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

PROJECT NUMBER: 249013

Exploiting dataflow parallelism in Teradevice Computing

D6.2 Advanced TERAFLUX Architecture

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

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

Lead contractor for the deliverable: UCY

Revision: See file name in document footer.

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

Change Control

Version #	Date	Author	Organization	Change History
1.0	11/03/11	Samer, Arandi Costas, Kyriacou George, Michael George, Mathaios Natalie, Masrujeh Pedro, Trancoso Paraskevas, Evripidou	UCY	
			UAU	
			UNIMAN	
			BSC	

Release Approval

Name	Role	Date
Samer Arandi	Originator	03.11.2011
Paraskevas, Evripidou	WP Leader	03.11.2011
Roberto, Giorgi	Project Coordinator for formal deliverable	30.12.2011

TABLE OF CONTENTS

List of Figures

List of Tables

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

Samer Arandi, Costas Kyriacou, George Michael, George Matheou, Natalie Masrujeh, Pedro Trancoso, Paraskevas Evripidou
UCY

Roberto Giorgi, Zhibin Yu, Sylvain Collange, Alberto Scionti
UNISI

Behram Khan, Salman Khan, Mikel Lujan and Ian Watson
UNIMAN

Yoav Etsion
BSC

Theo Ungerer, Bernhard Fechner, Arne Garbade, Sebastian Weis
UAU

Glossary

API	Application Program Interface
BSC	Barcelona Supercomputing Center
CFC	Control Flow Check, it identifies pure hardware, pure software or a hybrid approach to detect misbehavior in the execution of a thread by checking the execution results of each basic block and comparing them with the expected ones
CID	Core ID
COTSon	Software framework provided under the MIT license by HP-Labs
DDM	Data-Driven Multithreading
D-FDU	Fault Detection Unit that operates at the Node level
DMA	Direct Memory Access
DTA	Decoupled Threaded Architecture
D-TMU	Distributed Transactional Memory Unit
DTS	Distributed Thread Scheduler
D-TSU	Thread Scheduling Unit that operates at the Node level
DF-Thread	A Data-Flow Thread
DF-Frame	The frame memory associated to a Data-Flow thread
ECC	Error Correction Code
FDU	Fault Detection Unit
FFT	Frame Free Table
FP	Frame Pointer
IP	Instruction Pointer
ISA	Instruction Set Architecture
L-FDU	Local to a single core Fault Detection Unit
L-TMU	Local Transactional Memory Unit
L-TSU	Local to a single core Thread Scheduling Unit
Leading Thread	In the double execution approach, it represents the main executed thread
MAPE	It is the acronym of Monitoring, Analyzing, Planning and Executing. It is used to identify the four main action of a monitor autonomous computing system, used to detect faulty behaviors
MCA	Machine Check Architecture, it identifies hardware structures that support the detection of faulty behaviors within a core
NoC	Network-on-Chip
Node	Group of cores and additional hardware units, such as: L-TSU, D-TSU, L-FDU, D-FDU, memory controllers, network interfaces

OWM	Owner Writeable Memory
OWMP	Owner Writeable Memory Pointer
PLQ	Pre-Load Queue
PTQ	Pending TSCHEDULE Queue
StarSs	Star Superscalar
TFlux	Thread Flux
TM	Transactional Memory
TSU	Thread Scheduling Unit
UCY	University of Cyprus
UNIMAN	University of Manchester
UNISI	University of Siena

Executive Summary

In this deliverable we report the work performed within the context of WP6 for the second year of the TERAFLUX project. For this period, the goal was to complete Task 6.3 (M13-24) Advanced Architecture Definition. We present our progress in terms of:

- UNISI led the effort for proposing an advanced architectural template;
- UNISI worked toward the definition of a x86-64 ISA extension (namely T*), in order to support the Data-Flow execution model, through a set of instructions for scheduling Data-Flow threads (DF-Threads, defined in D6.1);
- UNISI worked toward a proposal of the architecture of hardware modules to support both the Data-Flow execution model and the Transactional Memory model for the DTA-style DF-threads. In particular, UNISI defined the architecture of the Distributed Thread Scheduler (DTS), which may support the DF-thread execution. For the proposed DTS architecture, UNISI provides an initial estimation of the area utilization;
- UNISI, UAU, MSFT, HP defined a mechanism through which recover from a fault, by re-executing a DTA-style Data-Flow thread whenever a fault is detected on a core.
- UCY performed the design and specification of the hardware support for dynamically scheduling DDM-style DF-threads
- BSC performed the design and specification of the hardware support for the Hierarchical Task Scheduling of TERAFLUX threads
- UNIMAN studied and proposed the architecture support for Transactions
- UAU studied and proposed the architecture support for Fault-Tolerance
- UCY designed the hardware support for Transactions and Fault-Tolerance for the execution of DDM-style DF-threads
- UCY explored the use of simpler cores (e.g. Intel Atom cores)

Our achievements show that our goals for this period have been met.

1. Introduction (UCY, UNISI)

The realization of future tera-device systems is bringing on the table many challenges, especially from the point of view of the programmability and reliability (see HiPEAC roadmap [1]). In order to properly analyze the implications of these challenges, the TERAFLUX Consortium decided to target at least a 1000-core platform. First of all a sufficiently rapid scheduling of threads that retain Data-Flow properties is beneficial for many reasons (reliability, speculation, reduction of unnecessary communication). However, the efficient implementation of such execution model requires a hardware unit that is in charge of scheduling the execution of threads across the cores (Section 2.3.1).

Research by UNISI [2] showed that for a program like Clustal-W [3] (i.e., an important application program used in molecular biology for the simultaneous alignment of nucleotide or amino acid sequences), the large majority of the application may be coded with Data-Flow threads (DF-Threads). The high frequency of operations required by Data-Flow threads therefore imposes the implementation of a direct support in the ISA (Section 2.2).

In a similar way, research by UCY showed that more than 15 different applications (kernels that represent common scientific operations, MiBench and NAS applications) [4, 5] can be efficiently executed using the DDM-style DF-Threads execution model.

During this year we have been working on improving the TERAFLUX architecture and execution model that had been developed during the first year in Tasks 6.1 and 6.2.

We have converged with the rest of the partners on a common architecture template that contains the different modules of the TERAFLUX architecture. All partners on all WPs will use this template as the basis and will extend it according to their research needs. The template is presented in Section 2.1.

We have also iterated on the development of the ISA extensions to support DF-threads. This is the architecture interface towards the compilation tools of WP4 (Section 2.2).

Furthermore, we have been developing different hardware modules to support:

- Scheduling of TERAFLUX threads;
- Transactions;
- Fault-Tolerance.

The former regards modules for the scheduling of coarse-grain threads and fine-grain DF-threads. The two latter modules are the result of the synergy between WP6 and WP3/WP5, respectively. The scheduling module has been designed and synthetized as to determine its

detailed specifications such as space requirement, latency and power consumption. Regarding the modules for Transaction and Fault-Tolerance support, several designs are being studied and evaluated at the moment. The specification for the modules will be used in their implementation and integration into the COTSon simulation platform in WP7. It is relevant to notice that even though these modules are being tested for small setups, they have been designed with scalability in mind as to achieve efficient execution for the 1000 cores TERAFLUX setup.
Finally, we are investigating the use of smaller and simpler cores (e.g. Intel Atom cores) as to increase the degree of parallelism available in the TERAFLUX chip.

1.1 Document structure

The rest of this document is organized as follows. In Section 2 we present the Advanced TERAFLUX Architecture. In particular we present the ISA extensions, the hardware modules for scheduling, transactions and fault-tolerance, and the hardware synthesis for the required estimations. In Section 3 we present the conclusions that include a summary of the work performed this year and a brief overlook of the tasks to be performed in year three.
Besides this organization of the deliverable we provide here a “quick-reference” to locate specific topics that were part of the WP6 objectives.

WP6 Objective from Annex-1	**Where it is explained**
The definition of an execution model supporting Data-Flow threads, integrating different approaches such as DTA, DDM and HTS.	Sections 2.3.1, 2.3.2
The definition of the ISA extensions in order to support the Data-Flow execution model.	Section 2.2
The integration of the Transactional Memory model.	Section 2.3.3
The definition of the main hardware components that support both the execution and the transactional memory models (i.e., thread scheduling operations, runtime support, transactions and fault recovery).	Sections 2.3, 2.4
The definition of a mechanism for recovering from a fault	Section 2.3.4

1.2 Relation to other deliverables

The work and material presented in this deliverable is an evolution of the basic architecture and execution model presented in D6.1. Furthermore, part of this work is based on the material presented in D3.3, D5.1 and D5.2.

As this is an integration work package, we constantly refer to activities carried out in other work packages. In particular:

- WP5: Design Exploration of FDUs and Core-Internal Fault-Detection (D5.1, D5.2)
- WP6: Basic TERAFLUX Architecture and Basic Execution Model (D6.1)
- WP7: Plan for Interface Deployment, Definition of ISA extensions custom devices and External COTSon API extensions (D7.1, D7.2, D7.3)

1.3 Previous Activities referred by this deliverable

In the Year 1, the following activities had been performed:

Task 6.1 (m1 - m12) – Basic execution model: several execution models for Data-Flow threads have been proposed, mainly DDM (UCY), DTA (UNISI) and StarSs (BSC). These different approaches will complement each other; furthermore they will be integrated with a Transactional Memory model. Both Instruction Set Extensions and a hardware model for a distributed scheduler will be proposed.

Task 6.2 (m1 - m12) – Basic architecture definition: the basic architecture that supports the Data-Flow threads execution model and the transactional memory model is defined. The architecture is based on the initial integration of 1000 complex cores (e.g., Intel Xeon cores), considering a homogeneous system. However, it is expected to improve the number of cores up to 10000 using a heterogeneous system, where simpler cores (e.g., Intel Atom cores) are coupled with complex ones.

1.4 Activities referred by this deliverable

In the Year 2, the following activities had been performed:

Task 6.3 (m12 - m24) – Advanced architecture definition: with respect to the basic architecture, adding hardware units to support scheduling activity and execution model, will result in an improved architecture and performance benefits.

2. Advanced TERAFLUX Architecture

2.1 Advanced Architecture Template (UNISI, UCY, UNIMAN, UAU, HP)

The realization of future tera-device systems is bringing on the table several challenges that the scientific community is dealing with [1]. Among the others the programmability and reliability issues impose big limitations in the usage of future tera-devices. In the TERAFLUX project, we decided to target both the programmability and reliability issues, by considering at least a 1000-core machine. One of the goals of the project is the definition of the architecture of such machine and we explored the less common path of a Data-Flow execution model.

Nevertheless, the complexity of such system has to be properly managed, and therefore the novel advanced architecture resembles very closely existing architecture and uses many of the existing architectural blocks. More noticeably, beside the large amount of cores, a set of additional hardware units needs to be defined to correctly support the scheduling and the execution of Data-Flow threads on the target machine: this can be synthetically referred to as "Resource Management Hardware". Moreover, this architecture has some precise properties that are detailed below and in the Appendix-1.

