Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

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

PROJECT NUMBER: 249013

TERA^{F}LUX

Exploiting dataflow parallelism in Teradevice Computing

D7.5 – Final Report and Documentation

D8.3 – Final Results from the combination of UD and TERAFLUX dataflow techniques

---

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

Start date of the project: January 1st, 2010   Duration: 51 months

Lead contractor for the deliverable: UNISI

Revision: See file name in document footer.

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

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 1 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Change Control

Version#	Date	Author	Organization	Change History
1	05.03.2014	**Alberto Scionti**	**UNISI**	**Initial document**
2	13.03.2014	**Alberto Scionti**	**UNISI**	**First draft**
3-13	14.05.2014	**Detailed author list is presented in a next page**	**ALL PARTNERS**	**Final Draft**
14-16	17.05.2014	**Roberto Giorgi**	**UNISI**	**Review**

Release Approval

Name	Role	Date
Alberto Scionti	**Originator**	**15.05.2014**
Roberto Giorgi	**WP Leader**	**17.05.2014**
Roberto Giorgi	**Project Coordinator for formal deliverable**	**18.05.2014**

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 2 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

TABLE OF CONTENTS

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 3 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 4 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 5 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD
and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 6 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

LIST OF FIGURES

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 7 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

LIST OF TABLES

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD
and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 8 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

List of contributors to the writing of the document.

Alberto Scionti, Haileyesus Kifle, Somnath Mazumdar, Roberto Giorgi
University of Siena

Nacho Navarro, Rosa Badia, Mateo Valero
Barcelona Supercomputing Center

Sebastian Weis, Theo Ungerer
Universitaet Augsburg

Pedro Trancoso, Skevos Evripidou, Giorgos Matheou
University of Cyprus

Amit Fuchs, Yaron Weinsberg
Microsoft Research and Development

Paolo Faraboschi
Hewlett Packard Española

Feng Li, Albert Cohen
INRIA

Mikel Lujan, Behram Khan
The University of Manchester

Stéphane Zuckerman, Jaime Arteaga, Guang Gao
University of Delaware

Laurent Morin
CAPS

Sylvain Girbal
THALES

Deliverable number: **D7.5 – D8.3**

Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

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 (the whole set of D-TSUs and L-TSUs)
D-FDU	Distributed Fault Detection Unit (per-node FDU, also L2-FDU)
D-TSU	Distributed Thread Scheduling Unit (per-node TSU, also L2-TSU)
Emulator	Tool capable of reproducing the functional behavior; synonymous in this context of Instruction Set Simulator (ISS)
ISA	Instruction Set (Architecture)
ISE	Instruction Set Extension
L-Thread	Legacy Thread: a thread consisting of legacy code
L-FDU	Local Fault Detection Unit (per-core FDU, also L1-FDU)
L-TSU	Local Thread Scheduling Unit (per-core TSU, also L1-TSU, or LSU)
MMS	Memory Model Support
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
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 (later renamed T*)
TBM	TERAFLUX Baseline Machine (the initial instance of the TERAFLUX machine)
TLPS	Thread-Level-Parallelism Support
TLS	Thread Local Storage
TM	Transactional Memory
TMS	Transactional Memory Support
TP	Threaded Procedure
Virtualizer	Synonymous with "Emulator"
VCPU	Virtual CPU or Virtual Core

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 10 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Executive Summary

This deliverable reports on the research carried out in the context of DoW - Tasks T7.1, T8.2, and T8.3. The goal is to provide documentation on the TERAFLUX simulation infrastructure (based on HP COTSon) in order to provide a unique reference for the first time and advanced users of the COTSon simulator.

To this purpose, the document provides a short "getting started" section and continues with an overview of the main features, such as the architecture, the virtualization layer (i.e., the SimNow component), timers, samplers, and interleavers. All the steps are detailed with the precise command and the expected outputs. In particular, all the metrics that can be gathered from the simulator and the storage structures (e.g., the database integrated in the simulator, log files) are presented.

With the aim of helping the user to run simulations quickly, in the document a set of simple examples are presented. These examples cover all the different characteristics of the simulator, such as the capability of running only functional simulation, the use of samplers, and the simulation of multi-node architectures. Starting from this base of knowledge, an advanced user can easily start to extend the simulation platform, in order to simulate and analyze the behavior of user-defined hardware and software components. Following this direction, this manual also presents a full set of "TERAFLUX examples", one from each partner, where different advanced aspects related to TERAFLUX research (e.g., definition of new images, integration of hardware component, etc.) are reported. These examples represent also a description of the integration activity, through the COTSon simulation platform, of the research of the TERAFLUX partners, as progressed during the project. The research example provided by UD also serves as the content of deliverable D8.3. The example illustrates the main progresses obtained from the integration of the UD run-time and the TERAFLUX platform.

From this premise, we can conclude that this document completes the series of deliverables for WP7 and WP8, and it's written at this time as the experience on using the tool has matured enough. As previously mentioned, we included also several advanced examples (see sections 7 – 17) to show possible usage in research projects aiming at evaluating future platforms with 1000+ cores. Hence, all goals of WP7 and WP8 for the fourth year were achieved. In the future, this document could constitute a basis for tutorials and will be released freely for further extensions and improvements.

Document Organization

The purpose of this document is to provide all the information needed by a new user to start using the common simulation platform (COTSon). The document is organized into two main parts: from page 5 to page 48 there is a general introduction and description of the simulation platform and its main components, while the rest of the document presents a set of examples demonstrating the use of the simulator for research activities within the TERAFLUX project (essentially, one example for each partner). Given this document organization, **we decide to use sections from 15.1 to 15.4, devoted to the research example from UD, to integrate the content of Deliverable D8.3 - Final Results from the combination of UD and TERAFLUX dataflow techniques.** Thus, example from UD describes the use of its DARTS run-time ported on the TERAFLUX platform. **For the purpose of completeness, we added sections from 18.1 to 18.4, in which we describe the DRT (Dataflow Run-Time), essentially a simple run-time library that allows to test T* compliant applications directly on the host system.** Also this section can be considered part of the Deliverable D8.3

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 11 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Relation to other deliverables

Since the work described in this deliverable refers to the activity of all the partners in using the common simulation platform for their specific research activities, this document shows relations to several other deliverables. In particular, as the reader will see by reading the rest of the document, the main relations are with:

- D2.1, D2.2, D2.3, D2.4: analysis and identification of dataflow potential target applications. Within the WP2, Thales has ported two main applications to the TERAFLUX execution model, as demonstrated in this document;
- D3.5: transactional memory and OWM memory support;
- D4.7: compiler technologies targeting dataflow applications;
- D5.4: resiliency techniques (e.g., fault detection mechanisms, etc.) and the OS support for reliable execution have been developed within the WP5;
- D6.3, D6.4: since the work carried out in WP6 refers to the development of the TERAFLUX architecture, several examples presented in this document clearly use the results coming from the WP6 (i.e., the TSUF, TSU4, and TSU++ models for the hardware TERAFLUX thread scheduler);
- D7.1, D7.2, D7.3, D7.4: this document is the result of the activity carried out within WP7 during the all project time-frame;
- D8.1, D8.2: this document presents the main results of the activity carried out in the context of WP8. In particular, UNISI and UD continued to exchange information regarding their respective execution models. The result of this cooperation (WP8) is the porting of UD run-time on the TERAFLUX system, as also demonstrated by the work in WP9;
- D9.1, D9.2, D9.3: this document presents an example showing the results obtained in the context of WP9.

Activities referred by this deliverable

This deliverable refers to the research carried out in Task 7.1 (m1-m51), Task 8.2 (m28-m51), and Task 8.3 (m28-m51). In particular, Task 7.1 *covers an ongoing activity for the entire duration of the project that ensures the tools are appropriately disseminated and supported within the consortium.*

As a summary of the previous work carried out in the context of WP7 (deliverables D7.1, D7.2, D7.3, and D7.4), during the first two years, the TERAFLUX partners started using COTSon, and modified it in order to implement (test and validate) new features, to meet their research needs. As a result of this activity, we are able to boot a 1000+ cores machine, based on the baseline architectural template described in D7.1. The target architecture can exploit all the features added by the various partners to the common platform: this is very important for the integration of the research efforts carried out in the various TERAFLUX WPs. In particular, an initial FDU interface with the TSU (both DTS style and DDM style), has been described in D7.2, and further detailed in D7.3. Similarly, in D7.3 a first model for the development to monitor power consumption and temperature was reported. Finally, the D7.4, reports the result of an initial knowledge transfer activity. In particular, the document provides a description of the integration research activity through the COTSon simulation platform, as progressed during the third year of the project, such as the development of the T* and TSU. Thanks to

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 12 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

an internal dissemination, partners have been also able to transfer their respective research knowledge to the other partners.

Task 8.2 and task 8.3 *cover the joint activity of UNISI and UD. The activity is mainly devoted to interacting each other towards the completion of porting the UD run-time in the TERAFLUX platform.* As reported in the Annex-I these tasks refer to ongoing activities covering the period of entering the Consortium by UD till the end of the project. As a summary of the initial work carried out in the context of WP8 (deliverables D8.1 and D8.2), UD and UNISI exchanged information on their respective execution models (UNISI shared information regarding activities of all the partners, acting as the representative of the previous TERAFLUX consortium). After this initial period, UD and UNISI started to identify the best way to integrate UD run-time and the TERAFLUX platform (essentially by analyzing the features of both the execution models).

The effectiveness of the Dataflow approach has been verified on both platforms (Cyclops-64 and TERFLUX) with a close match on scalability for same benchmarks. Transitioning tools from TERAFLUX to Cyclops-64 was also considered, but we finally chose the TERAFLUX platform as it was a better fit as an open-source based research platform

We started with existing tools on each platform and an extensive evaluation was carried out, achieving an overall improvement of those tools. As a result, we decided to use the UD Runtime (DARTS) which faithfully implements the dataflow-codelet model to show the potentiality of Cyclops-64 program execution model on the TERAFLUX platforms (experiments for integrations are included in this document, while the actual results were detailed in D9.3).

Conclusions

The first purpose of this document is to provide all the necessary information to start using the common simulation platform (i.e., COTSon), with a specific focus on the first installation and configuration phase. The document has also other two important purposes: presenting in a detailed form, all the components that characterize the simulation platform, so that the final user is enabled to start designing and developing new hardware and software components; second (but not less important) presenting a full list of research examples, that serve as a reference for the user in its research and developing activity for a teradevice system as described in TERAFLUX (cf. D6.2, D7.1).

This document represents also the culmination of all the work carried out by all the partners during the project time frame. By inserting examples specific of each work package, all the partners demonstrated to have achieved their research objectives through the usage of the common simulation platform. All examples have been tested by several partners and by the WP leader (UNISI).

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 13 of 100

1 Getting Started

The goal of this initial part is to enable the user to run a first initial example, starting from scratch in two simple steps.

1.1 Step1: installation

To use COTSon, you need to install also additional software components, such as AMD SimNow™, on your Linux system (we refer to Ubuntu 10.04, but similar steps can be done, e.g. on Fedora or other distributions).

The simplest way to get SimNow is through your internet browser (such as Mozilla Firefox, Google Chrome); you can just click on the following URL and download the Linux version of SimNow (at the time of writing this document, the latest version of SimNow is 4.6.2):

```
http://developer.amd.com/tools-and-sdks/cpu-development/simnow-simulator/
```

The installation process starts by creating the installation folder:

```
$ mkdir installation_dir
```

The following command will copy the downloaded package in that folder:

```
$ mv simnow-linux64-4.6.2pub.tar.gz installation_dir/
$ cd installation_dir
```

Another prerequisite is the availability of the 'subversion' package. At the same time you can install 'md5sum'. To install them, for Ubuntu or Debian issue:

```
$ sudo apt-get –y install subversion coreutils
```

Alternatively, for Fedora issue:

```
$ sudo yum –y install subversion coreutils
```

It's warmly recommended that you verify the correct download of the package with the following command:

```
$ md5sum simnow-linux64-4.6.2pub.tar.gz
```

Check that the produced string is the same as on AMD website. Then unpack the module as follows:

```
$ tar xvzf simnow-linux64-4.6.2pub.tar.gz
```

At this point, in order to download COTSon, the following command can be issued.

```
$ svn co https://svn.code.sf.net/p/cotson/code/trunk cotson
```

1.2.1 Configuring COTSon Simulator

Once the two components have been correctly downloaded, it is possible to run the configuration and installation process. The installation process consists of source file compilation, and installation in the host system. To run the compilation, the following command must be issued (administrative permission may be required to complete the process):

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 14 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

```
$ cd cotson
$ ./configure --simnow_dir ../simnow-linux64-4.6.2pub/
```

It is important to note that during the installation process an error message could be showed to notify the user about host system configuration. For the simulator installation it is required to set the virtual mapping to a minimum value of 4194304. The error message is:

```
…
SIMNOW_DIR: '../simnow-linux64-4.6.2pub/'
ERROR: vm.max_map_count = 2048757 is too small
    Increase it to at least 4194304 by running
        sudo sysctl -w vm.max_map_count=4194304

    To make it permanent, add the following line to /etc/sysctl.conf
        vm.max_map_count = 4194304
…
```

To continue without generating errors, you can issue:

```
$ sudo sysctl -w vm.max_map_count=4194304
```

Later you can make it permanent as suggested above. The installation process ends by issuing the following command (this may require 10 to 15 minutes depending on the speed of your machine):

```
$ make release
```

During the compilation phase some windows could be popped up. These windows are part of the installation process and are closed at the end of the installation.

1.2 Step 2: running a first example

In order to verify the correctness of the installation process (it is worthy to observe that during the simulation framework installation, several tests are automatically run to check the process), it is possible to run a simple example as follows. Move under the example folder:

```
$ cd src/examples
```

Start the functional simulation of a simple target architecture through the following command:

```
$ ../../bin/cotson functional.in
```

If everything is correct, the user should be prompted to press enter (or ctrl-c to abort). Pressing enter causes the following window to be displayed (Fig. 1):

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 15 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Fig. 1 – Graphical control window of the COTSon simulator

At this point, you can click inside the "black" window (enlarge it to see the last lines, the icon before the last one in the command bar), press the "play" button (seventh icon of the command bar in this picture) and issue, e.g., an 'ls' command. Once done, you can close this window a return to the shell of the host system.

1.3 COTSon simulator: look at a glance

COTSon is a simulation framework, whose aim is to provide an evaluation platform for real systems like current multi-core Personal Computers consisting of x86_64 processors and all classical peripherals, and running available operating systems such as Linux (or, not shown here, Windows™).

It was originally developed by HP Labs and AMD, and it targets cluster-level systems composed of hundreds or thousands of commodity multi-core nodes and their associated devices connected through a standard communication network like, e.g., a datacenter.

An accurate evaluation may require to model not only the functional behavior (like in common "virtualizers" like VMWare™, Virtualbox™ and similar) but also the timing behavior of the architectural components. With COTSon the evaluation can range from high-simulation speed (and an "idealistic timing model" of 1 instruction per cycle) through an accurate timing model (up to desired level of accuracy). Moreover, COTSon can trade simulation speed with accuracy by offering about seven built-in sampling policies that can enhance greatly the simulation speed (and the user can provide his/her own sampling policies).

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 16 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

1.4 Supported platforms

In order to run COTSon the user needs a computer equipped with a 64 bit processor. This is required in order to correctly run the AMD SimNow [TM] virtualization layer (this component is available only for Linux AMD64 and Windows XP 64-bit version, however the entire simulation framework is available only under the Linux environment. Hereafter we refer to the virtualization layer simply as SimNow). Currently, **COTSon** (**v680**) requires the **4.6.2pub** version of SimNow, while it supports the following Linux distributions:

Supported Linux Distributions		
Debian	**Fedora**	**Ubuntu**
Lenny	Werewolf	Intrepid
Squeeze	Leonidas	Jaunty
	Goddard	Karmic
	Laughlin	Lucid
	Lovelock	Maverick
	Verne	Natty
	Beefy Miracle	Oneiric
	Spherical Cow	Precise
	Schrödinger's Cat	Quantal
		Raring

Table 1 – COTSon installation: supported Linux distributions.

The minimum hardware configuration required for the installation is as follows:

- *Processor*: AMD Athlon[TM] 64 X2 Dual Core Processor 4600+ or equivalent;

- *Memory*: 2 GB of main memory (8GB or more recommended);

Please also note that for licensing issues the simulator should be run on AMD machines, even though Intel processors are also reported to function).

1.4.1   Running COTSon in a virtualized environment

Installation under Windows environment is supported, through the use of virtualization software (e.g., VirtualBox, VMware, etc.), by allocating enough resources to the guest machine. This kind of installation is also suited for shared environments, where a single server can host several virtualized machines. In this case virtualized machine can be remotely accessed. For further information on virtualization software, please refer to the specific manual of AMD SimNow.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 17 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

1.5 Document structure

The rest of the document is organized as follows. Section 2 and section 3, are devoted to the description of the main characteristics of the simulator. In particular, the guide focuses on the general architecture, the mechanism implemented to collect timing information, and the description of the main internal components (such as the virtualization layer, the interleavers, the samplers, etc.). An entire section is devoted to the user interface used to configure and interact with the simulator. COTSon adopts the LUA language (see Appendix-1) to provide a flexible way to describe the configuration of the target system (i.e., the architecture of the system to be simulated), and the parameters for the experiment setup (e.g., functional simulation vs. timing simulation, structure for storing collected measures, commands for the virtualization layer, etc.). Structures for collecting data during simulation are deeply described in section 5, while section 6 presents to the user a set of simple examples that illustrate all the features previously described. Following these examples the user should be able to set-up the simulation environment, and to run architectural simulations of interest. Finally, sections from 7 to 17 illustrate advanced examples that reflect research activity carried out in the TERAFLUX project at the scale of 1000+ cores [6][18]. They can be used as a reference for setting-up advanced simulation experiments. In particular, they can be used to understand how to extend the simulation infrastructure.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 18 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

2 Understanding COTSon: Design and Architecture

Simulation, combining some architectural structures, permits to create virtual systems in which hardware components are shaped, in order to make new functional units, or entire microprocessor systems. The aim of a simulator is to show, record and analyze the performances and the behavior of applications, and select the best architecture for each of them. Simulators can be also used to develop new software and hardware components that can be thus verified in their behavior. The increasing complexity of computing systems has made simulators the first choice for their design and analysis. In fact, a good simulator infrastructure can help researchers, designers and developers in verifying if their decisions are correct or not, possibly finding some optimal solutions. Speed, accuracy, full-system capability and ability to extract specific metrics are the main characteristics of a simulator and also what makes one simulator different from another.

COTSon is a simulation framework targeting many-core architectures, initially developed by HP Labs. The key feature of COTSon is the adoption of a *functional-directed* simulation approach, where fast functional emulators and timing models cooperate to improve the simulation accuracy at a speed sufficient to simulate the full stack of applications, middleware and OS. Functional simulation emulates the behavior of the hardware components (e.g., common devices such as disks, video, and network interfaces) of the target system, without considering latency information. On the contrary, timing simulation is used to assess the performance of the system. It models the operation latency of devices simulated by the functional simulator and assures that events generated by these devices are simulated in a correct time ordering.

2.1 Major Design Characteristics and comparison with other simulators

Depending on how the functional and the timing parts of the simulator are controlled and on their relationship, it is possible to define different types of simulations:

- *Timing-directed or execution-driven*: here the timing model of the simulator is in charge of driving the functional simulation. In this case the functional and timing parts are programmed tightly coupled to let the two parts cooperate easily;
- *Functional-first or trace-driven*: in this case the functional simulation produces an open-loop trace of the instructions that have been executed. Then, these instructions will be passed to the timing simulator. This type of simulator is usually built using particular libraries such as Atom or Pin;
- *Timing-first*: timing and functional models are decoupled and timing drives the simulation. In this approach the timing simulator precedes the functional simulator, and uses the latter to periodically check and correct the simulation state (eventually functional execution may have to be undone);
- *Functional-directed*: timing and functional models are decoupled and functional drives the simulation. In this approach was proposed to treat better complex benchmarks and to afford greater speed scalability; the timing feedback corrects the timing so that it becomes visible to the application running on the simulated machine.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 19 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

COTSon uses the later approach (functional directed simulation: the functional and timing simulation are clearly separated using two interfaces. This approach allows reusing existing functional simulators (very difficult to implement and maintain). COTSon's functional simulator is SimNow that functionally models most of the existing hardware that can be found on a modern AMD system (in this sense it supports generic X86_64 architectures). SimNow contains also the internal capability of timing simulation but such information is completely discarded when used in conjunction with COTSon: only the CPU capability is used in this case. COTSon is highly modular, and this characteristic enables users to select different timing models, depending on the particular experiment they want to perform. It is also possible to program new timing models (e.g., a new coherence protocol) or to adapt the existing ones (e.g., cache timing with MESI protocol), and incorporate them into COTSon. Another very important aspect of COTSon is the speed. In fact even if it is not significant in terms of simulation results, a full system model simulator can be five or six orders of magnitude slower than the real system, and this may become unsustainable, as it limits the coverage of experiments. To speed up the simulations COTSon uses virtual machine techniques for its functional simulation (that comprehends just in time compiling and code caching) and also sophisticated techniques such as "dynamic sampling".

2.2 Timing Feedback

As discussed in the previous section, the aim of COTSon is to achieve the best possible trade-off between simulation speed and accuracy for many-cores systems (e.g., systems equipped with hundreds or even thousands cores). To this end the design choice made was to use a functional-directed approach, where the functional simulation of the target architecture (fast) is periodically updated and its timing is integrated with information coming from timing models of the architecture components.

In a pure trace-driven systems in fact, there is no influence on the functional part coming from the timing part. This does not represent a big limitation in case of single core systems, but can be a problem in multicore systems. In fact the latter usually change their functional behavior depending on their performance. For example, threads in a multi-threaded application exhibit different interleaving patterns, depending on the performance of each thread (possibly running on different cores). On another level, many networking libraries such as Message Passing Interface (MPI) change their policies and algorithms depending on the particular performance of the network
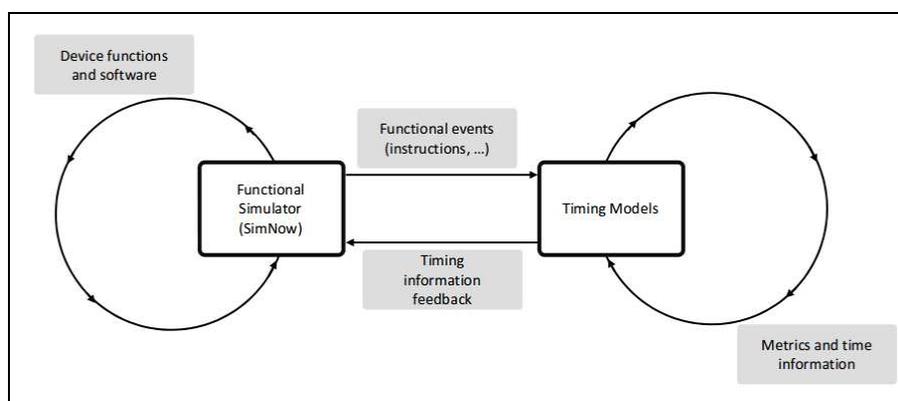
Fig. 2 - Interaction between functional simulation components and timing components in COTSon simulator.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 20 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Having *timing feedback*, i.e., a communication path from the timing to the functional simulator becomes fundamental for analyzing this kind of situations. From this viewpoint, COTSon makes its functional simulator run for a time interval Δt that is dynamically set. The produced stream of references (i.e., instructions and data memory accesses, but in general "events") is sent to the respective CPU timing models. At the end of such interval using the metrics coming from the CPU models, the actual time interval to process such stream of reference is known (say Δt') and it is given back to the functional simulator. The user can select different interval sizes to choose the accuracy-speed trade-off. Therefore, COTSon (realizing this trade-off between accuracy and speed) enables users to avoid uninteresting parts of the code (such as initial loading of the system) simulating them at lower accuracy.

2.3 Architecture

The COTSon architecture has been developed having in mind the simulation of clusters. From this viewpoint COTSon uses a SimNow instance to represent each node of the cluster. SimNow has been augmented, by HP-Labs and AMD, with a double communication layer to allow any device to export functional events and obtain timing information. All the events are directed by COTSon to the timing models.

There are two types of communication mechanisms exhibited by devices: synchronous and asynchronous. *Synchronous* communication is used for devices that immediately respond with timing information for each event received (and the event does not occur very frequently). An example of synchronous communication is the simulation of a disk read by the functional simulator: a read event (instead of an interrupt) is issued to COTSon, which delivers this event to a disk model that determines the operation's latency, which is used by SimNow to schedule the functional interrupt, which signals the end of the read.

Synchronous communication is not usable when there is a high frequency of events of this type (e.g., main memory accesses, CPU simulation, etc.). In these cases *asynchronous communication* is needed. Differently from the synchronous case, the SimNow simulator does not do a call per event, but produces "tokens" describing dynamic events, that will be parsed by COTSon and delivered to the appropriate timing modules. These modules will be asked by COTSon at specific moments to aggregate timing information (in term of number of instructions and cycles) and give them back to each functional core.
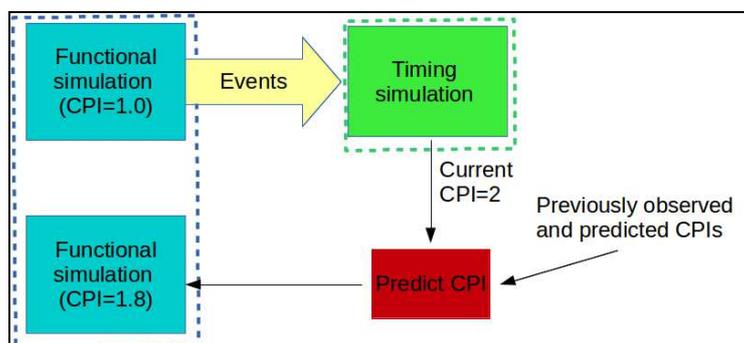
Fig. 3 – Example of timing feedback with asynchronous communication for estimating the IPC in COTSon.

For example, in Fig. 3 we show the situation when a timing module is used for a processor pipeline with the purpose of estimating the number of Cycles Per Instruction (CPI). The resulting CPI, given

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 21 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

back to the functional module, is used by SimNow to schedule the progress of instructions in each core and in this way the timing feedback is used for the functional simulations. However in many situations the timing feedback has to be filtered and modified, in order to obtain an increase in simulation accuracy. For example if a particular core is mostly idle it doesn't give an accurate estimate of the CPI. To solve this problem, COTSon offers a timing feedback interface that handles these modifications transparently. This interface is able to correct and predict future CPI by using mathematical models, such as Auto-Regressive-Moving-Average (ARMA) model, that is used, e.g., in forecasting time series. A simple example of the timing feedback mechanism is shown in Fig. 3.

2.4 COTSon installation structure

Once COTSon is installed the user will get a directory structure as follows:

- *bin*: contains binaries of the simulator;
- *data*: contains the *bsd* images and the *disk* images used to run simulations;
- *share contains some common scripting files*;
- *src*: contains all the files related to the development of the simulator;
- *sandbox*: it's the template of a 'sandbox' on the host used to control a node during the simulation
- *etc*: COTSon general configuration files
- *sbin*: COTSon general system binaries
- *daemon*: contains files for running the simulator in a distributed environment (not described in this document);
- *web*: COTSon web control (not described in this document)

The *src* directory has the following structure:

- *src/abaeterno/* it is the core COTSon infrastructure. This directory contains timers, samplers and the simnow interface;
- *src/common/* common utilities (metrics, options, etc.) for abaeterno and network;
- *src/disksim/* disksim distribution for COTSon;
- *src/distorm/* distorm (x86 disassembler) for COTSon;
- *src/examples/* simple simulation examples (we will analyze them after);
- *src/libluabind/* C++ binding for LUA (used for COTSon scripting);
- *src/network/* COTSon (HP) network mediator (for distributed synchronization);
- *src/mcpat used for power and area estimation through the HP McPAT tool*
- *src/slirp/* slirp library (NAT access from guest) for COTSon;
- *src/test.regression/* simple regression tests;
- *src/tools/* tools to support simulation experiments;

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 22 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

3 COTSon components: SimNOW, Samplers, Interleaver, Timers

The main parts of a COTSon node, are the functional simulator SimNow, the timing models (timers), the sampler, the interleaver, and the time predictor. Moreover, the network *Mediator* and the *Control* are two components of COTSon that allow the simulation of cluster configurations (Fig. 4). The dynamically loaded library (DLL) *abaeterno*, is also a fundamental part of COTSon, because, when loaded by SimNow, it determines the time the simulation is taking, and it contains the implementation of all types of timers, samplers, etc., that can be used by COTSon.
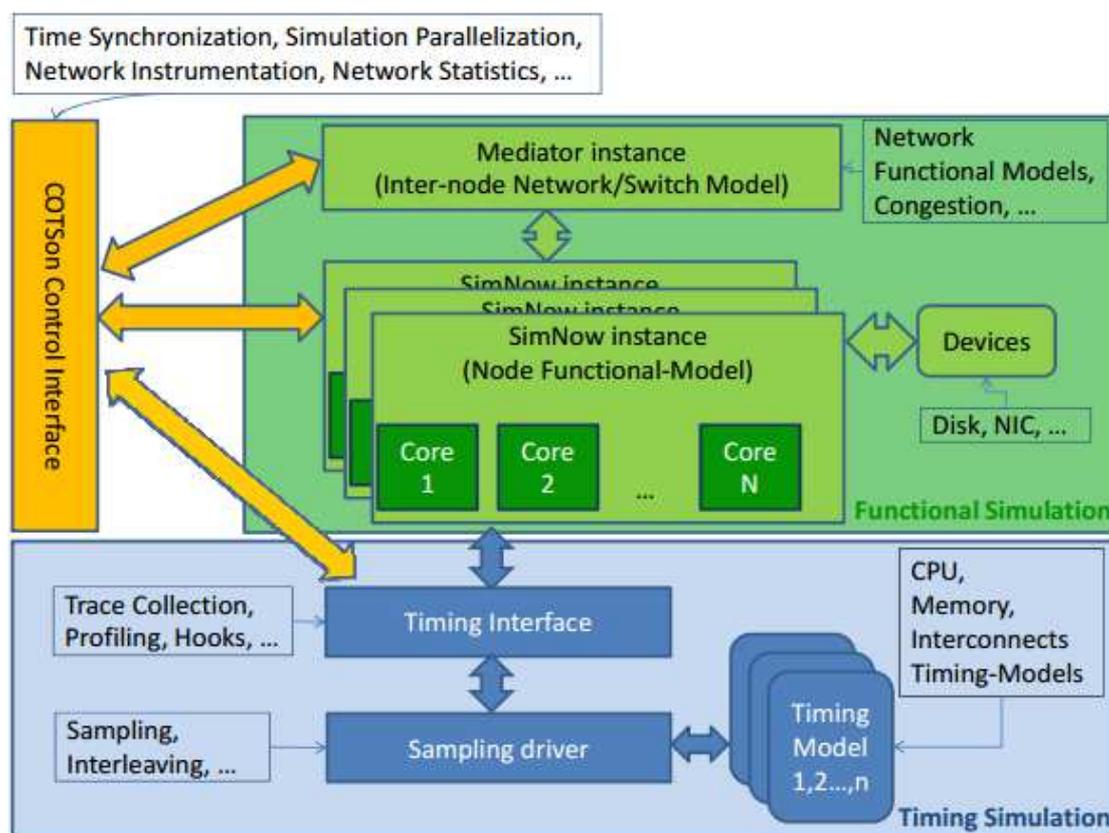
Fig. 4 – COTSon components overview

3.1 Virtualizer: short introduction to SimNow

It implements the x86 and x86_64 instruction sets, including system devices. It allows the user to configure a full-system architecture by changing the various components (i.e., CPU type, number of CPUs, organization, main memory size, etc.).

