H2020 FRAMEWORK PROGRAMME
ICT-01-2014: Smart Cyber-Physical Systems

PROJECT NUMBER: 645496

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

D4.1 – Programming Model Extensions

Due date of deliverable: 31st January 2016
Actual Submission: 9th February 2016

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

Lead contractor for the deliverable: BSC

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	Organization	Change History
0.1	18.01.2016	Xavier Martorell	BSC	v0.1
0.3	21.01.2016	Paolo Gai	EVI	v0.3
0.5	25.01.2016	Daniel Jimenez	BSC	v0.5
0.7	27.01.2016	Dimitris Theodoropoulos	FORTH	v0.7
1.0	06.01.2016	Xavier Martorell	BSC	v1.0
1.1	08.02.2015	Jem Macy, Roberto Giorgi	UNISI	Final version

Release Approval

Name	Role	Date
Xavier Martorell	WP Leader	08.02.2015
Roberto Giorgi	Project Coordinator for formal deliverable	09.02.2015

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

Daniel Jiménez, Carlos Alvarez, Xavier Martorell
CS Department
BSC – AXIOM

Paolo Gai
CS Department
Evidence – AXIOM

Dimitris Theodoropoulos, Dionisios Pnevmatikatos
CS Department
FORTH - AXIOM

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

TABLE OF CONTENTS

TABLE OF FIGURES –

GLOSSARY

ACP – Accelerator Coherency Port: an ARM AXI 64-bit port that can be used for DMA data transfers to the Zynq FPGA. It is suitable when there is a need to keep coherency with the memory hierarchy on the processor side

AXI – a proprietary protocol for buses introduced by ARM Ltd

AXIOM-acc – an FPGA accelerated system that performs a given function

AXIOM-arch – the architecture of an AXIOM (module or) board

AXIOM-core – the cores where the computations run in an AXIOM board

AXIOM-fpga – the programmable logic part in an AXIOM board

AXIOM-link – the interconnects that permits board-to-board communication in AXIOM

Bitstream – the binary code used for configuring the PL

BRAM – Block-RAM: a fast RAM that is available in the FPGA slices (in smaller blocks)

CUDA – NVIDIA programming model for GPUs

Conduit – A software stub that connects GASNet to a given network protocol or programming model

Device – in this context: it is the physical system that runs a 'device-annotated' part of the code

DMA – Direct Memory Access: a separate master that can take over local memory transfers

eMMC – Embedded Multi Media Card

FPGA – Field Programmable Gate Array

FPGA device – a device implemented on the FPGA to accelerate a portion of a program. In this document, it is used as a synonym of accelerator

GASNet – Global Address Space over Network: is a language-independent, low-level networking layer that provides network independent communication primitives

GP – General-Purpose Port: an ARM AXI 32-bit port that can be used for DMA data transfers to the Zynq FPGA. It is suitable for short transfers or control operations

Infiniband – a high-performance (costly) NI

IP – Intellectual Property system (either hardware or software)

HP – High-Performance Port: an ARM AXI 64-bit port that can be used for DMA data transfers to the Zynq FPGA. It is suitable when there is no need to keep coherency with the memory hierarchy on the processor side

Mercurium – the OmpSs compiler

Nanos++ -- the OmpSs runtime

MGT – Multi-Gigabit Transceiver

MPI – Message Passing Interface: library for writing portable message-passing programs

PL – Programmable Logic: the purely FPGA part of a SoC like ZYNQ

PS – Processing System: the hardwired IPs of a FPGA-hybrid SoC like ZYNQ

NI – Network Interface

OpenCL – Khronos group programming model for heterogeneous architectures

OmpSs – Extension of OpenMP programming model to support task dataflow programming

OmpSs@FPGA – FPGA extension of OmpSs

OmpSs@Cluster – Cluster extension of OmpSs

PHY – the physical implementation of the network interface

QSPI – Quad Serial Peripheral Interface

RDMA – Remote DMA: a DMA that can work from one computer to another computer

SoC – System on Chip

USB OTG – Universal Serial Bus On The Go

XSMLL – (pronounced X-SMALL) eXtended Shared-Memory Low-Level API (see D7.1)

X-Thread – a self-contained thread that can be distributed across boards through XSMLL

ZYNQ -- A System-on-Chip commercialized by XILINX, which includes FPGA and CPUs

Executive summary

This document describes the programming model extensions proposed for programming the AXIOM boards with OmpSs, the design of the support for the FPGA devices, the distributed cluster environment, and the communication layer.

Following the existing OmpSs support for CUDA and OpenCL, in the AXIOM project we map the OmpSs *target device* extensions onto heterogeneous nodes including accelerators based on FPGAs. Code annotated with the *target device (fpga)* and *task* directives will be automatically offloaded by the Mercurium compiler onto separate files, and compiled with the FPGA tools to build the accelerator for the FPGA (i.e.,. on the programmable logic part of a System-on-Chip as the Zynq or in general on the AXIOM-fpga). These tasks will be spawned by the Nanos++ runtime system to run on the FPGA and or on the cores (i.e., the ARM-A9 cores in case of the Zynq SoC or in general on the AXIOM-cores). Nanos++ will use a custom DMA library presented in this deliverable to take care of data transfers between the host memory and the FPGA devices.

The support for distributed cluster environments is based on the Nanos++ runtime system and its connection to the communication layer, through specific communications software. In this project we are considering the use of common tools, like MPI or GASNet, and also implementing our specific approach based on the XSMLL infrastructure (cf. deliverable D7.1).

An initial evaluation of the low level communication mechanisms to transfer data to and from the FPGA devices (i.e., to/from the AXIOM accelerators) is shown. These results will be used during the implementation of the support for the OmpSs extensions for FPGAs to decide the specific mechanism to use in each situation.

1 Introduction

1.1 Document structure

This deliverable is organized as follows

- Section 2 describes the programming model extensions proposed for programming the AXIOM boards with OmpSs.
- Section 3 describes the runtime support planed for distributed environments based on the connection of several AXIOM boards.
- Section 4 presents the design of the communication layer used to connect AXIOM boards and build the distributed system.
- Section 5 shows the evaluation of the low-level FPGA data transfer mechanisms to support OmpSs.

1.2 Relation to other deliverables

This document completes the description of the programming model extensions and their support, presented in the document "MS41 – Definition of the Programming Model Extensions", and designed to run on the AXIOM platform presented in the document "D6.1 – Technical specifications of AXIOM board".

1.3 Tasks involved in this deliverable

This deliverable is the result of the work developed in tasks:

- T4.1: Requirement definition for the programming model extensions
- T4.2: OmpSs programming model extensions
- T5.3: Parallel programming library
- T6.5: System interconnect

2 Programming Model Extensions

In this section, we describe the OmpSs Programming Model [3], the extensions planned for OmpSs to spawn tasks in the FPGA-device (an AXIOM accelerator), and the extensions needed to support the cluster version. This support has been designed and partially developed during the first year of the AXIOM project.

2.1 Introduction to the OmpSs Programming Model

The OmpSs Programming Model supports the execution of heterogeneous tasks written both in OpenCL and CUDA, and in the distributed cluster version. Both OpenCL and CUDA options require the programmer to provide the OpenCL or CUDA code, and use the *target clauses* to move the data to the associated accelerator. In the AXIOM project, we are using the same technique to spawn tasks to the FPGA, provided there is a compiler to generate the FPGA *bitstream* implementing the task, from C/C++ code, or there is an existing *bitstream* available with a known interface to access data. For ex-

ecuting tasks in the cluster version, the programmer needs to specify the task as plain C/C++ code. Execution on the OmpSs@cluster version automatically allows the runtime system to spawn tasks remotely.

The programming model will allow to parallelize applications on the AXIOM cluster, and spawn tasks on the FPGAs available on each board. Using OmpSs@cluster with FPGAs support, programmers will be able to express two levels of parallelism.

A first level of parallelism will be targeted to the AXIOM-cores, i.e. the cores that are available on the AXIOM-board (e.g., the ARM-A9 cores in the case of a Zynq SoC, c.f. Deliverable D6.1). Tasks at this level will be spread across the AXIOM boards, as if they would be executed on an SMP machine (see Sections 3 and 4 for the distributed cluster support).

A second level of task parallelism will be expressed through the OmpSs extensions targeting the FPGAs (see below, Section 2.2).

The OmpSs Programming Model is based on two main components and some additional tools:

- The Mercurium compiler [8] takes the source code as specified by the programmer and understands the OmpSs directives to transform the code to run on heterogeneous platforms, including OpenCL and CUDA accelerators. In this project, the compiler will be extended to also support FPGA-based accelerators.
- The Nanos++ runtime system, which is the responsible to manage and schedule parallel tasks, respecting their dependences, transferring the data needed to/from the accelerators, when needed and the lower-level interactions (cf. Section 3).
- Additionally, OmpSs can use the Extrae tool to generate execution traces that can be later visualized with the Paraver tool, and analyze their behavior. Both Extrae and Paraver are also developed at BSC. This complements the evaluation tools developed in WP7 (see Deliverable D7.1).

