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

PROJECT NUMBER: 249013

TERAFLUX

Exploiting dataflow parallelism in Tera-device Computing

D7.2– Definition of ISA extensions, custom devices and External COTSon API extensions (M1-M12)

Due date of deliverable: 31 December 2010
Actual Submission: 31 December 2010

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

Lead contractor for the deliverable: University of Siena (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)

Change Control

Version#	Author	Organization	Change History
0.1	Antoni Portero	UNISI	Initial template
0.2	Albert Cohen, Yoav Etsion, Lluis Vilanova, Paolo Faraboschi	INRIA, BSC, HP	Add information provided by INRIA, BSC and HP.
0.3	Yoav Etsion	BSC	Added sections 3.1.5 and 3.1.6
0.4	Arne Garbade	UAU	Added section 4.1.5
0.5	Yoav Etsion	BSC	Mod. sections 3.1.5 and 3.1.6
0.6	Salman Khan	UNIMAN	Section 4.1.7
0.7	Antoni Portero, Carol Concatto	UNISI	Section 3.1.2 (NoC)
0.8	Roberto Giorgi, Paolo Faraboschi	UNISI, HP	Intro. Section 2.2
0.9	Antoni Portero	UNISI	Section 2.1, 2.3
1.0	Rania Mameesh	UNISI	Section 3.1.2
1.1	Pedro Trancoso	UCY	Section 3.1.10
1.2	Avi Mendelson	MSFT	Section 3.1.12
1.3	Roberto Giorgi	UNISI	Section 2.1
1.4	Roberto Giorgi, Zhibin Yu, Rania Mameesh, Antoni Portero	UNISI	Section 3.1.5-4.0
1.5	Roberto Giorgi,	UNISI	Rewritten Section 2.1-2.5
1.6	Roberto Giorgi	UNISI	Rewritten Section 3.1-3.3, overall review,
1.7	Zhibin Yu, Rania Mameesh, Antoni Portero	UNISI	Proofreading
1.8	Roberto Giorgi	UNISI	Added Executive Summary, Introduction, Conclusions
1.9	ALL PARTNERS	ALL	Comments and proofreading
2.0	Roberto Giorgi	UNISI	Final check

Release Approval

Name	Role	Date
Antoni, Portero	Originator	23.11.2010
Roberto, Giorgi	WP Leader	31.12.2010
Roberto, Giorgi	Project Coordinator for formal deliverable	31.12.2010

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

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

Roberto Giorgi, Antonio Portero, Caroline Concatto, Rania Mameesh, Zhibin Yu
University of Siena (WP leader)

Yoav Etsion, Lluis Villanova, Nacho Navarro, Rosa Badia, Mateo Valero
Barcelona Supercomputing Center

Paolo Faraboschi
Hewlett Packard Espanola

Albert Cohen
INRIA

Doron Shamia, Avi Mendelson
Microsoft Research and Development

Arne Garbade, Sebastian Weiss, Theo Ungerer
Universitaet Augsburg

Pedro Trancoso, Skevos Evripidou
University of Cyprus

Behram Khan, Salman Khan, Mikel Lujan, Chris Kirkham, Ian Watson
The University of Manchester

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
CLUSTER	group of cores (synonymous of NODE)
COTSon	Software framework provided under the MIT license by HP-Labs
DDM	Data-Driven Multithreading
DTA	Decoupled Threaded Architecture
DF-Thread	a Data-Flow Thread
DF-Frame	the Frame memory associated to a Data-Flow thread
Emulator	Tool capable of reproducing the Functional Behavior; synonymous in this context of Instruction Set Emulator (ISS)
FDU	Fault Detection Unit
L-Thread	Legacy Thread: a thread consisting of legacy code
MMS	Memory Model Support
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 TSU 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
TBM	TERAFLUX Baseline Machine
TLPS	Thread-Level-Parallelism Support
TLS	Thread Local Storage
TM	Transactional Memory
TMS	Transactional Memory Support
TSU	Thread Scheduling Unit
Virtualizer	Synonymous of "Emulator"
VCPU	Virtual CPU or Virtual Core

Executive Summary

This document provides a report of the research activities performed during the first year (reporting period) towards the objectives of the TERAFLUX workpackage WP7. In order to measure the success of this Workpackage, we set two measurable objectives (cf. Annex-1, Description of Work):

1) *Producing an open-source simulator capable of simulating the future tera-device systems;*

2) *Integrate the main results from the all workpackages in this platform*

We believe that those objectives are fully achieved in the reporting period, despite the "start-up transient", as we try to illustrate in this document. This first year of activities forms the basis for much more productive interactions for the rest of the project.

WP7 is also an "integration" workpackage for the project, and therefore requires a constant attention and support: Task 7.1 started at month 1. After 6 months, we reached the first Milestone M7.1 and Deliverable D7.1. Then, we started other two tasks: Task7.2 and Task 7.3. All these three tasks will continue in the next year: hence, this deliverable reports many initiated and in-progress activities.

Most noticeably:

- ALL the **partners are able to use COTSon** (our common simulation framework) through a set of shared benchmarks (Project Milestone M7.1) and can commit in a single repository.
- COTSon has been **released as open-source** simulator, thus providing the reciprocal benefit for the TERAFLUX project and the international research community.
- We are able to boot a simulated system including Applications, OS (full-system simulation) for the TERAFLUX Baseline Machine (TBM) as designated in D7.1 consisting of more than 1000 cores. This put us in the world-wide frontline for experimenting with simulated system of such size and demonstrates a **continued relevance** of our objectives.
- We are now able to modify, e.g., the architecture of such 1000-core system. In general we can now **integrate** the proposed research from the Partners in a single platform, thus overcoming the limitation of pursuing separate research goals and rather achieving a common goal of improving such a large scale system as a whole.
- We aim at an even increased **impact** of our project using as vehicle the availability of the TERAFLUX system through the improved COTSon simulation framework. As of our knowledge, no other simulator provides similar benefits (Section 2).
- We took some **extra effort** (exceeding what planned) to provide a more convincing platform for modeling a Future Single-Chip/Package TERAFLUX through the modeling of an off-the-shelf Network-on-Chip, as we believe will benefit the whole project and the success of the simulation framework.

1. Introduction

The activities in this Workpackage are centered towards the use and development of the common simulation infrastructure based on COTSon simulator. We recall the terminology and briefly introduce COTSon (Sections 2) then we describe our exploration of different COTSon setups (Section 2.2).

As the TERAFLUX Consortium decided to target a large-scale simulation of 1000-cores, this posed us many practical challenges towards efficiently supporting this kind of simulation (Section 2.3).

We describe some activities related to the knowledge transfer (Section 2.4) and provide a practical reference also for the TERAFLUX partners, or to other external readers of this Public document in Section 2.

In Section 3, we provide a more detailed report on the status of the experiment and the new possibilities explored in the TERAFLUX project by means of the COTSon platform.

1.1 Relation to other deliverables

As this is an integration workpackage, we constantly refer to activities carried out in other workpackages. In particular:

- WP2 Algorithms examples
- WP3 Synchronization mechanism and Memory Model, Transactional Memory support
- WP4 Binary Format specification, Compilation Tools
- WP5 FDU, Fault injection.
- WP6 TSU support

1.2 Activities referred by this deliverable and document structure

The original structure of tasks was:

Task 7.1 - A definition phase: where all partners together will contribute to define the requirements and specifications for the desired ISA extension interfaces. For each new device, we currently envision a simplified interface where developers can quickly assert the benefits of a new idea, as well as a more detailed interface for the ideas that reach a more mature stage and want to be tested within the context of complete OS support

Task 7.2 - A simulator-core implementation phase: where UNISI with the guidance HP will implement the necessary changes in COTSon to support the defined ISA extensions and expose them in a new version of the COTSon SDK.

Task 7.3 - An application-specific implementation phase: where the interested partners will implement (test and validate) their specific models on the new SDK version and document the work to make it available to the rest of the consortium

This deliverable reports the status of the activities from the above three tasks organized as follows:

- **Overview of the TERAFLUX Simulation Platform**
- **Status of experiments and integration based on the COTSon simulation platform**

Besides this organization of the deliverable we provide here a "quick-reference" to locate specific topics that were part of the WP7 objectives.

WP7 Objective from Annex-1	**Where it is explained**
The capabilities to extend the ISA so that functions like thread-level speculation, transactional memory and atomic sections can be faithfully simulated (for WP3, WP4 and WP2).	Section 3.2.1, 3.2.3,
The interface for new custom devices like coherent or attached hardware accelerators (WP6 and WP3).	Section 3.4.1, 3.4.2, 3.5
The interface for fault injection and fault tracking to support the Reliability and Fault-Tolerance activities (WP5)	Section 3.6
The interface to add power models (for WP3 and WP6).	Section 3.2.2
Disseminate the COTSon platform and the knowledge on the tool.	Section 2.4
Add QEMU as another optional functional engine (compatible with open-source development and multiple ISAs	Sections 3.2.3, 3.3.2, 3.3.3
Define the extensions, devices and interfaces needed by the WP3 (programming model), WP4 (compilation tool), WP5 (reliability and fault tolerance) and WP6 (architecture).	Section 3
Design and implement the appropriate extensions, devices and interfaces in the simulator to satisfy the requirements: as ISA-extensions or architecture modifications, as ASM extensions or as fully functional devices and interfaces (fault injection and power) in the COTSon framework.	Section 3

2. Overview of the TERAFLUX Simulation Platform

In the TERAFLUX project, we decided to rely on the COTSon simulator from our Partner HP Labs. In this Section we aim to highlight its continued relevance towards the Research goals of the project. We recall and briefly explain the terminology that we use and we contrast the benefits of COTSon in comparison to other simulators (Section 2). We show how we carried out reference experiments to extract both reference numbers and to setup an appropriate simulator host for COTSon (Section 2.2). We describe different COTSon-based optional setup, and highlight advantages and disadvantages that motivate our efforts for an efficient large-scale (1000 cores target) simulation (Section 2.3).

2.1 Terminology recall and brief overview of COTSon advantages over other many-core simulator (Partner UNISI)

In this Section, we recall and briefly explain the terminology that we use and we contrast the benefits of COTSon [Argollo09] in comparison to other simulators.

Future architectures will expose a massive number of parallel simple processors (besides some bigger ILP focused processor) and on-chip memories connected through a network-on-chip, whose speed is more than hundred times faster than off-chip [Asanovic09]. The number of cores in a single die is increasing and postulated to reach1000 and beyond. How to simulate a many-core processor in such a scale is an interesting challenge, which we are fully taking in the TERAFLUX project.

In the computer architects' and researcher's tool-box, a cycle-accurate software simulator is one of the most important tools. Simulators allow us to carry out platform exploration of the different architectural options under several types of requirements (e.g., performance, reliability, power consumption, temperature). This also means that the simulator has to be able to estimate or measure the achievement of the needed requirement. However, a cycle-accurate simulator is extremely slow. Chiou reports that the simulators for single-core processors used at Intel and AMD often operates at 1KIPS (kilo instructions per second) to 10KIPS, leading one year to ten years to simulate a 3GHz target for 2 minutes [Chiou09]. Even worse, simulators are getting slower and more complicated under multi-core or many-core scenarios.

In the TERAFLUX project, the simulator (based on COTSon) is also a major output of the project: our aim is to constantly improve this tool (and publicly release updates) in order to impact the community and increase the possibility of addressing problems at the Tera-device scale. Moreover the simulator serves as an integration platform as it involves a close participation from all partners.

It is worthwhile to recall some terminologies about architecture and micro-architecture simulation. In this section, we will review the concept, classification, and detail challenges of simulators.

When we talk about simulation, another concept, emulation, is often mentioned. Sometimes, *emulation* and *simulation* are used interchangeably in the computing system literature. To make it easy for discussion, we distinguish these two concepts. Emulation means that the function of a platform is repeated on another platform. The main concern of emulation is the correctness of the function. Simulation is an extension of emulation. Besides ensuring the functional correctness, simulation must provide accurate time information which is related to performance. In this document, if there is no explicit specification, an emulator is related to the bare functional behavior, while a

simulator is related to performance. Other two widely used concepts are *functional simulator* and *performance simulator*.

To clean up a bit the terminology, in this context, we use just "simulator" to mean "timing simulator (cycle-accurate)", we use just "emulator" to mean "functional simulator" (in case of an emulated processor we also use Instruction Set Simulator or ISS [Knuth97]. Also we use "power simulator" when we have a power model or "power timing simulator" when we use both a power model and a timing model [Li09].

Since simulation is so important, there are many different kinds of simulators or simulation methodologies have been developed over the years. Again, there are different classification methods for simulators and some of them are possibly overlapped. According to whether the modeled processor is a single core or a multi-core, we have *single-core simulators* and *multi-core simulators*. SlackSim [Chen09] and SimpleScalar [Austin02] are examples of single-core simulator, while COTSon is a typical multi-core simulator [Argollo09].

Based on whether the OS behavior is included or not, there are *user-level simulators* and *full-system simulators*. SimpleScalar is also an example of user-level simulator. SimOS [Rosenblum95], Simics [Magnusson02] + GEMES [Martin05], SimNow + COTSon [Argollo09] are examples of full-system simulators.

According to whether the simulator can execute in parallel or not, we have *parallel simulators* and *sequential simulators*. Sequential simulators are quite accurate [Binkert06], [Renau05] but as the complexity of the simulated platform increases the simulation time gets unreasonably long. Parallel simulation [Argollo09], [Reinhardt93], [Dickens93], [Prakash98], [Das94] requires multiple processing cores to increase the simulation rate.

A (timing) simulator can also use different approaches depending on the relationship between the "functional model" (fm) and the "timing model" (tm) [Mauer02]: i) "*functional-first*" or "*trace-driven*", the fm is run first and separately and the tm is run later on in a completely decoupled fashion (the all fm is run before the tm is run); ii) "*timing directed*" or "execution driven", the fm and tm are closely coupled (no decoupling); iii) "*timing-first*", the tm drives the fm, both are completely decoupled, but the functional execution has to be checked later on and eventually undone; iv) "*functional-directed*", the fm drives the tm, both are completely decoupled, the functional model is always the right one but we need a timing feedback from tm to correct the timing behavior so that it becomes visible to the applications being simulated. [Argollo09].

Nowadays, power consumption, vulnerability, and thermal dissipation of processors are becoming more and more important. Therefore, besides performance simulators, there are simulators for the above issues as well. Some simulators combine several above aspects. For example, McPAT is a power simulator for multi-core processors developed at HP Labs. It can also be used as a power and performance model [Li09]. In the TERAFLUX project we use the McPAT tool to estimate power consumption (discussed in a subsequent section in this document).

Since the cycle-accurate simulation takes extremely long time, there is a large body of literature about how to accelerate the cycle-accurate simulation. The most popular simulation acceleration technique is *sampling*. This technique selects some instructions for cycle-accurate simulation while

other instructions are simulated in functional mode called fast-forwarding. According to the different sampling strategies, a lot of simulation sampling techniques are developed. T. M. Conte et al. employed simple random sampling strategy to speed up simulation [Conte 96]. R. E. Wunderlich et al. used systematic sampling in acceleration of micro-architecture simulation [Wunderlich03] [Wenisch06]. They also tried to use stratified sampling scheme for acceleration [Wunderlich04]. E. Perelman et al. applied representative sampling to speedup micro-architecture simulation [Perelman 03]. Zhibin Yu employed a more flexible sampling scheme, two-stage sampling, to accelerate micro-architecture simulations [Zhibin09].

Although sampling can accelerate simulation significantly with relatively high accuracy, it is complicated to prepare the sampling parameters before the real simulation. Therefore, simplifying the acceleration of micro-architecture is becoming important. A few of researchers have tried to simplify this procedure. UNISIM is one example because it provides transparent techniques for speed up [UNISIM10]. Another example is the CantorSim approach which employs fractals to simplify the parameter preparation for simulation acceleration [Zhibin10].

The simulation rate of software simulation by sampling techniques is limited by the speed of the functional simulator. Although the speed of the functional simulator is much faster compared to a performance simulator, it is still too slow for computer architects. Researchers have tried to accelerate performance simulator by using FPGAs. Penry et al. [Penry06] provide a much more detailed, low-level simulation and are targeting hardware designs with FPGA support. Their simulator, while fast for a cycle-accurate hardware model, does not provide the performance necessary for rapid exploration of different ideas or software development. Other examples are ProtoFlex [Chung09], FAST [Chiou07], and HASim [Dave06]. In these simulators, they use FPGA to accelerate the timing models. However, FPGA based simulation is less flexible than software based simulation. For instance, implementing a new model in an FPGA such as RAMP project is more difficult than software (typically requiring an RTL-level description), making it harder to quickly experiment with different designs [Gibeling06],[Krasnov07].

For functional simulator or emulators, there are also several existing systems. One of the most popular ones is QEMU [Bellard05]. QEMU (Quick Emulator) is Open Source software for creating emulation and virtual machine environments, developed by Fabrice Bellard. As an emulator, it is used to run operating systems and applications written for another hardware platform; for example, running ARM software on an x86-based PC. For virtualization, QEMU is used to emulate devices and certain privileged instructions and requires the KQEMU/KVM kernel module and the host operating system to provide a virtual machine environment. An extension of QEMU for emulating multi-/many-cores is COREMU [Wang11]. SimNow and SIMICS are other examples of functional simulators.

COTSon is a full-system simulator that uses the “functional-directed” approach; it can take advantage of sampling techniques; it uses AMD SimNow as emulator engine to functionally process the workload; it can run several emulator instances in parallel; in TERAFLUX, we are also developing support for using QEMU as an alternative open-source emulator engine.

In the rest of this section, we describe some multi-core simulators and contrast them with COTSon.

There are tons of multi-core simulators developed during the past 10 years. The typical examples are COTSon[Monchiero09], Trace-Factory [Giorgi97], SimFlex [Wenisch06], MPTLsim[Zeng09],

GEMS [Martin05], Graphite[Miller10], BigSim [Zheng04], FastMP [Kanaujia06], SlackSim [Chen09], Wisconsin Wind Tunnel(WWT) [Reinhardt93] [Mukherjee00], the simulator developed by Chidester and George [Chidester02], MPTLsim[Zeng09], and LIBERTY [Penry06].

SimFlex and GEMS both use an off-the-shelf sequential emulator (Intel's Simics) for functional modeling plus their own models for memory systems and core interactions. GEMS uses the timing-first simulation approach: their timing model drives Simics one instruction at a time which results in much lower performance than COTSon. SimFlex uses statistical sampling of the application to speed up the simulation but therefore does not observe its entire behavior.

MPTLsim is a cycle-accurate, full-system x86 and x86-64 multicore simulator. MPTLsim uses the hardware abstraction provided by the Xen hypervisor [Xen99] to fast-forward execution and reach a given point where simulation can start. MPTLsim provides a significant faster simulation rate compared to versions of GEMS. MPTLsim make use of a cycle accurate out-of-order core design implementing the x86-64 ISA. However, the fast-forwarding using Xen is completely opaque to the simulator and during the Xen execution nothing can be inferred about memory, instructions or I/O.

However, among the above multi-core simulators, as of our knowledge, only COTSon and Graphite currently claim they target for 1000 cores. Graphite is a simulator developed by MIT and still being developed, based on the PIN binary instrumentation package. We evaluated its initial version and found it lacking several usability features to be used in the TERAFLUX context at current stage. On the contrary, COTSon is relatively mature and allows simulating complete systems ranging from multicore nodes up to full clusters of multicores with complete network simulation. It is composed of a pluggable architecture, in which most features can be substituted for proprietary development, thus allowing researchers to use it as their simulation platform. For example, COTSon has been used for modeling 1000-core shared-memory multiprocessor [Monchiero09], or for the CORONA studies [Vantrease08].

COTSon [Argollo09], [Monchiero09] is a node-level parallel simulator (i.e., several SimNow instances can run in parallel, while each SimNow execution is sequential). COTSon uses AMD's SimNow for functional modeling. The sequential instruction stream coming out of each SimNow functional core is interleaved to account for correct time ordering before timing simulation. Currently, COTSon can perform multi-(guest-) machine simulations but only if the applications are written assuming a distributed memory machine (for example using a messaging library like MPI). Previous COTSon experiments were used with a slightly different methodology – a hybrid between trace-driven and feedback-driven simulation. It did not address specifically an "evolving machine" like the TERAFLUX one and the full-system functional-directed approach was limited to specific scenarios (e.g., datacenters).

Another possibility that we considered is to take advantage of the large parallelism of GPGPUs to emulate a guest core on the host GPU core [Raghav10]. Compared to COTSon, this is not yet fully developed (only part of the x86 ISA is emulated) and it is not a full-system simulator (it is only an emulator).

TERAFLUX uses and develops the COTSon platform. We believe that our choice is still a valid option to target a many-core machine of the size of 1000-cores, with the full potential of including the research in the EU FET TERACOMP objectives.

2.2 COTSon framework organization and possible setups

In this Section, we provide some further information about the COTSon framework. HP Labs' COTSon [Argollo09] simulator is a full system simulation infrastructure, based on AMD's SimNow. It allows simulating complete systems ranging from multi-core nodes up to full clusters of multicores with complete network simulation. It is composed of pluggable architecture, in which most features can be substitute for your own development, thus allowing researchers to use it as their simulation platform.

SimNow [SIMNow09] models the functional behavior of 1 node containing multiple cores (tested up to 32 cores as of version 4.6.2 used in 2010). It is based on dynamic binary translation principles along the lines of what is used in some is a virtual machine hypervisors (like e.g. Oracle's VirtualBox, VMware, MS Virtual-PC ([VirtualBox10], [VMWare10], MVMPC10])) with additional capabilities for timing simulation, but also many other possibilities for external extensions that we used in some experiments (see e.g. BSC's experiments on sharable memory support).

More specifically, SimNow is a configurable x86-64 dynamically-translating instruction-level platform simulator. However, note that in the COTSon, SimNow is essentially used as a full-system emulator.

In order to flexibly model a variety of architectural features, in the "Timing Models" (see Figure 1) we can provide the necessary timing behavior. For example, if we want to test different L1 cache sizes, we can provide a timing model for L1 caches and change cache size in such models. This is a good example to show that in this case, we disregard the internal timing information of the SimNow: therefore in such case, the SimNow then acts as an emulator or ISS (cf. Section 2). A very similar situation is for other architectural components where we introduced our custom timing models (e.g. the NoC – see Section 3.4.1).

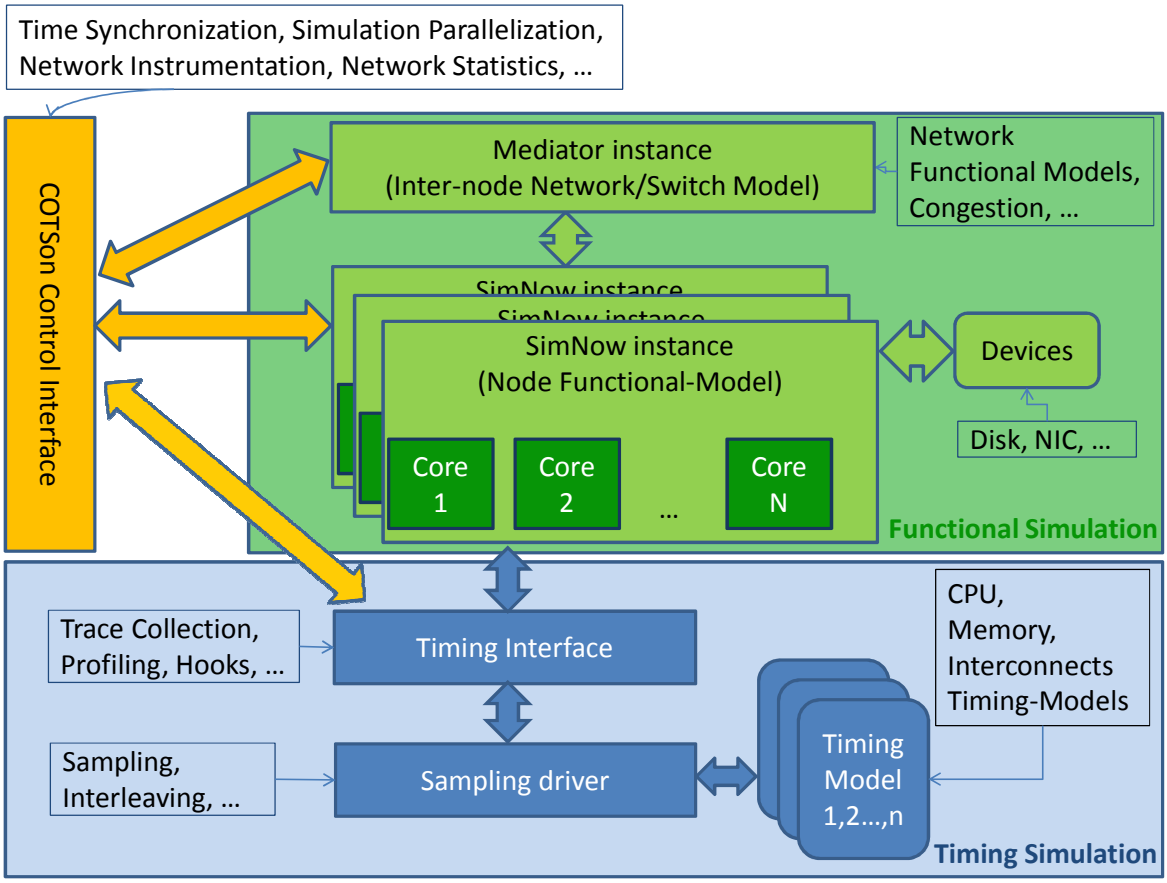


Figure 1 - COTSon Overview.

