FRAMEWORK PROGRAMME
ICT-01-2014: Smart Cyber-Physical Systems

PROJECT NUMBER: 645496

Agile, eXtensible, fast I/O Module for the cyber-physical era

D5.4 – Final operating system and documentation

Due date of deliverable: 31st January 2018
Actual Submission: 14th February 2018 (agreed extended date)

Start date of the project: 1st February 2015 Duration: 36 months

Lead contractor for the deliverable: EVI

Revision: See file name in document footer.

Project co-founded by the European Commission within the HORIZON FRAMEWORK PROGRAMME (2020)	
Dissemination Level: PU	
PU	Public
PP	Restricted to other programs participant (including the Commission Services)
RE	Restricted to a group specified by the consortium (including the Commission Services)
CO	Confidential, only for members of the consortium (including the Commission Services)

Change Control

Version#	**Date**	**Author**	**Organiz.**	**Change History**
0.1	05.01.2018	Paolo Gai, Stefano Garzarella	EVI	Initial version starting from D5.3
0.2	18.01.2018	Paolo Gai, Stefano Garzarella	EVI	Added real-time section
0.3	30.01.2018	Paolo Gai, Stefano Garzarella	EVI	Updated real-time section
0.4	08.02.2018	Paolo Gai, Stefano Garzarella, Xavier Martorell, Daniel Jiménez-González, Carlos Álvarez	EVI/BSC	Update load balancing and AXIOM SW Stack sections
0.5	09.02.2018	Roberto Giorgi	UNISI	DF-Threads
0.6	13.02.2018	Paolo Gai	EVI	Integration of reviewers' comments

Release Approval

Name	**Role**	**Date**
Paolo Gai	WP Leader	09.02.2018
Roberto Giorgi	Project Coordinator for formal deliverable	14.02.2018

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

Paolo Gai, Stefano Garzarella, Bruno Morelli
R&D Department, Evidence

Xavier Martorell, Daniel Jiménez-González, Carlos Álvarez
CS Departament
Barcelona Supercomputing Center (BSC) - AXIOM
Universitat Politècnica de Catalunya - AXIOM

Marco Procaccini, Farnam Khalili, Roberto Giorgi
University of Siena, Italy

© 2015-2018 AXIOM Consortium, All Rights Reserved.
Document marked as PU (Public) is published in Italy, for the AXIOM Consortium, on the www.AXIOM-project.eu web site and can be distributed to the Public.
All other trademarks and copyrights are the property of their respective owners. The list of author does not imply any claim of ownership on the Intellectual Properties described in this document.
The authors and the publishers make no expressed or implied warranty of any kind and assume no responsibilities for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information contained in this document.
This document is furnished under the terms of the AXIOM License Agreement (the "License") and may only be used or copied in accordance with the terms of the License. The information in this document is a work in progress, jointly developed by the members of AXIOM Consortium ("AXIOM") and is provided for informational use only.
The technology disclosed herein may be protected by one or more patents, copyrights, trademarks and/or trade secrets owned by or licensed to AXIOM Partners. The partners reserve all rights with respect to such technology and related materials. Any use of the protected technology and related material beyond the terms of the License without the prior written consent of AXIOM is prohibited. This document contains material that is confidential to AXIOM and its members and licensors. Until publication, the user should assume that all materials contained and/or referenced in this document are confidential and proprietary unless otherwise indicated or apparent from the nature of such materials (for example, references to publicly available forms or documents).
Disclosure or use of this document or any material contained herein, other than as expressly permitted, is prohibited without the prior written consent of AXIOM or such other party that may grant permission to use its proprietary material. The trademarks, logos, and service marks displayed in this document are the registered and unregistered trademarks of AXIOM, its members and its licensors. The copyright and trademarks owned by AXIOM, whether registered or unregistered, may not be used in connection with any product or service that is not owned, approved or distributed by AXIOM, and may not be used in any manner that is likely to cause customer confusion or that disparages AXIOM. Nothing contained in this document should be construed as granting by implication, estoppel, or otherwise, any license or right to use any copyright without the express written consent of AXIOM, its licensors or a third party owner of any such trademark.
Printed in Siena, Italy, Europe.
Part number: *Please refer to the File name in the document footer.*

EXCEPT AS OTHERWISE EXPRESSLY PROVIDED, THE AXIOM SPECIFICATION IS PROVIDED BY AXIOM TO MEMBERS "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS.
AXIOM SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES OF ANY KIND OR NATURE WHATSOEVER (INCLUDING, WITHOUT LIMITATION, ANY DAMAGES ARISING FROM LOSS OF USE OR LOST BUSINESS, REVENUE, PROFITS, DATA OR GOODWILL) ARISING IN CONNECTION WITH ANY INFRINGEMENT CLAIMS BY THIRD PARTIES OR THE SPECIFICATION, WHETHER IN AN ACTION IN CONTRACT, TORT, STRICT LIABILITY, NEGLIGENCE, OR ANY OTHER THEORY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TABLE OF CONTENTS

1 Executive summary 6
2 Introduction 7
2.1 Document structure 7
2.2 Relation to other deliverables 7
2.3 Tasks involved in this deliverable 8
2.4 Changes compared to D5.3 8
3 The AXIOM cluster and the AXIOM software stack 9
4 Components of the software stack 10
4.1 AXIOM Drivers, User libraries and applications 10
4.2 GASNet conduit 12
4.3 OmpSs framework 12
5 Using the AXIOM software stack 12
6 Mechanisms for load balancing 14
6.1 Improving load balancing with OmpSs@FPGA 14
6.2 Load balancing by dataflow-thread based execution 15
7 Analysis of the real-time guarantees 18
8 Archives released 22
9 Confirmation of DoA objectives and Conclusions 22
10 References 23
11 Appendix 1 25
1.1 Communication between XSMLL Hardware and Software 25
1.2 Packet format between XSMLL and NIC and timing 31

TABLE OF FIGURES

FIGURE 1 - THE AXIOM SOFTWARE STACK. 10
FIGURE 2 - EXECUTION TRACE OF MATRIX MULTIPLICATION USING 3 SMP CORES AND 1 FPGA IP CORE. 14
FIGURE 3 - EXECUTION OF MATRIX MULTIPLICATION WITH THE HYBRID WORKER AND SHOWING THE INSTRUMENTATION INTERNALS. 15
FIGURE 4 – XSMLL HARDWARE/SOFTWARE ARCHITECTURE. THERE ARE N NODES AND M CORES PER NODE. XTQ ARE THE XSMLL THREAD QUEUES AND XNM IS THE XSMLL NODE MANAGER THAT IMPLEMENTS SEVERAL FINITE STATE MACHINES. 15
FIGURE 5 – XSMLL MESSAGE PROTOCOL. 16
FIGURE 6 – XSMLL MESSAGE PROTOCOL. C=CORE, XTQ= XSMLL THREAD QUEUE, XWT=XSMLL WAITING TABLE, XFFT=XSMLL FREE-FRAME TABLE, XPTQ=XSMLL PENDING THREAD-REQUEST QUEUE. 17
FIGURE 7 – COMPARISON AMONG XSMLL EXECUTION TIME (SECONDS) AND OPENMPI, JUMP PROGRAMMING MODEL EXECUTION. THE BENCHMARK IS MATRIX MULTIPLICATION WITH SQUARE MATRIX SIZE OF 216, 432, 864 AND BLOCK SIZE OF 8. THE NODES ARE CONFIGURED WITH ONLY 1 CORE EACH. 17
FIGURE 8 – COMPARISON AMONG XSMLL L2-CACHE MISS RATE AND OPENMPI, JUMP PROGRAMMING MODEL L2 MISS-RATE. THE BENCHMARK IS MATRIX MULTIPLICATION WITH SQUARE MATRIX SIZE OF 216, 432, 864 AND BLOCK SIZE OF 8. THE NODES ARE CONFIGURED WITH ONLY 1 CORE EACH. 17
FIGURE 9 - IRQS ARE TYPICALLY SCHEDULED ON CORE 0, INTERRUPTING AND SLOWING DOWN COMPUTATIONAL THREADS. 19
FIGURE 10 - UNDER SCHED_DEADLINE, A TASK CONSUMING TOO MUCH CPU IS THROTTLED. 19
FIGURE 11 - EXECUTION TIME OF A MATRIX MULTIPLY VARYING THE SCHED_DEADLINE BANDWIDTH PARAMETER WITH A FIXED PERIOD (100 MS) FOR THE COMMUNICATION THREAD. 21
FIGURE 12 - EXECUTION TIME OF A MATRIX MULTIPLY VARYING THE SCHED_DEADLINE PERIOD PARAMETER WITH A FIXED CPU BANDWIDTH (5%) FOR THE COMMUNICATION THREAD. 22

GLOSSARY

API – Application Programming Interface

ARM – Instruction set architecture developed by ARM Holdings Ltd.
BLAS – Basic linear algebra subprograms
FPGA – Field Programmable Gate Array
IP – Intellectual property
LTS – Long Term Support
Mali – A GPU microarchitecture developed by ARM Holdings Ltd.
Mercurium – OmpSs compiler
Nanos++ – OmpSs runtime
NDA – Non-disclosure agreement
NEON – SIMD extensions for the ARM instruction set
NIC – Network interface
PL – Programmable logic
QEMU – Quick EMUlator
RTL – Register transfer language
RTSP – Real-time streaming protocol
SD – Secure Digital
SDK – Software Development Kit
SDSM – Software Distributed Shared Memory
SGEMM – Single-precision floating-point general matrix multiply
SIMD – Single instruction, multiple data
SMP – Symmetric multiprocessing
SMT – Simultaneous multithreading
SoC – System on chip

1 Executive summary

This document is an update of deliverable D5.3, and includes all the software developed in WP5, plus the results of task T5.4 (mechanism for load balancing and analysis of real-time guarantees).

The software is released both as source code and as precompiled Debian packages. It includes a complete software stack starting from the AXIOM Linux drivers and arriving to the OmpSs@Cluster programming library.

All this software has been released on the project public website:

https://download.axiom-project.eu/?dir=RUNTIME

2 Introduction

The content of this document is a set of "pointers" to a large quantity of public material, repositories, documentation and videos developed within the context of the AXIOM project. It also offers a brief guide that enables external users to re-use (for research, study or further development) the AXIOM stack at its final development stage. The results of D5.4 represents also the completion of the efforts initiated in the Task T5.4.

In the initial phase of the development of the software stack, as we described in the D5.3, we preferred to test the functionality of it on a QEMU based platform (in line with the Xilinx recommendations and related tools). Afterwards, when the AXIOM NIC implementation on the FPGA was available, we moved all the development on the AXIOM board (the QEMU support was no more updated).

This document reflects the adaptation of the stack to the real board, and is therefore a final guide to use the AXIOM Software Stack on the AXIOM board.