Figure 1 shows the existing syntax used in the target and task directives in OmpSs. Task directive clauses act as follows:

- *in*, *out*, and *inout* clauses allow the specification of the input, output only, and input/output ranges of data that are to be used by the task. This way, the Nanos++ runtime system takes care to manage the task dependences before and after executing this task.
- *concurrent* and *commutative* allow the specification of variants of *inout* dependences for *inout* data. Concurrent means that data is accessed with an explicit synchronization inside the task, so that the runtime system can exploit tasks in parallel. Commutative indicates that the tasks can be executed sequentially, but in any order (possibly different from the creation order).

```
#pragma omp target device ({ smp | opencl | cuda }) \
        [ implements ( function_name )] [ copy_deps | no_copy_deps ] \
        [ copy_in ( array_spec ,...)] [ copy_out (...)] [ copy_inout (...)]    \
        [ndrange (dim, …)] [shmem(...) ] [file(name)] [name(name)]
#pragma omp task [ in (...)] [ out (...)] [ inout (...)] [ concurrent (...)] [ commutative (...)] \
                        [ priority (P) ] [ label (name) ] \
                        [ shared(...)][private(...)][firstprivate(...)][default(...)] \
                        [untied][final (expression)][if (expression)]
 {code block or function prototype}
```

Figure 1 – OmpSs target and task directives

- *priority(P)* is used to specify the importance of the task. It is a hint to the scheduler, that may try to execute higher priority (P) tasks before lower priority tasks, always respecting the dependences between them.
- *label(name)* provides a name for the task, specifically for the instrumentation tools.
- *shared*, *private*, *firstprivate*, Default are clauses specifying the data sharing for the listed variables. They are compatible with the same clauses in OpenMP.
- *untied* means that the task can change processor after blocking in a *taskwait*. By default tasks are tied, and they execute always in the same processor after a *taskwait*.
- *final(expression)* indicates that this task will not create inner tasks.
- *if(expression)* indicates if the task can be deferred or not. If the expression evaluates to True, the task will be a regular task. If the expression evaluates to False, the task will be created and executed immediately by the same thread.

Target directive clauses act as follows:

- *device* specifies the specific device this task is to run on. “smp” means the host cores, “opencl” indicates an OpenCL-capable device, and “cuda”, a CUDA-capable device.
- *implements(function-name)* indicates that this task is equivalent to the function indicated, possibly for a different device, and the runtime system is free to schedule either one in the available device.
- *copy_deps* / *no_copy_deps*, indicate if the dependences listed in the task directive should also be kept consistent / or not with the accelerator.
- *copy_in*, *copy_out*, *copy_inout* list additional data that should be kept consistent with the accelerator.
- ndrange, shmem, file, and name clauses provide additional arguments for OpenCL and CUDA target tasks, which are not relevant for AXIOM.

Tasks can be associated to a code block or to a function. In the case of inline code annotations, tasks targeting the host cores are outlined as new functions by the Mercurium compiler and spawned as tasks to be executed on the SMP host. Tasks targeting the FPGA will be outlined by Mercurium onto separate files, and compiled through the Xilinx Vivado HLS in order to generate VHDL, and later through the Vivado tool to generate the bitstream for the FPGA [11]. Invoking tasks on the FPGA will be done by the Nanos++ runtime system by sending the data needed, executing the FPGA device, and getting the resulting data back to the host memory.

In the case of function interfaces annotated with the target device (fpga) directive, the invokations of such functions will be done in the FPGA device using the same parameter passing.

2.2 OmpSs extensions for the FPGAs

OmpSs needs to be extended to support the Zynq chip with the FPGA selected in the AXIOM project. The extensions to provide support for these chips in the Mercurium compiler are:

- To incorporate a new target device named “fpga”: in addition to the current *smp*, *cuda* and *opencl* devices, the “fpga” device will cause the Mercurium compiler to understand that the function annotated is to be compiled with the Xilinx Vivado HLS compiler, for the FPGA, in order to generate the bitstream.

With this extension, the compiler will generate code for the runtime system specifying the tasks that should be run in the FPGA device. The Nanos++ runtime system will also need to be extended, in the following way:

- Support to spawn tasks in the FPGA device.
- Support for the target clauses related to data transfers:
  - Data-copy clauses (copy_in, copy_out, copy_inout): for the FPGA target, they will trigger the data transfer of the data specified to/from the FPGA device.
  - Dependence-copy clauses (copy_deps, no_copy_deps): for the FPGA target, they will indicate if, additionally, the data dependences specified in the associated task should be transferred or not, with the directionality associated in the dependence clauses.
- Support for data transfers to/from the FPGA. The Nanos++ runtime will invoke the services of the DMA library developed to transfer data in the FPGA environment.
- Include the FPGA device in the support of the *implements* clause in order to allow several implementations of tasks to be scheduled in the processors/devices available.

The DMA library interface [6] provides the means to allocate buffers to exchange data between the Linux kernel and the FPGA hardware. In the current prototype, when the FPGA has been given the data to operate with, the IP kernel is automatically started, and after finishing, the results can be read from it. The current version of the interface is shown in the following tables. It may still change as the work on Nanos++ and the Linux driver proceeds in the project.

Method parameter	Description
Initializes the DMA userspace library and driver.	
`xdma_status  xdmaOpen (void);`	

Method parameter	Description
Cleans up the DMA userspace library and driver.	
`xdma_status  xdmaClose (void);`	

Method parameter		Description
uint32_t * num_devices	[out]	Number of FPGA-devices present in the system
Returns the number of devices present in the system.		

```
xdma_status  xdmaGetNumDevices (uint32_t * num_devices);
```

Method parameter	Description
uint32_t entries	[in] Number of FPGA-device handles that can be stored in the 'devices' array
xdma_device * devices	[out] Array to hold the device handles, at least of size *entries*
uint32_t * num_devices	[out] Actual number of device handles returned
Returns the device handles for the devices present in the system.	
`xdma_status  xdmaGetDevices (uint32_t entries, xdma_device * devices, uint32_t * num_devices);`	

Method parameter	Description
xdma_device device	[in] FPGA-device that will be connected to the newly allocated channel
xdma_dir direction	[in] Direction of the channel (XDMA_TO_DEVICE, XDMA_FROM_DEVICE)
xdma_channel_flags flags	[in] Channel flags (currently unused)
xdma_channel * channel	[out] Handle to the newly opened channel
Open a device channel. Each device can have one input and 1 output channels used to send/receive data.	
`xdma_status  xdmaOpenChannel (xdma_device device, xdma_dir direction, xdma_channel_flags flags, xdma_channel * channel);`	

Method parameter	Description
xdma_channel * channel	[in,out] DMA channel to be closed
Closes a DMA channel and releases its resources.	
`xdma_status  xdmaCloseChannel (xdma_channel * channel);`	

Method parameter	Description
uint8_t ** buffer	[out] Pointer to allocated buffer
xdma_buf_handle * handle	[out] DMA buffer handle
uint32_t len	[in] Buffer length in bytes
Allocates a buffer in kernel space to support data transfers to a DMA device. The buffer will be pinned.	
`xdma_status  xdmaAllocateKernelBuffer (uint8_t ** buffer, xdma_buf_handle  handle, uint32_t len);`	

Method parameter	Description
void * buffer	[in] Address of the buffer to be freed
xdma_buf_handle handle	[in] Buffer handle to be freed
Free a pinned buffer allocated in kernel space and unmap the region from user space.	
`xdma_status  xdmaFreeKernelBuffer (void * buffer, xdma_buf_handle handle);`	

Method parameter	Description
xdma_buf_handle buffer	[in] Buffer handle
uint32_t len	[in] Buffer length
uint32_t offset	[in] Transfer offset
xdma_xfer_mode mode	[in] Transfer mode (XDMA_SYNC, XDMA_ASYNC)
xdma_device device	[in] DMA device to be used to transfer data
xdma_channel channel	[in] DMA channel to use
xdma_transfer_handle * handle	[out] Handle identifying the DMA transfer, used with XDMA_ASYNC
Submits a pinned buffer allocated in kernel space for a DMA transfer.	
`xdma_status xdmaSubmitKBuffer (xdma_buf_handle buffer, uint32_r len, uint32_t offset, xdma_xfer_mode mode, xdma_device device, xdma_channel channel, xdma_transfer_handle * transfer);`	

Method parameter	Description
void * buffer	[in] Buffer to be transferred
uint32_t len	[in] Buffer length
xdma_xfer_mode mode	[in] Transfer mode (XDMA_SYNC, XDMA_ASYNC)
xdma_device device	[in] DMA device to be used to transfer data
xdma_channel channel	[in] DMA channel to use
xdma_transfer_handle * handle	[out] Handle identifying the DMA transfer, used with XDMA_ASYNC
Submits a user allocated buffer (i.e. using malloc) to be transferred through DMA.	
`xdma_status xdmaSubmitBuffer (void * buffer, uint32_r len, xdma_xfer_mode mode, xdma_device device, xdma_channel channel, xdma_transfer_handle * transfer);`	

Method parameter	Description
xdma_transfer_handle handle	[in] DMA transfer handle to be checked
Tests the status of a DMA transfer (finished, pending, or in error).	
`xdma_status xdmaTestTransfer (xdma_transfer_handle handle);`	

Method parameter	Description
xdma_transfer_handle handle	[in] DMA transfer handle to wait for
Waits for a transfer to finish (finished, pending, or in error).	
`xdma_status xdmaWaitTransfer (xdma_transfer_handle handle);`	

Method parameter	Description
xdma_transfer_handle handle	[in,out] DMA transfer handle to be released
Releases the data structures associated with a DMA transfer.	
`xdma_status xdmaReleaseTransfer (xdma_transfer_handle * handle);`	

2.3 Support for OmpSs@cluster

The OmpSs@Cluster [2] infrastructure uses a communication layer to launch tasks in remote nodes. Task descriptors and data travel on the communication layer. In our current implementation, this layer is GASNet [1], usually running on top of MPI [4]. The different alternatives to implement this approach that are currently under study in our project are presented in Section 3. The underlying communications layer is presented in Section 4.