The interconnection network among the nodes (see Figure 1) is provided by the "Mediator". HP provided a reference implementation of the network Mediator (also known as "Q Mediator" [Lugones09]) that essentially models an Ethernet Switch.

The Mediator provides (simulated) Ethernet functional connectivity among simulators and works with simulations distributed across multiple hosts. It manages the timing models (not shown in Figure 1, for the sake of clarity) for a networked group of nodes and is responsible for networking modeling (topologies, switches, cards, etc.), queuing up pending network packets and computing the delays due to network congestion.

UNISI and the other participating Partners extensively experimented with the COTSon platform. Many of the Partners were new to this framework. An extensive documentation and shared knowledge on the experiment and possible setups were documented on the internal wiki website (wiki.teraflux.eu). The necessary steps involved:

- Getting background on SimNow and COTSon. Getting background on QEMU: this has been identified, in Annex-1, as an open-source engine for increased flexibility alternative to SimNow

- Understanding the COTSon control interface, timing models, synchronization and multiple parallel simulators coordination required basic knowledge of additional languages such as Ruby [Cooper09], [Flanagan08] and Lua [Ierusalimschy06], [Ierusalimschy06b].

- Analyzing the software application set that performs networking functionality, such as Mediator.

- Providing BIOS images for 8, 16, 32 cores in each nodes. This step was carried out by HP labs and became very useful to test more complex scenarios.

- Modifying and managing a number of "Virtual Machine descriptions": the SimNow can store a node configuration in so called "BSD files" (see glossary), including saving the state of the machine at a given point of the execution.

The TERAFLUX wiki site provides extensive support for Partners, including the possibility of keeping track of discussion, documents, references and links.

2.3 Targeting a 1000-core simulation

The main reason to consider a 1000-core system from the very beginning is to have a platform more similar to what we expect to be a 1-TERA device chip/package.

There have been proposed several approaches to study and research a computing system of the size of 1000-cores. There are also prototype chips that currently work with 48 cores (Intel SCC [Mattson10]) that could eventually scale to a 1000-core processor [Mattson10b]. In the market also there exist 1000-core CC-NUMA machines (like the SGI Altix UV). Of course, many supercomputers reach that size: one can "just build" the machine.

The ambition of TERAFLUX is however to be able of *changing* such machine in a flexible way, while tackling research challenges on programmability, architectural design and reliability. Therefore, we have the need to stress the COTSon platform, in order to being able to simulate 1000 cores.

This Section includes four subsections. In subsection 2.3.1, we report some comparison done in competitive project like RAMP [Gibeling06],[Krasnov07], comparing physical SMP, physical Cluster, FPGA, Emulator and Simulator approaches towards 1000-core platforms. In subsection 0, we present some real hardware platforms architecture that we used in TERAFLUX (MareNostrum, "TFX2") or that we considered to use (i.e., Altix UV). Then in subsection 2.3.3 we discuss several setups that use COTSon to target a 1000-core simulation, presenting advantages and disadvantages. Finally, in subsection 2.3.4, we briefly report some results to find applications that: i) do not present "algorithmic bottleneck" when scaling to 1000 or more cores; ii) reasonably load interconnects and memory (without an "exponential explosion" of the dataset). This is very important to adequately stress the 1000-core simulation.

2.3.1 Comparison among platforms to evaluate novel 1000-core architectures

Table 1 reports a comparison between different platforms that allow us to evaluate and experiment with 1000-cores.

The main drawback of SMP is due to the high cost of 1000-core machines and the fact that it's not possible to modify the architecture. Another problem is the difficulty to observe the results and the inability to reconfigure the hardware to extend the ISA. Building a cluster of computers to get 1K cores is feasible but it has similar disadvantages as SMP. A major FPGA disadvantage is the complexity to build these kinds of systems since hardware and software must be set up. Available soft-processors such as NIOS II for Altera and micro-blaze for Xilinx do not support x86-64 ISA. They are currently restricted to 32bits.

An emulator (see previous Section) has the main problem that it does not provide timing, which draws us to the use of simulators. The main problem of simulators is the credibility but as much as the simulators evolve the credibility of these solutions increase. Performance is another problem, while we currently obtain a 1/10 to 1/1000 down speed for simulating a cluster of computers that it is still feasible.

Table 1 - Comparison among different approaches for doing research related to 1000-core computing system (Information revised from data of the RAMP project). GPA (Grade point average, A=5 points, B=4 points, C=3 points, D=2 points, E=1 points, F=0 points).

	SMP	Cluster	FPGA	Emulator	Simulator
Scalability (1K cores)	C	A	A	A	A
Cost (1K cores)	F(€40M)	C	B(€0.1-0.2M)	A+ (€0.01M)	A+(€0.01M)
Power/Space (Kw, racks)	D (120 kw, 12 racks)	D (120 kw, 12 racks)	A (1.5 kw, 0.3 racks)	A+ (0.1 kw, 0.1racks)	A+ (0.1 kw, 0.1 racks)
Observability	D	C	A+	A+	A+
Reproducibility	B	D	A+	A+	A+
Reconfigurability	D	C	A+	A+	A+
Credibility	A+	A+	B+/A-	F/D	C
Development time	B	B	C	A+	A+
Performance (clock)	A (2GHz)	A(3GHz)	C (0.1 GHz)	B(≈ 0.9 of original)	C(1/10 to 1/1000 SMP)
x86-64 ISA	A+	A+	F	A+	A+
Modifiable	F	F	B	A	A
GPA	D	D	B+/A-	B	A

2.3.2 Experiments on Physical Machines

The use of real platforms is needed for two main reasons:

1. In order to get some preliminary numbers for setting up the COTSon simulation environment; those numbers serve also as reference to tune up the simulation environment;

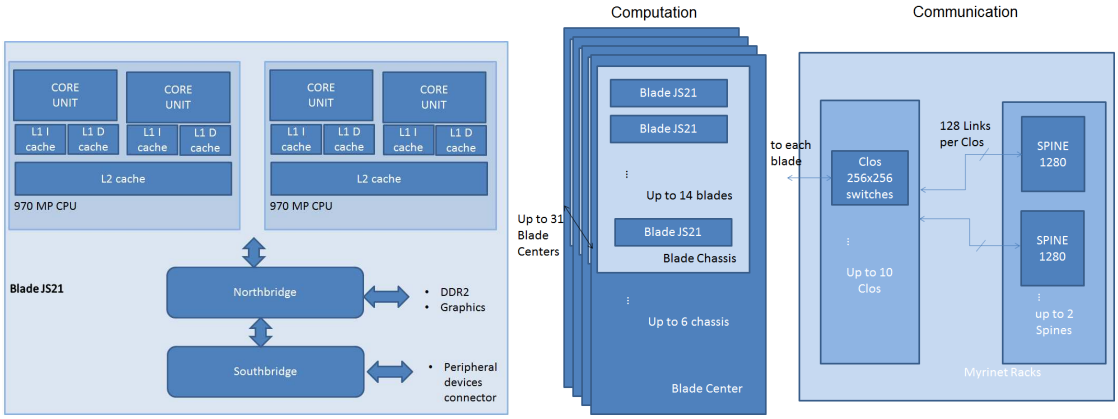
2. In order to provide and setup an adequate simulation host (to support the COTSon simulation itself).

It is not always possible to get the "perfect" setup: there are several reasons for that. Sometimes, the programming model cannot be changed, sometimes the machine cannot be changed, and sometimes we do not have all the necessary flexibility. Here we consider some of the options.

2.3.2.1 A Message-Passing Machine (BSC's MareNostrum)

We used the MPI library (Message Passing Interface) [myri10] and BSC Partner's supercomputer (MareNostrum) to run some initial tests on a 1000-core system. MareNostrum is one of the top 500 most powerful computers in the world[1]. Here we briefly describe the structure of the machine. Later we show some results from some benchmarks: a thorough analysis of benchmarks is in D2.1.

The computer at number 118/500 in the "top-500" is MareNostrum. MareNostrum is based on 2560 Blade JS21, each with 2 dual-core IBM 64 bits PowerPC 970MP, 2.3GHz with 10240 Cores, Rmax=63,83 (number identifying the maximum LINPACK performance), Rpeak=94,21 TeraFLOPS (peak performance) [bsc10], [kimble06] . In Figure 2a, we show a schema of the blade JS21 [kimble06, Bunting06]. In Figure 2b we show am overall schema of MareNostrum.



a) b)

Figure 2 - a) Blade Center JS21 schema with JS21 970MP processor, b) Overall schema of the MareNostrum (computation based on blade centers and connection is based on Myrinet switches)

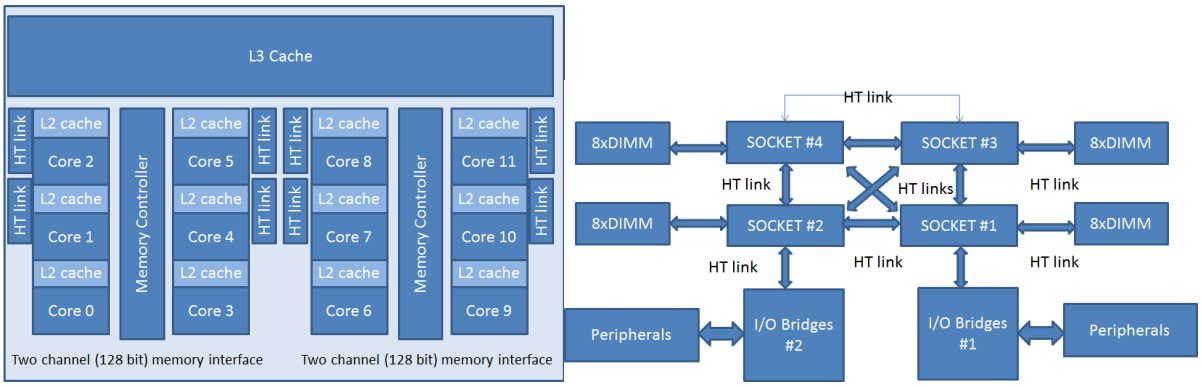
[1] Ranks and details of the top 500 non-distributed most powerful computer systems in the world are described in [top500]. The update list is published and updated twice a year. The top500 aims at providing a basis for tracking and detecting the trends in high-performance computing. It bases its ranking on a portable High-Performance LINPACK (HLP) benchmark written in FORTRAN for distributed-memory computers [top500].

2.3.2.2 Shared-Memory Machines

2.3.2.2.1 *The UNISI "TFX2" CC-NUMA on a board (48-cores and 256GB shared memory)*

The current COTSon simulation host at UNISI is a 48 cores (AMD x86-64) on a single board and 256GB of shared memory (NUMA single Board Symmetric Multi-Processor) named "TFX2". We will describe shortly the organization of this machine. Some partners (UAU, INRIA …) also used a similar machine, but with less memory and/or cores (thus limiting the simulation capabilities). In the next period (year 2), we plan to use a larger machine as soon as the hardware we need will be available within our planned budget. This machine has been provided as part of the existing facilities at UNISI.

This board has four processors (sockets) and each processor has 12 cores (Opteron 6168 1.9 GHz with 512KB L2 cache). It also has 32 quad-channel slots supporting up to 512GB of DDR3-1333 registered ECC. The BIOS is 16 Mb AMIBIOS SPI Flash ROM [Spm10].



a) b)

Figure 3 - a) AMD Opteron 6168 (12 cores), b) Board Level schema (HT=Hyper Transport).

2.3.2.2.2 *SGI Altix UV*

Another possible solution for the simulation host that has been considered (but not used currently) is using machines like SGI Altix. We investigated this option but we decided that, for the currents needs, machine like the one already presented ("TFX2" or "BabyCotson", cf. 2.4) suffices. Some computer center like the Spanish CESCA (Centre de Supercomputacio de Catalunya): http://www.cesca.es/ offers access to this kind of system. However, at this moment, we favor to extend a system like the one in 2.3.2.2.1 as this could fit our COTSon simulation needs for the rest of the project.

The Altix UV has more than 1K cores and 16 TB of memory. Altix UV scales up to 2,048 cores (256 sockets). It supports up to 16TB of global shared memory through in a single system image OS (cf. 2.3.3.4), i.e. relies on a Distributed Operating System.

2.3.3 How to simulate 1K cores: setup to simulate 1K cores

All the TERAFLUX Partners agreed that in order to address the challenges of a Tera-device system we should be able to simulate a platform encompassing:

i) A full-system (including OS and devices)

ii) Using "efficient applications" able to load the system and stress its limits

iii) Able to scale at the State-of-the-Art (about 1000 cores or 1K cores) as many work are addressing (see for example [Monchiero09], [Miller10], [Raghav10], [Gibeling06], [Krasnov07])

In order to explain, the motivations for the choices we are currently exploring, we briefly analyze here the options that we can use for a 1000-core simulation, highlighting advantages and disadvantages. We actually setup and tested them when it was necessary or useful to advance with the mainline experiments on the COTSon framework. We think it's useful also to recall them in order to make clear and motivate more strongly our choices on the COTSon framework.

In the following, it is important to note that our aim is to be able to choose flexibly the Programming Model and the Architecture. Of course, for reference reasons, the initial setup takes the existing available Programming Models and Architectures (i.e., TERAFLUX Baseline Machine), but we note (as also clearly stated in D7.1) to **evolve this machine towards the TERAFLUX models**. Therefore also the simulation environment must reflect this flexibility.

2.3.3.1 Setup #1: Physical Machines, MPI Programming Model

This setup is manly useful for reference. The i-th Physical Machine is a node including C_i cores. For simplicity we can assume a fixed number of cores per node, say C_N.

The simplest solution to scale to 1K cores is to connect N nodes with C_N cores each one so that N x C_N equals 1000. We represent this situation in Figure 4 (for simplicity we represent the case of just two nodes). The simplest off-the-shelf network can be Ethernet based.

Each computer runs a lightweight Operating System (e.g., Linux with MPI libraries).

On top of each OS runs a set of applications compatibles with MPI libraries. These applications with MPI allow load balancing of computations among the different CPUs, at the expense of software overhead.

Setup #1

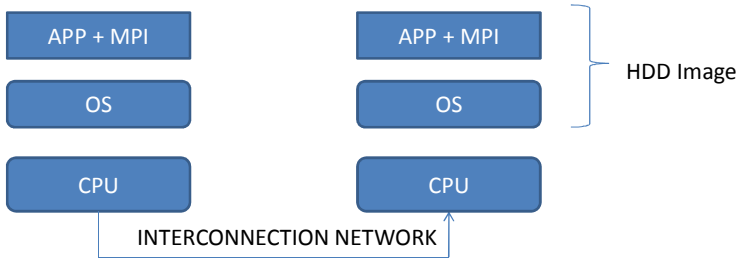


Figure 4 - Two Physical Machines running MPI applications

UNISI has some test hosts available, which are used for this task. BSC has provided access to the MareNostrum computer for more complex tests (in the latter case the interconnection network is obviously much faster, see 2.3.A.1).

In this setup, we can also extract "HDD images" (Hard-Disk Drive image) and then run those images as virtual disk of a Virtual Machine (see next Subsections).

Advantages:

- This setup can run both on a real machines (at least at small scale for tests) AND on the COTSon simulator as provided at the Month-1 of the TERAFLUX project.
- It is a rapid configuration to test parallel Applications in a first instance.
- Way to obtain some reference number for, e.g., execution times of the running applications or other characterizations of the applications (see D2.1 for a more extensive discussions on the collected data)

Disadvantages:

- **It's not possible to modify the architecture of the machine.**
- Taking into account that we aim to flexibly change the programming model and architecture (e.g. the dataflow based execution model and architecture proposed in D6.1), this setup may end up in poor performance when N (number of nodes) increases.
- Binds the application to the machine, which is exactly the opposite direction that we want to follow globally in TERAFLUX: we aim to decouple the Application (WP2) from the machine with appropriate Programming Models (WP3), Compilation Tools (WP4) and Execution Models (WP6).
- The run-time is constantly involved to appropriately schedule the ready tasks/threads on the available nodes.
- The Physical architecture that is a Distributed Machine not like the general one we aim in TERAFLUX for.

2.3.3.2 Setup #2: Virtual Machines running on Several Physical Machines, MPI Programming Model

This setup is also provided for reference. For simplicity, we assume that each Physical Machine is running exactly one Virtual Machine (each Physical Machine could run several Virtual Machines in a general case).

Here, instead of running the set of applications on top of a physical machine, we use a virtual machines (VMs) such as the one provided by SimNow or QEMU. In particular, one important

advantage is that we can use most of the off-the-shelf behavior of the VM while we can add the additional instructions (ISA extensions) that we identified in the first 6 months (see D.1).

A more extensive discussion on adding new instructions in SimNow is described in Section 3.2.1 in this document. In Section 3.2.3 we also show how we can add custom instructions in QEMU.

HP provided a sample simulation COTSon setup for the N nodes case (extension of the basic example "twonodes.in"). BSC has provided an HDD image (see Figure 5) with a first of Applications (a large subset of the Applications identified at Month-6 (Milestone M2.1)): the applications are fully configured to run in the COTSon framework. The HDD-image can be directly has a virtual disk of the COTSon simulation.

In Figure 5, we show an example of this setup. We also highlight that, instead of having a single (host) OS, in this case the Virtual Machine runs its own (guest) OS with a separate address space. The VM on a given host runs on top of the Host OS that manages this host.

In particular, in this case there are two levels of address spaces (the guest one and the host one). More recent x86 based computers have ways to map more efficiently guest to host address spaces (AMD RVI, Intel VT-x [AMD08], [INTEL10], [INTELVTX06]). We aim to take advantage of such mechanism to improve the simulation efficiency. However, at the current stage we can already use the COTSon component to run experiments by using two physical hosts (therefore clearly parallelizing the simulation itself).

To simulate 1k cores we can here use e.g., N Physical Hosts, each one with C_N cores, such that N x C_N =1000.

Setup #2

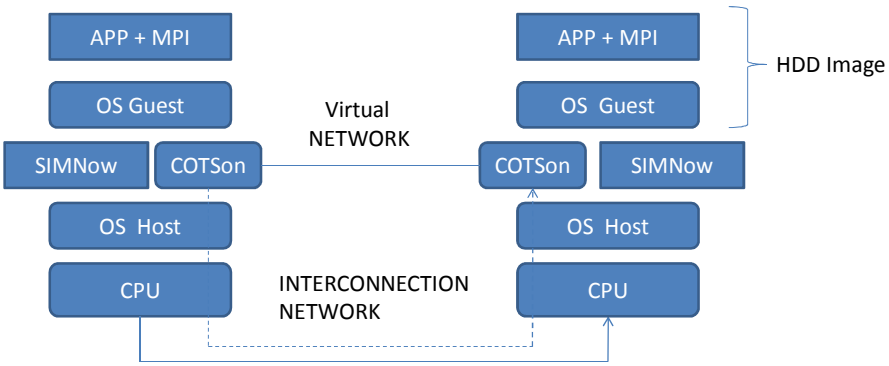


Figure 5 - Two Physical Machines each one running a Virtual Machine.

Advantages:

- This setup can run both on a real machines (at least at small scale for tests) and on the COTSon simulator as provided at the Month-1 of the TERAFLUX project.
- It allows us to modify system parameters like e.g. number of cores in each simulated instance.
- It allows for a parallelization of the simulation (the several instances are running in parallel on completely independent hosts).

Disadvantages:

- Taking into account that we aim to flexibly change the programming model and architecture (e.g. the dataflow based execution model and architecture proposed in D6.1), this setup may end up in poor performance when N (number of nodes) increases.
- Binds the Application to the Machine, which is exactly the opposite direction that we follow globally in TERAFLUX: we aim to decouple the Application (WP2) from the Machine with appropriate Programming Models (WP3), Compilation Tools (WP4) and Execution Models (WP6).
- The MPI run-time is constantly involved to appropriately schedule the ready tasks/threads on the available nodes.
- The communication and synchronization among the simulation instances adds up to the Application traffic on the Physical Interconnection Network, thus slowing down the simulation.
- The guest address space in one node can only pass information to the guest address space on another node through the Physical interconnection Network (operations for the support of the TERAFLUX Memory Model, such those described in sections 3.1.7, 3.1.8, may be considerably slowdown the simulation due to the copying of data buffers).
- We cannot take advantage of RVI/VT-x virtualization mechanisms across different Physical Machines.

2.3.3.3 Setup #3: Virtual Machines running on a SINGLE Physical computer, MPI Programming Model

This setup is the most used during the first year of experiments, and we aim to develop it more extensively as it provides several advantages, while we will progressively release the dependence from MPI.

In this setup we use just one host Machine. The Machine essentially provides Shared Memory and a number of cores (e.g., a CC-NUMA with 48 cores and 256 GB of memory as explained in 2.2.A.2 and in the experiment of next Sections). The interconnection Network is completely provided by the COTSon Mediator. This is shown in Figure 6.

To simulate 1k cores we can here use e.g., N COTSon (or better SimNow) instances, each one with C_N cores, such that N x C_N =1000.

Setup #3

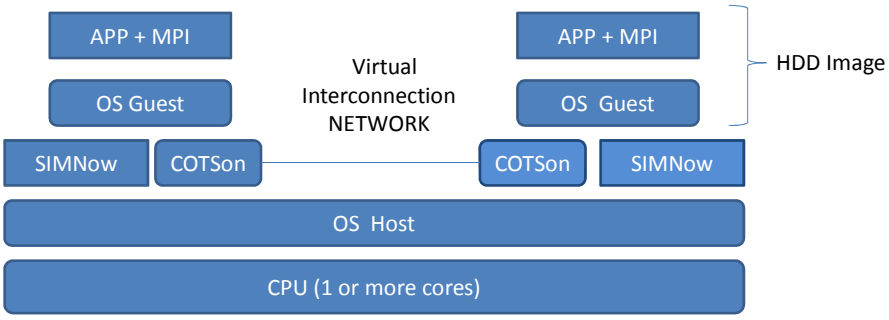


Figure 6 - One Physical Machine running two Virtual Machine instances that communicate through the Virtual Network (Mediator).

Advantages:

- This setup can run both on a real machines (at least at small scale for tests) AND on the COTSon simulator as provided at the Month-1 of the TERAFLUX project.
- It allows us to modify system parameters like e.g. number of cores in each simulated instance.
- It allows for a parallelization of the simulation (the several instances are running in parallel on the available cores – load balancing automatically provided by the Host OS scheduler).
- Possible to avoid copying buffers among instances because they reside in the Host Shared Memory Network (operations for the support of the TERAFLUX Memory Model, such those described in sections 3.1.7, 3.1.8 may take advantage of this).
- Possibility to take advantage of RVI/VT-x virtualization mechanisms across different Physical Machines (under development).
- The communication and synchronization among the simulation instances adds up to the Application traffic, but could bypass TCP/IP and avoid using the Physical Interconnection Network.
- No need to use the Physical Network.