SimNow provides several CPU models, dynamic translation of instructions (the instruction input stream is translated into C-like language and then is compiled for the native machine) and deterministic execution; it can simulate the majority of existing hardware uniprocessor and multiprocessor that are available on a modern AMD system. It also uses caching techniques and supports the booting of an unmodified Operating System (such as Windows and Linux) over which some complex applications can be executed. In full-speed mode SimNow performance is around 100-

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 23 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

200 MIPS (i.e., it has a 10x slowdown with respect to the native execution). It comes with several Broad-Sword Document (BSD) configurations, i.e., files containing setup parameters of a simulated target machine. The host machine, in which the simulator runs, and the guest machine, i.e. the simulated machine, can communicate through a toolbox called *Xtools*, mainly constituted of two commands: i) xput, which is run on the guest to copy a file from the guest to the host and ii) xget, which is run in th guest to copy a file from the host to the guest SimNow can be controlled from the shell (command line mode) or through a User Interface Window (graphical mode – see Fig. 5). When using the graphical mode, users see and modify the target system configuration (i.e., the configuration of simulated devices such as disk images, BIOS, DRAM and CPU) from the main windows, and they can access to the results of the simulation as well. The main window is divided in two main parts: one shows time results of the simulation, while in the other a console provides a textual interface for status information and a command-line control for the guest OS running in the host.

The part showing time results is called SimStats and it is composed of 4 components:

- *Host Seconds* (1): showing the number of seconds spent (both in user and system mode) by the host CPU, since the simulation has started;
- *Sim Seconds* (2): showing the time spent in the simulation since it has started;
- *Avg MIPS* (3): showing the instantaneous values of the simulator performances, that is measured in millions of executed (simulated) instructions per host.
- *MIPS* (4): showing the number of simulated instructions from the start of the simulation, divided by host seconds;

Below there is the *Console Window* (5): providing the guest output and control for the guest OS;
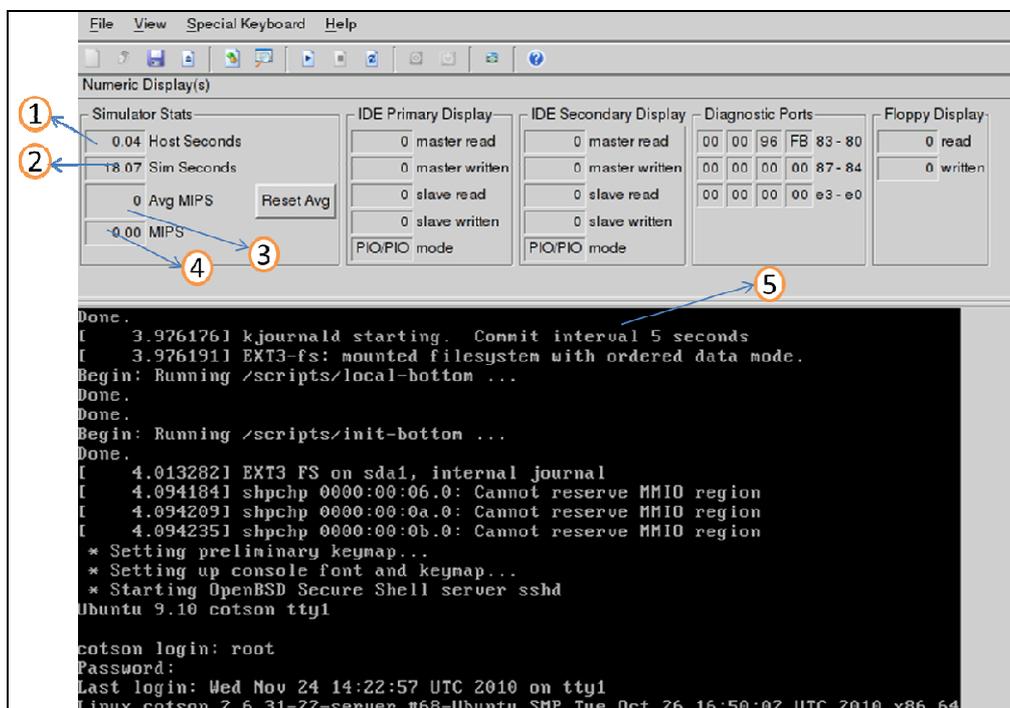
Fig. 5 – Graphical interface of the COTSon simulator. The window contains a toolbar from which interact with the simulator, a panel displaying statistical information, and a control panel from which interact with the guest system.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 24 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

3.2 Samplers

COTSon can be configured to use a full-speed functional modality or a sampled modality. The samplers are one of the most important parts of COTSon infrastructure, as they represent the way functional and timing simulations are integrated together. This can be seen also in Fig. 4, where the sampler is placed between the front-end (functional simulator) and the back-end (timing models of the architectural components) of the COTSon node. Sampling is crucial for asynchronous devices and it is the process through which the timing simulation (or simply simulation) is turned off or on. A good sampler is required to select a simulation interval such that the simulation metrics taken in that interval well approximates the statistics of the whole execution. So the timing simulation will be performed only in appropriate moments and for an appropriate duration, thus avoiding the slow-down of timing simulation.

The type of sampler required for a certain experiment and the lengths and the type of the samples can be configured by writing proper values in the COTSon configuration file (see Section 4). With this information, the sampler gives a command to enter one of the following phases:

- *Functional*: during this phase only functional simulation is performed and so no events are produced by the simulated devices, that so are simulated at full speed;
- *Warming* (simple/detailed): this phase is necessary to pass from functional to timing simulation; during it the timing models are warmed up to prepare them to the timing simulation. If only the high-hysteresis elements (such as caches and branch target buffers) are warmed up, the warming is said to be simple, otherwise, if also the low-hysteresis elements (such as reorder buffers and renaming tables) are warmed up, the warming is called detailed;
- *Simulation*: this phase is the opposite of the functional phase. Here the devices must produce events that are sent to the timing models, so that timing simulation can be performed;

In order to determine sampling intervals, it is necessary to find out what are the most representative and relevant parts of the application's execution. This selection is based on the phase analysis, which determines the phases of a program, i.e., the parts of the execution that have a similar behavior, independently of temporal adjacency. Depending on how the phases of a program are detected different samplers can be implemented. The most important samplers are *SMARTS*, *SimPoint*, *dynamic* samplers, and *interval-based* samplers. The first two require an a priori profiling or a preprocessing of the code and don't allow timing feedback.
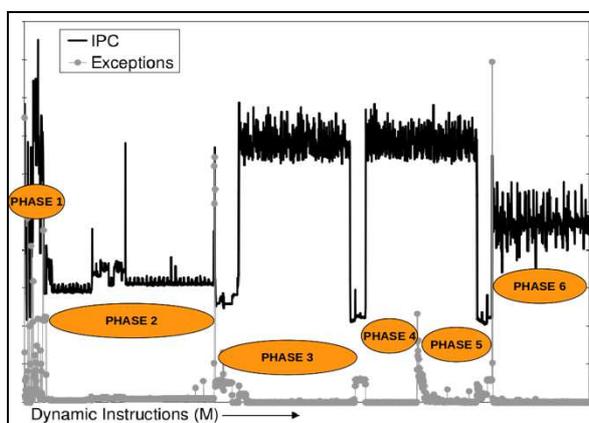
Fig. 6 – Correlation of the performance information acquired by the simulator with the running application phases.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 25 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Because of these two characteristics, they result to be less flexible than dynamic samplers and may be subject to errors due to the absence of timing feedback. In the interval-based sampler the duration of each phase (state) of the sampling (functional, warming, simulation) is fixed. Dynamic Sampling is based on the consideration that all functional simulators (such as fast emulators, like SimNow, or virtual machines, like VMware) keep track of internal statistics of two types:

- Those related to their internal structures (translation cache, software TLB), such as code cache invalidations, code exceptions, and I/O operations;
- Those related to the emulated code, such as number of executed instructions, memory accesses, exceptions, and bytes read or written to or from a device;

Both types of metrics are strictly related to the behavior and the performance of the emulated software and can be used to detect phase changes in an application's execution. Fig. 6 shows an example of how an internal statistic (number of code Exceptions) is correlated to the application's performance (IPC) and thus to the application's phases. The dynamic sampler lets a timing simulation start whenever the first-derivative of the chosen internal statistic overcomes a threshold. After a certain number of instructions, the simulation returns to be functional, until the next phase change is detected, and so on. Fig. 7 shows a schematic view of how Dynamic Sampling works.
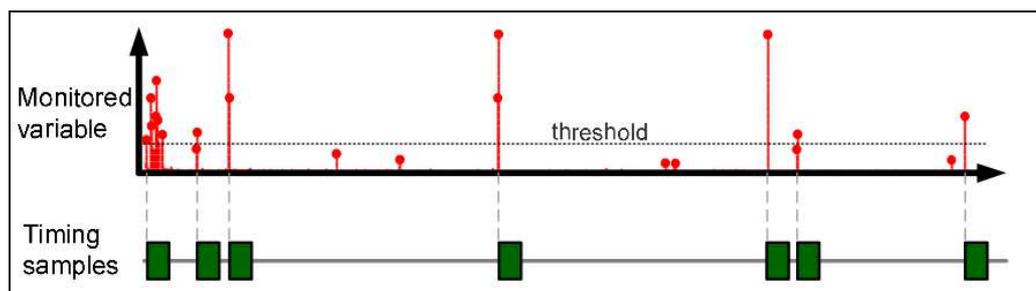
Fig. 7 - A schematic representation of how dynamic sampling works.

Different types of samplers can be selected by the user, writing appropriate values in the COTSon (LUA) configuration file.

3.3 Interleavers

The interleaver is a component that is used during the simulation of SMP (Symmetric Multi-Processor), i.e., multi-core systems. In fact, it supervises the buffering and the reordering of the events coming from the functional simulation. These operations are fundamental when multiple cores are simulated. To this end, SimNow simulates multi-cores with an interleaved sequence. After a certain interval of time, called *synchronization quantum*, during which the cores operate independently, all the cores arrive to the same point in time. After the synchronization quantum, all the events are stored in a queue and then they are interleaved. Only at this moment they are ready to be carried to the timing models of the CPUs.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 26 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

3.4 Timers

There is a timer for each architectural component that can be simulated, and its role is to collect events coming from the functional simulation, and use them to update the timing model of the component. In other words a timer is software that simulates the timing behavior of each component. There are timers for the CPU, for the Memory, for the disks, and for the NIC (Network Interface). The type of timer (e.g., timer0 – for an in-order superscalar processor, timer1 – for an out-of-order superscalar processor, bandwidth – for measuring the memory bandwidth, etc.) can be set in the COTSon configuration file. The feedback information is governed by the *time predictor*: based on the metrics collected by the timing simulation, it decides how to feedback information to the functional simulator.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 27 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

4 COTSon configuration

A simple COTSon configuration file (written in lua file 'functional.in') to run a functional simulation is shown in Fig. 8. It uses the 'functional template (first line), shows a graphical display for Simnow (second line), where ('simnow.commands') the architectural configuration of the SimNow uses the '1p.bsd' (fourth line, that also stores the snapshots and modifications of the running simulation), an off-the-shelf hard disk image with the Operating System (this remains unmodified during the simulation, fifth line), and we enable the journaling of the file system (sixth line)

```
one_node_script='functional'
display=os.getenv("DISPLAY")
simnow.commands=function()
    use_bsd('1p.bsd')
    use_hdd('karmic64.img')
    set_journal()
end
```

Fig. 8 – A simple COTSon configuration file (written in lua file 'functional.in'

4.1 Lua Scripting

The COTSon simulation infrastructure is controlled by setting all the relevant information about simulation and the target system configuration in an input configuration file. COTSon uses *Lua scripting language* to manage this configuration file. The Lua scripting language is powerful, fast, lightweight, and embeddable. It combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping. For further information about Lua language syntax, see Appendix A – Lua lexical conventions, and Appendix B – Lua language features.

Suppose the user wants to run the functional example (*functional.in*) present in the directory `cotson/src/examples`:

```
$ cd cotson/src/examples
```

Then simply issue the command:

```
$ ../../bin/cotson functional.in
```

This will launch the SimNow window as explained in Section 1.2 (Step 2: running a first example).

One of the nice features of the Lua scripts is that they accept Lua parameters either in files or in the command line. Anything that is not strictly an existing object, is considered part of the Lua syntax (see Appendix A – Lua lexical conventions). The Lua script is the concatenation of the contents of all the files and the Lua syntax, and it is passed to any part of COTSon that would need it (like the COTSon Control script – named 'cotson', the 'abaeterno' library). Even if not every part of the elements written in the Lua file is needed by these components, each of them can select the parts that are needed.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 28 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

4.2 Changing the configuration

The Lua configuration file used in COTSon is divided into 3 main sections:

- *LUA-SECTION-1*: describes general simulation options. This part is called *options table;*
- *LUA-SECTION-2*: describes options/commands for SimNow. This part is called *SimNow table*, and is used by the control scripts to determine how to set up the SimNow execution;
- *LUA-SECTION-3*: describes the target system configuration in details. This part is called *build function.* Anything inside it or in the *options table* is used by the abaeterno library. (Anything that follows may be by the COTSon control and web interface to determine what kind of execution to make);

4.2.1 Lua-Section-1 –options table

This first section in the Lua file is delimited by:

```
options={}
```

Here several options can be specified, in particular, the following variables can be set:

- *max_nanos*: is the variable where we specify how long we want the simulation to last in terms of nanoseconds (e.g. "10M", see Fig. 9);
- *sampler*: where the type and the various options of the sampler chosen can be specified (e.g. type="simple" indicates a detailed timing simulation (the opposite of the pure functional simulation) and quantum="100k" indicates how often the functional part has to synchronize with the timing part – see Fig. 9); also note how we can nest multiple lua commands.
- *heartbeat:* this is used to specify how to log statistics (e.g., type="file_last" indicates to dump all statistics in a file at the end of the simulation and in such case logfile="on_cpu_simple.log" indicates the name of the file – see Fig. 9). There can be instantiated up to eight heartbeat options ("heartbeat=", "heartbeat1=", …," heartbeat7=").

Other general options can be:
- *max_samples*: here the maximum number of samples is specified;
- *fastforward*: here it can be specified an amount of time that will be skipped by the simulation;

There are also several other types of sampler available like dynamic, interval (see Section 6.3 "Samplers: timing simulation"). Similarly for the *heartbeat,* it is possible to use the sqlite database (or files) and the statistics can be dumped at intervals during the simulation – see Section 5.2 Database structure for more details). Whenever the results are stored in the database, the user has to specify also two particular fields that are *experiment_id* and *experiment_ description*, needed to store the data in the correct field inside the database tables for storing more experiments. Below (Fig. 9) an example of COTSon configuration file – section 1, taken from the file *one_cpu_simple.in* is shown.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 29 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Fig. 9 - An example of lua-section-1 of the COTSon configuration file (see also the example src/example/one_simple_cpu.in).

4.2.2   Lua-Section-2 – SimNow options/commands

This section is opened by the line:

```
simnow.commands=function()
```

This part is where the SimNow commands are grouped. Then the following options must be set (depending on the type of example the user is running, it can use a subset of the options listed below):

- *use_bsd( )*: here the bsd location is set. Possible types of bsd are available in the folder cotson/data.
- *use_hdd( )*: here we set the position of where the hard disk image is located, for example karmic64.img is available in the folder cotson/data.
- *set_journal( )*: this function is needed to enable the journaling of the file system.
- *send_keyboard( )*: this function allows the user to run a command inside the OS of the simulated machine.

In Fig. 10 the reader can see an example of lua-section-2, taken from *one_cpu_simple.in*. Other option (not show in Fig. 10 - An example of lua-section-2 of the COTSon configuration file (see also one_simple_cpu.in)) can be:

- *execute( )*: here the user can select the name of a (guest) file to be executed during the simulation (e.g., a bash script file). This file is copied from the host to the guest at the beginning of the simulation and has to be in the same folder where the lua script is stored.
- *subscribe_result( )*: serves to automatically copy the listed files from the guest to the host at the end of the simulation.
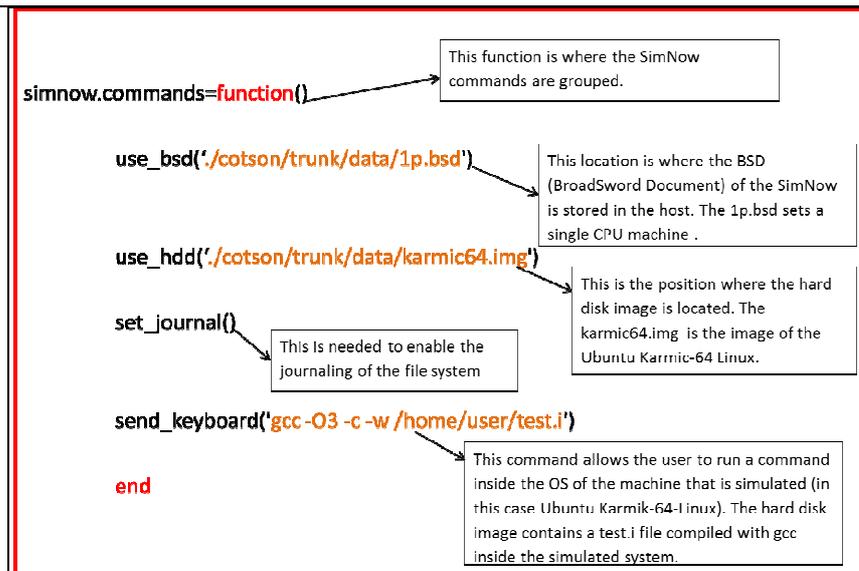
Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 30 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Fig. 10 - An example of lua-section-2 of the COTSon configuration file (see also one_simple_cpu.in)

4.2.3   Lua-Section-3 – configuration options

This section begins with the command (see Fig. 11):

```
function build()
```

After that, there is a part where the number of disks in the system is specified and for each disk the appropriate timer is set. Then, in the same way, it is found the number of the various Network Interfaces attached to the system and to each one a timer is assigned. Then we can specify the number of CPUs that are in the system. If the number is zero, the simulation is stopped. The numbering of the disks, NICs, CPUs will begin from zero (i.e., in a multi-core system CPUs are named as cpu0, cpu1, etc.). Similarly to disks and NICs, to each CPU a particular timer is assigned (e.g. "timer0" means a simple superscalar in-order processor). For the memory and caches, it is possible to decide the values of their main features, such as the latency. The memory is set following a hierarchical approach, in other words, usually the setting starts from the main memory, then the cache with its levels. For each cache level, we can set the values of some important variables, such as:

* *name*: determines the name of the considered cache level;
* *size*: determines the total size of the considered cache level;
* *latency*: determines the hit latency to access the considered cache level;
* *num_sets*: determines the number of sets that are present in the considered cache level;
* *write_policy*: determines the write policy of the considered cache level ("WB" means Write Back, "WT" means Write Through);
* *write_allocate*: if it is set to true, it means that the considered cache level is of type "write allocate", otherwise, the cache is of type "write-no-allocate";

Once all the memory components are set, we can connect them to the CPU using some particular commands such as:

```
cpu:instruction_cache(ic)
cpu:data_cache(dc)
cpu:instruction_tlb(it)
cpu:data_tlb(dt).
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 31 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

All the various parts previously described, can be seen in Fig. 11, which is an example of lua-sections3, again taken from *one_cpu_simple.in*.
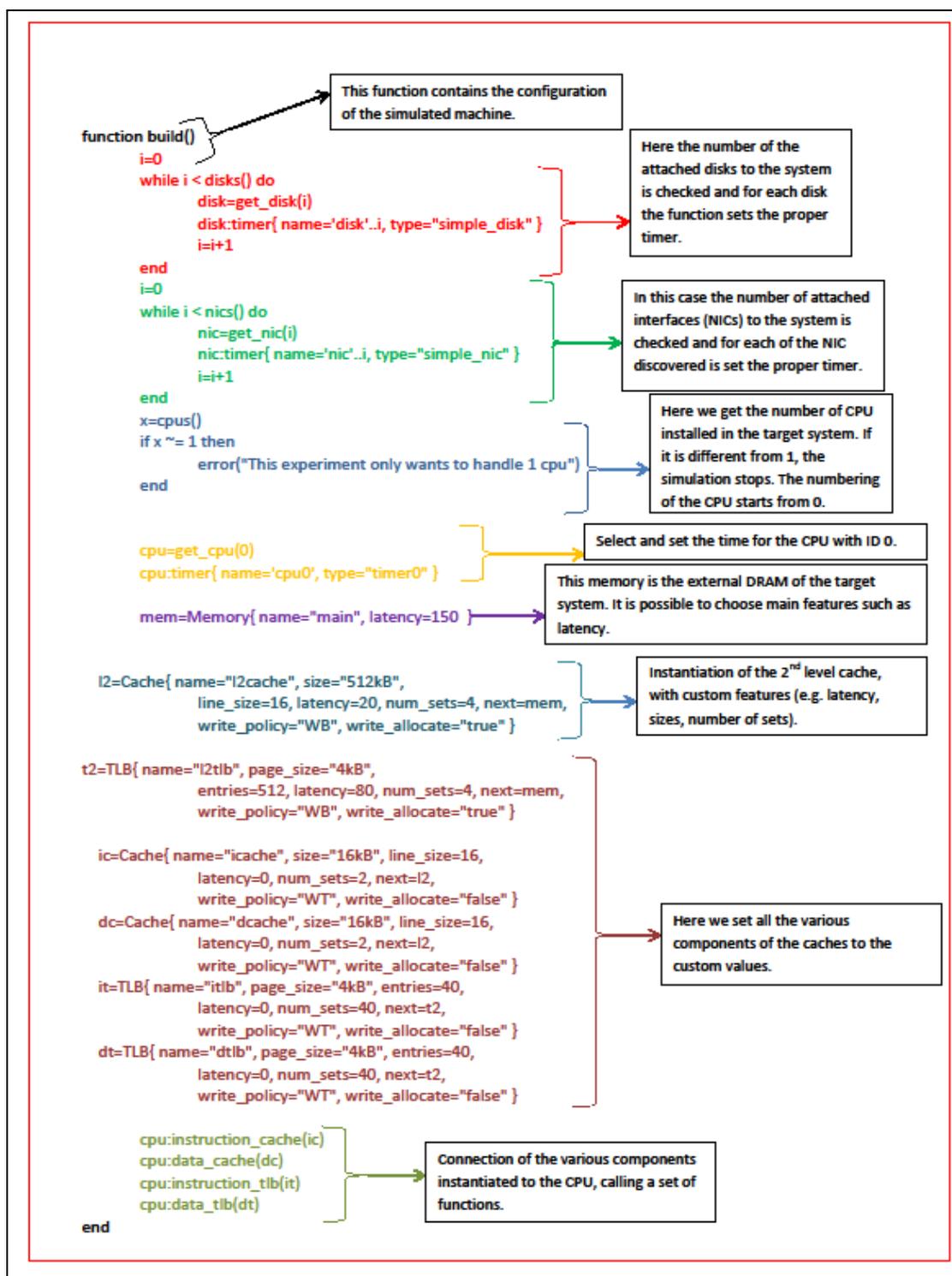
**Fig. 11 - An example of lua-section-3 of the COTSon configuration file (see also the example src/example/one_simple_cpu.in)**

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 32 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

5 Collecting Metrics

All the collected simulation measures can be permanently stored in a specific data structure. The user can chose which structure to use for storing information. The simulator provides two types of storing structures: the simplest is a *log file*, while the more advanced is represented by a *database*. Log file is generally enough to store data collected during a simulation. However, for keeping track of measures collected over several simulations, the database is the best choice. It allows maintaining information structured and it allows easily finding specific data by simply querying it. COTSon uses a flexible data storage resorting to a SQL server. By doing so, COTSon allows to search through simulation results in a more consistent way using a familiar declarative language like SQL.

5.1 Log structure

A *log file* is a simple text file, where all the information gathered by the simulator during a simulation is written. Since it is a text file, it can be automatically parsed at the end of the simulation. The main drawback of this structure is that it grows rapidly with the increase of simulation complexity.

5.2 Database structure

The simplest way to use a SQL server to store simulation heartbeats (i.e., periodic information collected by the simulator, such as instruction count, memory read misses, etc.) is to use SQLite server (currently at version 3). It should be installed by default with the Linux distribution. However, it is possible to check for its presence by using the following command:

```
$ sqlite3
```

One example that uses the database is governed by the "sqlite.in" lua script in the `src/examples` directory. To run it:

```
$ cd src/examples; make run_sqlite
```

You can check the content of the database by issuing:

```
$ sqlite3 /tmp/test.db
```

The tables in the database (hereafter DB for simplicity) can be analyzed by typing the following command (the SQLite server prompt is presented to the user):

```
sqlite> .tables
```

This should be the output the list of tables where results of the experiment are stored:

```
sqlite> .tables
experiments    heartbeats    metric_names  metrics       parameters
sqlite>
```

These are the tables where the SQLite module stores the data if we select *sqlite* as output for the simulation heartbeats and the data related to the experiment. In general, to enable the use of SQLite storage, the user has to change the configuration file adding the "heartbeat" line in the options section, as in the following example (see file 'sqlite.in' in the `src/examples` directory):

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 33 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

```
options = {
       heartbeat={
                    type="sqlite",
                    dbfile="/tmp/test.db",
                    experiment_id=1,
                    experiment_description="T1"
                 },
       }
}
```

In order to get same the data from this DB the user should first look for the needed metric id:

```
sqlite> select * from metric_names where name like '%dcache.write_miss%';
```

The user should get the following output:

```
sqlite> select * from metric_names where name like '%dcache.write_miss%';
76|cpu0.timer.dcache.write_miss
281|cpu0.timer.dcache.write_miss_rate
sqlite>
```

And then look for the associated data in the metrics table using the "metric_id" values.

```
sqlite> select * from metrics where metric_id = 76;
```

And obtain a long list (here we show only the last three elements):

```
283|76|62.0
284|76|133.0
285|76|16.0
298|76|80492.0
```

This is where things may not seem clear at first. The table is organized so that the first n-1 records contain the value for every sample in the value field. The last one contains the actual result (in this case the sum of all of the previous records). So the user can get the actual result with:

```
sqlite> select value from metrics where metric_id=76 and heartbeat_id is
(select max(heartbeat_id) from metrics);
```

The user should be the one showed below, which should also be the same obtained from the flat file:

```
sqlite> select value from metrics where metric_id=76 and heartbeat_id is (select max(heartbeat_id) from metrics);
80492.0
```

As far as the write miss rate is concerned things, again, change a bit. This time we are not looking for the sum but for a rate so we can only get the value directly:

```
sqlite> select value from metrics where metric_id=281 and heartbeat_id is
(select max(heartbeat_id) from metrics);
```

This time the expected values is:

```
sqlite> select value from metrics where metric_id=281 and heartbeat_id is (select max(heartbeat_id) from metrics);
0.120777796283876
sqlite>
```

You get more digits from this than from the flat file because the value field is a "float8". You can see this by looking at the table schema:

```
sqlite> .schema metrics
```

which outputs:

```
CREATE TABLE metrics ( heartbeat_id integer REFERENCES heartbeats,    metric_id    integer REFERENCES metric_names,    value    float8,        CONSTRAINT non_repeated_metric_per_heart UNIQUE(heartbeat_id,metric_id));
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 34 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

5.2.1   Using a PostgreSQL server:

While using SQLite can be very convenient as it gives you the ability to store your heartbeats in a SQL server without the hassle of configuring a real SQL server it may not be the best solution if the user wants to store a very big amount of data and if it wants to offload the burden of saving data to another machine. In this case the best solution, albeit more demanding from the administrator viewpoint, might be setting up a second computer with PostgreSQL and using it to store the heartbeats produced by the simulations.

As an example in the following the PostgreSQL server is supposed to run on the same machine running COTSon (note that the process to run it in a classical client-server configuration is the same as explained here).

As PostgreSQL is not usually installed by default it is necessary to install it. Type:

```
$ sudo apt-get –y install postgresql postgresql-client
```

Now the user should have its instance of PostgreSQL up and running on the specified machine. To verify it, the user can issue this command:

```
$ netstat -atp | grep post
```

This should be the output the user obtains:

```
tcp        0      0 localhost.lo:postgresql *:*                    LISTEN      -
```

If so then you can start configuring PostgreSQL to make it talk to COSTon.