2.1 Document structure

Section 3 provides a short description of the AXIOM architecture, useful to understand the various subsystems. Section 4 includes a description of the various subsystems, providing their location and documentation pointers. Section 5 provides a short guide on how to use the AXIOM software stack. Finally, Section 6 and 7 describes the mechanisms for load balancing and the analysis of the real time guarantees. Section 8 includes a list of all the files object of this release.

2.2 Relation to other deliverables

This document is linked to the following internal deliverables of the AXIOM project:

D5.1 Operating System and Documentation

> This document describes in detail the implementation of the AXIOM Linux distribution for the AXIOM board, and the AXIOM NIC interface.

D5.2 Remote Memory Access

> This document describes the cluster setup, memory organization, memory allocator, and task synchronization. It also includes details on the implementation of the Parallel Programming library.

D5.3 Parallel Programming Library and Documentation

> This document is a short guide of the software release that is part of Task T5.3. The software release includes a complete software stack starting from the AXIOM Linux drivers up to the OmpSs@Cluster programming library.

D4.2 AXIOM Code Generation and Instrumentation

> This document describes the OmpSs compilation and FPGA support as well as the instrumentation mechanism present in OmpSs.

D4.3 Evaluation of the Compiler and Tools Infrastructure.

> After the first prototype AXIOM board will be delivered, the testing will be necessary to verify the initial software toolchain. Final research results will be included in this deliverable.

2.3 Tasks involved in this deliverable

Task 5.4 (month 25 - 36): Load balancing and real-time guarantees

What:

This task will deal with the design and development of a Nanos++ scheduling policy taking into consideration task data affinity and load balancing across the nodes.

How:

The work will be done by partner BSC with help from partner EVI for integration. The possibility of providing real-time guarantees about latencies in both communication and processing will be investigated by partner EVI. UNISI will investigate different load-balancing techniques.

Expected output:

- Version of Nanos++ for the reference platform containing load balancing mechanisms.

- Analysis of real-time latencies.

D5.4: Final operating system and documentation [M36]

This deliverable will consist in an update of deliverable D5.3 containing also the mechanism for load balancing. An additional document will provide the analysis of the real-time guarantees that the system can meet.

2.4 Changes compared to D5.3

This deliverable contains some modifications and new contents compared to D5.3:

- The final release of AXIOM software stack runs on the AXIOM board and supports the AXIOM NIC developed on the Xilinx Zynq FPGA as part of the AXIOM project. We also replaced the buildroot filesystem and toolchain with the Ubuntu 16.04 filesystem and the Linaro toolchain.
- This document also contains the results of Task T5.4 about the possibility of providing real-time guarantees and the mechanism for load balancing.

3 The AXIOM cluster and the AXIOM software stack

The AXIOM system is composed by a set of computing boards connected together to form a cluster configuration called “AXIOM-cluster”. The connection between boards is implemented through the AXIOM-link, which is a custom network interface (NIC) developed in the Xilinx Zynq FPGA as part of the AXIOM project.

Figure 1 presents the high-level view of the AXIOM software stack. In particular, we note in the background that all nodes are interconnected (as planned) through the AXIOM-link. Moreover, each board has an FPGA part, including the AXIOM NIC and the XSMLL Layer. The software stack is then running on each board, and is composed by the following set of main components:

- A Linux distribution (not shown in Figure 1). A complete distribution based on the PetaLinux SDK with an Ubuntu 16.04 LTS root filesystem (see D5.1);
- A set of device drivers (used to provide support for the AXIOM NIC and for the AXIOM Memory Allocator);
- A set of user libraries to handle the interfacing between the applications and the kernel drivers;
- A set of utilities, including the daemons `axiom-init`, `axiom-ethtap` and the `axiom-run` spawner;
- The OmpSs programming libraries, including Nanos++, GASNet, and the AXIOM GASNet conduit;
- The compilation toolchain, including the Linaro GCC cross-compiler, and the Mercurium source-to-source compiler.

The following Section contains a list of the software packages, including a short description. In order to simplify the usage of the software stack, we provided a set of Debian packages to install the AXIOM software stack in a very simple way.

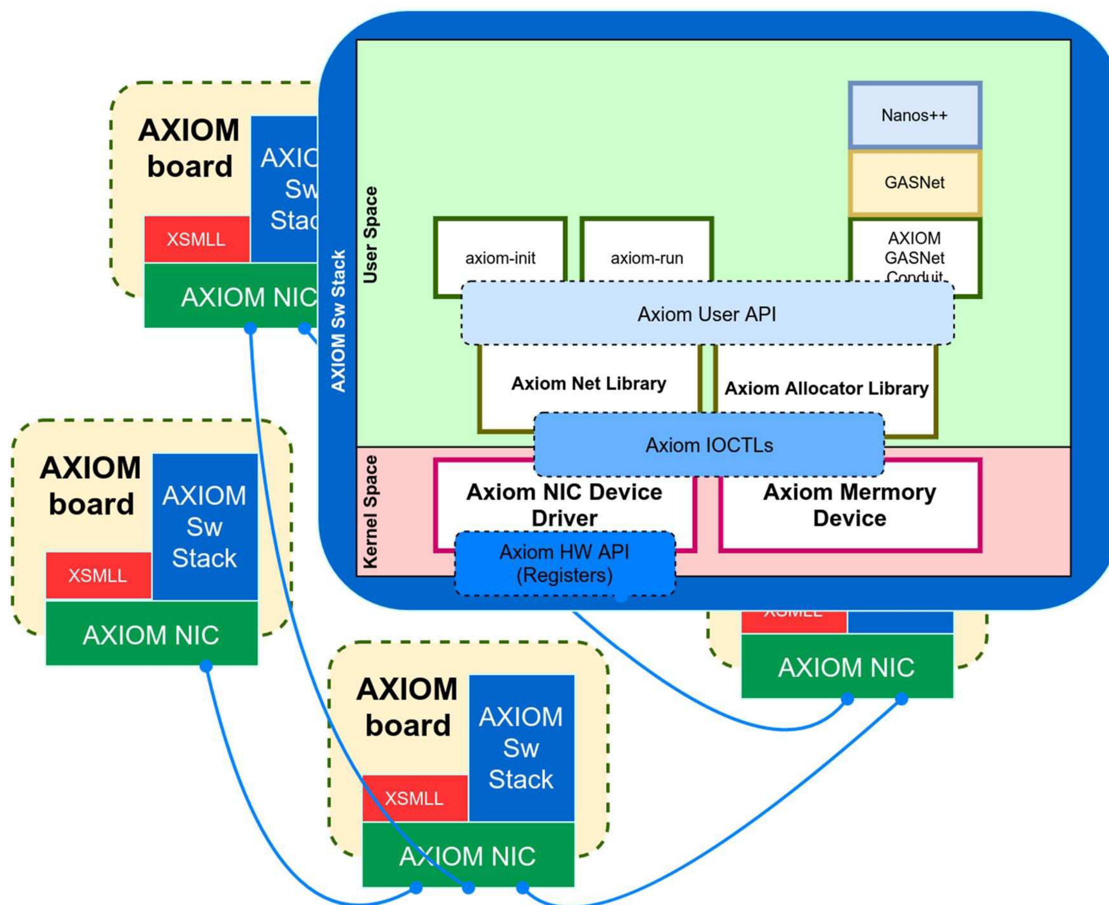

Figure 1 - The AXIOM software stack.

4 Components of the software stack

The following is a list of the main components of the AXIOM software stack. It is meant as a reference that will help to navigate the various packages and directories.

4.1 *AXIOM Drivers, User libraries and applications*

4.1.1 AXIOM NIC driver and libraries source code

- Short Description: Implementation of AXIOM NIC device driver, User Space libraries and documentation.
- GIT Repository: `https://git.axiom-project.eu/axiom-evi-nic`
- Directory in the source zip file: `axiom-evi-src-v1.0.tgz/axiom-evi-nic`
- Doxygen Documentation: in `axiom-v1.0-doxygen` PDF or HTML `files/axiom-evi-nic`

4.1.2 AXIOM allocator source code

- Short Description: Implementation of the three-level AXIOM allocator.

- GIT Repository: https://git.axiom-project.eu/axiom-allocator
- Directory in the source zip file: `axiom-evi-src-v1.0.tgz/axiom-allocator`
- Doxygen Documentation: in `axiom-v1.0-doxygen` PDF or HTML `files/axiom-allocator`

4.1.3 AXIOM memory driver source code

- Short Description: Implementation of the memory device driver to handle virtual to physical memory mapping.
- GIT Repository: https://git.axiom-project.eu/axiom-evi-allocator-drv
- Directory in the source zip file: `axiom-evi-src-v1.0.tgz/axiom-evi-allocator-drv`
- Doxygen Documentation: in `axiom-v1.0-doxygen` PDF or HTML `files/axiom-evi-allocator-drv`

4.1.4 AXIOM memory library source code

- Short Description: Implementation of three-level software allocator based on LMM.
- Repository: https://git.axiom-project.eu/axiom-evi-allocator-lib
- Directory in the source zip file: `axiom-evi-src-v1.0.tgz/axiom-evi-allocator-lib`
- Doxygen Documentation: in `axiom-v1.0-doxygen` PDF or HTML `files/axiom-evi-allocator-drv`

4.1.5 AXIOM application source code

- Short Description: Implementation of AXIOM support application and daemons (`axiom-init`, `axiom-ethtap`, `axiom-run`, etc.).
- GIT Repository: https://git.axiom-project.eu/axiom-evi-apps
- Directory in the source zip file: `axiom-evi-src-v1.0.tgz/axiom-evi-apps`
- Doxygen Documentation: in `axiom-v1.0-doxygen` PDF or HTML `files/axiom-evi-apps`

4.1.6 AXIOM scripts

- Short Description: Makefile and scripts to compile all components of the AXIOM software stack and to generate Debian packages.
  - WEB Repository: https://git.axiom-project.eu/axiom-evi/tree/master/scripts/
- Directory in the source zip file: `axiom-evi-src-v1.0.tgz/scripts`

4.1.7 AXIOM tests

- Short Description: Regression and benchmark tests for AXIOM NIC, GASNet and OmpSS.
  - WEB Repository: https://git.axiom-project.eu/axiom-evi/tree/master/tests/
- Directory in the source zip file: `axiom-evi-src-v1.0.tgz/tests`

4.2 GASNet conduit

- Short Description: Modified version of GASNet library that includes the new AXIOM conduit.
- GIT Repository: https://git.axiom-project.eu/axiom-evi-gasnet
- Directory in the source zip file: `axiom-evi-src-v1.0.tgz/axiom-evi-gasnet`

4.3 OmpSs framework

4.3.1 Extrae

- Short Description: Modified version of Extrae to support the trace of IOCTLs and AXIOM API.
- GIT Repository: https://git.axiom-project.eu/axiom-evi-extrae
- Directory in the source zip file: `axiom-evi-src-v1.0.tgz/axiom-evi-extrae`
- WEB Documentation: https://tools.bsc.es/extrae

4.3.2 Mercurium

- Short Description: Modified version of `mcxx` to support AXIOM GASNet conduit and cross-compilation.
- GIT Repository: https://git.axiom-project.eu/axiom-evi-mcxx
- Directory in the source zip file: `axiom-evi-src-v1.0.tgz/axiom-evi-mcxx`
- WEB Documentation: https://pm.bsc.es/mcxx

4.3.3 Nanos++

- Short Description: Modified version of `nanox` to support AXIOM GASNet conduit and cross-compilation.
- GIT Repository: https://git.axiom-project.eu/axiom-evi-nanox
- Directory in the source zip file: `axiom-evi-src-v1.0.tgz/axiom-evi-nanox`
- WEB Documentation: https://pm.bsc.es/nanox
- WEB OmpSs User's Guide: https://pm.bsc.es/ompss-docs/user-guide

5 Using the AXIOM software stack

The compilation procedure of the AXIOM software stack is documented in the `README` file stored inside the `axiom-evi-src-v1.0.tgz` package. The compilation can be performed on a Linux host machine with a recent distribution (in our case we used Ubuntu 16.04 LTS). The final artifacts of this procedure are Debian packages (`.deb` file extension) that are simple to install in the AXIOM OS based on Ubuntu 16.04 LTS.