Disadvantages:

- Taking into account that we aim to flexibly change the programming model and architecture (e.g. the dataflow based execution model and architecture proposed in D6.1), this setup may end up in poor performance when N (number of nodes) increases.
- Tightens the Application to the Machine, which is exactly the opposite direction that we follow globally in TERAFLUX: we aim to decouple the Application (WP2) from the Machine with appropriate Programming Models (WP3), Compilation Tools (WP4) and Execution Models (WP6).
- The MPI run-time is constantly involved to appropriately schedule the ready tasks/threads on the available nodes.

- The Physical architecture that is more natural to model is a Distributed Machine not like the general one we aim in TERAFLUX.

2.3.3.4 Setup #4: Virtual Machines running on a SINGLE Physical computer, Flexible Programming Model on top of a Distributed Machine Guest

In this setup we use a Single System Image OS to achieve the illusion of a shared-memory system on top of the simulated cluster as provided by COTSon. The situation is shown in Figure 7.

Setup #4

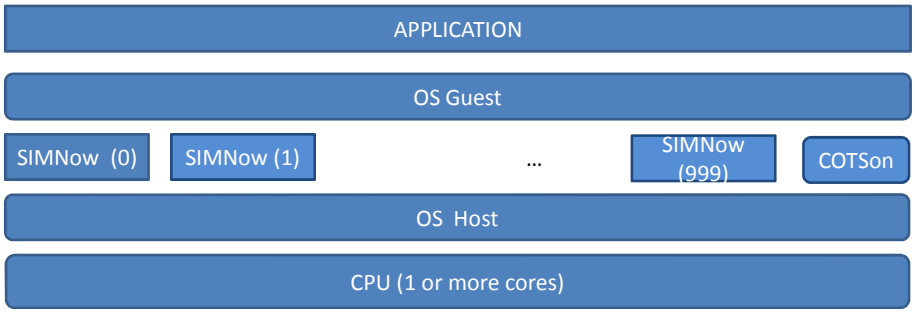


Figure 7 - VM instances governed by a Single Source Image (SSI) OS.

Advantages:

- Allows us to run Shared Memory applications like OpenMP ones (can still run MPI as if it was a single big node).
- Can run both on a real machines (at least at small scale for tests) and on the COTSon simulator as provided at the Month-1 of the TERAFLUX project.
- It allows us to modify system parameters like e.g. number of cores in each simulated instance.
- It allows for a parallelization of the simulation (the several instances are running in parallel on the available cores – load balancing automatically provided by the Host OS scheduler).
- Possible to avoid copying buffers among instances because they reside in the Host Shared Memory Network (operations for the support of the TERAFLUX Memory Model, such those described in sections 3.1.7, 3.1.8 may take advantage of this).
- Possibility to take advantage of RVI/VT-x virtualization mechanisms across different Physical Machines (under development).
- The communication and synchronization among the simulation instances adds up to the Application traffic, but could bypass TCP/IP and avoid using the Physical Interconnection Network.
- Load Balancing for the Application is managed by the Guest OS
- No need to use the Physical Network.

Disadvantages:

- This setup requires the use of a Distributed OS as Guest OS (like e.g., Kerrighed [KERRIGHED10], which offers the view of a unique SMP machine on top of a cluster) or in general a SSI (Single System Image) OS.
- Relatively poor performance when N (number of nodes) increases;
- Partially tightens the Application to the Machine, which is in the opposite direction in respect to what we follow globally in TERAFLUX: we aim to decouple the Application (WP2) from the Machine with appropriate Programming Models (WP3), Compilation Tools (WP4) and Execution Models (WP6).
- The underlying Guest Architecture is a "cluster", which is then more naturally mapped to a physical Distributed Machine not a generic one like we aim in TERAFLUX.

2.3.3.5 Setup #5: Virtual Machines running on a SINGLE Physical computer, Flexible Programming Model on top of a Shared-Memory Guest

This setup resembles the previous one but now we use a "standard" OS (like Linux). However, we need to perform a trick in the OS so that this main OS, which acts as "Master Node" is aware of all N of the VM instances (i.e., of the all SimNow or QEMU instances).The Guest therefore appears like a single node with as many cores as N x C_N (C_N is the number of cores provided by each VM). The "Slave" nodes just provide the cores in the same fashion as the Master Cores provides its ones. It is the Guest OS that provides the illusion of a large shared memory guest machine to the Applications. This solution is essentially the one already proposed in the Deliverable D7.1. In D7.1 additional details are provided. Those details will not be discussed here again.

The master core is aware of the precise memory map (Figure 8).

Setup #5

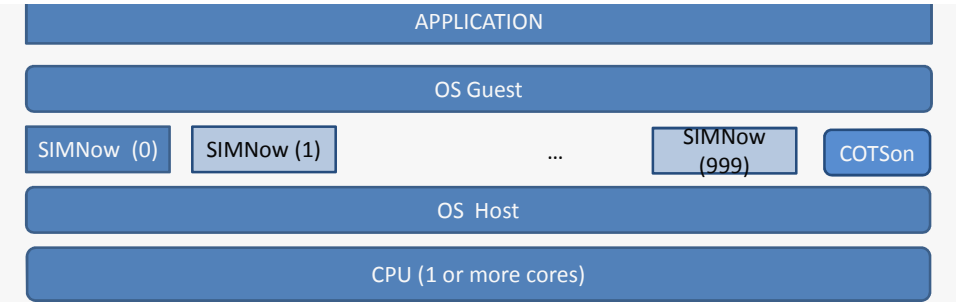


Figure 8 - One core aware of all the other cores (see also D7.1).

This kind of experiments is explored in Section 3.12 (OS support). In this case, we add a special module to the SimNow environment that maps a shared-memory block allocated on the host. This module presents itself to the guest OS as a physical device (e.g. PCIbar), and the shared block is seen by the guest OS as memory inside that device. This memory can then be *mapped* into the virtual

address space of guest processes using a special device driver in the guest OS. Also note that this setup is the same proposed in D7.1 (cf. D71. – Figure 9, reported below in Figure 9 for easy of reading)

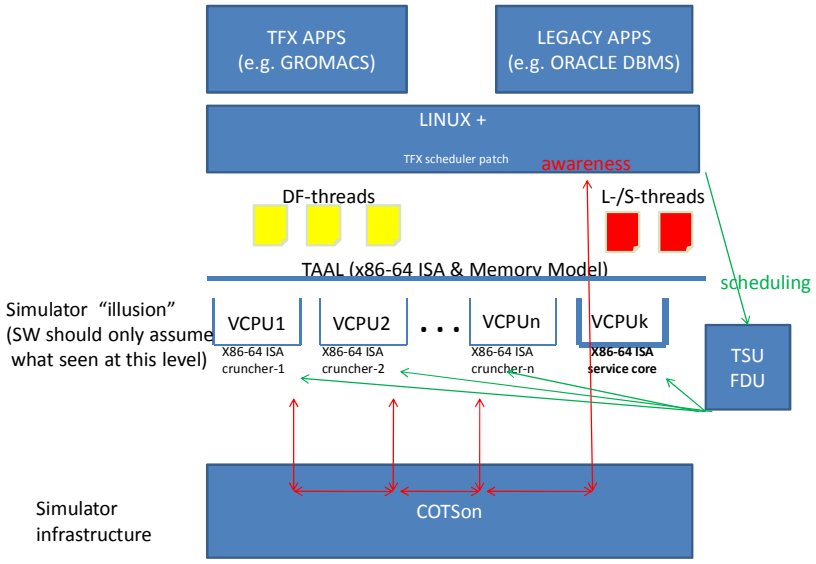


Figure 9 - The "simulator illusion" already proposed in D7.1 (this is the same as Setup #5).

Advantages:

- Allows us to run Shared Memory applications like OpenMP ones (can still run MPI as if it was a single big node).
- Can run both on a real machines (at least at small scale for tests) AND on the COTSon simulator as provided at the Month-1 of the TERAFLUX project.
- It allows us to modify system parameters like e.g. number of cores in each simulated instance.
- It allows for a parallelization of the simulation (the several instances are running in parallel on the available cores – load balancing automatically provided by the Host OS scheduler).
- Possible to avoid copying buffers among instances because they reside in the Host Shared Memory Network (operations for the support of the TERAFLUX Memory Model, such those described in sections 3.1.7, 3.1.8 may take advantage of this).
- Possibility to take advantage of RVI/VT-x virtualization mechanisms across different Physical Machines (under development).
- The communication and synchronization among the simulation instances adds up to the Application traffic, but could bypass TCP/IP and avoid using the Physical Interconnection Network.
- Load Balancing for the Application is managed by the Guest OS
- No need to use the Physical Network.
- No need to use a very different OS like an SSI OS.

- The underlying Guest Architecture is a shared memory machine, however thanks to the availability of a global address space, there is now full possibility of evolving the machine in a more "general one" like the one we aim to evolve during the TERAFLUX project. The TERAFLUX Execution Model can decouple completely the architecture of the machine.

Disadvantages:

- Relatively poor performance when N (number of nodes) increases; however, as other simulator like COREMU [Wang11] already demonstrated a high speed up in simulations even with 255 cores, we have good confidence that we can improve much the simulation speed going in a similar direction.
- Requires some patches to the Linux OS; however we shall need to patch anyway the Memory Manager and the Scheduler in order to properly support the TERAFLUX threads (DF+L+S) as outlined in deliverables D6.1 and D7.1.

2.3.3.6 Setup #6: SINGLE Virtual Machine running on a SINGLE Physical computer, Flexible Programming Model on top of a Shared-Memory Guest

As clearly shown in Figure 10, in this setup we substitute the many instances of several Virtual Machines (SimNow or QEMU ones) with a single instance of a Virtual Machine (COREMU [Wang11]).

COREMU [Wang11] is a scalable and portable full system emulator built on QEMU. Currently, COREMU supports x86-64 and ARM (MPCore Cortex A9) target on x64_86 Linux host system. COREMU is able to boot 255 emulated cores running Linux on one testing machine which has only, e.g., 4 physical cores with 2GB of physical memory.

Similar to the use of QEMU, in order to provide the timing models we need to patch this software in order to provide an interface to the COTSon. COTSon will provide the "timing feedback" as explained in the Section 2.

Setup #6

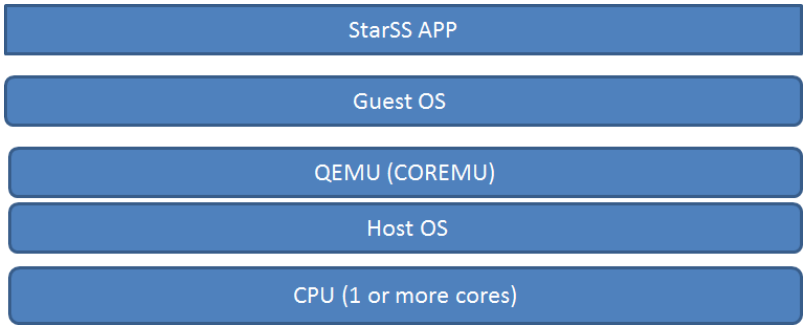


Figure 10 - One host CPU runs a VM with, e.g., 255 cores (COREMU), emulating shared memory communication.

Currently, there is the limitation of 255 QEMU instances. There are some approximations that allow connecting several QEMU instances via PCI [Gligor10] and the experiment in Section 3.9 (OS support) are usable also for this setup.

Advantages:

- Allows us to run Shared Memory applications like Open MP ones (can still run MPI as if it was a single big node).
- Parallelization is performed by several tricks inside the COREMU (they are in the form of patched to QEMU).
- Relatively acceptable performance when C_N (number of nodes) increases;
- It allows us to modify system parameters like e.g. number of cores in each simulated instance.
- Possible to avoid copying buffers among instances because they reside in the Host Shared Memory Network (operations for the support of the TERAFLUX Memory Model, such those described in sections 3.1.7, 3.1.8 may take advantage of this).
- Possibility to take advantage of RVI/VT-x virtualization mechanisms across different Physical Machines (under development).
- The communication and synchronization among the simulation instances adds up to the Application traffic, but could bypass TCP/IP and avoid using the Physical Interconnection Network.
- Load Balancing for the Application is managed by the Guest OS
- No need to use the Physical Network.
- No need to use a very different OS like an SSI OS.
- The underlying Guest Architecture is a shared memory machine, however thanks to the availability of a global address space, there is now full possibility of evolving the machine in a more “general one” like the one we aim to evolve during the TERAFLUX project. The TERAFLUX Execution Model can decouple completely the architecture of the machine.

Disadvantages:

- Requires modification of the COTSon in order to interface it with the COREMU (this work however is quite similar to that one necessary to interface QEMU).
- We do not know if we are able to overcome the current limit of 255 cores that are emulated by COREMU.
- We completely miss the timing interface as provided by COTSon (unless the modifications pointed out above are implemented).

2.3.4 The search for "efficient benchmarks/applications"

In order to adequately stress the 1000-core simulation, we needed some initial application that: i) do not present "algorithmic bottleneck" when scaling to 1000 or more cores; ii) reasonably load interconnects and memory (without an "exponential explosion" of the dataset). This is very important to adequately stress the 1000-core simulation. We call these kinds of benchmark "*efficient benchmarks/applications*".

While this analysis is performed more systematically in WP2 (see D2.1), we report also here some results done for the sake of immediately enabling partners to test the COTSon framework at a 1000-core scale.

A first step is detect if there are "algorithmic bottlenecks" is to analyze an application using tools like "cilkscreen" and "cilkview" to detect the *parallelism* of the application [cilk1, cilk2, cilk3].

Frigo, Leiserson, et al., defined the parallelism as the ratio between "work" and "span". The "work" is the total time needed to execute sequentially all threads in the Control-Data Flow Graph that represents the application. The span is the execution time of the computation on an infinite number of processors [Frigo98]. This gives us an upper bound on the available Thread Parallelism in the application: if there is not enough parallelism in the application, it's not worth to add more cores to the machine.

In order to do these tests, we currently need to use a different programming model like Cilk [cilk1, cilk2, cilk3]. However, this is very easy (Cilk has basically three construct to express parallelism: `cilk_spawn`, `cilk_sync`, `cilk_for`) for smaller programs. This topic is also investigated in WP2 (see D2.1 - sections 3.1-3.3).

For example, in Figure 11 we show the parallelism of the Simple Matrix Multiplication (same as in 2.3.2.2.1). This information tells us that in order have an efficient benchmark we must also consider an input data set consisting in matrices of size above 1024.

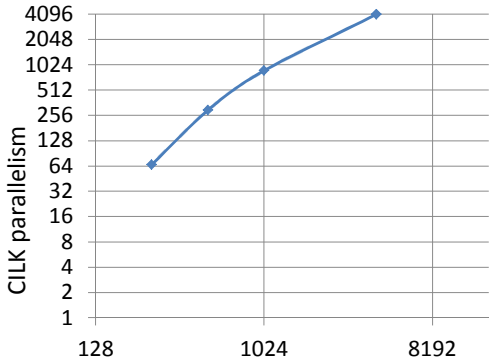


Figure 11 – The parallelism detected in a Simple Matrix Multiply benchmark for different matrix sizes.

2.4 Knowledge transfer and support model to the partners (Partners HP Labs, UNISI)

HP Labs made the COTSon simulation infrastructure available under an open source license (MIT) as part of Task 7.1 (months 1 – 12) (Knowledge transfer and Training). The tool is available for all partners at http://cotson.sourceforge.net. HP Labs also provided ongoing consulting and support to all partners to explain the simulation tool and customize it for each partner's research objectives

COTSon experiment distribution workflow

As shown in Figure 12 -, the first stage of our workflow development is to implement the TAAL subcomponents are individually and test them by using the COTSon simulator on the HW facilities locally available to the partners (codenamed "BabyCotson" machines). In a successive merging phase the results are validated so that experiments can be carried out on a simulation facility (codenamed "TeenagerCotson") made available at Partner UNISI, which includes all changes from every partner (Figure 12). What has been described in this document will be running and evaluated on the TeenagerCotson where all the subsystems must work seamlessly.

Starting from January 2010, an initial public release of the COTSon simulation infrastructure has been published under the MIT Open Source License. This allows people from all over the world to freely contribute to the development of COTSon. A public repository has been set up and is hosted on SourceForge.net. Subversion has been chosen as the revision control system for keeping track of all changes in the source files. Because COTSon is rapidly evolving and improving, the suggestion is to check out the development tree to get both the program sources and the extensive set of examples provided by HP partner.

UNISI hosts an additional local repository for TERAFLUX internal releases. That is why a local Subversion repository for the COTSon source code is held and managed by UNISI. All partners have access to it (same authentication access as the wiki collaborative site). The URL is https://teraflux.eu/svn/tfx.

Improving the COTSon impact and influencing the students

COTSon is written with portability principles in mind. However, it has been tested mainly on Ubuntu/Debian Linux distribution. Due to a larger availability of expertise on Fedora Linux Distribution, UNISI has also extended the availability for the Fedora distribution (specifically Fedora 14). This is also an important factor to enlarge the impact of the results of the project. Now, the sourceforge.net has been updated with also the UNISI contribution and can be used very easily on four very widespread Linux Distributions (Ubuntu/Debian, Fedora/RedHat).

COTSon is a very powerful tool, and thanks to the TERAFLUX project COTSon's usability was enhanced and therefore its acceptance will be expanded. In particular, we are now able to show that it

is relatively easy to scale up the simulation of Tera-device systems, encompassing e.g. 1000 cores (see also next Sections).

To show the improvement of the usability, UNISI has studied the reaction of students at the Faculty of Engineering of the University of Siena. The students were readily able to perform their experiments on different problems such as large databases on (simulated) datacenters and on larger experiments involving many thousands of concurrent threads. Furthermore, the students were also able, to modify part of the COTSon timers under our guidance.

One of the most important WP7 objectives for UNISI is also to integrate COTSon with QEMU which is open-source (under GPLv3 license). To cope with the GPLv3 at the goal is to provide "COTSon patches for QEMU", nevertheless this will largely increase the impact of the COTSon, for example to simulate other emulated platforms that are different from the x86-architecture.

Finally, to extend the availability of the COTSon, a smaller distribution "COTSon-on-a-stick" has been made available. Some initial presentation has also been done at popular events like the "Linux-Day". This also attracted the attention of other people of the Linux Community.

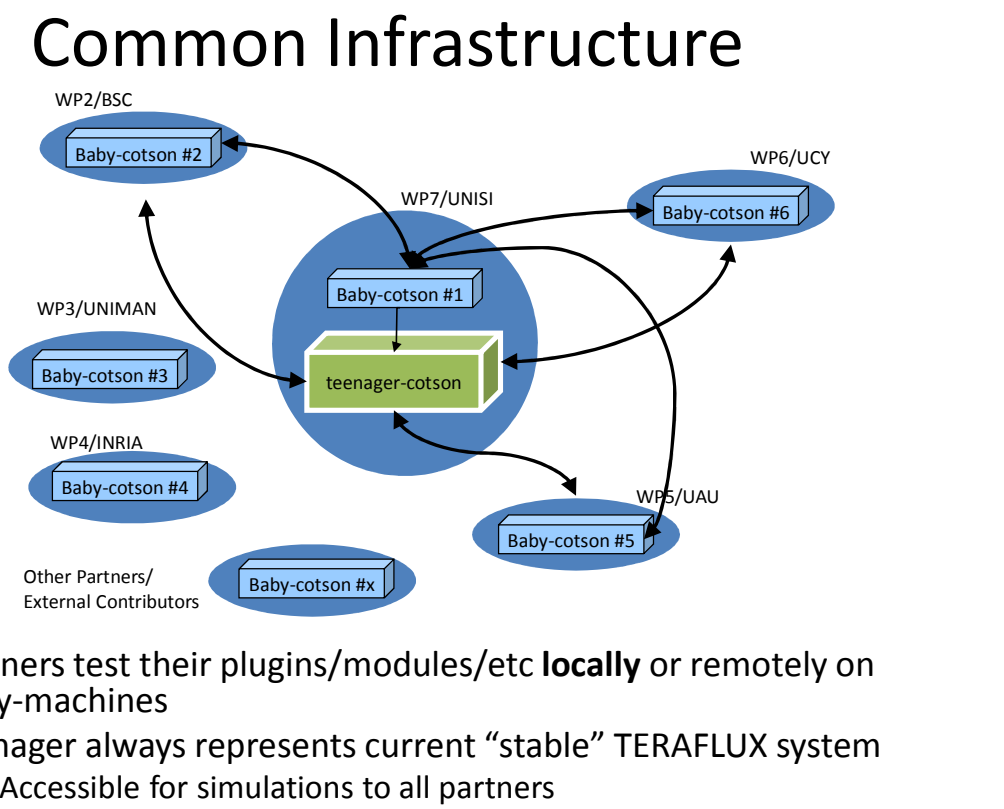


Figure 12 - COTSon experiment distribution among partners.

2.5 First COTSon simulations (Partners UNISI, HP)

2.5.1 Running examples on SimNow: 1 node with 32 cores

Currently, all the partners received an HDD image (karmic64.img http://wiki.teraflux.eu/upload) provided by Partner BSC with some of the applications analyzed in the WP2 (see also Deliverable D2.1).

Below we show some of the output of a COTSon full-system simulation in the case of a program that we also tested on the BSC's MareNostrum supercomputer (see Section 2.3.3): DMM (Dense Matrix Multiplier). In this case, the SimNow instance has a 32 core AMD models (so called "AweSim" model) with a nominal characteristics of 1600 MHz; the memory per nodes 8GB of DDR3 at 266Mhz. This example uses a mix of the StarSS and MPI (it is still an initial case study – not a final target for the TERAFLUX system)(Figure 13).

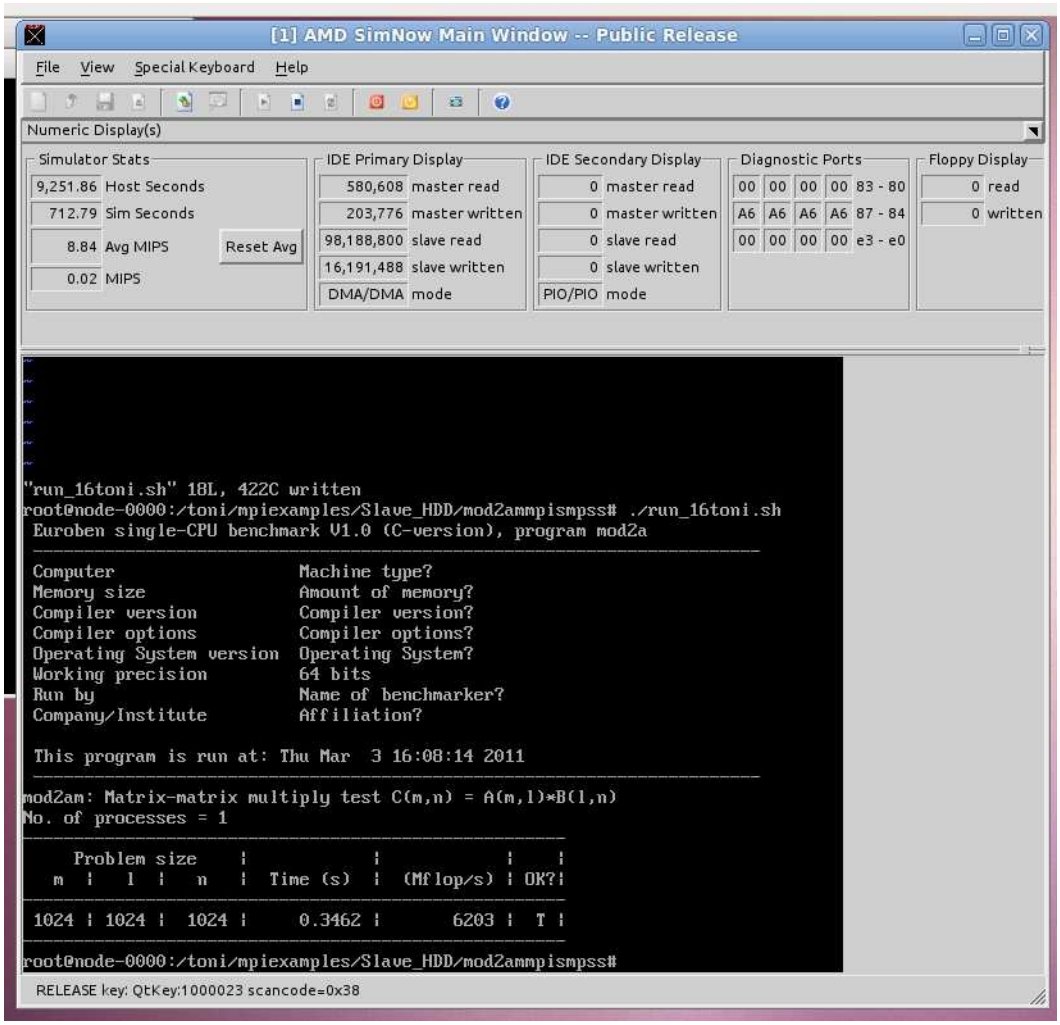


Figure 13 - Dense Matrix Multiplier (DMM) running on a SimNow instance with 32 cores. Input data are 2 square matrix of size 1024 CSS_NUM_CPUS (=number of "workers" in StarSS) is equal to 32.