5.2.2   Creating the COTSon PostgreSQL database:

In order to configure PostgreSQL the user has to create the "cotson" user in the database:

```
$ sudo –i
$ su - postgres
$ cd
$ createuser cotson
```

```
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

Answer "NO" (n) to the three questions following this command and then issue:

```
$ createdb cotson -O cotson
```

The user can verify that everything is ok by querying PostgreSQL and asking for the databases list:

```
$ psql -l
```

The output should be similar to the following

```
                            List of databases
   Name    |  Owner   | Encoding |  Collate    |   Ctype     |   Access privileges
-----------+----------+----------+-------------+-------------+----------------------
 cotson    | cotson   | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 postgres  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres         +
           |          |          |             |             | postgres=CTc/postgres
 template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres         +
           |          |          |             |             | postgres=CTc/postgres
(4 rows)
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 35 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

5.2.3 Configuring PostgreSQL for COTSon connection:

Once the database is ready, the user needs to configure it, in order to allow incoming connections from COTSon. To do so the user (still as 'postgres' user is ok) has to modify the following file:

```
/etc/postgresql/*/main/pg_hba.conf
```

Becoming root, then the user can change the file adding the lines highlighted below:

```
# TYPE  DATABASE    USER        CIDR-ADDRESS        METHOD


# "local" is for Unix domain socket connections only
local   all         all                             ident
# IPv4 local connections:
host    cotson      cotson              127.0.0.1/32    trust # add this line in this place
host    all         all         127.0.0.1/32        md5
# IPv6 local connections:
host    cotson      cotson          ::1/128                 trust # add this line in this place
host    all         all         ::1/128             md5
```

Then the last thing to do is to restart the PostgreSQL server. Still as a root issue the command:

```
$ /etc/init.d/postgresql restart
```

Finally:

```
$ psql -d cotson -U postgres -c "GRANT ALL PRIVILEGES ON DATABASE cotson
TO cotson;"
$ psql -d cotson -U postgres -c "ALTER USER cotson WITH PASSWORD
'cotson';"
```

At this point the user can press two times the "Ctrl-D" to exit the postgres user shell and the root shell.

5.2.4 Creating the PostgreSQL COTSon db schema:

Then, there is need for creating the database structure using the file "experiment_definition" in the 'src/tools/' directory.

```
$ cd src/tools
```

We modify for example add the following line at the end of the file, instead of:

```
INSERT INTO experiments(experiment_id, text) VALUES(1,'test');
```

We can write:

```
INSERT INTO experiments(experiment_id, description) VALUES(1,'T1');
```

Then we can enter again the DB with:

```
$ psql -h localhost -d cotson -U cotson
```

At the prompt, provide the password 'cotson'

To setup the database schema:

```
Postgres=# \i experiments_definition
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 36 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

This should output:

```
Password for user cotson:
psql (9.1.13)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.

cotson=> \i experiments_definition
CREATE SEQUENCE
psql:experiments_definition:7: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "non_repeated_experiments" for table "experiments"
CREATE TABLE
psql:experiments_definition:16: NOTICE:  CREATE TABLE / UNIQUE will create implicit index "non_repeated_options" for table "parameters"
CREATE TABLE
CREATE SEQUENCE
psql:experiments_definition:30: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "non_repeated_heart_ids" for table "heartbeats"
psql:experiments_definition:30: NOTICE:  CREATE TABLE / UNIQUE will create implicit index "non_repeated_heart_per_machine" for table "heartbeats"
CREATE TABLE
CREATE INDEX
CREATE INDEX
CREATE SEQUENCE
psql:experiments_definition:40: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "non_repeated_metric_ids" for table "metric_names"
CREATE TABLE
psql:experiments_definition:48: NOTICE:  CREATE TABLE / UNIQUE will create implicit index "non_repeated_metric_per_heart" for table "metrics"
CREATE TABLE
CREATE INDEX
INSERT 0 1
cotson=>
```

Then we return to the shell with "Ctrl-D".

5.2.5   Modifying the ".in" file to save our heartbeats in PostgreSQL:

At this point the configuration phase is completed. To check that this works, we can modify the sqlite.in example as follows:

```
$ cd src/examples
$ cp sqlite.in pgsql.in
```

Then we can modify the file "pgsql.in", by changing the heartbeat type from "sqlite" to "pgsql" and setup the "dbconn=…" line as shown below:

```
        heartbeat = {
                type="pgsql",
                dbconn="host=localhost dbname=cotson user=cotson password=cotson",
                experiment_id=EXP,
                experiment_description="T1"
        },}
```

5.2.6   Running COTSon with PostgreSQL

Now, the user is ready to run a complete experiment on COTSon and stores the collected statistics in the PostgreSQL database server.

```
$ ../../bin/cotson pgsql.in
```

The user should be aware that using PostgreSQL server on the same machine can be painful slow. As a rule of thumb, the user should expect that flat files are the fastest way to save your data, there is SQLite server as a middle speed solution, while PostgreSQL server (on the same machine) is the slowest option.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 37 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

6 Simple Examples

All the examples that will be refer to, can be found in the following path:

```
cotson/src/examples
```

A more complete verification test can be launched by typing the following command:

```
$ make run
```

In this case, several examples contained in the example folder are sequentially executed. Following this verification procedure, the reader can see different examples executing, each of them targeting a specific feature of the simulator.

From this folder, the user can also run a specific example that have been setup through the Makefile, by typing the following command:

```
$ make run_name_of_the_example
```

Where the string `name_of_the_example` identifies the file name associated to the example (type "ls *.in" to see names of possible examples. E.g., for running the "functional.in" example type:

```
$ make run_functional
```

6.1 Functional Simulation example (functional.in)

As said in the first part of the guide, a functional simulation doesn't use timing at all. For this reason it is very fast but assuming an ideal ("CPI=1" timing model). Here, the Lua file *functional.in* (see Fig. 12 below) that will be used.

```
one_node_script='functional'
display=os.getenv("DISPLAY")

simnow.commands=function()
    use_bsd('1p.bsd')
    use_hdd('karmic64.img')
    set_journal()
end
```

Fig. 12 – Lua configuration file for running a pure functional simulation with COTSon.

6.1.1 Goal of the experiment or example

As can be seen in the previous figure, in the script there is the option "*one_node_script=...*" that tells COTSon to refer to a template "functional", which contains default options for running a functional simulation. The second line of the code is needed to display the SimNow Graphical User Interface. Then, there are the SimNow commands that allow the user to choose the bsd and hdd by inserting their absolute paths or otherwise by placing the desired bsd and hdd in the directory cotson/data.

6.1.2 Location of the involved files

All the files needed to run the example are contained in the following folder:

```
$COTSONHOME/src/examples
```

Where $**COTSONHOME** is an environment variable identifying the installation path of the COTSon simulator.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 38 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

6.1.3 Detailed instructions to start

To run one example, move on the following folder and launch the simulator:

```
$ cd src/examples
$ make run_functional
```

To start the simulation it is necessary to press the start button (circled in red in Fig. 13 – see subsection 6.1.4). At this point the simulation has started and the prompt of the guest (emulated) machine can be used.

6.1.4 Expected output

After launching the application the graphical user interface should appear as follows:
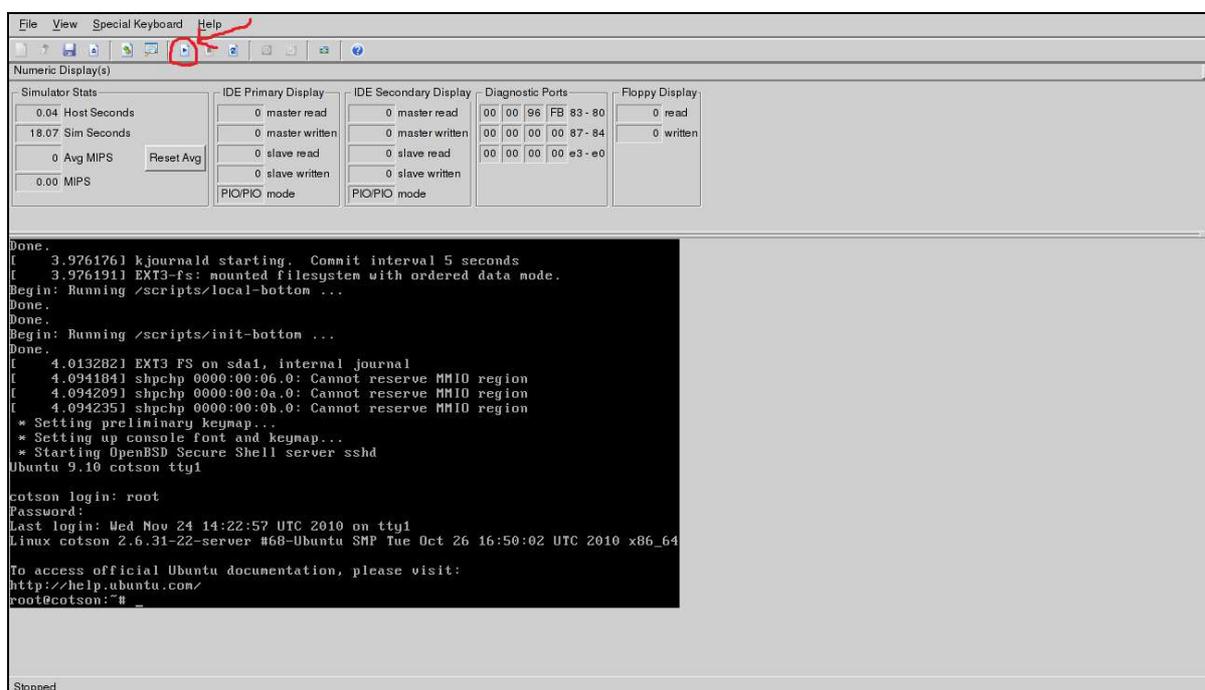
Fig. 13 – Expected output for the "functional.in" example

6.2 Memory tracing example (mem_tracer.in)

To analyze in detail the performance of a system, it is often useful to record a trace of the references that are flowing through the system. This is supported in COTSon through the "tracers". In the "mem_tracer.in" example we can see how to setup a tracer.

```
mem=Memory{ name="main", latency=150 }
trace=Tracer{ name="trace", trace_file=TRACE_OUT, next=mem }
bus=Bus{ name="bus", protocol='MOESI', latency=25, bandwidth=4, next=trace }
busT=Bus{ name="tlb_bus", protocol='MOESI', latency=25, bandwidth=4, next=mem }
```

Fig. 14 – Relevant lines of the Lua configuration file for the memory tracer example. In this case the lua script contains another variable (not shown here) that sets TRACE_FILE="/tmp/mem_tracer.txt.gz"

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 39 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

6.2.1   Goal of the experiment or example

Memory tracing is achieved by placing a transparent object that intercepts every memory request and dumps this information to a file for further analysis. This is how it is specified in the example **mem_tracer.in**. The "*trace=*" option inside the *build* function specifies to intercept every access to the main memory. The tracer is not only limited to the main memory, it is also possible to intercept a request to any memory unit in the memory hierarchy. Simply placing the tracer before L2 or L1 cache, it is possible to intercept every access to the respective cache. A memory tracer is added to the memory hierarchy through the line (see also Fig. 14):

```
trace=Tracer{ name="…", trace_file="…", next="…"}
```

The tracer is defined inside the *build* function of the Lua configuration script. Its parameters must be defined in a Lua table called *Tracer*. This table has three fields: (i) the field *name* specifies the name of the "tracer object", (ii) the field *trace_file* specifies the file where the trace output is dumped, and (iii) the field *next* specifies the name of the memory unit whose access is intercepted by the tracer. As mentioned above, this type of objects can be placed in any position of the memory hierarchy to trace different hardware blocks. In the example *mem_tracer.in* it is placed just before the main memory (setting next=mem), so it will record each memory access in a file, specified by writing:

```
trace_file='path_of_the_file'
```

The output of the tracer is a gzip compressed text file. A line in the output corresponds to a single memory access where each line is composed of five fields. The first field is a time-stamp of the access, the second field indicates the access type, i.e., 'r' for read and 'w' for write, the third and fourth fields indicate the physical and virtual addresses, respectively; finally, the fifth field specifies from the cpu where the access is originated and the type of transactions generated at each level of the memory hierarchy (see Fig. 15).

6.2.2   Location of the involved files

All the files needed to run the example are contained in the following folder:

```
$COTSONHOME/src/examples
```

Where $**COTSONHOME** is an environment variable identifying the installation path of the COTSon simulator.

6.2.3   Detailed instructions to start

 To run the example, move on the example folder and then run the example as follows:

```
$ cd src/examples
$ make run_mem_tracer
```

6.2.4   Expected output

After launching the application the following trace is produced by the program, and displayed on the host shell:

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 40 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Fig. 15 – Expected output for the memory trace simulation with COTSon simulator.

The same result can be found in the host file:

```
/tmp/mem_tracer.txt.gz
```

As can be seen in the Lua configuration file *mem_tracer.in*, the chosen sampler is of type interval, meaning that a timing simulation is done after fixed intervals of time, and has a fixed duration (more details on samplers are in Section 6.3). During the simulation, for each sample the time elapsed from the beginning of the simulation and the calculated IPC are printed on the shell screen (see below).

Modification to the sampling policy is available in the examples *trace_stats.in* and *mem_tracer2.in.* Here, the traces are obtained by changing the type of the CPU's timer (see Fig. 16) and setting TRACE_OUT='/tmp/mem_tracer.txt.gz'.

```lua
while i < cpus() do
    -- we assign a timer that dumps all instructions
    -- and prints stats of them at the end. The traces
    -- are stored in /tmp
    get_cpu(i):timer{ name='cpu'..i, type="trace_stats",
    trace_file=TRACE_OUT(i) }
    i=i+1
end
```

Fig. 16 – Lua configuration file for setting the timer to trace_stats.in example

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 41 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

The trace_stats is in this case a "fake" CPU timer (see './abaeterno/timer_cpu/trace_stats.cpp' for more details) that prints some trace statistics in the specified file. The output on the host screen in this case is:

```
TIME=0.03125 ms IPC ( 1 )
TIME=0.0625 ms IPC ( 1 )
TIME=0.09375 ms IPC ( 1 )
TIME=0.125 ms IPC ( 1 )
TIME=0.15625 ms IPC ( 1 )
TIME=0.1875 ms IPC ( 1 )
TIME=0.21875 ms IPC ( 1 )
MAX NANOS: 218750
```

While the trace file shows a detailed disassembly of the instructions:

```
0e4d8000 000000000101268c (02) 6a 4e                     PUSH 0x4e
         store   0xffffffff81775e50 0x0000000001775e50 (8)
0e4d8000 000000000101268e (02) eb 0a                     JMP 0x101269a
0e4d8000 000000000101269a (05) e9 a1030000               JMP 0x1012a40
0e4d8000 0000000001012a40 (05) 48 830424 80              ADD QWORD [RSP], -0x80
         load    0xffffffff81775e50 0x0000000001775e50 (8)
         store   0xffffffff81775e50 0x0000000001775e50 (8)
0e4d8000 0000000001012a45 (04) 48 83ec 50                SUB RSP, 0x50
0e4d8000 0000000001012a49 (05) e8 02f4ffff               CALL 0x1011e50
         store   0xffffffff81775df8 0x0000000001775df8 (8)
0e4d8000 0000000001011e50 (01) fc                        CLD
0e4d8000 0000000001011e51 (05) 48 897c24 50              MOV [RSP+0x50], RDI
         store   0xffffffff81775e48 0x0000000001775e48 (8)
0e4d8000 0000000001011e56 (05) 48 897424 48              MOV [RSP+0x48], RSI
         store   0xffffffff81775e40 0x0000000001775e40 (8)
0e4d8000 0000000001011e5b (05) 48 895424 40              MOV [RSP+0x40], RDX
         store   0xffffffff81775e38 0x0000000001775e38 (8)
0e4d8000 0000000001011e60 (05) 48 894c24 38              MOV [RSP+0x38], RCX
         store   0xffffffff81775e30 0x0000000001775e30 (8)
0e4d8000 0000000001011e65 (05) 48 894424 30              MOV [RSP+0x30], RAX
         store   0xffffffff81775e28 0x0000000001775e28 (8)
0e4d8000 0000000001011e6a (05) 4c 894424 28              MOV [RSP+0x28], R8
         store   0xffffffff81775e20 0x0000000001775e20 (8)
0e4d8000 0000000001011e6f (05) 4c 894c24 20              MOV [RSP+0x20], R9
         store   0xffffffff81775e18 0x0000000001775e18 (8)
0e4d8000 0000000001011e74 (05) 4c 895424 18              MOV [RSP+0x18], R10
         store   0xffffffff81775e10 0x0000000001775e10 (8)
0e4d8000 0000000001011e79 (05) 4c 895c24 10              MOV [RSP+0x10], R11
         store   0xffffffff81775e08 0x0000000001775e08 (8)
0e4d8000 0000000001011e7e (05) 48 8d7c24 e0              LEA RDI, [RSP-0x20]
0e4d8000 0000000001011e83 (05) 48 896c24 08              MOV [RSP+0x8], RBP
         store   0xffffffff81775e00 0x0000000001775e00 (8)
0e4d8000 0000000001011e88 (05) 48 8d6c24 08              LEA RBP, [RSP+0x8]
0e4d8000 0000000001011e8d (10) f787 88000000 03000000    TEST DWORD [RDI+0x88], 0x3
         load    0xffffffff81775e60 0x0000000001775e60 (4)
0e4d8000 0000000001011e97 (02) 74 06                     JZ 0x1011e9f
```

In the case of *mem_tracer2.in* example (see Fig. 17 below) the "fake" timer is "memtracer (see './abaeterno/timer_cpu/memory_tracer.cpp for more details).

```lua
while i < cpus() do
    cpu=get_cpu(i)
    cpu:timer{name='cpu'..i, type="memtracer",
        tracefile=TRACE_OUT,
        shared="false",
        binary="false",
        size="16MB",line_size=64,num_sets=8 }
    i=i+1
end
```

Fig. 17 - Lua configuration file for setting the timer to mem_tracer2.in example.

The output on the screen is:

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 42 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

```
TIME=93.4 ms IPC ( 1 1 1 1 )
TIME=96.7333 ms IPC ( 1 1 1 1 )
TIME=100.067 ms IPC ( 1 1 1 1 )
TIME=103.4 ms IPC ( 1 1 1 1 )
TIME=106.733 ms IPC ( 1 1 1 1 )
TIME=110.067 ms IPC ( 1 1 1 1 )
TIME=113.4 ms IPC ( 1 1 1 1 )
TIME=116.733 ms IPC ( 1 1 1 1 )
TIME=120.067 ms IPC ( 1 1 1 1 )
TIME=123.4 ms IPC ( 1 1 1 1 )
TIME=126.733 ms IPC ( 1.00017 1 1 1 )
<<< cpu 0 now=126733321 instrs=5041 cycles=5008 ld=1278 mr=82 st=612 mw=0 fl=0
<<< cpu 1 now=126733321 instrs=0 cycles=5008 ld=0 mr=0 st=0 mw=0 fl=0
<<< cpu 2 now=126733321 instrs=0 cycles=5008 ld=0 mr=0 st=0 mw=0 fl=0
<<< cpu 3 now=126733321 instrs=0 cycles=5008 ld=0 mr=0 st=0 mw=0 fl=0
TIME=130.067 ms IPC ( 1 1 1 1 )
TIME=133.4 ms IPC ( 1 1 1 1 )
TIME=136.733 ms IPC ( 1 1 1 1 )
TIME=140.067 ms IPC ( 1 1 1 1 )
TIME=143.4 ms IPC ( 1 1 1 1 )
TIME=146.733 ms IPC ( 1 1 1 1 )
TIME=150.067 ms IPC ( 1 1 1 1 )
TIME=153.4 ms IPC ( 1 1 1 1 )
TIME=156.733 ms IPC ( 0.940259 1 1 1 )
<<< cpu 0 now=156733318 instrs=12195 cycles=24793 ld=2506 mr=21 st=1295 mw=0 fl=0
<<< cpu 1 now=156733318 instrs=0 cycles=24793 ld=0 mr=0 st=0 mw=0 fl=0
<<< cpu 2 now=156733318 instrs=0 cycles=24793 ld=0 mr=0 st=0 mw=0 fl=0
<<< cpu 3 now=156733318 instrs=0 cycles=24793 ld=0 mr=0 st=0 mw=0 fl=0
TIME=160.067 ms IPC ( 1 1 1 1 )
TIME=163.4 ms IPC ( 1 1 1 1 )
MAX SAMPLES: 50
```

And the content of trace file is:

```
12591 r 0x0000000001797180 0x000000000D853000 [0]
12595 r 0x00000000018217C0 0x000000000D853000 [0]
12599 r 0x0000000001956D40 0x000000000D853000 [0]
12604 r 0x00000000017A2300 0x000000000D853000 [0]
12606 r 0x00000000019F3E80 0x000000000D853000 [0]
12607 r 0x0000000001911540 0x000000000D853000 [0]
12608 r 0x00000000FED000C0 0x000000000D853000 [0]
12612 r 0x00000000017A2380 0x000000000D853000 [0]
12613 r 0x00000000017A2340 0x000000000D853000 [0]
12655 r 0x0000000001821780 0x000000000D853000 [0]
12683 r 0x00000000019F3E40 0x000000000D853000 [0]
12725 r 0x0000000001820E80 0x000000000D853000 [0]
12732 r 0x00000000019FCB00 0x000000000D853000 [0]
12733 r 0x00000000019FCB40 0x000000000D853000 [0]
12740 r 0x0000000001A06240 0x000000000D853000 [0]
12769 r 0x0000000001A003C0 0x000000000D853000 [0]
12779 r 0x0000000001A002C0 0x000000000D853000 [0]
12780 r 0x0000000001912E80 0x000000000D853000 [0]
12847 r 0x0000000001956F40 0x000000000D853000 [0]
```

The values in this case represent in order: i) the number of nanoseconds (timestamp), ii) he type of operation (r for read, w for write), iii) the address involved, iv) the content of the x86 CR3 register, and v) the cpu identifier.

6.2.5   Defining the Region Of Interest (ROI)

Although the discussion of how to setup a the Region Of Interest is presented as part of a tracer example, the technique is general and serves to measure metrics related to the portion of the code that is marked by the user.

COTSon comes with the capability of timing simulation of a specific part of a benchmark, hereafter referred to as *Region Of Interest* (ROI). Currently this is achieved in two ways, the first one is to enable the timing just before the benchmark starts and to disable it right after the benchmark finishes. This approach considers the whole benchmark as the ROI. The second approach is to mark a portion

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 43 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

of the benchmark for which a timing simulation is required. A practical example of the first approach is provided inside *src/examples/tracer/* (see `Fig. 18`).

```
options = {
...
        sampler={ type="selective_tracing",quantity="2",
        constructor="sampler_constructor",changer="zone_changer" },
...
}

-- the function that decides what timer to use
function sampler_constructor(i)
        if i == 0 then return {type="no_timing", quantum="1M"} end
        if i == 1 then return {type="simple", quantum="1M"} end
end

-- the function that decides the zone sequence
function zone_changer(start,i)
        if start then -- entering zone i
        print("### entering zone " .. i)
                return 1
        end
        if not start then -- leaving zone i
        print("### leaving zone " .. i)
                return 0
        end
end
```

Fig. 18 – The definition of the ROI in the example cotson_tracer.in

To achieve this, the sampler to be used must be of type "*selective_tracing*", which in essence is a collection of other samplers, each of which is used when a certain condition is met during the entire simulation. For the specific scenario, the selective sampler is composed of two samplers: *no_timing* and *simple*. In this case, the simulation runs in a timing mode or in functional mode until a certain trigger is given by the application (see below), then another trigger stops the timing simulation, therefore freezing the timing statistics update.

The configuration file *cotson_tracer.in* (`Fig. 18`) is an example, which shows how these parameters are specified. *run.sh* is the script that executes inside SimNow (since it is specified by the "execute('run.sh')" simnow.commands function) and it contains specific commands (or "triggers") to mark the start and the end of the timing simulation. This requires that the selected hard-disk image (hdd) provides the 'cotson_tracer' executable (this is the case for the "karmic64,img" hdd that comes by default with COTSon) essentially, the *cotson_tracer* is an helper program that takes three arguments and is supposed to be used inside the execution script as in the following format:

```
cotson_tracer 10 1 0
./benchmark
cotson_tracer 10 1 1
```

The first argument specifies the type of the sampler used, number 10 is reserved for *selective_tracing*. The second argument is an integer value used as an identification of the simulation zone for which timing simulation is enable/disabled (in this case this indicates "Zone 1"). Finally, the third argument is a switch to enable/disable the timing simulation. Hence, cotson_tracer 10 1 0 implies that timing is enabled for zone 1 and cotson_tracer 10 1 1 implies that timing is disabled for zone 1.

A finer grain control is possible too. In this case, the steps are the following:

i)   The user as to include the "cotson_tracer.h" header provided in the src/example/tracer directory;

ii) The user can then mark the portion of code of interest (ROI) with a COTSON_INTERNAL(10,1,0) to start the timing simulation for "Zone 1" and COTSON_INTERNAL(10,1,0) to stop the timing simulation for "Zone 1";

Note that, in this case, it is not necessary to have the "cotson_tracer" helper program in the hdd image.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 44 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

6.3 Samplers: timing simulation

There are several types of samplers available (check their implementations in the folder cotson/src/abaeterno/sampler). Here we discuss more details about the following four samplers:

- **simple**: timing simulation is always on. For example this type of sampler is used in the example configuration *one_cpu_simple.in*;
- **interval**: the duration of each phase (state) of the sampling (functional, warming, simulation) is fixed. This type of sampler is used in the example configuration in *multiple_cpu_interval.in*;
- **dynamic**: the sample intervals are determined dynamically by the sampler according to the variation of a monitored variable This type of sampler is used in the example configuration in *dynamic.in*;
- **SMARTS:** the duration of each phase (state) of the sampling (functional, warming, simulation) is fixed, but the sampling instants are determined by a previous profiling phase. This type of sampler is used in the example configuration in *smarts.in*;

To specify the full timing simulation the lua file contains the following (see file *one_cpu_simple.in*):

```
sampler={ type="simple", quantum="100k" }, -- quantum is in cycles
```

To specify the interval based simulation, where the execution takes systematically a given amount of time for the functional, warming and timing simulation, the lua file contains the following (see file *multiple_cpu_interval.in*):

```
sampler={ type="interval", functional="1M", warming="100k", simulation="100k", },
            -- the sampler will execute warming, simulation and then functional for
            -- their respective interval lengths. After the first simulation sample,
            -- though it will finish (due to max_samples being 1)
```

To specify the interval based simulation, where the execution takes systematically a given amount of time for the functional, warming and timing simulation, the lua file contains the following (see file *smarts.in*); this is similar to the "interval sampling" but in this case a profiling phase is also required":

```
sampler={ type="smarts", functional="100k", warming="100k", simulation="100k", },
             -- the sampler will execute warming, simulation and then functional for
             -- their respective interval lengths until reaching 1M nanos
```

To specify the dynamic based simulation, where the execution is switched to full timing according to phases that are detected through an "non-timing" variable (in this case the variable is the number of exceptions on any cpu simulated), the lua file contains the following (see file *dynamic.in*):

```
sampler={ type="dynamic", functional="100k", warming="100k", simulation="100k",
             maxfunctional=10, sensitivity="90",
             variable={"cpu.*.other_exceptions"}, },
           -- the sampler will execute warming, simulation and then functional for
           -- their respective interval lengths until reaching 1M nanos
```

The length of the intervals, where functional, warming, full-timing simulation is performed, is specified in a way similar to the interval simulation. If the first-derivative of this variable goes beyond the sensitivity (set by the line *sensitivity="90"*) there is a phase change in the program and so a timing simulation can start. The variable *maxfunctional="10"* is needed to set the maximum number of time intervals passed in the functional state before a new timing simulation starts. This type of sampler is used in dynamic.in. As you can see from Fig. 20 the intervals between the printed values of time are not regular but they are variable.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 45 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

6.3.1   Goal of the experiment or example

The main purpose of the example is the illustration of the use of different sampler.

6.3.2   Location of the involved files

All the files needed to run the example are contained in the following folder:

```
$COTSONHOME/src/examples
```

where $**COTSONHOME** is an environment variable identifying the installation path of the COTSon simulator.

6.3.3   Detailed instructions to start for NO Sampling ("simple")

To run the example, move on the example folder and then run the example as follows:

```
$ cd src/examples
$ make run_one_cpu_simple.in
```

6.3.4   Expected output for NO Sampling ("simple")

After launching the application the following output should be obtained (see Fig. 21). In this case, the timing simulation is always on:
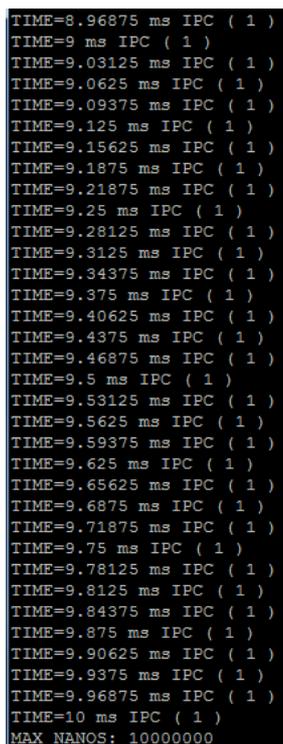

```
TIME=8.96875 ms IPC ( 1 )
TIME=9 ms IPC ( 1 )
TIME=9.03125 ms IPC ( 1 )
TIME=9.0625 ms IPC ( 1 )
TIME=9.09375 ms IPC ( 1 )
TIME=9.125 ms IPC ( 1 )
TIME=9.15625 ms IPC ( 1 )
TIME=9.1875 ms IPC ( 1 )
TIME=9.21875 ms IPC ( 1 )
TIME=9.25 ms IPC ( 1 )
TIME=9.28125 ms IPC ( 1 )
TIME=9.3125 ms IPC ( 1 )
TIME=9.34375 ms IPC ( 1 )
TIME=9.375 ms IPC ( 1 )
TIME=9.40625 ms IPC ( 1 )
TIME=9.4375 ms IPC ( 1 )
TIME=9.46875 ms IPC ( 1 )
TIME=9.5 ms IPC ( 1 )
TIME=9.53125 ms IPC ( 1 )
TIME=9.5625 ms IPC ( 1 )
TIME=9.59375 ms IPC ( 1 )
TIME=9.625 ms IPC ( 1 )
TIME=9.65625 ms IPC ( 1 )
TIME=9.6875 ms IPC ( 1 )
TIME=9.71875 ms IPC ( 1 )
TIME=9.75 ms IPC ( 1 )
TIME=9.78125 ms IPC ( 1 )
TIME=9.8125 ms IPC ( 1 )
TIME=9.84375 ms IPC ( 1 )
TIME=9.875 ms IPC ( 1 )
TIME=9.90625 ms IPC ( 1 )
TIME=9.9375 ms IPC ( 1 )
TIME=9.96875 ms IPC ( 1 )
TIME=10 ms IPC ( 1 )
MAX NANOS: 10000000
```

Fig. 19 – Expected output for "simple" sampler example. The example is based on the one_cpu_simple.in Lua configuration file.

6.3.5   Detailed instructions to start for Dynamic Sampling

To run the example, move on the example folder and then run the example as follows:

```
$ cd src/examples
$ make run_dynamic
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 46 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

6.3.6 Expected output for Dynamic Sampling

After launching the application the following output should be obtained (see Fig. 20):


```
TIME=0.09375 ms IPC ( 0.868536 )
TIME=0.21875 ms IPC ( 0.937296 )
TIME=0.625 ms IPC ( 1.01141 )
TIME=0.71875 ms IPC ( 1.05867 )
TIME=0.8125 ms IPC ( 1.25066 )
TIME=1.1875 ms IPC ( 0.607328 )
TIME=1.59375 ms IPC ( 1 )
TIME=2 ms IPC ( 1 )
TIME=2.40625 ms IPC ( 1 )
TIME=2.5625 ms IPC ( 1 )
TIME=2.6875 ms IPC ( 1 )
TIME=3.09375 ms IPC ( 1 )
TIME=3.5 ms IPC ( 1 )
TIME=3.6875 ms IPC ( 1 )
TIME=3.78125 ms IPC ( 1 )
TIME=4.1875 ms IPC ( 1 )
TIME=4.59375 ms IPC ( 1 )
TIME=5 ms IPC ( 1 )
TIME=5.1875 ms IPC ( 0.271158 )
TIME=5.59375 ms IPC ( 1 )
TIME=6 ms IPC ( 1 )
TIME=6.28125 ms IPC ( 1 )
TIME=6.375 ms IPC ( 1 )
TIME=6.78125 ms IPC ( 1 )
TIME=7.1875 ms IPC ( 1 )
TIME=7.3125 ms IPC ( 0.0359127 )
TIME=7.71875 ms IPC ( 1 )
TIME=8.125 ms IPC ( 1 )
TIME=8.28125 ms IPC ( 1 )
TIME=8.375 ms IPC ( 1 )
TIME=8.78125 ms IPC ( 1 )
TIME=9.1875 ms IPC ( 1 )
TIME=9.59375 ms IPC ( 1 )
TIME=10 ms IPC ( 1 )
MAX NANOS: 10000000
```

Fig. 20 – Expected output for dynamic sampler example. The example is based on the dynamic.in Lua configuration file.

