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

PROJECT NUMBER: 645496

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

D4.3 – Evaluation of the compiler and tools infrastructure

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

Start date of the project: February 1st, 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	05/02/2018	Xavier Martorell	BSC/UPC	Evaluation of Om-pSs@FPGA
0.2	07/02/2018	Daniel Jiménez-González	BSC/UPC	Evaluation of Om-pSs@Cluster
0.3	07/02/2018	Paolo Gai	EVI	Evaluation of Om-pSs@Cluster
0.8	08/02/2018	Carlos Álvarez	BSC/UPC	1st internal review
0.9	09/02/2018	David Oro	HERTA	2nd internal review

Release Approval

Name	**Role**	**Date**
Xavier Martorell	WP-leader	12.02.2018
Roberto Giorgi	Coordinator	14.02.2018

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

Xavier Martorell (UPC/BSC)
Daniel Jiménez-González (UPC/BSC)
Paolo Gai (EVI)
Carlos Álvarez (UPC/BSC)

TABLE OF CONTENTS

TABLE OF FIGURES

GLOSSARY

ARM – Instruction set architecture developed by ARM Holdings Ltd.

AXI – Advanced Extensible Interface

BRAM – Block RAM
DMA – Direct Memory Access
DoA - Description of Action (acronym set by the European Commission)

DSP – Digital Signal Processor
ELF – Executable and linkable Format

FF – Flip-Flop
FPGA – Field Programmable Gate Array
HLS – High Level Synthesis Language

IoT – Internet of Things
IP – Intellectual property

LUT – Look-Up Table
MHz – Mega Hertz

MPSoC – Multiprocessor System-on-Chip

MxM – Matrix Multiply
NIC – Network interface

OmpSs – OpenMP Superscalar
OpenBLAS – Basic Linear Algebra Subprograms optimized library

OpenMP – Open Multiprocessing standard
RAM – Random Access Memory
RDMA – Remote Direct Memory Access
SGEMM – Single-precision floating-point general matrix multiply
SMP – Symmetric multiprocessing

TCL – Tool Command Language

Executive summary

This Deliverable reports the main features and capabilities of the compiler and tools infrastructure developed within the context of the AXIOM project (including autoVivado and the runtime library). It also analyzes the results obtained during the evaluation of the developed tools and finally discusses the optimizations implemented for successfully targeting the AXIOM board.

1 Introduction

1.1 Document structure

This section enumerates the tasks involved in this Deliverable, as well as other related deliverables. An outline of the topics covered during next sections are included below as a reference:

- Section 2 explains the modifications carried out to the compiler (Mercurium [9]), and the toolset (i.e. Nanos++ and autoVivado) developed during the AXIOM project.
- Section 3 discusses the evaluation and optimization of such tools to target the AXIOM board.
- Section 4 evaluates the developed toolset on a cluster of AXIOM boards.
- Section 5 includes an explanation describing how the DoA objectives were achieved.
- Section 6 presents the final conclusions.

1.2 Relation to other deliverables

This deliverable is mainly related to D4.2 (AXIOM code generation and instrumentation), D5.4 (Final operating system and documentation), and D3.3 (Software and report on application porting). Mercurium FPGA annotations used in the source code were already described in Deliverable D4.2. The runtime and OS development were presented in D5.4. Finally, performance results of the use-cases applications using the tools described in this deliverable are reported in Deliverable D3.3.

1.3 Tasks involved in this deliverable

This deliverable involves Task 4.5: Evaluation and tuning of the code generation.

2 Development of the compiler and tools

During the third year of the AXIOM project, the development of the OmpSs [[4], [2]] compiler and related tools have been successfully completed. Figure 1 represents the support that we have implemented within the context of this project. As such, OmpSs applications are now able to run on several of the Zynq 7000-based boards (including 706, 702, and Zedboard), and on boards based on the Zynq Ultrascale+ MPSoC (including Trenz Electronics TE0808, and the AXIOM board). The next subsections present the final version of the compiler toolchain at its current development state.

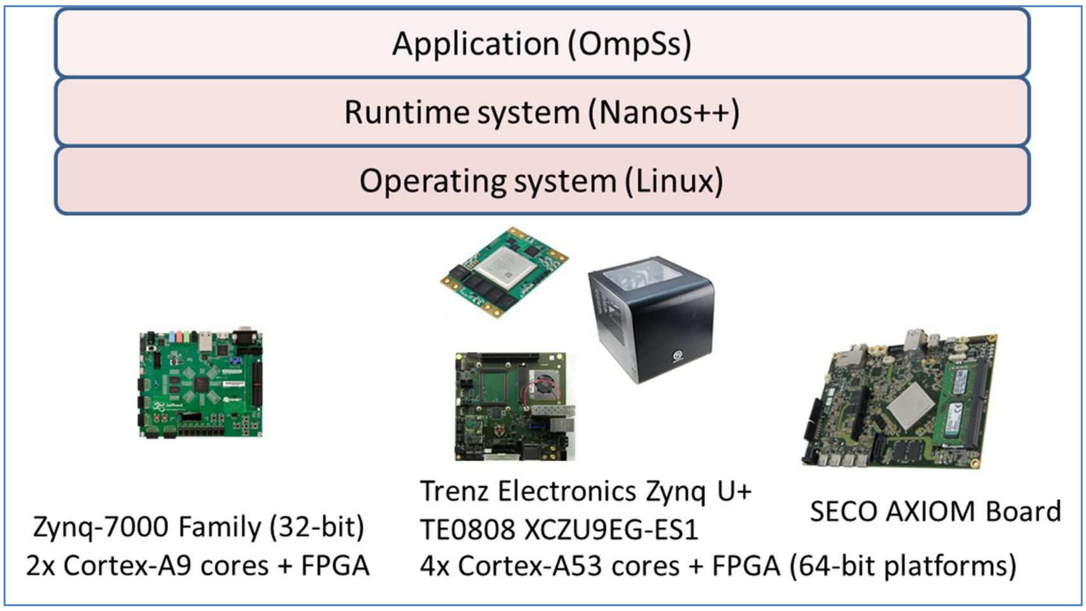

Application (OmpSs)
Runtime system (Nanos++)
Operating system (Linux)
Zynq-7000 Family (32-bit)
2x Cortex-A9 cores + FPGA
Trenz Electronics Zynq U+
TE0808 XCZU9EG-ES1
SECO AXIOM Board
4x Cortex-A53 cores + FPGA (64-bit platforms)

Figure 1. Boards supported by the AXIOM project

2.1 Compiler toolchain

The compiler toolchain consists of the **Mercurium compiler** and the **autoVivado plugin**. This toolset has the goal to parse a given application source code, annotated with the OmpSs directives, and later perform transformations, in order to get a binary program for the target system. In the case of the AXIOM board, the binary program consists of the ELF executable running on Linux, and the bitstream used to configure the FPGA fabric. The structure of the compilation toolchain is shown in Figure 2.