2.5.2 Running examples on SimNow+COTSon: 2 nodes connected by the simulated Ethernet (Mediator)

We can run all the examples provided in COTSon. In Figure 14 we show the two nodes example that it is based on two VMs running a Linux pinging from one node to the other (setup #3, cf. 2.3.3.3).

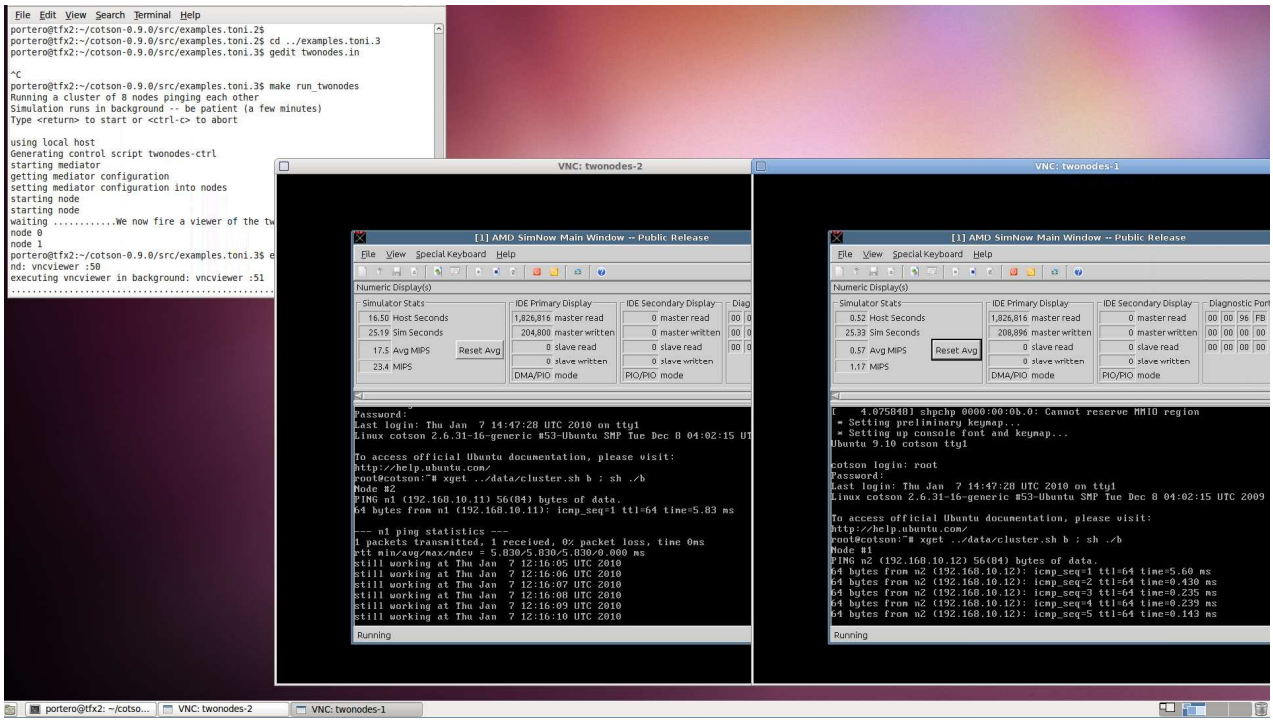


Figure 14 – Two nodes "*ping-ing* each other".

2.5.3 Booting 1000 thousand cores with SIMNow+COTSon – an instance of the TERAFLUX TBM (32 nodes x 32 cores)

Booting up 32 nodes with 32 processors each

This experiment is a relatively straight forward, once all the necessary setup is done, but it took us a considerable effort to be able to manage it for the first time. First of all we needed an adequate machine (see section 0). In particular, after verifying the possibility of setting up 32 nodes x 32 cores, we decided to use this node/core configuration. This is slightly different from our first idea of 64 nodes x 16 cores, but serves for demonstrating the feasibility of such simulation.

As explained in the Deliverable D7.1 this can represent an instance of the TBM (TERAFLUX Baseline Machine), which can be seen as a machine against which we would like to show the research improvements that we carry out during the project.

This represents one of the most important Milestones of the TERAFLUX project (M7.1): all the partners are not only able to simulate an initial test case in the common framework, but also an initial test case of an adequate size (1024 cores).

We report in Table 2 the memory and CPU utilization for booting 1024 cores (i.e., 32 nodes with 32 cores per node). Please note that even if in this experiment only 98 GB are showed as "used", in some initial phase of the boot process the memory request increases up to about 250GB.

Table 2 - Output of the "top" command under the Linux simulation host for the 1024 core experiment.

Cpu(s): 65.3%us	4.3%sy	0.0%ni	30.4%id
Mem: 264677636k total	**98931216k used**	165746420k free	69232k buffers
Swap: 67108860k total	0k used	67108860k free	543236k cached

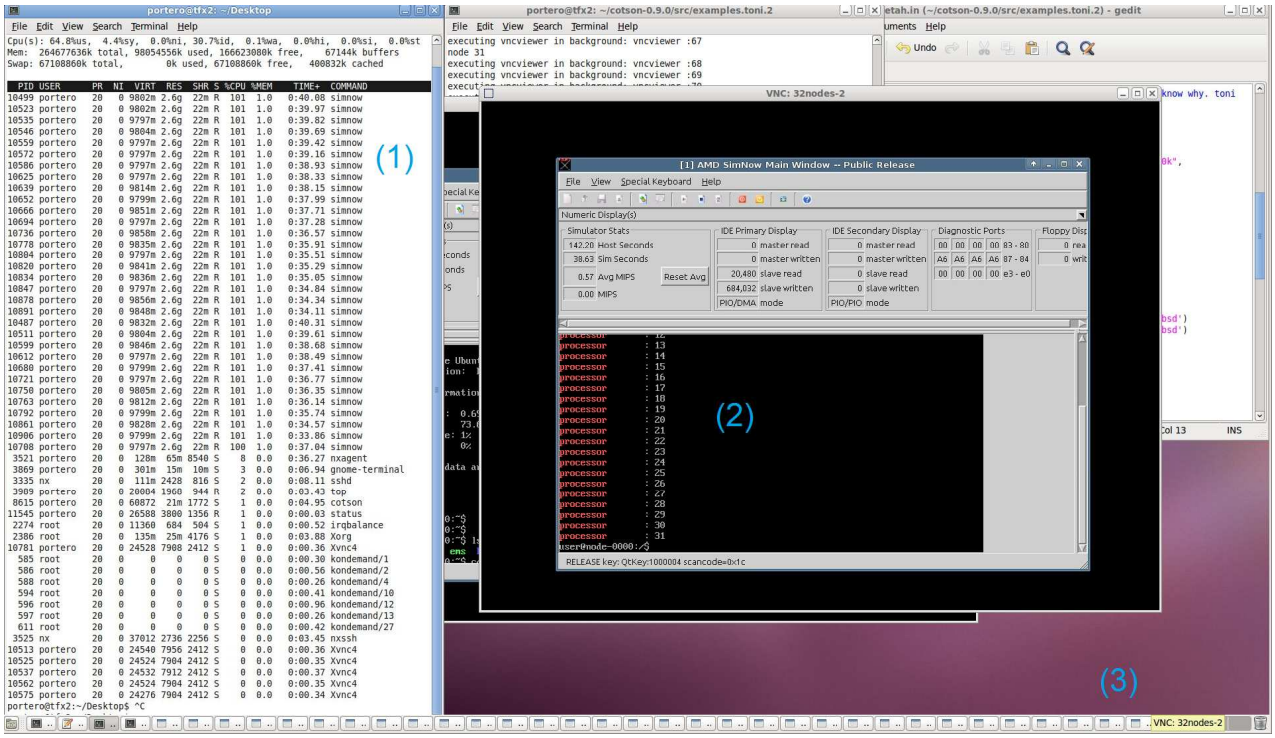


Figure 15 - Snapshot of the simulation of booting up 1024 cores. (1) COTSon execution of 32 SimNow instances. Command "top" shows the 32instances of SimNow. (2) Example of one VNC instance showing a SimNow instance with an AMD architecture x86-64 (BSD) of 32 cores (processors). Execution of the command *cat /proc/cpuinfo | grep processor* in the SimNow prompt (3) Thirty two VNC instances of SimNow. In total there are 32 instances of 32 cores, which sums to 1024 cores available.

In Figure 15 we show on the left the Linux `top` command. The `top` command provides the tasks that are being executed. In our case 32 SimNow instances; in the center a SimNow instance that executes the command*: cat /proc/cpuinfo | grep processor*. This command shows that the node has 32 processors. On the bottom of the snapshot, we can see the other 32 VNC SimNow instances all with similar environment.

3. Status of experiments and integration based on the COTSon simulation platform

In this Section we describe in more detail the status of separated experiments from the various partners and their effort towards the integration in the common platform.

3.1 Running benchmarks over the platforms under study (Partners BSC, UNISI)

In order to simplify the distribution of applications, and to provide a consistent simulation environment for all partners, BSC prepared a disk image containing several TERAFLUX reference applications. The image contains a minimal Linux installation as well, which enables booting it in COTSon (directly with QEMU), and simulating the pre-compiled applications with no additional hassle.

The image also includes an extensible management script for running applications or re-building the, which is explained on the wiki:

`https://wiki.teraflux.eu/bin/view/TERAFLUX/WP2/TFSM+EMS`

The image was distributed in a USB stick in the HiPEAC meeting.

The ISO image is available to the Consortium Partners in the collaborative wiki site:

`https://upload.teraflux.eu/uploads/COTSON_APPS/karmic64.img.bz2`

3.2 Simulation extensions and enhancements (Partners HP, UNISI)

Partner HP Labs worked on two specific areas of extensions and enhancements of the COTSon simulator for TERAFLUX: adding custom instructions (Section 3.2.1) and developing the interfaces for power modeling (Section 3.2.2). Partner UNISI worked on ISA extension in a QEMU (Section 3.2.3).

3.2.1 Mechanism to add Custom Instructions in SimNow and COTSon (Partner HP)

In order to provide a simple, extensible and efficient mechanism for custom instructions, we decided to overload the x86-instruction “CPUID”. The “CPUID” instruction intentionally includes an implementation-specific and vendor-specific behavior that we can use to extend the ISA with additional functionality. By default, CPUID read the EAX register and clobbers the EBX, EDX, and ECX registers. However, by providing appropriate compiler directives, we can also pass and return additional parameters and read/write simulated memory.

The following example () shows how to insert a custom instruction (called *__inst1*) in a piece of guest code.

```
inline static void __inst1()
{
    // Force a fence
    asm __volatile__ ("mov $0x8000000a,%%rax;\n cpuid;\n":::"memory");
}

foo()
{
    do_something();
    __inst1();
    do_something_else();
}
```

Figure 16 - Sample guest code using a custom instruction.

On the simulation side, implementing the custom instructions can be done either in the *analyzer* interface (to implement pure functional behavior within SimNow) or again the *monitor* interfaces (to also affect the COTSon timing analysis or other non-functional behavior). Regardless of the interface, since SimNow is a binary translating emulator, dealing with custom instructions involves

- Decoding and tagging the CPUID instruction during code translation.
- Catching a "special" set of reserved RAX opcodes during code execution.

For example, for the *analyzer* interface, this behavior could be implemented such as (simplified) show in Figure 17.

```
case ANALYZER_IN_TRANSLATE: {
    INTRANSLATESTRUCT *inst = reinterpret_cast<INTRANSLATESTRUCT*>(p1);
    UINT8* opc = inst->pOpcodeBuffer+inst->nOpcodeOffset;
    // Tag CPUID
    if (inst->nOpcodeCount >= 2 && opc[0]==0x0F && opc[1]==0xA2)
        inst->nInstructionTag = custom_inst_tag;
    break;
}
case ANALYZER_IN_EXECUTETAG: {
    UINT32 tag = reinterpret_cast<UINT64>(p1);
    if (tag == custom_inst_tag) {
        X8664SMMSTATE regs;
        cpu->GetX8664SMMState(&regs);
        UINT64 RAX = regs.IntegerRegs[15];
        switch(RAX) {
            case 0x8000000a: // The special INST1 code
                    run_inst1();
                break;

           default:
                break; // Not ours
         }
}
```

Figure 17 - Sample "analyzer" code intercepting a custom instruction in COTSon.

In the execution of run_inst1, we can read and write registers, operate virtual-to-physical address translations, read and write simulated memory (physical or virtual).

Using functional analyzers is convenient (and recommended) when implementing complex behavioral functionality that needs to be extensively tested without paying the price and burden of a timing backend. Depending on the specific implementation requirements, that functionality can then be migrated to COTSon itself, or a separate communication mechanism between the analyzer and

COTSon need to be established. The specification of this last mechanism is something we have started working on recently and we plan to release by the next milestone.

In COTSon (i.e., when using the *monitor* interface), we have built a special "tracer" wrapper class that achieves a similar functionality. However, impacting timing also requires correctly inserting events in the instruction queue so that the custom instructions are executed in the correct order with respect to the other regular instructions. This also requires a separate "simulation callback" that is only invoked when simulation is running a timing measurement sample. For example, an equivalent COTSon tracer that achieves the analyzer functionality expressed above would be accomplished by the following pair of private functions.

```
void inst1_functional(FunctionalState, uint64_t, uint64_t, uint64_t, uint64_t,
uint64_t);
InstructionInQueue inst1_simulation(Instruction*,uint64_t,uint64_t);
cpuid_call c1(0x1234,&inst1_functional,inst1_simulation); // registration

static char simbuf[20]; // Example of state passed between functional and
simulation

// The functional part (always executed)
void inst1_functional(
    FunctionalState f,
    uint64_t nanos,
    uint64_t devid,
    uint64_t a,uint64_t b,uint64_t c)
{
    // For example, read 20 bytes of the simulated memory buffer pointed to by 'c'
    // Notice the virtual->physical conversion first
    uint64_t vc = Cotson::Memory::physical_address(c);
    Cotson::Memory::read_physical_memory(vc,20,simbuf);
    // Do something with it...
}

// The simulation part (only executed during a sampling interval)
InstructionInQueue inst1_simulation(
    Instruction* inst,
    uint64_t nanos,
    uint64_t devid)
{
    uint64_t a,b,c;
    tie(a,b,c)=inst->cpuid_registers();
    // Note that for example we can't call Cotson::Memory here because we're past
    // functional execution, hence we have no injector available and
    // memory could have been overwritten in between.
    // Anything we need has to be collected in the functional part (eg. simbuf)
    cout << simbuf << endl; // Example, just print the simbuf out

    // Do something with the instruction, such as invoke the timer
    return DISCARD;
}
```

Figure 18 - Example "monitor" code for implementing the behavior of an instruction.

A complete example is available at http://cotson.svn.sourceforge.net/viewvc/cotson/branches/tflux-test/simnow/devel/analyzers/tm-test/ (for the analyzer interface) and (for the COTSon interface) at http://cotson.svn.sourceforge.net/viewvc/cotson/trunk/src/examples/tracer/. The custom instruction capabilities have been successfully used by UNIMAN to develop their initial Transactional Memory prototype.

3.2.2 Power Modeling (Partner HP)

In order to show how to collect and manipulate simulation events in such a way that they can be fed to other analysis (such as power) tools, we developed a mechanism that connects the results of a COTSon simulation to the McPAT power and area estimation tool (http://www.hpl.hp.com/research/mcpat/). McPAT (Multicore Power, Area, and Timing) is an integrated power, area, and timing modeling framework CPU architectures. It models power, area, and timing simultaneously and supports comprehensive early stage design space exploration for processor configurations ranging from 90nm to 22nm and beyond. McPAT includes models for the components of a complete chip multiprocessor, including in-order and out-of-order processor cores, networks-on-chip, shared caches, and integrated memory controllers. It models timing, area, and dynamic, short-circuit, and leakage power for each of the device types forecast in the ITRS roadmap including bulk CMOS, SOI, and double-gate transistors.

The cotson2mcpat conversion tool is a *perl* script that queries the COTSon result simulation database and leverages the McPAT XML interface, with a flow described by the following figure.

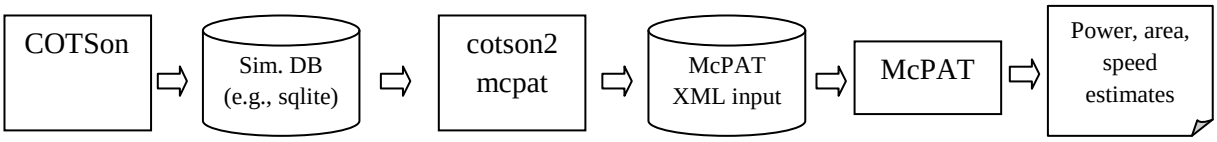


Figure 19 – Extracting power data in COTSon.

The script generates a McPAT input containing the relevant event information (instruction counts, memory accesses, etc.) that the power analysis tools then processes to produce an estimate of the simulation power consumption. The tool is currently designed to operate as *post-mortem*, but it would also be possible to invoke it during simulation (or repeatedly for individual simulation interval) to support generating power profiles over time.

The *cotson2mcpat* script traverses the simulation database using the SQL API and queries about the configuration parameters, the simulation sampling scheme, and the individual events. We expect this tool to be the foundation for the future TERAFLUX power and energy evaluations. Code and examples are available at http://cotson.svn.sourceforge.net/viewvc/cotson/trunk/src/mcpat/

3.2.3 x86-64 ISA Extension in QEMU Emulator (Partner UNISI)

We used QEMU as our x86-64 emulator and extended it with the new DTA instructions to support DTA. The following are the steps to add a new instruction to the x86-64 in QEMU:

1. Define a helper function that implements the new instruction. The helper function will be called by QEMU to emulate the new instruction whenever it is invoked in the program. The helper function name and input/output parameters are to be defined in the QEMU source file: qemu/target-i386/helper.h Example:

```
DEF_HELPER_0(TREAD, void) //output is void and 0 input parameters
```

2. Write down the implementation of the helper function defined in step 2. Add it to the QEMU source file: qemu/target-i386/op_helper.c, Note that you need to attach the prefix “helper_” before the name of the function. Example:

```
void helper_TREAD() {

	struct frame *f;
	struct continuation *cont=get_current();

	fprintf(fp_dta, "TREAD("TARGET_FMT_lx", "TARGET_FMT_lx")\n", EAX, ECX);

	if (unlikely(!cont)){
		printf("fload-ERROR: could not find a current continuation \n");
		exit(1);
	}

	f = &(cont->frame.frame);
	if (unlikely(f->__magic != SDF_FRAME_MAGIC)) {
		printf("fload-ERROR: invalid frame at "TARGET_FMT_lx"\n", EAX);
		exit(1);
	} else if (unlikely(ECX >= FRAME_SLOT)) {
		printf("WARNING: out of frame exception. Frame "TARGET_FMT_lx"\n",
EAX);
		exit(1);
	}
	EAX = f->data[ECX];
	stat.fload++;

	fprintf(fp_dta, "TReading EAX="TARGET_FMT_lu"\n", EAX);

	prn_ld((int64_t)&f->data[ECX]); //added by Rania
}
```

3. Select an opcode that is not used in the x86-64 ISA, the whole x86-64 ISA reference that defines the used and unused opcodes can be found at: http://ref.x86asm.net/coder.html . Example:

   `FF04`

4. Add the selected new opcode in the QEMU source file: qemu/target-i386/translate.c inside the function disas_insn in the case statement as one of the cases right before the “arith & logic” cases. Replace the first byte with 0x1. Example:

   `FF04` becomes `0x104`

5. Add the call to the helper function defined in steps 1 and 2 in the case body added in step 4 followed by a break statement. Note that you need to attach the prefix “gen_helper_” before the name of the function. Example:

```
case 0x104: //TREAD opcode

	gen_helper_TREAD();

	break;
```

3.3 ISA extensions and binary specification (Partners UCY, BSC, INRIA, UNISI)

In order to support the dataflow execution we propose to have some ISA extensions (Section 3.3.1). Targeting such instructions is also investigated in WP4 (see D4.3) by Partner INRIA. Together with Partners BSC, UCY and INRIA, UNISI is investigating a specification of the TERAFLUX binaries, such that they target the Execution Model defined in WP6 and initial specification done in the WP7 (D6.1, D7.1). In order to provide examples for the compilation tool-chain and the necessary modification in the compiler backend, we show code examples and provided an "imperfect" but fully functional tentative compilation tool-chain that uses modified software components (Section 3.3.2). Finally, we provide an example to show the first test runs of that program on the QEMU virtualizer (Section 3.3.3).

3.3.1 DTA-Transitional Instructions and Implications on Binaries that Are Targeted by the Compiler (Partner UNISI)

This specification is also part of the WP6 work. We report it here for convenience of the reader of having a self-contained document: Table 3is important to understand the examples and their output as simulated by the QEMU. Moreover, this document is going to be public, while other Deliverables still are under restricted release.

Table 3 reports a proposed instruction set extension (also indicated in D6.1 and D7.1) to support a DATAFLOW execution model for threads.

Instructions that are not reported in this table refer to the standard x86-64 instructions set *see D7.1 for more details).

In the examples below we consider a couple of simple programs that can be targeted by the compiler. Initially, a considerable effort has been spent in order to make these tools running in our internal tool-chain. We are now able to target fairly complex programs: however this effort is mainly intended to generate some training examples (as agreed at the kickoff meeting with partner INRIA, leading the WP4) so that the compiler could start targeting this code. Some results of this effort are also reported in the deliverable D4.3.

The Example 1 (Figure 20) shows a very simple program where we show how we can generate parallel code from sequential code in an automatic way. The granularity is extremely small in this case but this is only due to the simplicity of the example and the setting of the compiler to fragment the program at this level. More general program should be segmented at the granularity that the compiler deems appropriate.

The Example 2 (Figure 21) shows a recursive program that is compiled to exploit the dynamic execution model that builds on the DTA Execution Model. Similar examples are indicated in the D6.1, where also DDM is considered. Integration between DDM and DTA for a common compilation target has been also initiated and we plan to demonstrate more results of this in the common framework during the next periods.

Table 3 - DTA-transitional ISA extension. (The size of the operands is by default 1 machine word (e.g. 64 bits for x64 platforms)).

