, the pattern is only problematic if the faults occur simultaneously in the network. For successive faults of the *phantom fault*, however, does not occur at all.

Summarizing all problematic fault patterns shows that about 2.4% of successively occurring fault patterns lead to fault masking or a *phantom fault*. If simultaneous occurrence is assumed, the proportion of problematic faults pattern increases to about 10%. Nevertheless with increasing cluster sizes a continuous reduction in the percentage of the critical fault pattern can be observed. In addition, it should be noted that simultaneous faults in the network are indeed possible, but it can be expected that the probability of multiple faults of successive nature in the network is much more likely. This assumption is supported by the consideration of the causes of faults. Apart from wear-out faults caused by physical effects, it can be assumed that the reduction of feature sizes yield more permanent faults, but these faults, however, are more distributed over the chip and in time.

Table 8: Quantification of Patterns

	Simultaneous			Successive		
	5x5	7x5	7x7	5x5	7x5	7x7
I.V. ¹³	1,449	0,713	0,355	0,725	0,357	0,177
C.P.P. ¹⁴	6,522	6,061	5,674	1,631	1,515	1,419
C.S.F. ¹⁵	-	-	3,901	-	-	-
Sum	7,971	6,774	9,929	2,356	1,872	1,596

¹³ I.V.: Faults in the **I**mmEDIATE **V**icinity of the FDU.

¹⁴ C.P.P.: Faults with **C**ommon **P**artial **P**aths.

¹⁵ C.S.F.: **C**losely **S**paced **F**aults.

4 OS-Level Fault Tolerance in TERAFLUX

An operating system prototype facilitating research on various parallel, distributed and reliable execution algorithms, supporting data flow principles on multi-core and many-core (1,000-10,000+) shared-memory devices. Specifically, the operating system supports distributed execution of an application over the device using dataflow threads, it was designed to handle core soft (transient) errors and can handle node hard-failures such that the application can transparently continue execution as the work that was pending on the failed node is recovered and performed by the remaining nodes. The operating system does not rely on strong consistency of the shared-memory but only a weak, acquire/release, consistency model is assumed on all communication mechanisms between nodes.

Please note that what follows has been all experimentally implemented and made available in the public COTSon repository (branches/tflux-test/tfos), see D7.5 section 11 for usage details.

4.1 Basic Architecture

While this future machine has all of the thousands of cores on a single die, it is unlikely that they can all be efficiently managed using the same hardware and software architecture like the ones used on contemporary processors. To name a few reasons: much of the hardware support that exists today does not scale well as core number increases (e.g. maintain cache coherency across all cores), and traditional operating systems that were designed for a few cores also do not perform well as the number of cores increases. Since the workload of managing this massively parallel chip is impossible for a single kernel, another level of abstraction is needed: a single-chip distributed operating system.

4.1.1 Clustered Architecture

The cores are divided into small groups, nodes, which share communication links and hardware elements, such as caches and interconnect to the rest of the chip. The nodes can all be symmetric or heterogeneous, so there can be several types of nodes with different hardware capabilities. Each node runs its own kernel/microkernel that is responsible for managing its cores, local memory and other resources, schedule tasks and collaborate with the other nodes. A possible architecture could be similar to those that operate on contemporary super-computers and include (Figure 32):

- Front-end nodes that are responsible for user interactions, they run a general-purpose operating system such as Linux. They delegate the computational and communication workload to the other nodes.
- I/O nodes responsible for communication with the network, storage or other hardware devices, they can run a specialized kernel for that purpose.
- Compute nodes that run only computational tasks in a power-efficient manner. These nodes can have a very basic kernel and have no access to the outside world, so all communication and system operations are performed by the nodes running a full kernel.

The volatile memory is also partitioned, either physically or logically, into private per-node memory regions and a globally addressable shared memory. It is assumed that a hardware based coherency is maintained between over each node's private memory, such as in current general-purpose processors, but only a weaker consistency model is supported over the shared memory, which is more likely to

exist on such a system. A plausible consistency model for shared memory is acquire/release, meaning a reader must explicitly *acquire* a region of the memory to see its latest version, and a writer must *release* a region it modified, before the changes are visible on subsequent acquires. This is in line with the initial proposal in this project (cf. D7.1).