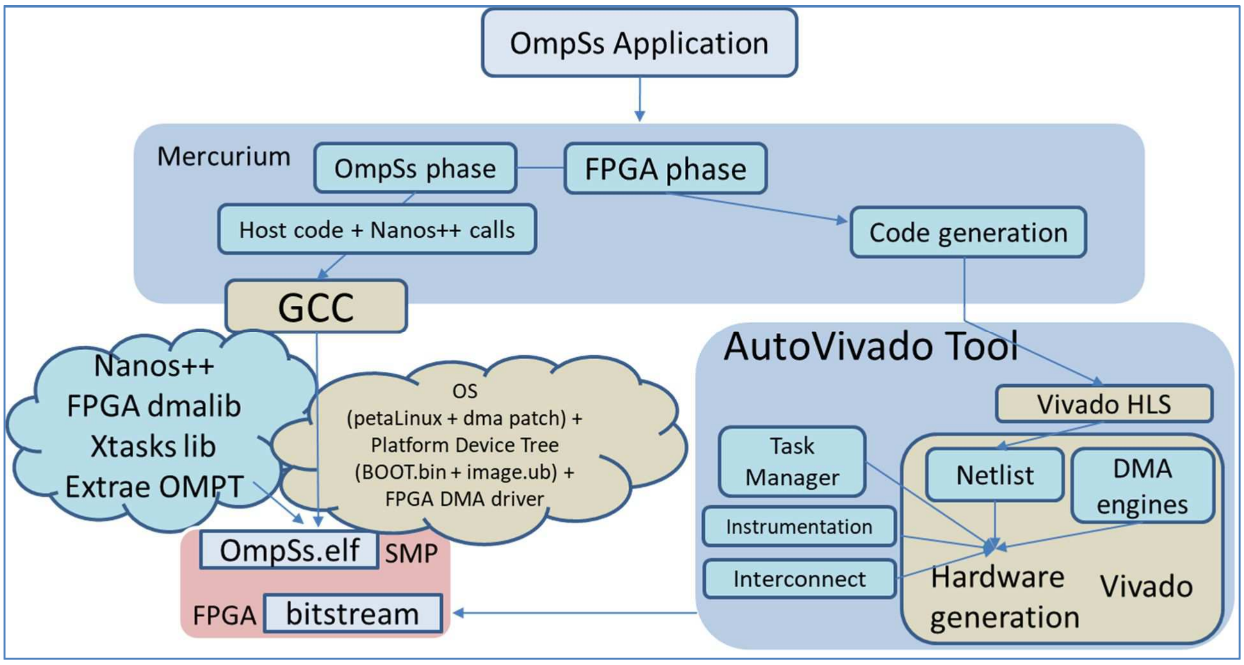


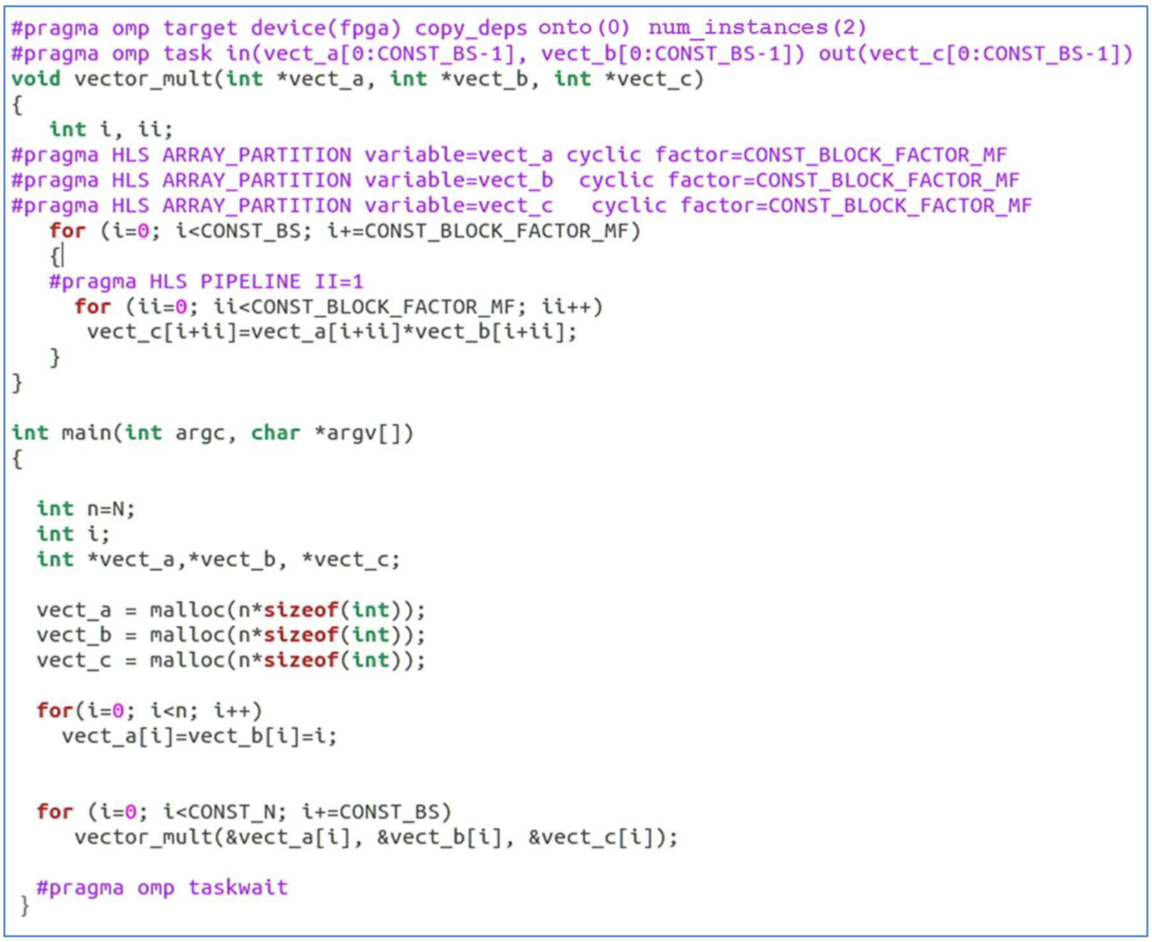
Figure 2. Structure of the compilation toolchain

The Mercurium compiler is used to understand the annotations provided by developers, and outline the code targeting the FPGA device, from the host code running on the SMP ARM cores. The Mercurium FPGA annotations have already been described in the AXIOM Deliverable D4.2 [[1]].

The generation of the bitstream to configure the FPGA is done automatically by taking the code outlined through a set of transformations using the Xilinx Vivado HLS toolchain [[5]] (see right-hand side of Figure 2). The integration of Mercurium and Vivado HLS is based on **autoVivado**, a Mercurium plugin consisting of a set of Tcl scripts [[7]]. They perform the Vivado HLS compilation of the code, linking them with a set of predefined IP cores for the target board. These predefined IP cores contain the infra-structure needed for task management, communication and instrumentation. They need to be provided specifically for each board. Currently our system supports various Xilinx Zynq-7000 (Zedboard, ZC702, and ZC706), the AXIOM Board, and Trenz Electronics TE0808 boards.

2.1.1 Code transformations

Figure 3 shows an example of code implementing a vector by vector multiplication, which will be used to explain the mechanism used for the transformations done for the FPGA. This code comes with the OmpSs annotations `omp target` and `omp task`. They indicate the data used by the function `vec-tor_mult`, and their directionality (`in`, `out`, `inout`). They also indicate that the programmer wants to generate two instances (`num_instances(2)`) of the IP onto the FPGA, and that the functionality implemented by `vector_mult` will be identified with the Id 0, indicated in the `onto` clause.



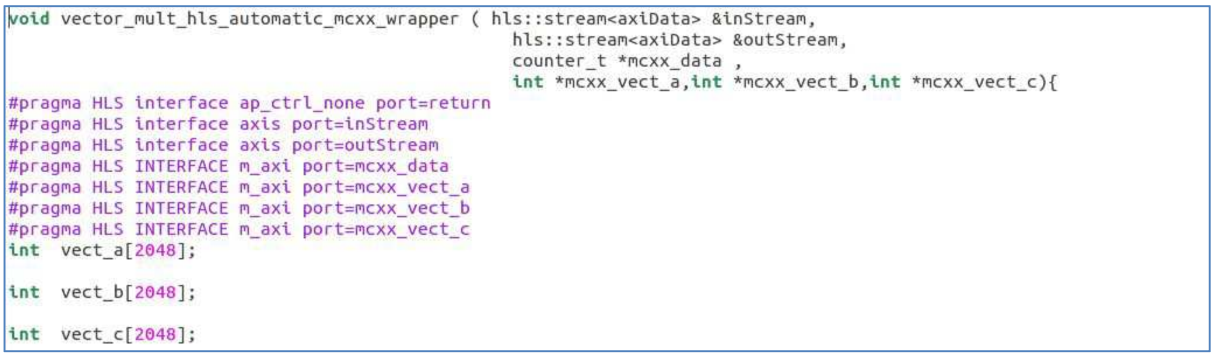
```
#pragma omp target device(fpga) copy_deps onto(0) num_instances(2)
#pragma omp task in(vect_a[0:CONST_BS-1], vect_b[0:CONST_BS-1]) out(vect_c[0:CONST_BS-1])
void vector_mult(int *vect_a, int *vect_b, int *vect_c)
{
   int i, ii;
#pragma HLS ARRAY_PARTITION variable=vect_a cyclic factor=CONST_BLOCK_FACTOR_MF
#pragma HLS ARRAY_PARTITION variable=vect_b  cyclic factor=CONST_BLOCK_FACTOR_MF
#pragma HLS ARRAY_PARTITION variable=vect_c   cyclic factor=CONST_BLOCK_FACTOR_MF
   for (i=0; i<CONST_BS; i+=CONST_BLOCK_FACTOR_MF)
   {
   #pragma HLS PIPELINE II=1
     for (ii=0; ii<CONST_BLOCK_FACTOR_MF; ii++)
      vect_c[i+ii]=vect_a[i+ii]*vect_b[i+ii];
   }
}

int main(int argc, char *argv[])
{

  int n=N;
  int i;
  int *vect_a,*vect_b, *vect_c;

  vect_a = malloc(n*sizeof(int));
  vect_b = malloc(n*sizeof(int));
  vect_c = malloc(n*sizeof(int));

  for(i=0; i<n; i++)
    vect_a[i]=vect_b[i]=i;


  for (i=0; i<CONST_N; i+=CONST_BS)
     vector_mult(&vect_a[i], &vect_b[i], &vect_c[i]);

  #pragma omp taskwait
 }
```

Figure 3. Original code annotated with the OmpSs directives

The code to be accelerated is also annotated with Xilinx HLS directives, this time targeting the Vivado HLS compiler. They indicate that the data should be spread across the available BlockRAM in the target device, and that the code should be pipelined.

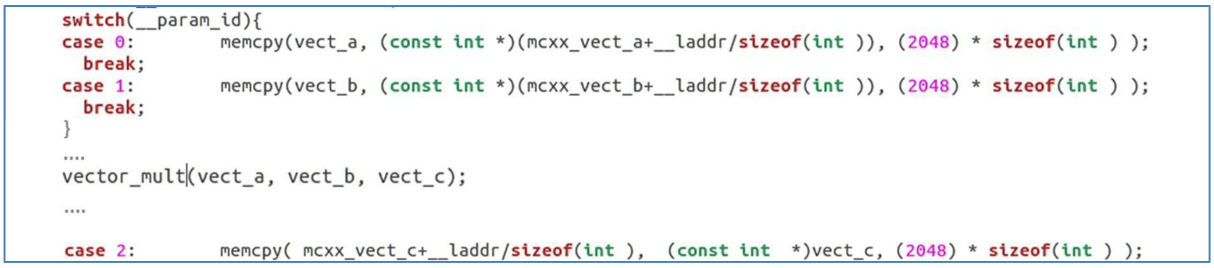
The Mercurium compiler outlines this code onto a separate file, and includes the wrapper function shown in Figure 4 (header). This wrapper function is in charge of getting the pointers to the data in the host memory, and performing the data transfers from and to the host memory (Figure 5 – function body). An IP Master AXI port is defined for each object in the host memory (`mcxx_vect_a`, `mcxx_vect_b`, and `mcxx_vect_c`). The pointers to the base addresses of the objects in host memory are passed directly using the function interface AXI Stream port (`inStream`). The memory addresses to the current block of data for each original function argument are obtained from the `inStream` descriptor, provided by our runtime system (`libxdma`) at the time of the kernel invocation. The HLS special function `memcpy` is used to transfer the data from the communication port described by adding the IP master AXI port pointer (bus) with the offset address of the block of data to compute (`mcxx_vect_a + __laddr / sizeof(datatype)`), onto the local BlockRAM (`vect_a`).

Afterwards, the kernel function is invoked (`vector_mult` in Figure 5). And finally, the output master AXI port (`mcxx_vect_c`) plus the host memory is used to transfer the data computed (`vect_c`), onto the destination block, by invoking the `memcpy` function again.



```
void vector_mult_hls_automatic_mcxx_wrapper ( hls::stream<axiData> &inStream,
                                               hls::stream<axiData> &outStream,
                                               counter_t *mcxx_data ,
                                               int *mcxx_vect_a,int *mcxx_vect_b,int *mcxx_vect_c){
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface axis port=inStream
#pragma HLS interface axis port=outStream
#pragma HLS INTERFACE m_axi port=mcxx_data
#pragma HLS INTERFACE m_axi port=mcxx_vect_a
#pragma HLS INTERFACE m_axi port=mcxx_vect_b
#pragma HLS INTERFACE m_axi port=mcxx_vect_c
int  vect_a[2048];

int  vect_b[2048];

int  vect_c[2048];
```

Figure 4. Header of the wrapper function generated by Mercurium



```
    switch(__param_id){
    case 0:         memcpy(vect_a, (const int *)(mcxx_vect_a+__laddr/sizeof(int )), (2048) * sizeof(int ) );
      break;
    case 1:         memcpy(vect_b, (const int *)(mcxx_vect_b+__laddr/sizeof(int )), (2048) * sizeof(int ) );
      break;
    }
    ....
    vector_mult(vect_a, vect_b, vect_c);
    ....

    case 2:         memcpy( mcxx_vect_c+__laddr/sizeof(int ),  (const int  *)vect_c, (2048) * sizeof(int ) );
```

Figure 5. Body of the wrapper function (`copy_in`**, execution of the IP,** `copy_out`**)**

This code is provided to Vivado HLS. This is done by creating a Vivado HLS project and incorporating on it the wrapper code, possibly other IP code files that the application may have targeting the FPGA, and several support files. Support files include the implementation of internal FPGA timing, and hardware instrumentation [[3]]. For each of the application IP kernels, and the support files, Vivado HLS synthesizes the hardware, and finally exports the generated IP to be included in the Vivado project. After the transformation, the tool summarizes in a report the utilization of the underlying hardware resources available in the target chip. Figure 6 shows a sample output of this report, obtained for the Zynq Ultrascale+ chip of the AXIOM board.

```
   FF:      309232 used  |  548160 available  -  56.41% utilization

  LUT:      126837 used  |  274080 available  -  46.27% utilization

 BRAM:        1045 used  |     1824 available  -  57.29% utilization

  DSP:        1230 used  |     2520 available  -  48,81% utilization

1 kernel(s) and 4 support IPs synthesized. 171s elapsed.
```

Figure 6. Resource occupancy report

With this information, 1) autoVivado can stop the compilation if any of the synthetized IPs cannot be mapped to the available hardware resources, and 2) developers can determine in advance if the IP fits together on the FPGA, thus deciding whether to continue or not with logic synthesis; or to later refine the IP design in order to better exploit the utilization of FPGA hardware resources. At this stage, the generated IP is ready to be added into the Vivado Project for conducting bitstream generation.