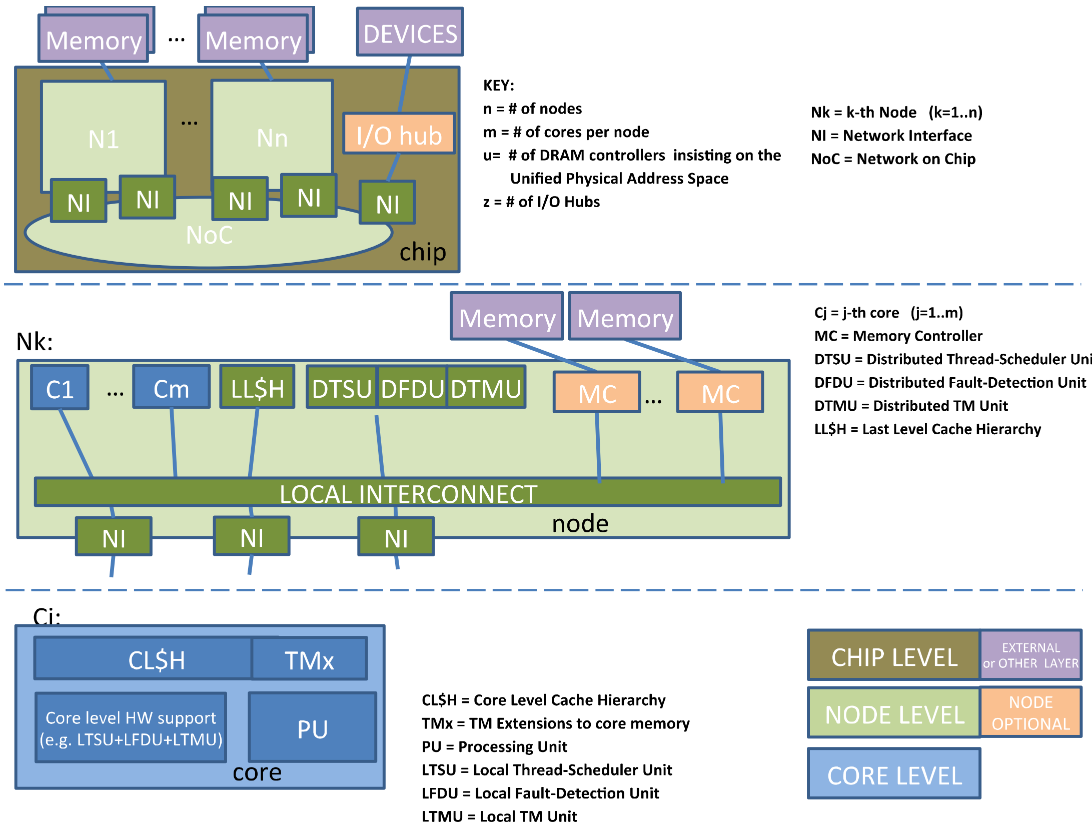

Figure 1: Advanced Architectural Template of a TERAFLUX system

Figure 1 shows a detailed view of what we call "Advanced Architectural Template" of a TERAFLUX system, highlighting the many off-the-shelf architectural blocks as well as the additional hardware units such as the thread scheduler and the fault detection units.

As shown in Figure 1, the system is organized hierarchically in two levels (in the future these levels may increase): at chip level, there are a certain number of Nodes (i.e., a group of cores and additional hardware units including a portion of the memory hierarchy). Each Node includes a set of cores, some additional hardware units and the top part of the memory hierarchy. In principle, the size and the topology of each Node can be defined both statically and dynamically. In the second case, the number of cores and their organization within the Nodes is defined according to the main characteristics and requirements of the applications. However, we established that each Node should be able to access a service core, not necessarily inside the local Node (see Appendix 1 – L1.0.x).

Each core is composed of a Processing Unit (PU) and a Core-Local Cache memory Hierarchy (CL$H). The processing unit can be either an off-the-shelf core (i.e., x86-64 core) or a dedicated one (in this case it may be possible to meet specific needs of the different types of threads that are running on the system). From this point of view, the architecture distinguishes between Service Cores (also called Larger Cores) and Auxiliary Cores. Service Cores are based on powerful cores designed for OS, I/O or ILP intensive codes (e.g., multi-threaded, multiple issue, out-of-order execution, etc.). These cores are intended to support the execution of S-threads and L-threads as reported in the deliverable D7.1. On the other hand, Auxiliary Cores are designed to be single-issue, power efficient computational cores. It is worth observing that both the two types of cores may have the same x86-64 ISA, while their features and timing models are different (this kind of architecture is also referred to as asymmetric). The x86-64 ISA is extended to enable the usage and exploitation of thread level parallelism (TLP), transactional memory (TM) and a specific memory model (see deliverable D7.1, see Appendix 1 – L0.0.x).

At the core level the following additional hardware units are defined (see Appendix 1 – L0.3.x):

- *Local Thread Scheduling Unit* (L-TSU): is in charge of scheduling Data-Flow threads on the corresponding core, and communicating with other L-TSUs and the Node's D-TSU;
- *Local Fault Detection Unit* (L-FDU): is in charge of detecting faults at the core level, and sending heartbeats to the Node's D-FDU.
- *Local Transaction Memory Unit* (L-TMU): has the responsibility of managing the versioning data and coordinate with its node-level D-TMU.

At the Node level, we defined other three hardware units:

- *Distributed Thread Scheduling Unit* (D-TSU): is mainly in charge for scheduling Data-Flow threads at the node level and communicating with other D-TSUs. Moreover, it holds the information on the association between running threads and cores in the Node.
- *Distributed Fault Detection Unit* (D-FDU): detects core, memory controller, and other D-FDU faults and provides recovery management features at the Node level. It closely communicates with its dedicated D-TSU.
- *Distributed Transactional Memory Unit* (D-TMU): has the responsibility of doing Transactional Memory conflict detection by coordinating with other D-TMUs

The definition of a Distributed Thread Scheduling Unit (D-TSU) and a Distributed Fault Detection Unit (D-FDU) allow us to avoid the single-point-of-failure, which is typical for monolithic designs. Moreover, it gives us possibility to scale out the architecture with the number of cores. Our thread scheduler uses information related to the temperature, power consumption, faultiness level and availability of each core to correctly schedule Data-Flow threads on-the-fly without necessarily involving software intervention and to the end of making the architecture more resilient to faults. In order to support this task, we considered proper timing models for the additional hardware scheduler and fault detection units (L-TSU, D-TSU, L-FDU and D-FDU, see Appendix 1 – L0.3.x, Appendix 1 – L2.2 and Appendix 1 – L0.P0). As discussed in Section 2.3.4, we want to exploit the Data-Flow execution model also for recovering after the detection of a fault. In this document we show how the scheduler has been designed to support re-execution of Data-Flow threads for the DTA-style [6] and DDM-style DF-threads, without any side effects on the overall application behavior (see Appendix 1 – L2.3).
The proposed architecture is designed to provide an efficient support for the selected execution model (see Appendix 1 – L2.x). In this sense, the architecture is intended to offer some beneficial properties:

- We do not assume any hardware global coherency mechanism, since all the resources are globally addressable (Unified Address Space – UAS, see Appendix 1 – L0.2.x);
- For the sake of simplicity of an initial implementation, we postpone a careful support of page faults for a later step; initially, we assume that code and data are loaded in the memory of the system (i.e., cache hierarchy and main external memory, see Appendix 1 – L2.5);
- We assume sequential consistency for the memory operations performed by the single thread (i.e., memory operations are sequentially consistent, see Appendix 1 – L2.7);

- All the threads running on the Auxiliary Cores can start I/O calls that are actually served by a Service Core. For the I/O operations management, we need to keep in mind that they are not memory consistency aware (see Appendix 1 – L2.6);
- The architecture will provide an explicit mechanism to "publish" all the executed changes and to make them visible by using the following protocol: (i) a signal is used to make visible the changes, while (ii) a second signal is used to inform that the "publishing" operation is finished (e.g., (i) a transaction attempts a commit, and (ii) the architecture hardware support makes it available to the system). See Appendix 1 – L0.2.7;
- The architecture may provide hardware support for protection (see Appendix 1 – L2.8);
- Finally, the architecture will expose an efficient mechanism for implementing virtualization (i.e., an efficient mechanism to map Virtual Cores into Physical Cores). See Appendix 1 – L2.4.

Communication among cores and additional hardware units within a Node is performed through a dedicated (local) interconnection system (e.g., a dedicated interconnection bus, a crossbar switch or a network-on-chip), while the communication among different Nodes is accomplished by the implementation of a classical network-on-chip (NoC) [7], along with the proper network interfaces (NIs). Since the design exploration of the NoCs is not a primary objective for the project, we considered state-of-the-art models and designs. However, we took into account timing models of the communication latencies for both the local to the Node interconnection networks and the inter-Nodes network. These timing models allow us the correct implementation of a simulation model for the interconnection system (see Appendix 1 – L0.1.x).

The access to external memory is served at the Node level by the Memory Controllers (MCs). As previously mentioned, the architecture provides a memory hierarchy that is split into two parts:

- A cache hierarchy defined for each core (i.e., each core provides a cache hierarchy composed of levels from L1 to Lk), called Core-Level Cache Hierarchy (or CL$H);
- A cache hierarchy defined for each Node core (i.e., each core provides a cache hierarchy composed of levels from Lk+1 to Ln), called Last-Level Cache Hierarchy (or LL$H).

These two hierarchical parts allow us the implementation of a globally addressable physical space that guarantees on-chip global accessibility when the system is in supervised mode (while it is not directly accessible when the system is in user mode), possibly with variable latencies. The memory hierarchies are also intended to support the chosen memory model (see Section 2.1). In particular, we want to allow management of thread local storage memory area (TLS), Data-Flow thread synchronization via Single Assignment Semantics (SASs) like OWM and DF-Frames and Transactions. As for the case of interconnection system, the definition of the memory hierarchy is not a primary objective for the TERAFLUX project, thus we considered state-of-the-art memory models and designs. However, we are considering both prefetching and DMA mechanisms, as well as specific time models for the access to all levels of the hierarchy. These timing models allow us the correct implementation of a simulation model for the memory system (see Appendix 1 – L0.2.x).

The subsequent sections will detail the definition of these hardware modules. Finally some preliminary estimation related to the hardware synthesis in terms of area, power consumption and latency is given.

2.2 ISA Extension (UNISI)

In order to support the execution of Data-Flow threads in the target system, we proposed an extension of the x86-64 Instruction Set Architecture (ISA), that we called *T**. The extension is designed upon the adopted memory model.

2.2.1 ISA Extension for DTA-Style (UNISI)

With respect to the DTA extension presented in the Deliverable D6.1, we introduced some slightly but important modification of the T* ISA extension (so we prefer a new name to avoid any confusion), as reported in Table 1. In particular:

- We changed the TCREATE into TSCHEDULE to indicate that there is NO synchronous code execution upon the definition of a new thread. The adoption of a new name allows us avoiding confusion, being more consistent with the instruction semantic. The new name has been also used in a recent publication [2]. For current testing purposes, we retain the use of the C flag (CF) – the value of CF is therefore undetermined after the TSCHEDULE.
- We removed the restriction for which it was assumed that data operand should be loaded into the fixed register RAX.

As for the previous ISA extension definition (see deliverables D6.1 and D7.2), here we assumed the size of the operands to be by default 1 machine word (e.g. 64 bits for x86-64 platforms).

As mentioned above, the implemented T* extension has been designed keeping in mind the selected memory model (FM, TLS, OWM, TM). The extension is composed of 6 instructions used to manage the creation and deletion of a DF-thread, reading and writing operations, the allocation and the release of a memory block for the thread. Table 1 depicts the six instructions. For each of them a detailed description is given, along with specific notes when required.

