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

PROJECT NUMBER: 645496

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

D4.2 – AXIOM Code Generation and Instrumentation

Due date of deliverable: 31st January 2017
Actual Submission: 7th February 2017 (agreed extended date)

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	16.01.2017	Daniel Jiménez	BSC/UPC	Initial version
0.2	16.01.2017	Carlos Álvarez	BSC/UPC	Review version
0.3	16.01.2017	Xavier Martorell	BSC/UPC	Review version
0.4	22.01.2017	Xavier Martorell	BSC/UPC	XSMLL exploration
0.5	24.01.2017	Paolo Gai	EVIDENCE	OmpSs@Cluster on QEMU
0.6	25.01.2017	Daniel Jiménez	BSC/UPC	Review-version
0.7	01.02.2017	Xavier Martorell	BSC/UPC	Include comments from reviewers
1.0	02.02.2017	Xavier Martorell	BSC/UPC	Final version

Release Approval

Name	Role	Date
Xavier Martorell	WP Leader	02.02.2017
Roberto Giorgi	Project Coordinator for formal deliverable	04.02.2017

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

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

Paolo Gai
R&D Department
Evidence– AXIOM

TABLE OF CONTENTS

TABLE OF FIGURES

GLOSSARY

ARM – Instruction set architecture developed by ARM Holdings Ltd.
ASIC – Application-specific integrated circuit
ATLAS – Automatically tuned lineal algebra software
BLAS – Basic linear algebra subprograms
CNN – Convolutional neural network
DTS – Device tree source, a description of the hardware peripherals in Linux/QEMU
FC – Fully-connected layer
GLFW – Open-source multiplatform library for OpenGL, OpenGL ES and Vulkan
IP – Intellectual property
LibAv – Open-source libraries derived from the FFmpeg project to handle multimedia data
LBP – Local binary pattern
LRN – Local response normalization
Mali – A GPU microarchitecture developed by ARM Holdings Ltd.
Mercurium – OmpSs compiler
MKL-DNN – Intel Corporation math kernel libraries for deep neural networks
Nanos++ – OmpSs runtime
NEON – SIMD extensions for the ARM instruction set
NDA – Non-disclosure agreement
OpenGL ES – Reduced specification of the OpenGL standard that targets embedded devices
PL – Programmable logic
PReLU – Parametric rectified linear unit
Qt – Cross-platform application framework developed by The Qt Company
ROC – Receive operating characteristic
RTL – Register transfer language
RTSP – Real-time streaming protocol
SIMD – Single instruction, multiple data
SMT – Simultaneous multithreading
SoC – System on chip
SGEMM – Single-precision floating-point general matrix multiply
STD – Standard deviation
SVS – Smart surveillance scenario for the AXIOM board
SHL – Smart home living scenario developed for the AXIOM project

Executive summary

This document describes the compiler and runtime support for the programming model extensions proposed for programming the AXIOM cluster with OmpSs [3,7], the instrumentation design for monitoring/tracing FPGA devices and the compiler design exploration to support the distributed cluster environment using XSMLL[6].

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 is 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 deliverable D4.1 to take care of data transfers between the host memory and the FPGA devices.

In the deliverable D4.1 we also presented the support for distributed cluster environments based on the Nanos++ runtime system and its connection to the communication layer, through specific communications software. There, we considered the use of common tools, like MPI or GASNet, and also implementing our specific approach based on the XSMLL infrastructure (cf. deliverable D7.1). In this deliverable, we evaluate the OmpSs@Cluster using 2 emulated Zynq Ultrascale+ nodes on QEMU and present the approach to follow to generate code from OmpSs applications to the XSMLL runtime system, using the Mercurium compiler.