2.1.2 Hardware Generation

Starting from the IP design autogenerated by Vivado HLS [[5]], the remaining parts of the compilation process follow the Xilinx Vivado [[6]] standard bitstream generation flow. This compilation process has been coded as a Tcl script, which is then parsed by the Vivado compiler to follow the steps included below:

- Generation of the block design
- Hardware synthesis
- Hardware implementation
- Bitstream generation
- Device tree generation

Additionally, the generated hardware logic block design can be also later displayed using the standard Vivado Design Suite. As an example, Figure 7 shows two instances of the `vect_mult` IP, the hardware instrumentation module (right side), and AXI interconnects used for data transfers from/to the IP cores (left side).

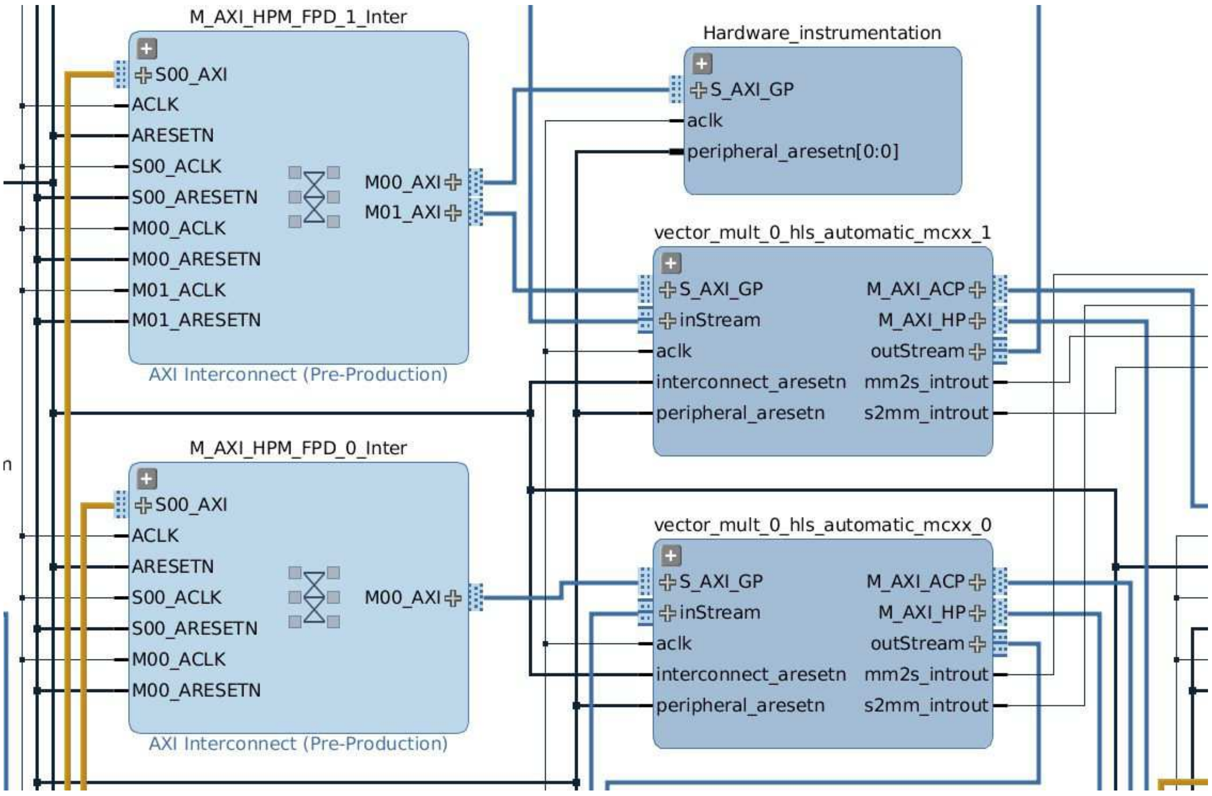


Figure 7. Instances of the `vector_mult` accelerator in the block design

2.1.3 autoVivado Performance Tuning

AutoVivado has been also improved during the third year of the AXIOM project. Now, it is capable of generating performance-oriented hardware designs. These new designs achieve a speedup of almost 2.5X when compared to previous autoVivado versions, which were oriented to optimize area instead. Figure 8 shows the speedup achieved in the matrix multiply application (MxM) for different matrix sizes when using only one 128x128 block accelerator to perform the tiled MxM.

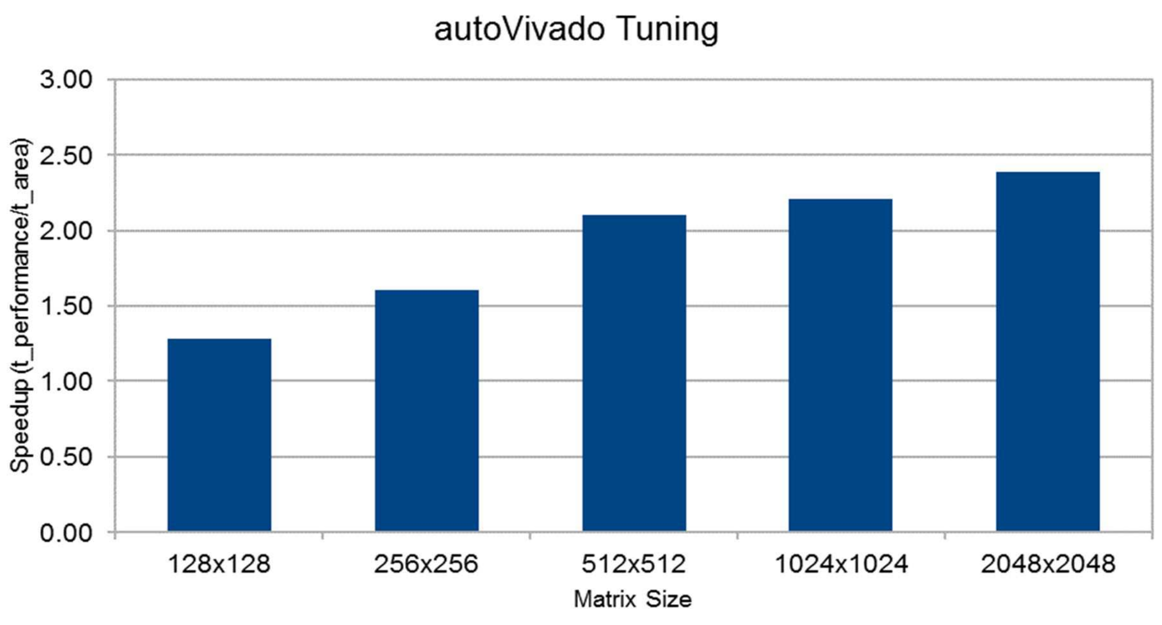


Figure 8. Speedup of the performance designs generated compared to the area hardware designs for different matrix sizes and one accelerator.

In any case, an area-optimized design may be interesting for some specific cases where we should fit several accelerators. For this reason, autoVivado allows to specify an option to set the target of the hardware design: performance or area.

2.1.4 Compilation and Compiler Options

When compiling the application, developers can enable or disable support for hardware instrumentation:

- Without instrumentation: `fpgacc --ompss -o program program.c`
- With instrumentation: `fpgacc --ompss --instrumentation -o program program.c`

On the other hand, in order to generate the required intermediate Vivado HLS, which includes the kernel wrapper, an additional Mercurium compiler option must be also set as it is described below:

- Generate bitstream: `--variable=bitstream_generation:ON`
- Do not generate bitstream: `--variable=bitstream_generation:OFF`

Bitstream generation is usually set in cross-compilation environment in an x86_64 server, and should be disabled when compiling on the boards.

At linking time, different options must be also specified in order to generate both the hardware design and its corresponding bitstream:

1. Target board
2. Clock frequency
3. Enable/disable support for hardware instrumentation
4. Optimize for area or performance
5. Vivado project name
6. Vivado project directory

The example included below demonstrates how to enable or set the previously described options:

- `--Wf,"--board=$(BOARD_NAME),--clock=200,\`
  `--hardware_instrumentation,--intercon_opt=performance"`
- `--Wf,"-v,--name=vivado_project_name,--dir=$(VIVADO_WORKSPACE)"`

Regarding the target platform, autoVivado supports the "`axiom`", "`trenz`" Zynq Ultrascale+ 64-bit boards, "`zynq706`", "`zynq702`", and "`zedboard`" 32-bit boards.

2.2 Runtime system

The Nanos++ runtime system was also updated to work with a newer version of the DMA library infrastructure, which was split into two different functionalities:

- *Xtasks*: This library provides support for accessing the FPGA device. Mainly, it is responsible for obtaining the list of IP accelerators synthesized on the FPGA, taking care of creating/destroying FPGA task descriptors, scheduling them to the hardware, and the synchronization needed to wait for their completion. Formerly, this functionality was embedded in the *libxdma* library.

- *Libxdma*: This library implements data transfer management from/to the FPGA device.