	T* INSTRUCTIONS	**IMPLIED COMPILER TARGET**
Synopsis	**TSCHEDULE *RS1, RS2, RD***	**TSCHEDULE(<*IP*>, <*SC*>, &<*frame pointer*>)**
Description	This instruction allocates the resources (a DF-frame of size ***RS2*** words and a corresponding entry in the Distributed Thread Scheduler – or DTS) for a new DF-thread and returns its Frame Pointer (FP) in ***RD***. ***RS1*** specifies the Instruction Pointer (IP) of the first instruction of the code of this DF-thread and ***RS2*** specifies the Synchronization Count (SC).	
Notes	The allocated DF-thread is not executed until its SC reaches 0. The TSCHEDULE can be conditional or non-conditional based on the value stored in the zero flag. If the zero flag is set to 1 then the TSCHEDULE will take effect, otherwise it is ignored.	
Synopsis	**TDESTROY**	**TDESTROY**
Description	The thread that invokes TDESTROY finishes and its DF-frame is freed, (the corresponding entry in the Distributed Thread Scheduler is also freed).	
Notes	-	
Synopsis	**TWRITE *RS, RD, offset***	***(<frame pointer> + <offset>) = (<source register>)**
Description	The data in ***RS*** is stored into the DF-frame pointed to by ***RD*** at the specified offset.	
Notes	***Side Effect***: The Distributed Thread Scheduler decrements the SC of the corresponding DF-thread entry (located through the FP): $SC_{FP} = SC_{FP}-1$	
Synopsis	**TREAD *offset*, *RD***	**(<destination register>) = *(<self frame pointer> + <offset>)**
Description	Loads the data indexed by 'offset' from the self (current thread) DF-frame into ***RD.***	
Notes	***Assumption***: the DTS has to load into the register implicitly used by TREAD the value <self_frame_pointer>. In a x86-64 implementation, we can reserve RAX for this purpose.	
Synopsis	**TALLOC RS1, RS2, RD**	**<pointer> = TALLOC (<size>, <type>)**
Description	Allocates a block of memory of ***RS1*** words. The pointer to it is stored in ***RD***. ***RS2*** specifies the special purpose memory type.	
Notes	The Distributed Thread Scheduler tracks the memory allocated. An implementation can code <type> in the 2 LSBs of <size>	
Synopsis	**TFREE *RS***	**TFREE(<pointer>)**
Description	Frees memory pointed to by ***RS***.	
Notes	The Distributed Thread Scheduler tracks the memory deallocated.	

Table 1: T* Instruction Set Extension for the x86-64 ISA

Moreover the DTS "continuation" (or status-holding data structure) associated to each DF-thread has been extended as shown in Figure 2.

A Distributed Thread Scheduler continuation (i.e., a data structure used to store information about Data-Flow threads that have to be scheduled for execution), or simply DTS continuation, stores a set of pointers and counters:

- *IP*: (Instruction Pointer) is the pointer to the code memory;
- *FP*: (Frame Pointer) is the pointer to the assigned frame memory, the inputs of a consumer threads are written in this memory region by one or more producer threads. Writers-Readers are N:1.
- *SC*: (Synchronization Count) holds the number of inputs (in word-multiple) needed by the thread to become ready for the execution. Each time a write is atomically performed on the frame memory the synchronization count is decremented. The thread becomes ready for the execution when its synchronization counter equals to zero;
- *CID*: (Core ID) holds the unique identifier for the core;
- *TLSP*: (Thread Local Storage Pointer): is the pointer to the local memory area of the thread, thus it is part of its address space. Since this memory region is private to the owner thread, no x86-64 consistency issues can arise. Writers-Readers are 1:1.
- *OWMP*: (Owner Writable Memory Pointer): is the pointer to the memory region associated to the thread which is typically written and subsequently ready by multiple threads. The OWM can be written by only one thread at a time, which becomes the write-owner of the memory region (cf. D3.1, D7.1). Writers-Readers are 1:N.
- *TMP*: (Transactional Memory Pointer): is the pointer to the memory region that is managed through the transactional model. The memory region is globally accessible, thus all the threads can write and read concurrently. Writers-Readers are N:N.

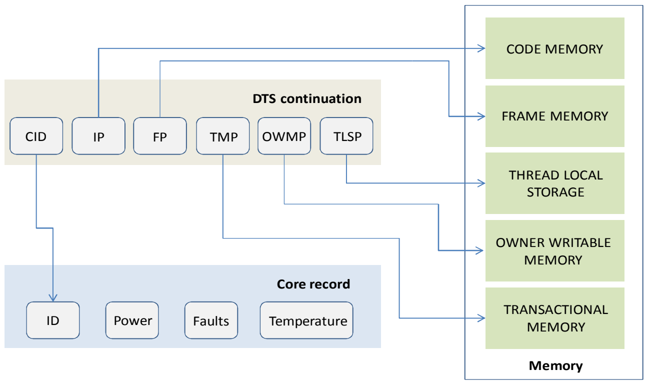

Figure 2: The DTS-continuation that is allocated by the T* instruction TSCHEDULE.

The three pointers TMP, OWMP and TLSP are used by the thread to access to the corresponding memory regions. To keep as simple as possible the data structure, the three memory regions can be modified in terms of size, allowing the system to satisfy subsequent requests from the executed threads. The two memory regions TM and OWM allow the data exchange among different threads and the execution of transactions in a simple way. In fact, the semantic associated to the access to each of these regions reduces the need of complex synchronization mechanisms.

We have also considered the situation where a given thread may access simultaneously more memory types of the TERAFLUX memory model (FM, TLS, OWM, TM), therefore the support for the set of the four corresponding pointers has been taken into account. Moreover, we considered also the situation when we can store any mix of pointers to OWM, TLS, TM in the frame memory, but this will be a future work.

A core record data structure is associated to each core in the Node. This data structure is available to the DTS in an efficient way. It stores information about the identifier number of the core and about the power consumption, the temperature and the number of detected faults. These three fields are dynamically updated to reflect the current state of the core. The state of the core is used by the Distributed Thread Scheduler (DTS, see also Section 2.2.2) to perform more accurate scheduling decisions. If the power consumption of the core is above a threshold the scheduler might decide to schedule ready threads on a different core to avoid an overloading situation. Similarly, if the temperature reached a threshold the scheduler might decide to not consider the core for the execution of ready threads. This policy can be applied by the DTS to guarantee higher level of reliability of the target system. In fact, using a core that exhibits high temperature for a long period can induce high levels of stress to the hardware circuit, reducing its lifetime and the overall reliability of the system (we acknowledge the IAB member Giuseppe Desoli for pointing out this important observation). Finally a similar policy can be adopted by considering the level of faultiness of the core.

2.2.2 ISA Extension for DDM-Style (UCY)

The ISA extension for DDM-style, presented in D6.1, was designed to be able to be implemented with a few regular instructions and TSU support. No further development was needed in year 2. This issue will be revisited in year 3. Notice that the execution of applications using the DDM-Style DF-threads does not require the use of special instructions as it was shown in [4, 5].

2.3 Hardware Modules

2.3.1 Thread Scheduling

2.3.1.1 Distributed Thread Scheduler (DTA-Style) (UNISI)

In the TERAFLUX system Data-Flow threads are scheduled and run asynchronously after: i) the input data has arrived and ii) the Distributed Thread Scheduler takes the decision to start the thread based on the available parameters in the core-record, the availability of resources (see

D6.1 for a more detailed description of this) and the need to repeat the execution due to a fault (see the subsequent sections and [6]) or due to a conflicting transaction.

The Distributed Thread Scheduler, or DTS, is designed with a distributed and hierarchical approach: in Figure 3 we abstract the design of the DTS from the same design presented in Figure 1: each core gets a Local Thread Scheduling Unit (L-TSU). All the L-TSUs can communicate at Node level and to the Node level thread scheduling unit, called Distributed Thread Scheduling Unit (D-TSU). Considering the distributed architecture of the scheduler, we called it Distributed Thread Scheduler (DTS). In the following sections we describe the main architectural design features we consider for the implementation of the DTS.

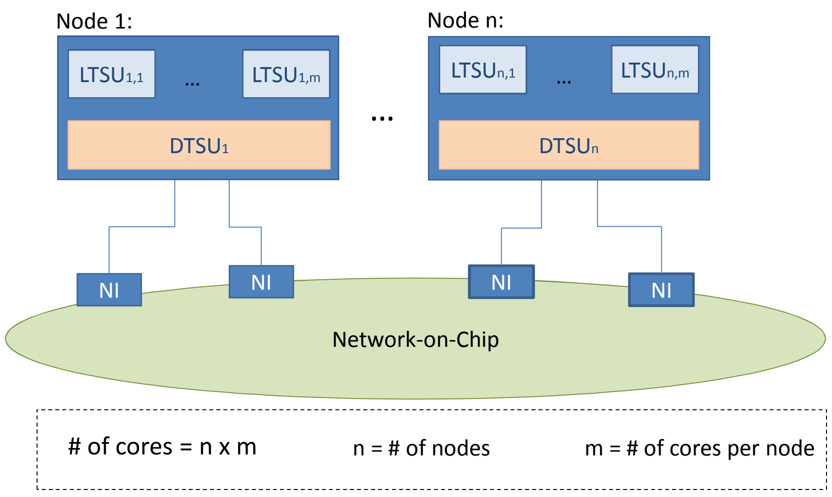

Figure 3: Overall view of the DTS organization

Figure 3 shows the overall view of the hierarchical DTS organization. Each Node in the system is comprised of one D-TSU and *n* local L-TSUs. As previously mentioned, the DTS is based on a hierarchical and distributed design. This hierarchical structure is currently proposed in two levels but could be extended in the future to multiple levels. The main scheduling activity is performed at the core level by the L-TSU. This unit maintains the list of all ready for the execution threads, and it manages them through the DTS continuation. In particular the L-TSU is responsible for the allocation of the memory regions associated to each thread (see Section 2.1 and Figure 2). This L-TSU activity is mainly caused by the execution of the TSCHEDULE, TDESTROY and TSTORE instructions. Similarly, at the end of the execution of the thread, the memory area allocated such thread is made free. The L-TSU exchanges information with the D-TSU.

The D-TSU is responsible for keeping a global view of the Node, and whenever a L-TSU requests a memory allocation, the D-TSU performs a look-up within its data structures. As the D-TSU knows the current Node-level resource allocation, it can return a memory pointer within the

Node or it can deliver the request to the neighbour D-TSUs. It is also responsible to periodically communicate with the Fault Detection Unit (D-FDU and L-FDU) to keep track of the faulty cores (i.e., in this way it avoids to return a memory pointer from a faulty core). D-TSU and L-TSU also implement the mechanism used to recover from a fault detected (see Section 2.2.3).

Information about temperature, power consumption and level of faultiness of each core in the Node are used by the L-TSU and the D-TSU to perform the Data-Flow threads scheduling.

From this viewpoint the thread scheduling decisions ensure that the temperature of each core remains below a predefined threshold. The DTS receives updates of the core temperature from the D-FDU.

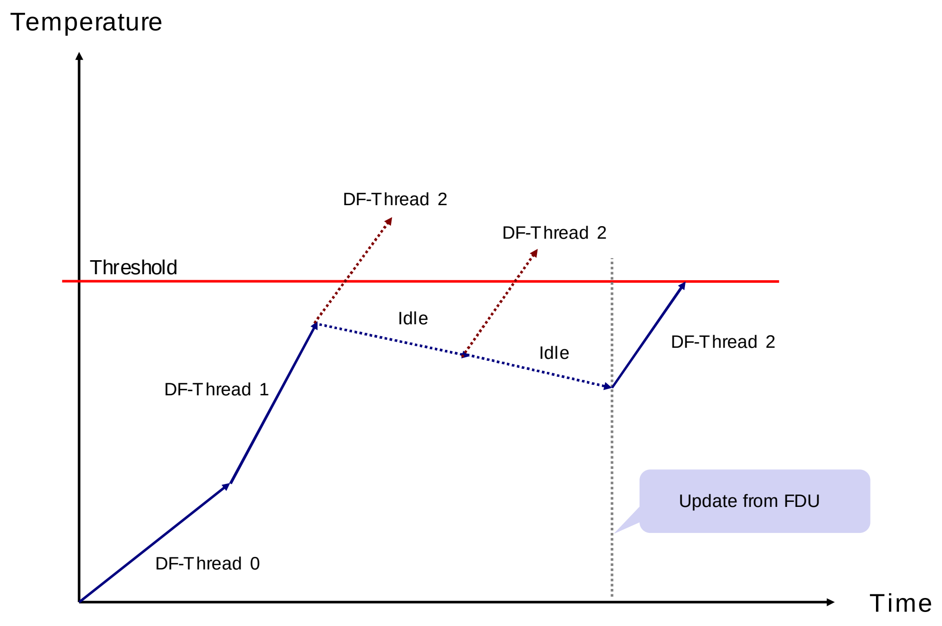

Figure 4: Example of Temperature-aware scheduling

Figure 4 shows an example of the temperature of a running core. The temperature threshold used to take scheduling decisions is highlighted with a horizontal red line. Blue arrows represent the current scheduled Data-Flow threads. Whenever a Data-Flow thread is assumed to increase the core temperature above the threshold, it is represented by a red-dashed arrow. Finally, blue-dashed arrows represent idle states of the core. The temperature-aware thread scheduling is distributed between the L-TSUs and managed by the D-TSUs as follows (cf. also D7.3):

- L-TSU level: when considering a candidate thread for scheduling, the L-TSU predicts the temperature change based on static energy bounds and the current dynamic voltage-frequency scaling (DVFS) state of the core. When it determines that the action of scheduling such thread would cause the core to exceed its temperature threshold, the core is put in an idle state. Temperature is estimated locally using step-by-step integration of the estimated energy as threads are scheduled on the core. This estimation is kept deliberately pessimistic. To minimize the bias in the temperature estimation, the DTS can use more accurate temperature of the heartbeat messages it receives periodically from the D-FDU.