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

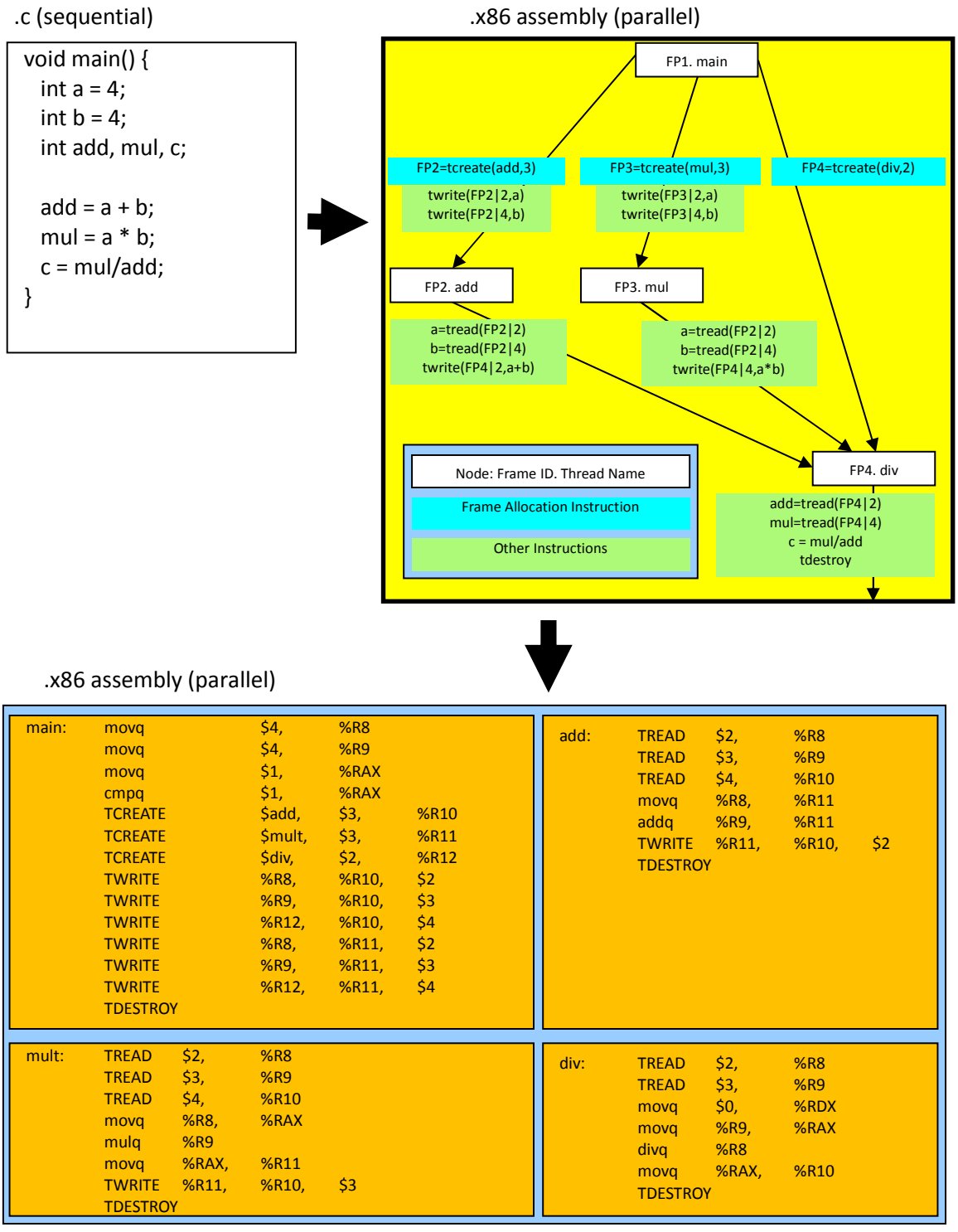


Figure 20 - Example 1: "Three-Treads"

Figure 21 -Example 2: Recursive Fibonacci

3.3.2 Preparing the way to compile from C to DTA-x86-64 instruction extensions

We describe here UNISI's internal tool-chain intended to show the feasibility of compilation form (sequential) C into (parallel) DTA-x86-64-assembly. These are the steps of our current tool chain:

1. The programmer writes the application in the high level language C. This is a sequential program.

2. We then run the sequential C program into a compiler (SCC, originally developed for the SDF architecture [Kavi01]) that converts it into a DTA intermediate representation (essentially the DTA legacy code – completely x86 unfriendly). This is the same application divided into parallel DF-threads.

3. We then use a translator to convert the parallel DF-threads into x86-64 instructions. Some DTA instructions cannot be translated to x86-64 because they are not supported by x86-64. We extended the x86-64 instruction set with new opcodes to support these new instructions. Moreover some tradeoffs had to be done in order to achieve a sensible result. The translator then converts the intermediate representations both to existing X86 instructions and to the new opcodes, which are represent the instructions in Table 3. The result is an application that is composed of parallel DF-threads written in x86-64 assembly coupled with new opcodes specifically for DTA-transitional instructions.

4. We then run the dataflow x86-64 assembly onto the QEMU x86-64-linux emulator to test our application for correctness. We of course, modified the QEMU x86-linux emulator to include the new DTA opcodes and their implementation as an extension to the X86 instruction set.

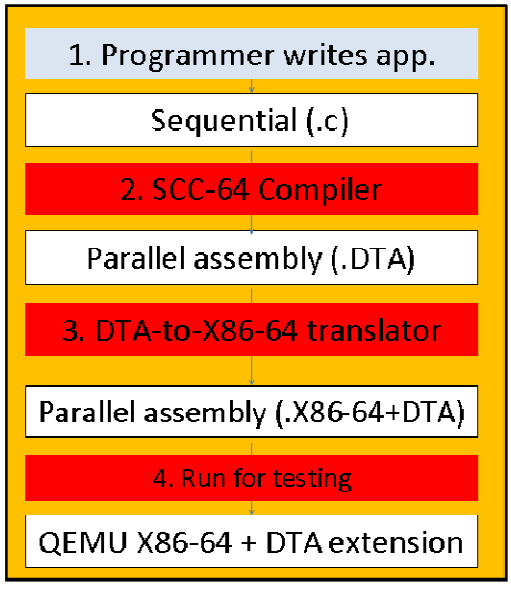


Figure 22 - From sequential to parallel programs with DTA extensions

3.3.3 An example of output obtained with the QEMU-DTA Example Fibonacci of N = 4 (UNISI)

Here we show the actual output of the first example of a C program, automatically generate and run on an actual simulation of the QEMU. These are very preliminary results, never published before.

We show them here to demonstrate our ability to run the generated code on the QEMU.

In the next period we can now both use QEMU instead of SimNow or reflect our ability of running such examples in the AMD SimNow. In both cases, we will be able to extract performance and power data for our examples.

When the WP4 will produce the GCC based tool chain, then we are able to demonstrate the most complex applications designated in the WP2. For the moment this will suffice us for tuning the architecture in WP6, and the simulation environment.

```
TSU Scheduling: dispatching continuation #0
switch_to: context switch from cont #2 to cont #0
switch_to: prevoius EIP = 000000000040066f
switch_to: new EIP = 000000000040066f
EIP==000000000040066f
TREAD(0000000000000000, 0000000000000002)
TReading EAX=5              //Result of fib(4) is 5
EIP==000000000040066f
EIP==000000000040066f
EIP==000000000040067d
TDESTROY(0) -- but still 0 are running
deleting continuation #0
switch_to: context switch from cont #0 to cont #1048576
switch_to: previous EIP = 000000000040067d
== TCREATE:                  15
== TREAD:                    44
== TWRITE:                   44
== TDESTROY:                 15
```

Figure 23 - Last line of the Fibonacci(4) program running on the QEMU DTA-x86-64 (the full output is reported in the appendix).

This experiment is intended to test and actually run in a Virtual Machine (QEMU) a program compiled for the DTA-x64 by extending the X86 ISA to include the instruction extensions (TCREATE, TREAD, TWRITE, and TDESTROY – see Table 3). The DTA instruction extensions are implemented in the X86 ISA in the QEMU emulator. The program we experiment with is *fibonacci* with parameter N = 4. The result is "5" as highlighted in the Figure 23. As shown 14 DF-threads were created using the TCREATE and 44 reads from frame memory and 44 writes to frame memory were done using TREAD and TWRITE respectively. The source code of the *fibonacci* in C is shown above as well as the DTA-x86-64 assembly. The QEMU emulator takes as input the DTA-x86-64 assembly of *fibonacci* and runs it using the x86-64 ISA + the DTA-transitional instruction extensions.

3.4 Custom devices (Partners UNISI, UCY)

In this Section we present some COTSon implementation of devices needed overall (NoC, Section 3.4.1), and as needed in WP6 such as Thread Scheduling Unit (TSU, Section 3.4.2). More details are given here on the NoC. The TSU is also described in more detail in D6.1.

3.4.1 Network on Chip (Partners UNISI)

The modeling of a NoC was not planned from the beginning but we have to do it to keep a credible platform for on-chip simulation of the TERAFLUX system. UNISI took the effort of modeling an off-the-shelf network (namely Xpipes [Bertozzi04]). Of course after implementing the basic model, we will be more quickly able to refine it with more sophisticated models if needed by the Partners.

Towards this goal, we use COTSon, which is a system level simulator that models nodes of multi-core CPUs, networking and I/O. It can run full OS and real application. To address the simulation challenge COTSon has a modular simulation infrastructure that decouples functionality and timing. COTSon uses AMD SimNow platform simulator for functionality and HP Labs timing for timing simulation. The functional simulation emulates the behavior of all components of the system. However it needs to verify the correctness of the performance results with the timing simulation. The timing simulation models the timing of all components. It measures performance (according to metrics) and informs the functional simulation of any new performance results. As one can see in Figure 24 fig. 1 the functional simulator is affected by the timing simulator, allowing the system to capture timing dependent affects.

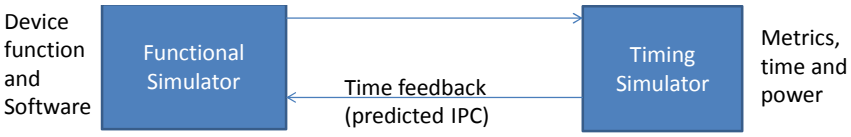


Figure 24 – COTSon's "functional-directed" approach: the time-feedback..

As we foresee that TERAFLUX will have a high required bandwidth, a Network-on-Chip needs to be used as a communication fabric to connect the many (possibly 1000 or more) processing elements; hence the tools that simulate these complex architectures should contain the simulation infrastructure of a NoC. This paper covers this subject. It brings the implementation and results to integrate a NoC with COTSon platform. The integration enables the behavior emulation of a system connected by a NoC. The NoC feeds COTSon with performance results (latency, throughput), as depicted in Figure 25.

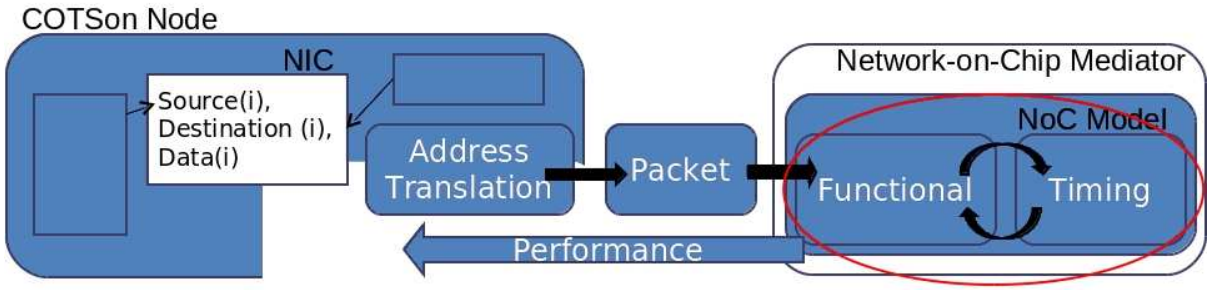


Figure 25 - Block Diagram of simulation with COTSon and NoC model.

The NoC model implementation is an interconnection network model inside the COTSon full-system framework (COTSon). The model consists of a low-level interconnection networks evaluation and models the detailed features of the state-of-art network. Researchers interested in investigating different network-on-chip can readily modify the modeled microarchitecture.

In Figure 25 one can see some details of the implementation. COTSon had a network interface to use Ethernet protocol, with this interface several nodes can communicate. The communication is done according to Figure 26. The MAC addresses of source and destination are translated to network addresses, and packets are used to transport data; however instead of a NoC (as shown in Figure 26) COTSon had a full network (with TCP/IP protocol - which is less efficient). The simulation becomes heavy for thousands of nodes; in this case we replace the old network structure for a simpler structure, as will be explained in the following section.

The NoC Infrastructure

The NoC is described in C++ and SystemC with a simple description of the routers architecture; hence the simulation of the NoC system is fast (less than 1ms for a 25x25 mesh and 10000 cycles of simulation). The NoC model follows the strategy used by COTSon; it is split in functional and timing simulation as depicted in Figure 26. The functional part contains the router blocks and the connections among them. The functional model is receives the packets coming from the source nodes and sends them to the destination nodes. The functional model sends the packet injection and departing time to the timing simulation when a performance result is requested.

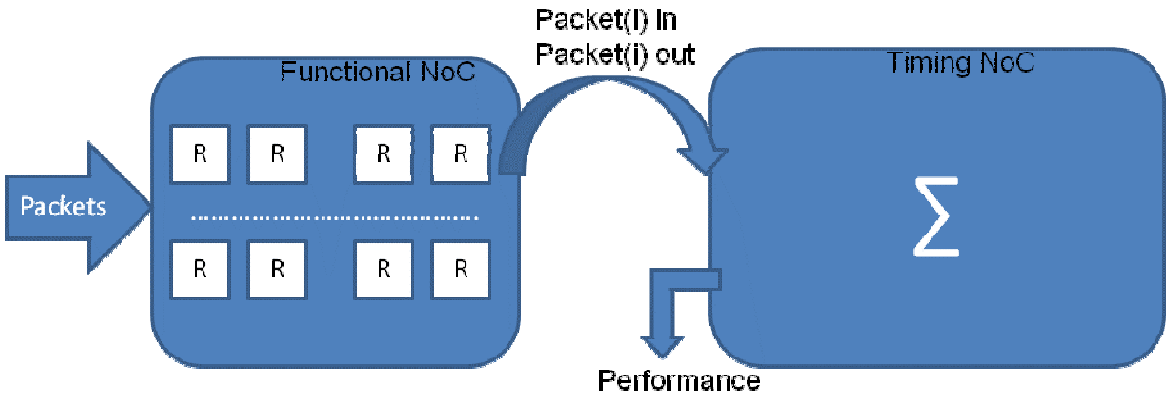


Figure 26 - Block Diagram of NoC model.

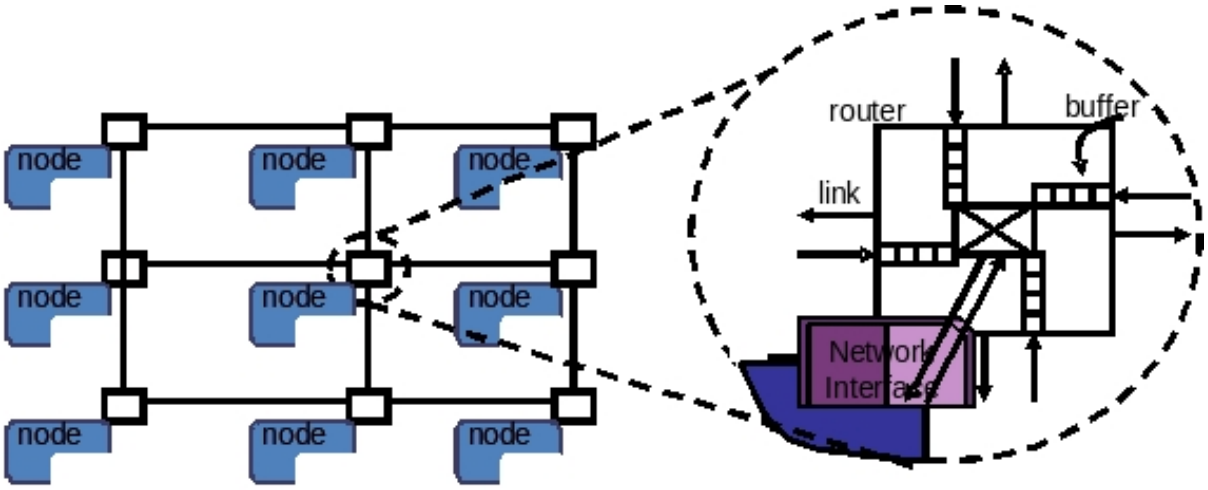
NoC Functional Implementation

NoC-Router design

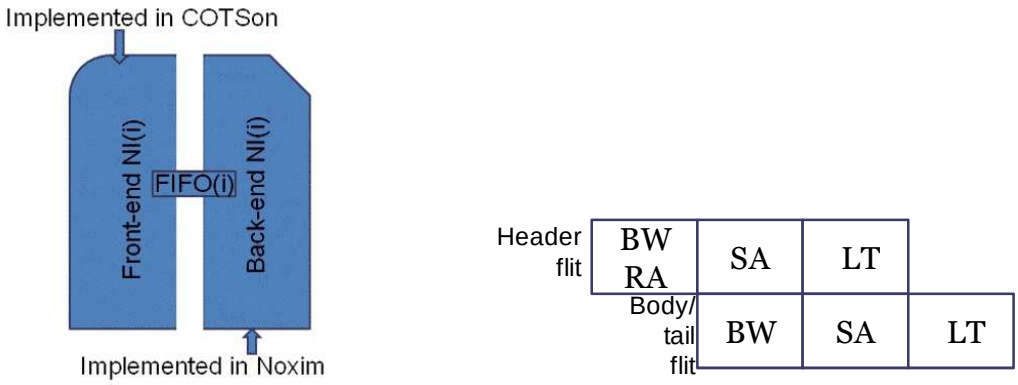
We used the NOXIM platform [Noxim09] as a start-up platform due to its simplicity of router description. In Figure 27a we show the NoC and the router block. The router can have any number of input and output ports depending on the topology and configuration. The major components of the router - which constitutes the router - are the input buffer, flow control logic and crossbar. The NoC is

modeled in flit-level buffering and these buffer sizes can be specified. The routing can be deterministic or adaptive.

In Figure 27c, we show the router pipeline that we model. A head flit on arriving at an input port, first gets decoded and buffered in the input buffer. Then, a request is sent to the routing algorithm simultaneously, and the output port for this packet is calculated. The switch allocation stage arbitrates for the switch input and output ports. On the winning switch, the flit moves to the crossbar. This is followed by link transversal to travel to the next node. In, e.g. a wormhole routing, body and tail flits follow a similar pipeline except that they do not go to the routing algorithm block. The tail flit on the leaving the router, realizes the path. This design was chosen because it is fast and of low complexity, while still providing reasonable throughput, making them suitable for the high clock frequencies. Moreover it has the following features: point-to-point links and switches, mesh or torus topology with configurable size, adaptive or deterministic routing algorithm and configurable buffer size.



(a)



(b) (c)

Figure 27 - Block Diagram of: (a) NoC and router, (b) network interface and (c) router pipeline.

NoC-Network-Interface design

One uses a shared FIFO to communicate COTSon platform with NoC. Each node to communicate with NoC has a FRONT-END NI, which creates and/or writes in a shared FIFO and a BACK-END NI. The COTSon node has the FRONT-END NI and the NoC has the BACK-END NI. The last one reads the shared FIFO created by COTSon node. That is, each node in COTSon has a FRONT-END Network interface and in the NoC the corresponding BACK-END NI as illustrated in Figure 27b.

The front-end NI receives from COTSon the following: **destination** address, **source** address, **data size** and **data**, which are all written in the FIFO. The pseudocode 1 above belongs to the FRONT-END NI in COTSon. This FIFO has infinite size once the NI cannot stop COTSon framework from sending data.

=> Pseudocode 1 of FRONT-END NI

```
1  void writeNI(int source, int destination, int data_size, int
   data[data_size]){
2  if FIFO not exist
3       id = source;
4       create FIFO(id)
5       FIFO(id) = destination, data_size,  data[data_size]
6  else
7      FIFO(id) = destination, data_size,  data[data_size]
8  end if;
9  return;
10 }
```

The BACK-END NI checks when there is new data to be sent. One checks when the FIFO has new data (as shown in line 2 of pseudocode 2). When the FIFO has new data it reads the data and sends it to the local output port of the router. The local port is responsible for sending data from the BACK-END NI to the local output port. The pseudocode 2 shows the BACK-END NI. **Payload** is the data to be transmitted through the NoC.

=> Pseudocode 2 of BACK-END NI

```
1  void readNI(int source){
2  if (!FIFO.newdata())
3       id = source;
4       destination = fifo(id);
5       data_size = fifo(id);
6       For i=0;i<data_size; i++
7       payload[i]= fifo(id);
8    else
9      do nothing;
10 end if;
11 return;
12 }
```

In Figure 28 we show a flowchart of the integration among COTSon nodes and NoC model. Each node in COTSon creates a shared FIFO with the node address of the NoC. Each node in COTSon translates the machine address to NoC address. The performance results as the data had been stored in a file; moreover the performance results are read by COTSon.

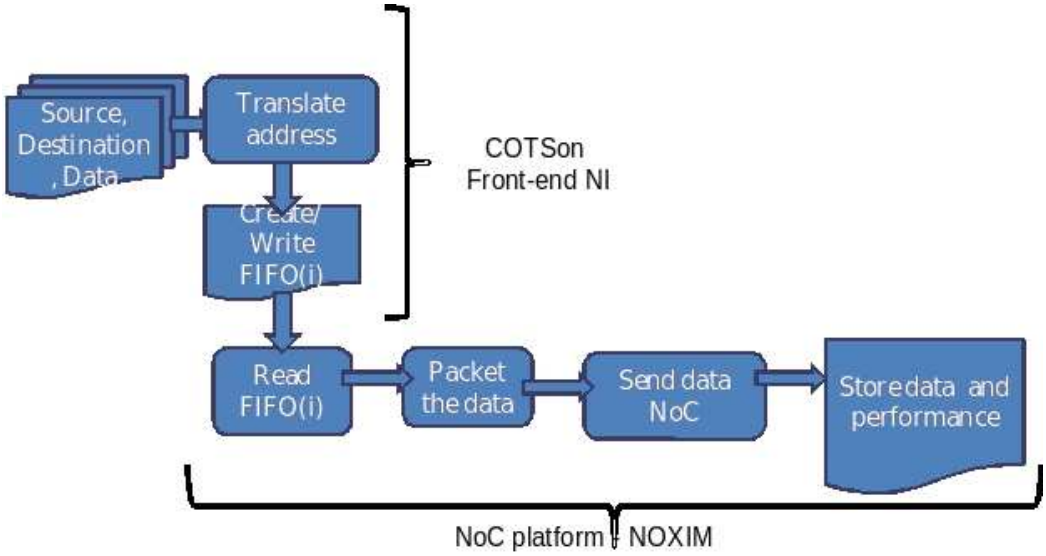


Figure 28 - Flowchart of the NoC model integration with COTSon.

NoC Timing Implementation

The timing simulation contains equations to compute the average latency. According to the packet injection and departing time one computes the latency of the NoC. That is, when the timing model is requested, it receives the inputs during the time of observation and computes the average latency as depicted in Figure 26. Using this strategy is possible to have the average congestion in the NoC.

Inside the NoC platform, in the file NoximNoC.cpp there is a set of functions that replaces the NI in COTSon. They are:

```
void NoximNoC::create_data(int local_id);

void NoximNoC::write_mem(int source, int destination, int *data);
```

The function *create_data* emulated the behavior of the node by creating data and *write_mem* emulates the behavior of the NI by creating and writing in a FIFO.

The status of this implementation is at an initial stage. We hope to finish the NoC implementation in the next period (Year-2). The result has been also developed according to the requirements in WP5, so that the WP5 work can later on rely on this COTSon implementation while WP5 will continue the development of their work without interdependencies. Nevertheless, we might expand the NoC configuration (adding virtual channels, etc.) hand in hand with the WP5.

