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

PROJECT NUMBER: 249013

TERAFLUX

Exploiting dataflow parallelism in Teradevice Computing

D6.3 – Fine-tuned TERAFLUX Execution Model

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

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#	Author	Organization	Change History
0.1	Pedro Trancoso	UCY	Initial outline
1.0	Pedro Trancoso, Skevos Evripidou, George Matheou, George Michael, Andreas Diavastos, Constantinos Christofi	UCY	UCY parts
1.1	Pedro Trancoso	UCY	Added contributions from partners
1.2	Andreas Diavastos	UCY	Formatting into template
2.0	Pedro Trancoso	UCY	Revised
2.1	George Michael, Andreas Diavastos, Paraskevas Evripidou	UCY	Revised

Release Approval

Name	**Role**	**Date**
Pedro Trancoso	Originator	07.12.2012
Paraskevas Evripidou	WP Leader	12.12.2012
Roberto Giorgi	Coordinator	19.12.2012

TABLE OF CONTENT

Glossary

Auxiliary Core	A core typically used to help the computation (any other core than service cores) also referred as "TERAFLUX core"
BSD	BroadSword Document – In this context, a file that contains the SimNow machine description for a given Virtual Machine
CDG	Codelet Graph
Cluster	Group of cores (synonymous of NODE)
Codelet	Set of instructions
COTSon	Software framework provided under the MIT license by HP-Labs
DDM	Data-Driven Multithreading
DF-Thread	A TERAFLUX Data-Flow Thread
DF-Frame	the Frame memory associated to a Data-Flow thread
DVFS	Dynamic Voltage and Frequency Scaling
DTA	Decoupled Threaded Architecture
DTS	Distributed Thread Scheduler
D-TSU	Distributed Thread Scheduling Unit
Emulator	Tool capable of reproducing the Functional Behavior; synonymous in this context of Instruction Set Simulator (ISS)
D-FDU	Distributed Fault Detection Unit
L-Thread	Legacy Thread: a thread consisting of legacy code
L-FDU	Local Fault Detection Unit
L-TSU	Local Thread Scheduling Unit
MMS	Memory Model Support
NIU	Network Interface Unit
NoC	Network on Chip
Non-DF-Thread	An L-Thread or S-Thread
Node	Group of cores (synonymous of Cluster)
OWM	Owner Writeable Memory
OS	Operating System
Per-Node-Manager	A hardware unit including the DTS and the FDU
PhyGAS	Physical Global Address Space
PK	Pico Kernel
Sharable-Memory	Memory that respects the FM,OWM,TM semantics of the TERAFLUX Memory Model
S-Thread	System Thread: a thread dealing with OS services or I/O
StarSs	A programming model introduced by Barcelona Supercomputing Center
Service Core	A core typically used for running the OS, or services, or dedicated I/O or legacy code
Simulator	Emulator that includes timing information; synonymous in this context of "Timing Simulator"
TAAL	TERAFLUX Architecture Abstraction Layer
TBM	TERAFLUX Baseline Machine
TLPS	Thread-Level-Parallelism Support

TLS	Thread Local Storage
TM	Transactional Memory
TMS	Transactional Memory Support
TP	Threaded Procedure
UAS	Unified Address Space (a.k.a. PhyGAS, Physical Global Address Space)
Virtualizer	Synonymous of “Emulator”
VCPU	Virtual CPU or Virtual Core

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

Constantinos Christofi, Andreas Diavastos, George Michael, George Matheou, Pedro Trancoso, Paraskevas Evripidou
UCY

Marco Solinas, Alberto Scionti, Andrea Mondelli,
Ho Nam, Antonio Portero, Roberto Giorgi
UNISI

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

Fahimeh Yazdanpanah, Daniel Jiménez-González, Carlos Alvarez Martínez,
Yoav Etsion and Rosa M. Badia
BSC

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

Executive Summary (UCY)

This document describes the work that was performed during the third year (M25-36) of the TERAFLUX project within the context of Task 6.4 (M25-48) "Fine-Tuned Execution Model". Note that this task extends for one more year and thus the work here presented does not reflect the completed objectives for this task but limits itself to the achievements for the third year (M25-36). Notice also that Task 6.5 (M25-48) "Abstraction Layer" was also active during this year but the results of the work in that task are not reported for the first year and are instead described in D6.4 at the end of the project. Milestone M6.3 reports the advances regarding the work for Task 6.5 for the first year.

The work in Task 6.4 was a collaboration work by the different partners involved and thus we report here the contribution of each partner.

- UCY developed and evaluated a runtime dependency resolution mechanism for the DDM_style of the TERAFLUX execution model
- UCY implemented and tested the execution of a DDM-style application on a multi-node system
- UNISI designed and evaluated a multi-node TSU working thanks to the implementation of the T* ISE
- BSC presented the evolution of the TaskSs implementation
- UNIMAN showed the TM support for the TERAFLUX architecture
- UCY presented a program analysis tool based on PARAVER

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

1 Introduction (UCY)

The basic TERAFLUX execution model was presented in the first two years of the project. This model is based on the dataflow concepts, where dataflow is used as the policy for scheduling threads (collections of instructions). Transactions are added to the dataflow threads as a way to explore more parallelism and improve the programmability. Several different types of dataflow threads were defined, as well as the memory model. In addition, we have adopted a template for the architecture proposed within the WP6, which is depicted in Figure 1. More details on the execution model and architecture can be found in D6.1 and D6.2.

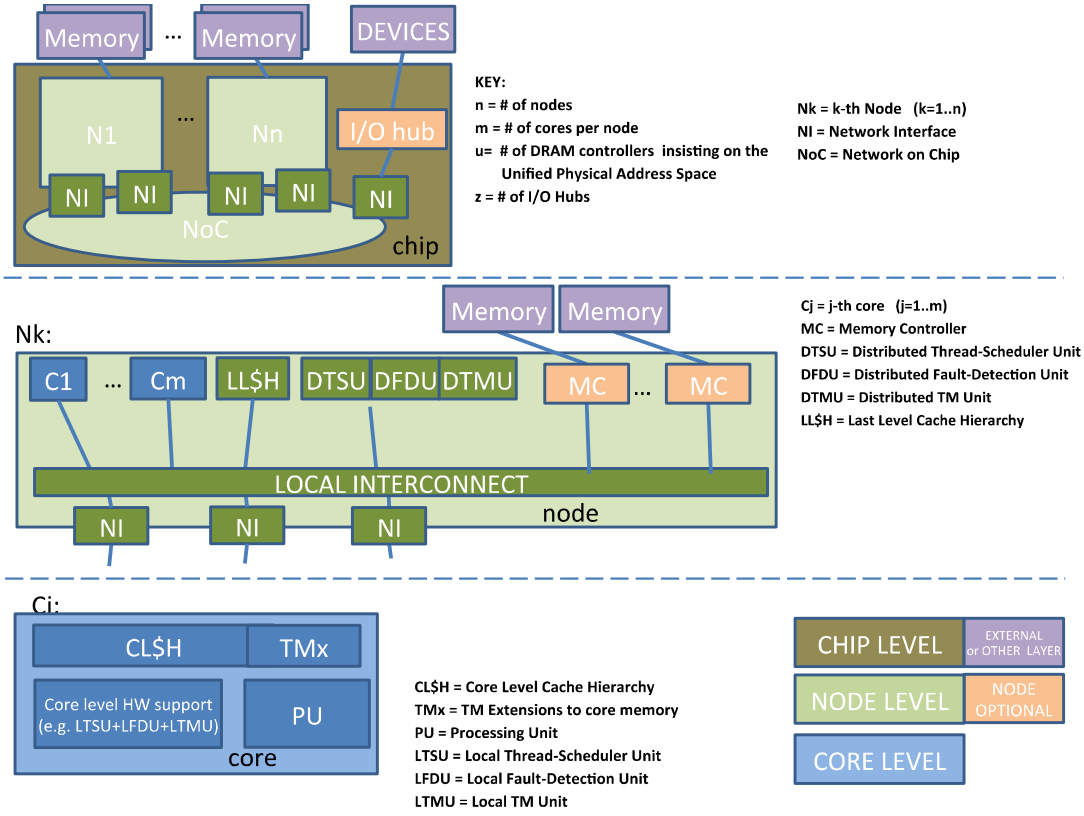

Figure 1: TERAFLUX Architecture Template

In year 3 of the TERAFLUX project, within the context of the architecture work package (WP6), most work was done towards exploring optimizations to the basic model and the implementation and development of the hardware modules. This work was done as part of Task 6.4 (M25-48) “Fine-Tuned Execution Model”.

This year, the partners have proposed ways to extend the model as to allow for the efficient execution across different nodes of multi-cores. This required extensions to the D-TSU, which are reported in this deliverable. The implementation of the memory model proposed in previous years and the extension of the T* instruction set have also been part of the efforts performed during this year.

While it is well known that the dataflow model is able to exploit the maximum available parallelism, making it efficient is a challenge. This is especially true for execution models that depend on the static definition of the dependencies. For this analysis programmers are many times faced with the task of

identifying the dependencies among threads. In some cases this might not be possible as dependencies may only be determined at runtime. Last year we have developed and tested the use of I-structures at the Node level. During this year we have experimented with an efficient mechanism to extend the execution for distributed systems. An alternative approach is to allow the use of dynamic dependence through the TSCHEDULE instruction as done in the T* approach (c.f. D7.1, D6.2), widely adopted and described in D4.6, D5.3 and D7.4.