3 Support for distributed environments

This Section provides a brief explanation of the support needed to integrate the OmpSs programming model into the networking support provided by the AXIOM boards.

Note: Further details will be specified in Deliverable D5.2 (due at month m18 in the project, i.e. 6 months after this deliverable), where we will describe in more detail the aspects related to the remote memory access.

The **main goal** that drives the choices described below is to provide an efficient and possibly lightweight implementation of the infrastructure that allows OmpSs to use the AXIOM networking infrastructure.Given the complexity of the framework, we realized that there are a number of possible ways for implementing this kind of support, each one requiring integration at different levels of the OmpSs toolchain. For this reason, during the project we are going to explore, implement and evaluate different options, in order to choose the ones that best fit the project objectives. The options we are exploring (described in more detail below) are (note the option numbers are the same depicted in Figure 2):

- Integration with OmpSs@cluster as a GASNet conduit based directly on the FORTH network interface (option 1).
- Integration with OmpSs@cluster as a GASNet conduit based on XSMLL [5][7] (option 2a);
- Integration with OmpSs as a Nanos++ [8] plugin based on XSMLL [5][7] (option 2b);
- Integration directly below Mercurium [8] (option 3);

Please note that the outcome of the evaluation is likely not to provide a single "winner", because each solution has its advantages and drawbacks. As we will discuss below, for example, we expect option 1 to be a pure software implementation over the FORTH network interface (potentially slower but with lower FPGA requirements), whereas option 2b will take advantage of the XSMLL dataflow approach [5][7] (thus potentially faster but with higher FPGA requirements).

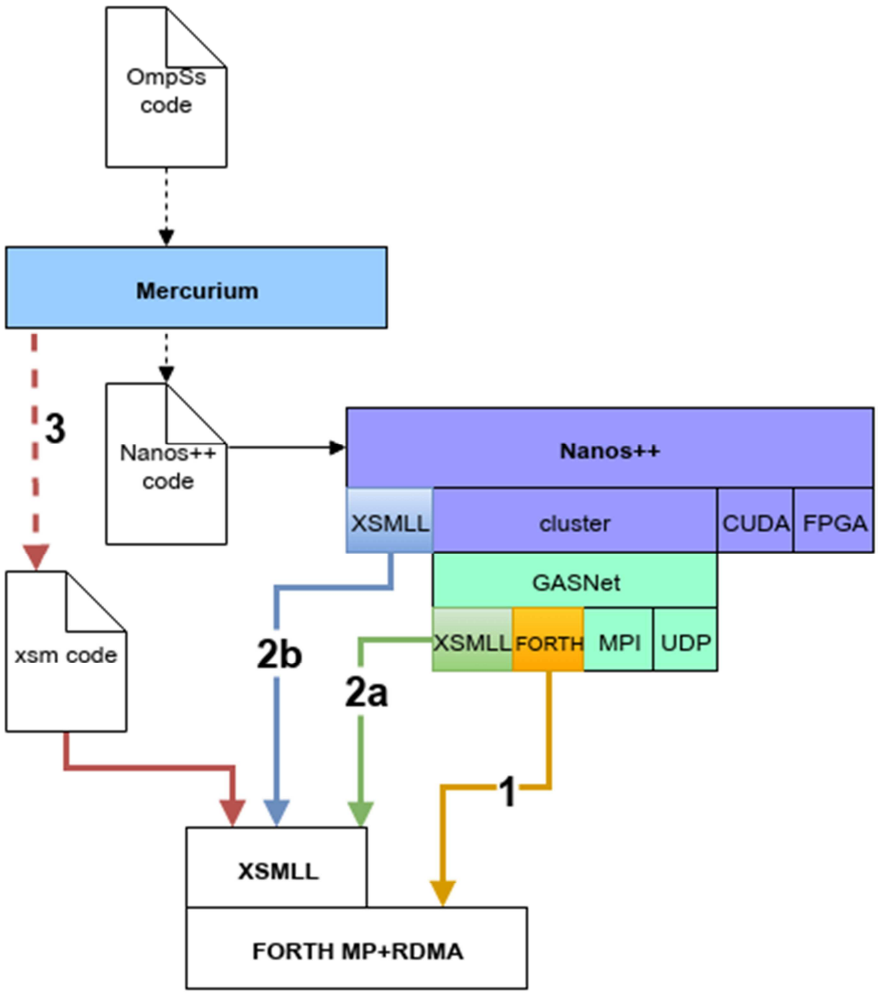

Figure 2– Architectural view related to the support of distributed environments.

Figure 2 shows an architectural view of the various options under exploration. In particular, we highlight the following main architectural components:

- The Mercurium source to source compiler;
- The Nanos++ Runtime library;
- A set of Nanos++ "targets". The interesting ones for this Section are the "cluster" target and the new "XSMLL" target developed in the context of the AXIOM Project (see later);
- The GASNet layer used for the implementation of the "cluster" target;
- The GASNet "conduits", which are basically plugins that enable GASNet to run on different transport (network) layers. Please note in particular MPI and UDP, which are the ones on which OmpSs@cluster typically works, and in addition to these, please note the new XSMLL and FORTH conduits in development in the context of the AXIOM Project (see later);
- The XSMLL Layer, providing fine-grained task dataflow;
- The FORTH Message Passing / RDMA networking support, implemented in FPGA and available to the ARM cores of the Xilinx Zynq microcontroller used in the AXIOM boards.

We can now describe shortly the options we are considering in the AXIOM Project. To ease the understanding of the reader, we are presenting them in reverse order, because in this way each step will be "incremental" to the previous one.

Option 1: GASNet conduit based directly on the AXIOM network interface (FORTH)

The straightforward way of integrating OmpSs with the AXIOM-link interconnect by FORTH is to operate at the level of OmpSs@cluster, providing a so called "conduit" of GASNet. A "conduit" of GASNet is a plugin of GASNet that allows the implementation of the full GASNet API on top of a "transport layer".

Currently, there are various transport layers that can be used, starting from the simple UDP layer, to MPI, Infiniband and others.

The approach here is to re-implement the GASNet conduit API using the Linux Kernel Driver in development in Task 5.1. The implementation will take advantage of the RDMA support provided by the FORTH for the AXIOM-link interconnect in a way similar to what done in the Infiniband conduit.

Option 2a: GASNet conduit based on XSMLL

This option is similar to Option 1, but it tries to use the XSMLL [5][7] Dataflow layer to provide a proper transport layer for the GASNet conduit.

After an initial implementation prototype done on the COTSon simulator [9][10] was developed as part of the WP7 activities (see deliverable D7.1), we foresee the fact that this option will probably not be the most efficient one. The reason for this is related to the semantic abstractions which are present at the various levels.

In particular, both the Nanos++ plugin interface and the XSMLL layers exposes a "task" semantic. On the other hand, the OmpSs@Cluster has the role to transform the Nanos++ plugin "task" semantic into something more manageable by a network interface. OmpSs@Cluster does that using the GASNet layer, which exports an "active message" semantic to the upper layer. OmpSs@cluster, in practice, transforms the "task" semantic into a "active message" semantic, leaving the rest to GASNet.

At the GASNet layer, the "active message" semantic exported at the top is then managed internally, and then implemented using simpler conduit interfaces. The conduit interfaces are using a sort of message exchanges using asynchronous message sends and synchronous message receive functionality.

As we can see, the initial "task" semantic, available at the Nanos++ plugin level, has been transformed into a message passing approach, available at the GASNet conduit level. To help this, partners UNISI and EVI are exploring the possibility to add a message passing extension to XSMLL (part of this work is developed in WP5). The appropriate granularity and operations are under exploration.

For the above reasons, we are not confident that this option will provide the best results, but was considered since it was a possibility apparently reasonable and simple.

Option 2b: Nanos++ plugin based on XSMLL

Given the limitations of option 2a, we decided to evaluate option 2b, which tries to avoid the semantic incompatibility between the various layers.

In particular, the idea is to develop an additional plugin to Nanos++, which will allow the direct usage of the XSMLL layer. This approach is for sure the most promising one, since the semantic exported by XSMLL and required by Nanos++ are somehow similar and based on a "task" semantic.

Moreover, the XSMLL interface will provide the runtime support allowing the distribution of X-Threads on the various nodes allowing the possibility to early test the results using the COTSon [9][10] simulator.