3.4.2 Thread Scheduling Unit - TSU (Partners UNISI, UCY)

The TSU has architectural block has been described in the deliverable D6.1. An initial implementation of the TSU under COTSon has been initiated in terms of QEMU extension. The examples in Section 3.3.3 demonstrate also that the functionality has been correctly implemented. Following the "functional-directed" approach of COTSon, the next step is to implement the timing model.

3.5 External COTSon API extensions (Partners BSC, HP, UNIMAN)

In this Section we describe some of the work towards API extensions to allow the low-level control of the COTSon simulation.

3.5.1 Communication among simulation instances (Partners BSC, HP)

The standard COTSon model assumes that individual OS instances communicate through a set of guest-exposed communication mechanisms and devices, such as (simulated) standard Ethernet. However, this approach has some important limitations in light of the TERAFLUX needs to exchange fine grain messages that should not be loaded with the overhead of complex communication mechanism such as TCP/IP.

In order to provide a more extensible and flexible communication mechanism, we have started developing the foundation for a back-channel (simulation side) mechanism that enables COTSon instances to exchange data without going through the Ethernet interface. HP Partner developed a simple example showing how two COTSon instances can communicate through a combination of host and guest commands. This is the foundation to model a global non-coherent. The discussion of the exact mechanism is still ongoing and several alternatives are being evaluated. Example code is available at http://cotson.svn.sourceforge.net/viewvc/cotson/branches/tflux-test/messaging/

In Figure 29 we show an example of the operations involved in sending a message buffer from one COTSon instance to another without having to go through the simulated TCP/IP stack. Basically, when the guest#1 (in COTSon instance #1) wants to send a message, it issues a special instruction (using the CPUID mechanism described in section 3.2.1) to inform the simulation system about the location of the message buffer (gbuffer1) and where it wants to send it to. The simulation runtime uses the buffer and receiving node information to reserve space in the receiving node, then reads the simulated memory locations (after performing a virtual-to-physical translation) into a local coalesced buffer (sbuffer1), which it transfer using standard host-to-host communication mechanisms to the host running the COTSon instance #2. The runtime system in the simulation of COTSon instance #1 receives the message buffer in its own local buffer (sbuffer2) and then proceeds to write it into a predefined simulated memory area (gbuffer2). After successful completion of writing gbuffer2, the guest #2 can be notified of a new message being ready for processing. The advantage of using this method, despite the complications of multiple buffer copying operations, is that most of the processing happens on the simulation side, so we can keep the effect on simulated time under control, depending on the specific communication mechanisms that we want to model.

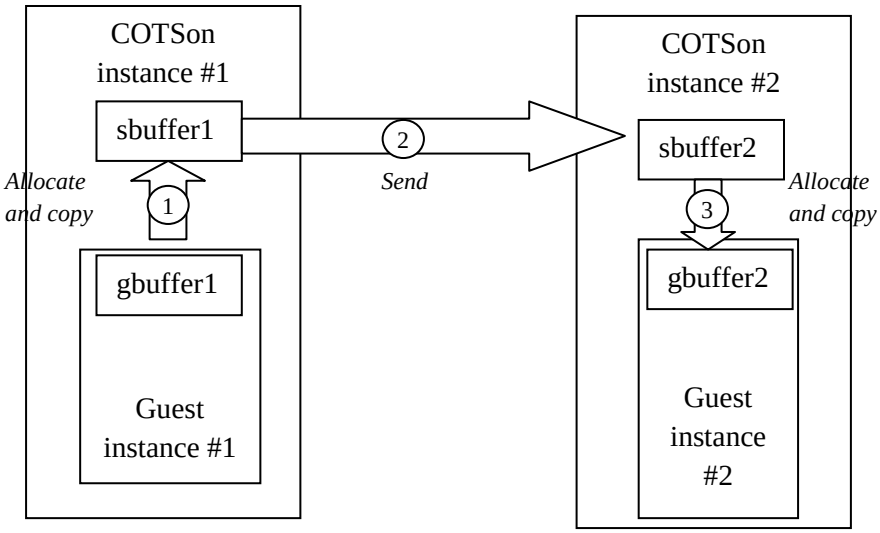


Figure 29 - COTSon communications experiment using simulation-side channels (i.e., without using simulated TCP/IP).

3.5.2 Release consistency experiments (Partners BSC, HP)

In the deliverable D7.1, we posed the foundation for the Memory Models (FM, TM, OWM, TLS + Code Memory).

One of the major simulation challenges of modeling the various memory spaces that have been defined in the TERAFLUX Memory Models deals with supporting a form of mechanism to handover the control on portion of memory across multiple simulation instances. In order to achieve scalability to the desired range (1,000 cores), it is not conceivable to have a fully coherent single memory address space that all cores can access. This is something that would be both unrealistic from an architecture perspective, but also unrealistic from a simulation point of view. As a consequence, multiple simulation instances (each running its own guest operating system or microkernel) and multiple simulated memory spaces (each under control of the individual guest kernel) are a necessity. At the same time, certain TERAFLUX memory spaces require transparent global addressing, so an important enhancement of the simulation toolset deals with how to reconcile the two contrasting requirements. In order to support that, we have analyzed several alternatives and have started evaluating some.

While this may appear as some form of Distributed Shared-Memory (DSM), we need to clarify that our final goal is to propose mechanisms that can work on a single-chip (see D7.1) and does not necessarily involve copying regions of memory from one core to the other of the TERAFLUX architecture. Supporting DSM over a set of communicating operating systems is an old and well understood problem for which several solutions have been proposed in the past. In the following sections, we describe some of the pros and cons of the alternatives we evaluated for COTSon and TERAFLUX.

3.5.3 A communication mechanism among separated COTSon/SimNow instances (Partner BSC)

The communication among simulation instances can be triggered by means of a special instruction (namely, ***CPUID***), in the sense that the simulator is able to capture and act upon, thus extending the ISA semantics of the simulated TERAFLUX architecture.

The mechanism pre-allocates a copy of the (guest) memory segment on each node (COTSON/SimNow instances), at a fixed virtual address. This mechanism should not be exposed to the programmer (applications in WP2) but allows the TERAFLUX runtime to explicitly issue data synchronization to COTSon in the form of acquire/release requests, whenever we need to make such segment modifiable in one of the two separated COTSon instances. A simple consistency model can be implemented: whenever a thread finishes writing to a shared object, it initiates a release operation that triggers COTSon to copy (broadcast) the new data to all other copies of the shared segment, residing in the different SimNow instances. Note that the data broadcast is executed by COTSon directly on the host, not the guest, therefore does not consume guest cycles.

The proposed implementation uses of acquire/release memory therefore guaranteeing that a sharable datum is only valid after it has been acquired, and before it is released. It is important to note that this mechanism enables concurrent sharing of read-only data: if two nodes acquire a released object, they both share a consistent view. However, if either node writes to the object, its consistency is no longer well-defined. This should not be a problem in the context of TERAFLUX, as it is assumed that concurrent writes (or asymmetric reader/writer sharing) are prohibited, and are prevented by either the dataflow semantics, or the TM mechanism.

The release consistency model could serve as an underlying memory semantic for all the memory regions defined for the project: the *private memory* or Thread-Local Storage (TLS) and the *sharable memory* or Frame Memory (FM), Owner Writable Memory (OWM), and Transactional Memory (TM). The triggering of object acquire and release operations does not require modifications to the application source code, but can rather be generated by the programming model's runtime, compiler, or TERAFLUX-specific architectural modules, such the TM implementation or the TSU. This mechanism could not be very efficient (due to the TCP/IP+copying), but we will investigate more efficient mechanism in the next periods.

The following consists, first, of a description of the API available to guest applications running in the TERAFLUX system and, then, a description of how this is implemented to make all simulator instances keep a consistent view of the system.

3.5.4 Memory model interface

All of the following routines are available to the guest system through the ***libtf*** library, most of which translate into the aforementioned special instruction (***CPUID***). This library must be first initialized by invoking the ***tf_init*** routine (residing in the ***tf_common.h*** header), prior to utilizing any of its routines.

3.5.4.1 Inter-process "sharable memory" (*tf_mem*)

The functional simulation of memory is provided, first, by the ***tf_mem*** subsystem (***tf_mem.h*** header). As TERAFLUX applications are intended to behave like a single multi-threaded application (thus

having a global address space), TERAFLUX-enabled processes can exploit this mechanism to allocate a sequential chunk of bytes when the subsystem is initialized (controlled by the ***TF_MEM_SIZE*** and ***TF_MEM_ADDRESS*** environment variables). Thus, all of them will allocate the same amount of memory at the same address.

In the case we run the COTSon on the some simulation host (cf. Setup #3, #4, #5, #6 – Sections 2.3.3.3, 2.3.3.4, 2.3.3.5, 2.3.3.6), the SimNow instances have an additional benefit for the simulation time: this memory is really shared among all the processes and in particular with each SimNow instance. Therefore, for the simulation purpose such memory will be functionally coherent among all processes running on the same instance. This has the added benefit that the functional simulation will not need to perform intra-node memory copies.

3.5.4.2 Sharable Memory allocation (*tf_mm_alloc*)

Sharable Memory allocation is provided by the ***tf_mm_alloc*** subsystem (***tf_mm_alloc.h*** header) which provides two different sets of operations.

On one hand, there are the ***tf_mm_alloc_malloc*** and ***tf_mm_alloc_free*** routines that will perform application-wide dynamic memory allocations and deallocations. These allocations will all happen inside the chunk of memory managed by the ***tf_mem*** subsystem, while regular dynamic memory allocations (i.e., standard ***malloc***) will appear as private to the process (i.e., allocating "TLS").

Additionally, there is a pair of routines implemented using memory consistency operations: ***tf_mm_alloc_set_global*** and ***tf_mm_alloc_get_global***. These two operations interact with the *global pointer*. This global pointer solves the "bootstrap problem" of communicating the address of the first dynamic memory allocation to all the processes, such that any process (usually the master process before invoking ***tf_xm_start***) can perform a memory allocation and let all the other processes know that address through these two routines.

3.5.4.3 Sharable Memory consistency (*tf_mm_cons*)

Memory consistency is provided by the ***tf_mm_cons*** subsystem (***tf_mm_cons.h*** header) and provides all the operations related to memory acquire and release, as mandated by the memory consistency model provided by the TERAFLUX system (cf. D7.1, D6.1).

On the lowest level, there are four routines (all of which must refer to memory provided by the ***tf_mem*** subsystem):

- ***tf_mm_cons_try_shared_acquire (address, size)*** : Tries to acquire a chunk of memory only for read purposes at an arbitrary address and of arbitrary length. If any other process has acquired for write purposes (exclusive acquire) a chunk intersecting with this one, the operation will fail and return the address of the first byte that could not be acquired. The same chunk of memory can be acquired for read purposes by multiple processes.
- ***tf_mm_cons_try_exclusive_acquire (address, size)***: Similar to this routine, performs an acquire intended for both read and write purposes, with the exception that only one slave process can have the exclusive ownership.

- ***tf_mm_cons_try_upgrade (address)***: Performs an upgrade from a shared acquire to an exclusive acquire, if no other process has this chunk as acquired.

- ***tf_mm_cons_release (address)***: Releases an already-acquired chunk of memory.

As the first three operations might fail and the simulator infrastructure cannot block the execution of a process until the operation succeeds, there exist three helper routines that loop until the related operation is successful:

- **tf_mm_cons_shared_acquire**

- ***tf_mm_cons_exclusive_acquire***

- ***tf_mm_cons_upgrade_acquire***

3.6 Implementation of Fault Detection Unit (FDU) and its interfaces (Partner UAU)

The main work of UAU with respect to WP7 will concern

- the implementation of the FDU and its interfaces (part of Task 7.2) and
- the fault injection and fault tracking (part of Task 7.3)

In the common TERAFLUX simulator, both tasks require a more advanced implementation of the TERAFLUX architecture in COTSon and are therefore deferred to a later stage of the project.

As preparative work UAU has built up knowledge about the COTSon simulator (Task 7.1) by learning about COTSon and taking part in COTSon trainings. UAU has participated in a Web-Seminar given by Vincent Lim (AMD) on 5. Nov. 2010.

Furthermore, UAU has bought an AMD Opteron-based multi-core machine (DELL PowerEdge R815) with 48 AMD cores and 128 GB RAM memory for TERAFLUX simulations – an investment of € 12,976 brought into TERAFLUX project as additional resources (not paid by TERAFLUX project). UAU installed the COTSon simulator on the multi-core and performed the following initial experiments:

- A COTSon node based on SimNow simulating up to 8 cores was running on a single Opteron core.
- 40 COTSon nodes simulating one core each were running on 40 Opteron cores. Communication between the COTSon nodes over the COTSon mediator was tested by a simple ping program (cf. Section 2.5.2).

The experiments have built up the know-how of how to work with COTSon such that an FDU implementation within the COTSon simulator can be done in the second project year as part of the overall TERAFLUX architecture simulation.

Moreover, as specified in Task 7.3, the fault models with the types of faults that have to be modeled were specified in Work Package 5 (Reliability) and are part of Deliverable D5.1.

Preliminary investigations concerned the injection of faults via the SimNow based COTSon cores. It was found out that SimNow contains the Performance Counter registers of AMD processor cores, but does not update them. So we think that explicit setting of the Performance Counter registers of SimNow can be used for fault injection of core internal faults in a later stage of the project.

3.7 Data-Driven Multithreading (DDM) (Partner UCY)

UCY created a runtime system for the COTSon platform that supports the Data-Driven Multithreading (DDM) model. This system has been proven to scale up to 12 cores (currently the largest COTSon setup at UCY) and can execute any DDM application. The applications tested are the Blocked Matrix Multiplication and Cholesky. The applications along with the system can execute on virtually any BSD with the Karmic Koala image (Ubuntu 9.10). The runtime and the applications have been bundled together into the image that was uploaded to the wiki site. These can be found it in https://wiki.teraflux.eu/upload/ under the folder /WP7/UCy. No extra configuration or libraries are needed in order to have a fully functional system.

3.8 Transactional Memory (Partner UNIMAN)

This section describes work done by UNIMAN with respect to WP7. The main focus of the discussion is the progress made so far in implementing a functional Transactional Memory (TM) system simulation in SimNow.

3.8.1 Transactional Memory Implementation

At Manchester, we have developed a functional TM simulation in SimNow as an *Analyzer* module. The TM module, including source code, can be downloaded from SourceForge website under the project name COTSon http://cotson.svn.sourceforge.net/viewvc/cotson/branches/tflux-test/simnow/devel/analyzers/tm-test/ . We must emphasize here that the functional simulation is by no means complete and we are constantly updating and extending it.

For a description of Transactional Memory, and a broad discussion of various design options, readers are referred to deliverable D3.1. The following sections briefly talk about the properties of TM that are implemented in our functional simulation and the extensions that we are planning.

We investigated several TM approaches, and they are currently being developed and tested in the simulation. In this section we give some details about the work in progress.

A. Data Versioning

A transaction needs to read and write data during its execution. Modifications made by a transaction must be done in isolation and must not be visible to other transactions until it successfully completes. Data versioning handles simultaneous storage of committed (*old*) and uncommitted (*new*) data. There are two main approaches to implement data versioning: *lazy data versioning* and *eager data versioning*.

1. Lazy Data Versioning

In lazy data versioning, a transaction performs memory updates by first making a private copy of the data it wants to modify and then performs the updates to the private copy of the data rather than to its *original* memory location. Any further reads and writes to that data are made to the private copy. Other transactions can also concurrently modify the data by making their own private copies.

When a transaction commits, it updates memory with its private copies of the data. If a transaction aborts, it simply discards all its private copies.

2. Eager Data Versioning

In eager data versioning, a transaction writes to shared data directly i.e. *in place* and maintains undo information in a private log.

If a transaction aborts, it restores all the original values of the data by copying the values from its undo log. If a transaction commits, it simply discards its undo log.

3. Data Versioning in our TM Implementation.

The current TM implementation in SimNow implements both eager and lazy data versioning. This is a parameter of the system and can be set for a given experiment. Most of the testing is being done with lazy data versioning, as we are currently focusing on lazy versioning and lazy conflict detection, which is explained in the next subsection.

B. Conflict Detection and Access Visibility

All TM systems must provide a conflict detection mechanism in order to check whether conflicts exist among transactions. If conflicts between transactions are detected immediately when they happen, it is known as *eager conflict detection,* and if conflict detection is delayed (e.g. detected at commit time), it is known as *lazy conflict detection.*

Eager Conflict Detection

In eager conflict detection, conflicts between transactions are detected immediately whenever a transaction declares its intent to modify shared data. By detecting conflicts early the TM system aborts conflicting transactions and avoids any further loss of computation time or resource usage by the aborted transaction.

Lazy Conflict Detection

In lazy conflict detection, a transaction checks for potential conflicts with other transactions after it completes the execution of the atomic block. This mechanism reduces the overhead of performing conflict detection on every memory access but it may reduce resource usage efficiency by wasting resources in allowing transactions to continue execution even after conflict has occurred (but not detected).

Conflict Detection Granularity

Conflict detection granularity describes the unit storage over which hardware TM systems detect conflict between transactions. Most of the hardware TM systems use *word granularity* or *block granularity*, where block can be a cache line, page or any adjacent, fixed-size group of words. Unlike software TM systems, hardware TM systems can associate metadata directly with the data at word or block granularity.

Implementation in the our TM Simulation

Our TM simulation implements both eager and lazy conflict detection. Like versioning, this is a parameter that can be set for a simulation. Our functional implementation uses word granularity but this is just for testing purposes as we propose block level granularity for our future implementations.

C. Contention Management

Conflict between two transactions can be resolved by aborting one of the transactions and allowing the other to continue execution. A TM system typically has a *contention manager*, which implements *conflict resolution policies* in order to decide which transaction to abort. A major concern for TM system is forward progress, and the contention managers provide support to the TM system to guarantee forward progress.

Contention Management Implementation

In our functional TM implementation we use very simple contention management policies.

In case of lazy conflict detection, priority is given to the transaction that is in its commit stage. Lazy conflict detection has a forward progress guarantee and in case of conflicts at least one transaction always makes progress.

For eager conflict detection there is an implicit priority based on the unique hardware identifier of the core on which the transaction is executing. If the core ID is lower, we say its priority is higher. This is purposefully made simple and arbitrary just to check the correctness and working of the overall TM implementation.

D. Nested Transactions

A nested transaction occurs when a new transaction is started by an instruction that is already inside an existing transaction. The new transaction is said to be nested within the existing transaction.

There are three possible ways to handle nested transactions.

- Nested transactions can *flatten* all the transactions into one big transaction. This is the simplest strategy; however, it has the downside that a failure in an internal transaction will cause everything to abort all the way back to the outer transaction, so potentially losing a large amount of work for a passing failure. This is of particular concern if the inner transactions are working with highly contested data.

- Nested *closed* transactions. In this situation if the transaction succeeds or a parent transaction aborts, then the semantics are exactly the same as for the flattened transaction, but if a nested transaction aborts then only that transaction aborts, not all the parent transactions as well. In addition as this implicitly has independent read sets for each level of the transaction, then collisions because of earlier nested transactions will not result in the whole transaction aborting. This is probably the most intuitive form of nested transactions.

- Nested *open* transactions. In this version, once a nested transaction commits successfully, all its changes are permanent, so they will not be undone if the parent transaction later aborts, and visible to other transactions. Open nesting transactions can be thought of as being completely independent. This will make transactions very hard to reason about, and should be avoided.

Implementation of Nested Transactions

Our initial TM implementation flattens all the nested transactions. However, we plan to extend the system to support closed nested transactions.

Benchmarks

For testing purposes, we are running our own test cases, as well as some of the STAMP benchmarks (Bayes, Genome, Kmeans, ssca2, Vacation), simulating up to 4 cores. We are able to run all these tests correctly in the COTSon environment that includes our extensions.

TM Constructs

In our current COTSon implementation we only allow two keywords for transactions.

Transaction begin

```
__tm_begin()
```

and transaction end

```
__tm_end()
```

These mark the span of a transaction. All memory accesses within the span are treated as transactional.

3.9 OS support for TERAFLUX (Partner Microsoft)

3.9.1 Fast Simulation Exploration through Emulation Techniques (MSFT)

Before starting simulating the system, using the COTSon simulation environment, it is important to understand how the operating system together with the new S/W environment will be executed on the future "real machine". To fulfill this task we assume that

1. The system is built out of compute nodes, each node may contain, e.g., 16 cores.
2. At this point of the research we assume that there is a single node that runs Linux. We term that node "Service Node" or "Linux Node". Please note that since each Node may contain16 cores it is unlikely that the entire node will be faulty.
3. We also assume that all other nodes in the systems are dedicated to run a "TERAFLUX code" and so we term them "TERAFLUX Nodes" " (also named "Auxiliary Cores" in D7.1).
4. In the future we plan to allow soft partitions between the number of nodes that runs Linux and the number of nodes that runs TERAFLUX Code so we can change it at "boot time".
5. A TERAFLUX node is controlled by a pico-kernel (L4 [L4KA10] or NaNOS [Ayguade07], [NANOS10], [Teruel07])) which is in charge of:
    a. Checking when tasks are ready to be executed
    b. Copying parameters
    c. Scheduling tasks for execution
    d. Checking the health of the node and its relative speed so that the global scheduler could do the load balance work properly.

       Routing operations such as exceptions, system calls, and I/O operations (as these operations cannot be handled by the TERAFLUX nodes) to be executed on the Linux node and their results to be copied back to a well-defined buffer.
    e. Allowing code on any node to access the virtual address space which is common between all the physical nodes.

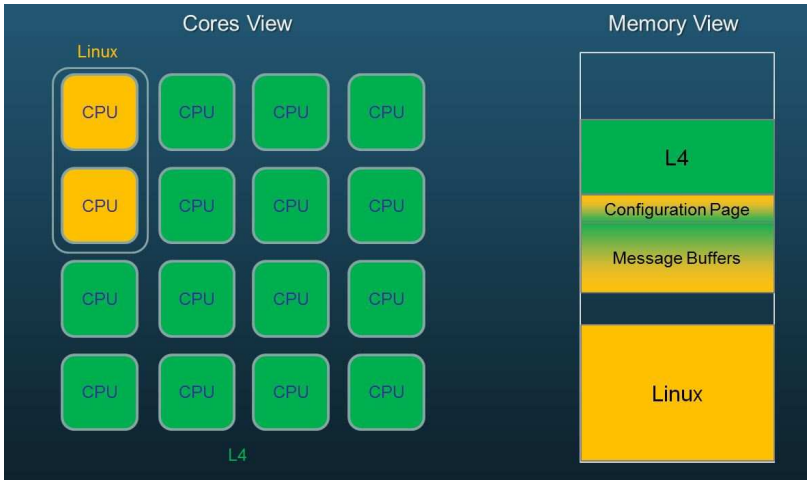


Figure 30 - Multi-core representation with Linux and L4.

During the first period of our simulation work we were focused on running the system on a real HW (8 way AMD machine which is partitioned between the service cores that run Linux and the TERAFLUX cores that runs L4), or using virtual processors on top of the virtualization layer.

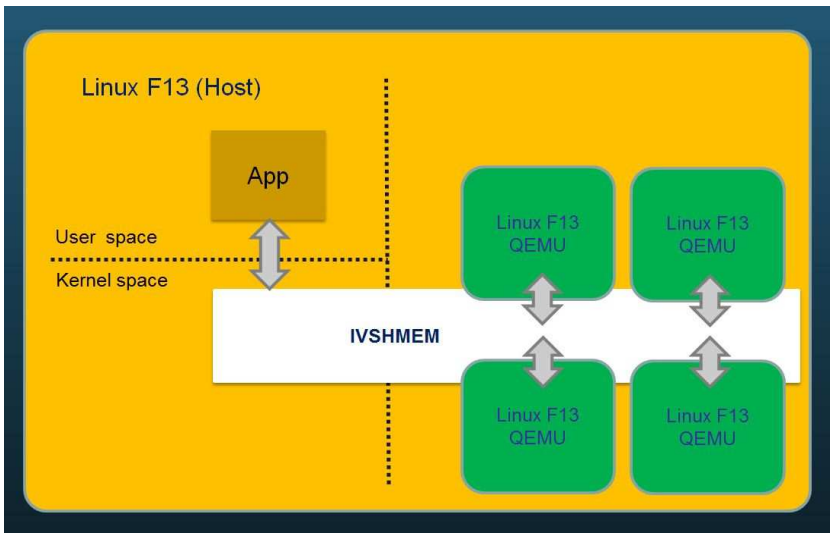


Figure 31 - QEMU instances with Linux connected through IVSHMEM

We implemented a system that can run instance of the L4 pico-kernel per physical node (Figure 30). Such a kernel represents the work a TERAFLUX node is doing. We also implemented a service node, running a Linux OS on a single core (Figure 31). All different cores share the common virtual address space, while the physical address space may or may not support full coherency.