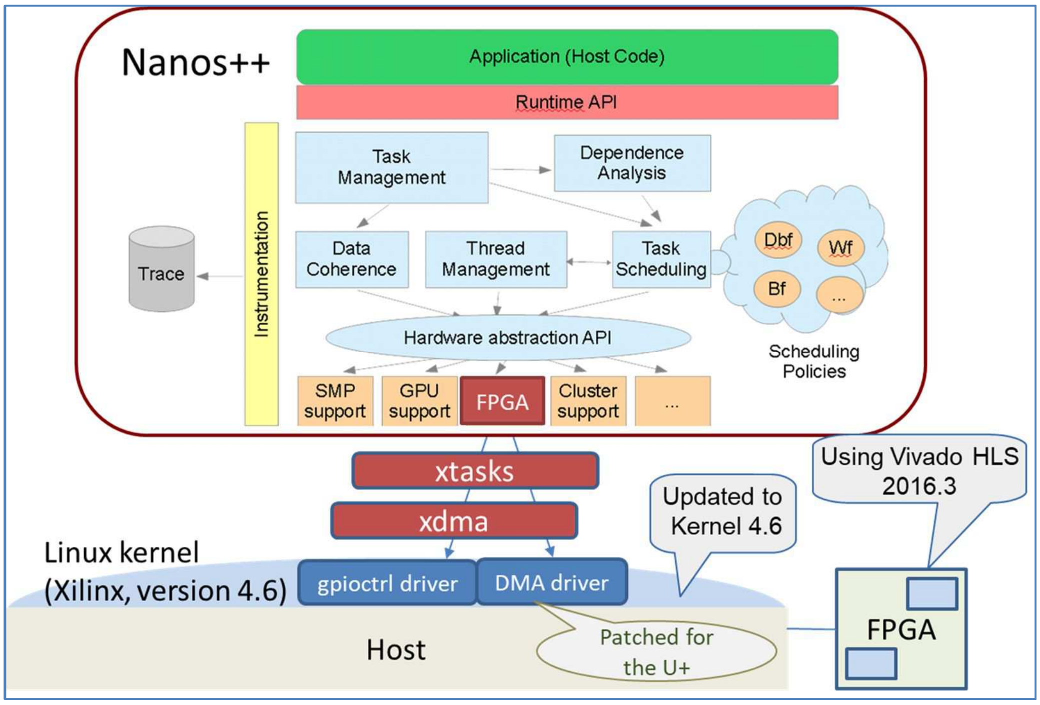


Figure 9. Architecture of the Nanos++ runtime system to enable support for FPGA acceleration

Additionally, the current version of Nanos++ runtime automatically takes care of any necessary copies from/to user space memory to/from pinned kernel memory, thus enabling DMA memory accesses from the FPGA accelerators. This automatic copy greatly increases the productivity of developers, and allows them to completely avoid pinned kernel memory on the copies from the accelerators. However, the compiler and the Nanos++ runtime also provide other mechanisms for helping advanced developers to deal with the kernel memory manually. This helps also developers on improving performance of the SMP part of code. This is due to the fact that kernel pinned memory currently has to be non-cacheable to avoid issues in the PS/PL coherence system. Therefore, in the case that developers simply wanted to directly use and manage the kernel pinned memory (non-cacheable), they would have had to deal with the data movements to cacheable memory to allow the SMP part of code to exploit the memory hierarchy of the SMP. This additional step is avoided with the infrastructure developed.

2.2.1 Execution environment

In order to execute the program, it is only required to call the binary with its corresponding arguments, as it is usually done:

```
./program  <input arguments>
```

In the case in which instrumentation is enabled, it is required to set up the runtime library with instrumentation support enabled, and also to specify the type of instrumentation to be used. We allow two types of instrumentation: *Extrae* (runtime and user application at SMP), and `ompt` support (on the FPGA). As an example, the following environment variables enable `ompt` instrumentation:

```
LD_PRELOAD=libomptrace.so \

   NX_ARGS="--instrumentation=ompt" ./program <input>
```

Additionally, the clock frequency of the accelerators synthesized on the FPGA must be also specified. This is due to the fact that the timestamp of the SMP must be synchronized with the timestamp of the FPGA accelerators. On top of that, it is also required to provide an `extrae.xml` configuration file:

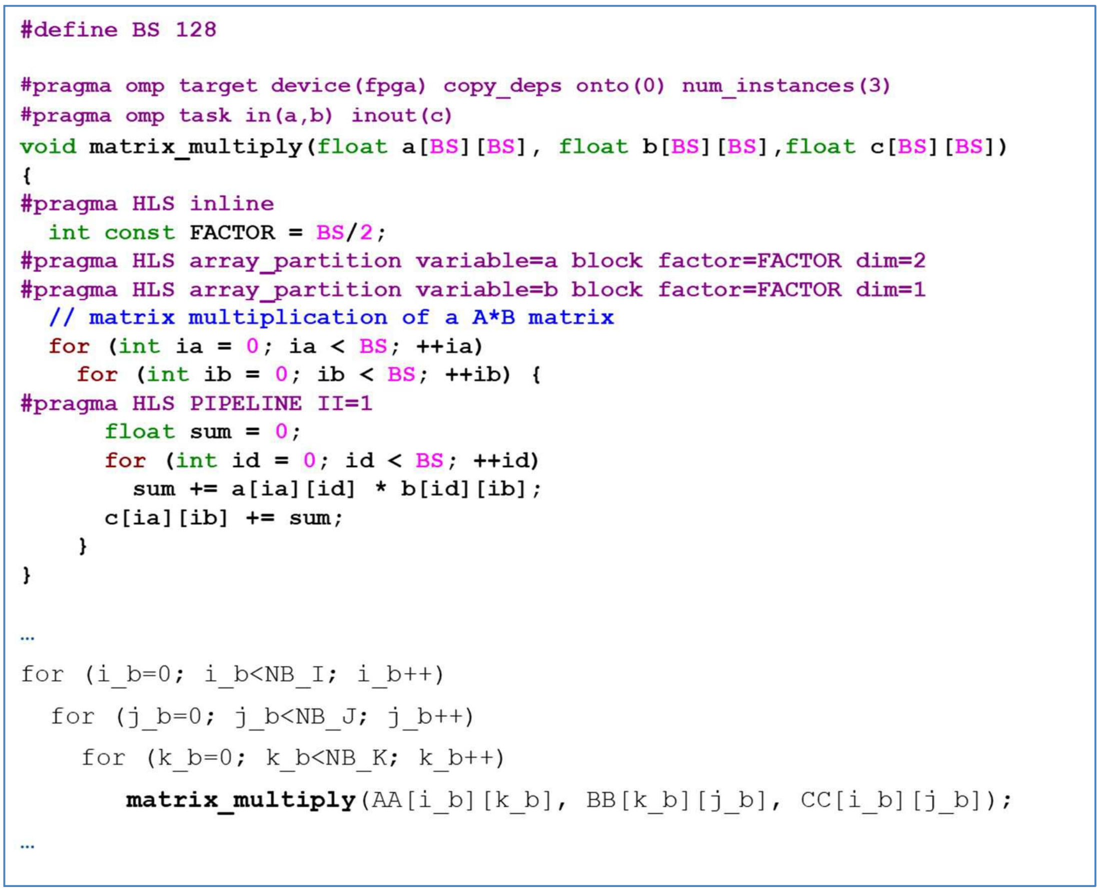
```
NX_ARGS="--fpga_freq=200"

export EXTRAE_CONFIG_FILE="extrae.xml"
```

2.3 Impact of the achievement on project/WP

The development of the compilation toolchain for the FPGA has allowed partners HERTA and VIMAR to port their applications to the FPGA device with minimal efforts. The details of the porting process are described in Deliverable D3.3 - Software and Report on Application Porting.

3 Evaluation and tuning of the FPGA compilation system

The FPGA compilation system was evaluated with the single precision matrix multiplication benchmark, which is a fundamental operation in convolutional neural networks (CNNs). Figure 10 shows the source code of this benchmark including both OmpSs and Vivado HLS annotations.



```
#define BS 128

#pragma omp target device(fpga) copy_deps onto(0) num_instances(3)
#pragma omp task in(a,b) inout(c)
void matrix_multiply(float a[BS][BS], float b[BS][BS],float c[BS][BS])
{
#pragma HLS inline
  int const FACTOR = BS/2;
#pragma HLS array_partition variable=a block factor=FACTOR dim=2
#pragma HLS array_partition variable=b block factor=FACTOR dim=1
  // matrix multiplication of a A*B matrix
  for (int ia = 0; ia < BS; ++ia)
    for (int ib = 0; ib < BS; ++ib) {
#pragma HLS PIPELINE II=1
      float sum = 0;
      for (int id = 0; id < BS; ++id)
        sum += a[ia][id] * b[id][ib];
      c[ia][ib] += sum;
    }
}

…
for (i_b=0; i_b<NB_I; i_b++)
  for (j_b=0; j_b<NB_J; j_b++)
    for (k_b=0; k_b<NB_K; k_b++)
      matrix_multiply(AA[i_b][k_b], BB[k_b][j_b], CC[i_b][j_b]);
…
```

Figure 10. Matrix multiply with OmpSs and Vivado HLS annotations

Several configurations of IP cores were tested on the FPGA using different clock speeds. The following table summarizes the experiments conducted.

IPs configuration	1*256, 3*128	Number of instances * size
Frequency (MHz)	200, 250, 300	Working frequency of the FPGA
Number of SMP cores	SMP: 1 to 4 FPGA: 3+1 helper, 2+2 helpers	Combination of SMP and helper threads
Number of FPGA helper threads	SMP: 0; FPGA: 1, 2	Helper threads are used to manage tasks on the FPGA

Number of pending tasks	4, 8, 16 and 32	Number of tasks sent to the IP cores before waiting for their finalization

Figure 11 shows the evaluation in GFlops of the different alternatives of matrix multiplication configurations for 2048x2048 matrices using different block sizes: 128x128 and 256x256. On the SMP cores, matrix multiplication used the OpenBLAS SGEMM kernel to multiply the matrices in parallel in the same blocked fashion as the application did on the FPGA. A single ARM Cortex A53 core delivered roughly 3 GFlops (Figure 11 "No IP" bars 1 core). Similarly, four ARM cores achieved 11.7 GFlops, showing good scalability of the OmpSs infrastructure on the SMP environment.

When adding an IP core (performance-oriented hardware design) of block size 256x256 running at 200 MHz, the performance was boosted to 25.8 GFlops. The 200 MHz FPGA implementation of the 256x256 block added up to 14 GFlops. In this latter case, one helper thread was used to execute FPGA tasks but also SMP tasks if there were not enough FPGA tasks for execution or the FPGA was busy, and 3 worker threads were running SMP tasks. This was due to the `implements` clause shown above, thus allowing a heterogeneous parallel execution of tuned tasks for both SMP and FPGAs.

Starting from this point, we increased the accelerator frequency to 250 and 300 MHz, which showed also an additional boost in performance. Additionally, we allowed the runtime system to provide up to 16 tasks to the FPGA before waiting for the previous tasks to be finished. Tuning this value also provided a further increase on performance. Specifically, for block sizes of 256x256 elements running at 300 MHz, the performance increased from 32.9 to 35.7 GFlops. This additional improvement was obtained from the reduction in the number of synchronizations needed by the runtime. Reducing the amount of synchronizations contributed to reduce the overhead of task management.