- D-TSU level: by using the existing mechanism of the scheduling augmented with the temperature monitoring fields of the core-record this unit can offload threads to the D-TSU of other Nodes. In both cases, the implementation may be based on task stealing: when a core or Node is idle and its temperature is lower than a threshold T_{steal}, the D-TSU will attempt to offload threads to the Preload Queues of other cores/Nodes. Precedence is given to the core with the highest temperature and longer queue. We plan to investigate the performance of more specific policies.

2.3.1.2 Thread Scheduling Unit (DDM-Style) (UCY)

This section describes the proposed hardware design of the Thread Scheduling Unit (TSU), a hardware support unit for the TERAFLUX processor, responsible for DDM-style thread scheduling based on the Data-Flow execution model. The DDM-style TSU offers support for code including function calls and recursion as well as runtime thread dependency resolution [8].

Hardware Design of a Thread Scheduler Unit

The goal of this work is the hardware design specification of the TSU with a Hardware Description Language (HDL) and its evaluation on a FPGA simulator as to provide accurate values for the number of resources, latency, and power consumption of the functional units of the TSU.

Figure 5 depicts the datapath of a Node with emphasis on the internal structures of the TSU and the FDU. The TSU consists of two units: the Distributed TSU (D-TSU) and the Local TSU (L-TSU). For the DDM-style execution model, the D-TSU is responsible for the scheduling of threads based on the Data-Flow execution model, and is common for all cores in the specific Node. Furthermore, the D-TSU is responsible for the assignment of threads to cores as well as the communication between D-TSUs of other Nodes through the NoC. The L-TSU is located within each core and can be accessed directly by the Processing Unit (PU). The L-TSU acts as a fast link between the PU and the D-TSU. More specifically, it acts as a buffer of pointers to the threads to be executed in the near future, as well as a cache prefetching unit.

The D-TSU consists of four units that operate asynchronously with each other. These units are the following:

(a) **Acknowledgement Unit (AU):** This unit consists of the *AckBuffer* and the *Acknowledgement Queue*. Its function is to receive messages concerning the consumers of the threads just completed by the local cores or from remote TSUs and forward those either to the Synchronization Unit or to the Network Interface Unit for further processing.

(b) **Synchronization Unit (SU):** This unit consists of the *Consumer/Context Select Unit*, the *Synchronization Memory* and the *Ready Queue*. Its function is to decrement the *Ready Count* entry for the specified consumer/context. If the *Ready Count* becomes zero then the thread is deemed executable and is shifted to the *Ready Queue* for further processing.

(c) **Scheduling Unit (SchU):** This unit consists of the *Graph Memory*, the *Core Select Unit*, the *Virtual-to-Physical Core Table*, the *Thread-to-Core List* and the *Ready Queue Buffer*. Its function is to read the identification of ready threads from the *Ready Queue*, retrieve the corresponding information from the *Graph Memory* and determine the core to which the thread must be assigned for execution.

(d) Network Interface Unit (NIU): This unit consists of the Receive and Transmit buffers. Its function is to receive thread continuations from remote D-TSUs and shift them into the AU, and to receive thread continuations for remote TSUs from the AU and forward them to the remote D-TSU.

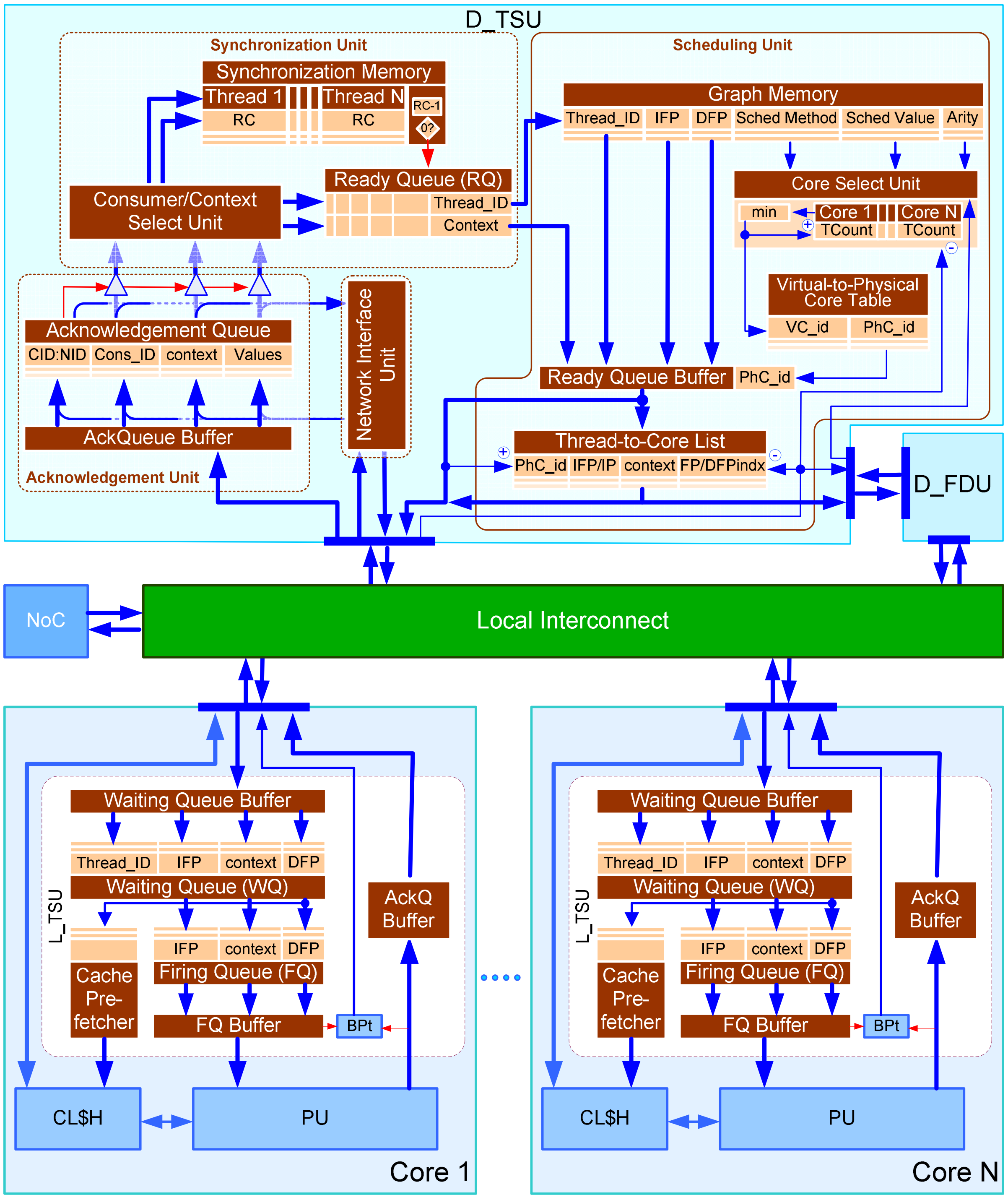

Figure 5: Datapath of the TSU and the FDU

Thread Scheduling Unit Operation

Upon completion of the execution of a thread, the PU issues a "**TUpdateConsumer (consumer_id, OP, context)**" operation that loads on the *AckQ Buffer* a continuation of the thread just executed, that includes information identifying the thread's consumers, as well as hints on how to treat their context. This information is forwarded to the Acknowledgement Unit (AU) of the D-TSU through the Local Interconnect. The AU receives continuation information from any one of the cores within the Node, or from the D-TSUs of other Nodes through the Network Interface unit (NI). The AU uses the information on the *CID:NID* field to determine if the received continuation refers to a thread to be executed locally or by a remote Node. In the first case the continuation is shifted to the Synchronization Unit (SU), while for remote threads, the continuation is forwarded to the NI for further processing.

The SU receives continuations of consumer threads from the AU. This information includes the thread ID number as well as hints on how to manipulate the context field of the thread. The SU locates the *Ready Count (RC)* of the consumer thread with its context in the SM and decrements it. If the RC becomes zero then the thread is deemed executable. In this case, the thread ID and its context are shifted in the *Ready Queue (RQ)* for further processing by the SchU.

The SchU reads the thread ID of the next thread to be processed from the RQ. The thread ID is used to address the entries in the GM, where the template of the thread is stored. The DDM model supports three thread scheduling mechanisms. In the current design we have implemented only the mechanism that is based on load balancing. To achieve this, an up/down counter is maintained for each core. When a thread is assigned to a core, its counter is incremented. When the core completes the execution of a thread, then it sends a signal to the SchU to decrements its counter. Hence, these counters show at any time the number of threads waiting in the WQ and the FQ of each core. The SchU assigns the thread to the core with the minimum count. If the counter reaches its maximum value, then the WQ and the FQ for this core are full.

The L-TSU receives the pointers of threads assigned to them through the *Waiting Queue Buffer.* These pointers include information on the data to be used by the thread. The *Cache Prefetching Unit* prefetches the data in the cache and the thread pointers are shifted into the FQ. The PU reads the IFP and DFP from the FQ Buffer and continues with the execution of the thread.

2.3.2 Hierarchical Thread Scheduler or HTS (BSC)

This section briefly recalls that in Deliverable D6.1, Section 5.2, we described the HTS or Hierarchical thread Scheduler as another possible approach to build support for the thread scheduling. In particular in Section D6.1-5.2.1 the Task Superscalar Pipeline is described.

2.3.3 Support for Transactions (UNIMAN)

In this section we describe the issues involved in designing a Transactional Memory system for TERAFLUX, and report on progress made so far.

In keeping with the principles guiding the TERAFLUX architecture more broadly, the Transactional Memory mechanisms are being designed with the following requirements in mind:

1. Performance should be achievable without an undue burden on the programmer.
2. The system should scale gracefully.
3. The system should be able to cope with, and if possible exploit, a hierarchical organization of cores into Nodes.

To maintain the Transactional Memory behaviour discussed elsewhere *[Deliverable 3.1]*, the Transactional Memory hardware must perform a number of tasks. Transactional modifications must be isolated from the rest of the system until commit time through the versioning of data. Further, the system needs to detect and to resolve conflicts. This means that before a transaction can commit, it must ensure that it has not consumed any values that have, since then, been modified. Transaction commits must appear to occur atomically. In case of conflicts, the system must resolve them. Transactions that are selected for abort must be rewound such that a consistent state is reached. Data versioning and transaction rollback are issues local to a core.