Afterwards, the possibility to optimize the implementation of XSMLL in FPGA is going to provide a performance boost to the implementation.

We expect to implement an initial version of this option during the second year of the project. This initial version will support a subset of the constructs available on the OmpSs programming model (e.g., the first implementation will be without the "taskwait" construct).

Option 3: Mercurium integration with XSMLL

Another option, which could be available, stems from the idea that the XSMLL execution model could be directly integrated into Mercurium, in a way to provide direct XSMLL code generation of OmpSs code.

If on one hand this option seems to provide an increased performance thanks to the direct usage of the XSMLL implementation (which can be optimized in FPGA), on the other hand it needs some special care in the integration with the other "targets" of OmpSs (in particular the cluster and FPGA targets in Nanos++).

Moreover, additional study needs to be done in order to support data dependencies with the richness of options currently provided by the OmpSs framework thanks to the Nanos++ implementation. In particular, task dependencies in OmpSs can be specified using complex dependency constraints, whereas the XSMLL interface offers a method based on counters; an implementation of option 3 will have the (non-straightforward) need to map those two methods together.

For these reasons, after a first evaluation we plan not to implement option 3 during the AXIOM Project.

4 Communication layer

The AXIOM communication layer is responsible for the data transmission among all available nodes in the network. It consists of a software library that allows the host CPU of each node to send and receive control and RDMA messages via its Network Interface (NI) module. The tables below summarize the current version of the NI software function prototypes. They may still change as the work on the interface and Linux driver proceeds in the project.

Method parameter	Description
uint8_t srcNode	The local node Id that will send raw data
uint8_t dstNode	The remote node id that will receive the raw data
uint32_t data	Data to be sent
Sends raw data to a remote node. Returns a unique positive message id on success, -1 otherwise	
`uint8_t sendRawData (uint8_t srcNode, uint8_t dstNode, uint32_t data);`	

Method parameter	Description
uint8_t srcNode	The local node Id that sends data to a remote node
uint8_t dstNode	The remote node's id where data will be stored
uint32_t *lSrcAddr	The local address from where data will be transmitted
unit32_t *RdstAddr	The remote address where transmitted data will be stored
uint16_t payloadSize	Size of data to be sent in words
Stores data to a remote node's memory. Returns a unique positive message id on success, -1 otherwise	
`uint8_t RDMAwrite (uint8_t srcNode, uint8_t dstNode, uint32_t *lSrcAddr, uint32_t *rDstAddr, unit16_t payloadSize);`	

Method parameter	Description
uint8_t srcNode	The local node Id that requests data from a remote node
uint8_t dstNode	The remote node id that will send the requested data
uint32_t *rSrcAddr	The remote address from where data will be fetched
unit32_t *lDstAddr	The local address where fetched data will be stored
uint16_t payloadSize	Size of data to be fetched in words
Requests data from a remote node to be stored locally. Returns a unique positive message id on success, -1 otherwise	
`uint8_t RDMAreq (uint8_t srcNode, uint8_t dstNode, uint32_t *rSrcAddr, uint32_t *lDstAddr, unit16_t payloadSize);`	

Method parameter	Description
Reads the NI status register.	
`uint32_t readNIStatusReg();`	

Method parameter	Description
uint8_t msgId	The RDMA request id that is pending data
Reads the HW counter value associated with a specific RDMA request id.	
`uint32_t readNIHWCounter(unit8_t msgId);`	

Method parameter	Description
uint8_t regMask	The register mask to be used for the configuration.
Sets the control registers for enabling local transmission ACKs and/or the PHY loopback mode configuration.	
`void setNIRegister(unit8_t regMask);`	

Method parameter	Description
uint8_t nodeId	The Id assigned to this node.
Sets the id of a local node.	
`void setNodeId(unit8_t nodeId);`	

Method parameter	Description
--	--
Returns the local node id.	
`unit8_t nodeId getNodeId();`	

Method parameter	Description
uint8_t dstNode	Remote node id to setup its routing table
uint32_t *nodeRoutingTable	Routing table to be sent
Sets the routing table of a particular node.	
`void setRoutingTable(unit8_t dstNode,uint32_t *nodeRoutingTable);`	

Method parameter	Description
uint8_t dstNode	Remote node id to read its routing table
Returns the routing table of a node.	
`uint32_t *nodeRoutingTable getRoutingTable(unit8_t dstNode);`	

Method parameter	Description
uint8_t dstNode	Remote node id to update its routing table
uint16_t nodeIdToUpdate	Entry to update within the routing table
Updates the routing table of a node for a particular entry.	
`void updateRoutingTable (unit8_t dstNode, unit16_t nodeIdToUpdate);`	

Method parameter	Description
uint8_t dstNode	Remote node id to remove entry from its routing table
uint32_t *nodeIdToRemove	Node id that no longer belongs to the routing table of dstNode
Invalidates an entry of the dstNode routing table.	
`void deleteNodeFromRoutingTable(unit8_t dstNode,uint8_t nodeIdToRemove);`	

Method parameter	Description
uint8_t ifId	Online interface Id where the identification message will be sent
Sends an identification packet to the neighbor node connected to an IF, and receives its id.	
`uint8_t neighborId identifyNeighborNode(unit8_t ifId);`	

Method parameter	Description
Transmits the node's neighbor – interface pairs to the "master node".	
`void reportNeighbors();`	

Figure 3 illustrates the NI structure and how the software library can be used for its configuration from the host CPU. During network initialization, a network topology algorithm is executed on each node to identify its neighbor ones. Within each node, the topology algorithm will use the functions *setNodeId*, *identifyNeighbotNode* and *reportNeighbors*, to discover its neighbors and report them to the master network node. The latter will generate each node's routing table (used by the router module during packet relaying if required), which can be configured by the host CPU via the NI software library (*setRoutingTable*). We should note that in case a node is added / removed to the network, the NI software library provides methods (*deleteNodeFromRoutingTable*, *updateRoutingTable*) to update ac-

cordingly the NI routing table / remove completely a node if needed. Moreover, helper functions (*nodeId*, *getRoutingTable*), can provide debugging capabilities.

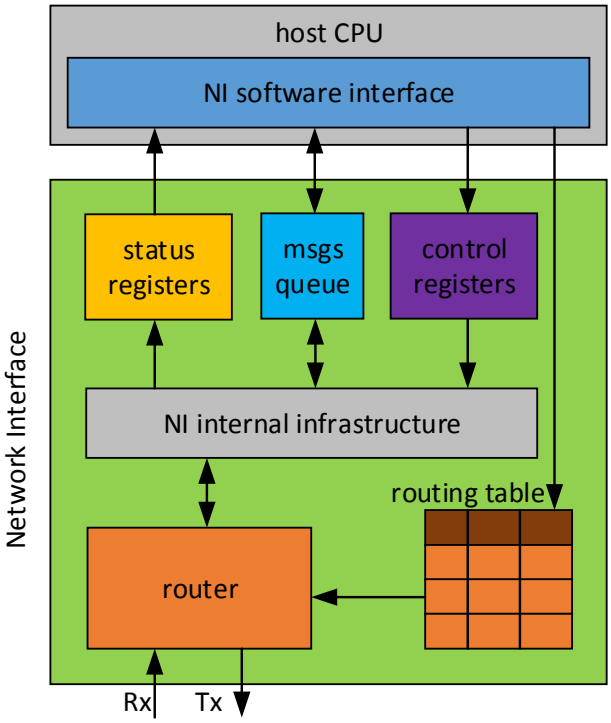

Figure 3 - Overview of the NI structure and its configuration from the host CPU via the software interface.

The NI employs a set of memory-mapped status and control registers. The status register is directly accessible by the NI library (*readNIStatusReg*) and the host CPU can read it to monitor various parameters, such as the DMA engine state (idle, busy), the messages queues fill status (empty, not empty, near full, etc), the PHY link state (connected, down), as well as the progress of on-going RDMA requests / writes (*readNIHWCounter*). The control registers are also directly accessible by the NI software (*setNIRegister*); the host CPU can write them to configure NI parameters, such as the node id, the PHY link loopback mode, and successful transmission notifications.

Finally, the NI employs a set of memory-mapped hardware queues, accessible by the NI software library. The CPU can use these queues to push asynchronous message transmissions, such as RDMA requests / writes (*RDMAreq*, *RDMAwrite*), and raw data transmissions (*sendRawData*). Also, it can check asynchronously for any received messages, and proceed to further processing if required.

5 Preliminary evaluations

In this section, we present some preliminary results that will be used in the AXIOM project to select proper implementation options for OmpSs to use the Zynq FPGA. Figure 4 presents the basic diagram of the Zynq chip with the main blocks, and the way they are interconnected. The central part of the figure is the Zynq chip. It is split in two main parts: at the top part, the Processing System (PS), contains the hardwired Cortex A9 processors and a memory controller. At the bottom, the Programmable Logic (PL) contains the FPGA

Memories represented at the sides of Figure 4 are the main host memory, or PS RAM (Host), on the left side, which is directly accessible from the A9 cores, and it is the one that Linux manages, with a capacity of 1GByte; and the FPGA memory, or PL RAM (FPGA), on the right side, which is directly accessible from the FPGA, and its capacity is also of 1GByte. Additionally, the PL-BRAM memory (shown inside the AXIOM accelerator IP) is the typical block RAM memory available inside FPGA chips. It is smaller in size, ranging from 3.3MBytes (on the 7015 chip) to 19.3MBytes (on the 7045 chip).