We have used the initial evaluation of the low level communication mechanisms to transfer data to and from the FPGA devices (i.e., to/from the AXIOM accelerators) shown in D4.1, to implement the OmpSs Extensions FPGA support. Indeed, the instrumentation support implemented and presented in this deliverable has increased the overall insight analysis, helping to improve the OmpSs@FPGA runtime support and the application performance. On the other hand, we have evaluated the impact of using different mapping of the data in the OmpSs@FPGA applications, implementing the best one as the default choice of the first automatic hardware generation. It is future work to include new features to tune the OmpSs applications in the deliverable D4.3. For instance, using cache capabilities and include a hardware/software partition design in the compiler generation.

1 Introduction

1.1 Document structure

This deliverable is organized as follows:

- Section 1 describes relation to other deliverables and tasks involved in it.
- Section 2 describes the compiler support for the programming model extensions proposed for programming the AXIOM cluster with OmpSs. Also, we present the XSMLL design exploration compiler support to generate code for distributed environments running for the XSMLL model.
- Section 3 describes the instrumentation support at software and hardware level to be able to trace FPGA device acceleration. Also, it explains the instrumentation support developed to analyze the OmpSs@Cluster communication on the AXIOM cluster.
- Section 4 shows the evaluation of the automatic generated OmpSs@FPGA using the instrumentation of the hardware acceleration of those accelerated applications in a Zynq-7000 SoC. Also, we present some preliminary results of OmpSs@Cluster on an AXIOM cluster (emulated with QEMU), using the GASNet Conduit implemented by Evidence over the API of the interconnection network implemented by FORTH.

1.2 Relation to other deliverables

This document describes the implementation of the programming model extensions, presented in the document "MS41 – Definition of the Programming Model Extensions" and Deliverable D4.1 "Programming model extensions". This deliverable is also aligned with D5.1 where the driver for the High-speed B2B interconnect is integrated/configured in the Linux distribution, and the deliverable D7.1 where different tools are presented.

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
- T4.3: Instrumentation of accelerated code
- T4.4: Code generation for the AXIOM environment
- T5.3: Parallel programming library

2 AXIOM OmpSs Compiler Support

In this section, we first briefly describe the OmpSs Programming Model [3], already introduced in D4.1. Then, we explain the extensions planned for OmpSs, to spawn tasks in the FPGA-device (an AXIOM accelerator), and their compiler support. Finally, the compilation support for the extensions needed to support the cluster version are explained for OmpSs@Cluster and a design exploration is

presented on how to transform application annotated with OmpSs directives, onto the XSMLL runtime system.

2.1 OmpSs Extensions

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 on the FPGA, 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 executing 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 of D4.1 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 [7] 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. This deliverable presents how the compiler has been extended to automatically generate and 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.

- Additionally, OmpSs can use the Extrae tool to generate execution traces that can be later visualized with the Paraver tool, and analyze their behavior. In Section 3 we present the instrumentation support for FPGA acceleration analysis. This support has been added at software (Extrae, Nanos++ runtime and Mercurium compiler) and hardware level in the FPGA. Both Extrae and Paraver are also developed at BSC. This complements the evaluation tools developed in WP7 (see Deliverable D7.1).