At that point of time, in order to allow the simulation to include large number of cores (nodes), we allow each core, together with its own OS, to run on top of a virtual machine, so the number of simulated cores we are simulating is not limited.

We also build a layer of global virtual address space that emulates the hardware that will allow the access of the processor to local and remote memory resources (we have a working prototype, but the work is still "work-in-progress"). We also started to investigate other alternatives for L4, such as the use of the NANOS kernel which is part of the StarSS project. The advantage of this system is that it is very small, however it does not support I/O, signaling, and similar low-level- simulation primitives which are essential for the connection between the pico-kernel and the Linux.

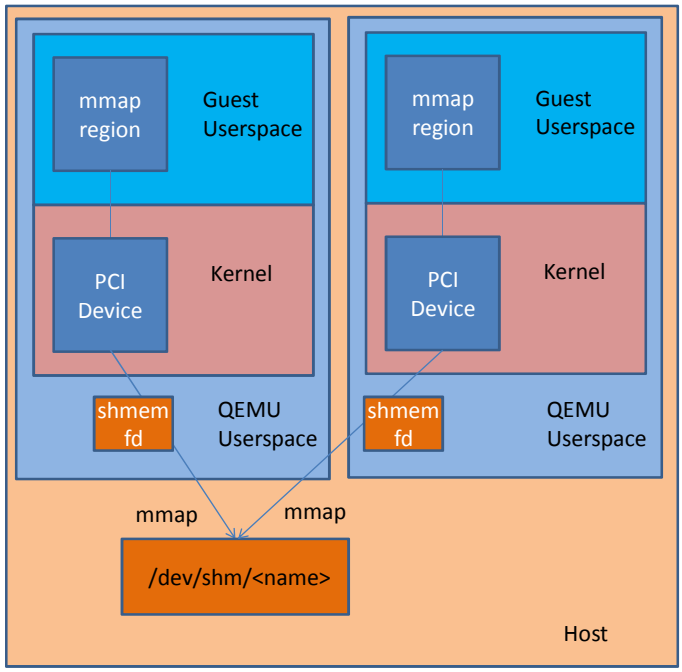


Figure 32 - QEMU maps a shared-memory area into RAM while it is exposed to the user as a PCI BAR.

The shared memory among the different nodes is implemented as a "mapped PCI memory" " (in this we follow a similar approach used in the Nahanni project [Macdonnel10]). By doing that, we can allow on one hand, to deal with local memory of each node and on the other hand, to emulate the ability to access any virtual memory remotely. The current version even requires doing some "fix-ups" when accessing the remote memory so that we can simulate the use of layer of message passing between the nodes to maintain SW consistency. This approach represents a lower level approach for implementing a communication method among QEMU instances (Figure 32, Figure 33). QEMU has been chosen as open source alternative engine to SimNow for the COTSon, as described in the WP7 Objectives.

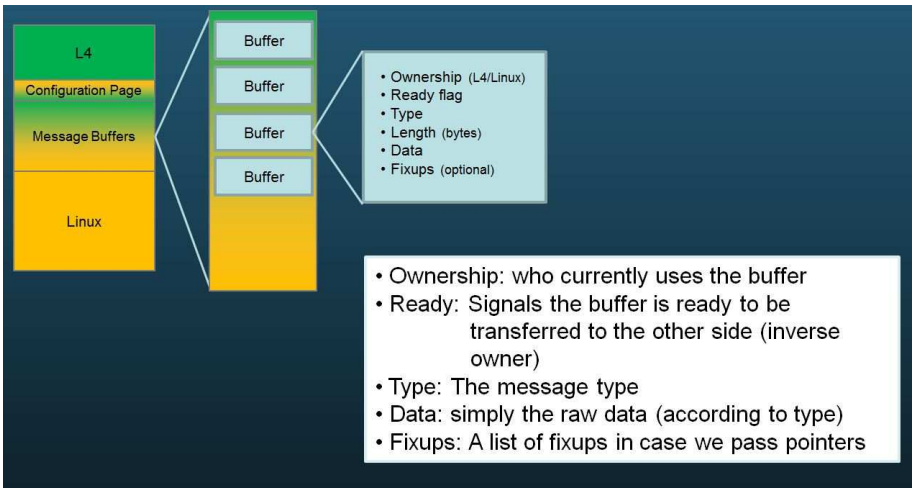


Figure 33 - Memory representation.

As next step, we would like to emulate the implementation of remote I/O from a TERAFLUX node to the Service node so that we could actually run a TERAFLUX code on a TERAFLUX full system.

As soon as we complete this part, we intend to move the mechanisms we developed on the stand alone simulator to the COTSon so that it could be integrated with all other activities the team is doing. After doing that we will start implementing fault detection, injection and handling mechanisms.

Please note that this part of the research is new and as far as we know was never done before. So we hope to be able to publish papers on it.

4. Conclusions

All the planned activities WP7 started and we successfully demonstrated the feasibility of simulating full-system at the scale of 1000-cores.

All the partners were involved in this Workpackage. Some partner like CAPS, THALES have started using the facilities at INRIA and are preparing to step up their contribution in the remainder of the project.

On the scientific side, we believe we are on a good path to show that it is possible to adopt a different execution model based on dataflow at thread level, and our initial experiments targeting a slightly modified x86-64 platform are encouraging us.

The Thread Scheduling Unit (TSU) lies at the heart of the new architecture: an initial implementation in the simulation framework demonstrates its feasibility and interoperability with a futuristic Tera-device platform. The integration with a Fault Detection Unit (FDU) will also start soon.

The current TERAFLUX simulation environment allow us to model the TERAFLUX architecture, considering for example interesting research directions towards 3-D stacked chip/package consisting of 1000-core, a NoC, a memory subsystem, without neglecting the necessity of running a full operating system and runtime to govern the I/O, legacy software, and other system processes (Figure 34).

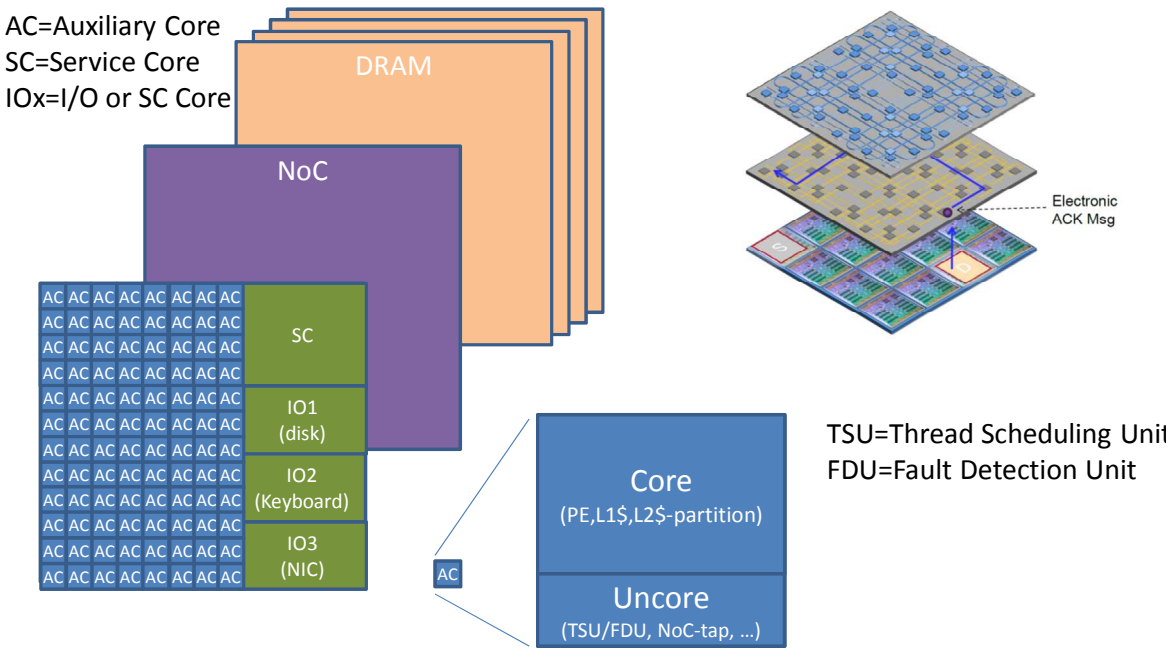


Figure 34 - Thinking at the simulation of a Future Tera-device System.

5. References

[AMD08] AMD White Paper, Virtualizing Server Workloads-Looking Beyond Current Assumptions, 2008.

[Argollo09] E. Argollo, et al. “COTSon infrastructure for full system simulation.” Operating Systems Rev., v.43, 52–61, 2009.

[Asanovic09] K. Asanovic, et al. “A view of the parallel computing landscape.” Commun. ACM, 52(10):56–67, 2009.

[Austin02] T. Austin, E. Larson, and D. Ernst, “SimpleScalar: An infrastructure for computer system modeling,” IEEE Computer, vol. 35, no. 2, pp. 59–67, 2002.

[Ayguade07] Ayguadé,. E.., . A.. Duran, . J.. Hoeflinger, . F.. Massaioli, and . X.. Teruel, "An Experimental Evaluation of the New OpenMP Tasking Model", Lecture Notes in Computer Science: Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing, vol. 5234: Springer, pp. 63-77, October, 2007.

[Beckman09] N. Beckman et al. , “Graphite: A Distributed parallel Simulator for Multicores”, Computer Science and Artificial Intelligence Laboratory Technical Report Massachusetts Institute of Technology, MIT-CSAIL-TR-2009-056

[Bellard05] F. Bellard, “QEMU, a fast and portable dynamic translator,” in ATEC’05: Proc. of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference, Berkeley, CA, USA, 2005.

[Bellens06] Pieter Bellens, Josep M. Perez, Rosa M. Badia and Jesus Labarta, "CellSS: A Programming Model for the Cell BE Architecture", in proceedings of the ACM/IEEE SC Conference, 2006.

[Bertozzi04] D. Bertozzi et al. “Xpipes: A network-on-chip architecture for gigascale system-on-chip”, Magazine, IEEE, Vol. 4, issue 2. 2004.

[Binkert06] N. L. Binkert, et al. “ The m5 simulator: Modeling networked systems.” IEEE Micro, 2006.

[Brewer92] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl, “Proteus: a high-performance parallel-architecture simulator,” in SIGMETRICS’92/PERFORMANCE ’92: Proc. of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems, New York, NY, USA, 1992, pp. 247–248.

[BSC10] www.bsc.es, December 2010.

[Bunting06] Trina Bunting, Wayne Kimble, “IBM Blade Center JS21 Technical Overview and Introduction, high-performance blade server ideal for extremely dense HPC clusters”, IBM.com/redbooks, March 2006.

[Chen09] J. Chen, M. Annavaram, and M. Dubois, “SlackSim: A Platform for Parallel Simulations of CMPs on CMPs,” SIGARCH Comput. Archit. News, vol. 37, no. 2, pp. 20–29, 2009.

[Chidester02] M. Chidester and A. George, “Parallel simulation of chip-multiprocessor architectures,” ACM Trans. Model. Comput. Simul., vol. 12, no. 3, pp. 176–200, 2002.

[Chiou07] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, “FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators,” in MICRO ’07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007, pp. 249–261.

[Chiou09] Derek Chiou, Hari Angepat, Nikhil A. Patil, and Dam Sunwoo, “Accurate Functional-First Multicore Simulators”,IEEE COMPUTER ARCHITECTURE LETTERS, VOL. 8, NO. 2, JULY-DECEMBER 2009

[CHU77] YAOHAN CHU, "Direct-execution computer architecture", ACM SIGARCH Computer Architecture News Homepage archive, Volume 6 Issue 5, December 1977

[Chung09] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, “ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs,” ACM Trans. Reconfigurable Technol. Syst., vol. 2, no. 2, pp. 1–32, 2009.

[cilk1] "Intel Flexes Parallel Programming Muscles", HPCwire (2010-09-02). Retrieved on 2010-09-14.

[cilk2] "Parallel Studio 2011: Now We Know What Happened to Ct, Cilk++, and RapidMind", Dr. Dobbs Journal (2010-09-02). Retrieved on 2010-09-14.

[cilk3] Robert D. Blumofe, Christopher F. Joerg Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. “Cilk: An Efficient Multithreaded Runtime System”, Proceedings of the Fifth ACM SIGPLAN

[Conte 96] T. M. Conte, M. A. Hirsch, and K. N Menezes, “Reducing State Loss for Effective Trace Sampling of Superscalar Processors”, Proceedings of International Conference on Computer Design, 1996, IEEE Computer Society, October 7-9, 1996, Austin, Texas, USA, pp. 468-477

[Cooper09] Peter Cooper, "Beginning Ruby, from Novice to Professional", Second Edition, Apress, ISBN-13:978-1-4302-2363-4, Springer 2009.

[Das94] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, “GTW: A Time Warp System for Shared Memory Multiprocessors,” in WSC ’94: Proceedings of the 26th conference on Winter simulation, 1994, pp. 1332–1339.

[Dave06] N. Dave, M. Pellauer, and J. Emer, “Implementing a functional/timing partitioned microprocessor simulator with an FPGA,” in 2nd Workshop on Architecture Research using FPGA Platforms (WARFP 2006), Feb 2006.

[Dickens93] P. M. Dickens, et al. “A distributed memory lapse: Parallel simulation of message-passing programs.” In Workshop on Parallel and Distributed Simulation, 1993.

[Etsion10] Yoav Etsion, et al. Implementing a Global Shared-Memory in COTSon, TERAFLUX, internal report Nov 2010.

[Flanagan08] David Flanagan, Yukihiro Matsumoto, "The Ruby Programming Language", O' Reilly Media, ISBN -13:978-0-596-51617-8, 2008.

[Frigo98] Matteo Frigo, Charles E. Leiserson, Keith H. Randall, "The implementation of the Cilk-5 multithreaded language", PLDI '98 Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation ACM New York, NY, USA 1998

[Gibeling06] Greg Gibeling, Andrew Schultz, and Krste Asanovic, The RAMP Architecture & Description Language, WARFP, Austin, TX, March 2006

[Giorgi97] R. Giorgi, C.A. Prete, G. Prina, L. Ricciardi, "Trace Factory: Generating Workloads for Trace-Driven Simulation of Shared-Bus Multiprocessors", IEEE Concurrency, ISSN:1092-3063, Los Alamitos, CA, USA, vol. 5, no. 4, Oct. 1997, pp. 54-68, doi 10.1109/4434.641627

[Giorgi08] R. Giorgi, Z. Popovic, N. Puzovic, "Implementing DTA support in CellSim", HiPEAC ACACES-2008, ISBN:978-90-382-1288-3, L'Aquila, Italy, July 2008, pp. 159-162.

[Gligor10] Marius Gligor, Nicolas Fournel and Frédéric Pétrot ,"Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation", Tima France

[Gomperts10] Roberto Gomperts, "SGI Technical Briefing at CESCA Altix UV ", December 2010

[Heinrich02] Joseph Heinrich , "Origin and Ony2 Theory of Operations Manual",007-3439-002 Silicon Graphics.

[Hughes02] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve, "Rsim: Simulating shared-memory multiprocessors with ilp processors," Computer, vol. 35, no. 2, pp. 40–49, 2002.

[Ierusalimschy06] Roberto Ierusalimschy, "Programming in Lua", Lua.org, ISBN 85-903798-2-5, 2006

[Ierusalimschy06b]Roberto Ierusalimschy, Luiz Henriquez de Figueiredo, Waldemar Celes, "Lua Reference Manual 5.1", Lua.org, ISBN 85-903798-3-3, 2006.

[INTEL10] Intel White Paper, Implementing and Expanding a Virtualized Environment, 2010.

[IntelVTX06] Intel Virtualization Technology Volume 10 Issue 03 Published August 10, 2006 ISSN 1535-864X DOI: 10.1535/itj.1003.01

[Jefferson85] D. R. Jefferson, "Virtual time," ACM Transactions on Programming Languages and Systems, vol. 7, no. 3, pp. 404–425, July 1985.

[Kanaujia06] S. Kanaujia, I. E. Papazian, J. Chamberlain, and J. Baxter, "FastMP: A multi-core simulation methodology," in MOBS 2006: Workshop on Modeling, Benchmarking and Simulation, June 2006.

[Keller10] Rainer Keller, David Kramer, Jan-Philipp Weiss, "Facing the Multicore-Challenge, Aspects of New Paradigms and Technologies in Parallel Computing", LNCS 6310, Springer-Verlag Berlin Heidelberg 2010, ISBN 0302-9743.

[KERRIGHED10] Website (December 2010): http://www.kerrighed.org/wiki/index.php/Main_Page

[kimble06] Trina Bunting, Wayne Kimble, "IBM Blade Center JS21 Technical Overview and Introduction, high-performance blade server ideal for extremely dense HPC clusters", IBM.com/redbooks, March 2006.

[Knuth97] Donald Knuth, "The art of computer programming", 1997 Volume 1, 3rd edition, Page 202.

[Krasnov07] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre-Yves Droz, RAMP Blue: A Message-Passing Manycore System In FPGAs, Proceedings of International Conference on Field Programmable Logic and Applications, Amsterdam, The Netherlands, August 2007.

[Kyriacou06] Kyriacou et al, "Data-Driven Multithreading Using Conventional Microprocessors", Parallel and Distributed Systems, IEEE Transactions on Oct. 2006 Volume: 17 Issue:10 On page(s): 1176 - 1188 ISSN: 1045-9219

[L4KA10] Website (December 2010): http://l4ka.org/

[Li09] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, Norman P. Jouppi , "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures", ACM MICRO 41, 2009

[Lugones09] Diego Lugones, Emilio Luque, Daniel Franco, Juan C. Moure, Dolores Rexach, Paolo Fabaroschi, Daniel Ortega, Galo Giménez, Ayose Falcón, "Initial studies of networking Simulation on COTSon", HP-Laboratories HPL-2009-24.

[Lv10] Huiwei Lv, Yuan Cheng, Lu Bai, Mingyu Chen, Dongrui Fan, and Ninghui Sun, "P-GAS: Parallelizing a Cycle-Accurate Event-Driven Many-Core Processor Simulator Using Parallel Discrete Event Simulation", Workshop on Principles of Advanced and Distributed Simulation (PADS), 2010 IEEE

[Macdonell10] Cam Macdonell, Nahami project: A shared memory interface for KVM, http://www.linux-kvm.org/wiki/images/e/e8/0.11.Nahanni-CamMacdonell.pdf, http://gitorious.org/nahanni, December 2010.

[Magnusson02] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," IEEE Computer, vol. 35, no. 2, pp. 50–58, Feb 2002.

[Martin05] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," SIGARCH Comput. Archit. News, vol. 33, no. 4, pp. 92–99, November 2005.

[Mattson10] T. Mattson et al., "The 48-core SCC processor: the programmer's view", SuperComputing conference, Nov. 2010.

[Mattson10b] T. Mattson (interview by Joab Jackson), "Intel: 1000-core Processor Possible", PCWorld, Nov. 2010.

[Mauer02] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pages 108–116, New York, NY, USA, 2002. ACM.

[Mazalan08] Mazalan, L.; Sufian, R.M.R.R.A.; Aziz, A.K.A.; Rohmad, M.S.; Manan, J.-l.A.; "Experience with the implementation of AES256 on L4 microkernel using drops (BID) environment" Information Technology, 2008. ITSim 2008. International Symposium on, Aug. 2008.

[Mazalan08] Mazalan, L.; Sufian, R.M.R.R.A.; Aziz, A.K.A.; Rohmad, M.S.; Manan, J.-l.A.; "Experience with the implementation of AES256 on L4 microkernel using drops (BID) environment" Information Technology, 2008. ITSim 2008. International Symposium on, Aug. 2008.

[Miller10] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep and Anant Agarwal, " Graphite: A Distributed Parallel Simulator for Multicores", HPCA-16, Proceeding of the 16th International Symposium on High-Performance Computer Architecture, January 2010.

[Monchiero09] M. Monchiero, J. H. Ahn, A. Falcon, D. Ortega, and P. Faraboschi, "How to simulate 1000 cores," SIGARCH Comput. Archit. News, vol. 37, no. 2, pp. 10–19, 2009.

[Mukherjee00] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus, "Wisconsin Wind Tunnel II: A fast, portable parallel architecture simulator," IEEE Concurrency, vol. 8, no. 4, pp. 12–20, Oct–Dec 2000.

[MVMPC10] http://www.microsoft.com/windows/virtual-pc/, December 2010.

[Myri10] www.myri.com, October 2010.

[NANOS10] Website (December 2010): http://nanos.ac.upc.edu/content/nanos4

[Noxim09] Noxim: network-on-chip simulator Website: http://sourceforge.net/projects/noxim/

[Penry06] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors, "Exploiting parallelism and structure to accelerate the simulation of chip multi-processors," in HPCA'06: The Twelfth International Symposium on High-Performance Computer Architecture, Feb 2006, pp. 29–40.

[Perelman 03]E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for Accurate and Efficient Simulation", Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, ACM Press, June, 2003, San Diego, California, USA, pp. 318-319

[Perez08] Josep M. Perez, Rosa M. Badia and Jesus Labarta, 2008. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. In Proceedings of IEEE Cluster 2008.

[Prakash98] S. Prakash and R. L. Bagrodia. "Mpi-sim: using parallel simulation to evaluate mpi programs." In WSC '98, 1998.

[Raghav10] Raghav, S.; Ruggiero, M.; Atienza, D.; Pinto, C.; Marongiu, A.; Benini, L.;" Scalable instruction set simulator for thousand-core architectures running on GPGPUs", High Performance Computing and Simulation (HPCS), 2010 International Conference on Issue Date: June 28 2010-July 2 2010 page(s): 459 - 466 , Print ISBN: 978-1-4244-6827-0

[Refsgi] www.sgi.com, October 2010

[Reinhardt93] S. K. Reinhardt, et al. "The Wisconsin wind tunnel: Virtual prototyping of parallel computers." Sigmetrics Conference on Measurement and Modeling of Computer Systems, '93.

[Renau05] J. Renau, et al. "SESC simulator," 2005. Available online at: http://sesc.sourceforge.net.

[Rosenblum95] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: The SimOS approach," IEEE Parallel & Distributed Technology: Systems & Applications, vol. 3, no. 4, pp. 34–43, Winter 1995.

[SIMNow09] AMD SimNow™ Simulator 4.6.1 User's Manual Revision Date 2.13 November 2009

[Spm10] SuperMicro, "H8QG6+-F H8QGi+-F USER'S MANUAL Revision 1.0", www.supermicro.com, 2010

[Teruel07] Teruel, X. Martorell, A. Duran, R. Ferrer, and E. Ayguadé, "Support for OpenMP tasks in Nanos v4", Proceedings of the 2007 conference of the Centre for Advanced Studies on Collaborative Research: IBM, pp. 256-259, October, 2007.

[top500] www.top500.org, October 2010.

[UNISIM10] http://unisim.org, December 2010.

[Vantrease08] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. P. Jouppi, M. Fiorentino, A. Davis, N. Binkert, R. G. Beausoleil, and J. H. Ahn. Corona: System implications of emerging nanophotonic technology. In ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture.

[VirtualBox10] http://www.virtualbox.org/, December 2010.

[VMWare10] http://www.vmware.com/, December 2010.

[Wang11] Zhaoguo Wang , Ran Liu, Yufei Chen, Xi Wu Fudan, Haibo Chen, Weihua Zhang, Binyu Zang, "COREMU: a scalable and portable parallel full-system emulator", accepted for publication in PPoPP '11 Proceedings of the 16th ACM symposium on Principles and practice of parallel programming.