In this environment, programmers locate their application data in the PS RAM (Host). From there data can be transferred to the PL-RAM (FPGA) when the amount of data is significant, and also when it is known to be reused often. And for smaller chunks of data (up to a few Mbytes), it is enough to move it directly to the PL-BRAM on the specific FPGA device (AXIOM accelerator IP in the figure).

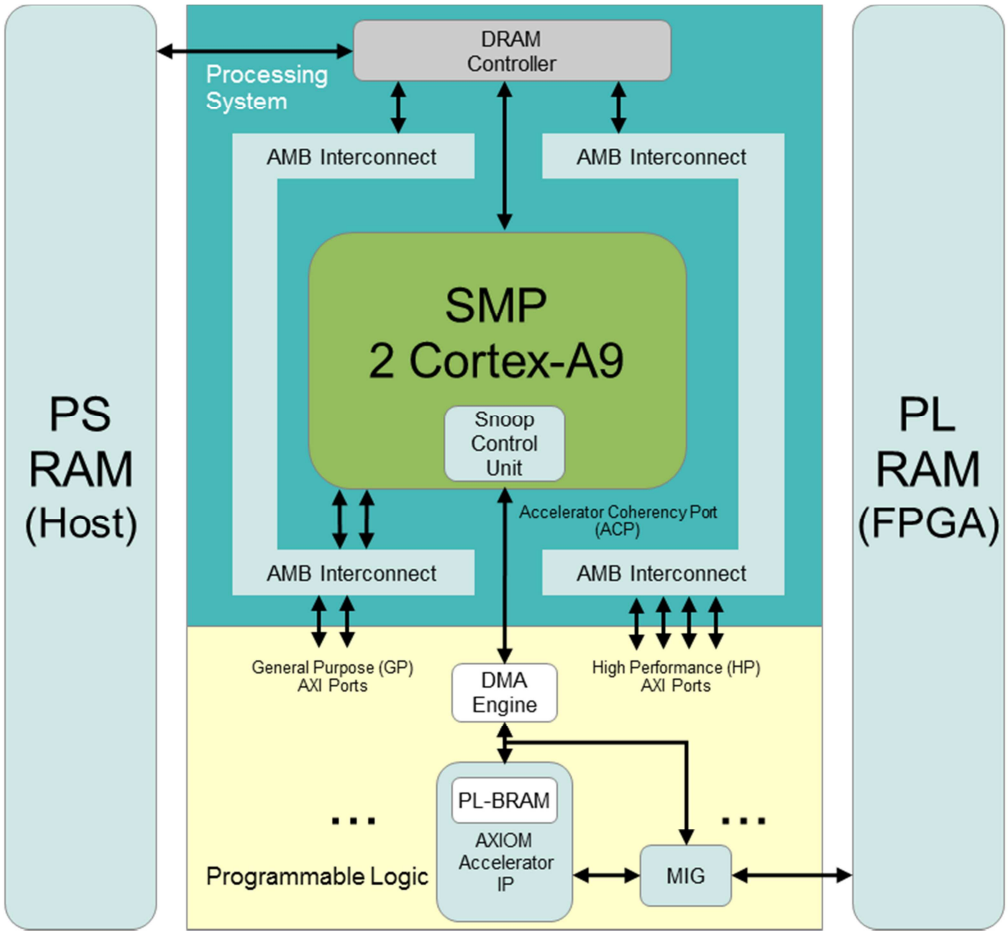

Figure 4 – Basic structure of the Xilinx Zynq chip and the connectivity of the ARM cores (PS) with the FPGA logic (PL)

In particular, we show measurements of the following data access methods:

- The data transfer communication connection between the Programmable Logic and the Host CPU in the Zynq FPGAs (path from the PS RAM of the host, to the PL BRAM on an accelerator in Figure 4)

- The data transfer type: synchronous or asynchronous at the Nanos++ FPGA device dependent layer
- Performance estimation and evaluation of hardware/software decisions based on the number of accelerators and size, and the heterogeneous execution
- Evaluation of the performance of the data transfers from the PS RAM of the host to the PL RAM of the FPGA, and then to the PL BRAM on the accelerator (see Figure 4).

5.1 PS RAM to PL BRAM analysis

In the Zynq boards, the type of memory transfer to implement is important, as it will influence the design and generation of the hardware accelerators to be run in the FPGA, and to create the device tree of the Linux operating system. This is necessary because the hardware accelerator implementations have to include on the FPGA the IPs needed to do the data transfers from the RAM of the Host to the BRAMs of the FPGA, and this IP of the DMA engine should be recognized by the Linux operating system at boot time.

This evaluation will help decide the best FPGA device to perform memory transfers that can be developed as a part of the FPGA device support in the Nanos++ runtime.

Figure 5 shows the accumulated memory bandwidth achieved when transferring a certain amount of 32-bit elements using the seven possible ports between the PL and the PS systems: 1 Accelerator Coherence Port (ACP), 4 High Performance (HP) ports, and 2 General Purpose (GP) Ports (see Figure 4). The experiment is done in standalone mode, and with a single ARM core driving the transfers. The accumulated memory bandwidth is shown for input (PS RAM to PL BRAM) and output (PL BRAM to PS RAM) memory transfers. The total accumulated memory bandwidth is of about 1.5GB/s.

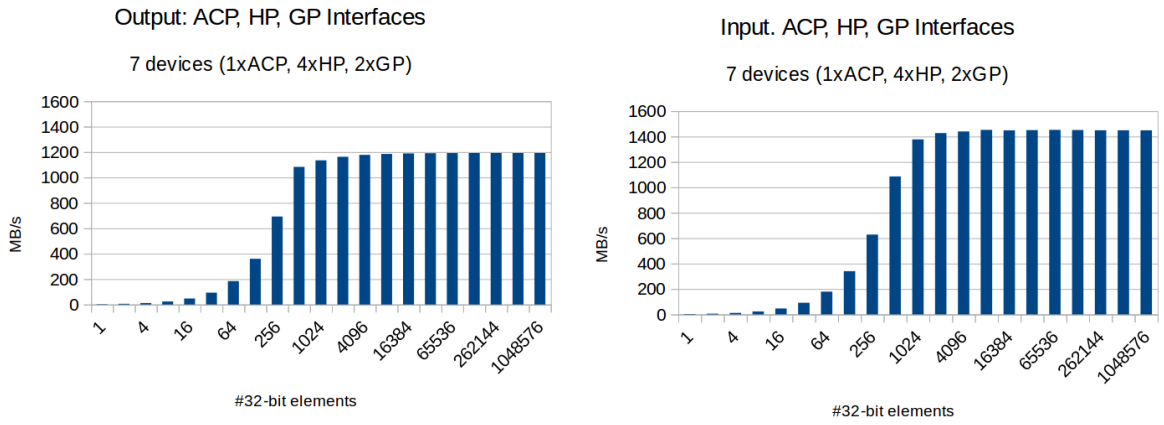

Figure 5 - Accumulated memory bandwidth (in Mbytes/s) using up to seven PS to PL connection ports (standalone execution, 1 ARM core).

Although the data rate reached in these experiments should be enough to fully support communications of 3-6 Gbit/s on the AXIOM-link connection, the specific IP cores used to implement the NI of the communication layer may limit the final communication rate achieved.

Figure 6 and Figure 7 show the separated memory bandwidth for each of the connection ports. The only ACP connection is shown, and in the case of the HP and GP interfaces, with up to 4 and 2 connections respectively. Table 1 shows the unitary and aggregated bandwidth for each of the port types, when working on a single direction. The peak memory bandwidth is not achieved for any connection, as shown in the figures.

The input memory bandwidth (Figure 6) for the ACP connection is worse than that achieved by the other connections. In this case, ACP maintains the coherence with the SMP. The output memory bandwidth (Figure 7) on the ACP connection is similar to 2 GP connections running in parallel and slightly better than two non-coherent HP connections.