When SD cards are ready, it is then possible to interconnect several AXIOM boards by plugging in USB-C cables to any of the four USB-C ports. Since those cables are bi-directional, it is only required to use a single cable to connect two boards. Thanks to the discovery algorithm implemented in the boards, they automatically see each other (plug and play). Therefore, the topology of the cluster will be discovered at run-time from the master node.

During the boot process, AXIOM modules and daemons are automatically loaded. To setup the cluster, it is only needed to connect to one board (which will become the master node) using the serial cable or the Ethernet port, and later plugging in a keyboard and the mouse.

In order to login into the AXIOM board, the following credentials must be used:

Username: `ubuntu`
Password: `ubuntu`

After login, cluster discovery can be started simply by opening a console, and later typing in the following commands:

```
# For security reasons, all AXIOM applications must be executed
# using root privileges

sudo su
axiom-make-master.sh
```

At the end of the discovery process, the master node prints the network topology and his routing table. A virtual IP network over AXIOM-link is also setup, and thus a remote connection to the other AXIOM boards in the cluster is possible using the IP 192.168.17.`NODEID` (e.g. through `ssh`). All IP network traffic on subnet 192.168.17.0/24 is tunneled trough the AXIOM-link.

On the console of one of the nodes, you can try the following AXIOM applications (the complete list is available by typing `axiom-` on the console, and then the tab command twice):

```
axiom-info
```

- Prints all the information related to the AXIOM NIC (node id, interface status, routing table… etc).

```
axiom-ping -d 1
```

- Pings (talking to the `axiom-init` daemon running on all nodes) node 1.

```
axiom-netperf -d 1 -t rdma -l 100M
```

- Starts a `netperf` to node 1 with message type RDMA and a 100-Mbyte payload.
- Note, before to start the `axiom-netperf` client, you must start the server on the target node using the following command:

```
axiom-netperf –s –n 8
```

Some OmpSs test applications are already installed on the AXIOM board (`/opt/axiom`). For example, in order to run the Matrix Multiply application on the AXIOM cluster, the following steps must be completed:

```
cd /opt/axiom/test_ompss
./run_test_ompss.sh ./ompss_evimm 4 1000
```

It also included a script to run the Matrix Multiply with the improvements discussed in Section 7:

```
./run_test_ompss_sched.sh ./ompss_evimm 4 1000
```

6 Mechanisms for load balancing

6.1 Improving load balancing with OmpSs@FPGA

In the original OmpSs infrastructure, FPGA helper threads were fully dedicated to manage the FPGA IP cores and data transfers. Usually, this leads to the loose of as much cores as helper threads the application uses. Also, SMP worker threads are exclusively dedicated to execute SMP tasks, and have no involvement on the management of the FPGA.

Figure 2 shows an execution trace of this version of the infrastructure, where there are 3 SMP cores executing matrix multiplication tasks, and one helper thread taking care of the 256x256 IP core in the FPGA. This matrix multiplication works on a matrix of 2048x2048 single precision floating point elements, on blocks of 256x256 elements, running on the AXIOM board. It is described in the AXIOM Deliverable D4.3 [1]. As it can be appreciated, the helper thread (Thread 1.1.4 in the figure) is busy less than 50% of the time. This fact allows to consider the possibility that this core acts also as a worker, and the management of the FPGA gets distributed across all threads.

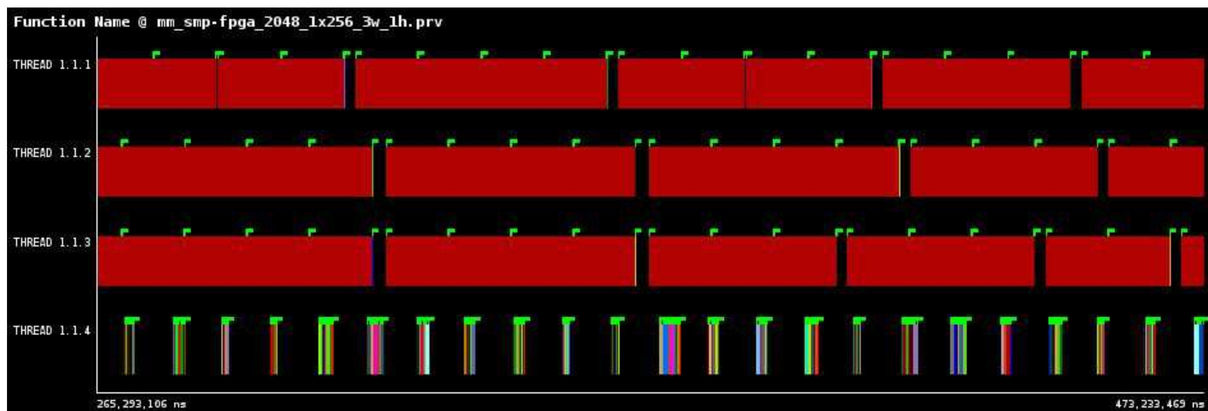

Figure 2 - Execution trace of matrix multiplication using 3 SMP cores and 1 FPGA IP core.

We have modified the infrastructure in order to support "hybrid workers". A hybrid worker is going to deal with the FPGA tasks when idle. For example, they can drain FPGA tasks that have already finalized, update the task graph, and release dependences of successor tasks.

With this technique, tasks execution is better balanced between the SMP cores and the FPGA, and we have achieved an increase in performance of a few GFlops in the matrix multiplication. For example, the previous execution with 3 SMP workers and 1 helper thread obtains 27.3 GFlops, with no hybrid support. When the hybrid support is added, the performance increases up to 29.2 GFlops. The reason is that the number of tasks executed in the FPGA increases as FPGA tasks finalized are released earlier by one of the idle workers. In this particular execution view, FPGA tasks raised from 178 to 184. Figure 3 shows the instrumentation internals of the hybrid execution. Observe how in this new execution, there are also 4 matrix multiplication tasks running in parallel (on each of the workers Threads 1.1.1, 1.1.6, 1.1.7 and 1.1.8), while the FPGA is executing tasks at a slightly higher rate than the execution without the hybrid workers (in Figure 3).

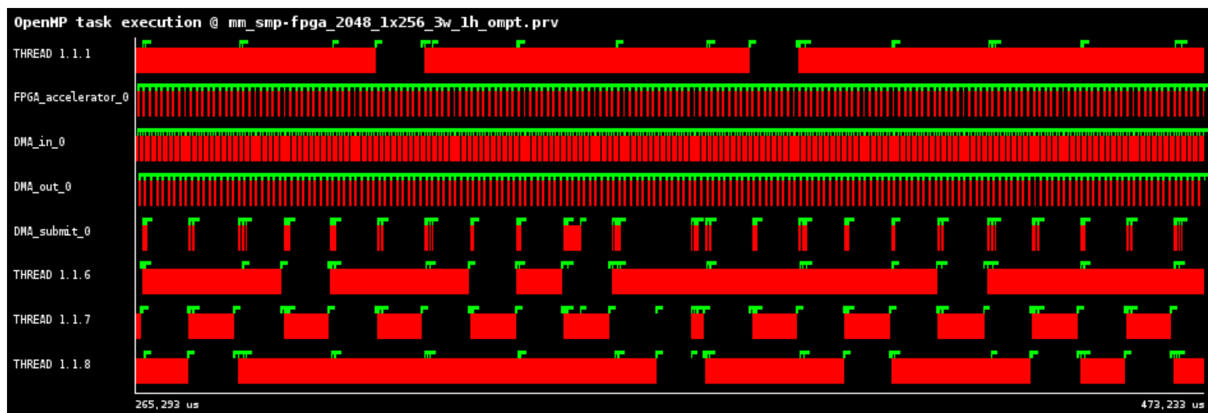

Figure 3 - Execution of matrix multiplication with the hybrid worker and showing the instrumentation internals.

6.1.1 Future work

We think that the hybrid worker approach presented in this deliverable can also be used in GPU environments, where there is also the need to have a number of helper threads to take care of the GPU events. We plan to work on this direction on our future development of OmpSs@CUDA and OmpSs@OpenCL.

6.2 Load balancing by dataflow-thread based execution

In relation with the task T5.4, we have experimented also another advanced approach for distributing threads across several nodes based on dataflow-threads (DF-Threads) [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]. The XSMLL model provides a robust and fast substrate to moderate the distributed threads across the nodes of a Multi-core/Multi-node computing system, which include thread scheduling, and appropriate management of data consistency and synchronization.

A simplified sketch of the architecture that we have in mind is represented in Figure 4. Here the focus is on the software components like drivers, protocols, runtime and I/O register specification (see also Appendix 1). As we can see from Figure 4, the software components (processing system or PS) need to queue operations through the undelaying hardware (programmable logic or PL)

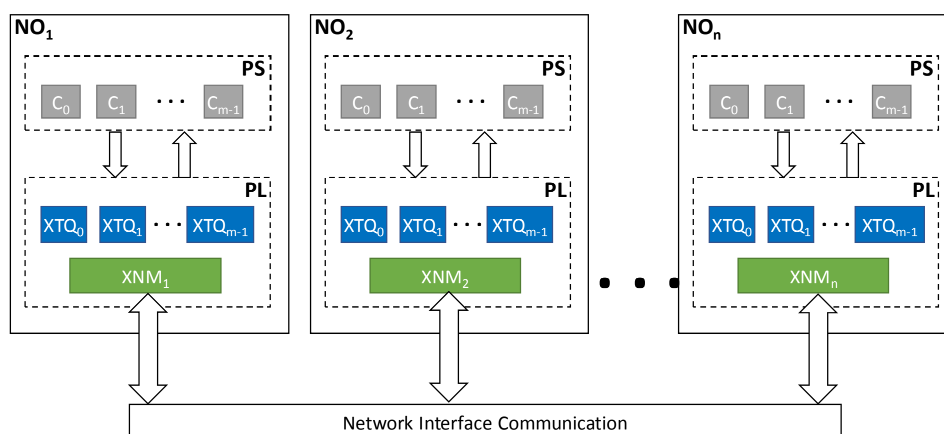

Figure 4 – XSMLL hardware/software architecture. There are n nodes and m cores per node. XTQ are the XSMLL Thread Queues and XNM is the XSMLL Node Manager that implements several finite state machines.