[Wenisch06] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical sampling of computer system simulation," IEEE Micro, vol. 26, no. 4, pp. 18–31, July-Aug 2006.

[Wunderlish03] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SAMRTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling", Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03), IEEE Computer Society, June 9-11, 2003, San Diego, USA, pp. 84-95

[Wunderlich04] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "An Evaluation of Stratified Sampling of Microarchitecture Simulations", Proceedings of the Third Annual Workshop on Duplicating, Deconstructing, and Debunking, IEEE Computer Society, June 19-23, 2004, Munchen, Germany, pp. 13-18

[Xen99] Xen community overview. http://xensource.com/xen

[Zeng09] Hui Zeng, Matt Yourst, Kanad Ghose and Dimitry Ponomarev, "MPTSim: A Cycle-Accurate, Full-System Simulator for x86-64 Multicore Architectures with Coherent Caches", ACM SIGARCH Computer Architecture News, May 2009, volume 37, issue 2.

[Zheng04] G. Zheng, G. Kakulapati, and L. V. Kal´e, "BigSim: A parallel simulator for performance prediction of extremely large parallel machines," in 18th International Parallel and Distributed Processing Symposium (IPDPS), Apr 2004, p. 78.

[Zhibin 09] Zhibin Yu, Hai Jin, Jian Chen, and John, L. K, "TSS: Applying two-stage sampling in micro-architecture simulations", in proceedings of IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009 (MASCOTS2009). pp. 1-9

[Zhibin 10] Zhibin Yu, Hai Jin, Jian Chen, and John, L. K, "CantorSim: Simplifying Acceleration of Micro-architecture Simulations", in proceedings of IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2010 (MASCOTS2010). pp. 370-377

Appendix 1 – QEMU-DTA output

Output of the SIMULATION of the Fibonacci (4) program, showing TSU and x86-64 ISA EXTENSIONS at work:

```
TCREATE(000000000040066f, 1, 0000000000000000) =
0
TCREATE(00000000004004e3, 3, 0000000000000000) =
1
TWRITE(0000000000000001, 0000000000000002,
0000000000000004)
TWRITE(0000000000000001, 0000000000000004,
0000000000000000)
TWRITE(0000000000000001, 0000000000000006,
0000000000000002)
EIP==0000000000400474
EIP==0000000000400474
EIP==00000000004004e2
TDESTROY(1048576) -- but still 0 are running
TSU Scheduling: dispatching continuation #1
switch_to: context switch from cont #1048576 to
cont #1
__switch_to: prevoius EIP = 00000000004004e2
__switch_to: new EIP = 00000000004004e3
EIP==00000000004004e3
TREAD(0000000000000000, 0000000000000002)
TReading EAX=4
TREAD(0000000000000004, 0000000000000004)
TReading EAX=0
TREAD(0000000000000000, 0000000000000006)
TReading EAX=2
EIP==00000000004004e3
EIP==00000000004004e3
EIP==0000000000400534
TCREATE_C()
TSU Scheduling: dispatching continuation #1
switch_to: context switch from cont #1 to cont #1
__switch_to: prevoius EIP = 0000000000400551
__switch_to: new EIP = 0000000000400551
EIP==0000000000400551
TCREATE(000000000040062f, 4, 0000000000000000) =
2
TCREATE(00000000004004e3, 3, 0000000000000000) =
3
TWRITE(0000000000000003, 0000000000000002,
0000000000000003)
TWRITE(0000000000000003, 0000000000000004,
0000000000000002)
TWRITE(0000000000000003, 0000000000000006,
0000000000000006)
TCREATE(00000000004004e3, 3, 0000000000000000) =
4
TWRITE(0000000000000004, 0000000000000002,
0000000000000002)
TWRITE(0000000000000004, 0000000000000004,
0000000000000002)
TWRITE(0000000000000004, 0000000000000006,
0000000000000008)
TWRITE(0000000000000002, 0000000000000002,
0000000000000000)
TWRITE(0000000000000002, 0000000000000004,
0000000000000002)
EIP==0000000000400551
EIP==0000000000400551
EIP==000000000040062f
TDESTROY(1) -- but still 2 are running
deleting continuation #1
TSU Scheduling: dispatching continuation #3
switch_to: context switch from cont #1 to cont #3
__switch_to: prevoius EIP = 000000000040062f
__switch_to: new EIP = 00000000004004e3
EIP==00000000004004e3
TREAD(0000000000000000, 0000000000000002)
TReading EAX=3
TREAD(0000000000000003, 0000000000000004)
TReading EAX=2
TREAD(0000000000000002, 0000000000000006)
TReading EAX=6
EIP==00000000004004e3
EIP==00000000004004e3
EIP==0000000000400534
TCREATE_C()
TSU Scheduling: dispatching continuation #4
switch_to: context switch from cont #3 to cont #4
__switch_to: prevoius EIP = 0000000000400551
__switch_to: new EIP = 00000000004004e3
EIP==00000000004004e3
TREAD(0000000000000000, 0000000000000002)
TReading EAX=2
TREAD(0000000000000002, 0000000000000004)
TReading EAX=2
TREAD(0000000000000002, 0000000000000006)
TReading EAX=8
EIP==00000000004004e3
EIP==00000000004004e3
EIP==0000000000400534
TCREATE_C()
TSU Scheduling: dispatching continuation #3
switch_to: context switch from cont #4 to cont #3
__switch_to: prevoius EIP = 0000000000400551
__switch_to: new EIP = 0000000000400551
EIP==0000000000400551
TCREATE(000000000040062f, 4, 0000000000000000) =
1
TCREATE(00000000004004e3, 3, 0000000000000000) =
5
TWRITE(0000000000000005, 0000000000000002,
0000000000000002)
TWRITE(0000000000000005, 0000000000000004,
0000000000000001)
TWRITE(0000000000000005, 0000000000000006,
0000000000000006)
TCREATE(00000000004004e3, 3, 0000000000000000) =
6
TWRITE(0000000000000006, 0000000000000002,
0000000000000001)
TWRITE(0000000000000006, 0000000000000004,
0000000000000001)
TWRITE(0000000000000006, 0000000000000006,
0000000000000008)
TWRITE(0000000000000001, 0000000000000002,
0000000000000002)
TWRITE(0000000000000001, 0000000000000004,
0000000000000006)
EIP==0000000000400551
EIP==0000000000400551
EIP==000000000040062f
TDESTROY(3) -- but still 3 are running
deleting continuation #3
TSU Scheduling: dispatching continuation #4
switch_to: context switch from cont #3 to cont #4
__switch_to: prevoius EIP = 000000000040062f
__switch_to: new EIP = 0000000000400551
EIP==0000000000400551
TCREATE(000000000040062f, 4, 0000000000000000) =
3
TCREATE(00000000004004e3, 3, 0000000000000000) =
7
TWRITE(0000000000000007, 0000000000000002,
0000000000000001)
TWRITE(0000000000000007, 0000000000000004,
0000000000000003)
TWRITE(0000000000000007, 0000000000000006,
0000000000000006)
TCREATE(00000000004004e3, 3, 0000000000000000) =
8
TWRITE(0000000000000008, 0000000000000002,
0000000000000000)
TWRITE(0000000000000008, 0000000000000004,
0000000000000003)
TWRITE(0000000000000008, 0000000000000006,
0000000000000008)
TWRITE(0000000000000003, 0000000000000002,
0000000000000002)
TWRITE(0000000000000003, 0000000000000004,
0000000000000008)
EIP==0000000000400551
```

```
EIP==0000000000400551
EIP==000000000040062f
TDESTROY(4) -- but still 4 are running
deleting continuation #4
TSU Scheduling: dispatching continuation #5
switch_to: context switch from cont #4 to cont #5
__switch_to: prevoius EIP = 000000000040062f
__switch_to: new EIP = 00000000004004e3
EIP==00000000004004e3
TREAD(0000000000000000, 0000000000000002)
TReading EAX=2
TREAD(0000000000000002, 0000000000000004)
TReading EAX=1
TREAD(0000000000000001, 0000000000000006)
TReading EAX=6
EIP==00000000004004e3
EIP==00000000004004e3
EIP==0000000000400534
TCREATE_C()
TSU Scheduling: dispatching continuation #6
switch_to: context switch from cont #5 to cont #6
__switch_to: prevoius EIP = 0000000000400551
__switch_to: new EIP = 00000000004004e3
EIP==00000000004004e3
TREAD(0000000000000000, 0000000000000002)
TReading EAX=1
TREAD(0000000000000001, 0000000000000004)
TReading EAX=1
TREAD(0000000000000001, 0000000000000006)
TReading EAX=8
EIP==00000000004004e3
EIP==00000000004004e3
EIP==0000000000400534
TCREATE_C()
EIP==0000000000400534
EIP==0000000000400534
EIP==0000000000400534
EIP==0000000000400540
TCREATE_C()
TSU Scheduling: dispatching continuation #7
switch_to: context switch from cont #6 to cont #7
__switch_to: prevoius EIP = 0000000000400540
__switch_to: new EIP = 00000000004004e3
EIP==00000000004004e3
TREAD(0000000000000000, 0000000000000002)
TReading EAX=1
TREAD(0000000000000001, 0000000000000004)
TReading EAX=3
TREAD(0000000000000003, 0000000000000006)
TReading EAX=6
EIP==00000000004004e3
EIP==00000000004004e3
EIP==0000000000400534
TCREATE_C()
EIP==0000000000400534
EIP==0000000000400534
EIP==0000000000400534
EIP==0000000000400540
TCREATE_C()
TSU Scheduling: dispatching continuation #8
switch_to: context switch from cont #7 to cont #8
__switch_to: prevoius EIP = 0000000000400540
__switch_to: new EIP = 00000000004004e3
EIP==00000000004004e3
TREAD(0000000000000000, 0000000000000002)
TReading EAX=0
TREAD(0000000000000000, 0000000000000004)
TReading EAX=3
TREAD(0000000000000003, 0000000000000006)
TReading EAX=8
EIP==00000000004004e3
EIP==00000000004004e3
EIP==0000000000400534
TCREATE_C()
EIP==0000000000400534
EIP==0000000000400534
EIP==0000000000400534
EIP==0000000000400540
TCREATE_C()
TSU Scheduling: dispatching continuation #5
switch_to: context switch from cont #8 to cont #5
__switch_to: prevoius EIP = 0000000000400540
__switch_to: new EIP = 0000000000400551
EIP==0000000000400551
TCREATE(000000000040062f, 4, 0000000000000000) =
4
TCREATE(00000000004004e3, 3, 0000000000000000) =
9
TWRITE(0000000000000009, 0000000000000002,
0000000000000001)
TWRITE(0000000000000009, 0000000000000004,
0000000000000004)
TWRITE(0000000000000009, 0000000000000006,
0000000000000006)
TCREATE(00000000004004e3, 3, 0000000000000000) =
10
TWRITE(000000000000000a, 0000000000000002,
0000000000000000)
TWRITE(000000000000000a, 0000000000000004,
0000000000000004)
TWRITE(000000000000000a, 0000000000000006,
0000000000000008)
TWRITE(0000000000000004, 0000000000000002,
0000000000000001)
TWRITE(0000000000000004, 0000000000000004,
0000000000000006)
EIP==0000000000400551
EIP==0000000000400551
EIP==000000000040062f
TDESTROY(5) -- but still 5 are running
deleting continuation #5
TSU Scheduling: dispatching continuation #6
switch_to: context switch from cont #5 to cont #6
__switch_to: prevoius EIP = 000000000040062f
__switch_to: new EIP = 0000000000400540
EIP==0000000000400540
TWRITE(0000000000000001, 0000000000000008,
0000000000000001)
EIP==0000000000400540
EIP==0000000000400540
EIP==0000000000400551
TDESTROY(6) -- but still 4 are running
deleting continuation #6
TSU Scheduling: dispatching continuation #7
switch_to: context switch from cont #6 to cont #7
__switch_to: prevoius EIP = 0000000000400551
__switch_to: new EIP = 0000000000400540
EIP==0000000000400540
TWRITE(0000000000000003, 0000000000000006,
0000000000000001)
EIP==0000000000400540
EIP==0000000000400540
EIP==0000000000400551
TDESTROY(7) -- but still 3 are running
deleting continuation #7
TSU Scheduling: dispatching continuation #8
switch_to: context switch from cont #7 to cont #8
__switch_to: prevoius EIP = 0000000000400551
__switch_to: new EIP = 0000000000400540
EIP==0000000000400540
TWRITE(0000000000000003, 0000000000000008,
0000000000000001)
EIP==0000000000400540
EIP==0000000000400540
EIP==0000000000400551
TDESTROY(8) -- but still 3 are running
deleting continuation #8
TSU Scheduling: dispatching continuation #9
switch_to: context switch from cont #8 to cont #9
__switch_to: prevoius EIP = 0000000000400551
__switch_to: new EIP = 00000000004004e3
EIP==00000000004004e3
TREAD(0000000000000000, 0000000000000002)
TReading EAX=1
TREAD(0000000000000001, 0000000000000004)
TReading EAX=4
TREAD(0000000000000004, 0000000000000006)
TReading EAX=6
EIP==00000000004004e3
EIP==00000000004004e3
EIP==0000000000400534
TCREATE_C()
EIP==0000000000400534
EIP==0000000000400534
```

```
EIP==0000000000400534
EIP==0000000000400540
TCREATE_C()
TSU Scheduling: dispatching continuation #10
switch_to: context switch from cont #9 to cont
#10
__switch_to: prevoius EIP = 0000000000400540
__switch_to: new EIP = 00000000004004e3
EIP==00000000004004e3
TREAD(0000000000000000, 0000000000000002)
TReading EAX=0
TREAD(0000000000000000, 0000000000000004)
TReading EAX=4
TREAD(0000000000000004, 0000000000000006)
TReading EAX=8
EIP==00000000004004e3
EIP==00000000004004e3
EIP==0000000000400534
TCREATE_C()
EIP==0000000000400534
EIP==0000000000400534
EIP==0000000000400534
EIP==0000000000400540
TCREATE_C()
TSU Scheduling: dispatching continuation #3
switch_to: context switch from cont #10 to cont
#3
__switch_to: prevoius EIP = 0000000000400540
__switch_to: new EIP = 000000000040062f
EIP==000000000040062f
TREAD(0000000000000000, 0000000000000002)
TReading EAX=2
TREAD(0000000000000002, 0000000000000004)
TReading EAX=8
TREAD(0000000000000008, 0000000000000006)
TReading EAX=1
TREAD(0000000000000001, 0000000000000008)
TReading EAX=1
TWRITE(0000000000000002, 0000000000000008,
0000000000000002)
EIP==000000000040062f
EIP==000000000040062f
EIP==000000000040066f
TDESTROY(3) -- but still 2 are running
deleting continuation #3
TSU Scheduling: dispatching continuation #9
switch_to: context switch from cont #3 to cont #9
__switch_to: prevoius EIP = 000000000040066f
__switch_to: new EIP = 0000000000400540
EIP==0000000000400540
TWRITE(0000000000000004, 0000000000000006,
0000000000000001)
EIP==0000000000400540
EIP==0000000000400540
EIP==0000000000400551
TDESTROY(9) -- but still 1 are running
deleting continuation #9
TSU Scheduling: dispatching continuation #10
switch_to: context switch from cont #9 to cont
#10
__switch_to: prevoius EIP = 0000000000400551
__switch_to: new EIP = 0000000000400540
EIP==0000000000400540
TWRITE(0000000000000004, 0000000000000008,
0000000000000001)
EIP==0000000000400540
EIP==0000000000400540
EIP==0000000000400551
TDESTROY(10) -- but still 1 are running
deleting continuation #10
TSU Scheduling: dispatching continuation #4
switch_to: context switch from cont #10 to cont
#4
__switch_to: prevoius EIP = 0000000000400551
__switch_to: new EIP = 000000000040062f
EIP==000000000040062f
TREAD(0000000000000000, 0000000000000002)
TReading EAX=1
TREAD(0000000000000001, 0000000000000004)
TReading EAX=6
TREAD(0000000000000006, 0000000000000006)
TReading EAX=1
TREAD(0000000000000001, 0000000000000008)
TReading EAX=1
TWRITE(0000000000000001, 0000000000000006,
0000000000000002)
EIP==000000000040062f
EIP==000000000040062f
EIP==000000000040066f
TDESTROY(4) -- but still 1 are running
deleting continuation #4
TSU Scheduling: dispatching continuation #1
switch_to: context switch from cont #4 to cont #1
__switch_to: prevoius EIP = 000000000040066f
__switch_to: new EIP = 000000000040062f
EIP==000000000040062f
TREAD(0000000000000000, 0000000000000002)
TReading EAX=2
TREAD(0000000000000002, 0000000000000004)
TReading EAX=6
TREAD(0000000000000006, 0000000000000006)
TReading EAX=2
TREAD(0000000000000002, 0000000000000008)
TReading EAX=1
TWRITE(0000000000000002, 0000000000000006,
0000000000000003)
EIP==000000000040062f
EIP==000000000040062f
EIP==000000000040066f
TDESTROY(1) -- but still 1 are running
deleting continuation #1
TSU Scheduling: dispatching continuation #2
switch_to: context switch from cont #1 to cont #2
__switch_to: prevoius EIP = 000000000040066f
__switch_to: new EIP = 000000000040062f
EIP==000000000040062f
TREAD(0000000000000000, 0000000000000002)
TReading EAX=0
TREAD(0000000000000000, 0000000000000004)
TReading EAX=2
TREAD(0000000000000002, 0000000000000006)
TReading EAX=3
TREAD(0000000000000003, 0000000000000008)
TReading EAX=2
TWRITE(0000000000000000, 0000000000000002,
0000000000000005)
EIP==000000000040062f
EIP==000000000040062f
EIP==000000000040066f
TDESTROY(2) -- but still 1 are running
deleting continuation #2
TSU Scheduling: dispatching continuation #0
switch_to: context switch from cont #2 to cont #0
__switch_to: prevoius EIP = 000000000040066f
__switch_to: new EIP = 000000000040066f
EIP==000000000040066f
TREAD(0000000000000000, 0000000000000002)
TReading EAX=5          //Result of fib(4) is 5
EIP==000000000040066f
EIP==000000000040066f
EIP==000000000040067d
TDESTROY(0) -- but still 0 are running
deleting continuation #0
switch_to: context switch from cont #0 to cont
#1048576
__switch_to: previous EIP = 000000000040067d
== TCREATE:                  15
== TREAD:                    44
== TWRITE:                   44
== TDESTROY:                 15
```

Appendix 2 - Implementation of the functional simulative model "Sharable Memory"

The functional simulation is provided by the ***libtf-model*** library, which is used by the simulation frontends (in this case, *COTSon*).

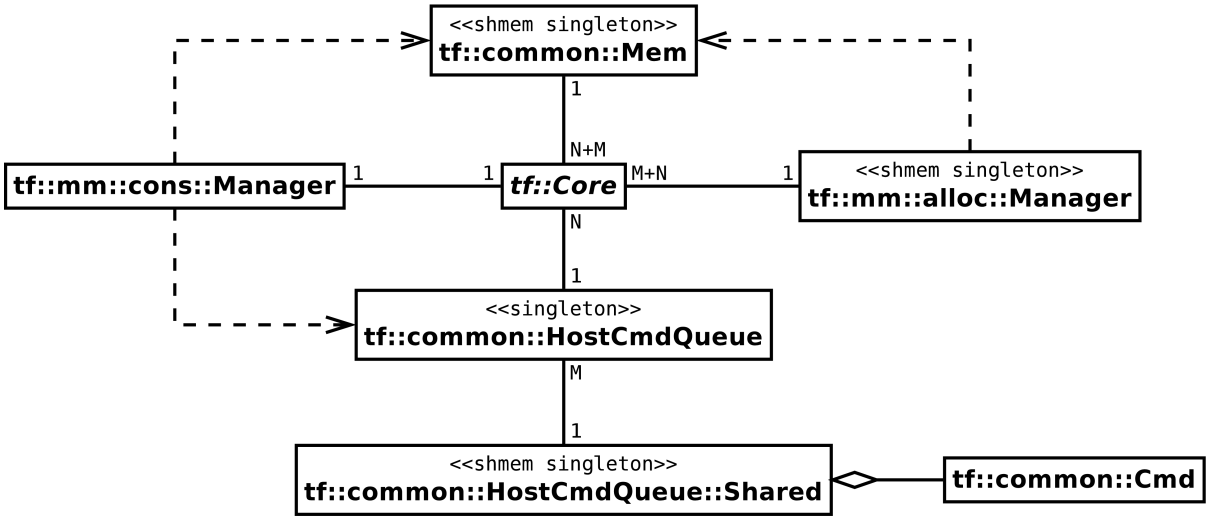


Figure 35 - Central Abstraction of the functional simulation model of the Shareable Memory.

This model encompasses the abstractions found in Figure 35, which show a simplified diagram of the central pieces implementing the functional simulation.

Each simulator instance contains a group of core (***tf::Core***) instances that represent one of the nodes in the TERAFLUX system. These cores provide an interface to the implementation of the ISA extensions.

Each SimNow instance contains a group of core (***tf::Core*** instances) that may represent one of the nodes in the TERAFLUX system (this is true in the TBM – TERAFLUX Baseline Machine – but may change for the final TERAFLUX system). These cores provide an interface to the implementation of the ISA extensions.

Each of these extensions can follow two implementation approaches. On one hand, it can be fully implemented in shared memory (identified by the *shmem singleton* keyword), like the centralized memory allocation manager. This is all provided through memory-mapped files, which can also contain the necessary inter-process mutexes to allow synchronization among different simulator instances.

Note that the memory allocation manager is centralized, as its operation will be seldom triggered, and thus there is not a need for a more complex guest-only implementation.

On the other hand, classes can use the more complex *command queue* abstraction. Using the command queue, subjects can put and get serializable commands (instances of ***tf::common::Cmd***) that are broadcasted to all command queue users.

This command queue is implemented as a ring buffer residing in the (host) shared memory (***tf::common::HostCmdQueue::Shared***), and a local interface (***tf:common::HostCmdQueue***). Whenever all local interfaces have consumed a command from the queue, its slot is marked as free for future reuse. Thus, the command queue is a broadcasting channel, with as many receivers as local interfaces have been instantiated (which usually amounts for one local interface per node).

The queue interface has three basic methods:

- ***get*** : Gets unread commands from the queue. This is the simplest method, as it simply consumes commands from the queue, and leaves its processing up to the user. This is used, for example, by ***tf::Core*** instances themselves before trying to perform a ***put***, as this method is optimized to have low contention on the command queue.

- ***put*** : Puts a new command into the list if its processing is successful. The processing of a command (whose implementation is command-specific) can return a negative result, in which case this indicates that the command was unsuccessful (e.g., a failed acquire) and therefore must not be put into the queue.
- ***get_and_put*** : Atomically gets and processes commands and puts a new one into the queue. In some cases, a command cannot be broadcasted nor even processed before atomically processing all pending commands of related types. This is the case of memory consistency management commands, which must be atomically performed from the point of view of the TERAFLUX system. To this effect, it atomically gets all pending commands, processing those of some specific types (e.g., all memory consistency management commands), and puts the new command into the queue. If any of the command processing is unsuccessful (both the read and put commands), the command id is not put into the queue (e.g., a failed acquire is not broadcasted to all other nodes).

Note that this command queue implementation only works on a host with shared memory, and therefore is valid only for simulation instances running on the same host. For the time being we are not exploring simulations spanning multiple hosts, as this would require much additional effort in order to find efficient mechanism to get rid of TCP/IP+copying as already noted (and, e.g., an extra process on each simulation host, which would forward commands among the different per-host command queues).

Finally, the ***tf::mm::cons::Manager*** instances, direct the simulation frontend to perform a memory copy from the ***tf::common::Mem*** into the simulated (guest) memory in the case of an acquire, and a reverse copy in the case of a release. As mentioned before, this is optimized to avoid copies when transferring memory ownership among cores inside the same SimNow instance.