Table 1 – Peak, single direction, bandwidth achieved for each type of communication port in the Zynq FPGA running at 100 Mhz. (in MBytes/s)

Port	**ACP**	**HP**	**GP**
Peak bandwidth (unitary)	381 MB/s	381 MB/s	381 MB/s
Peak bandwidth (aggregated)	381 MB/s (1 port)	1525 MB/s (4 ports)	762 MB/s (2 ports)

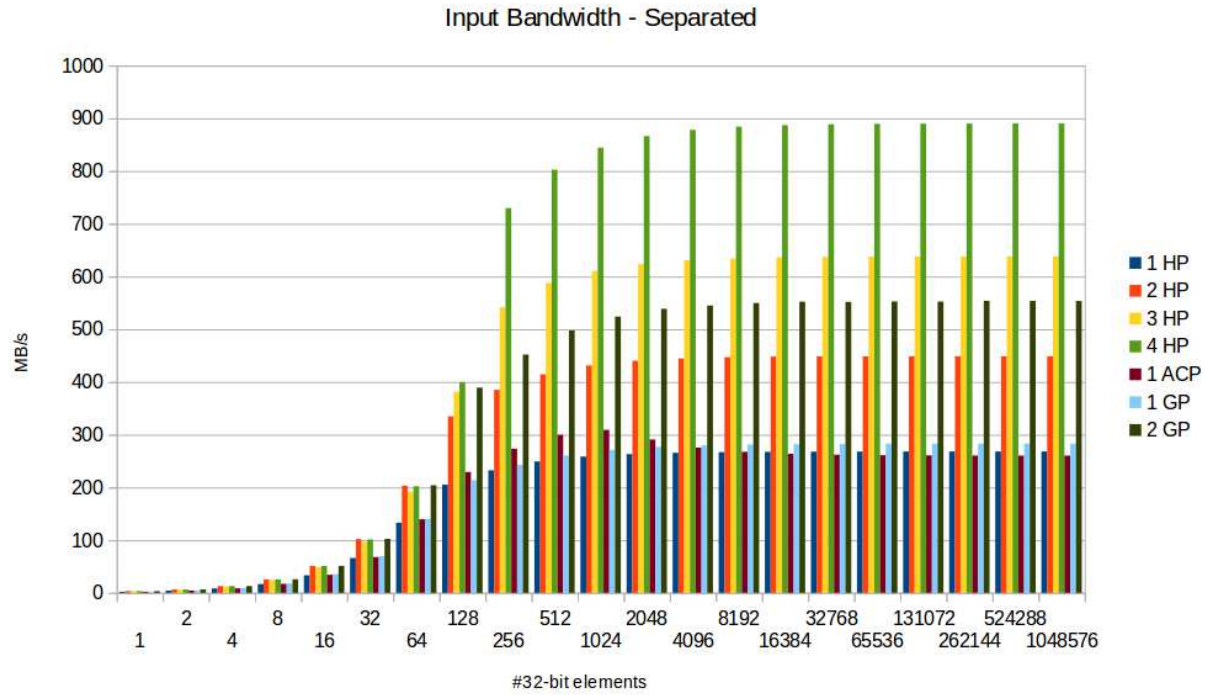

Figure 6 – Input (PS to PL) memory bandwidth for each of the PL to PS connections.

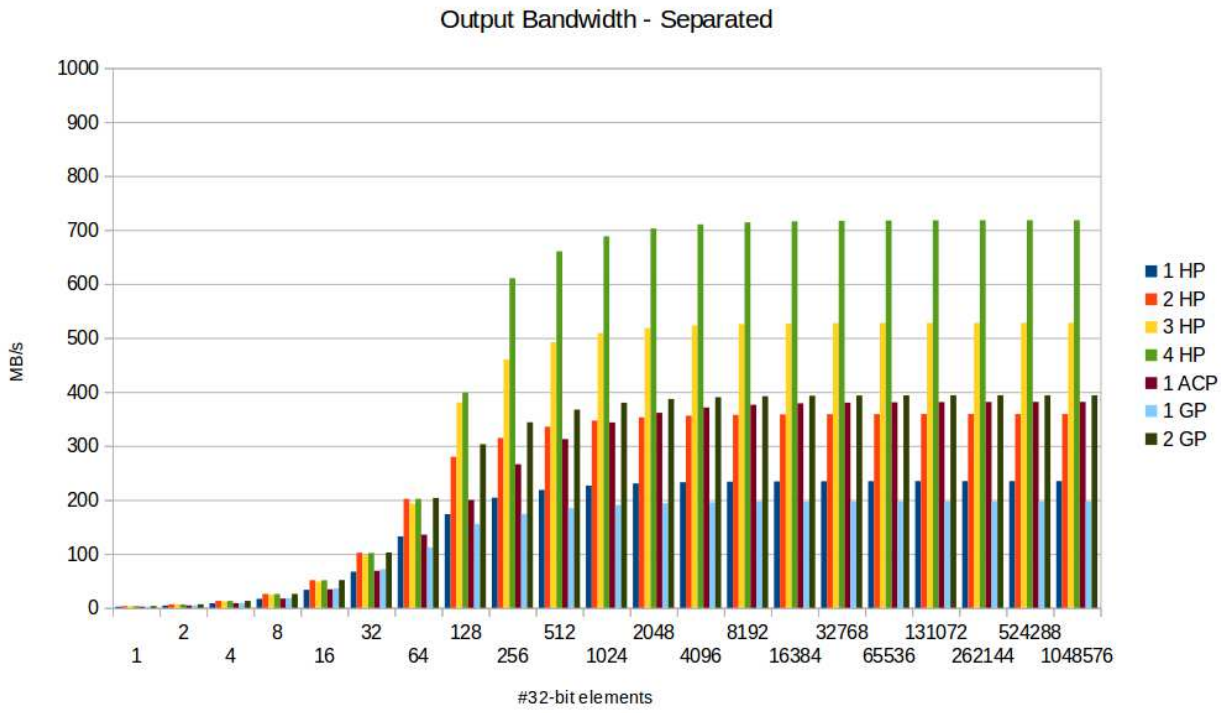

Figure 7 – Output (PL to PS) memory bandwidth for each of the PL to PS connections.

Therefore, the ACP does not seem to be the best choice based on the memory transfer bandwidth, but it keeps the coherence of the memory cache with the Host cores of the Zynq system, which is really important for heterogeneous parallel executions. We decided to evaluate if having more than one device connected to the ACP port may help getting better performance. Figure 8 (input) and Figure 9 (output) show the input and output memory bandwidth achieved using the ACP connection with:

- 1 FPGA device (i.e., one AXIOM accelerator on the FPGA), and 1 thread generating data (blue bar)
- 2 FPGA devices, and 1 thread generating data (red bar)
- 2 FPGA devices, and 2 threads generating data (yellow bar)

Figures also show the peak performance that can be achieved with 1 device (green curve) and 2 devices (brown curve). We have done the experiment varying the accelerator frequency since this also influences the memory bandwidth performance.

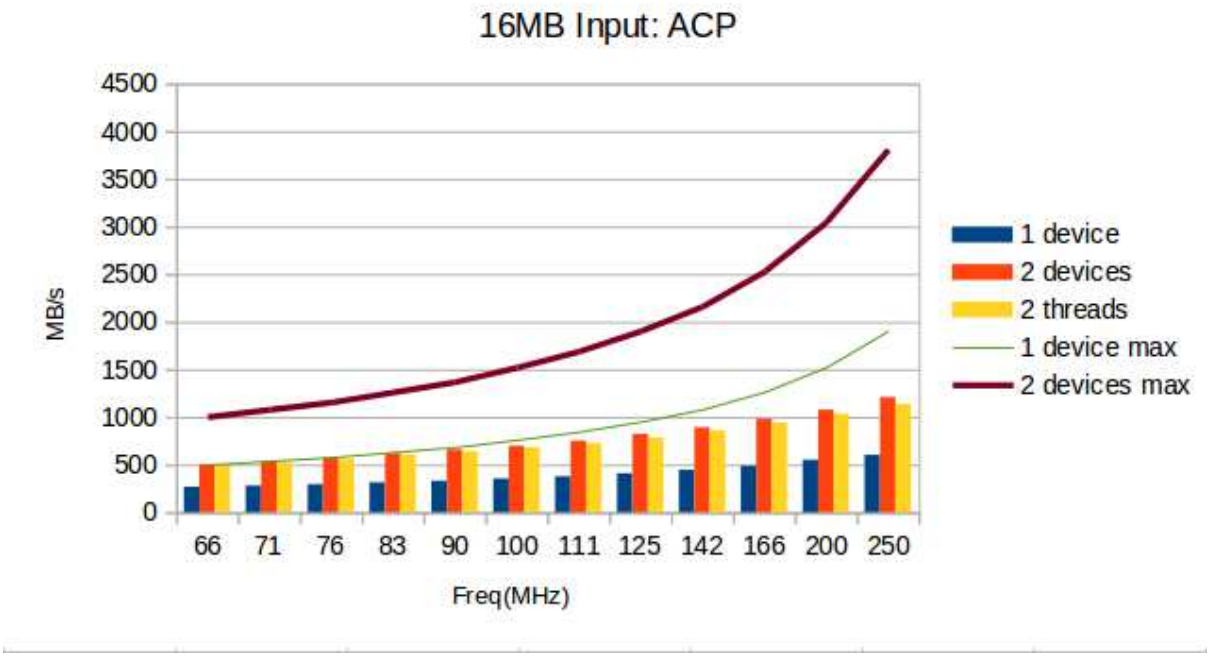

Figure 8 – Input (PS to PL) memory bandwidth using 1 or 2 ACP-connected devices, and 1 or 2 threads.

On one hand, results are really promising since we can double the DMA memory transfer bandwidth using 2 devices (with 1 or 2 threads) in the case of input memory transfers. This is a significant improvement if we look at the 100MHz frequency, but it is even more significant the bandwidth achieved when increasing the frequency. On the other hand, the output memory bandwidth achieved is doubled also when using 2 threads, but not for 1 thread. These results show that the ACP connection can achieve good memory bandwidth, keeping the coherence of the memories, if we use more than one device and more than one thread to use the accelerators connected to the ACP.