The following figure shows the existing syntax used in the target and task directives in OmpSs.

```
#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 directive*

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 the runtime system can exploit tasks in parallel. Commutative indicates the tasks that can be executed sequentially, but in any order (possibly different from the creation order).
- *priority(P)* is used to specify the importance of the task. It is a hint to the scheduler, which 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 where 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 to generate VHDL, finally the Vivado tool generate the bitstream for the FPGA [10]. Invoking tasks on the FPGA will be done by the Nanos++ runtime system by sending the data needed, executing by the FPGA device, and getting the resulting data back to the host memory.

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

2.2 OmpSs@FPGA Support Implementation

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 notify the Mercurium compiler that the function annotated is to be compiled with the Xilinx Vivado HLS compiler, for the FPGA, 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 available processors/devices.

The DMA library interface [5] 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.

In this section, we describe the automatic compilation to ARM32 binary and compatible bitstream with all the accelerators (tasks with target device fpga) from an OmpSs annotated application.

Following Figure shows the main phases of the bitstream generation and compilation of the OmpSs code. With this extension, the compiler generates the code for the runtime system specifying the tasks that should be run in the FPGA device. The code is compiled with a backend compiler (e.g., gcc) that will be executed in the Zynq-ARM cores.

This binary code (OmpSs.elf in Figure 2) will call the Nanos++ runtime with FPGA execution support. This support is based on the DMA library and the FPGA-DMA driver in the system. Indeed, the tasks with target device(fpga) are extracted, modified and, using the Xilinx toolchain transparently to the programmer, the bitstream with the FPGA accelerators is automatically generated.

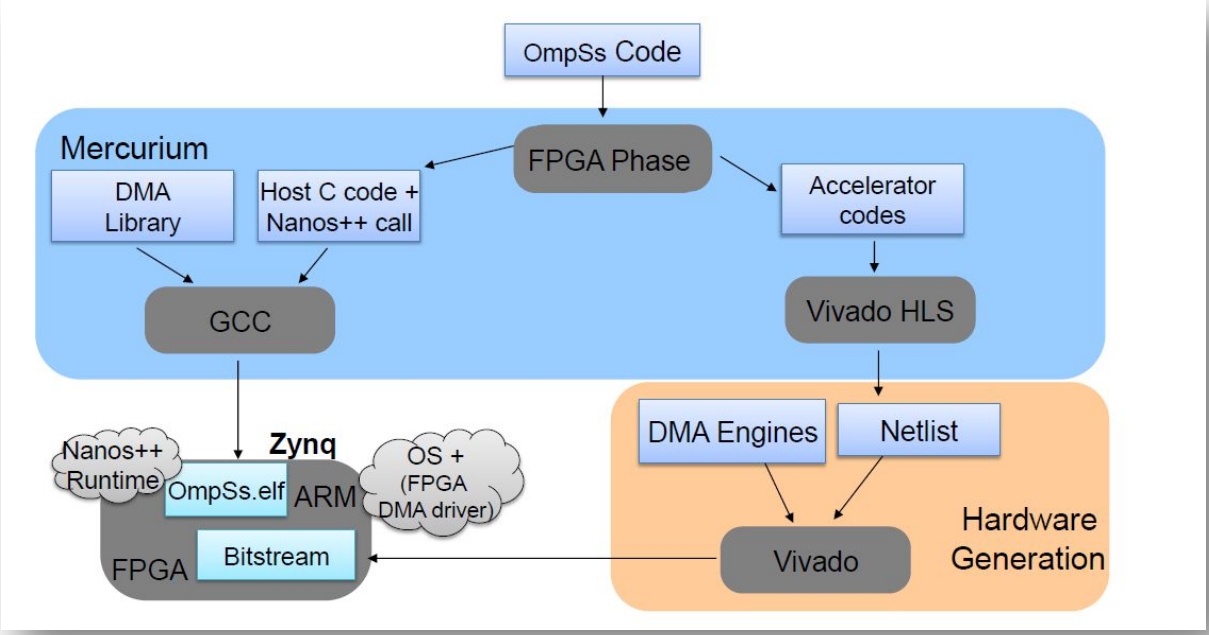


Figure 2. *Compilation Flow*

The runtime support to spawn FPGA tasks is very aligned with the FPGA accelerator interface. They should be able to understand each other. The runtime support has been extended to support for the target clauses related to data transfers. Data-copy clauses (copy_in, copy_out, copy_inout) trigger the data transfers to/from the FPGA device. In addition, dependence clauses will trigger data transfers to

the device by default and include the FPGA device in the support of the implements clause to allow several implementations of tasks to be scheduled in the available processors/devices.

The Nanos++ runtime now invokes the services of the DMA library, which is developed to transfer data in the FPGA environment. In terms of FPGA support, the DMA library interface provides the tools to interact with the Linux driver supporting the FPGA device. In the current prototype, when the data transferred to the FPGA hardware, the IP kernel is initiated automatically. The computation on the data proceeds to the end, and after finishing, the results can be read back to the host from the FPGA. The main DMA library primitives allow getting the number of IP accelerators present in the FPGA device, and the handles to operate with them. For each IP accelerator, the library allows to open input and output stream DMA channels to send/receive data to/from it. The library allows allocating special memory buffers in kernel space to exchange data between the Linux kernel and the FPGA hardware. Kernel buffers are pinned to physical memory to avoid swapping them out, while a stream DMA transfer is in progress. Buffers can be submitted for a stream DMA transfer to/from the specified device. Data transfers can be monitored to determine if they are in progress, they have finished, or a transfer error has occurred. This interface is used by the Nanos++ runtime system, to drive the work of the IP accelerators in the FPGA.

Therefore, the generated IP should wait for a signal to start, receiving input stream DMA data and finally can start the output stream DMA. Those signals are activated by the DMA library calls done by the runtime. The runtime must wait for the last activated signal in the output stream DMA data, coming from the IP accelerator. This implies that our C task should be embedded with a wrapper code that deals with the input and output management.

Figure 3 shows an example of matrix multiplication that has been annotated with OmpSs directives. Note that this code is independent of the execution platform (i.e., cluster, nodes with FPGAs, nodes with GPUs.), being the runtime responsible for taking care of the task execution scheduling to the devices or nodes of the cluster, transparently to the programmer. This code shows a parallel tiled matrix multiply where each of the tiles (BSxBS sub-matrix) is a task. A, B and C are NBxNB matrices of pointers to BSxBS sub-matrices.