Building on the work of previous years, we have been able to prototype this architecture by a combination of models in the simulation software (based on the COTSon framework [21], [22]) and of the high-level synthesis (HLS) provided by the Xilinx Vivado tools. In the Appendix 1, there are the precise I/O registers that we have defined. These registers are very important since they represent the "contract" between hardware and software. The software, in particular, has to be aware of those registers.

Each DF-thread operates regular instructions and uses the primitive commands/instructions outlined in D5.2 (XSCHEDULE… etc). We recall that a DF-thread expects no parameters, and also does not return parameters. As such, it will be started once the input data are ready. This is in contrast with classical `pthreads`-like execution model where a thread has direct access to any address of memory.

The most important challenges that XSMLL tries to solve are the following:

1) At the hardware level, fine grain threads are based on a dataflow execution model and can be distributed across the nodes (FPGAs)
2) At system level, the functionality of the system should not be affected if one or more of the Nodes are removed from the distributed system (i.e. existence of any error at any nodes)

Special direct access from the XSMLL hardware to/from the Network Interface (NI) that are both on the PL were also modeled and designed. The following picture summarizes a possible exchange of messages managed directly from the PL, and indirectly through the primitive commands such as XSCHEDULE (Figure 5).

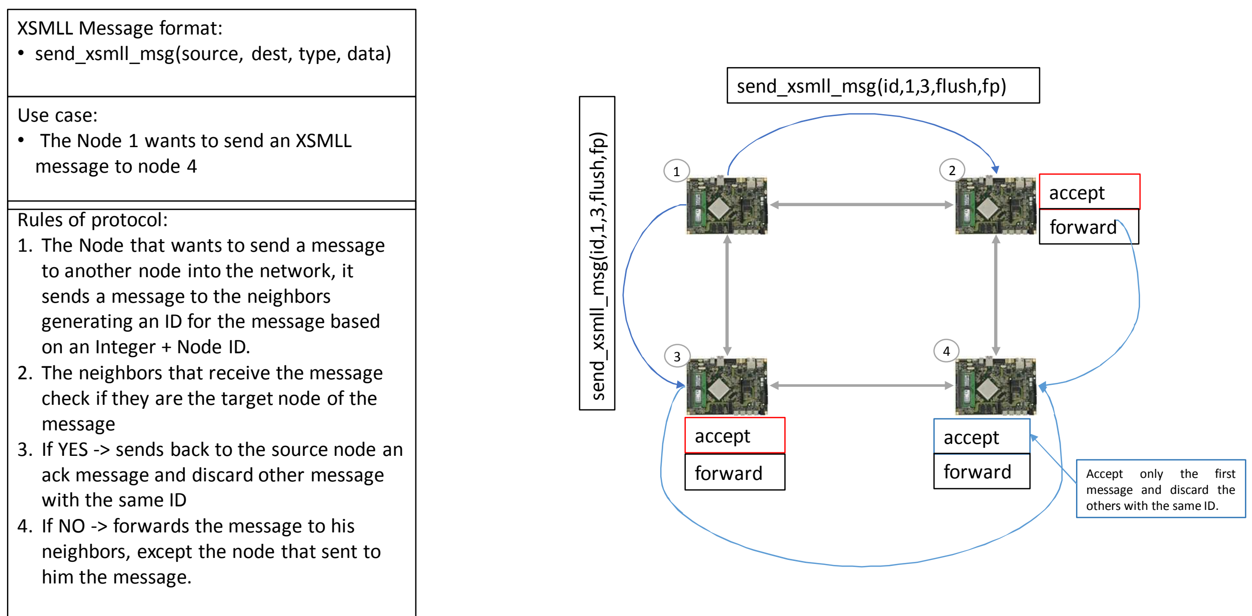

Figure 5 – XSMLL message protocol.

As an example, we report the time diagram of the interactions between cores, the XTQ and other internal queues (Figure 6). In order to verify the efficiency in distributing threads across the system we investigated the following programming models OpenMPI[24], JUMP[23], and how efficient the execution is when compared to our approach (i.e. based on the XSMLL execution model, which can rely on OmpSs as it was outlined in D4.2).

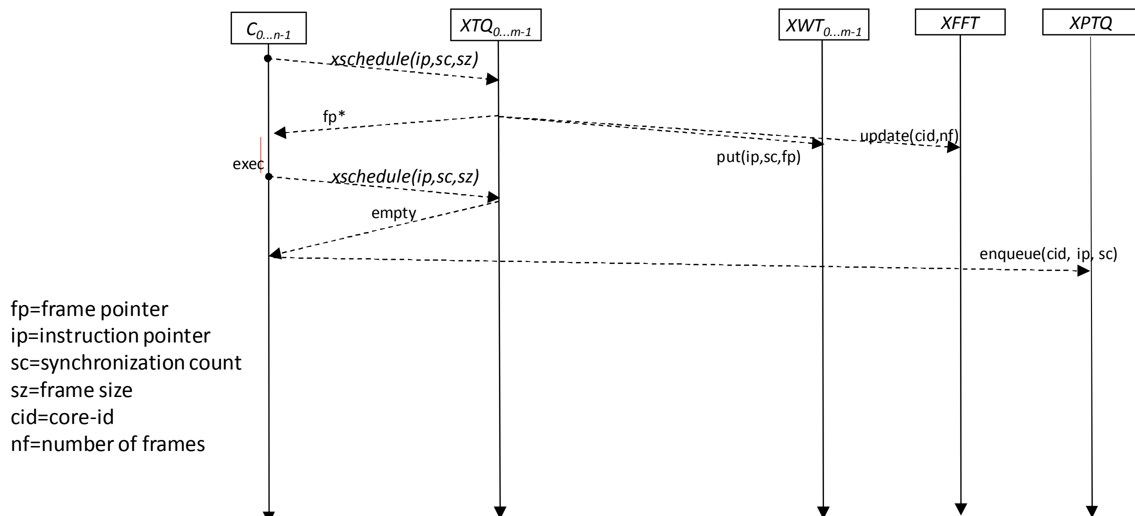

Figure 6 – XSMLL message protocol. C=core, XTQ= XSMLL Thread queue, XWT=XSMLL Waiting Table, XFFT=XSMLL Free-Frame Table, XPTQ=XSMLL Pending Thread-request Queue.

The preliminary results of this comparison (simulation based – for a comparison with the AXIOM boards see D7.3) are shown in Figure 7 and Figure 8. As it can be seen, XSMLL provides not only good scalability up to 16 nodes (or boards) but also highly competitive execution times.

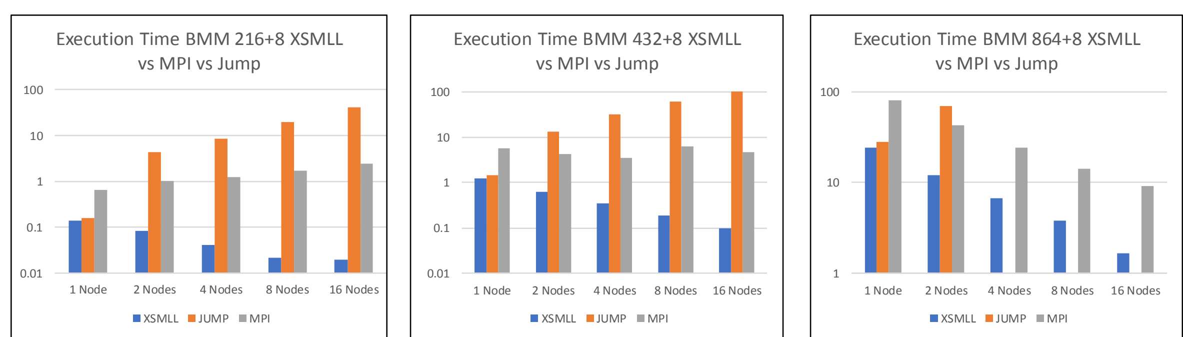

Figure 7 – Comparison among XSMLL execution time (seconds) and OpenMPI, JUMP programming model execution. The benchmark is matrix multiplication with square matrix size of 216, 432, 864 and block size of 8. The nodes are configured with only 1 core each.

Moreover, we think that there is still much space for further optimization of the XSMLL. In fact, by investigating the miss rate, we noticed that the locality of the XSMLL program is still unexploited. A policy that is under investigation is to inject the frame data into the caches before starting the execution of the DF-thread. Note: some bars are missing due to the fact that JUMP execution crashed (bug).

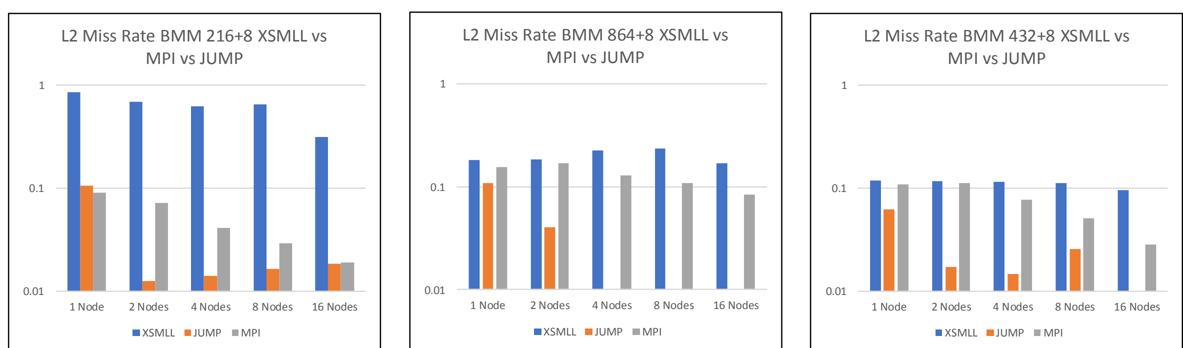

Figure 8 – Comparison among XSMLL L2-cache miss rate and OpenMPI, JUMP programming model L2 miss-rate. The benchmark is matrix multiplication with square matrix size of 216, 432, 864 and block size of 8. The nodes are configured with only 1 core each.