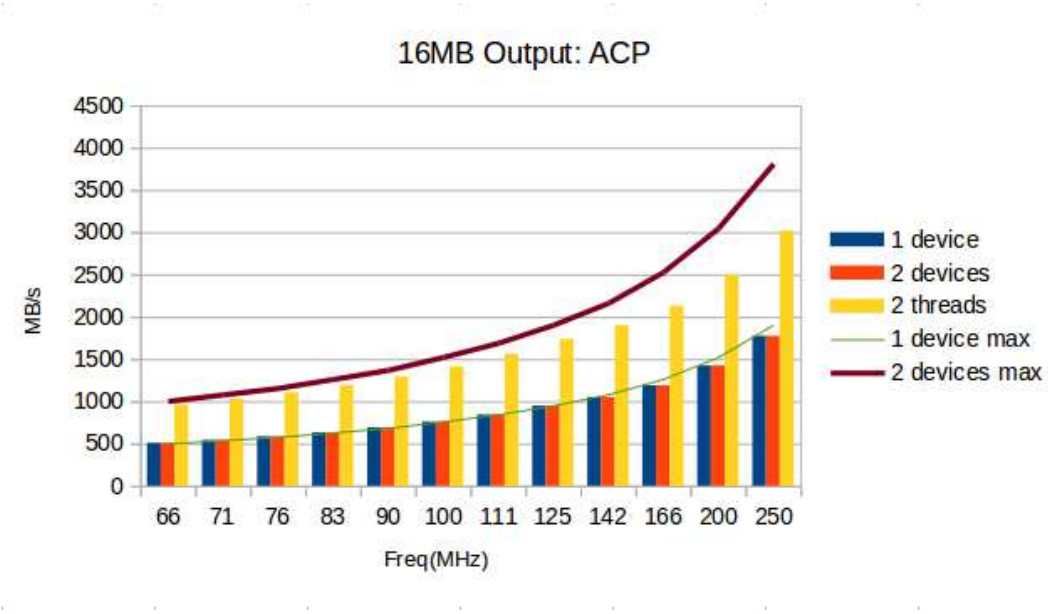

Figure 9– Output (PL to PS) memory bandwidth using 1 or 2 ACP-connected devices, and 1 or 2 threads.

5.2 Asynchronous vs Synchronous memory transfers

Another aspect that has been evaluated is the usage of synchronous or asynchronous memory transfers to improve the application performance. Figure 10 shows the performance comparison in GFLOPS for a Matrix Multiply (1024x1024) using a 128x128 MxM accelerator, at different frequencies, using synchronous and asynchronous memory transfers. The GFLOPS are calculated by dividing the amount of floating point operations (1024^3) by the time taken by the execution of the matrix multiplication. In the experiment with asynchronous transfers, the benchmark computes on a block, while there are transfers for other blocks in progress, thus overlapping communication of matrix blocks with computation on other blocks already transfered. In the synchronous version all data transfers have a DMA wait operation before proceeding.

Results show that (1) frequency helps to improve the MxM application performance, and (2) asynchronous memory transfers are worthy to be implemented inside the Nanos++ runtime.

5.3 Master/worker thread design in the Nanos++ runtime

Currently, OmpSs uses a schema with one master thread and one helper thread per device. However, in a Zynq system, with only 2 cores in the SMP, this can mean a significant impact due to context switches (three threads fighting to obtain 1 of the two SMP cores). We have evaluated the current behavior of this case, and also two more: (1) one helper thread is in charge of more than one FPGA accelerator, and (2) one SMP thread (worker or master) deals with SMP and accelerator executions.

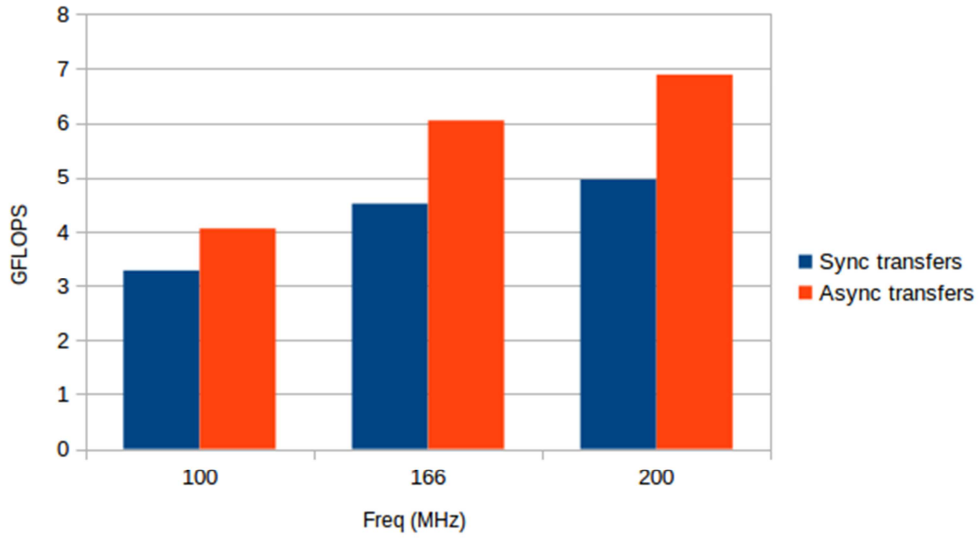

Figure 10– Performance obtained from a Tiled Matrix Multiplication benchmark (matrix size 1024x1024, tile size 128x128) in GFLOPS.

Figure 11 shows the performance results of those three situations using a prototype of the Nanos++ runtime. The evaluation is done for a MxM of 1024x1024, using 128x128 blocks. We also show performance results for the following cases:

- 1 acc: using just one accelerator
- 2 acc (1 helper thread): 2 accelerators are used by the same and unique helper thread.

- 2 acc (2 helper threads): 1 helper thread per accelerator.
- 2 acc (1 helper + hyb. Master): master thread can create tasks, execute them in the SMP and also execute them in the accelerators.
- 2 acc (1 helper) + smp: SMP is used to execute 128x128 MxM blocks, and 1 helper thread to run in two hardware accelerators.

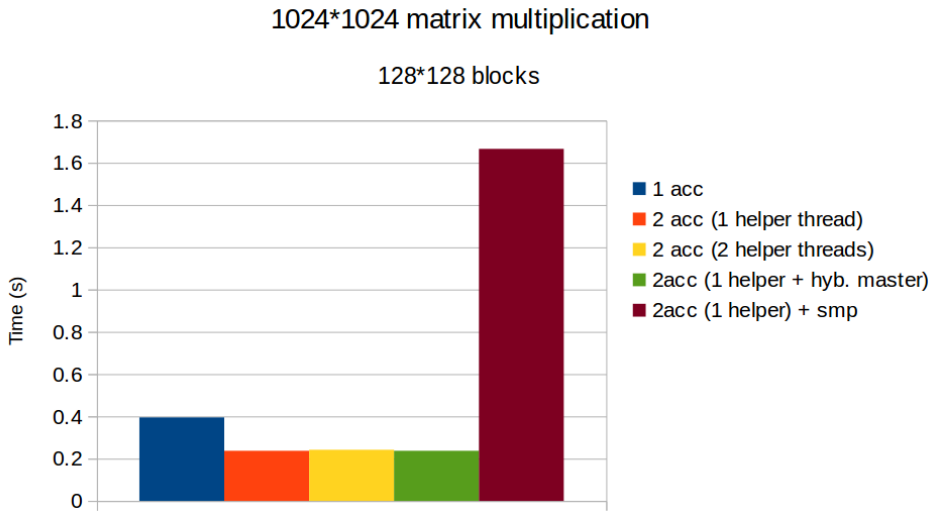

Figure 11- Execution time results for the Matrix Multiplication (matrix size 1024x1024, tile size 128x128).

The results do not show a significant impact on the performance having 2 helper threads and 1 master thread running in the Host cores of the Zynq chip. However, we can observe that the context switches on the system increases for our application, although it seems that it was not a significant problem for this case. On the other hand, the heterogeneous execution of 128x128 MxM blocks, with the current scheduling policy does not help at all (2 acc 1 helper + smp) to achieve performance. The MxM execution on the SMP is too slow compared to the hardware accelerators and any execution on it provokes a big load unbalance.

Related to this behavior, we started to evaluate the possibility of estimating the overall heterogeneous performance, so that we can decide which is the best hardware/software co-design decision to annotate the applications in a proper way.

Figure 12 shows the heterogeneous performance estimation for different hardware/software co-design alternatives to accelerate the MxM application. In this estimation we consider a tiled matrix multiply of 256x256, and blocks of 64x64 or 128x128 size. The evaluation estimates which is the best decision in a heterogeneous platform regarding to: block size, number of accelerators, and possible hybrid execution in a SMP host. The result shows the same trend in both estimation and real execution, and this is that SMP is not helping to improve the application performance, and the 128x128, 1 accelerator, no heterogeneous execution is the best choice. In this particular experiment, this happens because the execution of an SMP task takes much longer than the execution on the FPGA accelerator, and as a consequence, when the accelerator has finished all assigned tasks, there is still pending work to finish on the SMP side.