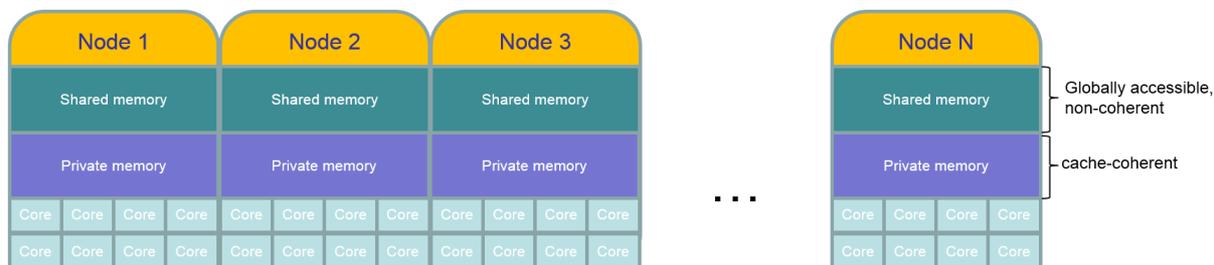


Figure 32: Logical system view.

4.2 Operating System Goals

4.2.1 Execution Model

The TERAFLUX operating system is aimed to support high task parallelism. To maximize parallelism, application execution is divided into many tiny threads; thread execution is governed by the dataflow model. DF-threads include the following properties:

- Scheduled to run only when all their inputs are ready.
- Have no side effects until completed in a non-faulty manner.
- When completed, the results are published and the dependents are notified.
- On error or core/node failure they can be safely restarted.

4.2.2 Fault Tolerance

Furthermore, the operating system must maintain reliability facing potential faults:

- Cores can permanently fail.
- Whole nodes can temporary or permanently fail.
- Cores can suffer from soft (transient) errors.

4.3 Runtime Environment

4.3.1 Memory Arrangement

Each node has a memory region assigned to it, which is only accessible from within the node. This space holds the local kernel structures and its runtime services, as well as the needed environment to support the execution of dataflow tasks (e.g. stacks, heaps, scheduler information). Each node also has a part of the shared memory under its control. This region can be read from and written to by all nodes. The shared region of each node holds the communication channels to the other nodes, and vital information on the threads running on that node, which is needed for recovery in the case of node failure (this is in-line with the TERAFLUX architectural template defined in 6.2, more on this later). Since this region is not consistent across all nodes, and explicit synchronization operations are needed to work with it (acquire/release), no atomic instructions such as compare-and-swap can be used by

several nodes on the same memory location. This means that conventional synchronization mechanisms modern operating systems rely on to synchronize between several cores, like spin-locks, cannot be placed in the shared memory as a means to synchronize between nodes. These limitations pushed the solution to use static allocations from shared memory wherever possible. To simplify and accelerate development, dynamic allocations are still used in a few cases. In those cases only the owner node of the shared region is allowed to allocate and de-allocate memory, thus avoiding the problems and overhead of multiple nodes synchronization.

4.3.2 Inter-node Communication

Since the shared memory is inconsistent across nodes, and requires explicit acquire and release calls to communicate through it, standard (library) data structures could not simply be placed in it to pass information between nodes. Custom structures were made that account for the inconsistency and acquire/release semantics.

4.3.2.1 Block Transfer Layer

In the beginning of each node's shared region, space is statically allocated for communication buffers of data *from* all the other nodes. There is one slice of memory dedicated for each node, each slice is a buffer for a FIFO “ring buffer” of small memory blocks of fixed size (Optimal size can be found from typical message size, currently 64 bytes). Each ring buffer is only written to by one “remote” node, and only read by the local node – it is a one-way communication channel between them. Since each node has one input buffer for all other nodes and one loopback buffer to itself, there are a total of n^2 buffers in the shared memory, which create a complete graph of bi-directional communication channels among the nodes.