7 Analysis of the real-time guarantees

This Section is related to the analysis of the AXIOM stack performed by EVI, in relationship with the possibility of providing real-time guarantees about latencies in both communication and processing.

In general, we need to start by saying that OpenMP “as is” is a parallel programming model which has been developed to provide throughput and full utilization of multicore architectures and, in the case of OmpSs@Cluster, cluster configurations. In general, the execution flow is based on the dynamic (at run-time) expansion of the parallel execution graph and the consequent execution of OpenMP tasks considering their data dependencies. All this is done in a way to maximize the overall system performance, without any consideration for real-time loads. Various efforts have been done in recent research to prove the applicability of OpenMP in contexts related to safety, especially in avionics and automotive. These efforts are typically composed at least by the following steps:

- Analysis of the timing behavior of the OpenMP tasking model.
- Compiler analysis tools to extract a representation of the parallel execution of an OpenMP program in the form of a DAG.
- Schedulability tests for OpenMP applications represented with DAGs.
- Static scheduling approaches to assign threads to cores based on DAGs.
- Run-time methods to enhance time predictability.

These steps were the basis of the fundamental work that was performed during the P-SOCRATES FP7 Project [2], which resulted in the UpScale Framework [3], where partner EVI provided the runtime operating system for running concurrent periodically scheduled OpenMP applications.

Additional efforts towards providing guarantees for real-time loads are related to the applicability of OpenMP with the Ada language to provide parallel predictability [4].

In the work presented in this Section we will be mostly investigate how the usage of operating system real-time mechanisms (such as priorities, and real-time schedulers such as `SCHED_DEADLINE`) could impact the overall response time/latency of execution of a computational task with OmpSs@Cluster.

The starting point of the activity has been the standard Nanos++ configuration for OmpSs@Cluster. In particular, when Nanos++ works in the cluster, it creates one "communication thread" on each node to handle the exchange of messages and memory in the cluster. This thread, through the GASNet API, uses the AXIOM conduit developed in this project to move memory and messages between nodes. The communication thread is typically pinned to the fourth core, where it executes “alone” to handle the communication in the fastest way as possible.

The first finding was related to the fact that the fourth core is not always the best for the allocation of the communication between threads. In particular, OmpSs@Cluster typically allocates the computational threads on core 0, 1, and 2, whereas the communication thread is typically allocated on core 4. The typical behavior of the communication thread is to actively spin while waiting for new activities to perform on the network; on the other hand, Linux systems schedule most of the interrupts on the core 0, thus interrupting the computational threads allocated on that core. This fact is evident in Figure 9, where we can see that core 0 has been interrupted by IRQs coming mainly from the network interface, while core 4 (not shown in Figure 9), is basically spinning. Therefore, the first action to improve the load balance in the system has been the move of the IRQs processing into the same core used for communication, leaving in this way the computational threads free of interferences.

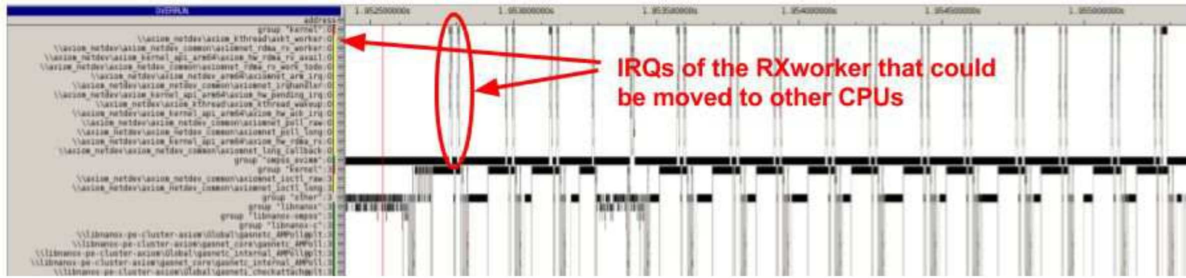

Figure 9 - IRQs are typically scheduled on Core 0, interrupting and slowing down computational threads.

The second finding is related to the current implementation of the "communication thread", which is done using the GASNet no-blocking API in a busy-wait loop. In this way, the latency of the communication is kept at the minimum, but this behavior has a drawback: one core is fully used by the "communication thread" without executing any task. Therefore, OmpSs@Cluster uses only N-1 cores on each board to execute tasks, allocating N-1 "working thread" and 1 core only for communicating with the other boards. (N = number of cores per board).

In Section 6 (and in the D4.3 deliverable), we described some possible improvements to OmpSs@Cluster, related to GASNet and Nanos++ modifications made to avoid busy waits and memory copies in order to improve performance and energy consumption. Some of those modifications required to the Nanos++ structure are big, and because of their impact on the OmpSs@Cluster architecture they have been implemented only partially during the AXIOM project timeframe. However, the usage of real-time priorities and in particular of the `SCHED_DEADLINE` scheduler opens the possibility of reducing the computational load of the communication thread, leaving space for additional computation to be performed on the fourth core.

In particular, the idea of `SCHED_DEADLINE` (see Figure 10) is to provide the possibility to specify a pair Budget/Period for a thread. It is then guaranteed that the thread will be able to execute a maximum execution time equal to the budget over the defined period. A thread trying to consume more CPU (the top one in the Figure 10) is throttled and resumed afterwards.

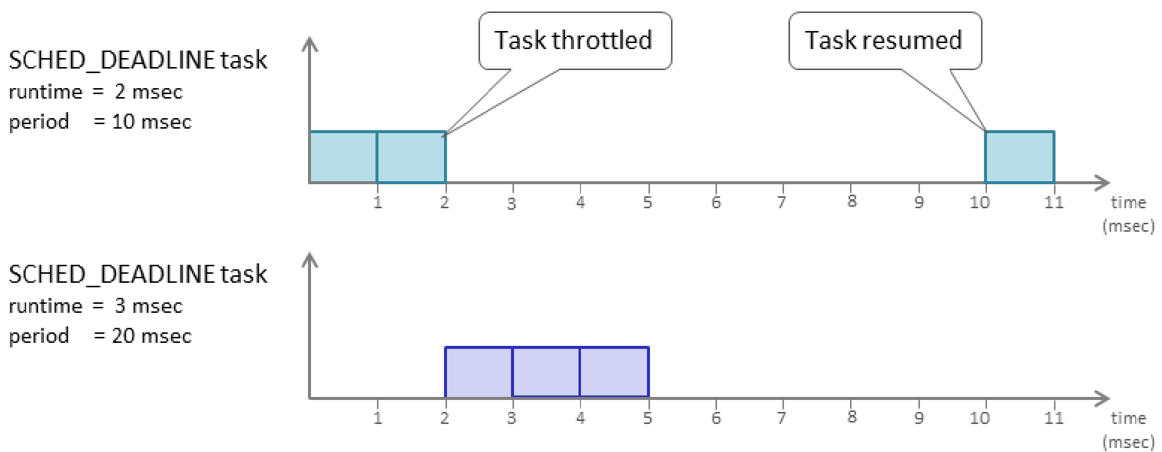

Figure 10 - Under SCHED_DEADLINE, a task consuming too much CPU is throttled.

Therefore, the main idea to improve the real-time features of the AXIOM software was to schedule the communication thread via the `SCHED_DEADLINE` feature of the Linux kernel. Since the communication thread is constantly executing, the resulting behaviour will be a throttling of the thread, which leaves free CPU time that can be allocated to other computational threads. Note that the future implementation of the usage of blocking primitives described above can still be applied with `SCHED_DEADLINE`. The result will be that the communication thread will be mostly sleeping waiting for events to appear. When the communication thread wakes up because of an action to perform, the thread will be immediately scheduled under the budget/period available at wake up, still resulting in an improved latency despite low processor utilization.

Once the communication thread has been "gated" by giving a `SCHED_DEADLINE` bandwidth (defined as the ratio budget/period), we can start to increase the system performance by slightly modifying Nanos++ to create an extra "working thread". In this way, we have N "working threads", one per CPU, plus 1 "communication thread". We then further modified Nanos++ in order to set the scheduling parameters for each low-level `pthread`, to be able to fine tune the execution of each worker (in detail, we allow the specification at runtime of the Linux scheduling policy and of the scheduling parameters of each `pthread`).

The main question at this point is what is the optimal bandwidth that should be assigned to the communication thread in order to maximize the response times and latencies of the system. To evaluate this, we performed a set of tests running a matrix multiplication application on a 2 node OmpSs@Cluster on the AXIOM boards. The tests have been performed in a way to assign decreasing bandwidth to the communication thread. The remaining free time has been then taken by the (additional) computational thread allocated on the same CPU.

Qualitatively, the expectation is that when the bandwidth of the communication thread is high, the communication thread mostly actively spins waiting for some activity to be performed, but no additional computation is performed. As long as the communication thread reduces its used bandwidth, we expect to see less active spinning, but more computation to be performed, at the expense of a slightly increased communication latency. Initially, the performance will be worse, because giving little time to a computational thread means slowing down its task, increasing the overall latency. Increasing it further provides a better CPU utilization and therefore a smaller execution time of the same application. The latencies will decrease up to a given bandwidth value (that in the case of the matrix multiplication was around 5% of the CPU bandwidth), where the communication tasks to be performed are slowed down worsening the overall execution time.

Figure 11 shows this behaviour. The period of the `SCHED_DEADLINE` budget for the communication task was fixed to 100 ms and the bandwidth given to it was starting at 90%, and then decreased by 20% at each step until 30% is reached, and afterwards decreasing by 1% from 20% to 1%. In addition to the `SCHED_DEADLINE` tests (blue line), we also added a test using `nice` (red line). `nice` is a feature present on most UNIX schedulers, which demotes a given task, with the result of giving less CPU bandwidth. This can be seen as a rough approximation of the behaviour obtained with `SCHED_DEADLINE`, but without any kind of guarantee from the timing perspective. Figure 11 also shows the "standard" result (orange line) obtained by giving 1 entire core (100% CPU bandwidth) to the communication thread. All the values showed in the Figure 11 and Figure 12 are an average of 5 runs of the same experiment.