6.3.7 Detailed instructions to start for Interval Sampling

To run the example, move on the example folder and then run the example as follows:

```
$ cd src/examples
$ make run_multiple_cpu_interval
```

6.3.8 Expected output for Interval Sampling

After launching the application the following output should be obtained (see Fig. 21). As a variant, in this case 4 CPUs are simulated, the simulation is fast-forwarded for 2 second and then the next 50 ms are simulated with full timing but up to 5 samples that are taken at successive regular instants:


```
Fast forward from 20000000000 ns to 20500000000 ns
Fast forward ended at 20500000000 ns
TIME=0.033333 ms IPC ( 0.127218 0.128478 0.13195 0.131923 )
TIME=0.433332 ms IPC ( 0.105847 0.10437 0.112398 0.110417 )
TIME=0.833331 ms IPC ( 0.0806739 0.078299 0.0855777 0.0838514 )
TIME=1.23333 ms IPC ( 0.191664 0.179845 0.209287 0.20506 )
TIME=1.63333 ms IPC ( 0.210793 0.199434 0.229783 0.234864 )
MAX SAMPLES: 5
```

Fig. 21 – Expected output for interval based sampler example. The example is based on the multiple_cpu_interval.in Lua configuration file.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 47 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

6.3.9 Detailed instructions to start for SMARTS Sampling

To run the example, move on the example folder and then run the example as follows:

```
$ cd src/examples
$ make run_smarts
```

6.3.10 Expected output for SMARTS Sampling

After launching the application the following output should be obtained (see Fig. 21). In this case, similarly to the dynamic sampling, the sampling instant are not uniformly distributed with the time:

```
TIME=6.96875 ms IPC ( 1 )
TIME=7.0625 ms IPC ( 1 )
TIME=7.15625 ms IPC ( 1 )
TIME=7.25 ms IPC ( 1 )
TIME=7.34375 ms IPC ( 0.0460997 )
TIME=7.4375 ms IPC ( 0.0147858 )
TIME=7.53125 ms IPC ( 0.811341 )
TIME=7.625 ms IPC ( 0.996047 )
TIME=7.71875 ms IPC ( 1 )
TIME=7.8125 ms IPC ( 1 )
TIME=7.90625 ms IPC ( 1 )
TIME=8 ms IPC ( 1 )
TIME=8.09375 ms IPC ( 1 )
TIME=8.1875 ms IPC ( 1 )
TIME=8.28125 ms IPC ( 1 )
TIME=8.375 ms IPC ( 1 )
TIME=8.46875 ms IPC ( 1 )
TIME=8.5625 ms IPC ( 1 )
TIME=8.65625 ms IPC ( 1 )
TIME=8.75 ms IPC ( 1 )
TIME=8.84375 ms IPC ( 1 )
TIME=8.9375 ms IPC ( 1 )
TIME=9.03125 ms IPC ( 1 )
TIME=9.125 ms IPC ( 1 )
TIME=9.21875 ms IPC ( 1 )
TIME=9.3125 ms IPC ( 1 )
TIME=9.40625 ms IPC ( 1 )
TIME=9.5 ms IPC ( 1 )
TIME=9.59375 ms IPC ( 1 )
TIME=9.6875 ms IPC ( 1 )
TIME=9.78125 ms IPC ( 1 )
TIME=9.875 ms IPC ( 1 )
TIME=9.96875 ms IPC ( 1 )
MAX NANOS: 10000000
```

Fig. 22 – Expected output for SMARTS sampler example. The example is based on the smarts.in Lua configuration file.

6.4 Simulation of Ethernet connected clusters

A cluster is a set of loosely coupled computers that work together as if they were a single computer. COTSon has the capability of simulating clusters that are interconnected through an Ethernet based network card and through a simulated switch (called "mediator") by using an individual full-system instance of SimNow for each node. It is worth of notice that the SimNow instance run in parallel if the simulation host has enough cores.

6.4.1 Goal of the experiment or example

When simulating a cluster with COTSon there is a software component that is needed to connect all the SimNow instances of the different COTSon nodes, called *Mediator* (i.e., a component in the

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 48 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

simulator architecture that is responsible to manage the network communication among different nodes of the simulated system – see also Fig. 4). This application, together with other external tools such as *Slirp*, allows more than one COTSon node (i.e., an instance of SimNow plus abaeterno) to communicate with the rest of the network. COTSon is responsible for coordinating the activity of the nodes, which are possibly running in different machines. The simplest example about clusters is ***twonodes.in*** that implements a cluster of two nodes pinging each other.

6.4.2 Location of the involved files

All the files needed to run the example are contained in the following folder:

```
$COTSONHOME/src/examples
```

Where $**COTSONHOME** is an environment variable identifying the installation path of the COTSon simulator.

6.4.3 Detailed instructions to start

To run the example, move on the example folder and then run the example as follows:

```
$ cd src/examples
$ make run_twonodes
```

6.4.4 Expected output

After launching the application the following output should be obtained (see Fig. 23):

Fig. 23 expected output for the example where mediator component is used. The example is based on the twonodes.in Lua configuration file.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 49 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

While the simulation is running, the following windows (see Fig. 24) should appear on the screen indicating that the two nodes have been booted up and they are communicating each other:
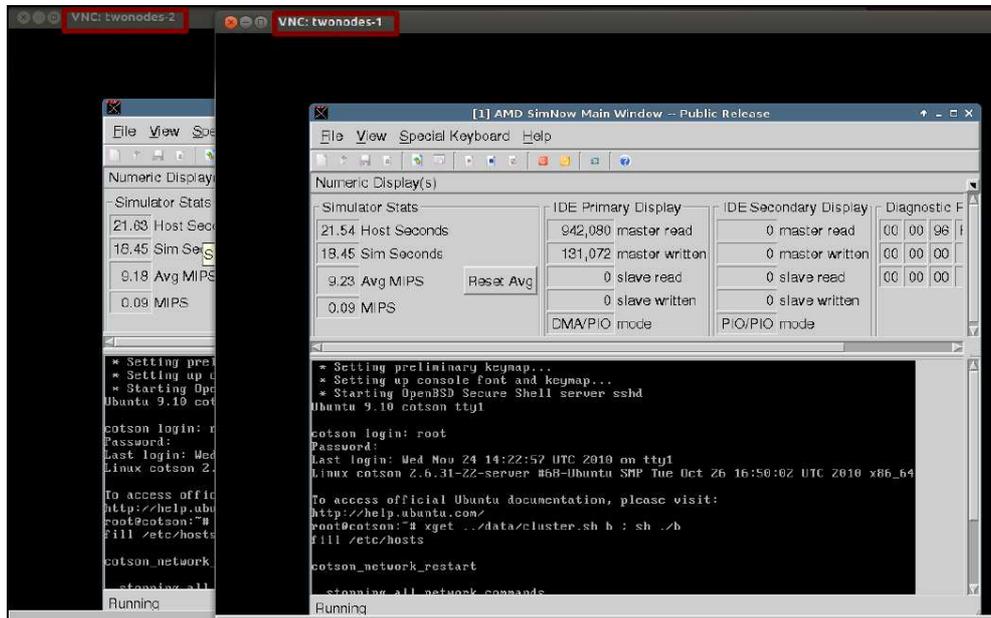
Fig. 24– Two simulator windows are used to manage the two communicating nodes of the simulated system.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 50 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

7 Research Use Case from BSC

This section shows how to use the TERAFLUX system image and benchmark repository that has been put in place to ensure partners use a common development platform and can reproduce each other's results.

7.1 Goal of the experiment or example

The goal is two-fold: to show how the system image can be used for development, and show how experiments from the benchmark repository can be run.

7.2 Location of the involved files

First of all, one must download the system image and verify its integrity by downloading files

```
wget http://www.teraflux.eu/sites/teraflux.eu/files/teraflux-v5.img.bz2
```

Then:

```
wget http://www.teraflux.eu/sites/teraflux.eu/files/teraflux-v5.img.bz2.md5
```

Then executing:

```
$ md5sum -c teraflux-v5.img.bz2.md5
teraflux-v5.img.bz2: OK
$ bzip2 -d teraflux-v5.img.bz2
```

Next, one must download the *Teraflux Simulation Manager* (*tfsm*), a simple script to help using the image:

```
$ svn co https://teraflux.eu/svn/tfx/tfsm
```

This script requires installing a few packages, as well as support for hardware virtualization in order to provide maximum performance during development and native testing:

```
$ sudo apt-get -y install qemu-kvm libvirt-bin vinagre qemu-system virt-manager gcc-4.4
…
$ sudo adduser `whoami` kvm
$ sudo addgroup libvirt
$ sudo adduser `whoami` libvirt
$ sudo modprobe kvm-amd
```

The benchmark repository is included in the image file, but it can also be independently downloaded:

```
$ svn co https://teraflux.eu/svn/tfx/ems
```

7.3 Detailed instructions to start

To start developing with the image, one must start *tfsm* with the following command:

```
$ ./tfsm/tfsm edit teraflux-v5.img 512 2
```

This will start a virtual machine with 2 cores and 512 MB of memory, ready to use for development and benchmark testing. Once the virtual machine is running, one can start installing programs and developing. Both the login and password are *user*.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 51 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

After the changes are ready, one can launch multiple nodes to test the benchmarks natively. First of all, the maximum number of nodes must be established (2 in this case), and the editable virtual machine must be stopped. The following commands have to be issued at the virtual machine prompt:

```
$ sudo ./guest/nodes 2
$ sudo halt
```

One can then start two identical nodes to run distributed benchmarks natively with *tfsm*:

```
$ ./tfsm/tfsm qemu teraflux-v5.img 2 512 2
Creating inter-node network...
Creating VMs...
You can now connect to the VMs (e.g., 'virt-manager' or 'vinagre :5900')
[Press enter to destroy all Vms]
```

The benchmarks are run with the *Experiment Management System* (*ems*) that is included in the image (this command again can be issued inside the virtual machine):

```
$ cd ems
$ ./ems run kernels/cholesky small
```

7.4 Expected output

```
Running 'kernels/cholesky/smpss' small into kernels/cholesky/smpss//run/1
$ cat kernels/cholesky/smpss/run/1/ems_output
+ cholesky_simple 64 64
25003147;        907
```

Since the experiment is natively run in "*qemu*" mode (using hardware virtualization), the actual contents of the *ems_output* file will change.

7.5 Further references to more in-depths

The *tfsm* script also includes commands to start SimNOW and COTSon nodes. Please refer to the README file in the *tfsm* repository, and the environment-specific details of other partners for more information on the necessary arguments.

The *ems* script also handles benchmark compilation, even though the TERAFLUX disk image comes with pre-compiled benchmarks. Please run *ems* without arguments and read the *README* file in the *ems* repository for more details. To update the benchmark repository in the TERAFLUX disk image run:

```
$ cd ems
$ svn https://teraflux.eu/svn/tfx/ems update
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 52 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

8   Research Use Case from CAPS

This section describe the experimental platform used to evaluate, first, the new CAPS compiler back-end developed during the project, and second the OpenACC dataflow extension, on the common TERAFLUX architecture using the SimNOW virtualization system and the COTSon simulation platform. The experimentation has been performed on a Convolution benchmark programmed in OpenHMPP and offloading the parallel computation on the CPU using a C back-end.

8.1 Goal of the experiment or example

The goal of the experiment is to validate the execution of the OpenHMPP Convolution benchmark on the COTSon system. This experiment will perform a functional validation of a code pre-compiled by the CAPS compiler by the execution of the binary together with the CAPS compiler runtime.

8.2 Location of the involved files

To run the experiment, one has to use the tools implemented by the collaborative effort from UNISI & BSC: the COTSon simulation platform with the associated SimNow virtualization system, and the *Teraflux Simulation Manager* (*tfsm*). The COTSon system is taken from the trunk:

```
$ svn co https://svn.code.sf.net/p/cotson/code/trunk cotson
```

The tfsm is fetched from the original source:

```
$ svn co https://teraflux.eu/svn/tfx/tfsm
```

The other files have been developed at CAPS entreprise using a branch of the CAPS many-core compiler and the access is subject to a formal request to CAPS entreprise:

- *karmic64-capse.img*: the image containing the CAPS compilation framework and the Convolution example, it contains pre-compiled files from the CAPS compiler, and requires only a minimal SDK;

- *CAPSCompilersRuntimes-3.3.4-TF.tar.bz2*: the CAPS compiler run-times for compiling the OpenHMPP applications;

- *CAPSCompilersSDK-3.3.4-TF.tar.bz2*: the CAPS compiler SDK (partial, without the compiler binaries, does not need a license token generator);

- *CAPSCompilersRuntimes-install.sh*: the automatic deployment script;

- *capse.in*: the Lua configuration script running the experiment with timing enabled;

- *capse-interactive.in*: the Lua configuration script running the functional simulator in interactive mode;

8.3 Detailed instructions to start

Deployment

This experiment requires the deployment of the CAPS-compiler run-time, and the recompilation of the Convolution application on a virtual machine image. For that purpose one has to use the "edit" mode of the *tfsm* (see previous section):

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 53 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

```
$ ./tfsm edit karmic64-capse.img 512 2
```

Then, one has to perform a standard installation of the prototype CAPS-compiler run-time and simply builds the Convolution application. Note that these operations are easier to perform when *tfsm* is modified to run QEMU with a tunnel for SSH in port 2222:

```
$ cp tfsm tfsm-capse
<REPLACE the corresponding lines below in tfsm-capse>
cmd_edit () {
  which $QEMU >/dev/null || error "cannot find QEMU: $QEMU"
  sys $QEMU -enable-kvm -hda $IMAGE -m $MEM -smp $NCORES -redir tcp:2222::22
}
```

Doing so, the update process can be automatize using `rsync` and `ssh` commands from the host:

```
$ ./tfsm-capse edit karmic64-capse.img 2048 8 &
$ scp -P 2222 CAPSCompilersRuntimes-install.sh root@localhost:/home/user/CAPSe/
$ scp -P 2222 CAPSCompilersRuntimes-3.3.4-TF.tar.bz2 root@localhost:/home/user/CAPSe/
$ scp -P 2222 CAPSCompilersSDK-3.3.4-TF.tar.bz2 root@localhost:/home/user/CAPSe/
$ ssh -p 2222 root@localhos7 /home/user/CAPSe/CAPSCompilersRuntimes-install.sh
$ ssh -p 2222 root@localhost 'shutdown -h now'
```

On Ubuntu/Debian Linux distributions, the usage of the QEMU virtual machine requires the user to belong to the "kvm" group (as in the previous example of Section 7). Note that in this example, the host machine is called "localhost" and executes the COTSon system. Once the deployment of the CAPS-compiler performed on the COTSon system has been done, the experimental snapshot is prepared using the SimNOW:

```
$ export PATH="$PATH:."; ln -s ../simnow-linux64-4.6.2pub/simnow
$ ./tfsm-capse simnow karmic64-capse.img 4 4p-reset.bsd
```

Note also that the *tfsm* script needs to know the installation location of the SimNow virtualization system (it can be set through the SIMNOW environmental variable). At the end of the boot process, the snapshot is prepared with the appropriate environment (in the console after the login root/root):

```
$ cd /home/user/CAPSe
$ source CAPSMC/bin/capsrt-env.sh
$ cd Convolution
$ make clean && make
```

After the initialization is completed, the user should stop the simulation and save the snapshot under the name "4p-capse.bsd" in the COTSon data directory.

COTSon Simulation

The functional validation is performed using a snapshot containing the CAPS-compiler run-time and the Convolution example ready to run. A very simple Lua configuration script (*capse.in*) is called using the following command:

```
$ ../cotson/bin/cotson capse.in
```

The lua configuration script activates the standard timing of the simulation using the abaeterno library and the "build" function. It also uses the "fastforward" keyword to delay the simulation up to the OpenHMPP kernel execution. The simulation can be switched in visual mode if the appropriate line comments are removed from the Lua configuration script. The core command of the script is the

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 54 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

following:

```
simnow.commands=function()
    use_bsd('4p-capse.bsd')
    use_hdd('karmic64-capse.img')
    set_journal()
    send_keyboard('./convol-hmpp.exe -e 1 data/Michal-*.tif -o ./Michal.tif')
end
```

The functional validation of the computation is done by the comparison of the picture generated by the Convolution execution with a valid reference. The timing result of the simulation is stored in the file "node.1.hmpp_simple.log".

An interactive mode is available with the script "capse-interactive.in", a variant of the previous one activating the functional simulator with the SimNow window enabled. The user has to run the simulator, and then it can interact with the application. An overview of the simulator window is given in Fig. 25.

8.4 Expected output

The deployment and installation output is the following. It must contain a correct compilation of the Convolution example and a proper execution.

```
[laorans@nova18 ~]$ ssh -p 2222 root@localhost /home/user/CAPSe/CAPSCompilersRuntimes-install.sh
root@localhost's password:
Uncompress package
------ Clean Convolution -----
rm -rf    *convolution*_c.hmg    *convolution*_c.hmg.o       *convolution*_c.hmg.rc.o    *convolution*_c.hmg.fatbin
rm -rf
rm -rf src/pictureInterface.o src/mainutils.o src/filters5x5.hmpp.o src/main-hmpp.hmpp.o convol-hmpp.exe src/filters5x5_c.translated.o src/main-hmpp_c.translated.o
rm -rf src/*.hmpp.o src/*.translated.o
rm -rf properties_tune_*.psc
rm -rf *_out.tif out*.tif
rm -rf core.*
rm -rf *.translated.i *.extracted.* *.halt.* *.hdpp.* *.inline.* *.preproc.* *.capstune.i
rm -rf *__hmpp_acc_region__*.o *__hmpp_acc_region__*.fatbin *__hmpp_acc_region__*.hmf
------ Build Convolution -----
gcc -Wall -fopenmp -DHMPP_V3b -DHMPP_OPTIM_2 -DHMPP_C -c -O3 -Isrc/ -o src/pictureInterface.o src/pictureInterface.c
gcc -Wall -fopenmp -DHMPP_V3b -DHMPP_OPTIM_2 -DHMPP_C -c -O3 -Isrc/ -o src/mainutils.o src/mainutils.c
gcc -Wall -fopenmp -DHMPP_V3b -DHMPP_OPTIM_2 -DHMPP_C -c -O3 -Isrc/ -o src/filters5x5.hmpp.o src/filters5x5_c.translated.i
src/filters5x5.c:48: warning: hmppsi_lookup defined but not used
src/filters5x5.c:54: warning: hmppsi_g_convolution_lookup defined but not used
gcc -Wall -fopenmp -DHMPP_V3b -DHMPP_OPTIM_2 -DHMPP_C -c -O3 -Isrc/ -o src/main-hmpp.hmpp.o src/main-hmpp_c.translated.i
src/main-hmpp.c: In function main:
src/main-hmpp.c:49: warning: ignoring #pragma hmpp
src/main-hmpp.c:58: warning: ignoring #pragma hmpp
src/main-hmpp.c:59: warning: ignoring #pragma hmpp
src/main-hmpp.c:60: warning: ignoring #pragma hmpp
src/main-hmpp.c: At top level:
src/main-hmpp.c:108: warning: hmppsi_lookup defined but not used
g++ -c -I/home/user/CAPSe/CAPSMC//include -I/home/user/CAPSe/CAPSMC//include/openacc  -fPIC -o convolution_c.hmg.o convolution_c.hmg.cc
g++ -shared  -fPIC -o convolution_c.hmg convolution_c.hmg.o
gcc -Wall -fopenmp -DHMPP_V3b -DHMPP_OPTIM_2 -DHMPP_C -O3 -o convol-hmpp.exe src/pictureInterface.o src/mainutils.o src/filters5x5.hmpp.o
src/main-hmpp.hmpp.o convolution_c.hmg -lm -ltiff -lz -Wl,-rpath,/home/user/CAPSe/CAPSMC//slib -Wl,-rpath,/home/user/CAPSe/CAPSMC//lib -
L/home/user/CAPSe/CAPSMC//lib -lhmpprti -lhmpprt -lhmpperr -lhmppstr -lhmppos -lhmppabi -lhmppos -lhmpplog -lphmpp -lhmpprl -lopenacci -lopenacc
------ Run Convolution -----
./convol-hmpp.exe -e 1 data/Michal-Osmenda-Mont_Saint_Michel-CC_BY_SA_2.0.tif -o  ./Michal-Osmenda-Mont_Saint_Michel-CC_BY_SA_2.0_out.tif
[    0.056758] ( 0) WARN : Cannot find libOpenCL.so: dlopen() failed: libOpenCL.so: cannot open shared object file: No such file or directory,
disabling OPENCL support.
[    0.058255] ( 0) INFO : --> allocate <convolution> at src/main-hmpp.c:48
[    0.058295] ( 0) INFO :          - Acquire the device 'host#0'
[    0.058362] ( 0) INFO :          - Allocate buffer 'filter5x5_1::heigh|filter5x5_2::heigh' (4 x [] = 4 bytes of host memory on device 'host#0')
[    0.058408] ( 0) INFO :          - Allocate buffer 'filter5x5_1::width|filter5x5_2::width' (4 x [] = 4 bytes of host memory on device 'host#0')
[    0.058440] ( 0) INFO :          - Allocate buffer 'filter5x5_1::inRaster|filter5x5_2::outRaster' (4 x [2793, 1920] = 21450240 bytes of host
memory on device 'host#0')
[    0.058473] ( 0) INFO :          - Allocate buffer 'filter5x5_1::outRaster|filter5x5_2::inRaster' (4 x [2793, 1920] = 21450240 bytes of host
memory on device 'host#0')
[    0.058505] ( 0) INFO : <-- allocate <convolution> at src/main-hmpp.c:48
[    0.058526] ( 0) INFO : --> allocate, data <convolution> at src/main-hmpp.c:50
[    0.058551] ( 0) INFO :          - Allocate mirror 0x4040a0 "stencil1" (4 x [25] = 100 bytes of host memory on device 'host#0')
[    0.058581] ( 0) INFO :          - Allocate mirror 0x404120 "stencil2" (4 x [25] = 100 bytes of host memory on device 'host#0')
[    0.058610] ( 0) INFO : <-- allocate, data <convolution> at src/main-hmpp.c:50
[    0.058633] ( 0) INFO : --> advancedload, data <convolution> at src/main-hmpp.c:54
[    0.058654] ( 0) INFO :          - Upload mirror 0x4040a0 "stencil1" (4 x [25] = 100 bytes to device 'host#0')
[    0.058697] ( 0) INFO :          - Upload mirror 0x404120 "stencil2" (4 x [25] = 100 bytes to device 'host#0')
[    0.058721] ( 0) INFO : <-- advancedload, data <convolution> at src/main-hmpp.c:54
[    0.058743] ( 0) INFO : --> advancedload, args <convolution> at src/main-hmpp.c:57
[    0.058759] ( 0) INFO :          - Bind buffer 'filter5x5_1::heigh|filter5x5_2::heigh' to address 0x7fff69c4afd8 'opt.m_expansion *
BYTES_PER_PIXEL * heigh'
[    0.058775] ( 0) INFO :          - Bind buffer 'filter5x5_1::width|filter5x5_2::width' to address 0x7fff69c4afd4 'width'
[    0.058790] ( 0) INFO :          - Bind buffer 'filter5x5_1::inRaster|filter5x5_2::outRaster' to address 0x2af3ac422010 'raster1'
[    0.058811] ( 0) INFO :          - Upload buffer 'filter5x5_1::heigh|filter5x5_2::heigh' (4 x [] = 4 bytes to device 'host#0')
[    0.058835] ( 0) INFO :          - Upload buffer 'filter5x5_1::width|filter5x5_2::width' (4 x [] = 4 bytes to device 'host#0')
[    0.058858] ( 0) INFO :          - Upload buffer 'filter5x5_1::inRaster|filter5x5_2::outRaster' (4 x [2793, 1920] = 21450240 bytes to device
'host#0')
[    0.070573] ( 0) INFO : <-- advancedload, args <convolution> at src/main-hmpp.c:57
[    0.070656] ( 0) INFO : --> callsite <convolution> at src/main-hmpp.c:62
[    0.070673] ( 0) INFO :          - Bind buffer 'filter5x5_1::heigh|filter5x5_2::heigh' to address 0x7fff69c4afd8 'opt.m_expansion *
BYTES_PER_PIXEL * heigh'
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 55 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

```
[    0.070690] ( 0) INFO :        - Bind buffer 'filter5x5_1::width|filter5x5_2::width' to address 0x7fff69c4afdc 'width'
[    0.070705] ( 0) INFO :        - Bind buffer 'filter5x5_1::filter' to address 0x4040a0 'stencil1'
[    0.070720] ( 0) INFO :        - Bind buffer 'filter5x5_1::inRaster|filter5x5_2::outRaster' to address 0x2af3ac422010 '(Raster2D_C ) raster1'
[    0.070735] ( 0) INFO :        - Bind buffer 'filter5x5_1::outRaster|filter5x5_2::inRaster' to address 0x2af3ad897010 '(Raster2D ) raster2'
[    0.070759] ( 0) INFO :        - Call codelet 'filter5x5_1' (on device 'host#0')
[    0.401762] ( 0) INFO : <-- callsite <convolution> at src/main-hmpp.c:62
[    0.401860] ( 0) INFO : --> callsite <convolution> at src/main-hmpp.c:65
[    0.401878] ( 0) INFO :        - Bind buffer 'filter5x5_1::heigh|filter5x5_2::heigh' to address 0x7fff69c4afd8 'opt.m_expansion *
BYTES_PER_PIXEL * heigh'
[    0.401895] ( 0) INFO :        - Bind buffer 'filter5x5_1::width|filter5x5_2::width' to address 0x7fff69c4afd4 'width'
[    0.401911] ( 0) INFO :        - Bind buffer 'filter5x5_2::filter' to address 0x404120 'stencil2'
[    0.401926] ( 0) INFO :        - Bind buffer 'filter5x5_1::outRaster|filter5x5_2::inRaster' to address 0x2af3ad897010 '(Raster2D_C ) raster2'
[    0.401941] ( 0) INFO :        - Bind buffer 'filter5x5_1::inRaster|filter5x5_2::outRaster' to address 0x2af3ac422010 '(Raster2D ) raster1'
[    0.401961] ( 0) INFO :        - Call codelet 'filter5x5_2' (on device 'host#0')
[    0.730020] ( 0) INFO : <-- callsite <convolution> at src/main-hmpp.c:65
[    0.730122] ( 0) INFO : --> delegatedstore, args <convolution> at src/main-hmpp.c:68
[    0.730146] ( 0) INFO :        - Download buffer 'filter5x5_1::inRaster|filter5x5_2::outRaster' (4 x [2793, 1920] = 21450240 bytes from device
'host#0')
[    0.735362] ( 0) INFO : <-- delegatedstore, args <convolution> at src/main-hmpp.c:68
[    0.839454] ( 0) INFO : --> free, data <convolution> at src/main-hmpp.c:89
[    0.839540] ( 0) INFO :        - Free mirror 0x4040a0 "stencil1" (4 x [25] = 100 bytes on device 'host#0')
[    0.839600] ( 0) INFO :        - Free mirror 0x404120 "stencil2" (4 x [25] = 100 bytes on device 'host#0')
[    0.839626] ( 0) INFO : <-- free, data <convolution> at src/main-hmpp.c:89
[    0.839645] ( 0) INFO : --> release <convolution> at src/main-hmpp.c:90
[    0.839663] ( 0) INFO :        - Free buffer 'filter5x5_1::outRaster|filter5x5_2::inRaster' (4 x [2793, 1920] = 21450240 bytes on device
'host#0')
[    0.841287] ( 0) INFO :        - Free buffer 'filter5x5_1::inRaster|filter5x5_2::outRaster' (4 x [2793, 1920] = 21450240 bytes on device
'host#0')
[    0.842541] ( 0) INFO :        - Free buffer 'filter5x5_1::heigh|filter5x5_2::heigh' (4 x [] = 4 bytes on device 'host#0')
[    0.842586] ( 0) INFO :        - Free buffer 'filter5x5_1::width|filter5x5_2::width' (4 x [] = 4 bytes on device 'host#0')
[    0.842627] ( 0) INFO :          - Release the device 'host#0'
[    0.842655] ( 0) INFO : <-- release <convolution> at src/main-hmpp.c:90
Kernel time: 0.676843
```

The correct result of the COTSon simulation in visual mode is showed in Fig. 25. Please note that the warning message is normal, considering that the platform does not support OpenCL.
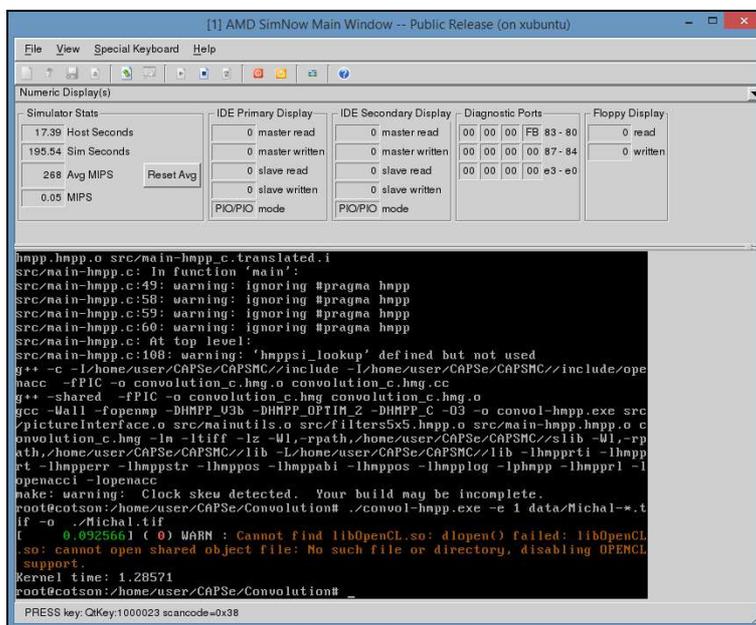
Fig. 25 – Results of a COTSon simulation on the OpenHMPP Convolution example.