Unlike versioning and rollback, conflict detection and resolution are system-wide issues. As such, these are the key issues that need to be addressed when designing a scalable Transactional Memory system. Early systems either relied on a broadcast medium or on centralized mechanisms, both of which resulted in poor scaling. Since then, there have been a number of proposals for scalable systems. Scalable TCC [9] extends a directory based coherence protocol to provide a transactional memory protocol. It still includes a centralized mechanism for ordering transaction commits. Upon commit, transactions communicate with all the directories in the system, and lock those that own data in the write set of the transaction. Since this locking is done at a coarse (directory) level serializing commits, it can be significant bottleneck. Scalable BulkSC [10] addresses some of these problems, and no longer locks entire directories when committing. It uses signatures to summarize read and write sets, in order to quickly establish whether transactions that are attempting to commit concurrently are guaranteed to be independent. EazyHTM [11] differs from the proposals discussed above, in that transactions are eagerly notified of any sharing. This results in more communication during the execution of a transaction, but means that truly parallel commits are possible in cases where transactions do not conflict. When conflicts do exist, their detection at commit time is greatly simplified.

The TERAFLUX TM system differs from earlier evaluations in a number of important ways. The first is scale. Scalable TCC, Scalable BulkSC and EazyHTM have all been evaluated only up to 64 processors. The second is the wider range of workloads we target. The third is that we operate in a different environment when it comes to memory coherence and consistency requirements. Because of the nature of Data-Flow computation, the TERAFLUX memory system does not need to provide the same coherence guarantees as current multicore systems. This changes the tradeoffs for TM protocols, since the marginal cost of protocol events depends on the underlying memory system. The fourth is the change in costs from global to local events due to the clustered architecture. Because of these differences, we revisit some of the basic ideas involved. The questions we aim to answer include the following:

- Is it better to exchange information about sharing between transactions as they go along or to do so only at commit time? The first option involves more communication, and possibly latency, when performing individual transactional loads and stores, but may simplify the commit process.
- If a logically centralized mechanism turns out to be the best choice, is it possible to physically distribute it in a manner that scales?
- How can we leverage the clustered architecture to provide good performance for transactions? Is it useful to have different Node local and global mechanisms? It is certainly possible to exploit the broadcast medium (or low latency local communication) within a Node to optimize certain cases where transactional sharing does not escape a cluster. However, this would only help if such cases are sufficiently common. This is related to the next point.
- What sharing patterns exist across a broad range of workloads? In light of these, what is the best balance between communication, storage and false sharing? It may be that consistent performance can only be achieved through adaptive mechanisms.
- Transmitting entire read and write sets involves significant communication overhead. Using signatures has been proposed to address this. However, this too has the potential to lead to false conflicts. If signatures are to be used, what is the best summarization technique, and is there scope for adaptive techniques?

Figure 1 presents TERAFLUX architectural template. In order to implement TM, we propose modifications at the cache level hierarchy within the core and also at the Node level to provide TM support across Nodes. The modifications enable the versioning of the modified data as well as the conflict detection mechanism. The modifications can include extra bits for cache lines (or coarser granularity) to represent the versioning status. These extra bits can be completed with Bloom filters as an efficient representation of sets (write-sets and read-sets). Within this context, we are considering how best to answer the questions posted above.

Currently, we are exploring both purely lazy and EazyHTM-like eager-lazy techniques. This has involved, at an implementation level, building functional support for Transactional Memory into simulation platform. We have interacted with AMD and HP to include it in the functional simulation. This has proven not to be easy endeavour. For performance evaluation, we have extended the COTSon simulator by implementing a directory based cache coherence protocol as a starting point for our baseline system. This is being extended with a timing model for commit mechanisms similar to Scalable TCC and Scalable BulkSC. This includes work in establishing simulation methodology. We are evaluating ways of modeling the Transactional Memory such that we can perform large-scale architectural simulations while maintaining acceptable accuracy.

2.3.3.1 DTS+TM: initial proposal for integrating TM support in the DTS (UNISI)

The thread scheduling capabilities of the DTS can be leveraged to handle Transactional Memory. Two different mechanisms can be used, depending on the complexity of the transaction: atomic threads and transactional threads. Both mechanisms are completely dynamic in the sense that they operate on dynamically calculated addresses where a concurrent operation will happen.

Atomic threads are a class of Data-Flow threads that support arbitrary operations on a single memory location or a single object. Figure 1 gives an example of the scheduling operations when two atomic threads access a common variable *x* in the shared memory. For efficiency reasons, both threads can be scheduled to the same core. The DTS will ensure their execution happens sequentially.

The frame of atomic threads contains a pointer to the memory location in the Transactional Memory (TM) space concerned by the atomic access. The core in which atomic threads are scheduled is decided according to their memory pointers. A main idea behind atomic-thread is to move the computation where the data resides, rather than move the data. Hence, the critical section is limited to the computations, and does not contain any long-latency load or store.

Another important point is that there is no concept of “spinning from a given core”: the computation is always enabled on the data availability. In the case of atomic-threads, the conflicting thread, either gets enabled immediately because there’s no other thread using the same data address in the TERAFLUX-TM or it will be just another thread in the waiting queue of the DTS (it will get a dummy increment to its Synchronization Count – SC). Once the eventual conflicting thread will commit, it will also send a dummy write to one of the waiting atomic-threads on the same data address. Again, this is managed completely within the existing DTS infrastructure.

This lightweight hardware mechanism leverages the existing scheduler. Atomic threads are dynamically scheduled. As one atomic thread may only operate on the same memory range, and because no ordering is enforced between atomic operations, deadlock conditions can be avoided.

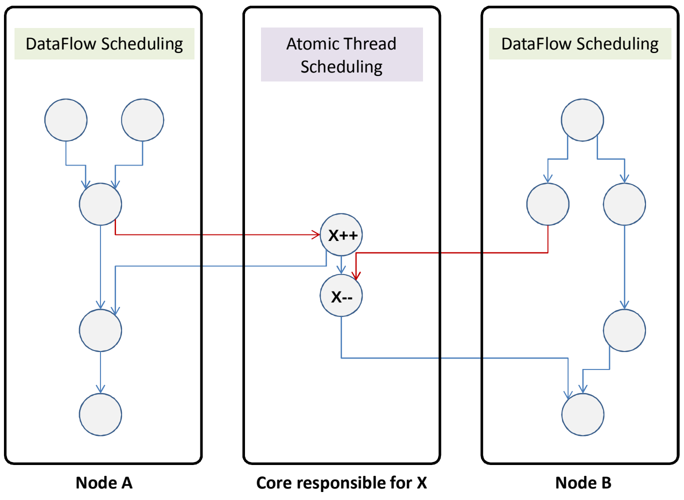

Figure 7 - Example of scheduling with atomic threads

To minimize congestion, the atomic memory space is distributed across all Nodes. In the initial implementation, we plan to use a straightforward modulo mapping from memory blocks to Nodes. Further, dynamic mappings that allow load-balancing and resiliency will be considered in a second phase. Although atomic threads are limited to a single memory location or object and are not composable, the hardware support for atomics provides a foundation to build generic software transactional memory implementations.

The second implementation option, transaction threads, consists in re-using the re-execution support dedicated to fault recovery built in the D-TSU and L-TSU to cancel transactions. Unlike atomic threads, transaction threads support an arbitrary number of read and write accesses to transactional memory locations.

One possibility to extend this mechanism to more complex scenarios is under research: for example, we could augment the frame of transactional thread with a data versioning record, which contains copies of the data written (similarly to lazy data versioning). When a transaction thread encounters a conflict and has to be canceled, we use the fault handling mechanism that is described in [6] and in the section below related to the Data-Flow re-execution.

In the current design of Figure 1, the L-TMU and D-TMU units are put in evidence as the blocks that may contain part of the logic connected to the Transactional Memory besides the DTS.

2.3.3.2 TM Support for the DDM-style DF-thread execution model (UCY, UNIMAN)

There are several different ways to support transactions. We will start with the description of the simplest way and then discuss the exploration of other alternative solutions that may be used in the future as optimizations.

For the DDM-style DF-threads, a Transaction is annotated in the code using pragma directives. These directives are translated into transaction instructions such as *transaction_begin* and *transaction_end*. The *transaction_begin* instruction marks an execution point for the thread in case it needs to be rolled back. The same instruction also triggers the D-TMU unit to monitor the accesses to the TM as to determine any conflicts on those accesses. When the execution of the transaction completes, a *transaction_end* instruction is issued. This instruction is in charge of triggering a check with the D-TMU unit as to determine if the transaction has succeeded or failed. In case of success the execution continues normally after the *transaction_end* instruction. Otherwise, the D-TMU restores the original state of the thread before the transaction started and execution is re-started from that point onwards. This procedure is transparent to the regular execution of the TSU as the thread is not executed until completion and thus the TUpdateConsumer message from the L-TSU to the D-TSU is not sent. The only issue is that feedback from the re-execution of the thread should be sent to the D-TSU as a way to better schedule the threads in case that two conflicting threads are continuously scheduled at the same time. This is achieved by adding a field *re-exe* to the thread template and each time the *transaction_end* instruction returns FAIL, while rolling-back the state, the D-TMU sends a *incrementReExe(threadID)* to the D-TSU. This message can also include other information about the other conflicting transaction.

The above described approach relies fully on the operation of the D-TMU. In the future we will be exploring other alternative approaches which offload some of the operations to the TSU. For example, given the characteristics of the DF-Threads and that they have no side-effects, we will explore the restarting of the thread by the L-TSU. Thus upon completion of a thread, the D-TMU module would be queried for the success of the transaction. In case of success, the thread would send the regular TUpdateConsumer to the D-TSU, otherwise the same thread would be re-executed by the L-TSU.

2.3.4 Integration of Fault Detection Techniques into the TERAFLUX Architecture (UAU)

The Deliverables D5.1 and D5.2 describe Fault Detection Units (FDUs) on core and Node level, which are integrated as hardware support units in the TERAFLUX fault detection and recovery architecture. Furthermore, Deliverable D5.2 distinguishes between the Distributed Fault Detection Units (D-FDUs) on Node level and Local Fault Detection Units (L-FDUs) on core level.

The D-FDU is an adaptable observer-controller unit. As such, it autonomously queries and gathers the information of all cores within its Node by heartbeat messages over the potentially unreliable intra-Node interconnect. In addition, D-FDUs monitor each other sending heartbeat messages over the inter-Node interconnect in order to detect faults of other D-FDUs in other Nodes. Also I/O controllers and memory controller are subjected to the same heartbeat message based monitoring like normal cores. The D-FDU analyzes the gathered information and provides the D-TSU with information about the state of the whole Node and other D-FDUs. The D-FDU is supported by the L-FDUs with each Node's core.

The L-FDU is a small hardware component integrated with each core to support fault detection and data gathering by the D-FDU by extracting information from the core-internal fault detection mechanisms described in the Deliverables D5.1 and D5.2, i.e., the Machine Check Architecture (MCA), the Performance Counters, and the Control Flow Checker (see D5.2 Section 2.2.1 "Control Flow Checker"). We may incorporate information provided by wear-out detection sensors in project year 3, when UAU focuses on the intra-Node fault detection techniques.

Based on the information provided by the L-FDU, the D-FDU detects core and link faults and proactively prevents faults by individual voltage and frequency scaling. Finally, the D-FDU informs the Node's D-TSU about the faulty core and links, while the D-TSU is responsible for the appropriate thread scheduling.

More details about the fault detection and recovery architecture in TERAFLUX can be found in the Deliverables D5.1 and D5.2, and publications [6,12].

Refined Interface

From the fault tolerance point of view we propose to share information regarding reliability and performance. Additionally, we provide a special reliability value called *wear-out factor* (WOF). In the following, we describe briefly those key metrics and how we plan to publish them to the D-TSU.