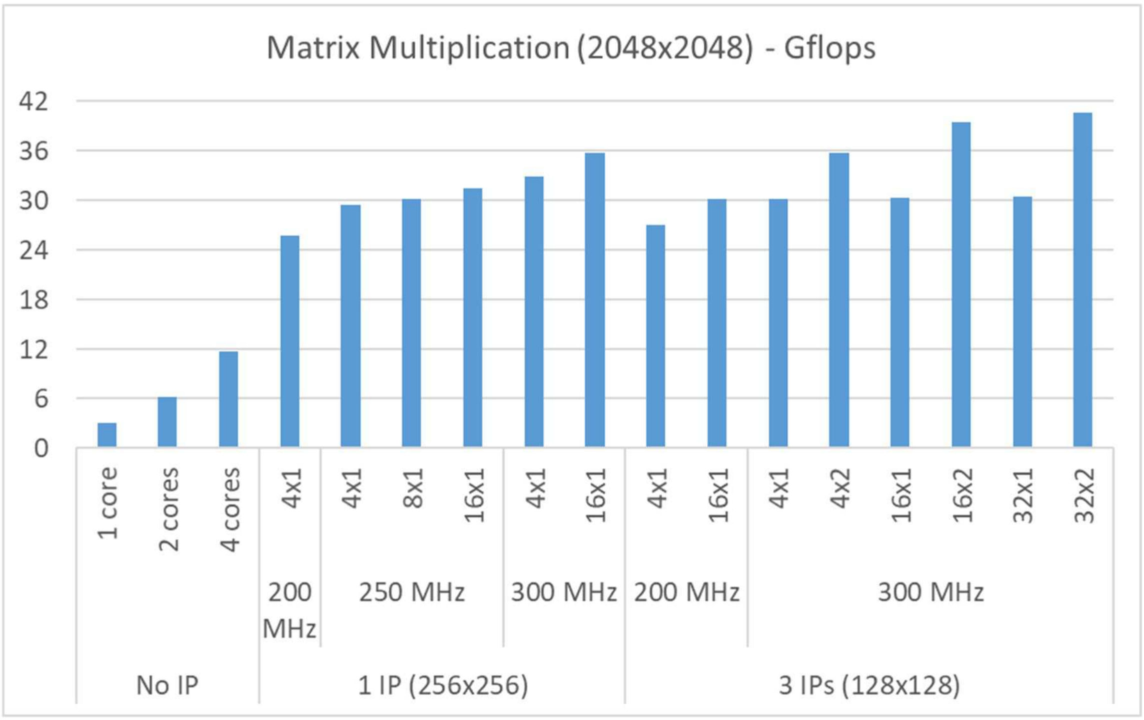


Figure 11. Evaluation of matrix multiplication on the AXIOM board

Figure 11 also shows the performance of 3 IP cores running in parallel on the FPGA. The additional parallelism yielded better performance. When running at 200 MHz, and with up to 16 pending tasks, the performance was equivalent to the one using a block size of 256x256 elements and 8 pending tasks. When moving to 300 MHz, the runtime behavior showed that it is important to have at least 2 helper threads available. Otherwise, the performance would have not been improved at all, since having only one helper thread did not provide enough FPGA tasks to the accelerators. In any case, those two helper threads are implemented in a hybrid way. That is, they can decide to run SMP tasks if the accelerators are busy, in addition to the two worker threads that are also executing SMP tasks. This combination of helper and worker threads achieved the best performance of the MxM. Observe that the combinations of 4x2, 16x2, and 32x2 (pending tasks and helper threads) keep increasing in performance, while the alternatives of 4x1, 16x1 and 32x1, which used a single helper thread, did not scale anymore.

We have also analyzed the internals of the execution on the FPGA IP cores. Figure 12 shows the usage of the 3 FPGA IP cores implementing block matrix multiplications of 128x128 elements. The execution environment uses 2 SMP cores to execute matrix multiplication tasks (THREAD 1.1.1, and 1.1.16), and 2 helper threads (THREAD 1.1.14, and 1.1.15). Lines labeled DMA_submit_* show the activation of the corresponding IP core (0, 1, or 2). DMA_in_* represent the data movement going into the FPGA IP. FPGA_accelerator_* indicates the time while the block by block computation is done. Finally, DMA_out_* indicates the execution of the data transfer out of the IP core.

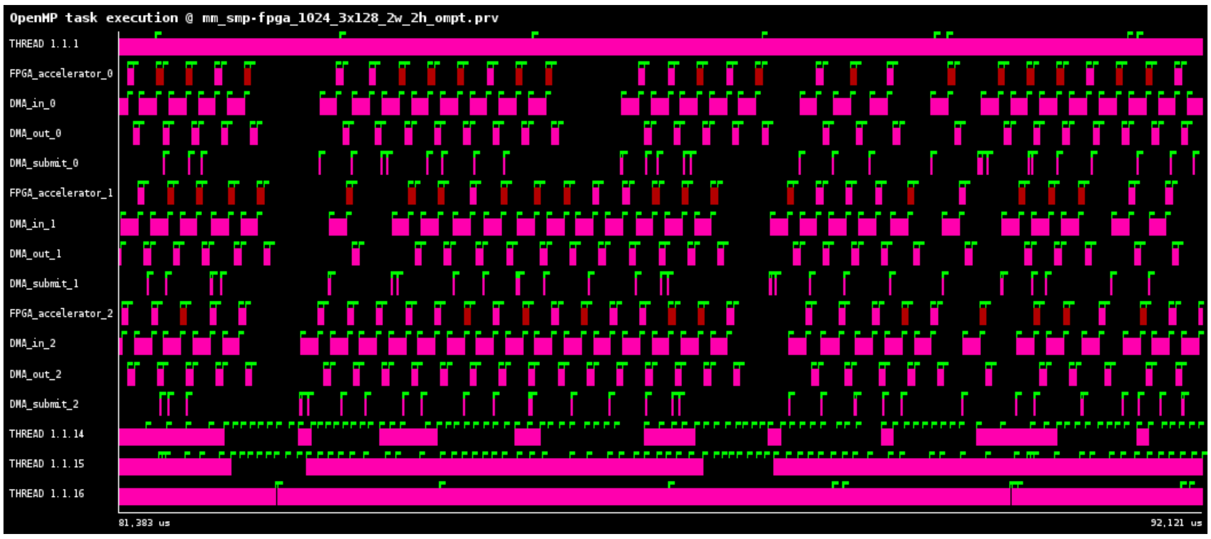


Figure 12. Execution internals of the FPGA IPs, on a matrix multiplication of 1024x1024 single precision elements, in blocks of 128x128, showing the behavior of the use of 3 FPGA IPs

As it can be observed, data input to the FPGA IP cores is the most consumed resource, while the execution of the block matrix product, and the data output took very similar elapsed times. Further improvements to achieve better performance would involve trying to schedule data transfers towards the FPGA using different policies: for example, including prefetch techniques to send data in advance of the task execution.

4 Evaluation and tuning of the OmpSs@Cluster system

In this section, we present the tuning and evaluation of OmpSs@Cluster. As part of the work related to OmpSs code optimizations, partners EVI and BSC performed an analysis of the bottlenecks currently present in the generic OmpSs@Cluster source code when applied to the AXIOM software stack. In particular, possible bottlenecks were identified, analyzed and solved improving performance when possible.

4.1 Communication Thread vs Hybrid communication Thread

In order to work, the OmpSs@Cluster needs to perform communication with the various nodes in the cluster. As noted in D5.3, this communication is implemented using the GASNet framework, which allows a platform independent message exchange and memory handling between nodes.

The GASNet framework is organized with “conduits”, to support multiples communication protocols (e.g., UDP, MPI, InfiniBand). As described in D5.2, EVI developed a new conduit on top of the AXIOM-link, to allow OmpSs@Cluster to work on top of the AXIOM cluster.

The typical setup implemented in OmpSs@Cluster is composed by a set of nodes, typically each one exposing more than one core.

In previous applications of OmpSs@Cluster it was noted that the main limitation for enabling a full utilization of the system was related to the communication latencies between nodes. For that reason, in order to limit the overall latency, it was decided to allocate one core to a communication thread, letting it handle at maximum speed (and minimum latency) the communication between nodes. An active polling strategy was therefore devised in the implementation of GASNet.

Therefore, the standard configuration suggested for OmpSs@Cluster is then composed by N-1 cores “allocated” to OmpSs working threads, with the last core allocated to a “communication thread”.

The polling strategy currently implemented in the GASNet communication thread has two drawbacks:

- On one side, the core where the communication thread is allocated spins continuously, keeping the core active also if there are no actions to perform, impacting on the overall energy consumption.
- On the other side, the fact that a core is allocated to the communication thread removes some possible computational power to each node. Having the possibility to use part of that computational power could have a positive impact on the overall speedup and execution time.

For this reason, the GASNet conduit implementation performed in the AXIOM project included the support for blocking primitives. These blocking primitives are based on the AXIOM network driver notifications (which are interrupt-driven), allowing threads to block waiting for the reception of GASNet messages.

In order to be able to use the blocking primitives, some modifications are required in Nanos++. In particular, Nanos++ uses a thread to handle the exchange of messages and memory copies. That thread is bound to one CPU, and its implementation continuously calls the `gasnet_AMpoll()` function of GASNet. The `gasnet AMpoll()` is the GASNet function responsible for checking the pending messages and run the handlers linked to the active messages.

The idea is then to change the implementation of Nanos++, because continuously calling the `gasnet_AMpoll()` keeps the thread always active and the CPU always busy. In order to do that, the polling function has to be replaced with its blocking primitive, which is named `GASNET_BLOCKUNTIL(cond)`. The implementation of this macro blocks execution of the current thread and keeps handling the network traffic until the provided condition becomes true. Unfortunately, the network polling functions are called in several parts of Nanos++, and the change of the semantics from non-blocking to blocking is not simple, and impacts on the overall architecture of OmpSs@Cluster (mainly because the system, while polling, also does some background bookkeeping activities that with the blocking version have to be moved to other threads).

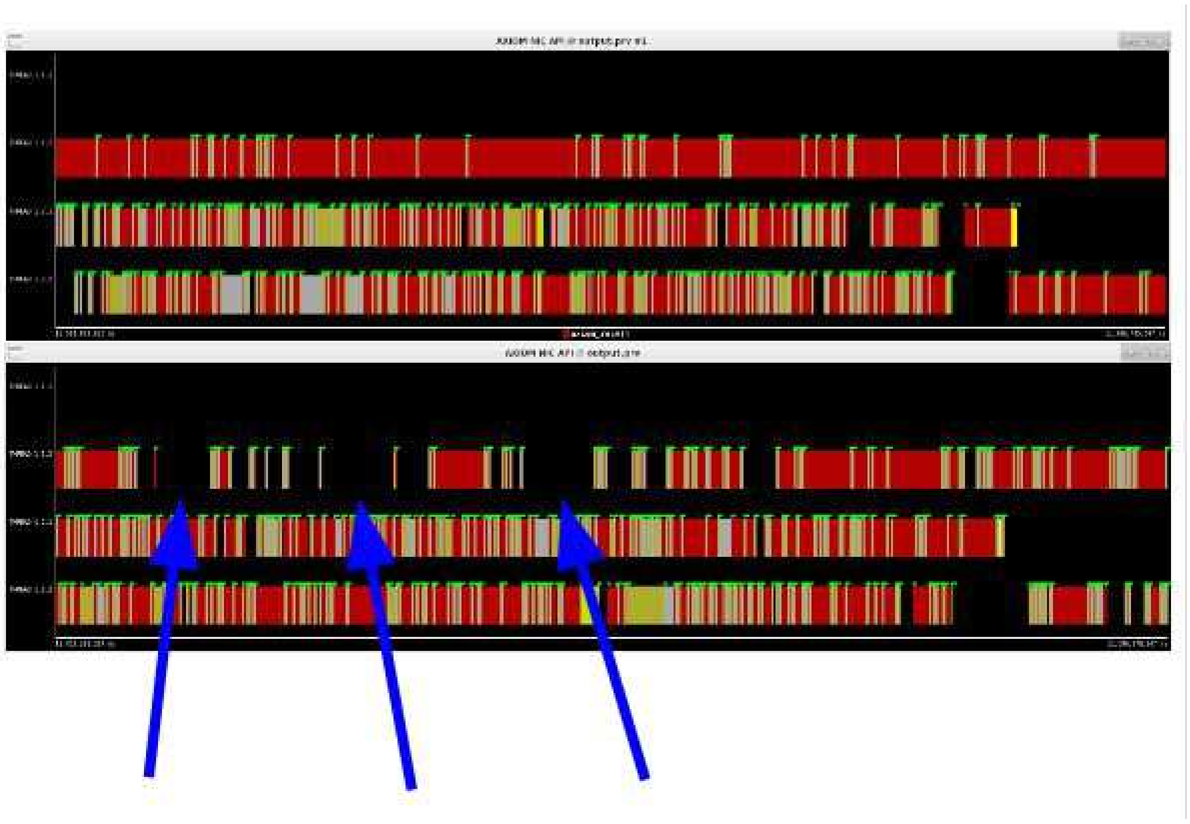


Figure 13. Comparison between two OmpSs@Cluster executions with polling (top) and with blocking (bottom). The black regions in the bottom part are related to CPU time freed due to the usage of blocking instead of polling

As a preliminary result, we tried the substitution of the polling functions with their blocking counterparts in a subset of the locations in the source code. A preliminary result is shown in Figure 13, where we compare two traces recorded with Extrae. At the top, there is a trace with the polling implementation. At the bottom, there is a trace with a preliminary use of blocking API. The main differences are the "holes" in the execution highlighted with the blue arrows where the blocking API allow us to save power and possibly do other work.