With the assumption that each channel is only written to by one node and read by one node (SPSC), it was made lock-free over the inconsistent memory using appropriate acquire and release calls.

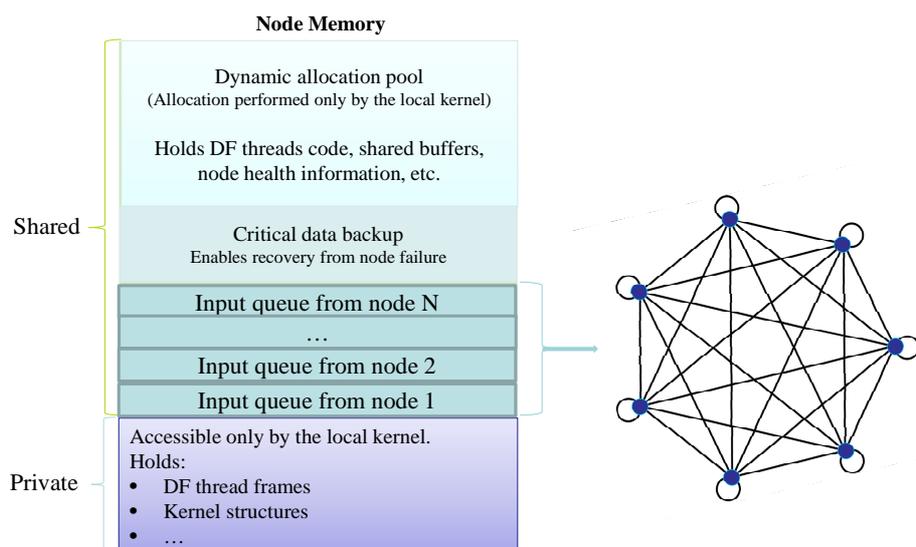


Figure 33: Memory Arrangement

4.3.2.2 Messaging Layer

Messages are packetized to fixed size frames and sent over the queue to the destination node. Each node polls on the input queues from the other nodes and processes the pending messages. Primary messages:

- ThreadLoad: Contains the actual thread binary to execute or its identifier, inputs information and shared memory requirements.
- ThreadWrite: Contains the destination address (Combination of the thread ID and input offset) and its value.
- Heartbeat: Notifying that the source node is alive and contains state information used for load balancing.

4.3.3 Node Failure Tolerance

The system was designed to be able to maintain correct operation of the operating system and the running applications in case of node failure.

4.3.3.1 Message Recovery

The buffers holding the messaging channels are kept in shared memory and can be accessed if the node crashed, for recovering the pending messages and handling them on a different node. Two main steps should be noticed that enable messages recovery:

- When a node processes a message it received, it does not remove it from the input queue until it performed the required operation.
- The processing of messages that are critical for correct operation (like thread-writes and scheduling requests) involves carefully updating the shared context backup region of the node with the new information.

The combination of these steps allows the recovery of all messages that were pending in the crashed node, and also of the message that was being processed when the node failed. Special care must be taken however when processing the first recovered message, as it might have already committed its full or partial results to the backup store before it was removed from the message queue. The backup region was structured with this in mind, so these cases are easily detected and resolved to avoid corruption and repeated operations.

4.3.3.2 Context Backup

A node maintains the collection of its pending and running thread descriptors in the shared memory, each descriptor contains the thread identifier, binary to execute, inputs state etc. - everything needed to describe the thread. This acts as the backup in case the node failed, and allows the recovery of all the threads that belonged to that node. A thread descriptor also has room for the input values the thread requires, so thread-write messages update the destination descriptor with new information until all inputs are received. This backup is simply a copy of the ThreadLoad messages the node received but not yet completed. As mentioned regarding messages recovery, while processing a ThreadLoad or ThreadWrite message we first carefully update the backup in shared memory, before removing the message from the queue, so critical information is never lost if the node suddenly stops at any point. The node chosen to recover the abandoned threads can simply process the ThreadLoad messages from the backup, or (only if all the inputs were received already) forward them to another node completely as-is. All of the messages still waiting in the queue that were not yet processed could be appropriately

processed by the recovery node. Note that if the recovery node chose to forward a thread elsewhere, this thread will already have all its inputs available and is ready for execution, thus it is not expected that there will be any pending messages regarding it that the recovery node could not handle.