The reliability value (RV) is a measure of the expected reliability of a certain core. A value of 0 represents a complete unreliable core, where a value of 1 stands for a reliable thread execution for a certain amount of time. The D-FDU, however, will proactively take actions to reach a value of 1.

The performance value (PV) is the representation of the core's thread execution capabilities. That means, if $PV = 0$, then the core is not able to execute any thread (the reason here could be a high core temperature), whereas $PV = 1$ represents the state that the core is running at highest clock speed. The PV could also be seen as a metric for reliability (a bad performance value can be a sign of a permanent or intermittent fault), the PV and RV values may differ clearly. Imagine a situation, where the temperature of a core is too high for a fault free thread execution. The FDU may decrease the clock rate of this core resulting in a reliable thread execution (high RV) at a slow clock rate (low PV).

The wear-out factor (WOF) represents the "age" of a core. As we already stated in D5.1 we have to deal with "aging" components. An unusual frequent usage of a core may accelerate the aging for this core. In order to measure the grade of aging, we propose the use of wear-out sensors. From these sensor values we calculate normalized wear-out factor. The D-TSU may take advantage of the WOF value by placing the threads on those cores, which have not suffered from aging so far.

All three values are stored into a list called *"Core State List".* This list resides at the moment in the D-FDU. It contains also the CID value, which represents the unique core identifier used by both execution models DTA and DDM. In order to avoid creating extra load on the communication network, this table is only updated by the FDUs whenever the changes in the values in the table pass a certain threshold.

The integration of these fields in the common "Core Record" will be investigate in Year-3, while specifying a more detailed TSU-FDU interface.

Core State List			
CID	RV	PV	WOF

Table 2: The Core State List contains reliability and performance values for each core in the Node.

2.3.4.1 Data-Flow re-execution in presence of faults (UNISI, UAU, MSFT, HP)

According to the estimates of the International Technology Roadmap for Semiconductor [13], a system like that the TERAFLUX consortium chosen as a target for the project, will be based on not reliable transistors. The unreliability of the transistors is mainly due to the continuous shrinking of their size and the reduction of the voltage supply. In these conditions cosmic rays, thermal fluctuations, and manufacturing process variations will induce more likely failures in future systems than current ones [14]. To overcome with the increasing failures coming with new manufacturing technologies, the integration of fault-tolerant mechanisms becomes a must.

In this context, we explored the Data-Flow execution model to provide fault-tolerance against transient, intermittent, and permanent faults. The proposed mechanism is mainly based on the re-execution of the Data-Flow threads, control-flow checking, and check pointing. Moreover, the approach targets standard x86-64 cores with incorporated control flow instructions to support micro-control flow within a thread.

According to the DTS architecture (described previously in this document), faults are hierarchically managed. The L-FDU monitors continuously the core to which is connected to and sends periodically health state messages to the D-FDU. Additionally, if an urgent event takes place, the L-FDU sends an urgent message. The D-FDU is responsible to collect all health states of the cores within the Node. Moreover it communicates the overall state of the Node to the D-TSU, allowing the scheduling of threads on fault-free cores. In our approach, D-FDU is also responsible to monitor other D-FDU units in other Nodes in a peer-to-peer fashion, and to monitor L-FDU enhanced memory controllers similar to cores.

We based the activity of the D-FDU on the implementation of a MAPE autonomous computing system, which applies the following four actions: monitoring (M), analyzing (A), Planning (P) and Executing (E). It is expected to receive health status messages from the monitored cores (i.e., both regular cores and D-TSU) in a certain amount of time. A missing message is interpreted by the D-FDU as a fault appeared in the core. If subsequent health messages also not arrived at the D-FDU, the associated core is assumed as permanently faulty.

An event-driven mechanism is added, in order to allow L-FDUs to inform the D-FDU of the occurrence of a fault. All the information gathered by the D-FDU is periodically sent to the D-TSU, allowing a correct scheduling of the threads. As previously mentioned, the D-FDUs are monitoring each other against faults in a peer-to-peer fashion. This mechanism results in a complete hardware assisted fault detection process. Further, to improve the reliability of the target system, we consider D-FDU able to act proactively, thus dynamically changing both the frequency and the voltage supply of the Node depending on the fault rate, temperature and workload.

L-FDUs use two mechanisms in order to detect a fault: reading the machine check architecture (MCA) registers and reading the control flow checker (CFC). We consider a minimal MCA in each core able to detect faults for fetched instructions and data, for ECC checksum errors in registers, Frame Memory, and caches. We also add a control flow checker. It works as follows: at compile time a software check point (i.e., a set of instructions without control flow such as jumps, branches, calls, etc.) is inserted at the beginning and at the end of a basic block. This additional piece of code contains information about the expected behavior of the core when the basic block is executed. At run-time a dedicated hardware unit monitors the pipeline comparing the behavior with the expected one (gathered from the software instrumentation of the check point).

Threads with a special reliability demand can be executed twice (either on two cores or subsequently on the same core) to ensure the correctness of the thread execution. For dual execution, we duplicate the continuation of the double executed thread. Then the D-TSU schedules the threads (a leading thread and a tailing thread) accordingly to its scheduling policy. Write backs from the finished thread execution are re-directed to the D-TSU. The D-TSU buffer this writes and holds them back until the correctness of the thread execution has been verified.

After the threads have finished their execution, the core writes back its result (which is now redirected to the D-TSU). While the result is written back, the L-FDU tracks the write back and calculates a signature from them. This signature is sent to the D-FDU, where it is compared with the signature from the duplicated execution. Equal signatures indicate that both threads were executed the same way (most likely without an error). We can assume this, because a fault is unlikely to appear in both thread executions resulting in the same error. Therefore, we start recovery only in case where the signatures are not equal.

If the signature comparison detects a fault, the D-FDU signals the D-TSU the re-execution of the leading thread. After the third execution, we may have enough result sets (2 sets from the dual execution and 1 of the single re-execution) to determine which executions were correct.

The adapted Data-Flow execution model allows scheduling a thread for a re-execution without any side effects, since there is no dependency from the execution of other threads during its execution, and since all the writes are performed at the end of the execution. In particular all the writes are kept in a temporary buffer until the final D-FDU response is available. Thus in case of a fault free execution all the buffered writes are committed, while in case of a faulty condition they will be simply discarded.

It is worth noting that the usage of MCA and CFC at the core level to detect a faulty behavior and the above described double-execution of the threads are two complementary approaches that we want to explore in the TERAFLUX target system to guarantee an adequate level of resiliency.

2.3.4.2 Fault-tolerance support for the DDM-style DF-threads execution model (UCY)

For the support of core faults we use three metadata tables in the D-TSU. One table is the *Core-Status* table, which is a copy of the Core-Status-List table from the D-FDU. In order to avoid creating extra load on the communication network, the TSU copy of this table is only updated whenever the changes in the values in the table pass a certain threshold. Another is the *Thread-to-Core* table that includes the *threadIDs* of all threads sent for execution to each core. Finally the other table is the Virtual-to-Physical-Core table that contains the mapping of the virtual cores to the physical ones. This is used as a transparent way to support core failures without implying any changes to the scheduling policy that may have been determined statically at compile time. For example, at the beginning each virtual core point to a corresponding physical core. But if during the execution the system has core X failing, then we will pick another physical core to execute the work of virtual core X. The choice of this core will be done using the information in the core-status table as to pick a core that is less loaded and thus more able to handle the extra work.

In case of a core failure, the D-FDU unit will notify the D-TSU of that fact with a *coreFail(CoreID)* message. The D-TSU will first select a substitute core for the work by querying the Core-Status table for a less loaded core. Then will exchange the physical core assigned to the virtual coreID to be the new substitute core. After that it will use the coreID to get all the threads that were executing on that thread which are stored in the Thread-to-Core table. It will then send all those thread information to the new physical core that is assigned to the virtual coreID. Transparently to the rest of the threads the execution will continue normally. Additionally, recovery actions may be required and will be investigated in year 3.

In the future we will explore alternative approaches that may dynamically redistribute the balance of the work in case of failures that end up overloading certain physical cores.

2.4 Hardware Synthesis

2.4.1 DTS (UNISI)

Given the proposed architecture, we estimated the area consumption of the proposed DTS unit. In order to correctly estimate the area consumption we needed to define the internal architecture of the DTS units (i.e., the D-TSU and L-TSU) in terms of internal structures, and to select an evaluation metric.

For the definition of the internal DTS architecture, we initially used the model proposed in D6.2 for DTA. The main area cost for the DTS comes from the internal structures used to manage DTA-style threads. In particular the L-TSU has the Pre-Load Queue (PLQ) and the Waiting Table (WT), while the D-TSU is mainly composed of the Pending TSCHEDULE Queue (PTQ) and the Frame Free Table (FFT). The presence of these four structures is directly related to the operations performed by the two hardware units. In fact, the L-TSU is mainly responsible for the allocation of frame memory regions for the execution of the threads and for executing threads whose synchronization counter equals zero (i.e., all the inputs are available). These threads are managed through the PLQ circular buffer. The threads whose synchronization counter is greater than zero are managed by the WT. Since the D-TSU is responsible for distributing the workload among the cores, the FFT is used for counting the number of free frame regions within each core. This information is directly related to the load on each core. Whenever a frame creation request is received, the D-TSU selects one of the cores, looking for a free frame memory region. Since this request can potentially fail, the PTQ is used to temporary store pending requests. Whenever frames become free, threads are removed from the PTQ and scheduled on a core.

Since the way DTS unit works is strictly related to the adopted memory model, we consider specific fields contained both in the DTS continuation and in the Core Record (CR). From this viewpoint, DTS continuation has pointers to manage different memory regions such as TM, OWM and TLS regions. The CR has a set of registers containing information about the core identifier (CID), the power consumption, the temperature and the faultiness level of the core. All these structures must be considered for the correct overall area estimation of the DTS unit.

In order to measure the area cost, we observed that these structures are essentially devoted to store information. Thus, we decided to use the Register Bit Equivalent (rbe) [15] to estimate their area. The rbe measures the area consumption in terms of the area of a single bit storage cell. A single bit storage cell is basically built as a six-transistor SRAM cell with high bandwidth and that is isolated from its input/output circuit. Therefore, the area occupation is well-known for each technology Node.

Hereafter we report the estimated area cost, considering a single Node. Table 3 shows the main area costs for the four storage structures that compose the L-TSU and the D-TSU.

Hardware Unit	**Internal Structure**	**Area [rbe]**
L-TSU	PLQ	$n_F \cdot (\text{size}_{IP} + \text{size}_{FP} + \text{size}_{TLSP} + \text{size}_{OWMP} + \text{size}_{TMP} + \text{size}_{CID})$
	WT	$n_F \cdot (\text{size}_{IP} + \text{size}_{FP} + \text{size}_{SC} + \text{size}_{TLSP} + \text{size}_{OWMP} + \text{size}_{TMP})$
	CR	$\text{size}_{CID} + \text{size}_{Fault} + \text{size}_{Power} + \text{size}_{Temp}$
D-TSU	FFT	$\text{size}_{FFT\text{-}entry} \cdot n_{DTA\text{-}PU}$
	PTQ	$n_{PTQ} \cdot (\text{size}_{IP} + \text{size}_{SC} + \text{size}_{ID} + \text{size}_{TLSP} + \text{size}_{OWMP} + \text{size}_{TMP})$

Table 2: L-TSU and D-TSU area estimation in terms of register bit equivalent for the internal structures

As reported in Table 3 the area cost of the PLQ, WT, FFT and PTQ depends on the relative size of these structures. If we look at the single core, the n_F parameter models the number of frame memory regions that can be managed (this number is also equal to the number of threads that can be managed by each core, in the hypothesis of a single frame area associated to each thread). The PLQ needs to store only the instruction pointer (size_{IP}) , the frame pointer (size_{FP}) , the OWM pointer (size_{OWMP}) , the TLS pointer (size_{TLSP}) , the TM pointer (size_{TMP}) and the core identifier (size_{CID}) for each thread ready to execute. The WT entry is slightly different, since it has to store the synchronization counter for each waiting thread (size_{SC}). For each core, a CR structure is maintained, storing dynamically updated information about the power consumption, temperature and faultiness level, plus a replica of the core identifier. All these information can be easily stored in a set of four registers with respectively the $\text{size}_{CID} + \text{size}_{Fault} + \text{size}_{Power} + \text{size}_{Temp}$ relative sizes. Here, it is worth to recall that the TERAFLUX architecture is based on a large set of Nodes, each of them comprising a variable number of cores. For all of them we can assume that a fixed length of 1B is enough to represent up to 256 cores in a single Node, and up to 256 levels of power consumption, temperature and faultiness level (i.e., $\text{size}_{CID} = \text{size}_{Fault} = \text{size}_{Power} = \text{size}_{Temp} = 8$ bit).