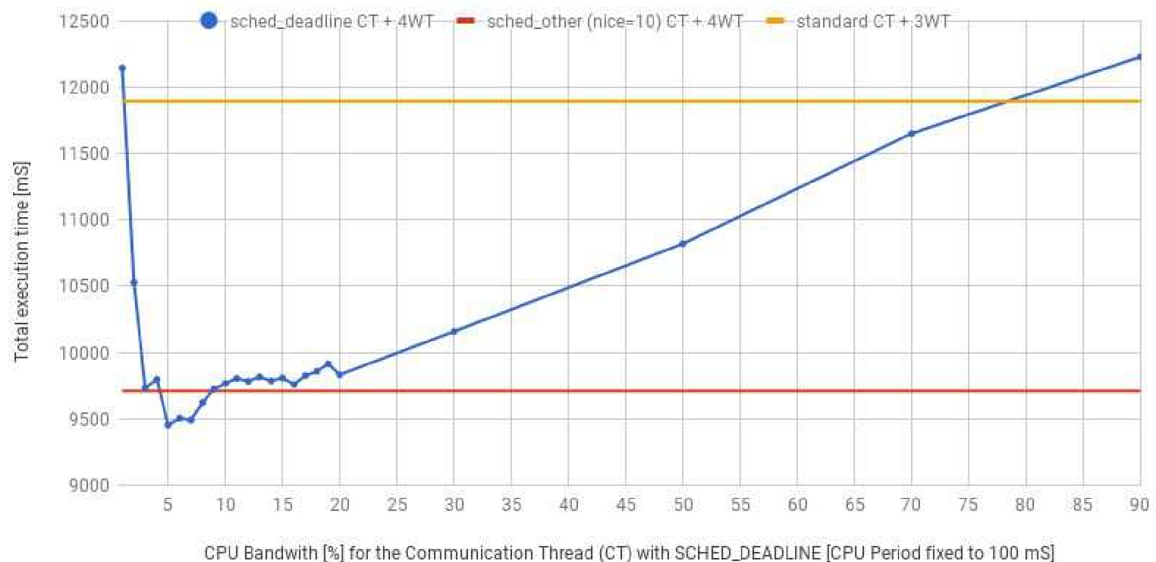

Figure 11 - Execution time of a matrix multiply varying the SCHED_DEADLINE bandwidth parameter with a fixed period (100 ms) for the Communication Thread.

Figure 11 shows that the best performance is obtained assigning 5% of CPU bandwidth to the communication task, and leaving the remaining 95% to the computational tasks.

It has to be noted that the parameters that can be selected in `SCHED_DEADLINE` are two (period and budget), but in the previous examples all tests have been performed with a fixed "reasonable" period. In order to evaluate the dependency on the choice of the period, we also did another experiment, shown in Figure 12.

In this case, we fixed the CPU bandwidth to 5% (which is the best one found in the previous experiment), and we changed the period of the task. The red and orange lines are the same of the Figure 11. The blue line shows the impact of a change of the period in a `SCHED_DEADLINE` reservation: by using a high value for the period means that the communication thread will be given coarse grained amount of times, that likely will be spent doing some communication work, and after that will be "wasted" in spinning until the budget ends. A small period provides more fine-grained execution, at the expense of a greater context switch overhead (5% bandwidth with a period of 1 ms is at the limit of the precision of the current architecture).

Another interesting finding is related to the fact that the performance at 200 ms is worse than the one at 300 ms. This may be due to the fact that the typical matrix multiplication OmpSs task in the example has an execution time near to 200 ms, and not being able to perform the communication right after the end of a task may imply for the communication thread a waiting of another period, thus increasing the overall latency.

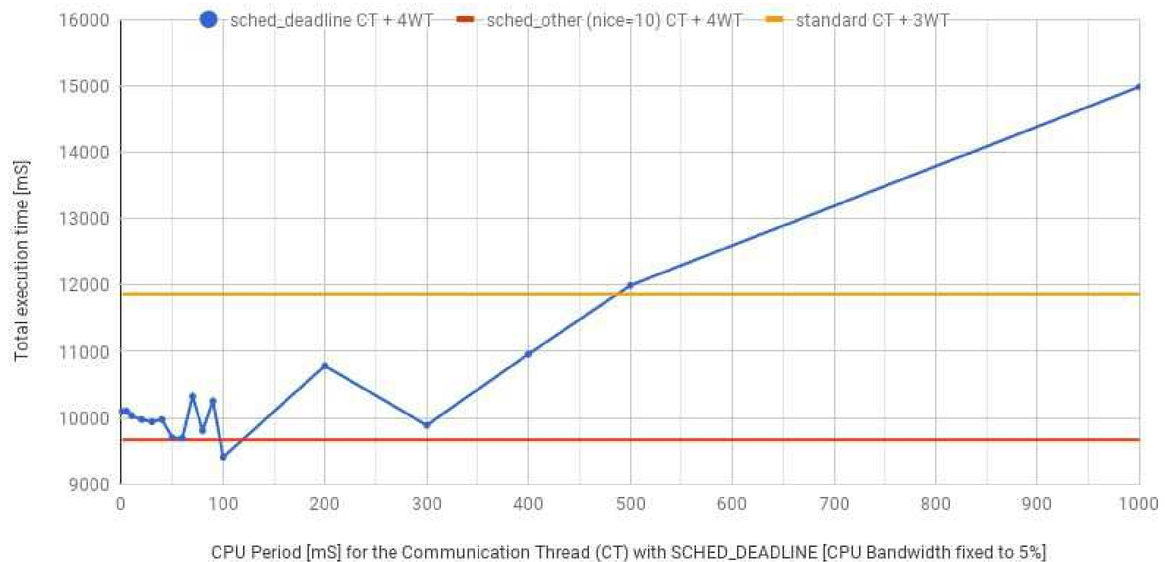

Figure 12 - Execution time of a matrix multiply varying the SCHED_DEADLINE period parameter with a fixed CPU bandwidth (5%) for the Communication Thread.

In conclusion, we believe that a proper coexistence in the same cores of the computational thread with the communication thread in the OmpSs@Cluster in conjunction with the real-time scheduling policies available in modern Linux kernels, provides additional benefits in terms of better overall usage of the computational resources and lower overall execution times. We also believe that the bandwidth assigned to the communication thread should be evaluated on a per application basis, based on the communication load of the specific case.

Starting from these considerations, the implementation of the Cluster Hybrid approach in Nanos++ described in D4.3 further enhances the performance of the system by limiting the active spinning performed by the communication thread, thus de facto limiting the need of the usage of the scheduling policies described in this Section.

8 Archives released

All the software packages and documentation included as part of this release of the AXIOM project software stack are available for public download from the following address:

https://download.axiom-project.eu/?dir=RUNTIME

9 Confirmation of DoA objectives and Conclusions

This document shortly described the open-source release of the software developed during the AXIOM project. The software includes a full network stack with RDMA support, running on AXIOM boards based on the Xilinx Ultrascale+ chip. The software also includes the complete support for the OmpSs@Cluster Parallel Programming Library, including support for the AXIOM cluster configuration and the possibility to trace the software execution using the Extrae tool. An evaluation of the load balancing and of the real-time performance of the system has also been included.

Other related publications of this project can be found in the references [5]-[43].

All the DoA objectives were successfully met.

10 References

1. AXIOM Consortium, "D4.3 - Evaluation of the compiler and tools infrastructure", February 2018.
2. P-SOCRATES FP7 Project, http://www.p-socrates.eu/.
3. UpScale Framework, http://www.upscale-sdk.com/.
4. Sara Royuela, Luis Miguel Pinho and Eduardo Quinones, Converging Safety and High-performance Domains: Integrating OpenMP into Ada, In the Design, Automation, and Test in Europe conference (DATE). Dresden (Germany), March 19-23, 2018.
5. R. Giorgi, “Transactional memory on a dataflow architecture for accelerating Haskell,” WSEAS Trans. Computers, vol. 14, pp. 794–805, 2015.
6. R. Giorgi and P. Faraboschi, “An introduction to DF-Threads and their execution model,” in IEEE MPP, Paris, France, Oct. 2014, pp. 60–65.
7. R. Giorgi and A. Scionti, “A scalable thread scheduling co-processor based on data-flow principles,” ELSEVIER Future Generation Computer Systems, vol. 53, pp. 100–108, July 2015.
8. S. Weis, A. Garbade, B. Fechner, A. Mendelson, R. Giorgi, and T. Ungerer, “Architectural support for fault tolerance in a Teradevice dataflow system,” Springer Int.l Journal of Parallel Programming, pp. 1–25, May 2014.
9. D. Theodoropoulos et al., “The AXIOM project (agile, extensible, fast I/O module),” in IEEE Proc. 15th Int.l Conf. on Embedded Computer Systems: Architecture, MOdeling and Simulation, July 2015.
10. R. Giorgi, "Scalable Embedded Systems: Towards the Convergence of High-Performance and Embedded Computing", Proc. 13th IEEE/IFIP Int.l Conf. on Embedded and Ubiquitous Computing (EUC 2015), Oct. 2015.
11. R. Giorgi, "Exploring Future Many-Core Architectures: The TERAFLUX Evaluation Framework", Elsevier, 2017, pp. 33-72.
12. R. Giorgi, "Exploring Dataflow-based Thread Level Parallelism in Cyber-Physical Systems", Proc. ACM Int.l Conf. on Computing Frontiers, New York, NY, USA, 2016, pp. 6.
13. A. Rizzo, G. Burresi, F. Montefoschi, M. Caporali, R. Giorgi, "Making IoT with UDOO", Interaction Design and Architecture(s), vol. 1, no. 30, Dec. 2016, pp. 95-112.
14. L. Verdoscia, R. Giorgi, "A Data-Flow Soft-Core Processor for Accelerating Scientific Calculation on FPGAs", Mathematical Problems in Engineering, vol. 2016, no. 1, Apr. 2016, pp. 1-21.
15. R. Giorgi, N. Bettin, P. Gai, X. Martorell, A. Rizzo, "AXIOM: A Flexible Platform for the Smart Home", Springer Int.l Publishing, Cham, 2016, pp. 57-74.
16. P. Burgio, C. Alvarez, E. Ayguade, A. Filgueras, D. Jimenez-Gonzalez, X. Martorell, N. Navarro, R. Giorgi, "Simulating next-generation cyber-physical computing platforms", Ada User Journal, vol. 37, no. 1, Mar. 2016, pp. 59-63.
17. R. Giorgi, "AXIOM: A 64-bit reconfigurable hardware/software platform for scalable embedded computing", 6th Mediterranean Conf. on Embedded Computing (MECO), June 2017, pp. 113-116.
18. R. Giorgi, "Accelerating Haskell on a Dataflow Architecture: a case study including Transactional Memory", Proc. Int.l Conf. on Computer Engineering and Applications (CEA), Dubai, UAE, Feb. 2015, pp. 91-100.
19. N. Ho, A. Mondelli, A. Scionti, M. Solinas, A. Portero, R. Giorgi, "Enhancing an x86_64 Multi-Core Architecture with Data-Flow Execution Support", ACM Computing Frontiers, May 2015, pp. 1-2.
20. N. Ho, A. Portero, M. Solinas, A. Scionti, A. Mondelli, P. Faraboschi, R. Giorgi, "Simulating a Multi-core x86-64 Architecture with Hardware ISA Extension Supporting a Data-Flow Execution Model", IEEE Proc. AIMS-2014, Madrid, Spain, Nov. 2014, pp. 264-269.
21. A. Portero, A. Scionti, Z. Yu, P. Faraboschi, C. Concatto, L. Carro, A. Garbade, S. Weis, T. Ungerer, R. Giorgi, "Simulating the Future kilo-x86-64 core Processors and their Infrastructure", 45th Annual Simulation Symp. (ANSS12), Orlando, FL, Mar 2012, pp. 62-67.
22. Argollo, E., Falcón, A., Faraboschi, P., Monchiero, M., and Ortega, D. 2009. COTSon: infrastructure for full system simulation. SIGOPS Oper. Syst. Rev. 43, 1 (Jan. 2009), 52-61.
23. B.W.L. Cheung, C.L. Wang, Kai Hwang; “JUMP-DP: A Software DSM System with Low-Latency Communication Support”, Int’l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA2000), pp. 445-451, June 2000.
24. E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham, and T.S. Woodall, “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”, In Proc. 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, Sep. 2004.
25. L. Verdoscia, R. Vaccaro, R. Giorgi, "A matrix multiplier case study for an evaluation of a configurable Dataflow-Machine", ACM CF'15 - LP-EMS, May 2015, pp. 1-6. DOI:10.1145/2742854.2747287