These mechanisms ensure that the latest state of all threads can always be recovered from a crashed node, combining the information from the descriptors store and the pending messages queues, no matter at which point a node crashed.

4.3.3.3 Detection & Recovery from Node Failures

A simple distributed watchdog mechanism was made to detect offline nodes and initiate the recovery process. Each node sends heartbeat messages to all other nodes periodically. Concordantly, a service running in each node periodically checks that a heartbeat was recently received from all other nodes. When a crashed node is detected, a chosen node is assigned to take over the pending work of the dead node. The recovery node loads the thread descriptors that were kept in the backup segment of the node, and process the pending messages from the input queues. This node maintains control over the available resources of the disabled node, like memory and communication channels, so they are kept utilized by the system to minimize performance loss. Additionally to the life sign the heartbeat message represents, each node also appends its state to it (workload, temperature, fault rate etc.), this information is used by other nodes to have smarter scheduling policies and load balancing. Currently only two extra parameters are included with the message, which are the number of threads the node holds (considered in load balancing), and the time the message was sent (can be used to measure latencies). Other useful measurements can easily be added.

4.3.4 Thread Execution Procedure

The process can be described in the following high level stages:

- A scheduling request is created, specifying thread code (by name or actual binary to execute), the number inputs to wait for and the regions of global memory it needs to access.
- The request is submitted to the local scheduler, which decides what node should be assigned to execute the thread based on load-balancing and performance considerations. It can use the local node state, information collected from heartbeats and space and time locality of other threads with similar properties. Currently only a simple algorithm is implemented; it tries to launch the threads on the local node until it reached a specified threshold on the number of existing threads, above which it will pick another node, which owns the lowest number of threads (this workload information gained from heartbeats), that node might still be itself. Of course many other algorithms can be investigated. The scheduler assigns a system-wide identifier to the new thread.
- The scheduling request is sent to the assigned node (can be loop-back), as a message over the shared-memory communication channels.
- When the target node receives and starts to process the message it will first copy the thread information to its backup store in shared-memory, before removing the message from the queue. Thus the thread information is protected from being lost or corrupted in case of a failure during its processing.
- The node will create a local realization of the thread in its local memory, and add it to the list of pending threads. It will wait there until all its inputs were also received by ThreadWrite messages.

- When a ThreadWrite message is processed, the destination thread descriptor in the shared memory is first updated with the new information. This, again, is done before the message is removed from the queue, so the new data is not lost even if the node crashed after a thread received some of its inputs but not yet executed.
 - When the thread received all of its inputs it is ready for execution. At this point another scheduling decision can be made to forward the thread to be executed on another node, time passed since the creation of the thread could affect the decision on which is the best node to run on. The ownership is easily transferred when the thread has all of its inputs, because no further nodes will be concerned on where to send the ThreadWrite messages.
 - The shared memory regions the thread needs to read are refreshed in local memory (acquired) and the thread is executed. All of the operations it performs which have side-effects are not actually performed but are buffered in local memory until the thread completes. If Double Execution is enabled for the thread, it is launched twice (on different cores), with separate buffers. When both executions complete, their buffers are compared for equality, if they don't match the thread is launched again until the result buffers match. The buffer contains instructions of three types:
 - Shared-memory write operations. We normally don't want to commit speculative information to the global memory, before the thread was completed and passed the Double Execution test. Although depending on the application nature, these might not damage the application logic, because the writes will not trigger an operation until an appropriate ThreadWrite will notify of it.
 - Thread scheduling requests. Scheduling requests submitted need to return an identifier to the new thread, which can be used as the target of subsequent ThreadWrite operations or even their value (allowing the created threads to communicate among themselves). When this request is handled, a temporary thread identifier is immediately returned to the caller when the request is buffered, not yet the global identifier that could be used system-wide. Since we are still unsure that the running thread will complete successfully and will compare identical to the Double Execution thread, we don't commit the scheduling request to the rest of the system until the parent thread completed. Besides saving the complicated cleanup work in the case there was an error later, this also adds no delay in the running thread execution from invoking the scheduler and load-balancer to assign a global thread identifier. Another advantage is that when we wait until we have all scheduling requests from the thread, we can make more informed decisions on how or where we should run them, possibly combining the requests.
 - Writes to threads. These are needed to be buffered in two cases: The target thread that is written to and/or the value, are the result of a previous scheduling request, and are therefore only temporary identifiers at this point. The write cannot be performed before the temporary identifiers are only replaced by their final value when the thread is finalized and the scheduler has assigned a global identifier for them. Another reason for caching these when Double Execution is used is obvious; we are unsure of a single result's correctness and shouldn't publish it before we tested against the Double Execution thread.
 - When the thread execution completes (and passed the Double Execution test, if enabled), the thread is put in a list of completed threads to be finalized, its results from the local buffer need to be committed system-wide. In turn, the finalization process commits the results from the buffer as follows:
-