In terms of hardware modules to support the execution model, in addition to the Thread scheduling modules for the support of DTA- and DDM-style dataflow threads, which are reported in D7.4, during this year there was a special effort in developing the modules for support of coarse grain threads (the TaskSs module) and transactions (TM module). The former allows for the system to explore dynamically coarse-grain dataflow threads as a combined or alternate model to the fine-grain DTA- and DDM-style dataflow threads. The latter helps in the support of the efficient execution of transactions for exploring the access to shared modifiable variables within dataflow threads.

Lastly, the successful execution of a parallel application depends also on the careful analysis of its execution and overcoming eventual bottlenecks in either the application or the runtime support for the proposed model. This year we have adopted an existing tool for the analysis of the execution of TERAFLUX applications. With this tool it is possible to analyze the status of the different queues in the runtime and the time spent in different routines of both the application and runtime. This analysis helps in tuning the runtime and also determining bottlenecks in the application.

Overall, the work presented in this document reflects the work in improving the execution model proposed in the previous two years. This work was performed within Task 6.4 but it is relevant to note again that this task spawns for one more year and thus we still expect the evolution of the model to continue for the next year of the project. This upcoming evolution will be presented at the end of next year as an annex to this deliverable.

1.1 Document structure

This year we focused on fine tuning the execution model. The main work is on the optimizations and support for the execution model. This is presented in Section 2. The first optimization presented is the work on exploiting the runtime dependency resolution. Then we present the support for the model to allow the distributed execution of applications, the support for the memory model, the T* instruction execution, the coarse-grain execution under TaskSs, and the support for Transactional Memory (TM). The development of a program analysis tool is presented in Section 3. Section 4 presents the conclusions and future work, i.e. the work that is within the context of Task 6.4 but that will be completed within the next year.

1.2 Relation to other deliverables

The work here presented is based on the material presented in previous deliverables such as D6.1 where we presented the basic TERAFLUX architecture and model, and D6.2 where we presented the advanced TERAFLUX architecture. In addition, aspects of the evolution of the model are closely related to the material presented other deliverables presented this year. In D5.3 we have the aspects related to the programming model and in particular the addition of transactions to the dataflow model. That work results in the TM support module reported in this deliverable. Another evolution to the model is reported in D5.3 regarding the double execution of dataflow threads for the fault-tolerance

support. Finally, the modules reported in this deliverable are implemented into the COTSon platform and this effort is reported in D7.4. Some initial architectural decision was reported in D7.1.

1.3 Previous activities referred by this deliverable

The work reported in this deliverable is based on the work performed in Task 6.1 (Basic execution model) where we proposed the dataflow model for the TERAFLUX project and in Tasks 6.2 and 6.3 (Basic and Advanced architecture definition) where we proposed the architecture for the TERAFLUX many-core processor. In this deliverable we present optimizations and implementation issues related to the execution model as to exploit the characteristics of the architecture proposed before.

1.4 Activities referred by this deliverable

The work performed in TERAFLUX in the context of WP6 in year three (M25-36) was according to its two active tasks:

- Task 6.4 (M25-48) “Fine-Tuned Execution Model” and
- Task 6.5 (M25-48) “Abstraction Layer”.

This deliverable focuses on reporting the work from Task 6.4. Notice that this task extends for one more year and thus the work here presented does not reflect the completed objectives for this task but limits itself to the achievements for the third year (M25-36). The work performed for Task 6.5 is not reported in this deliverable but instead will be reported by the end of year four in deliverable D6.4.

2 Execution Model Support and Optimization

2.1 Runtime Dependency Resolution (UCY)

In this Section we describe the lightweight runtime dependency resolution mechanism using I-structures. This mechanism was developed for the DDM-style execution model.

2.1.1 I-Structures

An *I-Structure* [1] is a type of storage controller that obeys the single-assignment semantics, i.e. each element is written only once but can be read multiple times. If a read request arrives for a data element that has not been written yet, the controller defers the read until the write arrives. This property of *I-Structures* provides the synchronization needed for exploiting producer-consumer parallelism without the risk of read-write races. The basic idea is to add status bits to the storage cells and a queue for holding deferred reads. Using the *I-Structure* idea is possible to discover the dataflow producer-consumer dependencies at runtime.

The state of each element of the *I-structure* is depicted in the state diagram in Figure 2.

and can be one of the bellow:

- *Absent*: nothing has been written into the element yet and no attempt has been made to read it. A write operation is allowed. This is the entry state.
- *Present*: the element can be read but not written.
- *Waiting*: nothing has been written into the element yet, but at least one read request was attempted (deferred read). When this cell is written all the deferred reads must be satisfied.
- *Error*: a second write to the same element is not allowed thus it results in an error in the execution.

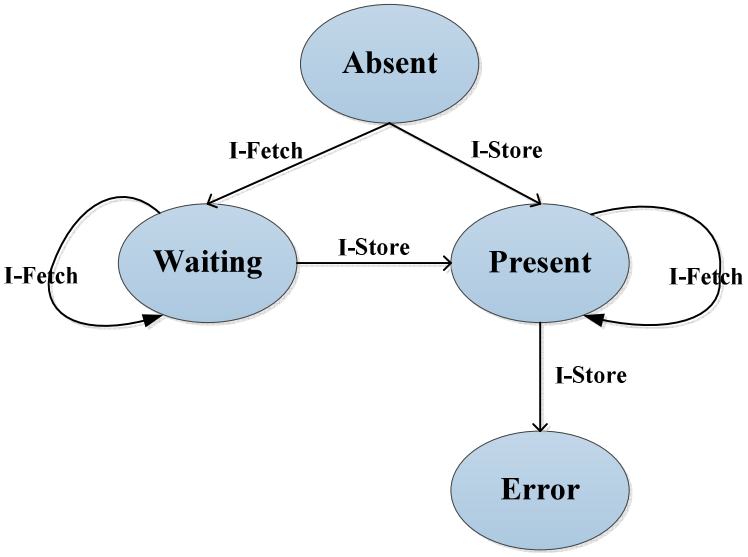

Figure 2: State transition diagram for I-Structure Elements. A read operation referred to as an I-Fetch while a write operation as an I-Store

2.1.2 Runtime Dependency Resolution

A typical DDM-style program is statically partitioned into a number of DF-threads. The producer consumer dependencies among the threads are also discovered statically. A problem arises when part or all of the threads perform read operation(s) on data items whose address is not known at compile time, i.e. it is resolved at runtime. In Figure 3 sketches our approach for dealing with runtime dependency. We apply the following algorithm for each Data-structure that has runtime dependencies:

- For every thread *t* that performs at least one read operation on data items which address is resolved at run-time a *proxy* thread *t'* is introduced.
    - Thread *t'* replaces *t* in all the Consumer Lists of its explicit producer threads, i.e. threads that are identified as its producers at compile-time.
- The RC (Ready Count) of *t'* is set to the number of explicit producers of *t*.
    - The RC of *t* is set to the number of read operations performed on data items with addresses resolved at runtime.
    - For every such read a special I-structure Fetch (I-Fetch) request is issued by *t'* with the parameters *<address of the data to read, identifier of t, contexts of t>*.
    - When the I-structure receives an I-Fetch request it checks whether the data has been stored in that specific address
        - if the data is present a request is sent to the TSU to decrement the RC of thread *t*,
        - if the data is not present, a request is added into a pending list inside the I-Structure.
- For every thread producing data that for is potentially read by *t*, a special I-structure Store (I-Store) request is issued

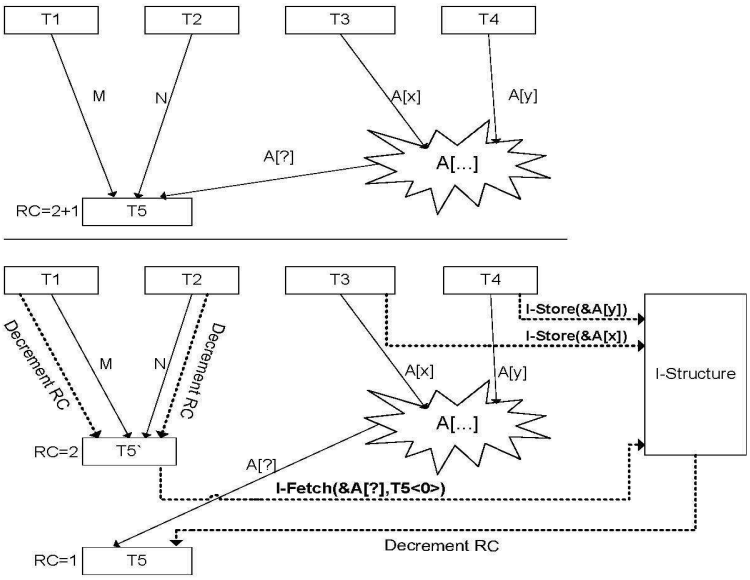

Figure 3: The top part of the figure depicts a Threaded graph with a run time dependency A[?]. The bottom part illustrates our I-Structure approach for DDM

2.1.3 Evaluation

In order to study the effect of the performance overhead of the I-Structure operations we compare three versions that run on a single node. The first version (C-D) utilizes the Compile-time

Dependency resolution. The second version (RC-D) combines both approaches, i.e. part of the dependencies are resolved at Runtime (using I-Fetch and I-Store operations) and the rest are resolved at Compile-time The third version (R-D) utilizes the Runtime approach for resolving the dependencies.