26. G. Burresi, R. Giorgi, "A Field Experience for a Vehicle Recognition System using Magnetic Sensors", IEEE MECO 2015, Budva, Montenegro, June 2015, pp. 178-181. DOI:10.1109/MECO.2015.7181897,
27. A. Mondelli, N. Ho, A. Scionti, M. Solinas, A. Portero, R. Giorgi, "Dataflow Support in x86-64 Multicore Architectures through Small Hardware Extensions", IEEE Proc. DSD, August 2015, pp. 526-529.. DOI: 10.1109/DSD.2015.62
28. C. Alvarez, E. Ayguade, J. Bueno, A. Filgueras, D. Jimenez-Gonzalez, X. Martorell, N. Navarro, D. Theodoropoulos, D. Pnevmatikatos, C. Scordino, P. Gai, C. Segura, C. Fernandez, D. Oro, J. Saeta, P. Passera, A. Pomella, A. Rizzo, R. Giorgi, "The AXIOM Software Layers", IEEE Proc. 18th EUROMICRO-DSD, Aug. 2015, pp. 117-124. DOI:10.1109/DSD.2015.52
29. D. Jiménez-González, C. Álvarez, A. Filgueras, X. Martorell, J. Langer, J. Noguera, K. Vissers. “Coarse-grain performance estimator for heterogeneous parallel computing architectures like Zynq all-programmable SoC”. arXiv preprint arXiv:1508.06830.
30. R. Giorgi, A. Scionti, "A scalable thread scheduling co-processor based on data-flow principles", ELSEVIER Future Generation Computer Systems, Amsterdam, Netherlands, vol. 53, Dec. 2015, pp. 100-108. DOI:10.1016/j.future.2014.12.014
31. R. Giorgi, "Exploring Dataflow-based Thread Level Parallelism in Cyber-physical Systems", Proc. ACM Int.l Conf. on Computing Frontiers, New York, NY, USA, 2016, pp. 6. DOI:10.1145/2903150.2906829
32. C. Alvarez, E. Ayguade, J. Bosch, J. Bueno, A. Cherkashin, A. Filgueras, D. Jiminez-Gonzalez, X. Martorell, N. Navarro, M. Vidal, D. Theodoropoulos, D. Pnevmatikatos, D. Catani, D. Oro, C. Fernandez, C. Segura, J. Rodriguez, J. Hernando, C. Scordino, P. Gai, P. Passera, A. Pomella, N. Bettin, A. Rizzo, R. Giorgi, "The AXIOM Software Layers", ELSEVIER Microprocessors and Microsystems, vol. 47, Part B, 2016, pp. 262-277. DOI:10.1016/j.micpro.2016.07.002
33. S. Mazumdar, E. Ayguade, N. Bettin, S. Bueno J. and Ermini, A. Filgueras, D. Jimenez-Gonzalez, C. Martinez, X. Martorell, F. Montefoschi, D. Oro, D. Pnevmatikatos, A. Rizzo, D. Theodoropoulos, R. Giorgi, "AXIOM: A Hardware-Software Platform for Cyber Physical Systems", 2016 Euromicro Conf. on Digital System Design (DSD), Aug 2016, pp. 539-546. DOI:10.1109/DSD.2016.80
34. G. Llort, A. Filgueras, D. Jiménez-González, H. Servat, X. Teruel, E. Mercadal and J. Labarta. “The Secrets of the Accelerators Unveiled: Tracing Heterogeneous Executions Through OMPT”. In International Workshop on OpenMP (pp. 217-236). Springer, Cham. DOI: 10.1007/978-3-319-45550-1_16
35. R. Giorgi, N. Bettin, P. Gai, X. Martorell, A. Rizzo, "AXIOM: A Flexible Platform for the Smart Home", Springer Int.l Publishing, Cham, 2016, pp. 57-74. DOI:10.1007/978-3-319-42304-3_3
36. R. Giorgi, S. Mazumdar, S. Viola, P. Gai, S. Garzarella, B. Morelli, D. Pnevmatikatos, D. Theodoropoulos, C. Alvarez, E. Ayguade, J. Bueno, A. Filgueras, D. Jimenez-Gonzalez, X. Martorell, "Modeling Multi-Board Communication in the AXIOM Cyber-Physical System", Ada User Journal, vol. 37, no. 4, December 2016, pp. 228-235. ISSN: 1381-6551
37. M. Wagner, G. Llort, A. Filgueras, D. Jiménez-González, H. Servat, X. Teruel, and E. Ayguadé. “Monitoring Heterogeneous Applications with the OpenMP Tools Interface”. In Tools for High Performance Computing 2016 (pp. 41-57). Springer. DOI: 10.1007/978-3-319-56702-0_3
38. D. Theodoropoulos, S. Mazumdar, E. Ayguade, N. Bettin, J. Bueno, S. Ermini, A. Filgueras, D. Jimenez-Gonzalez, C. Alvarez Martinez, X. Martorell, F. Montefoschi, D. Oro, D. Pnevmatikatos, A. Rizzo, P. Gai, S. Garzarella, B. Morelli, A. Pomella, R. Giorgi, "The AXIOM platform for next-generation cyber physical systems", Microprocessors and Microsystems, 2017. DOI:10.1016/j.micpro.2017.05.018
39. A. Rizzo, F. Montefoschi, M. Caporali, A. Gisondi, G. Burresi, R. Giorgi, "Rapid Prototyping IoT Solutions Based on Machine Learning", Proc. European Conf. on Cognitive Ergonomics 2017, New York, NY, USA, 2017, pp. 4. DOI:10.1145/3121283.3121291
40. D. Theodoropoulos, D. Pnevmatikatos, S. Garzarella, P. Gai, A. Rizzo, R. Giorgi. “AXIOM: enabling parallel processing in cyber-physical systems.” Reconfigurable Computing Workshop, Lausanne, CH. Sep 2016. Pp.1-2.
41. J. Bosch Pons, “Asynchronous runtime for task-based dataflow programming models.” Jul. 2017. Master's Thesis. Universitat Politecnica de Catalunya.
42. S. Mazumdar and R. Giorgi. "A Survey on Hardware and Software Support for Thread Level Parallelism." arXiv preprint arXiv:1603.09274 (2016).
43. C. Scordino and B. Morelli. “Sharing memory in modern distributed applications”. In Proceedings of the 31st Annual ACM Symposium on Applied Computing (pp. 1918-1921) ACM, April 2016.

11 Appendix 1

1.1 Communication between XSMLL Hardware and Software

This section illustrates detailed register maps through which software and XSMLL communicate to each other. In the other words, the memory map between these two modules is pictured in this section as well. Table 11-1 shows associated register with their addresses and descriptions.

Table 11-1- Register maps between XSMLL and Software

Register Name	Address	Size	Type	Description
Module Ctrl Registers				
XSMLL_AP_CTRL	0xB000_0000 + 0x00	32	Rd/Rw	Register to enable XSMLL, check IDLE flag and check
Status Registers				
VersionReg_XSMLL	0xB000_0000 + 0x10	32	ro	Version Register
Status1Reg_XSMLL	0xB000_0000 + 0x18	64	ro	XSMLL Status Register #1
Status2Reg_XSMLL	0xB000_0000 + 0x24	64	ro	XSMLL Status Register #2
DebugNode1Reg_XSMLL	0xB000_0000 + 0x30	64	ro	XSMLL Debug Register #1
DebugNode2Reg_XSMLL	0xB000_0000 + 0x3C	64	ro	XSMLL Debug Register #2
Control Registers				
OpcodeReg_XSMLL	0xB000_0000 + 0x48	8	Rd/Wr	This field specifies the type of operation for XSMLL mod-
Arg1Reg_XSMLL	0xB000_0000 + 0x50	64	Rd/Wr	One of operand of desired XSMLL instruction
Arg2Reg_XSMLL	0xB000_0000 + 0x5C	64	Rd/Wr	One of operand of desired XSMLL instruction
ReturnReg_XSMLL	0xB000_0000 + 0x68	64	Rd/Wr	The result/return value of desire XSML instruction
Interrupt Registers				
IrqMskReg_XSMLL	0xB000_0000 + 0x74	32	Rd/Wr	XSMLL Mask Interrupt Register
IrqPndReg_XSMLL	0xB000_0000 + 0x7C	32	Rd/Wr	XSMLL Pending Interrupt Register
Test RX Registers				
TestRxBuffer1Reg_XSMLL	0xB000_0000 + 0x84	64	Rd/Wr	To test the received raw message
TestRxBuffer2Reg_XSMLL	0xB000_0000 + 0x90	64	Rd/Wr	To test the received raw message
TestRxBuffer3Reg_XSMLL	0xB000_0000 + 0x9C	64	Rd/Wr	To test the received raw message
TestRxBuffer4Reg_XSMLL	0xB000_0000 + 0xA8	64	Rd/Wr	To test the received raw message
TestRxBuffer5Reg_XSMLL	0xB000_0000 + 0xB4	64	Rd/Wr	To test the received raw message
TestRxBuffer6Reg_XSMLL	0xB000_0000 + 0xC0	64	Rd/Wr	To test the received raw message
XSMLL RDMA				
DmaStart	0xB000_0000 + 0xCC	64	Rd/Wr	The start address of RDMA zone
DmaEnd	0xB000_0000 + 0xD8	64	Rd/Wr	The end address of RDMA zone
Dest	0xB000_0000 + 0xE4	64	Rd/Wr	The destination address
Initialization Registers				
GlobMemStartReg_XSMLL	0xB000_0000 + 0xF0	64	Rd/Wr	Start address of global memory region
GlobMemEndReg_XSMLL	0xB000_0000 + 0xFC	64	Rd/Wr	End address of global memory region
GlobMemFrameSiz-	0xB000_0000 + 0x108	64	Rd/Wr	Global Memory size in bytes
XRQStartReg_XSMLL	0xB000_0000 + 0x114	64	Rd/Wr	Start address of XRQ memory region
XRQEndReg_XSMLL	0xB000_0000 + 0x120	64	Rd/Wr	End address of XRQ memory region
XRQFrameSizeReg_XSMLL	0xB000_0000 + 0x12C	64	Rd/Wr	XRQ memory size in bytes

1.1.1 Module Control Registers

1.1.1.1 XSMLL_AP_CTRL

Register Name: XSMLL_AP_CTRL
Address: 0xB000_0000 + 0x00
Size: 32 bits
Type: wr/rd

Field Name	Bit(s)	Init Value	type	Description
Ap_start	0	0	Read/Write	To start the XSMLL module, user can write '1' value.
Ap_done	1	1	read	If XSMLL complete all its task without any internal problem this flag will be enabled.
Ap_idle	2	1	read	If XSMLL is in Idle state, this flag is enabled.
Ap_ready	3	1	read	If XSMLL is ready to be going to start, this flag is enabled.