- Commit write requests to the shared memory (a *release* operation).
- Invoke the scheduler for each scheduling request the thread performed, it will choose a destination node to run the thread while considering performance optimizations and load-balancing, generate a global identifier for the new thread and send the request to the destination node (possibly local loop-back).
- Replace temporary thread identifiers submitted in the buffered ThreadWrite instructions either in the destination field or the value field with the final ones generated in the previous step and send them.

4.4 Implementation Details

4.4.1 Thread Identifier

A small dive to the structure of a thread identifier could help understand how these mechanisms could be implemented efficiently. A thread identifier is 64 bits in size so it can be transferred efficiently; it is bit-mapped to contain the following fields:

- The ID of the node that created the thread.
- The ID of the node that was originally destined to execute the thread. This field does not change even if the thread is later re-assigned to a different node (for load-balancing reasons or when it was recovered from a crashed node).
- Source-local unique identifier of the thread. It is unique only among threads that were created by the node that created the thread (enables global identifier generation without synchronization with the other nodes).
- Bits reserved for frame offset in ThreadWrite requests. Using these bits enables a ThreadWrite request to only contain two 64-bit values, making it efficiently buffered by the running thread and later transferred if needed. One of the values is the combined destination thread ID and offset within the thread's frame, and one is the actual value to write. One of these bits is a 'Translate Value' flag, specifying whether the value associated with the write to the target thread is a temporary thread identifier, returned to a running thread before its completion. When this flag is turned on the thread finalizing process knows it should translate the value to write from a temporary thread identifier to the final one that was assigned to this thread (and clear the flag). Note that the target of a write might also need translation; this is detected simply by the target node field containing a predefined invalid node ID.

4.4.2 Thread Binaries

The threads themselves are dataflow tasks, usually small pieces of code and its static data, can be just a single function. They are not whole programs but work only as a part of a larger application. To launch a distributed application over the entire system, not all of it needs to be loaded (and possibly copied) to all of the nodes. Only a single node is required to load the whole executable (possibly only the front-end node), and the rest of the nodes will only copy/load the code of the threads that they need to execute, saving a lot of bandwidth and space when the system contains hundreds of nodes. For example, the binaries copied to other nodes in the Fibonacci implementation are no bigger than a few hundred bytes.

Each thread is compiled as PIC (Position-independent code) so it can be copied and executed from the shared memory which can be mapped differently in each node. There are several ways threads binary code can be created: It can be compiled to be separate from the user executable, or compiled as a part

of it. If the thread binary is separate from the application executable it can be pre-compiled before the application is launched or at runtime by a JIT (Just-in-time) compiler. These combinations are supported simultaneously.

There are two ways a node can get the thread binary it needs to execute a ThreadLoad message it received, depending on the threads nature, probably one of the following is preferable:

- The ThreadLoad message contains the actual binary for execution. The executing node does not need to have access to disk or any global storage. This also means a node can execute threads from any application without preparation. This way is probably preferable if the threads are small or if many kinds of them are dynamically compiled at runtime.
- Only the binary name is included in the ThreadLoad message, a node receiving this information will look in its buffer of previously launched threads for the specified thread and try to load it. If the thread is not in the buffer the node assumes it was pre-compiled and tries to load it from disk. This option is probably better if threads are only pre-compiled or are relatively large in size.