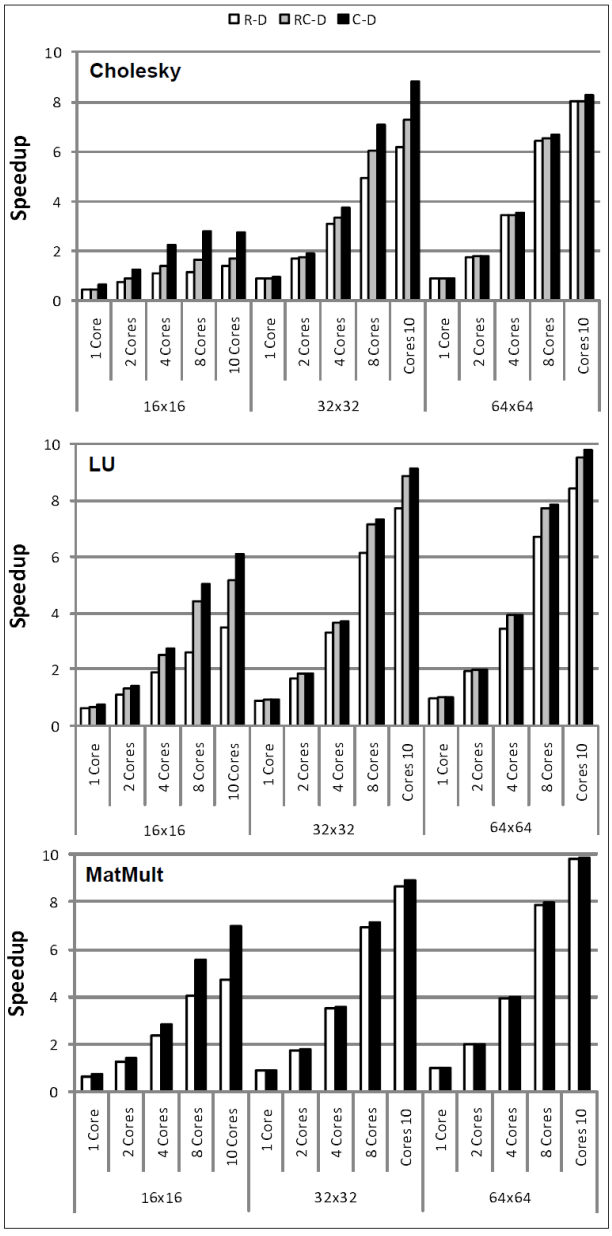

Figure 5: Speedup comparison: runtime-determined dependencies (R-D) vs. runtime & compile-time determined dependencies (RC-D) vs. compile-time determined dependencies (C-D) approaches

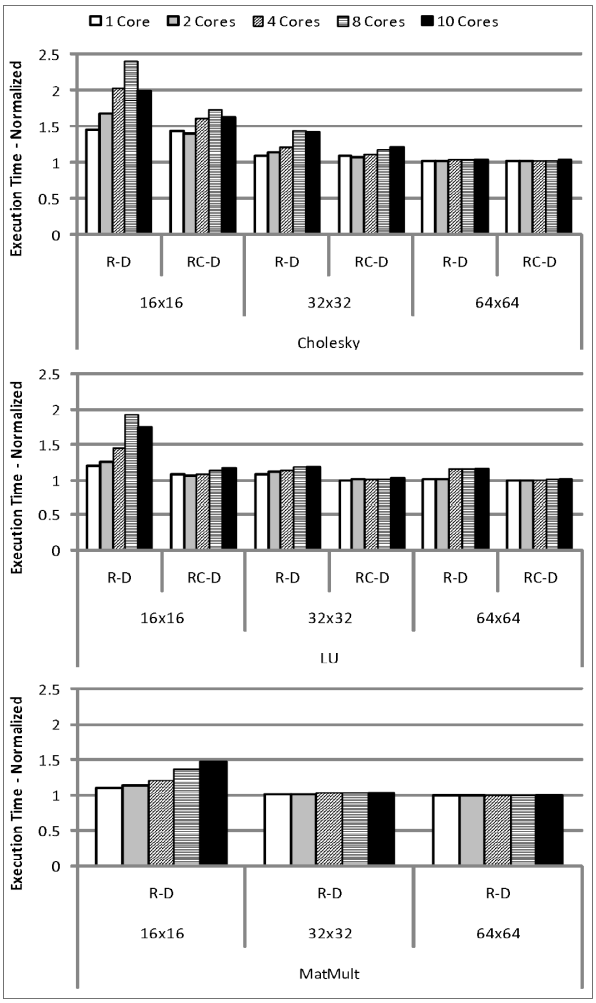

Figure 4: Execution time comparison: execution time using the runtime-determined dependencies approach vs. the runtime-determined & compile-time determined dependencies approach normalized to the execution time using the compile-time determined dependencies approaches

A quick description of the applications and their sample code is the following:

- Cholesky: blocked Cholesky decomposition (SMPSs code examples http://www.bsc.es/plantillaH.php?cat_id=425)

- LU: blocked LU decomposition (CellSs code examples http://www.bsc.es/plantillaH.php?cat_id=421)
- MatMult: blocked matrix multiplication

Figure 5 depicts the comparison of the performance of the three versions for various thread sizes (16x16, 32x32 and 64x64 array decompositions) with input size for these examples two 4096x4096 matrices. The execution time reported is from the native execution of the application using the DDM-style model implementation of the TSU at user-level, on a 12-core machine composed of two 6-core AMD Opteron processors with 32GB of RAM.

The results demonstrate that, as expected, the best performance is delivered by the version utilizing the compile-time approach (C-D), followed by the version utilizing the combination of the compile-time and runtime approaches (RC-D). Note that for the MatMult benchmark, only the first and third versions are available, as the threads in this program have only one data dependency.

Figure 4 illustrates the execution time of the R-D and RC-D versions normalized to the execution time of the C-D version. The results demonstrate that, as we increase the granularity of the threads by increasing the size of the blocks the threads operate on, the total number of blocks decreases and so does the total number of thread invocations. Consequently the number of I-Fetch and I-Store operations decreases, thus reducing the overheads. Moreover, increasing the granularity of the threads amortizes the I-Structure operations overheads.

The results of these executions give us the motive to apply I-Structure to distributed systems as we observed good performance on single node execution.

2.2 Multi-node Execution (UCY)

The original implementation of the DDM-style execution model was for single Muli-core Node. In the context of Task 6.4 we have extended this to support the execution of applications across multiple Nodes.

The inherent tolerance for latencies of the execution model allows extending the execution across multiple distributed nodes with lesser overhead than other approaches. This is achieved by tolerating inter-node latencies resulting from data and synchronization communications with the execution of threads.

To facilitate the distributed multi-node execution:

1) We developed the Network Interface Unit (NIU), which is a software extension module for the DDM-style runtime that allows the execution across an off-chip network. Each node is an independent multi-core machine running one D-TSU as many L-TSUs as the number of cores, on a conventional OS.
2) We made changes to existing TSU data structures and created new ones to support the new functionality.
3) We implemented a distributed memory mechanism that allows us to share data throughout a set of nodes running a TSU using Global Addressing Space.

4) We utilized the Dijkstra–Scholten termination detection algorithm.

Distributed DDM-style programs are fundamentally the same as single-node ones except for:

(1) the distribution of data across Nodes at startup and during execution, and
(2) the gathering of data post-execution.

The DDM-style runtime is extended to handle remote memory accesses, resulting from producer and consumer threads residing on different nodes. This is handled by forwarding the data to the node where the consumer is scheduled to run. DDM-threads enforce the single assignment semantics for data exchanged among threads. Thus, traditional coherence-management operations are not required for data movements among the nodes. For the distribution of threads across the cores of the system nodes we employ a static scheme, in which the mapping is determined at compile time and does not change during the execution. This simplifies the scheduling and data management tasks and, in the presence of an accurate knowledge of the threads execution loads, can lead to a very efficient and balanced parallel execution. It is important to note that a static distribution only specifies where the thread will be scheduled once its ready, however, when the thread is ready is decided based on data-availability.

The DDM-style runtime adopts a distributed organization consisting of multiple D-TSU units (one per node) communicating across the network to coordinate the overall DDM execution, as shown in the figure 6.

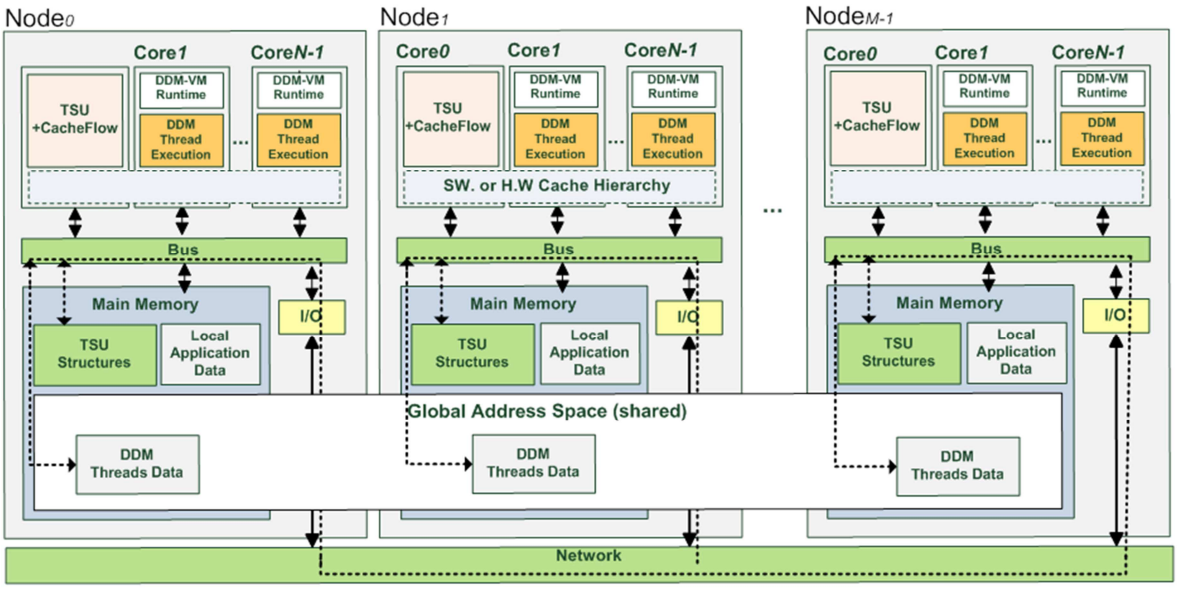

Figure 6: The Distributed DDM-style Architecture

Comparing to the original DDM-style architecture reported in the previous deliverables, for the support of the distributed execution we made some changes to a few TSU structures:

- On the Graph Memory (GM), for each node, we only load the meta-data of the threads that are expected to execute on that node.
- The Synchronization Memory (SM) requires extra attention as the allocation of SM entries of a thread is directly influenced by the assigned scheduling policy. If a thread is assigned to a core using a deterministic policy based on information that is static and shared to all nodes,

the TSU has enough information to handle the allocation for the SM implementation. On the other hand, if the thread is assigned a custom policy written by the application programmer, the SM implementation needs to be re-examined to avoid excessive redundant allocations. In the current implementation we allocate the entire range of the SM entries on each node, regardless which threads will execute on each node. In the future we need to improve this to reduce the amount of memory used by the TSU and build a mechanism that allocates SM entries in regards to the scheduling policy.

The rest of the TSU structures remain unchanged. However, we have added two new structures to support distributed execution:

- The Distributed Acknowledgement Queue (AQ): This queue holds the decrement RC requests coming from the TSUs on the remote nodes.
- Forward Table (FT): This table holds the address and size of the data that will be forwarded to remote nodes.

When a thread finishes its execution and its consumer is on the same node, an entry is inserted in the AQ of the local node D-TSU. If the consumer belongs to a remote node, a message containing the invocation (ThreadId, context) is sent to the remote node D-TSU.

To support this communication a new software module is added to the TSU: The Network Interface Unit (NIU). The NIU is a software module that relies on the underlying network hardware interface. We developed our own connectivity layer using non-blocking TCP sockets. The NIU is responsible for managing the network initialization, establishing connections with the other nodes in the system and providing communication services to the TSUs during the execution. The NIU also supports distributing/gathering data across the global address space in the system at startup and post-execution of the DDM-style program. Termination is detected using the Dijkstra–Scholten distributed termination algorithm. The current NIU implementation uses the available Ethernet network for the exchange of messages as it is used in the user-level DDM-style runtime. With the TSU integration with COTSon, the NIU will take advantage of the existing interconnection network for faster exchange of messages.

2.2.1 Unified Address Space

A Unified Address Space (UAS) is supported across all the nodes in the system. We employ data forwarding, in which the data produced by a thread is forwarded to the node where the consumer is scheduled to run. The distribution of program data in the UAS across the nodes occurs at startup and the gathering of the results after the program execution ends. An address referring to the UAS consists of the ordered pair (node id, local address). The first component refers to the node identifier and the second refers to a conventional main memory address on that node. Coherence-management operations are not required between the nodes. The mapping of the program data into the UAS depends on the assignment of the program threads. The data of a certain invocations is mapped to the part of the UAS belonging to the node where this invocation is scheduled to run. The movement of data between producers and consumers running on different nodes during the execution is managed automatically by the DDM-style runtime without the intervention of the programmer.

2.2.2 Evaluation

For the evaluation of the DDM-style distributed system we used 5 different applications. A quick description of these applications and their sample code is the following:

- Cholesky: blocked Cholesky decomposition (SMPSs code examples http://www.bsc.es/plantillaH.php?cat_id=425)
- CONV2D: two-dimensional convolution, where a mask is applied to a 2D image (based on the code described in http://www.stanford.edu/group/sequoia/cgi-bin/node/185 and found the Sequoia SDK examples)
- IDCT: inverse discrete cosine transform (based on the IDCT kernel in the mpeg library code similar to one found in http://www.irisa.fr/master/COURS/CAPS/CoursCD/HTML/Codes/ExercicesScap/exercice9/idct.c)
- mult: blocked matrix multiplication
- trapez: trapezoidal rule of integration

For the evaluation we used two input data sets for most applications (Cholesky, CONV2D, Mult) and their sizes were 4096x4096 and 8192x8192. For IDCT we used 8192x8192 and 16384x16384 due to the small execution time of the application. For Trapez we used 657M and 1250M steps.

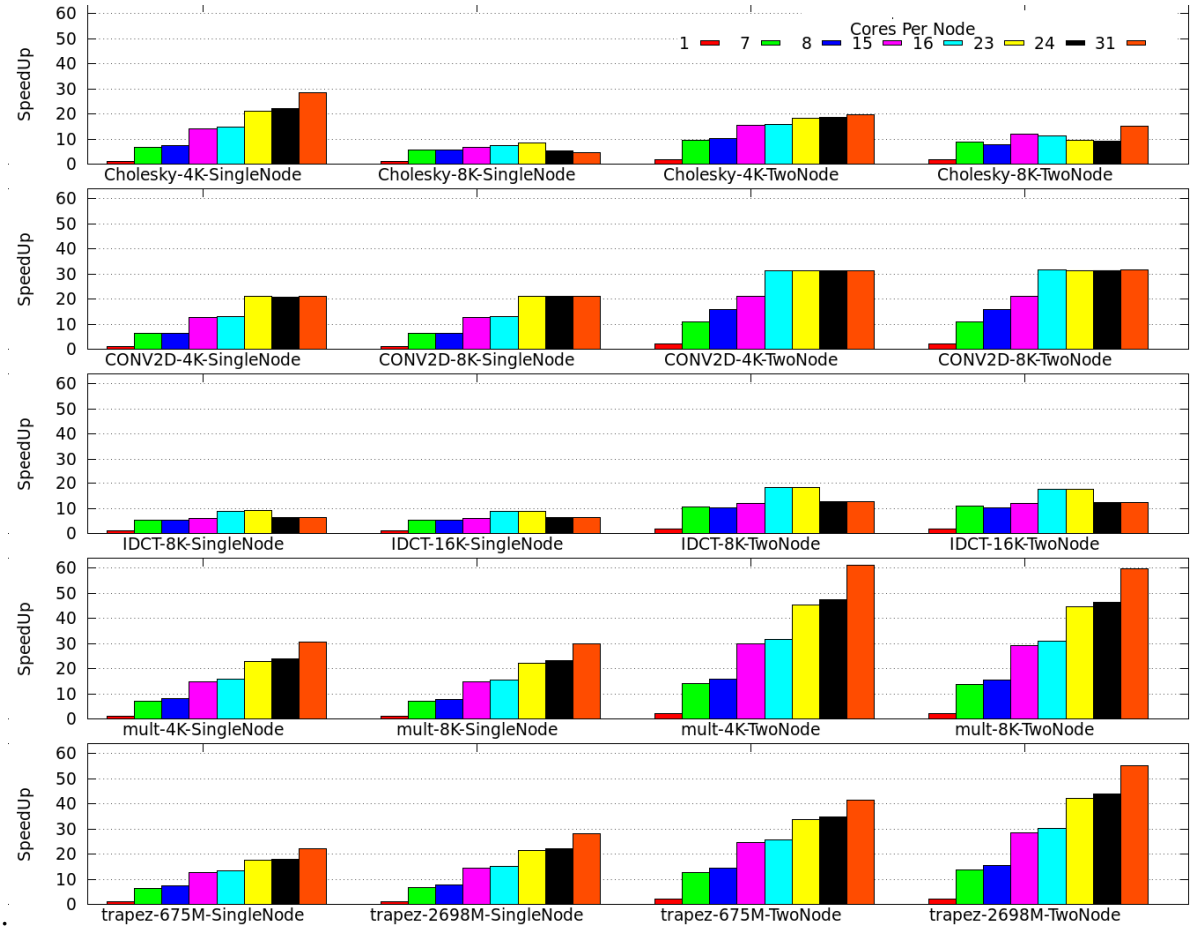

Figure 7: Speedup comparison: first two columns present the comparison between identical configurations with different input size. Following two columns present the comparison of the previous executions using a second identical node through an Ethernet network

In Figure 7 we present the speedup obtained for the execution of four applications, Cholesky, CONV2D, IDCT, mult, and Trapez, implemented on the DDM-style model using the user-level

software TSU implementation. The speedup reported is compared to the sequential execution of the same application. The execution was done natively on two 32-core machines, each composed of four 8-core AMD Opteron processors and 48GB of RAM. We can see that increasing the problem size does not necessarily give us better performance as applications like Cholesky that have "bottlenecks" in certain parts of the algorithm which result in performance degradation. Also, we can see that doubling the resources by using a second system always achieves better results than the single node execution even considering the unavoidable network delays. We can observe that the performance, for most applications, is not improved linearly but instead by a smaller rate. Matrix Multiply and Trapez integration scale almost linearly due to the large parallelism available in their algorithms.

2.2.3 Runtime Dependency Resolution on Distributed Systems

To support runtime dependency resolution on distributed systems we employ the same approach to the UAS for the distribution the I-Structure entries across all the nodes in the system. When the I-structure receives an I-Fetch for data that belongs to a remote node, the I-Fetch request is forwarded to the appropriate node. We employ data forwarding, in which the data produced by a thread is forwarded to the I-structure of the node where they were distributed at compile time. If data belongs to the I-Structure of the current node, then the data is forwarded to all nodes that issued an I-Fetch for that data.