Looking at the Node level, the $n_{DTA\text{-}PU}$ and the n_{PTQ} parameters, respectively gives the number cores in the Node, and the number of entries in the PTQ buffer. Similarly to the WT structure, the PTQ entry stores also the identifier of the core from which the request for a frame area is originated (size_{ID}). Finally, the FFT structure needs only to hold an entry for each processing unit (i.e., each core) in the Node.

The number of bits required to store the frame pointer (size_{FP}) associated to a thread can be assumed the same as the other pointers. The number of bits required to store an entry in the

FFT (size $_{\text{FFT-entry}}$) equals $log_2\ (n_F) + 1$.Finally, the number of bits required to store the identifier of the frame allocation request (size $_{\text{ID}}$) is equal to $log_2\ (n_{DTA\text{-}PU})$.

The total hardware area cost is the sum of the costs of the single hardware structures implemented in the L-TSU and D-TSU. As an example (similarly to [16]), let us consider a Node with 8 cores ($n_{\text{DTA-PU}} = 8$). If we suppose that each core:

- Has a local memory (here we can assume that the cache hierarchy is compressed to a single level, thus no cache memories are implemented at the core level) of 512KB;
- The instruction pointer has a length of 64 bits (size $_{\text{IP}} = 64$);
- Maximum value for the synchronization counter equals to 256 (i.e., size $_{\text{SC}} = 8$);
- Number of frames per core equals to 8, with each frame composed of 64 entries of 8 bytes ($n_F = 8$);
- D-TSU has the possibility to store up to 8 pending requests ($n_{\text{PTQ}} = 8$);

Given these values, the space required for managing the frame regions equals to 4KB, thus leaving enough space to store the code and data in the cache memory (512KB - 4KB). Reasoning about the occupancy of the core-level structures, we have a value of 2624 bit for the PLQ and the WT, while we have a fixed size for the word equals to 64 bit (i.e., 8B). The cost of the CR structure is equal to 4B. Summing the cost of PLQ, WT and CR, we obtain a total cost for the single L-TSU unit of 5280 bit; thus, the total cost for the implementation of the L-TSU units equals to 42240 bit (i.e., corresponding to 5,28KB). This value represents the total cost for the L-TSU obtained by multiplying the occupancy of each structure on each core by the number of cores in the node. On the other hand, the area cost for the D-TSU structures reaches 276B. In particular, the FFT has occupancy of 4B, while the occupancy of the PTQ structure is 2176 bit. Taking into account these values, from our first estimation the total cost for the implementation of the DTS reaches the value of about 5.56KB that represents the 1.08% of the total cache memory available on the Node. Increasing the number of frames up to 80 (10 times the previous value) will increase the area occupancy up to about 52.51KB (about 10% of the cache memory). It is worth to recall here that the size of 512KB for a cache memory is a common value for current many-core chips; hence, we expect that the area cost for the DTS will become negligible in the context of a tera-device with the availability of larger cache memories.

2.4.2 DDM-TSU (UCY)

For validation and evaluation purposes, the *DDM*-TSU has been developed using the Xilinx Embedded Development Kit (EDK) having as a target the Xilinx Virtex-5 FPGA running at 153MHz. The sizes of the *DDM*-TSU structures are given in Table 4. The hardware budget for the *DDM*-TSU implementation is shown in Table 5.

Structure	Number of Threads	Bytes/Thread	Memory Size
Graph Memory	256	16	4KB
Synchronization Memory	64	64	4KB
Acknowledgement Queue	16	4	64B
Ready Queue	16	4	64B
Waiting Queue/core	8	4	32B
Firing Queue/core	8	4	32B

Table 4: Sizes of DDM-TSU structures

	F/Fs	LUTs	BRAMs
TSU FPGA Resources	3764	3835	12
% of FPGA resources	11.5	11.7	9.1

Table 5: DDM-TSU Hardware budget

Table 6 shows the latency of the four units of the D-TSU as well as the latency of the L-TSU. The access time is measured in CPU cycles. Here, it is assumed that the CPU clock frequency is the same as the FPGA implementation frequency.

Unit	Latency (Cycles)
D_TSU	
Acknowledgement Unit	3
Synchronization Unit (for one consumer)	7
Scheduling Unit	5
Network Interface Unit	Not measured
L_TSU	
AckQ Buffer	1
Firing Queue and Cache Prefetcher	11

Table 6: Latency of the DDM-TSU units

The number of cycles needed to process a single consumer, from the moment that the PU loads the *AckQ Buffer* with its continuation, until the PU reads its IFP from the Firing Queue is 27 cycles. This latency does not include the latency of the *Local Interconnect*. It should be noted, however, that the four units of the D-TSU as well as the L-TSU operate asynchronously from each other. Therefore, the maximum latency experienced is the latency of the *Cache Prefetcher,* which is 11 cycles.

In terms of the estimation for the power consumption, for the whole *DDM*-TSU as presented above, when running at 100MHz we obtain the following:

- Logic Power = 28.88 mW
- Signal Power = 3.31 mW
- Dynamic Power = 32.19 mW

Just to put these values into perspective, considering the same 100MHz operating frequency a Xilinx Microblaze processor with no cache has a dynamic power of 142mW, while an Intel Pentium processor has a dynamic power of 10.1 W.

2.4.3 Task Superscalar Pipeline (BSC)

The goal of our work is making a hardware design for the Task Superscalar architecture, prototyping it with a Hardware Description Language (HDL) and simulating it with a HDL simulator and finally synthesizing it in a Field Programmable Gate Array (FPGA) device, in order to create a real hardware prototype of the Task Superscalar hardware module.

We are currently implementing a basic hardware prototype described in VHDL and verifying its functionality using the Modelsim simulator. The pipeline receives tasks from a task generator and asynchronously decodes the task dependencies, generates the data dependency graph, and schedules tasks as they become ready. Ready tasks are sent to the execution backend, which consists of a ready queue, task scheduler, and a many-core fabric. Figure 8 illustrates the initial prototype, which is composed of one Pipeline gateway (GW), two Task Reservation Stations (TRS), one Object Renaming Tables (ORT) and one Object Versioning Tables (OVT).

The gateway is responsible for controlling the flow of tasks into the pipeline. It gets non-speculative tasks from task generator memory and allocates TRS space for them. It also sends operands of the allocated tasks to the different modules to data dependency analysis.

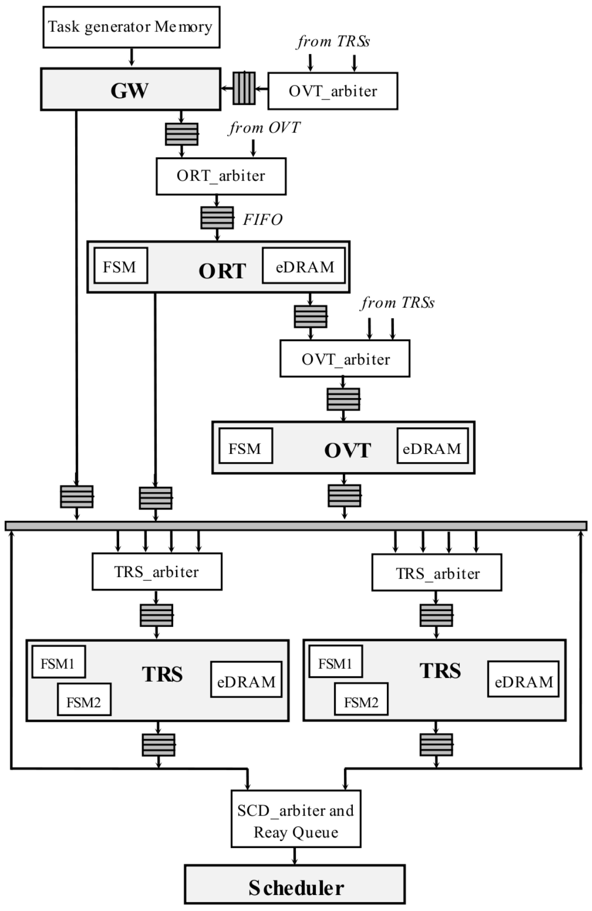

Figure 8: Block diagram of the first version of the hardware scheduler of Task Superscalar

TRSs store the in-flight task information in their eDRAM and track the readiness of task operands. As such, TRSs are effectively embedded with the data dependency graph. Inter-TRSs communication is used to register consumers with producers, and notify consumers when data is ready. Each TRS has 2 Finite State Machines (FSMs) one for selecting a TRS state according to the type of input packets, and the other is for saving in-flight task information, tracking the readiness of the operands, sending a task for execution, notifying the consumers of operands after a task being finished, and releasing a finished-task and its operands.

The ORT maps memory operands to the most recent task accessing the same memory object, and thereby detect object dependencies. The ORT has an 8-way associated eDRAM for saving the information of the operands. The OVT tracks live operand versions, created whenever a new data producer is decoded. The functionality of the OVT, therefore, resembles that of a physical register file, but only for maintaining operand meta-data. Effectively, the OVT manages data anti- and output-dependencies, either through operand renaming, or by chaining different inout operands and unblocking them in-order (sending a data ready message whenever a version is released). OVT has a memory for saving the versions of the operands. ORT and OVT has FSMs for processing input packets.

The blocks (GW, ORT, OVT and TRSs) are coded individually at the behavioral level of description. The modules are being debugged at the functional level. Each block has an independent clock signal and the blocks communicate with each other through FIFOs for sending and receiving different messages. At this point of the design, we use a memory for saving generated tasks instead of task generator thread only for simulating and testing the functionality of the pipeline. One main consideration in coding the design with VHDL is to judiciously allocate memory to the controllers, data transfer and registers in order minimize the consumption of FPGA memory resources.

Table 7 presents memory storage and access latency of the basic components. We use four-element FIFOs with latency of 2 cycles, one for activating read or write enables and the other for completing reading or writing operation. The arbiters are responsible for selecting one of the input messages based on round robin algorithm. The memory latencies described include one cycle for activating access enable (e.g., *write_enable* or *read_enable*) and one cycle for completing writing or reading operations. Additionally, allocating TRS memory and searching the ORT memory takes 5 cycles more, on average, than other operations.