8.5 *Further references to more in-depths*

Further details about the CAPS many-core compiler can be found in deliverables D3.5 and D4.7.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 56 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

9 Research Use Case from HP

This section describes the mapping of TERAFLUX applications, compiled to T* ISA, and running on the simulation platform. This work was driven by HP in collaboration with all partners.

9.1 Goal of the experiment or example

By performing simulations and analyzing the results with a full-system simulator, one can gain a thorough understanding of how the proposed architecture behaves, how to improve it, and how to validate the results. The focus of this section is not the precise timing model in simulation, but the capability to simulate interesting benchmarks on thousands of cores, and multiple nodes, through the T* ISA. While the current evaluation does not yet provide precise inter-node timing results, the preliminary evaluation already enables scalability measurements, addressing the dominant performance bottlenecks of the applications.

Another aspect of this section is the mitigation of resource requirements in many-node simulation. Multiple nodes simulation of parallel programs requires more resources than single node simulation. Unless precautions are taken, programs with tremendous parallelism or running on a large number of nodes will saturate memory resources, and even deadlock, on any host machine. In the following the resource requirements in the host and guest machine will be analyzed, and a set of solutions to reduce the memory usage both in host machine and guest machine will be also proposed. The solutions are implemented and integrated in the COTSon simulator.
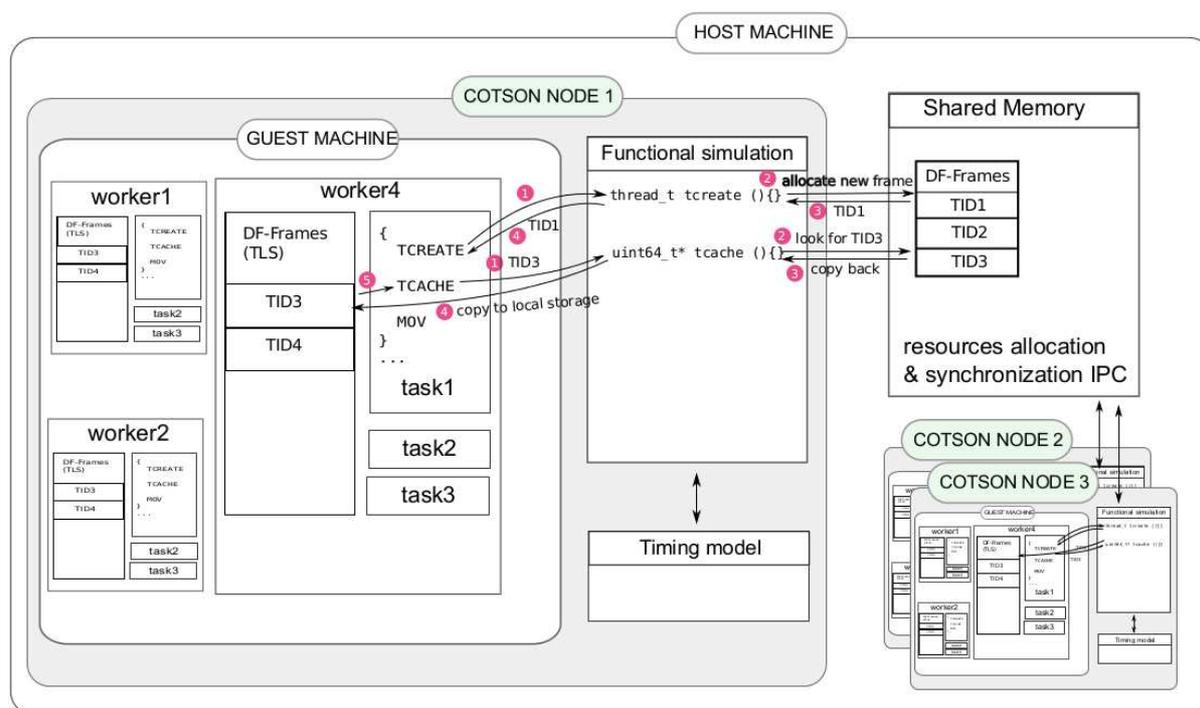
Fig. 26 – Multi-node simulation with COTSon

Fig. 26 shows the multiple-nodes simulation structure on COTSon. The host machine is where the COTSon instances are running on. COTSon supports multiple-nodes simulation by allowing multiple

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 57 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

instances of COTSon, while the communication and synchronization of the instances go through the mediator component. The *guest* machine is the machine (both hardware and operating system level) that is simulated by a COTSon instance. One worker for each CPU within the guest machine is created. Each worker will poll the centralized task queue for ready tasks. At the execution of each task, the T* instructions will be trapped by COTSon for functional simulation. In the figure, task 1 in worker 4 (COTSon node 1), TCREATE (i.e., TSCHEDULE in D6.3, D6.4, D7.4,) and TCACHE will be trapped by COTSon, and call the registered functions *tcreate* and *tload* on the COTSon node where the guest machine is simulated on, respectively. For the purpose of illustrating how dataflow applications are managed within the simulation platform, the T* instructions' implementation in the COTSon simulator (for further information, the reader can refer to deliverables D7.2, D7.4 and deliverables D6.2, and D6.3) is briefly recalled:

- **TCREATE** is trapped by COTSon to the functional simulation, and then the registered function *tcreate* will be called (Fig. 23, step 1 and 2). It will try to allocate a new DF-frame for the new DF-thread in the shared memory. If allocation is successful, the new identifier for the DF-frame (TID1 in this case) will be returned as the result of the execution of the assemble TCREATE. DF-frames in shared memory is shared by all COTSon processes, and protected with locks.

- **TCACHE** is used to cache the remote frame locally. It will be trapped by the functional simulation, and then the registered function *tcache* will be called. The DF-frame id is passed along with TCACHE. In step 2, it will look up for TID3 in the shared DF-Frames, if it is found, the entire DF-frame will be copied from host to guest. More precisely, the DF-frame will be copied from the shared memory to the local heap for this worker thread and the local copy's pointer will be returned to TCACHE finally (step 5). Then in this task, one could directly modify/read this DF-frame. At the time *tdestroy* is called, the modifications will be synchronized and could be seen by other tasks/nodes.

- **TLOAD** is a shortcut for a specialized, current-thread version of TCACHE. It will be trapped by the functional simulation, and then the registered function *tload* will be called. The current thread id is stored within thread local storage and used to get current DF-frame in the shared DF-Frames, if it is found, the DF-Frame will be copied from host to guest, and the local copy's pointer will be returned to TLOAD. Another difference between TLOAD and TCACHE is that the frame loaded by TLOAD is read-only. The data stored in the DF-frame is needed by the computations in the current thread.

- **TDECREASE** makes the target thread designated by thread id to be decremented by *n* either at the time it is called (eager tdecrease) or upon termination of the current thread (lazy tdecrease, at the time TDESTORY is called). It will be trapped by the functional simulation, and the registered function *tdecrease* will be called. In eager tdecrease, the target DF-frame id and is passed along with TDECREASE. It will look up for the target DF-frame, once it is found, it decreases the synchronization Count (SC) by *n*. Then it checks the value SC after decrement, if it reaches to zero, the corresponding thread is moved to the ready queue. In lazy tdecrease, the TDECREASE instruction will be cached.

- **TDESTROY** is trapped to the functional simulation, resulting in a call to the registered function *tdestroy*. This function will terminate the current thread and deallocate its DF-frame in Shared DF-Frames. If running in lazy mode, it will aggregate and execute the cached instructions (e.g. several TDECREASE to the same thread will be aggregated to a single TDECREASE) before deallocation.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 58 of 100

Note that the implementation of the T* ISA extension in the COTSon simulator covers the development of the distributed Thread Scheduling Unit models (TSUF described in this section, and TSU4 described in section 17).

With the aim of investigating the performance of the T* instruction implementation in the COTSon simulator, a set of experiments with multiple-node simulations using 5 benchmarks (Fibonacci, Gauss Seidel, Matrix Multiplication, Sparse LU and Viola Jones - Thales's pedestrian detection) have been conducted. Except for Fibonacci, all benchmarks make use of the Owner Writable Memory (OWM) support. The benchmarks have been implemented in two different flavors. One flavor is to write programs with the low level T* instruction set using C-level "built-in"s (Fibonacci and Matrix Multiplication); the other flavor uses OpenStream and the TERAFLUX compiler support to express dataflow parallelism, and has been used for the more complex benchmarks (Gauss Seidel, Sparse LU and Viola Jones). The multi-node implementation for the latter benchmarks uses the OpenStream extension for OWM. The run-time support library for OpenStream (to match dependences over streams) is integrated into the COTSon run-time.
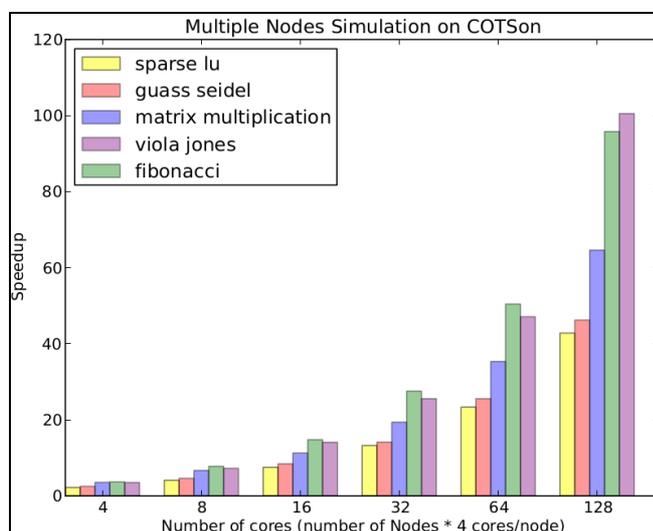
Fig. 27 – Speedup of five different dataflow benchmarks running on different number of cores/nodes.

A few results on 128 cores and 32 nodes are shown in Fig. 27. More details can be found in the WP2 deliverable. With the aim of enabling the reader to run one of these specific benchmarking examples, in the following the single node simulation of Matrix Multiplication benchmark is described in details.

9.2 Location of the involved files

All example files and instructions are provided on the TSUF branch of COTSon (we assume here that the checkout of $COTSON-ROOT involves not only the trunk as in Sect. 1.1, but also the branches):

```
$COTSON-ROOT/branches/tflux-test/tsuf
```

The software stack uses the DF-proxies branch of the OpenStream compiler, where we integrated our T* backend implementation and OWM support (cf. D4.7). The simulated architecture uses SimNow version 4.6.2, and the most recent version of COTSon with support for T* architecture (the TSUF branch).

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

9.3 Detailed instructions to start

The *Matrix Multiply* kernel generates a moderate number of dataflow threads (namely DF-Threads), but stress more the TERAFLUX architecture from the computational viewpoint. In order to run the example, move on the correct folder:

```
$ cd $COTSON-ROOT/branches/tflux-test/tsuf
```

Open the *Makefile* file with a text editor and check that the first line is correctly pointing the main COTSon folder. Then, in the same file set the variable *TESTS* to *matmul*, in order to run the selected benchmark:

```
$ vi Makefile
COTSON_ROOT=$(shell bash -c 'cd ../../../trunk; pwd')
COTSON_SRC=$(COTSON_ROOT)/src
TSUSIM=tflux_tsu.so
TESTS = matmul
...
```

At this point one needs to run the build process for the local folder. This operation is necessary to build the shared object library (*tflux_tsu.so*) that contains the code used to implement the thread scheduling unit:

```
$ make
```

The next step is to enter in the benchmark folder and modify the local *Makefile* file (through a text editor), setting up the proper configuration of the simulated system (i.e., size of the input of the benchmark, number of cores, etc.). In particular, set the variable *COTSON* to point the main simulation folder corresponding to the position *COTSON-ROOT/trunk*. Then, set the size of the benchmark input modifying the value associated to the variable *SZ* (here the value is 35). The number of cores used by the simulated system is expressed by the value of the *NT* variable (in this example we run on a single node with 4 cores).

```
all: $(TESTS)
COTSON=$(shell bash -c 'cd ../../../../trunk; pwd')/bin/cotson
DFDIR= $(shell bash -c 'cd ..; pwd')
DFRT=$(DFDIR)/dflib.o
DFLIBS=-lpthread
TSUSIM=$(DFDIR)/tflux_tsu.so
PWD=$(shell pwd)
RM=rm -rf
TSCRIPT=$(PWD)/tsutest
WSDIR=./libworkstream_df
WSOPTS=-g -O0 -ffast-math -D_GNU_SOURCE -I . -fPIC -Wall -Wextra -lpthread
OWMSZ=32000000
SZ=35
NT=4
TESTS = matmul
HTMTESTS = tmtest_htm
...
```

With the next step the reader has to check the Lua configuration file. Since a single node simulation is running, the reader needs to open the *tsu_single.lua* file with a text editor, and comment the *display* variable so that the whole simulation output will be displayed on the console and copied also on text file. The *use_bsd()* function is set to *4p.bsd* in order to launch a 4-cores system with SimNow. Similarly, the *sampler* object is set to *no_timing*, in order to run a pure functional simulation. To run a timing simulation, the user must change the value of this object to *simple*.

```
runid="tsu"
abaeterno_so=TSUSIM

wd=os.getenv("PWD")
tmpdir=wd
debug=true
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 60 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

```
-- clean_sandbox=false
TSULAT=1

options = {
    --max_nanos='3G',
    exit_trigger='terminate',
    sampler={type="no_timing", quantum="10M"},
    -- sampler={type="interval",functional="20M",warming="100k",simulation="100k"},
    -- sampler={type="simple", quantum="3M"},
    heartbeat={ type="file_last", logfile=runid..".log" },
    -- interleaver_order="round_robin",
    custom_asm=true,
    time_feedback=true,
    tsu_ignore_errors="true",
    -- tsu_speculative_threads=true,
    tsu_statfile="/tmp/xx.dat",
    -- tsu_destroy_polls=true,
    -- tsu_keep_target_frames="false",

    tsu_def_lat=1*TSULAT,
    tsu_rd_lat=20*TSULAT,
    tsu_wr_lat=10*TSULAT,
    tsu_sub_lat=100*TSULAT,
    tsu_sch_lat=1000*TSULAT,
}

one_node_script="run_interactive"
-- display=os.getenv("DISPLAY")
copy_files_prefix=runid.."."
-- clean_sandbox=false

simnow.commands=function()
    -- use_bsd('8p.bsd')
    -- use_bsd('16p.bsd')
    -- use_bsd('32p.bsd')
    use_bsd('4p.bsd')
    use_hdd('karmic64.img')
    set_journal()
    execute(SCRIPT)
end

function build()
    i=0
    while i < disks() do
        disk=get_disk(i)
        disk:timer{ name='disk'..i, type="simple_disk", }
        i=i+1
    end
    i=0
    while i < nics() do
        nic=get_nic(i)
...
```

At this point is possible to launch the simulation as follows:

```
$ make run_single
```

9.4 Expected output

The following files are involved in the output process. The file *node.1.tsu.log* contains the statistics gathered by COTSon during the simulation:

```
Input values:

cpu0.bpred_perfect                          false
cpu0.branch_mispred_penalty                 8
cpu0.commit_cpi                             1.0
cpu0.dcache.fudge                           1.0
cpu0.icache.fudge                           1.0
cpu0.twolev.hlength                         14
cpu0.twolev.l1_size                         1
cpu0.twolev.l2_size                         16kB
cpu0.twolev.use_xor                         1
cpu0.type                                   timer0
cpu1.bpred_perfect                          false
cpu1.branch_mispred_penalty                 8
cpu1.commit_cpi                             1.0
cpu1.dcache.fudge                           1.0
cpu1.icache.fudge                           1.0
cpu1.twolev.hlength                         14
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 61 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

```
cpu1.twolev.l1_size                             1
cpu1.twolev.l2_size                             16kB
...
...

Output values:

cpu0.cycles                                     1309999869
cpu0.haltcount                                  819583852
cpu0.hb_ATC_flush                               0
cpu0.hb_CR3_different                           0
cpu0.hb_CR3_equal                               0
cpu0.hb_ev_Exception                            0
cpu0.hb_ev_HW_interrupt                         0
cpu0.hb_ev_SW_interrupt                         0
cpu0.idlecount                                  823247239
cpu0.instcount                                  486752630
cpu0.invalid_translation_bytes                  318860
cpu0.iocount                                    1946489
cpu0.metadata_bytes                             23073536
cpu0.other_exceptions                           896760
cpu0.plain_invalidations                        1297
cpu0.range_invalidations                        77
cpu0.read_mmios                                 650
cpu0.read_pios                                  603
cpu0.segv_exceptions                            62303
cpu0.timer.cycles                               0
cpu0.timer.instructions                         0
cpu0.timer.twolev.lookup                        0
cpu0.timer.twolev.misses                        0
cpu0.timer.twolev.reset                         0
cpu0.timer.twolev.update                        0
cpu0.trace_cache_size                           0
cpu0.valid_translation_bytes                    36613967
cpu0.write_mmios                                886
cpu0.write_pios                                 2063
cpu1.cycles                                      1309999869
cpu1.haltcount                                  859197061
cpu1.hb_ATC_flush                               0
cpu1.hb_CR3_different                           0
cpu1.hb_CR3_equal                               0
cpu1.hb_ev_Exception                            0
cpu1.hb_ev_HW_interrupt                         0
cpu1.hb_ev_SW_interrupt                         0
cpu1.idlecount                                  860191233
...
...
```

The file *node.1.stdout.log* contains the output generated by the benchmark and the simulator during the simulation:

```
kernel.randomize_va_space = 0
+ /etc/init.d/ssh stop
 * Stopping OpenBSD Secure Shell server sshd
   ...done.
+ pkill -9 dhclient3
+ ifconfig eth0 down
+ echo performance
+ cat /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_max_freq
+ cat /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_max_freq
+ echo performance
+ cat /sys/devices/system/cpu/cpu1/cpufreq/cpuinfo_max_freq
+ cat /sys/devices/system/cpu/cpu1/cpufreq/cpuinfo_max_freq
+ echo performance
+ cat /sys/devices/system/cpu/cpu2/cpufreq/cpuinfo_max_freq
+ cat /sys/devices/system/cpu/cpu2/cpufreq/cpuinfo_max_freq
+ echo performance
+ cat /sys/devices/system/cpu/cpu3/cpufreq/cpuinfo_max_freq
+ cat /sys/devices/system/cpu/cpu3/cpufreq/cpuinfo_max_freq
+ echo Local config done
Local config done

RUNNING matmul
DF owm 0x7ffff6179000 32000000
Creating 4 workers for 4 cores
Starting workers
Starting master node 1 nodes 1 workers 4
Deallocate OWM at 0x7ffff6179000
All workers done, goodbye
============================================
block 2 sum = 6183107
block 0 sum = 6279596
block 1 sum = 6434683
block 7 sum = 6514228
block 4 sum = 6256864
block 5 sum = 6292689
block 9 sum = 6359774
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD
and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 62 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

```
block 8 sum = 6118062
block 11 sum = 6462022
block 6 sum = 6273600
block 3 sum = 6374453
block 13 sum = 6488416
block 10 sum = 6295426
block 12 sum = 6443866
block 14 sum = 6361545
block 15 sum = 6359904
block 17 sum = 6445377
block 19 sum = 6307741
block 20 sum = 6313001
block 16 sum = 6475514
block 23 sum = 6785729
block 18 sum = 6426926
block 25 sum = 6543575
block 21 sum = 6345925
block 26 sum = 6163990
block 29 sum = 6219195
block 22 sum = 6139551
block 31 sum = 6299559
block 30 sum = 6272789
block 24 sum = 6353918
block 33 sum = 6275531
block 34 sum = 6361807
block 27 sum = 6375751
block 35 sum = 6657941
block 36 sum = 6500855
block 37 sum = 6081004
block 32 sum = 6534934
block 39 sum = 6283410
block 38 sum = 6244325
block 28 sum = 6293559
*** SUCCESS ***
==================   DF STATS   =====================
df time:  145072751 ns (145.073 ms)
  core  0: 435126644 insts 435126651 xc     91602 ic, 435218253 cycles
  core  1: 435144577 insts 435144856 xc     73397 ic, 435218253 cycles
  core  2: 435166946 insts 435167225 xc     51028 ic, 435218253 cycles
  core  3: 435079704 insts 435050187 xc    168066 ic, 435218253 cycles
```

On the screen of the console, the user should observe the following output:

```
...
$1 exec> keyboard.key 23 A3
$
$1 exec> keyboard.key 39 B9
$
$1 exec> keyboard.key 34 B4
$
$1 exec> keyboard.key 35 B5
$
$1 exec> keyboard.key 30 B0
$
$1 exec> keyboard.key 1C 9C
$
$1 exec> go
$+++ TRESET(START) nanos 179838554
$+++ TSchedule 83 TDestroy 82 TCache 1478582 TLoad 162 Polls 82 TDecrease 80
$+++ TFINISH nanos 328405990 (diff 148567436 ns, 148.567 ms)
$EXIT TRIGGER: terminate
$copying node 1 output to /home/scionti/Tools/cotson-release/branches/tflux-$test/tsuf/test
$cleaning sandboxes
```

9.5 Further references to more in-depths

Resource usage optimization involves a careful memory management technique, and a heuristic for task creation throttling. These are described in Chapter 7 of Feng Li's thesis (INRIA) – an extract of which is presented in the next Section 10.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 63 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

10 Research Use Case from INRIA

One general criticism targeting dataflow computing is the cumbersome/inefficient management of complex data structures. The functional nature of pure dataflow programs implies that all operations are side-effect free. The absence of side effect means that if tokens are allowed to carry vectors, arrays, or other complex data structures, an operation on a data structure results in a new data structure. Which will greatly increase the communication overhead in practice. The problem of efficiently representing and manipulating complex data structures in a dataflow execution model has remained a fundamental and practical challenge. The vertically integrated design and flow of TERAFLUX addresses this challenge. In the following, the design and usage scenarios of Owner Writable Memory (OWM, designed in WP3 and WP6) is described. The OWM memory model is loosely coupled. Compared to word-based cache coherence, the protocol is largely simplified with the assumption that users have to synchronize all the tasks that access to the same OWM subregion to preserve the ownership atomicity. There is usually a trade-off between programmability and flexibility, in TERAFLUX some of the complexity of the hardware design is shifted to the user, but at the same time, it provides a compilation tool chain to simplify this procedure. The OWM extension to OpenStream provides an easy to use compilation support. Complementary support for complex data structures also involve *Transactional Memory*, see the D3.5 and D6.4 deliverables for details.

10.1 Goal of the experiment or example

The Owner Writable Memory model (OWM) has been proposed in TERAFLUX to reduce the communication overheads when complex data structures are passed over threads. The name and idea originates from Prof. Ian Watson from the Unversity of Manchester. The design and semantics of language support for OWM is presented in the D3.5 deliverable. This section mainly covers the execution model for OWM and its application to concrete use cases.

The OWM protocol was first formalized by François Gindraud during his Master's thesis. A short overview is provided in this deliverable. The OWM protocol is inspired from a distributed, directory-based MSI cache coherence protocol. The global OWM memory address is mapped locally to each node on the NoC. Before a task can access to an OWM subregion, it has to claim ownership beforehand through a TSUBSCRIBE. The owner will always keep track of the nodes that hold a valid copy of the subregion. One important property of resolving the ownership of an OWM subregion is handled as follows:

- The globally addressable OWM is distributed over the platform's nodes. For a given OWM region, one may tell the node it is originates from (i.e., its allocation) by the address. This node is the region's first owner.

- When ownership changes, the first owner always keeps the information of the current owner. When claim ownership or data requests have been received, it forwards the requests to its owner and renew the ownership information. One problem with the MSI is the atomicity of bus events. On the NoC, one can assume that all the messages will eventually arrive without packet loss or duplication, in any order. So it must be ensured that a task accesses a region in W mode will invalidate all the copies of that region on other nodes before the tasks depends on being executed. Adding a memory semantic TPUBLISH can enforce this property. When all the modifications are done within the OWM subregion, the owner task has to execute TPUBLISH on the region explicitly to ensure all the other nodes depend on the new data will be invalidated.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 64 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Each node on the NoC operates on two message queues, a send queue and a receive queue. Nodes communicate via messages. The message sending is abstracted as removing one message from the send queue of the source node, and add it atomically to the receive queue of the destination node. The protocol could be divided into three message types:

- **DataRequest** and **DataAnswer** messages are equivalent to a BusRd event in the MSI coherence protocol for directory caches. The request will be sent to the first owner of this region, and forwarded to the current owner. When the owner node receives this request, it replies with a DataAnswer message containing the fresh data, and add the request node to the list of valid nodes. When the request node receives the DataAnswer, it updates the local copy of the OWM region, sets the valid flag as true, and resets the requested flag.

- **OwnerRequest** and **OwnerAnswer** are similar to the BusM event in MSI. In snooping MSI the bus is guaranteed that only one busM event could occur. In OWM memory model, the enforced dependences are added between tasks so that no ownership change could occur if there is another node claimed the ownership and did not publish the data yet. The request message will be sent to the first owner of this region, and will be forwarded to the current owner. The first owner will update the ownership information by checking the OwnerRequest message. When the destination node receives this message, it sets the valid flag to be true, and send OwnerAnswer which packs the data and ownership response metadata information to the new owner. When the request node receives this message, it will update the region it requests by the data received. The valid set information is also sent in the metadata by the previous owner, the request node will update this information, and add the previous owner to this set.

- Invalidation complements the ownership transfer process. In this case an explicitly invalidation request is sent to other nodes that have a local copy upon modification. The **InvalidateRequest** is sent to all the nodes in the valid set. The valid set will be copied to Waiting Invalidation Acknowledge Set (WIAS) before it is reset. When the node receives an InvalidateRequest, it sets the valid flag to false, and send back the **InvalidateAck** message to acknowledge the sender. When the sender receives InvalidateAck, it removes the source node from WIAS.

OWM is one single memory region, but it could be further divided into smaller subregions for finer granularity. The **owm_tsubscribe** and **owm_tpublish** are introduced as an extension to the T* ISA extension for supporting OWM. One could subscribe (by calling *owm_tsubscribe*) part of OWM region to a thread, which means, before this thread is executed, the ownership of the subregion should be acquired, and ready for access. One thread could publish the modifications to the OWM region it acquired by calling *owm_tpublish*. Before the modifications are published, any read from another thread is not guaranteed to see consistent data. OWM is a weak memory model; it is the programmer's responsibility to take care of data consistency and dependences.

Here is the detailed description for the OWM instructions extending the T* ISA:

- *void owm_tsubscribe(void *tid, int off, int offowm, int size, int mode)*

  Subscribes the OWM subregion described by *offowm, size, mode* to be cached before executing dataflow thread with thread id (*tid*): offowm is the initial offset to the global OWM

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 65 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

region, size is the size of the OWM subregion to be subscribed, mode describes the access mode to the region, it could be read-only, write-only or read-write. The pointer to the local cached OWM region is stored in DF-frame described by (tid, off), where tid is the thread id, and off is the offset in the thread's DF-frame.

- *void owm_tpublish(void *regptr, int size)*

  Publishes the modification to the OWM region described by *regptr, size*. If size is 0, it writes the region starting at regptr using the size that was registered during the owm subscribe operation. This way, different threads can be subscribed to different segments of the same region using different sizes.

OWM is integrated into the OpenStream compiler as a language extension. One could use OpenStream to decompose programs into tasks and to explicit the flow of data among them, thus exposing data, task, and pipeline parallelism. The OWM extension of OpenStream takes the form of a simple cache clause in the task pragma:

```
#pragma omp task cache (ACCESS_MODE:MEM[OFF:SIZE])
```

The cache clause subscribes the task with the OWM subregion described by *MEM[off:size]* with read (R), write (W) or read-write (RW) access mode (*ACCESS_MODE*). The current syntax supports only one dimensional arrays, but it could be easily extended to multiple dimension arrays. A simple example is presented in the D3.5 deliverable.

As illustrated below with matrix multiplication, the OWM extension can be easily integrated into dataflow programs. The user may use OpenStream constructs to synchronize between tasks. Feng Li's PhD thesis presents other use cases. OWM support is implemented in the OpenStream compiler. The lowered built-in functions are translated directly to the T* ISA, linked with part of the OpenStream library (run-time related with streaming operations), and part of the run-time support in the COTSon simulator. In the implementation of benchmarks where two-dimensional arrays are used, one usually has to remap the memory regions as a single dimension array, which might have extra cost. An abstract polyhedral representation could be used in this case to represent an OWM region in multiple dimension arrays situation.

10.2 Location of the involved files

All example files and instructions are provided on the TSUF branch of COTSon.