2.3 Fine-tuning the TERAFLUX Frame Memory for 1024 cores (UNISI)

Within the Teraflux project, a general template for the architecture of the envisioned Data-Flow target machine has been defined (see deliverables D6.1 [2], D6.2 [3] and milestone M7.1 [4]). The template sets basic features that specific architectural implementations must have in order to efficiently execute Data-Flow threads. As detailed in deliverable D6.1 and D6.2, the target machine architecture is organized hierarchically into nodes, each of them clustering a set of cores, the last levels of cache memory, the TSU, the FDU, and one or more memory controllers. All the resources are connected through a dedicated interconnection system (e.g., a NoC), both at the intra-node level and at the inter-node level. A picture of the reference architecture can be found in D6.2 (Figure 1 of page 12), and is reproduced in Figure 7 for completeness.

With the aim of completely distributing both the thread scheduling and fault detection activities, the thread scheduler and the fault detection elements are split into a node-level unit, and a core-level unit (hereafter we will refer to the thread scheduler as the Distributed Thread Scheduler - DTS). The DTS is responsible for all the activities concerning the thread life-cycle. In particular it is responsible for allocating frames, keeping the list of waiting threads, keep the list of executable threads, and distributing executable threads among the available cores. All these activities are strictly related to the adopted memory model, supporting the thread execution model ([5,6,7,8]).

In the following, we describe a first attempt to define a fine-tuned architectural support for executing Data-Flow threads based on T* ISA extension (T* is an extension of the x86-64 Instruction Set Architecture [9,10,11]). UNISI started evaluating the implementation of frame memory to manage DF-threads. As first test we verified that it is possible to execute the very same programs (e.g., Fibonacci and Matrix Multiply) both on: (i) single node machine with 32 cores, and (ii) multi-nodes machine with a total of 1024 cores (see also figure 7 and figure 8).

Examples

With the aim of developing and fine-tuning the frame memory support to DF-threads execution in a DTA-style, UNISI considered an implementation of the architectural template presented in D6.1 ([2]) and D6.2 ([3]). In particular, frame memory support for DF-threads has been implemented within the common simulation infrastructure (i.e., the COTSon simulator), and evaluated using two different simple benchmark applications: Fibonacci and Matrix Multiply. The purpose of these experiments is: (i) to demonstrate that the frame memory structure envisioned within the TERAFLUX consortium correctly works, and (ii) that the system supporting frame memory can scale almost linearly with the increase of the number of available cores (see also Figure 8 and Figure 9).

Fibonacci (single- and multi-node)

The Fibonacci benchmark application recursively computes the Fibonacci integer sequence, given the integer number for which the computation has to be performed. The benchmark exploits T* ISA extensions.

The advantage of a T* like architecture is demonstrated by the fact that we are now able to distribute the computation of the same T* binary not only on the cores within a single node but also among the cores of several nodes. UNISI evaluated the benchmark both in the case of a 32 cores single-node TERAFLUX machine (i.e. following the template architecture described in D6.2, Section 2.1 – [3]), and in the case of a multi-nodes (with 32 cores per node) TERAFLUX machine with a total of 1024

cores. Figure 8 shows the results obtained running the Recursive Fibonacci benchmark calculating the 40th element of the Fibonacci sequence. Left and right graphs in Figure 8 shows thespeedup, when considering respectively the single-node and multi-nodes simulated machine. As shown in the two graphs, the system performs with an almost linear scaling, demonstrating the correctness of the frame memory implementation and the ability too seamlessly distribute the computation across several nodes.

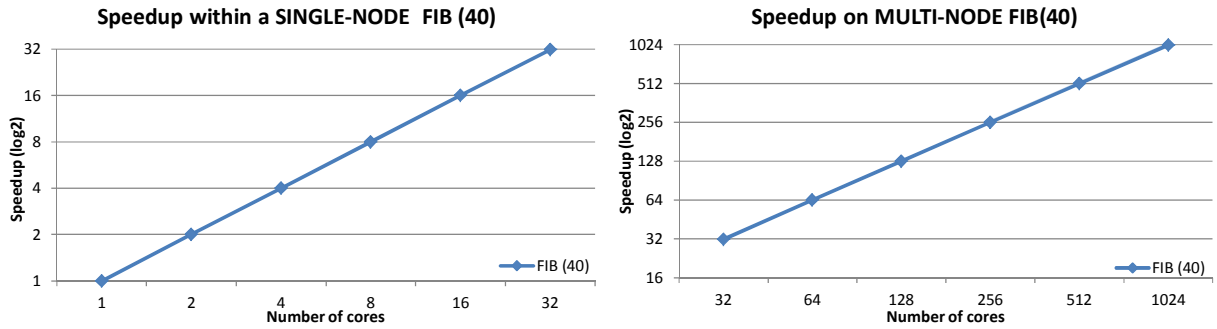

Figure 8: Speedup expressed on a log2 scale for the Recursive Fibonacci benchmark, computing the 40th element of the Fibonacci sequence. The T* ISA extension allows the benchmark to scale almost perfectly both in the single-node and in the multi-node

Matrix Multiply (single- and multi-node)

Similarly to the previous case, Matrix Multiply allowed UNISI to evaluate the behaviour of a single- and multi- nodes machine with the support for the frame memory. The benchmark computes the multiplication of two square matrixes of the same size. UNISI has tested the benchmark with matrices of different sizes. The Matrix Multiply benchmark has been generated in the executable form, which exploits T* ISA extensions ([9,10,11]). Moreover, UNISI evaluated the benchmark both in the case of a 32 cores single-node TERAFLUX machine, and in the case of a multi-nodes TERAFLUX machine with a total of 1024 cores and 32-cores per node. Similarly to the previous example, Figure 9 shows the results obtained running the benchmark application computing the matrix multiplication for two square matrices with a size of 512x512. Left and right graphs in Figure 9 show the speedup, when considering respectively the single-node and multi-nodes simulated machine. As shown in the two graphs, again, the system performs with an almost linear scaling, demonstrating the correctness of the frame memory implementation and the advantage of T*.

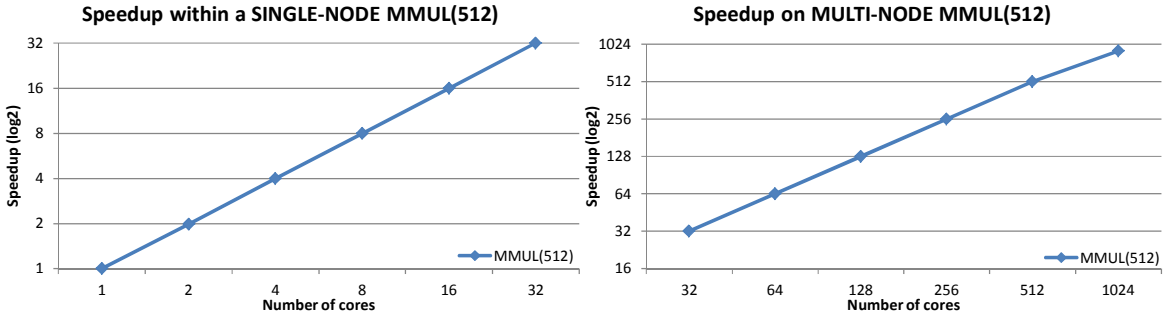

Figure 9: Speedup expressed on log2 scale for the Matrix Multiply benchmark, using input matrices with a size of 512x512. The T* ISA extension allows the benchmark to scale almost perfectly both in the single-node and in the multi-node case

2.4 Instruction execution (UNISI)

The architecture implementation directly supports the life-cycle of DF-threads, by means of the T* instruction set extension (T* ISA) [9,10,11]. Currently the ISA extension accounts for four instructions that are suited for allocating and deallocating frames during the initial and the final phases of DF-threads' life (namely TSCHEDULE and TDESTROY, described in Section 3.3.1), and for accessing the frame memory during the DF-threads execution (namely TREAD and TWRITE, described in Section 3.3.2). From the architectural viewpoint, the management of frames accesses can be seen as mediated by the Memory Management Unit (MMU), whose functions are extended in order to correctly addressing the frame memory (the exact mechanisms for managing the memory virtualization are currently under research).

2.4.1 DF-threads life cycle (TSCHEDULE, TDESTROY)

The main Distributed Thread Scheduler (DTS) internal structures in the L-TSUs are (the area cost has been evaluated in the second year, D6.2, section 2.4): the Waiting List/Table (WL or **WT**) and the Pre-Load Queue (PQ or **PLQ**).

Assuming that each core has the capability of managing the information of n_F frames: the WT holds up to n_F DF-Thread continuations (i.e., a tuple <IP,FP,SC, …>, IP=Instruction Pointer, FP=Frame Pointer, SC=Synchronization Count), whose input frame is waiting for coming data; the PLQ queues up to n_F DF-Thread continuations (in this case SC==0, so that field is not present), which are globally ready to be executed.

Please note that the DTS still retains the possibility to assign the DF-threads as it retains productive (i.e. respecting the power/ temperature/performance/resilience envelope) until the last moment.

The main Distributed Thread Scheduler (DTS) internal structures in the D-TSUs are (as evaluated in the second year, D6.2, section 2.4): the Free Frame Table (**FFT**) and the Pending Tschedule Request Queue (**PTQ**) are in the D-TSU.

Assuming that each node contains m cores: the FFT holds m entries that, at any time and for each core, track the Free Frames Number (FFN); the PTQ queues up to n_{PTQ} tuples consisting of the ID of the Core (CID) that issued a TSCHEDULE operation that cannot be served locally to such core or immediately (i.e., <CID, IP, SC> or the CID and the TSCHEDULE parameters).