Which mode to use is currently determined statically for each thread but it could be automatically determined at runtime based on the thread size or history.

4.4.3 Fail Tolerant Synchronization Count

The synchronization count field alone is not enough to reconstruct the latest state of a thread after a node failure. Take the simple case of a thread waiting for two input values, in the case that a node failed during the commit phase of a completed thread that writes to the waiting thread, but after the process already sent one value and the target SC was decreased by one. The almost-finished thread will be recovered and executed again, entering the commit phase and sending the first value again. At that point the SC of the waiting thread will be decreased to zero and the thread will execute even though it only received one of its inputs. This problem cannot be solved with the SC aggregation alone, so additional information tracking the state of individual inputs was added. This responsibility was assigned to the receiver, where the added logic is simple and ThreadWrite messages are kept small. A simple solution was to keep a bitmap of offsets written within the frame (in units of 64bit), so a write of 64bits to the k th position in the frame will clear the k th bit in the map, only after actually writing and publishing the new value. This way, the latest SC of a thread can always be recovered in any case of failure, using the 'Original SC' field and the bitmap. Repeated writes from rescheduled threads will not corrupt thread executions since the receiver will see that the destination offset bit is already cleared and will not decrease the SC it keeps in local memory. Note that currently the bitmap is 64 bits in size, limiting the size of a thread frame to a maximum of 512 bytes, of course a more flexible solution can be made.

4.4.4 Integration into TERAFLUX

Development was aimed to be able to reliably run the dataflow threads devised by TERAFLUX. The system uses the TSUF variant of TSU that implements a shared memory model of TERAFLUX, with a shared-memory consistency model similar to acquire/release. The operating system is simulated over virtual machines, using the COTSon simulator. The host runs a VM instance for each node (with several cores), each node boots into its own kernel. The host allocates a shared memory region, which the VMs can access over a simulator layer that implements the acquire/release semantics.