```
http://sourceforge.net/p/cotson/code/HEAD/tree/branches/tflux-test/tsuf/README
```

The software stack uses the DF-proxies branch of the OpenStream compiler, where the T* back-end implementation and OWM support are integrated. Information regarding the OpenStream compiler can be found at:

```
http://openstream.info/download
```

And for the GIT repository itself:

```
git clone http://git.code.sf.net/p/open-stream/code
```

The simulated architecture uses SimNow version 4.6.2, and the most recent version of COTSon with support for T* architecture (the TSUF branch).

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 66 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

10.3 Detailed instructions to start

The sources for the compiler can be downloaded directly from the official repository (see previous section), using the following command:

```
$ git clone git://git.code.df.net/p/open-stream/code $COTSON-HOME/open-stream
```

After having downloaded the sources from the official repository the following actions should be done for installing the compiler:

```
$ cd $COTSON-HOME/open-stream/
$ make
```

This automatically performs the following actions:

- Download the sources of any missing libraries needed by OpenStream;
- Build and locally install these dependences;
- Build and locally install the compiler and runtime libraries in **open-stream/install/** folder;
- Build the OpenStream codes in the **open-stream/examples/** folder;

After the compilation process has finished it is possible to move on the example directory and launch one of the available examples. For the purposes of this document the Matrix Multiplication example is illustrated. Matrix Multiplication is a good example to show the expressiveness of OWM in concrete use cases. This characteristic will be illustrated in this example in three phases: in the first phase, one task allocates and initializes all the matrices in the OWM memory; in the second phase, the matrix is partitioned to several blocks, each task will cache the OWM subregion it needs and compute the results, then store the results to the output matrix; and a final task will wait till the end of all the previously created tasks, print and verify the results. A detailed description is provided following the path of the three phases.

10.4 Expected output

The code fragment in Fig. 28 shows the code for matrix allocation and initialization. The input matrices A, B and output matrix C are allocated by calling *tstar_owm_allocate*, while *fill_matrix* initializes all the matrices. The cache pragma subscribes matrices A, B, C in write mode. At the time fill_matrix is executed, all the OWM subregion it subscribes will be ready for writing. The modification will be published at the end of the task. Stream *init* is used to synchronize between phase one and phase two, so that the computation could only be started when the initialization finishes.

```
int init __attribute__((stream));

DATA *A = tstar_owm_alloc (N * N * sizeof (DATA));
DATA *B = tstar_owm_alloc (N * N * sizeof (DATA));
DATA *C = tstar_owm_alloc (N * N * sizeof (DATA));

#pragma omp task cache (W: A[:N*N], B[:N*N], C[:N*N]) output (init)
fill_matrix (A, B, C, N);
```

Fig. 28 – Matrix product – input.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 67 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

```
for (j = nb = 0; j < N; j += BLOCKSZ, ++nb) {
    int aoff = N * j; int boff = 0; int coff = N * j;

    #pragma omp task cache (R: A[aoff:N*BLOCKSZ], B[boff:N*N]) \
    cache (W: C[coff:N*BLOCKSZ]) output (finish)
    {
      for (int jj = j; jj < j + BLOCKSZ; ++jj) {
        for (int i = 0; i < N; i++) {
          DATA t = 0;
          for (int k = 0; k < N; k++) {
            t += A[jj * N + k] * B[k * N + i];
          }
          C[i + jj * N] = t;
        }
      }
    }
  }
```

Fig. 29 – Matrix product – input.

The main computations are done in the following phase. Fig. 29 shows the code for matrix multiplication. The matrix is divided into blocks, each thread caches BLOCKSZ rows of matrix A, and the entire matrix B in read mode, and BLOCKSZ rows of matrix C in write mode. Once the thread is executed, it computes ABLOCKSZ•N•BN•N = CBLOCKSZ•N. At the end of each thread, the modification to matrix C is published and thus available for reading by other threads. Each task created in this phase writes a single value to stream *finish*. Stream finish acts as a waiting barrier in the last task, which will wait for the termination of all threads created in this phase.

```
#pragma omp task cache (R: A[:N*N], B[:N*N], C[:N*N]) \
input (finish >> final_view[N/BLOCKSZ])
{
  dump_result_and_verify (A, B, C, N);
}
```

Fig. 30 – Matrix product – input.

Fig. 30 shows the final thread, which waits for the termination of all the threads created in phase two. Once all the computations are done, it will output the results and do the verification if necessary. Stream finish acts as a barrier, waits for N/BLOCKSZ inputs from stream finish. Each thread created in phase two writes to stream finish once finished.

10.5 Further references to more in-depths

The semantics, dedicated memory model and coherence protocol for OWM will be the subject of a joint publication of the project partners. The Master thesis of François Gindraud is currently the most accurate information and is available on request. Further experiments are reported in the D2.4 deliverable and in Feng Li's PhD thesis. The experimental validation of OWM memory model is presented in Chapter 7 of Feng Li's PhD thesis. We have studied four benchmarks with OWM support: matrix multiplication, sparse LU, Gauss Seidel and Viola & Jones (pedestrian detection); those benchmarks are validated with COTSon's TSUF branch.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 68 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

11 Research Use Case from MSFT

This section demonstrates how to run the TERAFLUX operating system prototype that was developed to support research and experimentation with the various parallel, distributed and reliable execution algorithms that are suggested in TERAFLUX. Specifically, the operating system supports execution of a distributed application over the many-core device using dataflow threads, it was designed to handle core soft errors with Double Execution mechanism and can handle node hard-failures such that the application can transparently continue execution as the work that was pending on the failed node is recovered and executed by the remaining nodes.

The system is simulated over COTSon (running a SimNow instance for each of the nodes) with a slightly modified version of TSUF, which implements a shared memory mechanism with a weak consistency model similar to acquire/release. This shared memory is the only mechanism utilized by the operating system for inter-node communications and shared data.

11.1 Goal of the experiment or example

This experiment launches a distributed Fibonacci sequence computation over the TERAFLUX operating system. Its goal is to demonstrate how the operating system executes a massively parallel application made of dataflow threads over all of the cores in the system.

During execution, the simulation displays the operations performed by the run-time and the user code in the virtual monitor of each SimNow instance, additionally, the output is logged and can be examined after execution. Soft-errors can be injected randomly to the results to demonstrate the Double Execution in action, and complete node failure can be triggered by the user to watch the recovery mechanism.

Various compile flags control some of the run-time mechanisms (e.g., scheduling algorithm, Double Execution, etc.), and what type of log messages are seen.

11.2 Location of the involved files

The runtime files and sample application are contained in the following folder:

```
$COTSONHOME/branches/tflux-test/tfos/
```

Where **COTSONHOME** is an environmental variable identifying the path where the COTSon simulator was checked out with:

```
$ svn co https://svn.code.sf.net/p/cotson/code/ $COTSONHOME
```

11.3 Detailed instructions to start

To run this example first checkout and build COTSon, then go to the *tfos-tsuf* folder mentioned above:

```
$ cd $COTSONHOME/branches/tflux-test/tfos/
```

Now start the simulation by executing:

```
$ make run_multi
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD
and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 69 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

After startup, the default simulation view will display general information about the node and list several commands (e.g., show logs, test node failure, etc.) that can be interactively triggered by the user with keyboard command on the SimNow window.

Some parameters can be configured similarly to those in TSUF, for example the number of nodes in the system is specified in *os-tests/tsu_multi.lua*:

```
cluster_nodes=4
```

The number of cores in each node is specified by the bsd file used:

```
--use_bsd('4p.bsd')
use_bsd('16p.bsd')
--use_bsd('32p.bsd')
```

To test node crashes it is recommended to have more than 4 cores in each node. Notice that the bsd's with large number of cores are not created using the default build configuration, they can be downloaded from:

```
https://upload.teraflux.eu/uploads/BSDS/bsds_images_initialized_for_karmc64_1Ghz.tar.gz
```

Some other parameters are specified in *os-tests/Makefile*:

```
OWMSZ=67108864 # Size of the shared region.
SZ=44 # Parameter for the application (e.g. Fibonacci number).
#NT=32 # Number of TSUF workers. Leave undefined to use the number of cores.
```

Several parameters are specified as compile time flags. Some flags control the nature of the dataflow jobs. For example:

```
#define DOUBLE_EXECUTION
//#define INJECT_CORRUPTIONS
```

The above macro is used to determine whether to globally enable Double Execution, and whether to randomly corrupt some of the threads results to see the mechanism in action.

The following macro defines whether to include the actual job binary in the control message or only its name:

```
#define SEND_JOB_NAMES
```

When it's disabled, each job message is self-contained and allows immediate execution on any node without access to shared storage (of the precompiled jobs), at the cost of possibly sending the same binary code many times. Although jobs are usually small (100-200 bytes for Fibonacci) this can be avoided by sending a small job identifier instead of the binary code, later used to load the job from the common file system (subsequent requests are loaded from cache).

Simple scheduling algorithms can be chosen with the macros:

```
// Prefer to schedule on the local node until memory usage is high, then
// a secondary method is used. If this is not defined, the method chosen below is //
immediately used.
#define PREFER_LOCAL
// Define only one of the following:
//#define RANDOM_SCHED_POLICY
#define UNIFORM_DISTRIBUTION_POLICY
```

Those are very simple but demonstrate how the information gathered from heartbeats can be used to help load balancing among nodes.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 70 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

11.4 Expected output

When launched, node instances will open in SimNow windows and display the simulation progress:
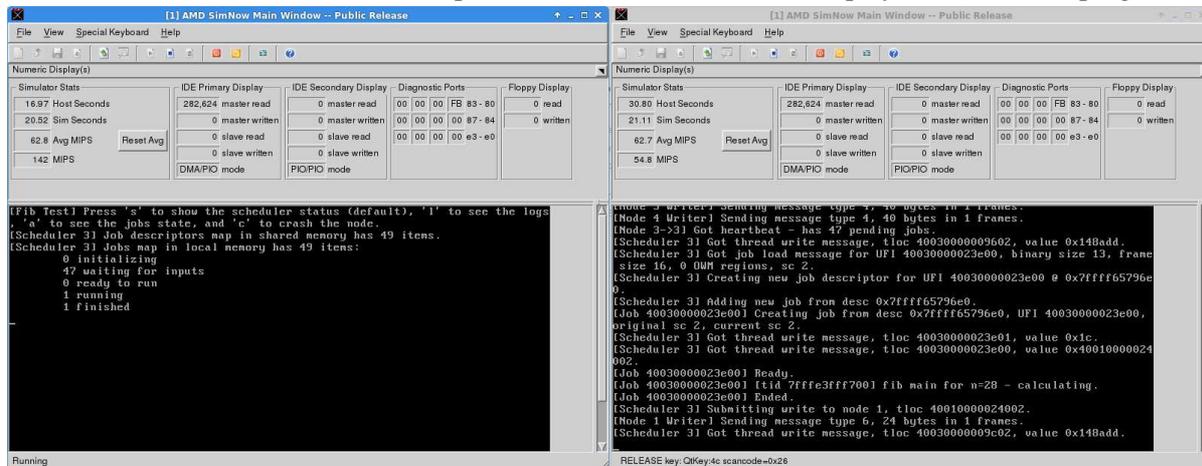
Fig. 31 – Two nodes (two SimNow instances) running on the COTSon simulator.

When the simulation completes, the output of each node can be examined in the *stdout* log files, the output of node 1 could be for example:

```
[[Manager 1] Simulation parameters:
[Manager 1]       16 cores in 4 nodes with 4 cores each.
[Manager 1]       64MB public shared memory, 16MB per node.
[Manager 1]       4*1MB message queues, leaves 12MB for dynamic allocation.
[Manager 1] Starting service thread, ip 0x4202e0.
[Scheduler 1] Dynamic allocation area rounded from 0x7ffff46f4140 to 0x7ffff46f5000, size 12MB.
[Manager 1] Starting service thread, ip 0x40c360.
[Test] Computing fibonacci(41).
[Scheduler 1] Starting message pump.
[Scheduler 1] Submitting job fib_reporter_job with UFI 10010000000200.
[Node 1 Writer] Sending message type 1, 73 bytes in 2 frames.
[Scheduler 1] Submitting job fib_main_job with UFI 10010000000400.
[Node 1 Writer] Sending message type 1, 77 bytes in 2 frames.
[Scheduler 1] Finalizing 0: Write destination updated from VFP 200 to UFI 10010000000400.
[Scheduler 1] Submitting write to node 1, tloc 10010000000400.
[Node 1 Writer] Sending message type 6, 24 bytes in 1 frames.
[Scheduler 1] Finalizing 0: Write destination updated from VFP 200 to UFI 10010000000400.
[Scheduler 1] Submitting write to node 1, tloc 10010000000401.
[Node 1 Writer] Sending message type 6, 24 bytes in 1 frames.
[Scheduler 1] Got job load message for UFI 10010000000200, binary size 17, frame size 8, sc 1.
[Scheduler 1] Creating new job descriptor for UFI 10010000000200 @ 0x7ffff46f5140.
[Job 10010000000200] Creating job from desc 0x7ffff46f5140, UFI 10010000000200, original sc 1, current sc 1.
[BinariesStore] Adding job binary: fib_reporter_job, 142 bytes.
[Scheduler 1] Got job load message for UFI 10010000000400, binary size 13, frame size 16 , sc 2.
[Scheduler 1] Creating new job descriptor for UFI 10010000000400 @ 0x7ffff46f51e0.
[Job 10010000000400] Creating job from desc 0x7ffff46f51e0, UFI 10010000000400, original sc 2, current sc 2.
[BinariesStore] Adding job binary: fib_main_job, 618 bytes.
[Scheduler 1] Got thread write message, tloc 10010000000400, value 0x10010000000200.
[Scheduler 1] Got thread write message, tloc 10010000000401, value 0x29.
[Job 10010000000400] Ready.
[Job 10010000000400] [tid 7fffe3fff700] fib main for n=41 - spawning.
[Job 10010000000400] Ended.
...
[Scheduler 1] Got thread write message, tloc 10010000000200, value 0x9de8d6d.
[Job 10010000000200] Ready.
[Job 10010000000200] [tid 7fffe3fff700] report: fib result = 165580141
[Job 10010000000200] [tid 7fffe3fff700] Exit requested.
[Job 10010000000200] Ended.
[Scheduler 1] Sending termination requests...
[Node 1 Writer] Sending message type 7, 8 bytes in 1 frames.
[Node 2 Writer] Sending message type 7, 8 bytes in 1 frames.
[Node 3 Writer] Sending message type 7, 8 bytes in 1 frames.
[Node 4 Writer] Sending message type 7, 8 bytes in 1 frames.
[Node 1->1] Got terminate message.
[Scheduler 1] Exiting.
```

If a node (node 3 in the example) was killed by user input, the recovery node (node 1 was chosen in the example) will begin to take over and process the work of the failed node and display:

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 71 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Fig. 32 – Output of the simulation when a node in the system fails.

The log should show:

```
...
[Watchdog] Node 3 probably died, no heart beat received in the last 189 milliseconds.
[Manager 1] Starting recovery procedure for node 3.
[Manager 1] Starting service thread, ip 0x406d60.
[Recovery Scheduler for 3] Checking shared segment sanity...
[Recovery Scheduler for 3] Job descriptors map in shared memory has 37 items.
[Recovery Scheduler for 3] Adding new job from desc 0x7ffff65793c0.
[Job 1003000000bc00] Creating job from desc 0x7ffff65793c0, UFI 1003000000bc00, original sc 2, current sc 0.
[Job 1003000000bc00] Ready.
[Job 1003000000bc00] [tid 7fffe3fff700] fib main for n=29 - calculating.
[Recovery Scheduler for 3] Adding new job from desc 0x7ffff65791e0.
[Job 2003000000e600] Creating job from desc 0x7ffff65791e0, UFI 2003000000e600, original sc 2, current sc 2.
[Recovery Scheduler for 3] Adding new job from desc 0x7ffff6579140.
[Job 30030000000c00] Creating job from desc 0x7ffff6579140, UFI 30030000000c00, original sc 3, current sc 2.
... <More recovered jobs information> ...
[Recovery Scheduler for 3] Has 46 jobs in local memory:
        0 initializing
        35 waiting for inputs
        8 ready
        3 running
        0 finished
        0 total completed
[Recovery Scheduler for 3] Starting message pump.
[Recovery Scheduler for 3] Got job load message for UFI 1003000000c200, binary size 13, frame size 16, sc 2.
[Recovery Scheduler for 3] Creating new job descriptor for UFI 1003000000c200 @ 0x7ffff657a860.
[Job 1003000000c200] Creating job from desc 0x7ffff657a860, UFI 1003000000c200, original sc 2, current sc 2.
[Recovery Scheduler for 3] Got thread write message, tloc 2003000000e601, value 0x1e.
... <More recovered messages processing> ...
```

If Double Execution and random error injections are enabled, an injected soft-error will produce output similar to the following:

Fig. 33 – Double Execution of dataflow threads, and the corresponding verification output.

This is a simple implementation of Double Execution; each job is executed twice (notice the different *tid* on each execution), and the results are not committed to the shared memory until the results of both threads are ready and compared equal. When an error is injected, the mechanism detects it and launches the job again on two threads.

11.5 Further references to more in-depths

For more details on the operating system structure and its mechanisms that support the reliable execution of Data-Flow threads while assuming incoherent shared memory and possibility of node hard-failures, please refer to deliverable D5.4 Section 4. This information is also contained in the *TFOS.pdf* document in the source folder.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 72 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

12 Research Use Case from THALES

This section shows a subset of the experiments performed on the applications provided by Thales, to evaluate the TERAFLUX architecture and associated tools in an industrial context. More details on these analyses can be found in deliverable D2.4.

THALES provided the following two use-cases: the Radar application and the Pedestrian Detection application. This document focuses on the later one, the Radar application, providing some easy instructions for its installation and test.

The Radar application is an airborne radar application embedded in planes to detect the position and radial speed of another flying target despite the presence of jamming devices. It is based on the Space-Time Adaptive Processing (STAP) algorithm. This application detailed in D2.1 and D2.3 is characterized by:

- Real-time constraints expressed in the form of throughput requirements;
- The pure dataflow behavior of a signal processing application;
- But very large data (5th dimensional data) being transferred between each task/filter;
- T necessity to manipulate this data (e.g., rotate, transpose, etc.) for each filter to benefits from cache locality;

12.1 Goal of the experiment or example

The goals of the experiments are: first, to evaluate the scalability of the proposed architecture and associated dataflow execution models in the context of real-time applications, selecting one application that is very dataflow friendly (radar).

Second, to evaluate the ergonomics of the tools and associated dataflow languages, and to evaluate the cost of porting legacy single-core applications to the TERAFLUX platform, including the parallelization costs versus the obtained speedups, using the available execution models.

Third, to estimate what are the best parallelization options for porting classification algorithms and signal-processing algorithms to teradevices. In the case of the Radar application its parallelization is quite straightforward alongside the dataflow pipeline (more details can be found on D2.4).

12.2 Location of the involved files

To start, the *tsuf* version of TSU must be checked out with:

```
$ svn co https://svn.code.sf.net/p/cotson/code/branches/tflux-test/tsuf/ $TSUF_HOME
```

The Radar benchmark (STAP) can be checked out with:

```
$ svn co https://svn.code.sf.net/p/teraflux-stap/code $STAP_HOME
```

12.3 Detailed instructions to start

Before using the Radar application the following steps must be followed:

1. Checkout, build and install COTSON;
2. Checkout, build and install the TSUF version of the distributed Thread Scheduling Unit (TSU);
3. Checkout, build and install the SimNow simulator;
4. Checkout, build and install the TERAFLUX-version of the OpenStream compiler;
5. (Optional) Checkout, build and install the OmpSs compiler (not compatible with the Thread Scheduling Unit models);

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 73 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

A `Makefile` is included with the application. Simply type `make` to see all the available options. The `makefile` should be updated with the paths of the previously installed software (i.e., COTSon, SimNow, OpenStream and optionally OmpSs). Below the options that concern the OpenStream with TSU support version of the Radar application:

```
$ make <stap-os-cotson|run-os-cotson-small|run-os-cotson-large|run-os-cotson-
huge|run-os-cotson-multi-small|run-os-cotson-multi-large|run-os-cotson-multi-
huge|clean-os-cotson>
$    Build OpenStream version of the application.
$    Run COTSON OpenStream version on small input.
$    Run COTSON OpenStream version on large input.
$    Run COTSON OpenStream version on huge input.
$    Run multi COTSON OpenStream version on small input.
$    Run multi COTSON OpenStream version on large input.
$    Run multi COTSON OpenStream version on huge input.
$    Clean files created by the OpenStream application.
```

To launch a single node TSU execution with the small dataset just launch *make run-os-cotson-small*. The *-cotson-multi-* variations will execute a multiple node TSU simulation. Three different input sets are provided for evaluation.

The sources provide a *$STAP_HOME/resources* folder with the TSU configuration files, the default use machine configurations provided by COTSon, modify them to use larger/smaller configurations.

12.4 Expected output

The Radar application doesn't provide any visual output. It takes a radar signal and detects moving objects. When running the TERAFLUX version of the application with the `make run-os-cotson-<small|large|huge>` command it generates as output the detected objects in a text file with the name of the selected input set: `<small|large|huge>.txt`. The `Makefile` command `run-os-cotson-<small|large|huge>` places the output file in `run/<os-cotson>`. The user can check that the result is correct by comparing this output against the output of the sequential single core x86 version that can be run with the `make run-seq-<small|large|huge>` command that generates its output file in `run/seq` folder.

Some speedup results for the Radar application observed with different configurations (4 cores per node) of the TERAFLUX machine compared to the sequential version are reported in table 2.

Table 2 – Radar application speedup against sequential execution

Dataset			
Cores	**Small**	**Large**	**Huge**
4	3.48	3.48	3.48
8	6.22	6.24	6.26
16	10.28	10.41	10.44
32	14.33	14.59	14.63
64	16.96	18.08	17.92

12.5 Further references to more in-depths

More details on the Radar and the Pedestrian Detection applications use-cases can be found in deliverables D2.1 and D2.2. Some implementation details are provided in deliverable D2.3, whereas the final evaluation is part of deliverable D2.4.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 74 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

13 Research Use Case from UAU

This section shows a simplified experiment to investigate the performance overhead induced by the fault detection mechanisms developed in TERAFLUX. A more detailed analysis can be found in Deliverable D5.4.

13.1 Goal of the experiment

The goal of this experiment is to show the performance overhead of pessimistic and optimistic Double Execution of *Fibonacci(31)* for one TERAFLUX node with 4 cores. The configuration of the simulator is similar to the one described in Deliverable D5.4.

13.2 Location of the involved files

To start, the fault-tolerant version of the Thread Scheduling Unit (TSU) must be checked out with:

```
$ svn co https://svn.code.sf.net/p/cotson/code/branches/tflux-test/ft-tsu/ $FT_TSU_HOME
```

The fault-tolerant version of the TSU (*tflux_tsu.cpp*), the used cpu timer (*timer_uau.cpp*), and the COTSon configuration skeleton (*tsu_bench.lua*) used for the experiment can all be found in *$FT_TSU_HOME*.

The benchmarks are stored in:

```
$ FT_TSU_HOME/examples
```

13.3 Detailed instructions to start

Before the experiment can be started, the required dependencies must be installed by:

```
$ FT_TSU_HOME/configure --simnow_dir /path/to/simnow
```

The *configure* script will perform the following tasks:

1. Checkout and build the COTSon simulator;

2. Build and link all required files in *$FT_TSU_HOME*;

Afterwards the experiment can be started with:

```
$ FT_TSU_HOME/run_example --res_folder /path/to/results_folder
```

Where the *res_folder* option describes the folder where the results of the experiments will be stored.

13.4 Expected output

After the experiment has finished the execution, the raw output files of the simulator runs can be found in the *res_folder*.

Finally, the simulator outputs can be aggregated by a script, which creates an *example_results.csv* file in the *res_folder*:

```
$ FT_TSU_HOME/build_example_table.sh --res_folder /path/to/results_folder
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 75 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

The following tables show the results extracted from the *example_results.csv* for regular dataflow execution (Table 3), pessimistic Double Execution (Table 4), and optimistic Double Execution (Table 5). For a better classification of the example execution, we also present the results for TERAFLUX nodes with 1, 2, 8, 16, and 32 cores. The results extracted from the *example_results.csv* are highlighted in yellow. Based on the execution times, the run-time overhead for pessimistic and optimistic Double Execution (compared to the baseline regular execution) can be additionally calculated. Since the objective is to depict the overhead solely induced by Double Execution, the overhead has been normalized to the regular execution time using half of the cores.

Table 3 – Node Utilization and Execution Time of the Baseline Dataflow Execution

Cores	Node Utilization [%]	Execution Time [ns]
1	99.9	34,762,104
2	99.9	17,769,355
4	99.7	9,209,017
8	98.4	4,864,722
16	96.7	2,550,796

Table 4 – Node Utilization and Execution Time of Pessimistic Double Execution

Cores	Node Utilization[%]	Execution Time[ns]	Runtime Overhead [%]
2	99.2	35,751,164	2.8
4	99.0	18,741,358	5.4
8	99.2	9,680,112	5.1
16	98.3	5,080,112	4.4
32	94.1	2,921,200	14.5

Table 5 – Node Utilization and Execution Time of Optimistic Double Execution

Cores	Node Utilization [%]	Execution Time [ns]	Runtime Overhead [%]
2	99.7	35,611,170	2.4
4	99.5	18,358,568	3.3
8	99.7	9,500,460	3.1
16	98.4	4,996,302	2.7
32	97.0	2,723,690	6.7

13.5 Further references to more in-depths

Please refer to Deliverable D5.4 for a deeper analysis of the fault tolerance mechanisms in TERAFLUX.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 76 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

14 Research Use Case from UCY

In this document the steps followed to integrate the DDM-Style TSU in the COTSon/SimNow simulation framework are described. The integration allows using the features of the TSU from a client code without having the TSU executing at user level. The DDM-style TSU has been integrated into COTSon by using as template the TSU version 2 developed in the project – namely *TSU2* (it integrates also a simplified timing model), and the TSU++ implementation for DDM-style execution. The *TSU2* operates as an intermediate API to provide communication between the user application and the simulator. A single queue has been used to store threads that are ready for execution and a FIFO policy for scheduling. The TSU does not operate in busy-wait mode but instead it is performing event-driven execution, which seems to make simulation faster.

14.1 Goal of the experiment or example

The goal of the experiment is to show the execution of a given benchmark application (i.e., in this case the *Cholesky decomposition* application) upon the TSU++ implementation for the TERAFLUX architecture using the DDM-style execution model.

The Data-Driven Multithreading Virtual Machine (DDM-VM) is a virtual machine that supports DDM execution on homogeneous and heterogeneous multicore systems. The DDM-VM is composed of:

- Thread Scheduling Unit (TSU), which is implemented as a software module executing on one of the cores. Such TSU model is written in C language;
- Run-time support system that (with the help of the TSU) handles the tasks of thread scheduling, execution instantiation and data management implicitly on the rest of the cores;

The TSU++ is a software implementation of the DDM-VM's TSU that is written in C++ language. It allows a programmer to write parallel data-driven programs using the object oriented styling. A program is described as a graph of tasks and dependencies between those tasks. The TSU++ also supports distributed execution on independent multi-core systems/nodes. For this functionality, a Network Interface Unit (NIU) is implemented as a software module that is executing on the same core as the TSU, as well as a Shared Global Address Space (S-GAS) is supported across all the nodes in the system to facilitate data movement.

Differences over DDM-VM's TSU

- The TSU++ it consists of C++ classes which have a well-defined purpose and are easy to test;
- Tasks are defined as functions; hence, there is no need for "*goto*" statements;
- The development of DDM programs is easier since there is no need to program using macros. All the programmer's TSU communication needs are accessible via a TSU object.

The TSU++ is supported also on Windows OS.

14.2 Location of the involved files

The directory containing all the involved files is located at:

```
$COTSONHOME/code/branches/timing-unisi/tsu.ddm
```

The directory containing the source code of the TSU++ implementation is located at:

```
$COTSONHOME/code/branches/timing-unisi/tsu.ddm/TSU
```

The directory containing the applications that can be run on COTSon is located at:

```
$COTSONHOME/code/branches/timing-unisi/tsu.ddm/App
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 77 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

14.3 Detailed instructions to start

The steps for integrating the TSU++ implementation of DDM-Style based on *TSU2* are the following:

➢ Download COTSon and SimNow
- o Download COTSon from COTSon Repository by typing in the shell:
  svn co https://svn.code.sf.net/p/cotson/code/ **$COTSONHOME**
  For Example: svn co https://svn.code.sf.net/p/cotson/code/ cotson
- o Download SimNow Simulator from:
  http://developer.amd.com/tools-and-sdks/cpu-development/simnow-simulator/
- o Uncompressed the SimNow file