The Core Record (CR) is another per-core internal structure that maintains constantly updated information such as: Power, Temperature, Faultiness. This structure could be placed either at Core Level (as initially designed in D6.2), Node Level or globally.

Table 1: Description of TSCHEDULE and TDESTROY T* instructions

T* instruction	Description
TSCHEDULE	1. At the execution stage in the pipeline of a Processing Unit, a frame request is sent to the L-TSU (indicating also other associated info such as <TSCHEDULE, IP, SC>); 2. The request is sent to the node's D-TSU, via the local interconnect; 3. The requests from any core arriving to such D-TSU are queued in the PTQ ; 4. The D-TSU sees the node availability of frames through the FFT: after a FFT lookup the

	frame request is served preferably in the requesting node or wherever there is frame availability (in this step an assignment policy can guide the decision); 5. The D-TSU structures (FFT, PTQ) are updated; 6. The L-TSU structures (WT, PLQ) are updated; 7. The requesting core receives the FP to the available frame and the instruction execution terminates; Note: some implementation trick (virtual FPs or TIDs) could be used to make the above operations faster.
TDESTROY	1. At the execution stage in the pipeline of a Processing Unit, a frame deallocation (and DF-thread completion request) is sent to the L-TSU (indicating also other associated info such as <TDESTROY, FP>); 2. The request is sent to the node's D-TSU, via the local interconnect; 3. The L-TSU structures (WT, PLQ) are updated; 4. The D-TSU structure (FFT) is updated and the instruction execution terminates.

2.4.2 DF-threads frame memory interaction (TREAD, TWRITE)

Similarly to the previous sub-section, the following Table 2 shows the details about the interaction of the processing unit, the L-TSU, and the D-TSU during the execution of the TREAD and TWRITE instructions. The former allows a DF-thread to read data from its assigned frame, while the latter is used to write data in the frame of consumer(s) DF-thread. The interaction implements the architectural template described in D6.2, section 2.4 [3].

Table 2: Description of TREAD and TWRITE T* instructions

T* instruction	Description
TREAD	1. At the execution stage in the pipeline of a Processing Unit, the PU sends a <TREAD, FP, Offset> request to the L-TSU; 2. The L-TSU returns the value read from the frame memory pointed by FP, at the specified offset.
TWRITE	1. At the execution stage in the pipeline of a PU, a <TWRITE, FP, Offset, Data> request is sent to the L-TSU; 2. The write operation can proceed along the memory hierarchy as any other write operation; 3. The SC associated to the DF-Thread's Frame (that is detected by the effective address being contained within such Frame address range) should be decremented: 4. The core (either local or remote) that detects a write belonging to the Frames that it manages will be responsible to update the Frame's SC.

2.5 TaskSs (BSC)

Objectives

The goal of this work is to implement a functional prototype of the Task Superscalar design presented by Etsion et al. [12,13]. To achieve this goal a simple but functional implementation of the hardware design has been selected. This prototype is shown in Figure 9. As it can be seen the Task Superscalar frontend employs a tiled design, and is managed by an asynchronous point-to-point protocol. It is composed of four different modules: Pipeline gateway (GW), task reservation stations (TRS), object renaming tables (ORT), and object versioning tables (OVT).

The GW is responsible for pushing the flow of tasks into the pipeline including: allocating TRS space for new tasks, distributing tasks to the different modules, and blocking task generating threads whenever the pipeline fills.

TRSs store the meta-data of the in-flight tasks and, for each task, track the readiness of its parameters. To do this, TRSs maintain the data dependency graph, communicating to each other in order to relate consumers to producers and notify consumers when data is ready.

The ORTs match memory parameters to the most recent task accessing them, and thereby detect object dependencies. Furthermore, each ORT has exactly one OVT associated with it. The OVT tracks all the live versions of every parameter the associated ORT stores. That helps TRSs to maintain the data dependency graph. The functionality of the OVTs (and their associated ORT) therefore resembles that of a physical register file, but only to maintain meta-data of parameters. Effectively, the OVT also manages anti- and output-dependencies, either through parameter renaming, or by chaining different bidirectional (inout) parameters and unblocking them in-order.

The implemented prototype is composed by the minimum number of modules that maintain the prototype functional and the Task Scheduler has been replaced by a Ready Queue (RQ) that will dispatch tasks following a simple Round-Robin protocol.

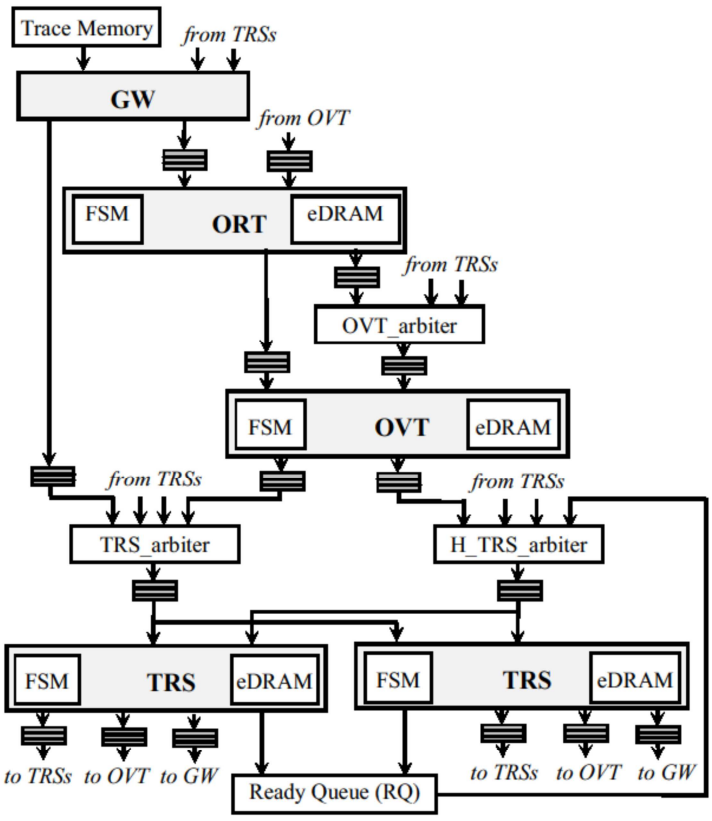

Figure 10: Implemented prototype of the Task Superscalar architecture

Current Status

Currently all the modules that compose the prototype are completely functional when simulated and isolated from the rest of the system. Also, all of them and their corresponding memories have been synthesized targeting commercial devices. To accomplish this goal the modules' design and VHDL code have been modified to accomplish the constraints imposed by such devices (as available free memory or registers). The next step is connecting all the modules and simulating the resulting system to verify its correct behavior. After that, the whole system will be synthesized and its performance will be measured.

The improvements made this year are the following:

- The functional simulation of all the modules has been finished.
- The network has been optimized to decrease the number of packets and the number of connections needed between modules, while maintaining the same functionality.

- All the memories have been synthesized "standalone" targeting a commercial FPGA device. Their design (HDL code) has been tuned to better suit the constraints of real FPGA devices.

- All modules have been synthesized "standalone" targeting a commercial FPGA device.

- The Finite State Machines (FSM) of all the modules have been optimized in order to shorten the number of cycles needed to process the packets as shown in Table 1.

Results

At this point we can present the first results obtained by the hardware implementation of the prototype. Table 3 shows the processing time that the Finite State Machine (FSM) of every module needs to process an incoming packet.

Table 3: Latencies of processing the packets.

Packet	Processing latency	Responsible Unit	Size (bits)
ContIssue	2 cycles	GW	24
Create Version	• 2 cycles if first time (input or output) • 3 cycles if *Not* first time and input • 4 cycles if *Not* first time and output	OVT	240
DataReady	2 cycles	TRS	64
DropParam	2 cycles	OVT	40
DropVersion	• 2 cycles for deleting the last version • 3 cycles for deleting the last version and the previous one	ORT	120
Finish	4 cycles + 2 additional cycles for loading each parameter	TRS	88
Forming Execute packet	1 cycle (for loading meta-data of a task) + 2 cycles for loading each parameter + 1 cycle for sending the task to the ready queue	TRS	176+(#param*75)
IssueAck	2 cycles	GW	24
Issue	• 5 cycles for tasks without any parameter • 3 cycles for tasks with at least one parameter + 2 additional cycles for each parameter (using ParamTRS (200 bits) or DirectParam (96 bits) packets)	TRS	160

ParamORT	2 cycles	ORT	168

Also we have results from synthesis of each module isolated from the rest of the system (but including their local memories). Table 4 shows a selection of these results as maximum working frequency, slice logic utilization including the number of slice flip flops (FFs, the number of one-bit registers used in the whole FPGA) and slice look-up tables (LUTs) that are used as logic or distributed RAM or even as shift registers. Macro Statistics include the number of required resources such as registers (R), comparators (C), multiplexers (M), adders/subtracters (A), tri-state buffers (T) and xor gates (X). Finally, the table presents the % of use that every module represents in each target FPGA (the Xilinx xc7vh290t that has the smallest number of LUTs among the available devices of its family and the Xilinx xc7v2000t that is the biggest one in the family [14]). The table also includes the results for the FIFOs and Arbiters needed to implement the network connections and an estimation (total) of how many resources will use the whole system once all the modules are connected.

Table 4: Synthesis result of main modules (R:registers, M:multiplexer, A:adder/subtractor, C:comparator, T:tristate buffers, X:xor).