Because of the complexity of the changes in Nanos++ to support GASNet blocking APIs, we decided to stop the implementation. An alternative way to limit the CPU bandwidth for the polling thread and share the core between communication thread and one more working thread will be also described in Deliverable D5.4 (that solution involves the usage of `SCHED_DEADLINE` to limit the amount of CPU power used by the communication thread.

Another alternative is to modify the behavior of the dedicated thread for communication (i.e. communication thread) so that it can be also used to execute SMP tasks. The idea is that the thread will not continuously call the `gasnet_AMpoll()` function if there are SMP tasks to be done. That is mostly critical when using accelerators that need helper threads running FPGA tasks. In this case, the number of worker threads running SMP tasks is reduced to the maximum number of cores minus the number of helper threads and communication threads.

We have implemented this idea so that the communication thread can execute SMP tasks as well if there is no work to be done in the communication. The activity of this communication thread will depend on the total number of tasks to be executed, and the number of tasks that we want to send every time that the communication thread decides to look for communication to be done. Figure 14 shows part of a Paraver execution trace of the master node of a cluster execution of the MxM. There, the second thread (i.e. second horizontal line) is the communication thread which submits remote SMP tasks every time it decides to perform communication work. The rest of the time it is executing SMP tasks. That allows the system to exploit all its resources without calling all the time to the `gasnet_AMpoll()` function. We call this configuration of the communication thread, cluster hybrid, meaning that it is capable of executing both communication and work tasks. This approach effectively solves the aforementioned problem using an alternative, more efficient way than the blocking mechanism.

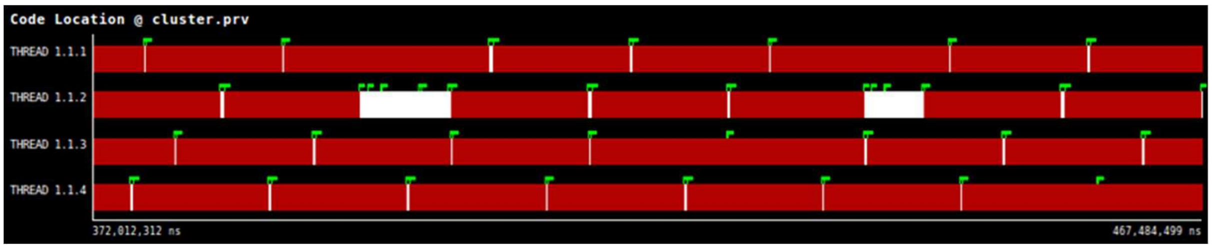


Figure 14. Paraver execution trace visualization of a part of the OmpSs@Cluster execution of the MxM. This trace only shows the master node execution of 4 threads running SMP tasks. The second thread corresponds to the communication thread.

Figure 15 shows the execution time of a MxM of 2048x2048 dimensions in an AXIOM cluster with two nodes, using 3 FPGA accelerators of 128x128 per node. The figure shows four different configurations of the runtime: 1) **sn** means that there is a dedicated communication thread in the slave node and a dedicated communication thread in the master node, both continuously calling the `gasnet_AMpoll()` function; 2) **sh** means that there is a dedicated worker communication thread in the slave node but the communication thread in the master node can submit SMP tasks (i.e. is a hybrid thread); 3) **nn** stands for no dedicated slave worker communication thread neither hybrid master communication thread, and 4) **nh** stands for no dedicated slave communication worker with a hybrid master communication thread.

In the cases in which the hybrid communication thread is not activated, the number of worker threads in both master and remote node is one. Therefore, allowing the communication thread to send SMP tasks, and avoiding a continuous polling from the communication thread, increases the performance almost 1.4X. There is also a worker communication thread, in each remote node that is not affecting the performance significantly. However, we have observed that in situations with less threads (i.e. such as in the original Zynq boards) it also increases performance when this slave dedicated thread is deactivated. In addition, this deactivation will result in energy savings for the **nh** configuration compared to the **sh** one.

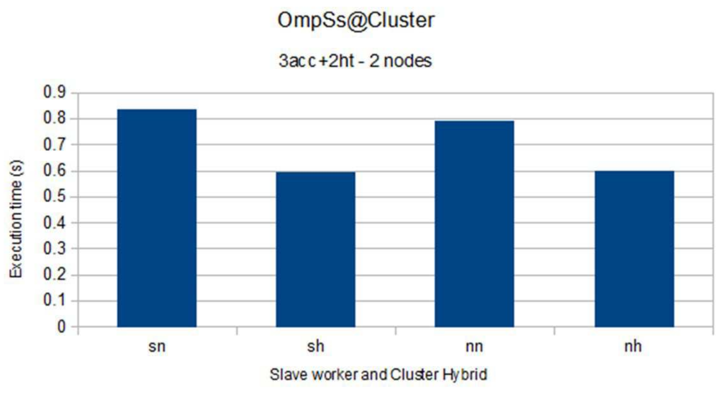


Figure 15. Four different configurations of the communication thread and the slave worker thread for an OmpSs@Cluster MxM application using FPGA accelerators and SMP threads

On the other hand, when using only the FPGA accelerators to perform the MxM, there is no impact of enabling or disabling the hybrid communication threads. Figure 16 shows the evaluation of the benchmark when running FPGA tasks only (no SMP tasks), showing no performance impact of the hybrid implementation.

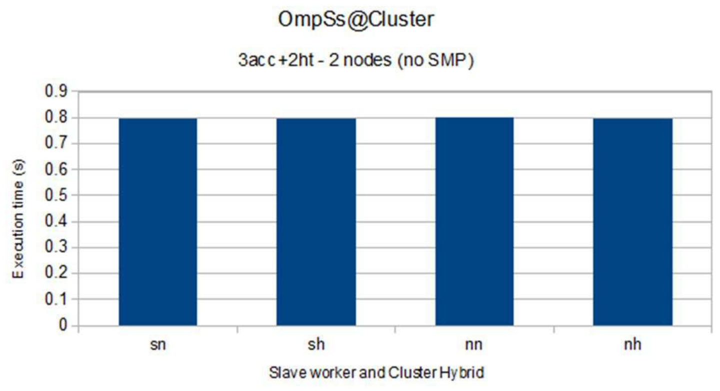


Figure 16. Four different configurations of the communication thread and the slave worker thread for an OmpSs@Cluster MxM application using only FPGA accelerators

On the other hand, when using only SMP tasks, the performance impact of having hybrid or slave dedicated worker is not significant. Figure 17 shows the execution time (s) of two of the four different configurations: **sn** and **nh**, for pre-send of 8, 16, 32 and 64 SMP tasks. The impact of pre-send tasks, number of tasks submitted in a batch to the remote node, is evaluated in next section. Here, although the scale is from 1.12 to 1.24 seconds, and although the performance difference cannot be significant, it can be seen that having hybrid communication threads can help when the pre-send is high. There is a different relationship between the number of tasks to be sent and the work to be done by the communication thread depending on the fact of having or not communication threads able to execute SMP tasks. In the case of using the hybrid approach with 8 pre-send, it may happen that the remote node has nothing to do because hybrid communication threads are executing SMP tasks rather than submitting tasks to the remote node. That fact can explain a small slowdown in performance compared to **nn** (not showed in the figure).

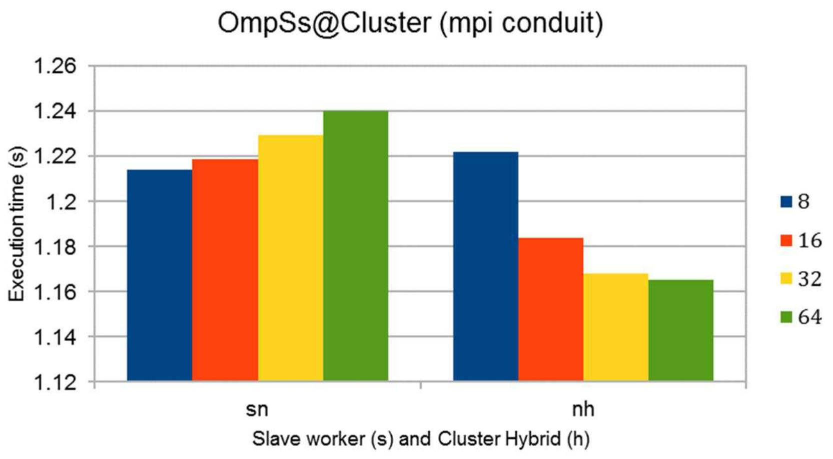


Figure 17. Execution time of an OmpSs@Cluster MxM using two nodes and MPI conduit for two different configurations of the slave worker thread and the communication thread. Different pre-send number of tasks are analyzed

4.2 Communication Thread: Pre-send (remote submit) Impact

The number of tasks offloaded, in a batch, each time the communication thread of the master decides to send tasks to the remote node can have a significant performance impact. Figure 18 and Figure 19 show the performance impact measured as the speedup compared to 1 node of increasing the number of tasks from 1 to 4 in the case of using the dedicated AXIOM communication or from 1 to 32 in the case of using the MPI conduit (Ethernet) communication, respectively. We are using here the **nh** configuration, that is, no dedicated communication thread in the slave node and hybrid thread in the master node.