Component	**Latency (cycles)**	**Required storage**	**Details of sizes**

FIFO	2	1Kbit	FIFO_ptr_size=2b
			FIFO_data=256b

SCD_arbiter and Ready Queue	2	5Kbit	FIFO_ptr_size=2b
			FIFO_data=1325b

Arbiter	2	1Kbit	

OVT_memory	2	256KB (2Mbit)	OVT_adrs_width=13b
			OVT_data_width=256b

ORT_memory	2 – 5	256KB (2Mbit)	ORT_n_way=8b
			ORT_set_size=2048b
			ORT_way_size=256 b
			ORT_adrs_width=64b
			ORT_index_size=10b
			ORT_tag_size=54b
			ORT_data_width=201b

TRS_memory	2 - 4	256KB (2Mbit)	For saving 512 tasks:
			TRS_adrs_width=13b
			TRS_data_width=256b
			block_size=16b
			block_number=512b

Trace generator memory	2	256KB(2Mbit)	For saving 512 tasks:
			TG_adrs_width=9b
			TG_data_width=4096b

Table 7: Latency and sizes of basic components of the design

2.5 Exploring simpler cores (UCY)

One of the goals of this project is to explore the use of simpler cores (e.g. Intel Atom cores). This allows having a larger number of cores for the same power budget. Since the code of the DF-threads is usually simple and relatively small, it is easier to explore parallelism with a larger number of single-issue in-order cores other than fewer superscalar out-of-order cores. In order to avoid the complexity of handling multi-object code version of the same source or dynamic binary translation, we have decided to explore the different cores but supporting the same ISA (x86). Chips with different cores that support the same ISA are known as asymmetric multi-core processors. For the estimate of the benefits of this approach we looked at the specifications for two state-of-the-art processors: a multi-core for a server and a single-core for an ultra-light mobile system. We looked at processors that used the same technology (45nm) and a similar operating frequency (1.6-1.7GHz). For the multi-core processor we found the AMD Opteron 6164HE (code-named “Magny-Cours”). This processor has a rated power consumption of 85W, which can be translated into approximately 7.083W per core. For the mobile processor we found the Intel Atom Z530. This processor has a rated power consumption of 2W. Consequently, by exchanging the complex Opteron processor with simpler Atom processors, for the same power budget we can have approximately 3.5x more cores. What we are exploring is the use of both types of processors and have either the compiler make a simple analysis of the requirements of a thread or having the user add hints to the program as to allow the runtime system to assign a thread to one or the other type of cores. The performance ration between complex and simpler cores will be also taken into account.

3 Conclusions (UCY)

In this document we presented the work performed within the context of WP6 and more specifically T6.3 – Advanced Architecture Definition. We presented the hardware models for the support of coarse- and fine-grain thread scheduling, transactions and fault-tolerance. Furthermore, we presented the specifications in terms of space, latency and power consumption for the thread scheduling modules. We also presented the TERAFLUX architecture template as well as a study on the benefits of replacing complex cores with simpler more efficient smaller cores.

For year three, WP6 will focus on two tasks T6.4 – Fine-tuned execution model – and T6.5 – Abstraction layer. Both tasks will run in parallel for year three and four. The first task will focus on the evolution of the execution model as to develop a more sophisticated scheduling of threads that allows for better fairness use of resources, reduction of hot spots or better power consumption, and reduction of the memory latency overheads by applying prefetching or thread migration techniques. The second task will focus on the development of an abstraction layer that has as its goal hide the complexities of the underlying architecture, achieve a better resource management and handle faulty devices.

REFERENCES

[1] http://www.hipeac.net/roadmap – Last visited in November 2011.

[2] Y. Zhibin, A. Righi, R. Giorgi, A Case Study on the Design Trade-off of a Thread Level Data Flow based Many-core Architecture. In: Future Computing 2011 – The third International Conference on Future Computational Technologies and Applications, pages 100-106, Rome, Italy, 2011. ISBN: 978-1-61208-154-0.

[3] J. D. Thompson, D. G. Higgins, and T. J. Gibson, CLUSTAL w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. In: NUCLEIC ACIDS RESEARCH, vol. 22, pp. 4673-4673. 1994.

[4] S. Arandi and P. Evripidou, "DDM-VMc: the data-driven multithreading virtual machine for the cell processor," In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC '11). ACM, New York, NY, USA, 25-34.

[5] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, P. Trancoso, "TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems," Parallel Processing, 2008. ICPP '08. 37th International Conference on , vol., no., pp.25-34, 9-12 Sept. 2008

[6] S. Weis, A. Garbade, J. Wolf, B. Fechner, A. Mendelson, R. Giorgi, T. Ungerer, A Fault Detection and Recovery Architecture for a Teradevice Data-Flow System, In: DFM-2011: Data-Flow Execution Models for Extreme Scale Computing, Oct. 2011, pp. 39-45.

[7] W. J. Dally, B. Towles, Route packets, not wires: on-chip interconnection networks. In Proceedings of the 38th Annual Design Automation Conference, Las Vegas, Nevada, USA. DAC'01. ACM, New York, NY, 684-689. doi=http://doi.acm.org/10.1145/378239.379048

[8] S. Arandi, G. Michael, C. Kyriacou, and P. Evripidou, "Combining Compile and Run-time Dependency Resolution in Data-Driven Multithreading", in Proceedings of the First Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM2011), October 2011

[9] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. Cao Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. Proceedings of the International Symposium on High Performance Computer Architecture, February 2007.

[10] Xuehai Qian, Wonsun Ahn, and Josep Torrellas. Scalablebulk: Scalable cache coherence for atomic blocks in a lazy environment. Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pages 447–458, 2010.

[11] Saša Tomić , Cristian Perfumo , Chinmay Kulkarni , Adrià Armejach , Adrián Cristal , Osman Unsal , Tim Harris , Mateo Valero, EazyHTM: eager-lazy hardware transactional memory, Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, December 12-16, 2009.

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

[13] International Technology Roadmap for Semiconductors 2009 Edition. Website. Available online at http://www.itrs.net, visited on November 24th 2011.

[14] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In Proceedings of the 2004 International Conference on Dependable Systems and Networks, pages 177-186, Washington, DC, USA, 2004. IEEE Computer Society.

[15] M. J. Flynn, Computer Architecture, 1995, Sudbury, Massachussets: Jones and Barlett Publishers.

[16] R. Giorgi, Z. Popovic, N. Puzovic, Implementing Fine/Medium Grained TLP Support in a Many-Core Architecture, In: Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2009, ISBN: 978-3-642-03137-3 Samos, Greece, 2 July 2009, pp. 78-87, doi=10.1007/978-3-642-03138-0.

[17] M. Steinhaus, R. Kolla, J. L. Larriba-Pey, T. Ungerer, and M. Valero, Transistor count and Chip-Space estimation of simulated Microprocessors, Research report UPC-DAC-2001-16, UPC Barcelona Spain, 2001.

Appendix 1 – TERAFLUX Architectural Properties

This appendix summarizes the properties of the TERAFLUX architecture the Consortium has chosen as a target. In particular these properties refer to the description given in the Section 2 of this document and to the Figure 1 reported in the Section 2. The properties are organized in three different levels (this was already defined in milestone M7.1 and extended during the rest of Year 1 and Year2, but we report it also here for formal delivery to the Commission):

- *Level 0 (L0.x)*: in this level the main architectural properties of each building block are described. Building blocks are all the main hardware functional units that are part of the TERAFLUX architecture (e.g., processor cores, network-on-chip, distributed thread scheduler, etc.);
- *Level 1 (L1.x)*: in this level the main organization of the building blocks is described. In particular, this level describes how the single building blocks are grouped and are interconnected each other in the overall architecture.
- *Level 2 (L2.x)*: gives a description of the architectural properties that provide a hardware support for the Data-Flow execution model.

In the following we give a detailed description of each level.

Level 0 – Building Blocks

- L0.0 – Asymmetric Processor Cores with the same x86-64 ISA (possibly including the T* extension): we can distinguish at least two types of cores and needed features or model to be supported:
    1. Service Cores: powerful cores for OS, I/O or ILP intensive codes (e.g., multi-threaded, multiple issue, out-of-order execution, etc.). These cores will support the execution of S-threads and L-threads (see deliverable D7.1);
    2. Auxiliary Cores: simpler cores for power efficient computations (e.g., single issue, in-order, etc.);
    3. ISA extensions to support TLP, TM and the selected memory model (see deliverable D7.1);
    4. Timing models for both service and auxiliary cores.
- L0.1 – Network-on-Chip (NoC):
    1. Topological connections of internal resources (e.g., memory blocks, cores, accelerators, etc.) within a Node (i.e., cluster of cores) and among different Nodes;
    2. Use of the state-of-the-art models and designs for implementing the NoC;

3. Timing models for the communications latencies among Nodes and among the internal resources of a Node;

4. The network-on-chip is split into two sub network infrastructures: (i) the inter-Node network can be based on a classical NoC design, and (ii) the intra-Node network has to be defined by the Node designer (local network).

- L0.2 – Memory Hierarchy:

    1. Adoption of a globally addressable physical space (Unified Address Space) to guarantee on-chip global accessibility when the system runs in supervisor mode. However, this unified address space is not directly accessible when the system runs in user mode;

    2. Adoption of a memory model that supports: thread local storage (TLS) areas, Data-Flow threads synchronization using Single Assignment Semantic (SAS) and Transactions;

    3. Use of the state-of-the-art models and designs for implementing of the single memory blocks and the memory hierarchy;

    4. Integration of both pre-fetching and DMA mechanisms;

    5. Timing models for measuring the latencies of accessing all the levels of the hierarchy;

    6. There is no support for hardware global coherency;

    7. Implementation of an explicit mechanism to “publish” data: (i) a signal to make visible the changes, and (ii) a signal to notify that the “publishing” phase is finished (from that all the changes are visible to all the other resources).

- L0.3 –Thread Scheduling Units (TSUs) and Fault Detection Units (FDUs):

    1. The two additional units are based on a distributed architecture that allows scalability and avoids a single-point-of-failure. From this point of view, they are organized in a hierarchy: 1-local element within each core (both for the TSUs and the FDUs) and 1-group element within each Node (both for the TSUs and the FDUs).

    2. The TSU are based on the following micro-architectural properties: (i) they are able to manage Data-Flow threads and their associated information and, (ii) they are able to manage additional information for the power consumption, temperature, availability and faultiness level of each core.

    3. Timing models for the TSUs and the FDUs.

- L0.P0 – Power and thermal models for all the building blocks.

Level 1 – Organization of the Building Blocks

- L1.0 – The TERAFLUX architecture is composed of the building blocks defined at level 0 organized in the following manner: a set of Nodes (i.e., clusters/groups of cores) each of them consisting of a number of auxiliary cores with a TSU Node-element and an FDU Node-element. The size and organization of the Nodes can be static or dynamic:

    1. Each Node may have access (not necessarily exclusive) to a service core for OS and I/O operations. This service core may be part of the Node;

    2. Fast access to a portion of the physical memory. From this point of view the communication infrastructure is implemented as part of the NoC or bus-based or a combination of the two;

    3. The Node can have a static organization (i.e., fixed composition of cores and hardware units) or a dynamic organization (i.e., the composition of the cores and functional units, as well as the size of the Node can be varied on the basis of the specific requirements and characteristics of the applications).

Level 2 – Architectural support for the execution model

L2.0 – Efficient hardware support for handling the different types of threads executed by the machine (e.g., DF1, DF1b, DF2, etc.) can be implemented as part of the TSUs.

L2.1 – Possible exploration of the specialization of the cores in order to meet the different requirements of the different types of threads.

L2.2 – Hardware support for a power and thermal management within each Node and across all the Nodes. It may be part of the TSUs.

L2.3 – Hardware support for re-execution of the threads in case of fault detection. It may be implemented within the TSUs and the FDUs.

L2.4 – Support for the virtualization by mapping virtual CPUs into physical cores.

L2.5 – For sake of simplicity of the initial implementation we postpone the management of page faults: in the initial versions, all the code and data are loaded in the memory.

L2.6 – Threads running on auxiliary cores can start I/O operations that will be served by a service core. It is worth recalling that I/O operations are not aware of the memory consistency.

L2.7 – Sequential consistency for memory operations of the single thread (i.e., memory operations are sequentially consistent).

L2.8 – A protection hardware support can be part of the architecture, but it is considered not mandatory.