Component	Freq. (MHz)	#Slice LUTs	#Slice FFs	Macro Statistics	Usage of xc7vh290t	Usage of xc7v2000t
GW	389.63	9756	600	455R,4A,198M	4.4%	0.8%
TRS	142.93	39771	4800	1970R,9A,2C,945M, 497X	18%	3.2%
OVT	297.40	26843	2917	919R,1A,266M	12.3%	2.2%
ORT	201.57	28067	13654	8856R,21C,1137M, 8427X	13%	2.3%
FIFOs	498.32	8190	4770	108R,54A,90M	3.7%	0.6%
Arbiters	703.60	3102	2397	768R,771M,768T	1.4%	0.25%
Total	142.93	115729	29138	13094R,3407M,68A, 268T,8924X	53%	9.5%

2.6 TM Support (UNIMAN)

Transactional Memory (TM) attempts to simplify concurrent programming by allowing a group of load and store operations to execute in an atomic way. It is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing. TM systems can exist in hardware, software or as hybrid implementations. This discussion is related to the hardware implementation of TM systems.

TM hardware must perform a number of tasks.

Transaction modifications are isolated from the rest of the system until commit time through data versioning.

The system detects and resolves read-set and write-set conflicts.

Transaction commits appear to occur atomically.

In case of a conflict leading to an aborted transaction, a consistent state is reached after rewinding.

Transactional mechanisms are being designed while keeping following requirements under consideration.

Performance should be achievable without an undue burden on the programmer.

The system should scale gracefully to a large size with large amounts of concurrency.

The system should be able to cope with, and if possible exploit, a hierarchical organization of cores into nodes (clusters).

Our research at Manchester University aims to answer following questions regarding TM hardware system.

Is it better to exchange information about sharing between transactions as they go along or to do so only at the commit time?

How can we leverage the node (clustered) architecture to provide good performance for transactions?

Is it useful to have different local and global mechanisms?

What sharing patterns exist across a broad range of workloads?

What is the best balance between communication, storage and false sharing?

We have developed a TM implementation on COTSon that performs lazy version management and lazy conflict detection. The initial implementation is similar to Stanford TCC implementation [15].

The programmer only marks TM-Begin() and TM-End() regions and the underlying system ensures that the transactions run concurrently while maintaining Atomicity, Consistency and Isolation properties.

This model has been extended and a scalable version of the system has been developed. The scalable system is also a purely lazy implementation but the commit process takes advantage of the hierarchical organization of cores into nodes (clusters). The committed changes are broadcast within the node (cluster) but outside the node the invalidations are sent only to the nodes that were actually sharing the committed data. In order to implement the scalable TM system we have used a directory based cache coherence protocol as a starting point for our baseline system. The directories are extended with information about sharing of transactional data across nodes. The implementation is available on [18], as described in D7.4.

Figure 11 shows the diagram of the scalable model that we are evaluating.

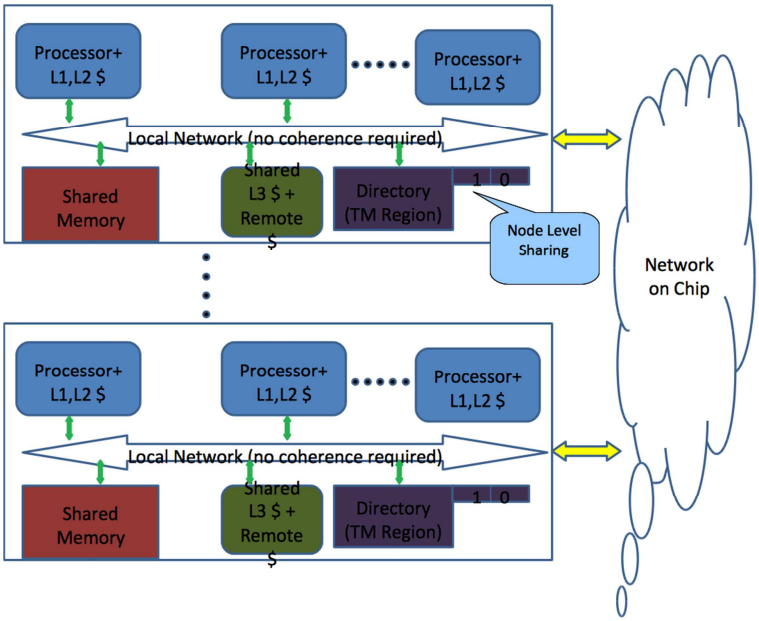

Figure 11: Clusters with extensions to cache and directory to support Transactional Memory

In this system each core has its own private L1 and L2 cache. L1 is write-through while L2 is a write back cache. Within each node (cluster) there is a shared L3+remote cache, a directory and part of distributed memory.

The directory tracks transactional memory regions and maintains information at cache line granularity. Each directory entry contains a bit vector to represent the sharers. Sharers are maintained at the cluster level.

L1 and L2 caches are used to maintain data versioning. During commit, a transaction first occupies all the directories in its R/W-set and marks all the cache lines in its write-set. The occupy and mark process is similar to Scalable TCC [15]. After completing the occupy phase the transaction locks the L3 controller and then writes back all its modified TM lines to the shared L3. After writing back all its data, the transaction unlocks the L3 controller and then sends commit messages to all the directories in its write set so that they can send the relevant invalidations. The write back is required so that L3 contains the most up to date copy and can respond to any requests to the cluster.

There are many optimizations possible to our initial implementation. For example using bloom filters to reduce the size of our read/write-set and to lock directories at lower level of granularity [16].

We have developed a functional model of purely lazy TM support into SimNow (c.f. D7.4). This had to be done through a special mechanism, since COTSon interfaces with SimNow in a manner such that functional modifications to execution are not possible. This included interacting with AMD as well as our partners at HP, with AMD extending some interfaces into SimNow to allow the implementation. A more sophisticated model for timing transaction commits is currently being developed [18].

For the purpose of evaluation, we have run two of the STAMP benchmarks on the scalable TM hardware. KMeans implements a clustering algorithm. This has a parallel phase to assign objects to clusters, and a serial phase to recalculate cluster centres. Vacation simulates a travel booking database for cars, hotels and flights. Each table (cars, hotels, flights) is represented as a transactional hash table. Reservations as performed as transactions, which change the availability of each booked item in the database.

Figure 12. shows the performance results of these benchmarks. Both benchmarks are available in [18].

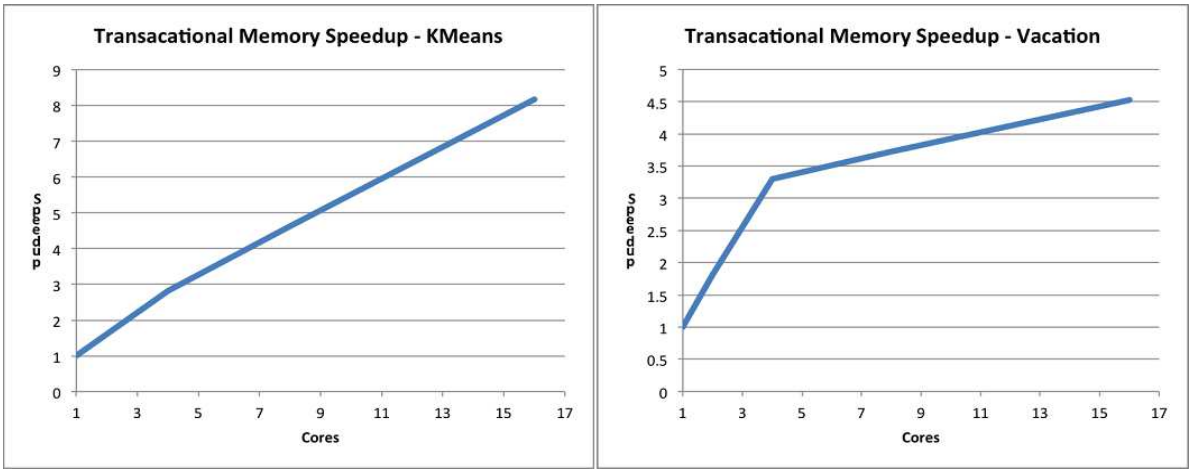

Figure 12: Performance results for Kmeans and Vacation Benchmarks

3 Program Analysis Tool (UCY)

Paraver [17] offers a graphical representation useful in performance analysis of parallel applications. It supports a detailed qualitative and quantitative analysis of the performance bottlenecks of parallel applications. Using the Paraver tool the programmer has the ability to optimize the performance of the application by focusing on the main performance bottlenecks and to minimize the hardware requirements. Paraver imports trace files of the application that are produced by the EXTRAE tool. In general, Paraver offers detailed quantitative analysis of application performance, simultaneous comparative analysis of multiple trace files and quick analysis of large traces files. Originally, Paraver supports MPI, OpenMP, Pthreads, CUDA, and OmpSs applications.

Through the Paraver tool the user can monitor the degree of parallelism of applications, the timeline of the processor during execution, the evolutionary process of changing the values of a specific (preselected) variable, monitor the load balance of different parallel loops and the instruction/cycle ratio of each thread. The Paraver tool can also be used for simultaneous analysis of different trace files that can help the programmer compare different versions of the same code, compare the behavior of the application on two different machines, find the differences between two different executions of the same application and finally determine the impact of the application problem size on the execution.

For our work, we were interested in adapting this tool to the DDM-style runtime system. To produce performance results for a DDM-style application we embedded function calls to the EXTRAE API inside the application code. Passing then the application with the EXTRAE tool we generate the trace file for the application that will be used as input to the Paraver tool to produce performance analysis results for the original DDM-style application.

In an attempt to analyze the performance of the TSU, we embedded EXTRAE API calls in the most important functions of the TSU.