➢ Configure and Install Cotson With TSU++
- o cd $COTSONHOME/branches/timing-unisi/trunk
- o sudo sysctl -w vm.max_map_count=4194304 (every time the system restarts)
- o sudo apt-get install ruby1.8 ruby1.9.1
- o sudo ./configure --simnow_dir *<the file where the SimNow is located>*
  For Example: sudo ./configure --simnow_dir ../../../../simnow-linux64-4.6.2pub/
- o sudo mount -o remount,size=8G /dev/shm (set the size of your RAM. Here it's 8GB)
- o cd $COTSONHOME/branches/timing-unisi/; sudo make build
- o Download the DDM file (tsu.ddm) from this URL:
  https://www8.cs.ucy.ac.cy/projects/ddmgroup/wp/teraflux/cotson/
- o Extract the file. You should have a folder named tsu.ddm
- o Move the tsu.ddm folder into this path: $COTSONHOME/branches/timing-unisi/
- o cd $COTSONHOME/branches/timing-unisi/tsu.ddm and execute:
  - ▪ make clean; make

➢ Executing DDM applications
- o Go to $COTSONHOME/branches/timing-unisi/tsu.ddm
- o Modify the *script.bash* file

```
$ xget $COTSONHOME/branches/timing-unisi/tsu.ddm/TSUClient ./TSUClient
$ chmod +x ./TSUClient
$ ./TSUClient 1 4 5 1024 32
```

The *script.bash* file contains the appropriate script code to execute the TSU's executable. Below is the content of the script.bash file. The command of the first line is responsible for transferring the executable (*TSUClient*) in the simulator. The command of the second line changes the permissions of the executable, i.e., it gives execution permissions to the current user. Finally, the command of the third line executes the DDM application in the simulator. The TSUClient takes the following arguments:

- • *Program Id*: it indicates the benchmark that the user wants to execute. For example, 0 corresponds to matrix multiply and 1 corresponds to Cholesky decomposition;
- • *Cores:* represents the number of cores;
- • *AQ Threshold*: it determines how many tasks will be given to the least loaded worker before checking for the next worker with the minimum load. The default is 5;
- • *Matrix Size*: is the size of the matrix to be used (valid only for specific benchmarks);
- • *Block Size*: another parameter considered only in specific benchmarks;
- • *Iterations*: it represents the number of times the user wants to execute the application (this argument is optional);

- o make run

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 78 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

14.4 Expected output

For the purpose of evaluation, the *Cholesky decomposition* application (which is one of the most complex applications available at the moment) has been chosen. Fig. 34 shows a screenshot for the execution of TSU++ on the COTSon simulator. The output timings are shown on the right.
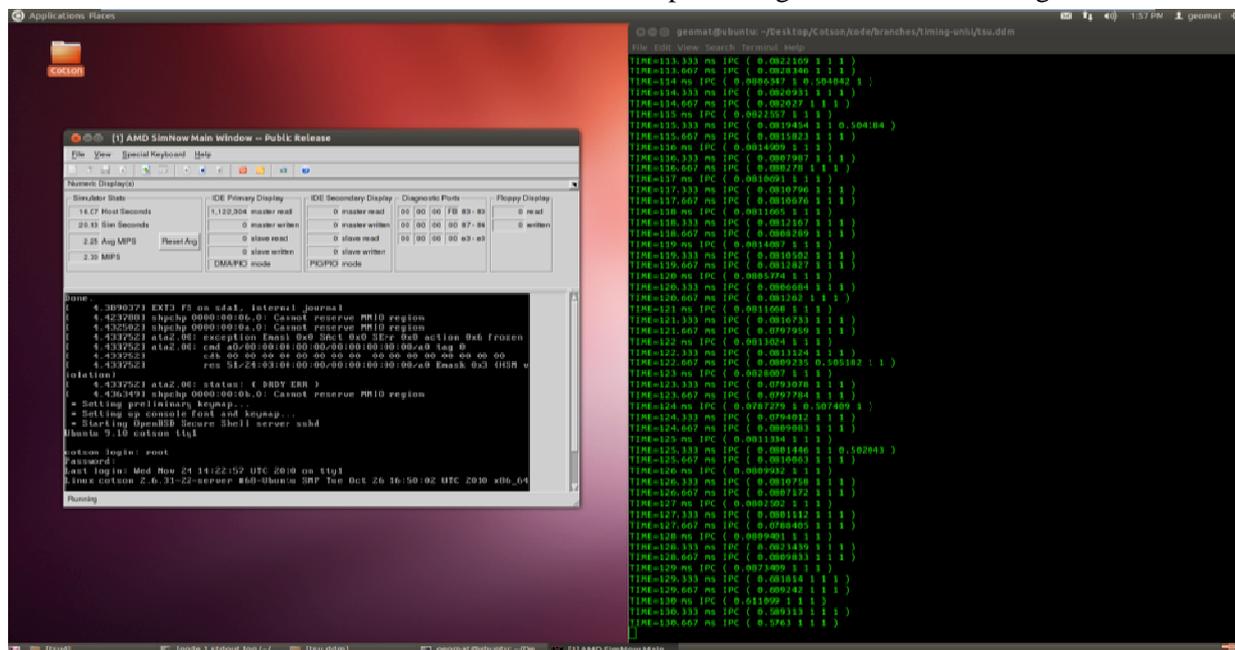

Fig. 34 - Executing TSU++ on COTSon.

The output is stored in the *node.1.stdout.log* file. It should display a content similar to the following:

```
Worker 0: stack 0xa2f000 16384
Worker 1: stack 0xa34000 16384
Worker 2: stack 0xa39000 16384
Worker 3: stack 0xa3e000 16384
Program: Cholesky decomposition, Cores 4, AQ threshold: 2, Matrix Size: 2048, BlockSize: 32
Deallocate worker frame at 0xa2f000
Deallocate worker frame at 0xa34000
Deallocate worker frame at 0xa39000
Deallocate worker frame at 0xa3e000
All workers done, goodbye
Speedup:     3.480233
Serial time:   24.089845
Parallel time: 6.921906
```

14.5 Further references to more in-depths

Further information and details about the TSU++ code is available in the deliverable D6.4.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 79 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

15 Research Use Case from UD

The *Delaware Adaptive Run-Time System* (DARTS) is a software implementation of the *Codelet Model* proposed by Zuckerman et al. [4], and presented in D9.1 and D9.2. It was written with two main objectives in mind: (1) to be a faithful implementation of the Codelet Model, and (2) to be modular, so that further research to explore fine-grain event-driven program execution models could be performed.

DARTS relies on the *hwloc library* [1] to map the topology of the underlying hardware to the Codelet abstract machine model required to specify how many synchronization units (similar to DF-Threads' thread scheduling units) and compute units (or cores) there should be, and how they should be physically grouped. It also relies on the lock-free data structures provided by Intel Threading Building Blocks [3] if they are present on the system for efficient work queuing.

Further details about the implementation of DARTS on the generic X86 architecture can be found in the Euro-Par publication [2] and in D9.3. A detailed explanation of the port of DARTS to the TERAFLUX simulation infrastructure, including a discussion of the necessary trade-offs, is also available in D9.3.

15.1 Goal of the experiment or example

This example demonstrates how to build and run examples that come with the port of DARTS on COTSon simulation infrastructure. In the following it will be demonstrated how to first build DARTS, then run the experiments. The focus will be on the *merge sort* example, however all the other experiments can be built using a similar methodology.

15.2 Location of the involved files

The archive for DARTS-TSUF can be found at:

```
$COTSON_ROOT/branches/ud-darts/darts-tsuf
```

The directory containing scripts to run the recursive Fibonacci sequence computation, Matrix Multiplication, and Merge Sort examples is located at:

```
$COTSON_ROOT/branches/ud-darts/scripts
```

15.3 Detailed instructions to start

The Merge Sort example can be run by typing the following commands. In the following, it is considered that the COTSon repository is located in the path pointed by the variable **$COTSON**. The directory where to install and run the experiments is pointed by the variable **$PATH_TO_EXPERIMENTS** (note that the two variables can be defined by the user).

- Building DARTS-TSUF. After having checked the COTSon's files out, do:

```
$ cd $PATH_TO_EXPERIMENTS/
$ mkdir $PATH_TO_EXPERIMENTS/darts-build
$ cd $PATH_TO_EXPERIMENTS/darts-build
$ cmake $COTSON_ROOT/branches/ud-darts/darts-tsuf
$ make
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 80 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

- Running the DARTS-TSUF merge-sort example. First, copy the scripts from the script folder as follows:

```
$ mkdir $PATH_TO_EXPERIMENTS/scripts
$ cd $PATH_TO_EXPERIMENTS/scripts
$ cp $COTSON_ROOT/branches/ud-darts/scripts/* .
```

Configure the *config.lua* script so that it points to the right *tflux_tsu.so* library, as well as the right script to run (in this example, *msort.sh*). Then edit *msort.sh*'s variables:

```
$ export OUTPUT_PATH=$PATH_TO_EXPERIMENTS
$ export DARTS_PATH=$PATH_TO_EXPERIMENTS/darts-build
$ export COTSON_PATH=$COTSON_ROOT/trunk/bin
$ ./launch.sh
```

15.4 Expected output

The output is stored in the *results.txt* file. It should display a content similar to the following:

```
DF owm 0x7ffff7674000 10000000
Creating 1 workers for 1 cores
Starting workers
Starting master node 1 nodes 1 workers 1
mergesort(500)
Done
Time:2.39678e+08 ns
Deallocate OWM at 0x7ffff7674000
All workers done, goodbye
============================================================
==================    DF STATS    =====================
df time:  240736779 ns (240.737 ms)
core  0:  23360631 insts 240736779 xc        0 ic, 240736779 cycles
```

The number of elements to be sorted is displayed (the example tries to merge 500 random numbers). If the simulation went through, the "Done" message is displayed, followed (on the next line) by the amount of in-simulation nanoseconds it took to run the experiment.

15.5 Further references to more in-depths

More details about the DARTS run-time and the Codelet model can be found in the deliverables D9.1, D9.2, and D9.3. Deliverable D9.3 also explain the process of porting the run-time on an x86-based TERAFLUX architecture.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 81 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

16 Research Use Case from UNIMAN

Our main goal is the design and implementation of Transactional memory (TM) system in the COTSon simulator. We have developed TM system that supports lazy and eager version management and conflict detection mechanism. The TM models have been extended and a scalable TM system has been developed. The scalable system is a purely lazy implementation but the commit process takes advantage of a hierarchical organization of cores into nodes. The committed changes are broadcasted within the node 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 directory based cache coherence protocol as a starting point for our baseline version.

In the following subsections, we will be explaining in detail of how to run our TM models in the COTSon simulator along with the directory based protocols on which our scalable TM version is based on.

16.1 Goal of the experiment or example

The main goal of the experiment is to show how to run different benchmarks on the TM system developed in COTSon. We will show how to run applications on scalable directory based simulator as well as the TM system implemented on top of the directory infrastructure. We will also be giving detailed description of running dataflow benchmarks with transactions running on the simulator. We will be showing how the TM model works along with the TSU to run dataflow plus transactional memory benchmarks.

16.2 Location of the involved files

The complete TM infrastructure is present in the following two locations.

```
$COTSONHOME/branches/tm-uniman
```

And

```
$COTSONHOME/branches/tflux-test/tsuf
```

First is the cache coherent NUMA architecture. The code for this directory based coherent architecture is present in:

```
$COTSONHOME/branches/tm-uniman/trunk/src
```

The configuration files for the scalable system are present in:

```
$COTSONHOME/branches/tm-uniman/trunk/src/example/uniman/cc_numa_tracer
```

The code for the TM system developed at uniman is present in

```
$COTSONHOME/branches/tm-uniman/trunk/src
```

And the configuration files are in

```
$COTSONHOME/branches/tm-uniman/trunk/src/example/uniman/tm_tracer
```

The code for the scalable TM system is present in

```
$COTSONHOME/branches/tm-uniman/trunk/src
```

And the configuration files are in

```
$COTSONHOME/branches/tm-uniman/trunk/src/example/uniman/tm_tracer_scalable
```

Finally the configuration files to run TM system along with the TSU to run dataflow plus transactional benchmarks are present in

```
$COTSONHOME/branches/tflux-test/tsuf/test
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 82 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

We will be looking at all these files and give example of running simple benchmarks on all these configurations in order to help the user in using our TM infrastructure for further experimentation.

16.3 Detailed instructions to start

The first step is to check out the full COTSon repository (including branches) and set `$COTSONHOME`:

```
$ svn co https://svn.code.sf.net/p/cotson/code cotson
$ export COTSONHOME=<installation_dir>/cotson
```

Next the user has to compile the main trunk and also the 'branches/tm-uniman/trunk':

```
$ cd $COTSONHOME/trunk
$ ./configure –simnow_dir <path_to_simnow_installation>
```

if the configure terminate successfully than just type:

```
$ make
```

Again for "branches/tm-uniman/trunk":

```
$ cd $COTSONHOME/branches/tm-uniman/trunk
$ ./configure –simnow_dir <path_to_simnow_installation>
$ make
```

Running benchmarks on Scalable ccNUMA architecture

In order to run scalable directory based *ccNUMA* architecture we need to configure the COTSon simulator:

```
$ cd $COTSONHOME/branches/tm-uniman/trunk/src/examples/uniman/cc_numa_tracer
```

The main file that configures the system is *cotson_tracer.in*. Fig. 35, shows the snapshot of that configuration file.

Fig. 35 – Configuring ccNUMA architecture in COTSon.

The configuration file sets up the number of nodes in the system *totalNumOfNodes* as well as total number of cores in each node. It also sets up the directory structure and the protocol being used to implement coherency.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 83 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

In the same directory there is the file *run.sh*, which contains paths of all the benchmarks that need to run on the simulator (in the examples directory there are several benchmarks, in this case the default is Micro-Benchmarks/microtest). In order to run benchmarks the user just needs to type *make*. The result containing all the execution statistics is saved in the log file after the simulation exits successfully, in the same directory.

Running benchmarks on TM architecture

Configuration files for TM architecture are reached by issuing:

```
$ cd $COTSONHOME/branches/tm-uniman/trunk/src/examples/uniman/tm_tracer
```

cotson_tracer.in file configures the simulator to run TM benchmarks. Fig. 36 shows the screenshot of that configuration file.


```
print("Creating bus and busT for node " .. j)
bus=Bus{ name="bus", protocol='TM_Coherence', latency=25, bandwidth=4, next=l3, node_id=j }
busT=Bus{ name="tlb_bus", protocol='TM_Coherence', latency=25, bandwidth=4, next=mem, node_id=j }


k=0
while k < cpusPerNode do

        print ("Creating cpu Id " .. i .. " for node Id " .. j)

        cpu=get_cpu(i)
        cpu:timer{ name='cpu'..i, type="timer0" }

        l2=Tm_Cache{ name="l2cache", size="256kB",
                    line_size=64, latency=10,
                    num_sets=16, next=bus,
                    write_policy="WB", write_allocate="true", sharing="private", node_id=j, tm_protocol=tmProtocolType  }

        ic=Tm_Cache{ name="icache", size="64kB",
                    line_size=64, latency=2,
                    num_sets=16, next=l2,
                    write_policy="WT", write_allocate="false", sharing="private", node_id=j  }

        dc=Tm_Cache{ name="dcache", size="64kB",
                    line_size=64, latency=2,
                    num_sets=16, next=l2,
                    write_policy="WT", write_allocate="false", sharing="private", node_id=j, tm_protocol=tmProtocolType  }
```

Fig. 36 – Configuring TM architecture in COTSon.

As shown in the figure, the configuration file sets up the TM protocol. It configures the network and the caches used in implementing TM protocol. The caches are modified to contain extra information for saving and committing transactional data.

In the same directory there is the file *run.sh*, which contains paths of all the benchmarks that need to run on the simulator (in this case, the path to *vacation* binary). In order to run benchmark the user just needs to type *make*. The result containing all the statistics of the execution is saved in the log file after the simulation exits successfully, in the same directory.

Running benchmarks on Scalable TM System

Scalable TM system builds on top of directory based protocols. The configuration files to implement the scalable TM system are reached by issuing:

```
$ cd $COTSONHOME/branches/tm-uniman/trunk/src/examples/uniman/tm_tracer_scalable
```

cotson_tracer.in file configures the simulator to run TM benchmarks. Fig. 37 shows the screenshot of the configuration file.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 84 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

As shown in the figure, the configuration file sets up the scalable TM protocol. It configures the network, the caches and the directories used in implementing TM protocol. The caches and the directories are modified to contain extra information for saving and committing transactional data and implementing the TM protocol. Directories are configured to implement the TM protocol rather than conventional coherence protocol.

To run the benchmark (*Micro-Benchmarks/microtest* in this case) the user has to do a *make*. The file *run.sh* contains paths of all the benchmarks and the log file contains all the stats of the execution.

Fig. 37 – Configuring TM architecture in COTSon.

Running dataflow plus TM benchmark in COTSon using TSU and TM hardware

This section explains how to set up the simulator so that it has both the TSU and TM hardware working together to run applications that have dataflow and transaction properties.

In order to run dataflow and transaction benchmarks, the COTSon simulator needs to implement the TSU hardware as well as TM hardware so that both aspects of the applications can be handled in hardware for greater efficiency.

The configuration files to set up TM mechanism along with TSU hardware are reached by issuing:

```
$ cd $COTSONHOME/branches/tflux-test/tsuf
$ make
$ cd $COTSONHOME/branches/tflux-test/tsuf/test
$ make run_htm_single     (or make run_htm_multi)
```

There are two configuration files *tsu_tm_single.lua* and *tsu_tm_multi.lua* to run single node and multi node simulation respectively. The user has to do a *make* run_htm_single or make run_htm_multi. The snapshot of the make file in shown in Fig. 38.

As shown in the figure, the makefile sets up TM running on single and multi-node with the TSU hardware. The *tsu_tm_single.lua* and *tsu_tm_multi.lua files* configure the network, the caches and the directories used in implementing TM protocol.

To run the benchmark the user has to do a *make*. HTMTESTS variable in the makefile contains the list of the benchmarks to run. The log file contains all the stats after the execution exits successfully.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 85 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

```
HTMTESTS = tmtest_htm

all: $(TESTS)

run_htm_single: clean $(HTMTESTS)
    echo "sudo sysctl -w kernel.randomize_va_space=0" > $(TSCRIPT); \
    echo "xget $(PWD)/config_multi.sh cfg.sh; chmod +x cfg.sh" >> $(TSCRIPT); \
    echo "sudo sh -x cfg.sh" >> $(TSCRIPT); \
    echo "export DF_WWORKERS=$(NT)" >> $(TSCRIPT); \
    echo 'export DF_OWMSZ=$(OWMSZ)' >> $(TSCRIPT); \
    for n in $(HTMTESTS); do \
        echo "xget $(PWD)/$$n $$n; chmod +x $$n; echo ''; echo 'RUNNING $$n'" >> $(TSCRIPT); \
        echo "./$$n $(SZ) 2>&1 | tee -a LOG; xput LOG $(PWD)/LOG" >> $(TSCRIPT); \
        echo "sleep 0.1" >> $(TSCRIPT); \
    done; \
    echo "touch terminate; xput terminate terminate" >> $(TSCRIPT); \
    $(COTSON) $CRIPT="'$(TSCRIPT)'" TSUSIM="'$(TSUSIM)'" tsu_tm_single.lua

run_htm_multi: clean $(HTMTESTS)
    echo "sudo sysctl -w kernel.randomize_va_space=0" > $(TSCRIPT); \
    echo "export DF_WWORKERS=$(NT)" >> $(TSCRIPT); \
    echo 'export DF_NODEID=$$1' >> $(TSCRIPT); \
    echo 'export DF_NNODES=$$2' >> $(TSCRIPT); \
    echo 'export DF_OWMSZ=$(OWMSZ)' >> $(TSCRIPT); \
    for n in $(HTMTESTS); do \
        echo "xget $(PWD)/$$n $$n; chmod +x $$n; echo ''; echo 'RUNNING $$n'" >> $(TSCRIPT); \
        echo "./$$n $(SZ)" >> $(TSCRIPT); \
        echo "sleep 1" >> $(TSCRIPT); \
    done; \
    $(COTSON) $CRIPT="'$(TSCRIPT)'" TSUSIM="'$(TSUSIM)'" tsu_tm_multi.lua
```

Fig. 38 – Makefile to setup TM and TSU hardware for single and multimode simulation.

16.4 Expected output

This section explains some of the output files that are generated when the execution successfully exits. We will also be showing some screen shots to show the execution in progress and the output that should be expected when running the benchmarks.

Running benchmark on Scalable ccNUMA architecture

Fig. 39 shows the devices when running ccNUMA COTSon simulation. The *cotson_tracer.in* sets up the number of cores in the system as shown in Fig. 40. In this example the number of cores is 4, which is reflected in Fig. 39. The log file is generated when the execution successfully exits. Fig. 41shows the snapshot of the log file, which is generated when the matrix multiplication example finished execution. The log file shows the cache stats of the simulation running with 4 cores.
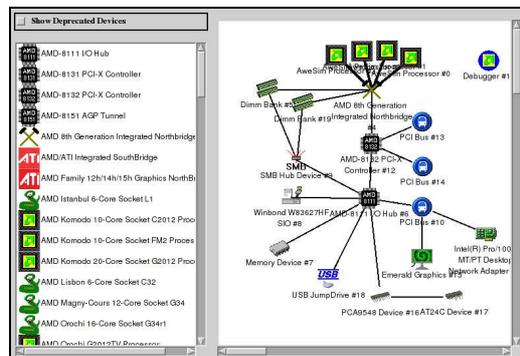

Fig. 39 – Device window while running COTSon simulation

Fig. 40 – cotson_tracer.in configuration file setting up the number of cores in the simulated machine

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 86 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Fig. 41 – Log file showing icache statistics for the cpu 0.

Running benchmarks on TM architecture

Fig. 42 shows the COTSon simulation running *vacation* transactional memory benchmark. As you can see in the figure the number of commits and aborts are printed in the console.
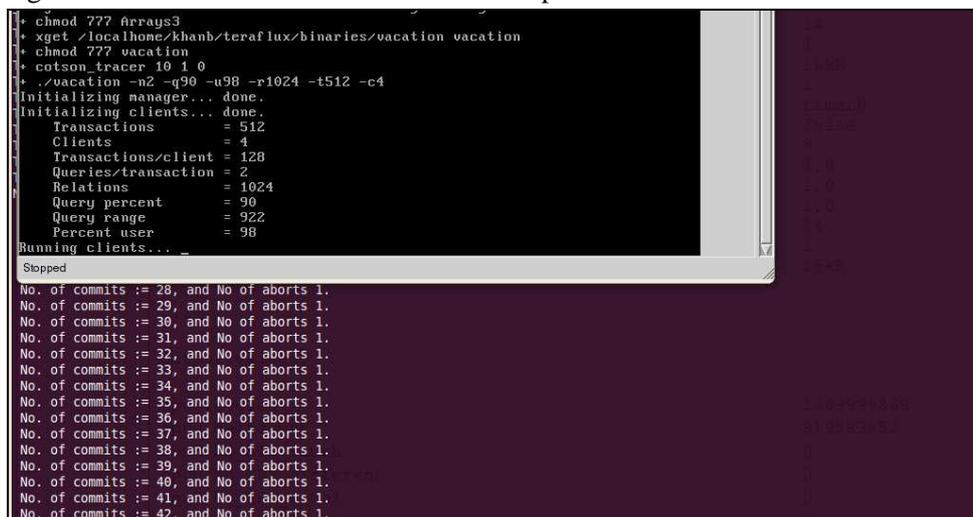
Fig. 42 – COTSon graphical main window and the console output.

The output of the benchmark is printed on the COTSon main graphical window. Finally the simulation stats are written to the log file that is created in the same folder where the configuration files are present.

Running benchmarks on Scalable TM architecture

Fig. 43 shows COTSon simulation running *Genome* benchmark. The figure shows how the scalable TM system is configured containing many nodes, distributed memory structure and a shared L3 cache within each node.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 87 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Fig. 43 – Configuring the scalable TM architecture in COTSon.

This configuration is setup in the *cotson_tracer.lua* configuration file. The structure of the system can be changed by making modifications in the lua file. The user can increase or decrease the number of cores within a node. The levels of cache hierarchy, the directory and network structure can also be configured. The log file is created when the simulation exits successfully.
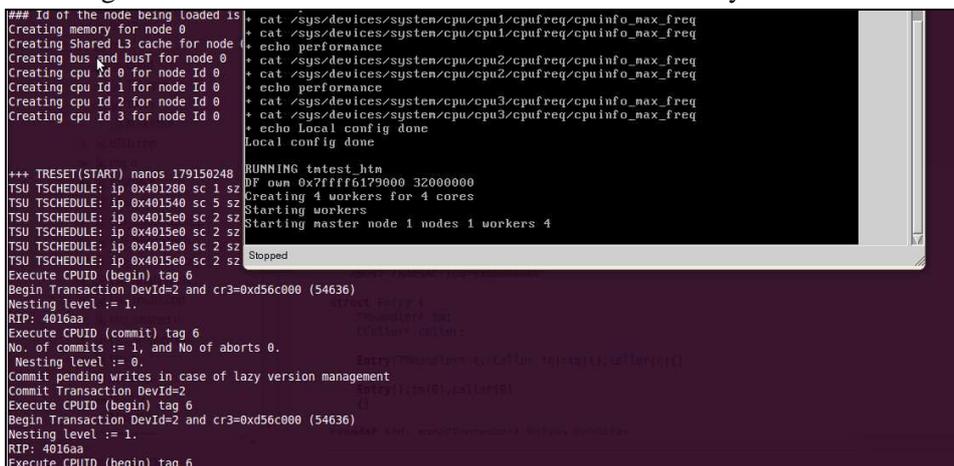

Fig. 44 – COTSon simulation setting up and running TM and TSU hardware.

Running dataflow plus TM benchmark in COTSon using TSU and TM hardware

The final experiment we will show in this report is how to run TM hardware along with the TSU hardware for benchmarks that have transactions and dataflow properties. Fig. 44 shows the COTSon simulation configuring and then running a simple micro benchmark using the TM and TSU hardware. The dataflow instructions are handled by the TSU hardware and the transactional memory instructions are handled by the TM hardware. The log file is created at the end, with all the simulation statistics.

16.5 Further references to more in-depths

Refer to previous deliverables (D7.4, D7.3, D7.2 and D7.1) for more details about the TM models and their integration with the common simulation platform.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 88 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

17 Research Use Case from UNISI

One of the main building blocks of the TERAFLUX project is the implementation of the Thread Scheduling Unit (TSU) model, running in the COTSon simulation platform. As result of the research activity, several versions of the TSU model has been implemented and made available to the other partners. The two most stable versions at the current moment are the TSUF and the TSU4. Both of them allow the execution of dataflow benchmark kernels (such as the *recursive Fibonacci*, and *Matrix Multiplication*) both on a single node simulated system, and a multi-node simulated system. The purpose of the TSU model is the scheduling of dataflow threads (namely DF-Threads) among the available cores, as expected from the hardware counterpart.

17.1 Goal of the experiment or example

The main goal of the experiment is to show how to run a dataflow benchmark application using the TSU model developed within the COTSon simulator. To this end, the following subsections describe how to run a simple test using the TSU4 model (for the TSUF implementation, refers to the chapter 9, sections from 9.1 to 9.5). The experiment allows the user to understand how the scheduling unit model has been integrated in the simulation platform, and which information it provides to the user.

17.2 Location of the involved files

The scheduling unit model is distributed in a dedicated directory contained in the *branches* folder:

```
$COTSONROOT/branches/timing-unisi/tsu4
```

17.3 Detailed instructions to start

As an example, detailed instructions to run the *recursive Fibonacci* benchmark kernel on the TSU4 model of the thread scheduling unit will be provided. This benchmark is used to stress the thread scheduling unit since it is able to generate a huge number of DF-Threads even for a small size of the input. In order to run the example, move on the correct folder:

```
$ cd $COTSONROOT/branches/timing-unisi/tsu4
```

Open the *Makefile* file with a text editor and check that the first line is correctly pointing the source folder in the trunk COTSon folder. Then, in the same file set the variable *TESTS* to *fib*, in order to run the selected benchmark:

```
$ vim Makefile
ROOT=../../../trunk/src
DATE=$(shell date +%s)
PWD=$(shell pwd)
MCAST=$(shell expr 1 + $(DATE) % 250)
DEBUG=1
TESTS = fib

all: tsu_monitor.o tsu_manager.o tflux_tsu.so tsumon $(TESTS)
...
```

Open the *run_script.sh* file with a text editor. In the opened file set the variable *TESTS* to *fib*, in order to run the selected benchmark. In order to properly set the configuration of the simulated system (i.e., size of the input of the benchmark, number of cores, etc.), the following variables must be checked: *NUM_NODE* defines the number of nodes composing the system, *CORES* defines the number of cores in each node, *SZ* and *MT_SIZE* define the input size for the used benchmark (*SZ* refers to the *Fibonacci* kernel, while *MT_SIZE* refers to the *Matrix Multiplication* kernel). In this example the *Fibonacci* kernel with 14 as the input size is run. *SH_MEM* variable defines the name of the object in the host system used to implement the shared memory across the nodes. Finally, *OUTPUT* variable point to the folder where the simulation output will be recorded (set also *TSU_STATS*, *SCRIPT*, and *REPORT_DIR* variables).