Therefore, this is a good starting point to analyze different scheduling policies that can help to avoid situations like those in Figure 11, where the heterogeneous execution (i.e., acc + smp) was not a good choice.

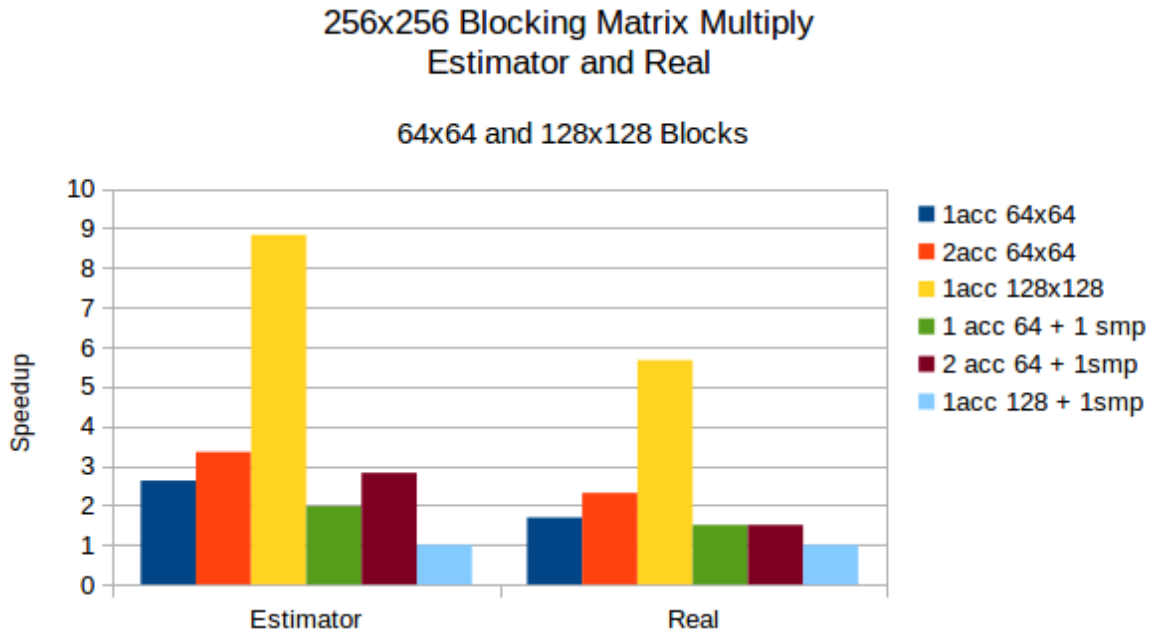

Figure 12– Heterogeneous performance estimation vs. real execution.

5.4 PS RAM (host) to PL RAM (FPGA) evaluation

Finally, we have also evaluated the possibility of accessing the external PL RAM (1GB of memory). Using the external memory of the PL can help to re-use the data among accelerators without having to access the SMP (PS) RAM every time (see Figure 4). Therefore, Nanos++ will initialize a DMA from SMP RAM to FPGA RAM, and then, the accelerator will be able to access and read the FPGA RAM by itself, and will allow the SMP to do other processing.

Figure 13 shows the performance results for a MxM multiplication of 2048x2048 single precission floating poing using 128x128 or 256x256 blocks in a 706 board with a PL at 200MHz , and a PS at 800MHz. In particular, we show in the first bar the execution time of a MxM of this size, block 128x128, using PS RAM to PL BRAM. Second and Third bars show the performance results for 128x128 and 256x256 blocking MxM. In both cases we distinguish from top to bottom: computation part, FPGA RAM to BRAM communication, and then PS RAM to PL RAM communication. Those preliminary results show that the RAM to RAM communication has a very promising small cost. Also, there is a trade off between the overall cost on communication and computation, as it can be seen in the figure. The 128x128 double the communication time of the 256x256, but it is compensated by the smaller computation cost of the 128x128.

In any case, this very small memory transfer cost between RAMs can be taken into account to use the external FPGA memory RAM as a global shared memory is used in the GPUs in OmpSs.

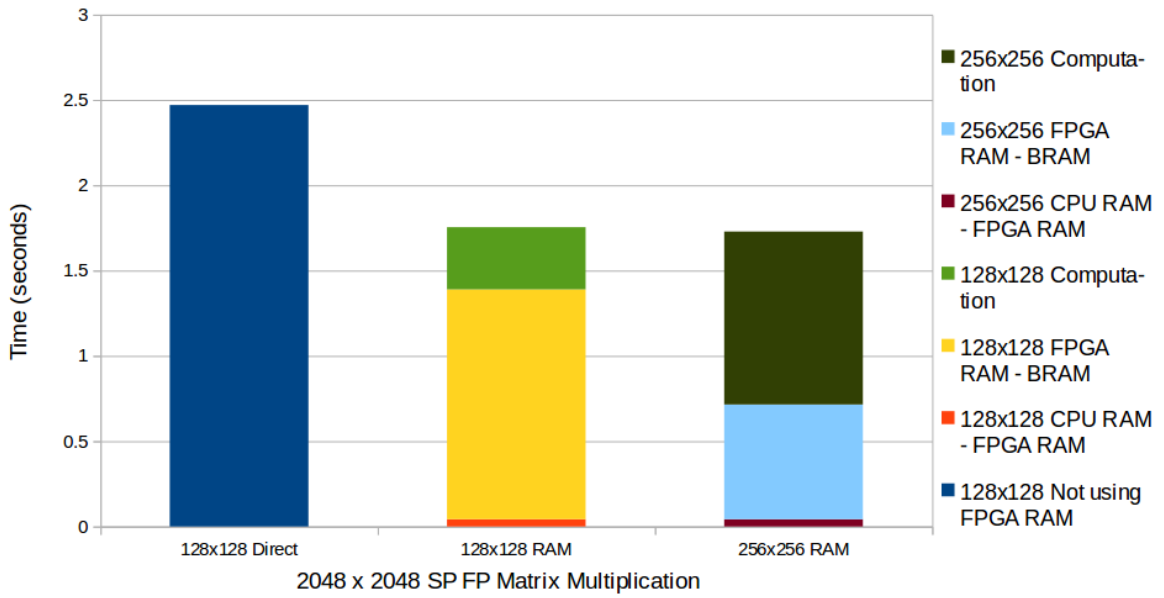

Figure 13– Memory transfer comparison PS RAM to PL RAM vs. PS RAM to PL BRAM.

6 Confirmation of DoA objectives

Describe how the deliverables conform to the DoA stated objectives, using the sample table if appropriate.

PLANNED	DELIVERED
DELIVERABLE:	
• Specification of the OmpSs extensions, cluster support, communication layer and evaluation of the low-level mechanism of the FPGA in the Zynq board.	Report

7 Conclusion

We have presented the programming model extensions proposed for programming the AXIOM boards with OmpSs, the design of the support for the FPGA devices, the distributed cluster environment, and the communication layer.

Programming the AXIOM environment will be based on the OmpSs programming model with cluster support, and extended to support target devices in the board FPGA. The Nanos++ runtime will use the DMA library design presented in this deliverable to take care of data transfers between the host memory and the FPGA devices.

Supporting distributed environments will be based on the communication layer (AXIOM-link) provided by FORTH to exchange data between AXIOM boards. In the project we are considering the use

of common tools, like MPI or GASNet, and also implementing our specific approach based on the XSMLL infrastructure.

An initial evaluation of the low level communication mechanisms to transfer data to and from the FPGA devices is shown. These results will be used during the design of the support for the OmpSs extensions for FPGAs to decide the specific mechanism to use in the implementation.

The next steps that will be taken in the project will be to work and provide the implementation for the different components that we have now designed:

- The final prototype targeting the FPGA devices, based on the DMA library to transfer data to and from the FPGA.
- The support for distributed systems based on common tools and the XSMLL infrastructure.
- The communication layer implemented in the FPGA.

We will also evaluate the possibility to reuse some of the components implementing the communication layer, specially the DMA devices, to be used to transfer data to the accelerators in the FPGA. This way, we can save some of the FPGA resources and fit larger accelerators on it.

References

1. Dan Bonachea; GASNet Specification, v1.1. Report No. USB/CSD-02-1207. CS Division, EECS Department, University of California, Berkeley; October 2002; http://gasnet.lbl.gov/CSD-02-1207.pdf
2. Javier Bueno, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta; Implementing OmpSs support for regions of data in architectures with multiple address spaces. ICS 2013: 359-368 (2013).
3. Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, Judit Planas; OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures. Parallel Processing Letters 21(2): 173-193 (2011).
4. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 3.0; September 2012; http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf
5. 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.
6. DMA driver for AXIOM: https://git.axiom-project.eu/?p=axiom-dma
7. XSMLL API for AXIOM: https://git.axiom-project.eu/?p=XSMLL
8. OmpSs website: http://pm.bsc.es/ompss
9. COTSon website: http://cotson.sourceforge.net/
10. 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
11. Xilinx, Inc. Vivado Design Suite – HLx Edition, http://www.xilinx.com/products/design-tools/vivado.html