The calls added to the code are: (a) by the preprocessor tool and (b) to a version of the runtime system. An example of such calls is presented in Figure 13 bellow.

```
int main(int argc, char **argv)
{
      Extrae_init();
{
      myIP = (char *) malloc(sizeof(char)*24);
      myName = (char *) malloc(sizeof(char)*24);
       ◆
       ◆
       ◆
       ◆
      free(A);
      free(B);
      free(C);

      Extrae_fini();

      return 1 ;
}
```

Figure 13: Extrae calls added to DDM-style program to monitor events by Paraver

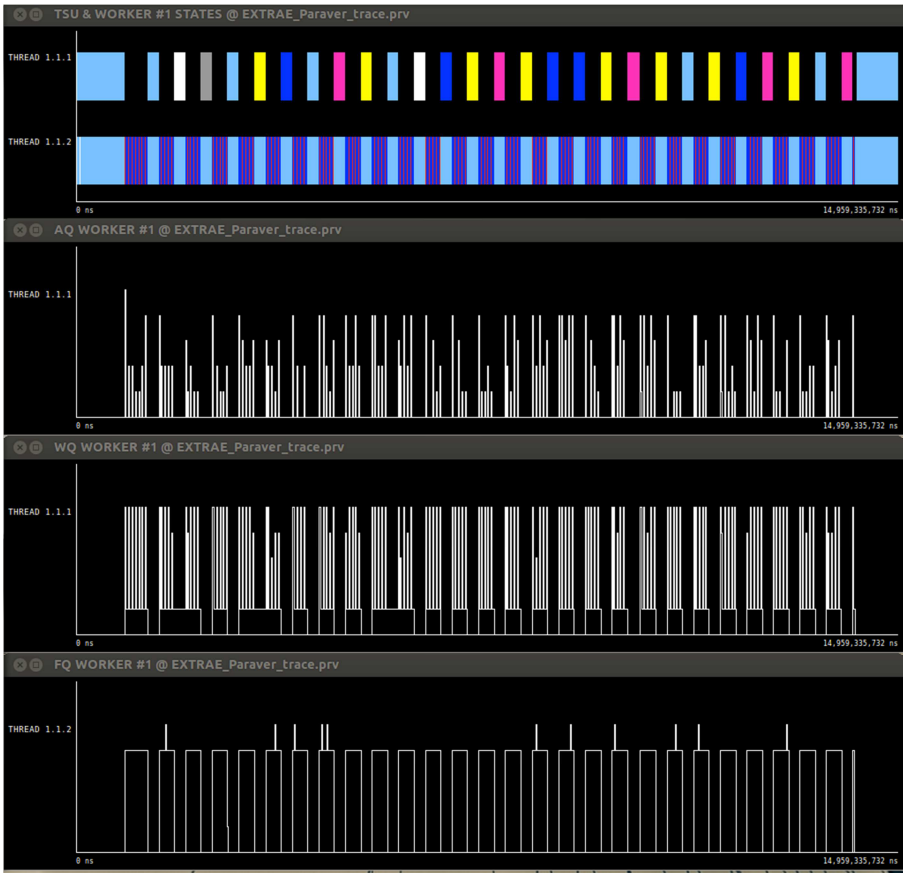

Figure 14: TSU performance analysis when executing MULT

The use of such a tool is very important to the tuning of the model as it is possible to observe the bottlenecks of the execution in the different runtime functions and also the use of the most relevant data structures.

In Figure 14 we present an example of the TSU performance analysis when running the MULT application. The results show the sizes of the Acknowledgement Queue (AQ), the Waiting Queue (WQ) and the Fire Queue (FQ). The first chart shows in color code the actions which the TSU and the application were performing during the execution time (Some of the important colors for the first line, which represents the TSU, are following: White - Decrement Consumers, Yellow – Moving threads from WQ to FQ, Pink – Checking the sizes of the queues). The second chart shows the size of the AQ, the third the size of the WQ and the last one shows the size of the FQ.

On the first chart we can see in cyan the idle time of the worker, which is the time spent until the TSU assigns work to the specific worker. If we follow a hypothetical vertical line to the next charts we can get a further understanding of the reasons for this fact. In the second chart, during the idle time we see the TSU serving AQ tasks and thus decreasing the remaining data in the queue. In the third chart we have the WQ which gets more entries as the AQ is being served, we can clearly see that there is a threshold were the TSU stops adding records. On the last chart we have the FQ where we can see again that the TSU assigns jobs to be executed for the current worker using again a threshold. From this information it is possible to infer that the threads being executed are too small for the processing speed of the TSU and that is the reason for the idle periods where the worker has no work to fetch from the FQ.

4 Conclusions and Future Work (UCY)

In this document we have presented the achievements for the third year of the TERAFLUX project regarding Task 6.4. We have included extensions and optimizations of the model, description of modules to support the model, and analysis tools. The contributions reported are from the different partners that participate in this Task.

We described techniques to improve the execution of applications such as the dynamic resolution of the dependencies. We also presented initial results from T* by executing an unmodified binary also on multiple nodes, and finally the Transactional Memory support. We closed with the presentation of a tool that allows for the monitoring of runtime execution and data structure utilization as to identify potential bottlenecks and performance issues in applications.

As Task 6.4 extends for one more year, in the next year we will finalize some of the techniques mentioned in this report. In particular we plan to continue the work on prefetching, the use of scratchpad memories, explore more comprehensive scheduling policies, and the memory model. More details will also be explored concerning the implementation of the hardware modules such as their timing and power consumption.

References

[1] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-structures: data structures for parallel computing. ACM Trans. Program. Lang. Syst. 11, 4 (October 1989), 598-632.

[2] S. Arandi, P. Trancoso, P. Evripidou, R. Mameesh, R. Giorgi, I. Watson, Y. Etsion, T. Ungerer, *Basic TERAFLUX Architecture and Basic Execution Model (internal use)*, TERAFLUX Deliverable – D6.1.

[3] S. Arandi, C. Kyriacou, G. Michael, G. Matheou, N. Masrujeh, P. Trancoso, P. Evripidou, R. Giorgi, Z. Yu, S. Collange, A. Scionti, B. Khan, S. Khan, M. Lujan, I. Watson, Y. Etsion, T. Ungerer, B. Fechner, A. Garbade, S. Weis, *Advanced TERAFLUX Architecture*, TERAFLUX Deliverable – D6.2.

[4] R. Giorgi, R. Mameesh, Y. Etsion, N. Navarro, R. Badia, M. Valero, F. Bodin, P. Faraboschi, A. Cohen, A. Mendelson, D. Shamia, S. Yehia, S. Girbal, P. Bonnot, A. Garbade, S. Weiss, T. Ungerer, S. Evripidou, P. Trancoso, I. Watson, M. Lujan, C. Kirkham, *List of architecture requirements*, TERAFLUX Milestone – M7.1.

[5] R. Giorgi, A. Scionti, A. Portero, P. Faraboschi, *Architectural Simulation in the Kilo-core Era*, Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012), poster presentation, London, UK, March 3–7 2012, ISBN 978-1-4503-0759-8.

[6] A. Portero, A. Scionti, Z. Yu, P. Faraboschi, C. Concatto, L. Carro, A. Garbade, S. Weis, T. Ungerer, R. Giorgi, *Simulating the Future kilo-x86-64 core Processors and their Infrastructure*, 2012 Spring Simulation Multiconference (SpringSim'12), March 26 - 29, 2012; Orlando, FL, USA.

[7] H. Nam, A. Portero, A. Scionti, R. Giorgi, *A Novel Architecture and Simulation for Executing Decoupled Threads in Future 1-Kilo-Core Chip*, ACACES Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems. July 11, 2012. p.83-86. Academia Press, Ghent (Belgium) ISBN 978-90-382-19.

[8] A. Portero, A. Scionti, M. Solinas, H. Nam, R. Giorgi, *Simulation infrastructure for the next kilo x86-64 Chips*, ACACES Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems. July 11, 2012. p.87-90. Academia Press, Ghent (Belgium) ISBN 978-90-382-19.

[9] A. Portero, Z. Yu, R. Giorgi, *T-Star (T*): An x86-64 ISA Extension to support thread execution on many cores*, HiPEAC ACACES-2011, ISBN:978 90 382 17987, Fiuggi, Italy, July 2011, pp. 277-280.

[10] R. Giorgi, *TERAFLUX: Exploiting Dataflow Parallelism in Teradevices*, ACM International Conference on Computing Frontiers - 2012 (CF'12), May 15–17, 2012, Cagliari, Italy. ACM 978-1-4503-1215-8/12/05.

[11] R. Giorgi, A. Portero, C. Concatto, R. Mameesh, Z. Yu, Y. Etsion, L. Villanova, N. Navarro, R. Badia, M. Valero, P. Faraboschi, A. Cohen, D. Shamia, A. Mendelson, A. Garbade, S. Weiss, T. Ungerer, P. Trancoso, S. Evripidou, B. Khan, S. Khan, M. Lujan, C. Kirkham, I. Watson, *Definition of ISA extensions and custom devices and External COTSon API extensions,* TERAFLUX Deliverable – D7.2.

[12] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero. Task Superscalar: An out-of-order task pipeline. pages 89-100, 2010.

[13] Y. Etsion, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero. Task

Superscalar: Using processors as functional units. 2010.

[14] http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/index.htm, visited December 2012.

[15] 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.

[16] 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.

[17] Barcelona Supercomputing Center. Paraver: Performance Analysis Tool: Details and Intelligence. http://www.bsc.es/computer-sciences/performance-tools/paraver

[18] http://cotson.svn.sourceforge.net/viewvc/cotson/