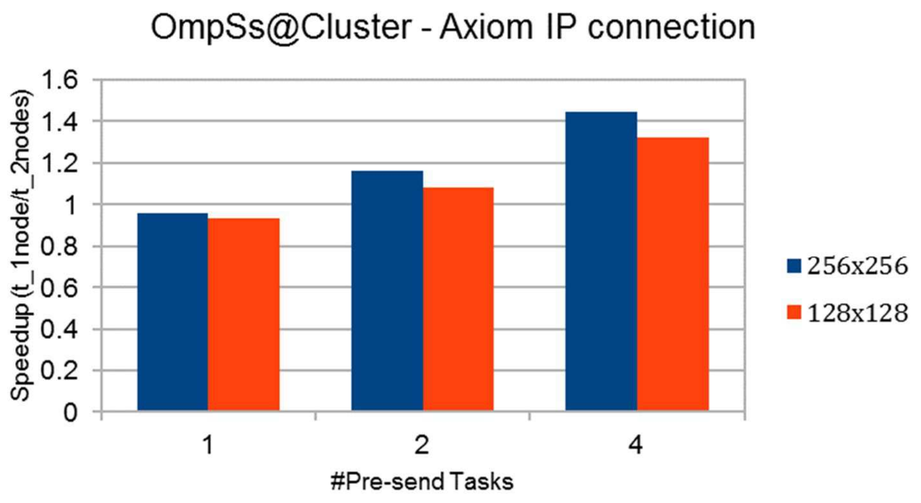


Figure 18. Speedup of the OmpSs@Cluster MxM multiplication when using two nodes compared to one node, both using the maximum number of worker threads available. AXIOM IP conduit is used for performing communications

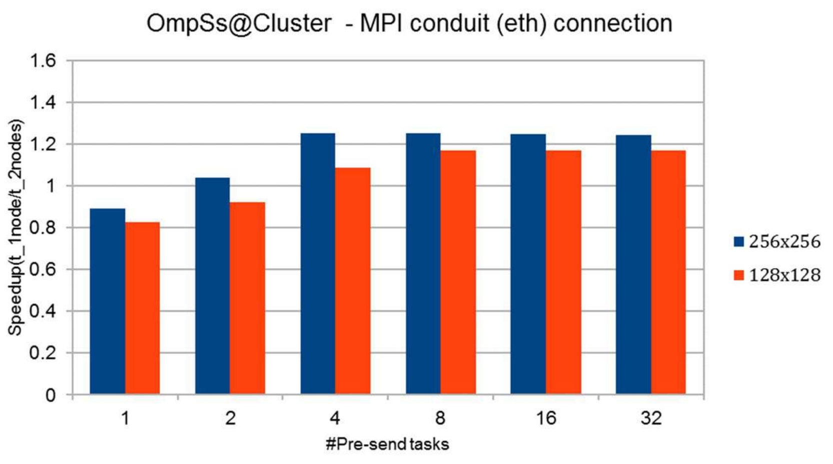


Figure 19. Speedup of the OmpSs@Cluster MxM multiplication when using two nodes compared to one node, both using the maximum number of worker threads available. MPI conduit is used for communications

The speedup achieved in both cases proportionally increases with the number of pre-send tasks. This is mostly due to: 1) The number of effective calls to the `gasnet_AMpoll()` increases due to the higher number of tasks in the remote node, and 2) the number of tasks available in each node increases, thus increasing the possibilities to run tasks in parallel. Note that this improvement is actually better in the case of the dedicated AXIOM communication.

4.3 AXIOM dedicated conduit vs MPI conduit overhead

The AXIOM network interface, developed in the AXIOM project, supports RDMA transactions implemented through RDMA Read, RDMA Write, and LONG messages. These RDMA operations are allowed only on a subset of the available physical memory, which is handled by the AXIOM memory allocator (see Deliverable D5.2).

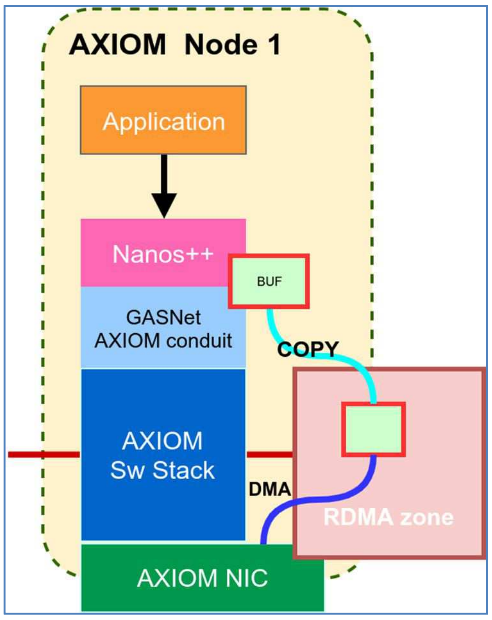


Figure 20. Extra copy in the GASNet AXIOM conduit needed for the AXIOM RDMA

The implementation of GASNet conduit performed in the AXIOM project considers this aspect, and in case the memory region to be copied using RDMA is located outside the RDMA region, then an additional memory copy is performed (see Figure 20). In particular, if Node 1 wants to copy a region of memory outside the RDMA region to Node 2 using the AXIOM RDMA features, then:

- Node 1 first copies the memory inside its RDMA zone (this is done automatically inside the GASNet AXIOM conduit implementation when it detects that the memory region to copy is outside the RDMA region).
- Then the AXIOM APIs are used to do an RDMA transfer to the RDMA region of Node 2.
- Finally, Node 2 copies back the data from its RDMA region to the destination address.

Figure 21 depicts a screenshot of the Lauterbach PowerTrace tool [[8]] which highlights the time spent in the memory copy of a message allocated in a cached region to the RDMA non-cached region. The screenshot is taken on Node 1 during the execution of a matrix multiply example with OmpSs@Cluster on two AXIOM boards.

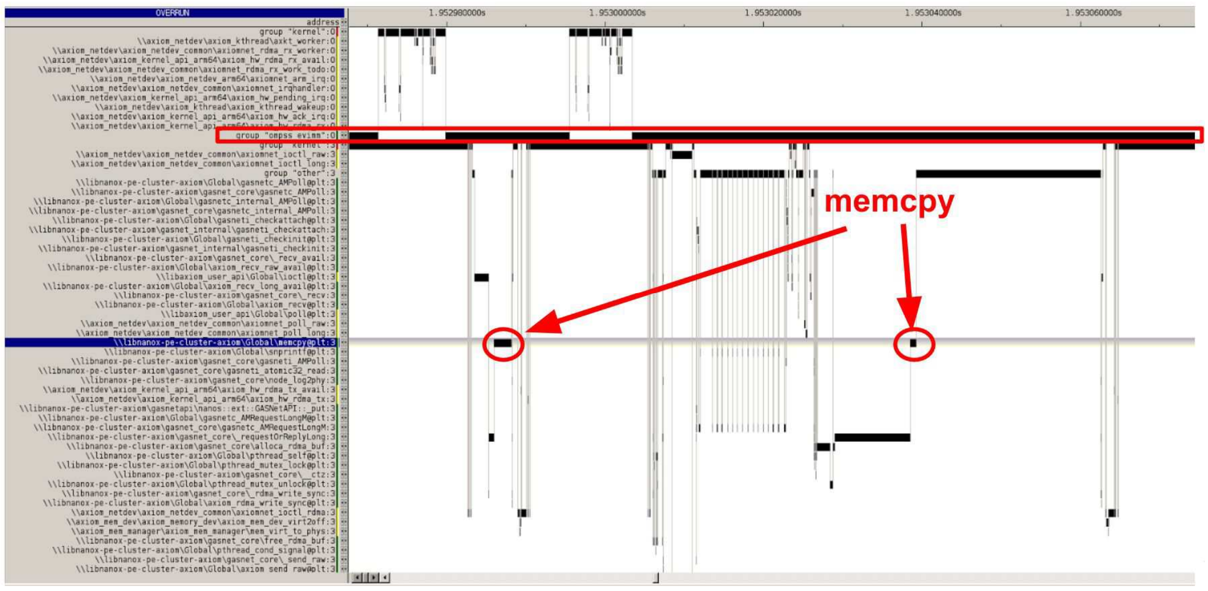


Figure 21. Lauterbach trace that shows the time used by the `memcpy()` to move data in the RDMA region

Figure 22 shows the detail of all the steps done in the master node and the remote node in order to do the submission and execution of a single task. In particular, it is shown, as actors, the different memory spaces used in the execution: from left to right, master user space in the master node, RDMA space in the master node, RDMA space in the remote node and user space in the remote node. Looking at the number of copies: there is a copy from user to RDMA and RDMA to user to copy in data in the remote node, and two other copies to copy out the data. All those copies are necessary with the current implementation of OmpSs@Cluster.