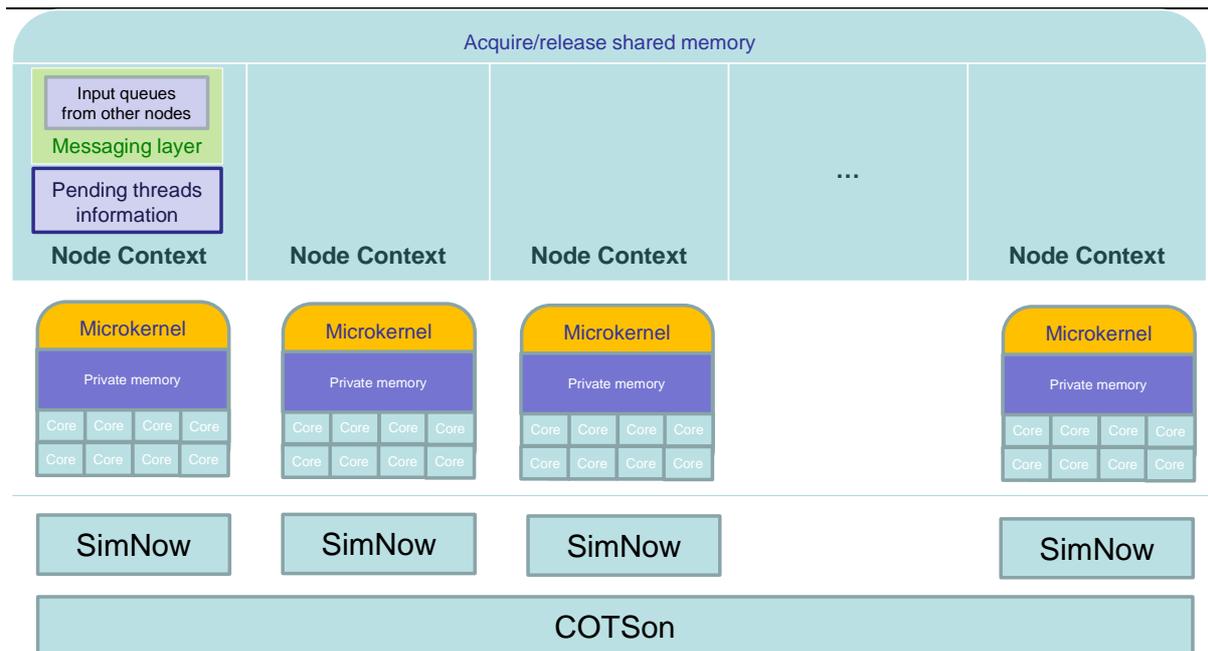


Figure 34: Simulation Overview

4.4.4.1 Shared-Memory Support

Each node runs the normal LTSU and DTSU but their responsibilities are limited to a subset of the original ones, just the hardware support that deals with shared memory access is relied on by the OS. They do not communicate to other nodes through any existing strong-consistency shared structures hidden in the simulator (e.g. by shared DF threads pools), besides the ones that simulate the shared memory semantics. All communication between nodes is done in software at the OS level, using the shared memory with the weaker consistency mode (OWM, cf. D3.5, D4.7) supplied by TSUF. Thread management and scheduling is also done in software only, not relying on TSU mechanisms (*tpoll*) for user threads, or the existence of the user executable loaded in all nodes (required user DF threads code is transferred over OWM). More specifically, all *tschedules* are immediately followed by a *tconstrain* to the local node, and no *twrite/tdecrease* instructions ever operate on threads from other nodes (i.e. *twrites* are mapped to standard MOV operations), thus the reachable space of TSU is constrained only to the local node.

4.4.4.2 Service Threads

The OS needs access to the shared memory (since the message queues, thread binaries and thread descriptors are stored there), so it uses a special kind of DF threads internally for some of its service threads, created with a similar interface as the one to create user DF threads.

These threads are not the normal dataflow threads that have no side-effects, have a short life-span and just need a single OWM acquire/release, some of them are persistent to the lifetime of the system, they have side-effects before their completion and they require a more powerful access to the OWM. They are responsible to maintain the communication channels through the shared memory, poll on message queues from other nodes and run message pumps. Support for such threads was not considered in TSUF so some modifications were needed, including the addition of a new T* instruction, *tacquire*, which receives a pointer to one of the readable OWM regions the thread

subscribed to and reloads it to guest memory. This instruction was not needed in ``normal" DF threads, because it is expected that their inputs are already present when the thread starts and do not change while it runs, contrary to an OS message pump thread for example. Such a thread needs to repeatedly acquire the queue regions for new messages, and a writer thread may need to write data to the queue and release several times.

Additionally, to simplify development and possibly increase performance, it was suggested to allow acquire and release operations only on a part of a region. The existence of this feature is dependent on the assumed hardware support, but for now at least it was simple to implement in the simulator. This feature was useful in many cases, for example when dealing with the shared FIFO buffers, the entire buffer is stored in a single region, but still different blocks can be written to and published independently, saving the overhead of publishing unmodified data and the bookkeeping of managing a region for each (relatively small) block.

Being aware of the dataflow instruction set, we can be more detailed with the contents of a thread descriptor that is passed in ThreadLoad messages and kept in the backup store. A new dataflow thread is created with additional details, as specified by the TSUF instructions, those need to be kept with the thread information when it is assigned to a node or restored from backup. Besides obviously specifying the thread routine (as explained before; we either include the thread name¹⁶ or the actual binary), also the frame size of the thread, the synchronization count and the OWM region definitions the thread need to access are included with the thread descriptor.

4.4.4.3 Dataflow Threads

The user dataflow threads can be one of the two forms:

- TSUF DF threads, created with *tschedule*, *tsubscribe* etc. OWM access from user threads is only supported in this mode, since normal pthreads cannot currently subscribe to it. This mode enforces a limit on the number of running DF threads, which is the number of worker threads that constantly call *tpoll* for work.
- Normal pthreads. Since all thread management and scheduling is done in software, in this case, there is no need for the user threads to be on TSUF worker threads. The OS service threads are still DF threads because they need OWM access but the user threads are not. To keep user code exactly the same when switching between modes, the runtime creates an environment similar to that created by TSUF. This mode enables the local operating system to manage the computation resources in the node and share them among running threads, making it much more flexible and efficient.