1.1.2 Status Registers

1.1.2.1 VersionReg_XSMLL

Register Name: VersionReg_XSMLL
Address: 0xB000_0000 + 0x10
Size: 32 bits
Type: ro

Field Name	Bit(s)	Init Value	Description
Date	7:0	0x00	BCD format representing the Date
Month	15:8	0x00	BCD format representing the Month
Year	23:16	0x00	BCD format representing the Year
Bit_Version	31:24	0x01	Bit Stream Version

1.1.2.2 Status1Reg_XSMLL

Register Name: Status1Reg_XSMLL
Address: 0xB000_0000 + 0x18
Size: 64 bits
Type: ro

Field Name	Bit(s)	Init Value	Description	
XSMLL_Busy	0	0b0	0b0:	XTQ is Idle
XWT_Empty	1	0b1	0b0:	XWT is empty
XWT_Full	2	0b0	0b0:	XWT is not full
XPLQ_Empty	3	0b1	0b0:	XPLQ is empty
XPLQ_Full	4	0b0	0b0:	XPLQ is not full
XTQ_Empty	5	0b1	0b0:	XTQ is empty
XTQ_Full	6	0b0	0b0:	XTQ is not full
XCRT_Empty	7	0b1	0b0:	XCRT is empty
XCRT_Full	8	0b0	0b0:	XCRT is not full
XFFQ_Empty	9	0b1	0b0:	XFFQ is empty
XFFQ_Full	10	0b0	0b0:	XFFQ is not full
XPTQ_Empty	11	0b1	0b0:	XPTQ is empty
XPTQ_Full	12	0b0	0b0:	XPTQ is not full
XNFFT_Empty	13	0b1	0b0:	XNFFT is empty
			0b0:	XNFFT is not full
XNI_Empty	15	0b1	0b0:	XNI is empty
XNI_Full	16	0b0	0b0:	XNI is not full
Node1_Req	17	0b0	0b0:	when neighbor Node1 has sent no request
Node2_Req	18	0b0	0b0:	when neighbor Node1 has sent no request
Node3_Req	19	0b0	0b0:	when neighbor Node1 has sent no request
Node4_Req	20	0b0	0b0:	when neighbor Node1 has sent no request
XTQ_State	22:21	0b00	0b00:	Waiting State
XTQ_Req	23	0b0	0b0:	No request has been generated to other nodes

1.1.2.3 Status2Reg_XSMLL

Register Name: Status2Reg_XSMLL
Address: 0xB000_0000 + 0x24
Size: 64 bits
Type: ro

Field Name	Bit(s)	Init Value	Description
Reserved	63:0	0x0000_0000_0000_0000	Reserved

1.1.2.4 DebugNode1Reg_XSMLL

Register Name: DebugNode1Reg_XSMLL
Address: 0xB000_0000 + 0x30
Size: 64 bits
Type: ro

Field Name	Bit(s)	Init Value	Description
Reserved	63:0	0x0000_0000_0000_0000	Reserved

1.1.2.5 DebugNode2Reg_XSMLL

Register Name: XSMLL_DebugNode2
Address: 0xB000_0000 + 0x3C
Size: 64 bits
Type: ro

Field Name	Bit(s)	Init Value	Description
Reserved	63:0	0x0000_0000_0000_0000	Reserved

1.1.3 Control Registers

1.1.3.1 OpcodeReg_XSMLL

Register Name: OpcodeReg_XSMLL
Address: 0xB000_0000 + 0x48
Size: 8 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
OpcodeReg_XSMLL	7:0	0x00	0x01: XCTRL
Reserved	31:8	zero	Reserved

1.1.3.2 Arg1Reg_XSMLL

Register Name: Arg1Reg_XSMLL
Address: 0xB000_0000 + 0x50
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
Arg1Reg_XSMLL	63:0	0x0000_0000_0000_0000	The input arguments of desired instruction

1.1.3.3 Arg2Reg_XSMLL

Register Name: Arg2Reg_XSMLL
Address: 0xB000_0000 + 0x5C
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
Arg2Reg_XSMLL	63:0	0x0000_0000_0000_0000	The input arguments of desired instruction

1.1.3.4 ReturnReg_XSMLL

Register Name: ReturnReg_XSMLL
Address: 0xB000_0000 + 0x68
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s) Offset	Init Value	Description
ReturnReg_XSMLL	63:0	0x0000_0000_0000_0000	The return value of desired instruction (if existed)

1.1.4 Interrupt Registers

1.1.4.1 IrqMskReg_XSMLL

Register Name: IrqMskReg_XSMLL
Address: 0xB000_0000 + 0x74
Size: 32 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
Reserved	31:1	zero	Reserved, writes are ignored, read is zero
IrqMsgReg_XSMLL	0	0b0	0b0: Interrupt for return value is enabled

1.1.4.2 IrqPndReg_XSMLL

Register Name: IrqPndReg_XSMLL
Address: 0xB000_0000 + 0x7C
Size: 32 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
Reserved	31:1	zero	Reserved, writes are ignored, read is zero
IrqPndReg_XSMLL	0	0b0	Once return value for desired instruction is ready, this bit is set by HW. This

1.1.5 Test RX Registers

1.1.5.1 TestRxBuffer1Reg_XSMLL

Register Name: TestRxBuffer1Reg_XSMLL
Address: 0xB000_0000 + 0x84
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
Reserved	63:0	0x0000_0000_0000_0000	Reserved

1.1.5.2 TestRxBuffer2Reg_XSMLL

Register Name: TestRxBuffer2Reg_XSMLL
Address: 0xB000_0000 + 0x90
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
Reserved	63:0	0x0000_0000_0000_0000	Reserved

1.1.5.3 TestRxBuffer3Reg_XSMLL

Register Name: TestRxBuffer3Reg_XSMLL
Address: 0xB000_0000 + 0x9C
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
Reserved	63:0	0x0000_0000_0000_0000	Reserved

1.1.5.4 TestRxBuffer4Reg_XSMLL

Register Name: TestRxBuffer4Reg_XSMLL
Address: 0xB000_0000 + 0xA8
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
Reserved	63:0	0x0000_0000_0000_0000	Reserved

1.1.5.5 TestRxBuffer5Reg_XSMLL

Register Name: TestRxBuffer5Reg_XSMLL
Address: 0xB000_0000 + 0xB4
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
Reserved	63:0	0x0000_0000_0000_0000	Reserved

1.1.5.6 TestRxBuffer6Reg_XSMLL

Register Name: TestRxBuffer6Reg_XSMLL
Address: 0xB000_0000 + 0xC0
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
Reserved	63:0	0x0000_0000_0000_0000	Reserved

1.1.6 XSMLL RDMA

1.1.6.1 DmaStart

Register Name: DmaStart
Address: 0xB000_0000 + 0xCC
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
DmaStart	63:0	0x0000_0000_0000_0000	The start address of dma zone in DDR

1.1.6.2 DmaEnd

Register Name: DmaEnd
Address: 0xB000_0000 + 0xD8
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
DmaEnd	63:0	0x0000_0000_0000_0000	The end address of dma zone in DDR

1.1.6.3 Dest

Register Name: Dest
Address: 0xB000_0000 + 0xE4
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
Dest	63:0	0x0000_0000_0000_0000	The destination address of DMA performance

1.1.7 Initialization Registers

1.1.7.1 GlobMemStartReg_XSMLL

Register Name: GlobMemStartReg_XSMLL
Address: 0xB000_0000 + 0xF0
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
GlobMemStar-	63:0	0x0000_0000_0000_0000	The start address of global memory region in DDR

1.1.7.2 GlobMemEndReg_XSMLL

Register Name: GlobMemEndReg_XSMLL
Address: 0xB000_0000 + 0xFC
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
GlobMemEn-	63:0	0x0000_0000_0000_0000	The End address of global memory region in DDR

1.1.7.3 GlobMemFrameSizeReg_XSMLL

Register Name: GlobMemFrameSizeReg_XSMLL
Address: 0xB000_0000 + 0x108
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
GlobMemFrameSiz-	63:0	0x0000_0000_0000_0000	The Size of global memory region in DDR in bytes

1.1.7.4 XRQStartReg_XSMLL

Register Name: XRQStartReg_XSMLL
Address: 0xB000_0000 + 0x114
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
XRQStartReg_XSMLL	63:0	0x0000_0000_0000_0000	The start address of XRQ table region in DDR in bytes

1.1.7.5 XRQEndReg_XSMLL

Register Name: XRQEndReg_XSMLL
Address: 0xB000_0000 + 0x120
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
XRQEndReg_XSMLL	63:0	0x0000_0000_0000_0000	The End address of XRQ table region in DDR in bytes

1.1.7.6 XRQFrameSizeReg_XSMLL

Register Name: XRQFrameSizeReg_XSMLL
Address: 0xB000_0000 + 0x12C
Size: 64 bits
Type: Rd/Wr

Field Name	Bit(s)	Init Value	Description
XRQFrameSiz-	63:0	0x0000_0000_0000_0000	The frame size of XRQ table region in DDR in bytes

1.2 Packet format between XSMLL and NIC and timing

OPERATION	INPUT(bit)	OUTPUT(bit)	TIME (Clock Cycle)
Fifo_enqueue	64(Fp)	64 (Fp)	1
Fifo_dequeue	64(Fp)	64 (Fp)	1
Write_register_in BRAM(XWT)	128(fp,ip,sc)	64(ack)	1
Read_register_in_BRAM(XWT)	64(Fp)	64(FP return)	11
Write_frame_in_globalMem (DDR4)	64(Fp)	64(ack)	
Read_frame_in_globalMem (DDR4)	64(Fp)	64(Fp)	