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 89 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

```
$ vim run_script.sh
!/bin/bash

# number of nodes
NUM_NODE=1
# benchmarks
TESTS="fib"
# number of cores per node
CORES=4
TTCORES=$(($CORES*$NUM_NODE))
#for fibonacci
SZ=14
#for matrix multiply
MT_SIZE=512 # matrix size
#shared memory name (unique for each simulation)
SH_MEM="DTHREADSharedMemory"

if [[ $SH_MEM ]]; then
    export DTHREAD_OBJ=$SH_MEM"1"
    export DTHREAD_READY_OBJ=$SH_MEM"2"
    export DTSU_SYNC_OBJ=$SH_MEM"3"
fi

BIN_BENCH_DIR=$PWD

if [ -z $OUTPUT ] ; then OUTPUT=./S-LOG ; fi

SCRIPT="$OUTPUT/script"
TSU_STATS="$OUTPUT/stats"
FILE_LAST_LOG="file_last"
REPORT_DIR="$OUTPUT/report"
...
```

The Lua configuration file is set to run a timing simulation (*sampler* object is set to *simple*) of the target system:

```
$ vim tsu.lua
abaeterno_so="tflux_tsu.so"
wd=os.getenv("PWD")

tmpdir=wd
runid="tsu"
-- clean_sandbox=false

options = {
        --max_nanos='3G',
        exit_trigger='terminate',
        -- sampler={type="no_timing", quantum="10M"},
        sampler={type="simple", quantum="10M"},
        heartbeat={ type="file_last", logfile=runid..".log" },
        custom_asm=true,
        tsu_ignore_errors=true,
        -- tsu_speculative_threads=true,
        -- tsu_statfile="/tmp/xx.dat",
}

one_node_script="run_interactive"
-- display=os.getenv("DISPLAY")
copy_files_prefix=runid.."."
-- clean_sandbox=false

simnow.commands=function()
        -- use_bsd('32p.bsd')
        use_bsd('4p.bsd')
        -- use_bsd(BSDS)
        use_hdd('karmic64.img')
        --use_hdd('debian.img')
        set_journal()
    send_keyboard('xget '..SCRIPT..' script')
    send_keyboard('sh -x script | tee LOG 2>&1')
end

function build()
        i=0
...
```

At this point is possible to launch the simulation. To this end, the reader needs to open two console windows. In the first console (after moving in the *$COTSON-ROOT/branches/timing-unisi/tsu4*) the

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 90 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

reader launches the external monitor (i.e., the object that is used to manage the shared memory across the nodes)

```
$ make run_tsumon
```

Once the monitor is running, the following output should be presented:

```
$Booting TSU Monitor ...
$Start TSU Monitor
$TSU Monitor is configured with 1 nodes
$TSU Monitor is initializing shared memory (DTHREADSharedMemory1) $....
$TSU Monitor is initializing ready shared memory (DTHREADSharedMemory2) ....
$TSU Monitor is initializing sync shared memory (DTHREADSharedMemory3) ....
$TSU message queue m2n(DTHREADSharedMemory1mq_mon2node0) for node(0) is initializing....
$TSU message queue n2m(DTHREADSharedMemory1mq_node2mon0) for node(0) is initializing....
$Initialization for shared memory finished!
```

Finally, on the second console the user launches the benchmark execution as follows:

```
$ make run
```

17.4 Expected output

The following files are involved in the output process. The file *node.1.tsu.log* contains the statistics gathered by COTSon during the simulation:

```
Input values:
cpu0.bpred_perfect                      false
cpu0.branch_mispred_penalty             8
cpu0.commit_cpi                         1.0
cpu0.dcache.fudge                       1.0
cpu0.icache.fudge                       1.0
cpu0.twolev.hlength                     14
cpu0.twolev.l1_size                     1
cpu0.twolev.l2_size                     16kB
cpu0.twolev.use_xor                     1
cpu0.type                               timer0
cpu1.bpred_perfect                      false
cpu1.branch_mispred_penalty             8
cpu1.commit_cpi                         1.0
cpu1.dcache.fudge                       1.0
cpu1.icache.fudge                       1.0
cpu1.twolev.hlength                     14
cpu1.twolev.l1_size                     1
cpu1.twolev.l2_size                     16kB
...
Output values:
cpu0.cycles                             149999985
cpu0.haltcount                          108195301
cpu0.hb_ATC_flush                       67
cpu0.hb_CR3_different                   36
cpu0.hb_CR3_equal                       31
cpu0.hb_ev_Exception                    692
cpu0.hb_ev_HW_interrupt                 219
cpu0.hb_ev_SW_interrupt                 0
cpu0.idlecount                          112802301
cpu0.instcount                          24655697
cpu0.invalid_translation_bytes          1936557
cpu0.iocount                            4069258
cpu0.metadata_bytes                     10468840
cpu0.other_exceptions                   210511
cpu0.plain_invalidations                2988
cpu0.range_invalidations                32
cpu0.read_mmios                         368
cpu0.read_pios                          1062
cpu0.segv_exceptions                    0
cpu0.timer.cycles                       37823009
cpu0.timer.instructions                 24147071
cpu0.timer.twolev.lookup                2048709
cpu0.timer.twolev.misses                83286
cpu0.timer.twolev.reset                 0
cpu0.timer.twolev.update                2048709
cpu0.trace_cache_size                   0
cpu0.valid_translation_bytes            90649248
cpu0.write_mmios                        564
cpu0.write_pios                         4169
cpu1.cycles                             149999985
cpu1.haltcount                          121463351
cpu1.hb_ATC_flush                       24
cpu1.hb_CR3_different                   1
cpu1.hb_CR3_equal                       23
cpu1.hb_ev_Exception                    504
cpu1.hb_ev_HW_interrupt                 30
cpu1.hb_ev_SW_interrupt                 0
cpu1.idlecount                          121840334
...
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 91 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

The file *terminal_fib_0_4* (enter in the subfolder *S-LOG* – see the *Makefile* configuration in the previous subsection) contains the output generated by the benchmark and the simulator during the simulation:

```
Loading module abaeterno.so.
Using image path: "/home/scionti/Tools/cotson-release/trunk/data"
Known Device: Deerhound RevB QuadCore Socket L1
Known Device: Intel(R) Pro/1000 MT/PT Desktop Network Adapter
Known Device: USB JumpDrive
Known Device: AMD-8111 I/O Hub
Known Device: AMD-8131 PCI-X Controller
Known Device: AMD-8132 PCI-X Controller
Known Device: AMD-8151 AGP Tunnel
Known Device: Debugger
...

1 exec> open /home/scionti/Tools/cotson-release/trunk/data/4p.bsd
Opening "/home/scionti/Tools/cotson-release/trunk/data/4p.bsd"
created device Machine
Instructions per Microsecond: 3000
CPU Model Name: Opteron
System Bus Frequency: 100
CPU Clock Mul: 4
Turbo_Port61: 0
Turbo_Vsync: 0
Guard Memory Required: TRUE
CPU Manages Cycles: TRUE
Disk Block Cache Size: 64K
Disk Block Cache Depth: 5
Disk Block Cache Bits: 12
info: creating device #0 "AMD 8th Generation Integrated Northbridge"
info: creating device #1 "Dimm Bank"
info: creating device #2 "AMD-8111 I/O Hub"
ATA: Image [/home/scionti/Tools/cotson-release/trunk/data/karmic64.img] does not have an ID field.
info: creating device #3 "Memory Device"
info: creating device #4 "Winbond W83627HF SIO"
...
BSD Load completed!

1 exec> ide:0.image master /home/scionti/Tools/cotson-release/trunk/data/karmic64.img
ATA: Image [/home/scionti/Tools/cotson-release/trunk/data/karmic64.img] does not have an ID field.
MASTER drive Image file is now /home/scionti/Tools/cotson-release/trunk/data/karmic64.img

1 exec> ide:0.journal master on
Journaling was already enabled

1 exec> keyboard.key 2D AD

1 exec> keyboard.key 22 A2

1 exec> keyboard.key 12 92

1 exec> keyboard.key 14 94

1 exec> keyboard.key 39 B9

1 exec> keyboard.key 34 B4

1 exec> keyboard.key 35 B5
...
1 exec> go
TIME=3.33333 ms IPC ( 0.993879 0.707539 1 1 )
TIME=6.66667 ms IPC ( 0.991326 0.98112 1 1 )
TIME=10 ms IPC ( 1 1 1 1 )
TIME=13.3333 ms IPC ( 0.958046 0.8369 1 0.814146 )
TIME=16.6667 ms IPC ( 0.99788 1 1 0.99697 )
TIME=20 ms IPC ( 0.968307 0.966541 1 0.995992 )
TIME=23.3333 ms IPC ( 0.774968 0.774076 1 0.982427 )
TIME=26.6667 ms IPC ( 0.995373 1 1 0.965398 )
TIME=30 ms IPC ( 1 1 1 1 )
TIME=33.3333 ms IPC ( 0.998995 0.999206 1 1 )
TIME=36.6667 ms IPC ( 1 1 1 0.99992 )
TIME=40 ms IPC ( 1 1 1 1 )
TIME=43.3333 ms IPC ( 0.99907 0.999325 1 1 )
TIME=46.6667 ms IPC ( 1 1 1 0.999914 )
TIME=50 ms IPC ( 1 1 1 1 )
TIME=53.3333 ms IPC ( 0.999072 0.999391 1 1 )
TIME=56.6667 ms IPC ( 1 1 1 0.99992 )
TIME=60 ms IPC ( 1 1 1 1 )
TIME=63.3333 ms IPC ( 0.998833 0.999323 1 1 )
TIME=66.6667 ms IPC ( 1 1 1 1 )
TIME=70 ms IPC ( 1 1 1 1 )
TIME=73.3333 ms IPC ( 0.998844 0.998883 1 1 )
TIME=76.6667 ms IPC ( 1 1 1 1 )
TIME=80 ms IPC ( 1 1 1 1 )
TIME=83.3333 ms IPC ( 0.998409 0.999379 1 1 )
TIME=86.6667 ms IPC ( 1 1 1 1 )
TIME=90 ms IPC ( 1 1 1 1 )
TIME=93.3333 ms IPC ( 0.998437 0.999356 1 1 )
TIME=96.6667 ms IPC ( 1 1 1 1 )
TIME=100 ms IPC ( 1 1 1 1 )
TIME=103.333 ms IPC ( 0.998433 0.999404 1 1 )
TIME=106.667 ms IPC ( 1 1 1 1 )
TIME=110 ms IPC ( 1 1 1 1 )
...
```

17.5 Further references to more in-depths

Refer to previous deliverables (D7.4, D7.3, D7.2 and D7.1) for more details about the TSU models and their integration with the common simulation platform.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 92 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

18 DRT - A tool for native testing of T* based programs

DARTS is not the only research effort for providing an efficient way to execute application on large computing systems. Looking towards building exascale systems (e.g., next generation supercomputers, large data-centers, etc.), the OCR project (Open Community Run-time Framework for Exascale Systems [5]) has been set up by Intel and other academic and industrial partners. The main objective of the OCR project is the implementation from the scratch (but reusing as much as possible current design aspects of run-time systems) of a software level, which is able to help meeting the requests of future exascale systems (i.e., high performance, low power consumption, use of different programming models and languages, etc.). This piece of software should provide a clear and common interface for both the upper side software modules, and the hardware infrastructure.

On the same direction, but with different goals in mind, the TERAFLUX project proposed the Dataflow Run-Time – DRT. In particular, with the aim of facilitating the development and debugging of dataflow-oriented applications using the T* ISA extension, within the TERAFLUX project, a run-time library (DRT) has been devised. DRT is a piece of agile software that helps in providing very efficient environment to run programs with a dataflow execution model. It is organized as a library. The library is intended to be linked with the application source code, allowing the execution of the application directly on the host system. More specifically, the run-time exposes the same interface of the library used within the simulator to execute dataflow applications. The library contains functions that wrap T* instructions. Similarly, the DRT contains functions that reproduce the same functional behavior of their T* equivalent (cf. deliverables D7.1, D7.2, and D7.3 to deeply analyze the T* Instruction Set Extension). The run-time Application Programming Interface (API) has been designed to provide a two-way mechanism in which it supports the development of an efficient compiler and on another side, to provide for a good architectural support.

In the proposed approach, the DRT allows showing how easily can be to harness the maximum capacity of the computing nodes in the TERAFLUX project using the dataflow execution model. The main objective to provide this piece of software is to show users that DRT can easily provide a very small and powerful run-time, for executing different piece of codes that are coded in different programming model, but how easily can be executed in a dataflow style.

18.1 Goal of the experiment

DRT provides a simple script file for the "first time" whole checking. Currently, some initial examples have been tested, from simple (like the classical recursive Fibonacci sequence computation and matrix multiplication). DRT contains some environment variables that help the user to retrieve more information during the dataflow application execution. Two of them are: DRT_DEBUG and DRT_FSIZE. DRT_DEBUG can be used to get more detailed information about the current execution. DRT_FSIZE is used to set the size of internal frame (allocated memory) queue.

18.2 Location of the involved files

The source code is uploaded for public access in following repository. The repository is available at:

```
http://sourceforge.net/projects/drt
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 93 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

18.3 Detailed instructions to start

In this section, it will be shown one sample example, and how to download and compile DRT in a Linux-based system.

Step 1: the user needs to download the code from the repository. User can access the source code from its Linux terminal executing the svn command. In the terminal just type:

```
$ svn checkout svn://svn.code.sf.net/p/drt/code/ drt-code
$ cd drt-code
```

Pressing the enter key will start the download process (which can be seen in the below snapshot).

Fig. 45 – A DRT snapshot showing the download process.

Step2: The user can notice the script file *tregression.sh*, which can be used to check whether all the files are compiled successfully or not. After executing this script, it will generate one reference file and one output file for each example. The reader can also control the debugging information level by exporting a new variable called DRT_DEBUG.

```
$ ./tregression.sh
```


Fig. 46 – A DRT snapshot showing the result of the tregression.sh script. During the compilation process, it is produced in output an OK message (if no error is encountered)

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 94 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

18.4 Expected output

The final step will be to check a simple example: the recursive calculation of the Fibonacci sequence. The program calculates the 15th Fibonacci number implementing the dataflow execution model.

```
File  Edit  View  Search  Terminal  Help
[mazumdar@sarc8 drt-code]$ ./fib4 15
computing fibonacci(15)
result = 610, expected = 610, SUCCESS
[mazumdar@sarc8 drt-code]$ []
```

Fig. 47 – DRT example execution: recursive Fibonacci sequence with input set to 15 and debug level set to 0.

As shown in Fig. 47, the program terminates with a correct result. As already mentioned, DRT can also provide detailed information using the DRT_DEBUG variable. The level of verbosity can be increased using the increasing numbers (i.e., 0, 1, 2, 3, etc.). In the above example, the environmental variable has been set to 0, by exporting it as *DRT_DEBUG=0*. It is worth noting that 0 corresponds to the default debug value. To increase the verbosity level, just set the debug value to 1 (i.e., export the variable as DRT_DEBUG=1). Fig. 48 shows the result of the program execution with the new debug level set.
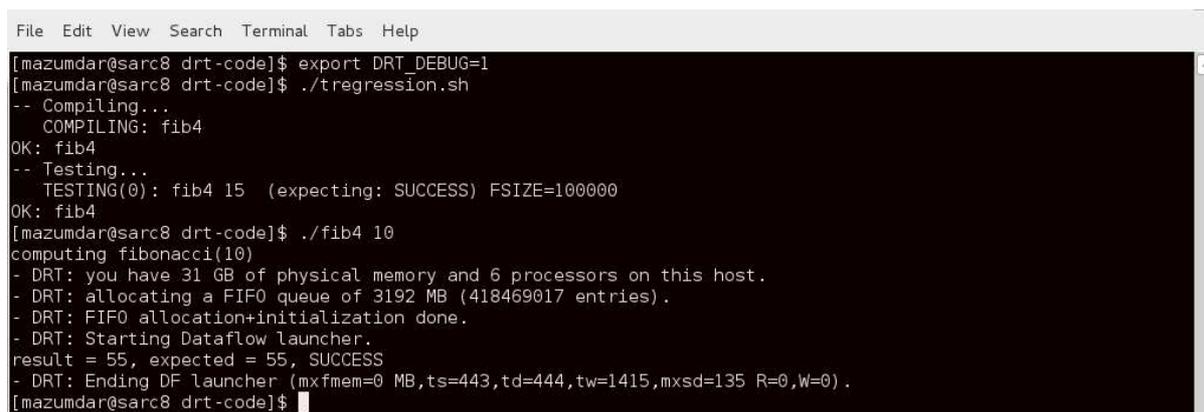
```
File  Edit  View  Search  Terminal  Tabs  Help
[mazumdar@sarc8 drt-code]$ export DRT_DEBUG=1
[mazumdar@sarc8 drt-code]$ ./tregression.sh
-- Compiling...
   COMPILING: fib4
OK: fib4
-- Testing...
   TESTING(0): fib4 15  (expecting: SUCCESS) FSIZE=100000
OK: fib4
[mazumdar@sarc8 drt-code]$ ./fib4 10
computing fibonacci(10)
- DRT: you have 31 GB of physical memory and 6 processors on this host.
- DRT: allocating a FIFO queue of 3192 MB (418469017 entries).
- DRT: FIFO allocation+initialization done.
- DRT: Starting Dataflow launcher.
result = 55, expected = 55, SUCCESS
- DRT: Ending DF launcher (mxfmem=0 MB,ts=443,td=444,tw=1415,mxsd=135 R=0,W=0).
[mazumdar@sarc8 drt-code]$ █
```

Fig. 48 – DRT example execution: recursive Fibonacci sequence with input set to 15 and debug level set to 1.

So, by increasing this verbosity level the user can retrieve more information about the current execution.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 95 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

References

[1] Broquedis, F.; Clet-Ortega, J.; Moreaud, S.; Furmento, N.; Goglin, B.; Mercier, G.; Thibault, S.; Namyst, R., *hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications*, Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on , vol., no., pp.180,186, 17-19 Feb. 2010.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5452445&isnumber=5452403

[2] Joshua Suetterlein, Stéphane Zuckerman, and Guang R. Gao. 2013. An implementation of the codelet model. In Proceedings of the 19th international conference on Parallel Processing (Euro-Par'13), Felix Wolf, Bernd Mohr, and Dieter Mey (Eds.). Springer-Verlag, Berlin, Heidelberg, 633-644. DOI=10.1007/978-3-642-40047-6_63.
URL: http://dx.doi.org/10.1007/978-3-642-40047-6_63

[3] Thomas Willhalm and Nicolae Popovici. 2008. Putting intel® threading building blocks to work. In Proceedings of the 1st international workshop on Multicore software engineering (IWMSE '08). ACM, New York, NY, USA, 3-4. DOI=10.1145/1370082.1370085.
URL: http://doi.acm.org/10.1145/1370082.1370085

[4] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. 2011. Using a "codelet" program execution model for exascale machines: position paper. In Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT '11). ACM, New York, NY, USA, 64-69. DOI=10.1145/2000417.2000424. URL: http://doi.acm.org/10.1145/2000417.2000424

[5] The Open Community Runtime Framework for Exascale Systems.
URL: https://01.org/open-community-runtime

[6] Solinas M., Badia, R.M., Bodin F., Cohen, A., Evripidou, P., Faraboschi P., Fechner B., Gao G.R., Garbade A., Girbal S., Goodman D., Khan B., Koliai S., Feng Li, Lujan M., Morin L., Mendelson A., Navarro N., Pop A., Trancoso P., Ungerer T., Valero M., Weis S., Watson I., Zuckermann S., Giorgi R. "The TERAFLUX Project: Exploiting the DataFlow Paradigm in Next Generation Teradevices". 16th Euromicro Conference on Digital System Design, September 2013, doi: 10.1109/DSD.2013.39

[7] Aaron Landwehr, Stephane Zuckerman, Guang R. Gao "Toward a Self-aware System for Exascale Architectures", In Proceedings of Euro-Par 2013: Parallel Processing Workshops; the 1st Workshop on Runtime and Operating Systems for the Many-core Era (ROME 2013), Aachen, Germany, August 2013.

[8] Joshua Suettlerlein, Stephane Zuckerman, Guang R. Gao: "An Implementation of the Codelet Model". Euro-Par 2013, Aachen (Germany), August 2013, doi: 10.1007/978-3-642-40047-6_63.

[9] Bernhard Fechner, Arne Garbade, Sebastian Weis, Theo Ungerer: "Fault Detection and Tolerance Mechanisms for Future 1000 Core Systems". HPCS 2013, Helsinki (Finland), July 2013, doi: 10.1109/HPCSim.2013.6641467.

[10] George Matheou, Paraskevas Evripidou: "Verilog-based simulation of hardware support for Data-flow concurrency on Multicore systems". SAMOS XIII 2013, Samos (Greece), July 2013, doi: 10.1109/SAMOS.2013.6621136.

[11] Javier Bueno, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta: "Implementing OmpSs support for regions of data in architectures with multiple address spaces". ISC '13: Proceedings of the 27th international ACM conference on International conference on supercomputing, June 2013, doi: 10.1145/2464996.2465017.

[12] Fahimeh Yazdanpanah, Carlos Alvarz-Martinez, Daniel Jimenez-Gonzalez, Yoav Etsion: "Hybrid Dataflow/von-Neumann Architectures". Parallel and Distributed Systems, IEEE Transactions on (Volume:PP , Issue: 99), April 2013, doi: 10.1109/TPDS.2013.125.

[13] A. Garbade, S. Weis, S. Schlingmann, B. Fechner, T. Ungerer, "Impact of Message-Based Fault Detectors on a Network on Chip," in 21st International Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP), Belfast, February 2013, doi: 10.1109/PDP.2013.76.

[14] Daniel Goodman, Behram Khan, Salman Khan, Mikel Luján, Ian Watson: "Software transactional memories for Scala. J. Parallel Distrib. Comput." (JPDC) 73(2):150-163, February 2013, doi: 10.1016/j.jpdc.2012.09.015.

[15] Nhat Minh Lê, Antoniu Pop, Albert Cohen, Francesco Zappa Nardelli: "Correct and efficient work-stealing for weak memory models". In Symp. on Principles and Practice of Parallel Programming (PPoPP), Shenzhen, China, February 2013, doi: 10.1145/2517327.2442524 and doi: 10.1145/2442516.2442524

[16] Boubacar Diouf, Can Hantaş, Albert Cohen, Özcan Özturk, Jens Palsberg. "A decoupled local memory allocator". ACM Transactions on Architecture and Code Optimization (TACO), selected for presentation at the HiPEAC 2013 Conf., January 2013, doi:10.1145/2400682.2400693

[17] Antoniu Pop and Albert Cohen. "OpenStream: Expressiveness and Data-Flow compilation of OpenMP streaming programs". ACM Transactions on Architecture and Code Optimization (TACO), selected for presentation at the HiPEAC 2013 Conf., January 2013, doi: 10.1145/2400682.2400712

[18] R. Giorgi, R. M. Badia, F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R. Gao, A. Garbade, R. Gayatri, S. Girbal, D. Goodman, B. Khan, S. Koliaï, J. Landwehr, N. Minh L, F. Li, M. Lujàn, A. Mendelson, L. Morin, N. Navarro, T. Patejko, A. Pop, P. Trancoso, T. Ungerer, I. Watson, S. Weis, S. Zuckerman, M. Valero "TERAFLUX: Harnessing dataflow in next generation teradevices", Journal of Microprocessors and Microsystems: Embedded Hardware Design (MICPRO), April 2014, doi: doi.org/10.1016/j.micpro.2014.04.001

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 96 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Appendix A – Lua lexical conventions

Names (also called *identifiers*) in Lua can be any string of letters, digits, and underscores, not beginning with a digit. This coincides with the definition of names in most languages. (The definition of letter depends on the current locale: any character considered alphabetic by the current locale can be used in an identifier.) Identifiers are used to name variables and table fields. The following *keywords* are reserved and cannot be used as names:

and	**break**	**do**	**else**	**elseif**
end	**false**	**for**	**function**	**if**
in	**local**	**nil**	**not**	**or**
repeat	**return**	**then**	**true**	**until**
while				

Lua is a case-sensitive language: and is a reserved word, but And and AND are two different, valid names. As a convention, names starting with an underscore followed by uppercase letters (such as VERSION) are reserved for internal global variables used by Lua. The following strings denote other tokens:

+	-	*	/	%
^	#	==	~=	<=
>=	<	>	=	(
)	{	}	[	]
;	:	,	.	..
...				

Literal strings can be delimited by matching single or double quotes, and can contain the following C-like escape sequences: '\a' (bell), '\b' (backspace), '\f' (form feed), '\n' (newline), '\r' (carriage return), '\t' (horizontal tab), '\v' (vertical tab), '\\' (backslash), '\"' (quotation mark [double quote]), and '\'' (apostrophe [single quote]). Moreover, a backslash followed by a real newline results in a newline in the string. A character in a string can also be specified by its numerical value using the escape sequence *ddd*, where *ddd* is a sequence of up to three decimal digits. (Note that if a numerical escape is to be followed by a digit, it must be expressed using exactly three digits.) Strings in Lua can contain any 8-bit value, including embedded zeros, which can be specified as '\0'.

Literal strings can also be defined using a long format enclosed by *long brackets*. We define an *opening long bracket of level n* as an opening square bracket followed by *n* equal signs followed by another opening square bracket. So, an opening long bracket of level 0 is written as [[, an opening long bracket of level 1 is written as[=[, and so on. A *closing long bracket* is defined similarly; for instance, a closing long bracket of level 4 is written as]====]. A long string starts with an opening long bracket of any level and ends at the first closing long bracket of the same level. Literals in this bracketed form can run for several lines, do not interpret any escape sequences, and ignore long brackets of any other level. They can contain anything except a closing bracket of the proper level.

For convenience, when the opening long bracket is immediately followed by a newline, the newline is not included in the string. As an example, in a system using ASCII (in which 'a' is coded as 97, newline is coded as 10, and '1' is coded as 49), the five literal strings below denote the same string:

```
a = 'alo\n123"'
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 97 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

```
a = "alo\n123\""
a = '\97lo\10\04923"'
a = [[alo
123"]]
a = [==[
alo
123"]==]
```

A *numerical constant* can be written with an optional decimal part and an optional decimal exponent. Lua also accepts integer hexadecimal constants, by prefixing them with 0x. Examples of valid numerical constants are:

```
3   3.0   3.1416   314.16e-2   0.31416E1   0xff   0x56
```

A *comment* starts with a double hyphen (--) anywhere outside a string. If the text immediately after --
 is not an opening long bracket, the comment is a *short comment*, which runs until the end of the line. Otherwise, it is a *long comment*, which runs until the corresponding closing long bracket. Long comments are frequently used to disable code temporarily.

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 98 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Appendix B – Lua language features

Lua is commonly described as a "multi-paradigm" language, providing a small set of general features that can be extended to fit different problem types, rather than providing a more complex and rigid specification to match a single paradigm. Lua, for instance, does not contain explicit support for inheritance, but allows it to be implemented with metatables. Similarly, Lua allows programmers to implement namespaces, classes, and other related features using its single table implementation; first-class functions allow the employment of many techniques from functional programming; and full lexical scoping allows fine-grained information hiding to enforce the principle of least privilege. In general, Lua strives to provide flexible meta-features that can be extended as needed, rather than supply a feature-set specific to one programming paradigm. As a result, the base language is light – the full reference interpreter is only about 180 kB compiled – and easily adaptable to a broad range of applications. Lua is a dynamically typed language intended for use as an extension or scripting language, and is compact enough to fit on a variety of host platforms. It supports only a small number of atomic data structures such as boolean values, numbers (double-precision floating point by default), and strings. Typical data structures such as arrays, sets, lists, and records can be represented using Lua's single native data structure, the table, which is essentially a heterogeneous associative array. Lua implements a small set of advanced features such as first-class functions, garbage collection, closures, proper tail calls, coercion (automatic conversion between string and number values at run time), coroutines (cooperative multitasking) and dynamic module loading. By including only a minimum set of data types, Lua attempts to strike a balance between power and size.

Loops

Lua has four types of loops: the while loop, the repeat loop (similar to a do while loop), the for loop, and the generic for loop.

```
--condition = true
while condition do
  --statements
end


repeat
  --statements
until condition


--delta may be negative, allowing the for loop to count down or up
for i = first,last,delta do
  --statements
  --example: print(i)
end
```

The generic *for* loop, would iterate over the table *_G* using the standard iterator function pairs, until it returns *nil*:

```
for key, value in pairs(_G) do
  print(key, value)
end
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 99 of 100

Project: **TERAFLUX** - Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Functions

Lua's treatment of functions as first-class values is shown in the following example, where the print function's behavior is modified:

```lua
do
  local oldprint = print
  -- Store current print function as oldprint
  function print(s)
    --[[ Redefine print function, the usual print function can still be used
         through oldprint. The new one has only one argument.]]
    oldprint(s == "foo" and "bar" or s)
  end
end
```

Any future calls to print will now be routed through the new function, and because of Lua's lexical scoping, the old print function will only be accessible by the new, modified print.

Tables

Tables are the most important data structure (and, by design, the only built-in composite data type) in Lua, and are the foundation of all user-created types. They are conceptually similar to associative arrays in PHP, dictionaries in Python and Hashes in Ruby or Perl.

A table is a collection of key and data pairs, where the data is referenced by key; in other words, it's a hashed heterogeneous associative array. A key (index) can be any value but nil and NaN. A numeric key of 1 is considered distinct from a string key of "1". Tables are created using the {} constructor syntax:

```lua
a_table = {} -- Creates a new, empty table
```

Tables are always passed by reference.

Record

A table is often used as structure (or record) by using strings as keys. Because such use is very common, Lua features a special syntax for accessing such fields. Example:

```lua
point = { x = 10, y = 20 }   -- Create new table
print(point["x"])            -- Prints 10
print(point.x)               -- Has exactly the same meaning as line above
```

Array

By using a numerical key, the table resembles an array data type. Lua arrays are 1-based: the first index is 1 rather than 0 as it is for many other programming languages (though an explicit index of 0 is allowed). A simple array of strings:

```lua
array = { "a", "b", "c", "d" } -- Indices are assigned automatically.
print(array[2])                -- Prints "b". Automatic indexing in starts at 1.
print(#array)                  -- Prints 4.
                               -- # is length operator for tables and strings.
array[0] = "z"                 -- Zero is a legal index.
print(#array)                  -- Still prints 4, as Lua arrays are 1-based.
```

Deliverable number: **D7.5 – D8.3**
Deliverable name: **Final Report and Documentation + Final Results from the combination of UD and TERAFLUX dataflow techniques**
File name: TERAFLUX-D75-v17.doc   Page 100 of 100