4.5 Related Research

Research on intra-node reliability mechanisms (specifically, for a case of core failure that does not limit the functionality of the remaining cores in the node) could most likely be incorporated and utilized by this system, thus completing the picture by providing both a local and a global solution. The system could handle core errors and failures more efficiently if some of the problems are handled within the node, preventing the node from failing and affecting the entire system by the global recovery procedure.

¹⁶ Thread name can obviously be replaced with a more efficient identifier.

References

- [1] A. Portero, A. Scionti, Z. Yu, P. Faraboschi, C. Concatto, L. Caro, A. Garbade, S. Weis, T. Ungerer and R. Giorgi, "Simulating the Future kilo-x86-64 core Processors and their Infrastructure," in *2012 Spring Simulation Multiconference (SpringSim'12)*, Orlando, FL, USA, 2012.
- [2] M. Solinas, R. Badia, F. Bodin., A. Cohen, P. Evripidou, P. Faraboschi., G.R. Gao, A. Garbade, S. Girbal, D. Goodman., B. Khan, S. Koliai, F. Li, M. Lujan, L. Morin, A. Mendelson, N. Navarro, A. Pop, P. Trancoso, T. Ungerer, M. Valero, S. Weis, I. Watson, S. Zuckerman and R. Giorgi, "The TERAFLUX Project: Exploiting the DataFlow Paradigm in Next Generation Teradevices," in *2013 Euromicro Conference on Digital System Design (DSD)*, 2013.
- [3] S. Weis, G. A., J. Wolf, B. Fechner, A. Mendelson, R. Giorgi and T. Ungerer, "A Fault Detection and Recovery Architecture for a Teradevice Dataflow System," in *Data-Flow Execution Models for Extreme Scale Computing (DFM)*, Galveston Island, Texas, USA, 2011.
- [4] S. Weis, A. Garbade, S. Schlingmann and 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.
- [5] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," *International Symposium on Fault-Tolerant Computing*, pp. 84-91, 1999.
- [6] C. LaFrieda, E. Ipek, J. Martinez and R. Manohar, "Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN '07*, 2007.
- [7] M. Rashid and M. Huang, "Supporting highly-decoupled thread-level redundancy for parallel programs," in *IEEE 14th International Symposium on High Performance Computer Architecture, 2008. HPCA 2008.*, 2008.
- [8] 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.
- [9] J. Duato, S. Yalamanchili and N. Lionel, *Interconnection Networks: An Engineering Approach*, Morgan Kaufmann Publishers Inc., 2002.
- [10] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann Publishers Inc., 2003.
- [11] J. Kim and A. Chien, "An evaluation of planar-adaptive routing (PAR)," in *Proceedings of the*

Fourth IEEE Symposium on Parallel and Distributed Processing, 1992, 1992.

- [12] A. Garbade, S. Weis, S. Schlingmann, B. Fechner and T. Ungerer, "Fault Localization in NoCs Exploiting Periodic Heartbeat Messages in a Many-Core Environment," in *2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, Boston, Massachusetts, USA, 2013.
- [13] B. Schroeder, E. Pinheiro and W. Weber, "DRAM errors in the wild: a large-scale field study," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, New York, NY, USA, 2009.
- [14] B. Schroeder and G. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," *IEEE Transactions on Dependable and Secure Computing*, pp. 337-350, 2010.
- [15] E. B. Nightingale, J. R. Douceur and V. Orgovan, "Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs," in *Proceedings of the sixth conference on Computer systems (EuroSys '11)*, New York, NY, USA, 2011.
- [16] R. Giorgi, "TERAFLUX: Exploiting Dataflow Parallelism in Teradevices", ACM Computing Frontiers, pp.303-304, Cagliari, Italy, May 2012.