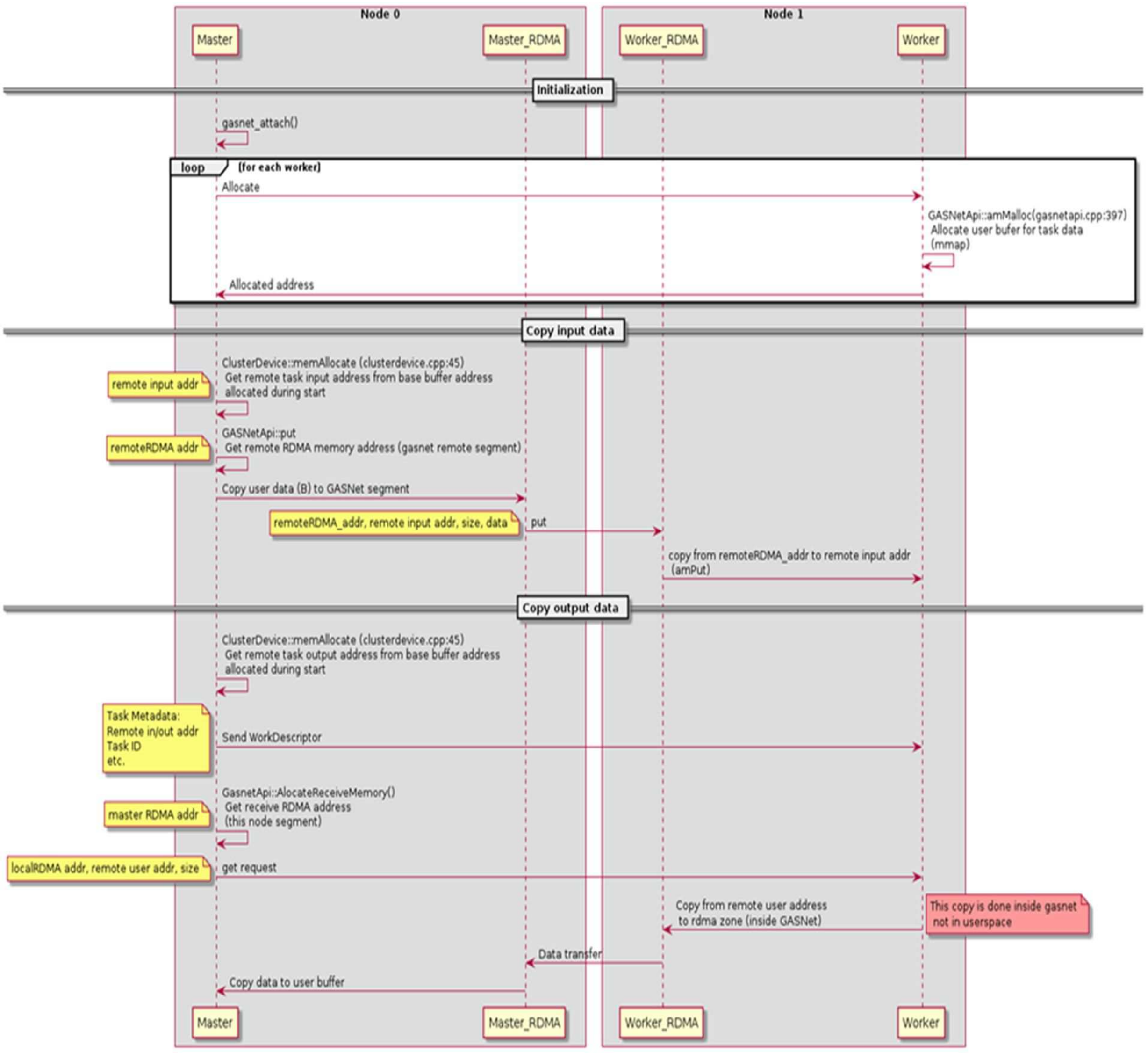

Node 0
Node 1
Master
Master_RDMA
Worker_RDMA
Worker
Initialization
gasnet_attach()
loop [for each worker]
Allocate
GASNetApi::amMalloc(gasnetapi.cpp:397) Allocate user bufer for task data (mmap)
Allocated address
Copy input data
remote input addr
ClusterDevice::memAllocate (clusterdevice.cpp:45) Get remote task input address from base buffer address allocated during start
remoteRDMA addr
GASNetApi::put Get remote RDMA memory address (gasnet remote segment)
Copy user data (B) to GASNet segment
remoteRDMA_addr, remote input addr, size, data
put
copy from remoteRDMA_addr to remote input addr (amPut)
Copy output data
ClusterDevice::memAllocate (clusterdevice.cpp:45) Get remote task output address from base buffer address allocated during start
Task Metadata: Remote in/out addr Task ID etc.
Send WorkDescriptor
master RDMA addr
GasnetApi::AlocateReceiveMemory() Get receive RDMA address (this node segment)
localRDMA addr, remote user addr, size
get request
Copy from remote user address to rdma zone (inside GASNet)
This copy is done inside gasnet not in userspace
Data transfer
Copy data to user buffer

Figure 22. Diagram of actions and copies between the two different memory spaces in the master and remote node for the execution of a single SMP task

The current implementation of OmpSs@Cluster uses a custom allocator to handle the memory allocations used by the nodes. The memory used by the OmpSs memory allocator is not handled by the AXIOM memory allocator. A possible alternative would be directly allocating memory in the AXIOM memory allocator in the RDMA. However, there is a tradeoff to be considered for this possible alternative. In particular, everything depends on whether the RDMA zone is cached or not. Allocating the memory used by the application in a non-cached zone can in fact be worse than having the data allocated in a cached zone and then perform an additional memory copy when needed. At the moment of developing the integration of the AXIOM NIC with OmpSs@Cluster, the support for cache coherency in the implementation of the AXIOM NIC was not available. Indeed, there are also known issues with the management of cached pinned memory, necessary for FPGA executions that are not yet solved by the FPGA vendor. Due to those reasons, we have decided to keep the two memory spaces separated.

In any case, this overhead is not significant and the performance improvement of the AXIOM platform over the MPI conduit is significant. Figure 23 shows the speedup of the execution of the MxM using two nodes, and different block sizes (128x128 and 256x256), using the AXIOM NIC compared to the same application using the MPI conduit. The overall improvement is more than 1.2X when the number of tasks is large (128x128 case), and the number of pre-send tasks is 4.

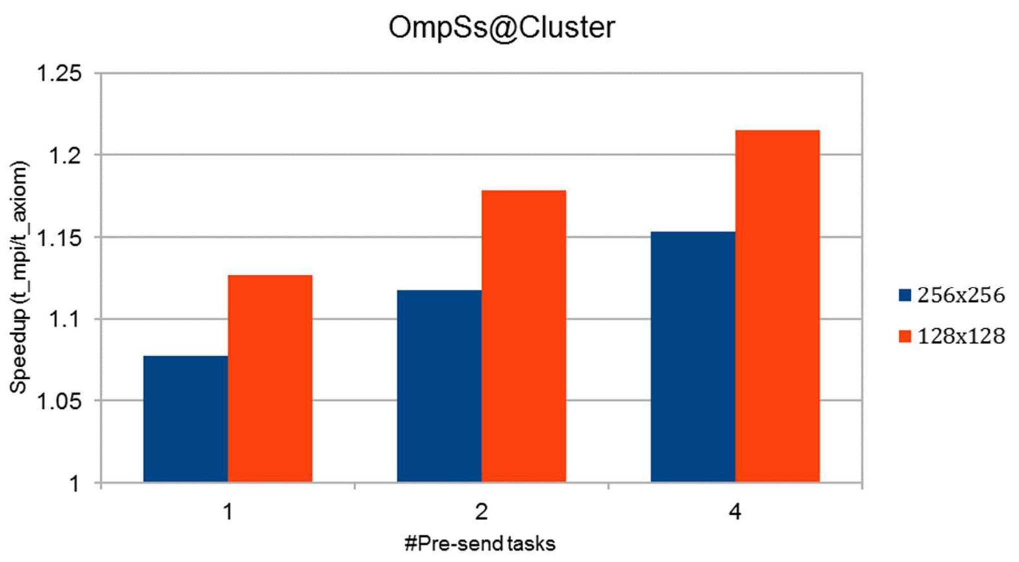


Figure 23. Speedup of the OmpSs@Cluster MxM using axiom NIC compared to using MPI conduit (Ethernet) for two different blocking sizes, and different number of pre-send tasks

5 Confirmation of DoA objectives

PLANNED	DELIVERED
DELIVERABLE: D4.3	
• Results of task T4.5 evaluation and tuning of the code generation.	OmpSs providing support for the FPGA IP cores and cluster environments.
• Instrumentation internals from the FPGA.	Described in deliverables D4.2, D5.4, and also shown in this deliverable.

6 Conclusion

This deliverable describes the design decisions undertaken for the implementation of the Mercurium compiler and required tools for targeting the AXIOM board (i.e. Nanos++, and autoVivado).

Current implementation provides a very good scalability for the tested kernel: matrix multiply. Moreover, the obtained results show that such scalability is achieved in both intra an inter-node resources of the board (i.e. SMPs and FPGA). To obtain these results, a first version of the framework was developed and evaluated on the actual board. After this work was carried out, an extensive optimization was conducted to exploit the underlying hardware resources of the AXIOM board, and thus get an additional performance boost. As a result, we proved that the same C program annotated with OmpSs directives can leverage the heterogeneous resources of the board: ARM cores and the FPGA reconfigurable logic.

Additionally, we also demonstrated that it was possible to simultaneously exploit all the resources of a cluster of AXIOM boards (i.e. SMPs, FPGAs and the AXIOM conduit) while keeping scalability. In order to solve this major challenge, we relied on our AXIOM NIC, which was a crucial component required for further improving performance, and proved that it outperforms other classical approaches such as the MPI Ethernet conduit. Also, these experiments proved the feasibility of using HPC-tailored programming models such as OmpSs to efficiently target embedded platforms specifically designed for IoT environments.

Other related publications of this project can be found in the references [10]-[38].

References

[1] AXIOM Consortium, “D4.2 – AXIOM Code Generation and Instrumentation”, February 2016.

[2] 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).

[3] Germán Llort, Antonio Filgueras, Daniel Jiménez-González, Harald Servat, Xavier Teruel, Estanislao Mercadal, Carlos Álvarez, Judit Giménez, Xavier Martorell, Eduard Ayguadé, Jesús Labarta: “The Secrets of the Accelerators Unveiled: Tracing Heterogeneous Executions Through OMPT”. IWOMP 2016: 217-236.

[4] OmpSs website: http://pm.bsc.es/ompss

[5] Xilinx Inc. “Xilinx High-Level Synthesis”, https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, 2017.

[6] Xilinx Inc. “Vivado Design Suite – HLx Editions”, https://www.xilinx.com/products/design-tools/vivado.html, 2017.

[7] Xilinx Inc. “Vivado Design Suite Tcl Command Reference Guide”, UG835, April 2017, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug835-vivado-tcl-commands.pdf

[8] Lauterbach GmbH, “PowerTrace Tools”, http://www.lauterbach.com/product-overview_flyer_web.pdf

[9] Balart, Jairo, et al. "Nanos mercurium: a research compiler for openmp." Proceedings of the European Workshop on OpenMP. Vol. 8. 2004.

[10] 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. ISBN: 978-1-61804-276-7

[11] R. Giorgi, "Exploring Future Many-Core Architectures: The TERAFLUX Evaluation Framework", Elsevier, 2017, pp. 33-72. DOI:10.1016/bs.adcom.2016.09.002

[12] R. Giorgi, "Transactional Memory on a Dataflow Architecture for Accelerating Haskell", WSEAS Trans. Computers, vol. 14, 2015, pp. 546-558. ISSN: 1109-2750

[13] 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, Ischia, Italy, May 2015. DOI:10.1145/2742854.2742896

[14] 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

[15] 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,

[16] D. Theodoropoulos, D. Pnevmatikatos, C. Alvarez, E. Ayguade, J. Bueno, A. Filgueras, D. Jimenez-Gonzalez, X. Martorell, N. Navarro, C. Segura, C. Fernandez, D. Oro, J. Saeta, P. Gai, C. Scordino, A. Rizzo, R. Giorgi, "The AXIOM project (Agile, eXtensible, fast I/O Module)", IEEE Proc. 15th Int.l Conf. on Embedded Computer Systems: Architecture, MOdeling and Simulation, July 2015, pp. 262-269.DOI: 10.1109/SAMOS.2015.7363684

[17] 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

[18] 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

[19] 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.

[20] R. Giorgi, "Scalable Embedded Systems: Towards the Convergence of High-Performance and Embedded Computing", Proc. 13th IEEE/IFIP Int.l Conf. on Embedded and, Oct. 2015, pp. 148-153. DOI:10.1109/EUC.2015.34

[21] 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

[22] P. Burgio, C. Alvarez, E. Ayguade, A. Filgueras, D. Jiminez-Gonzalez, X. Martorell, N. Navarro, R. Giorgi, "Simulating next-generation Cyber-physical computing platforms", Ada User Journal, vol. 36, no. 4, Dec. 2015, pp. 259-263. ISSN: 1381-6551

[23] 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, (article ID 3190234). DOI:10.1155/2016/3190234

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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. ISSN: 1826-9745

[31] 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

[32] 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

[33] 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. DOI:10.1109/MECO.2017.7977173

[34] 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

[35] 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.

[36] J. Bosch Pons, "Asynchronous runtime for task-based dataflow programming models." Jul. 2017. Master's Thesis. Universitat Politecnica de Catalunya.

[37] S. Mazumdar and R. Giorgi. "A Survey on Hardware and Software Support for Thread Level Parallelism." arXiv preprint arXiv:1603.09274 (2016).

[38] 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.