```
const int BS=VALUE;
#pragma omp target device(fpga, smp) copy_deps
#pragma omp task in([BS*BS]a, [BS*BS]b) \
                 inout([BS*BS]c)
void matrix_multiply(int BS,
                     float *a, float *b, float *c) {
    for (int ia = 0; ia < BS; ++ia)
        for (int ib = 0; ib < BS; ++ib) {
            float sum = 0;
            for (int id = 0; id < BS; ++id)
                sum += a[ia*BS+id] * b[id*Bs+ib];
            c[ia*BS+ib] = sum;
        }
}

...
int main( int argc, char * argv[] ){
 ...
   for (i=0; i < NB; i++) {
     for (j=0; j < NB; j++) {
       for (k=0; k < NB; k++) {
         matrix_multiply(BS,A[i][k],B[k][j],C[i][j]);
       }
     }
   }
#pragma omp taskwait
 ...
```

Figure 3. *OmpSs annotations on Matrix Multiply*

Each of those tasks has two input dependencies and an output dependence that will be managed at runtime by Nanos++. Those tasks will be able to be scheduled/fired to a SMP or FPGA, as it is annotated in the target device directive, depending on the resource availability.

The copy_deps clause associated to the target directive hints the Nanos++ runtime to copy the data related to the input and output dependencies to/from the device when necessary. Indeed, the copy_deps (or copy_in/out/inout clauses) will allow the FPGA mercurium phase (Figure 2) to determine which wrapper code should be generated for the task to be accelerated.

The data transfer from the PS (Processing System – SMP ARM) to the PL (Programmable logic) using the DMA library is gathered into only one input stream to the kernel, meanwhile only one output stream is expected, as a result of gathering the output information in the IP. Therefore, the wrapper code will have three different stages:

1. Read the unique stream data input and scatter the information among the different inputs of the task to be computed.

2. Compute the FPGA task with the input data received

3. Gather the output data resulting of the execution of the task, and gather them into a unique stream data output.

Figure 4 shows a global scheme of the wrapper that should be generated by the compiler. For the first stage, note that the matrix multiply code has three inputs: *a* and *b,* where input dependences, and *c* which is input and output dependences. Therefore, the input stream should be scatter among those three variables, locally declared in the code (place in the BRAM once synthesized). Figure 4 shows the necessary code for reading the stream input data to fill *the a* variable. The copy is done through the read().data and copying into an union of an integer and a float. A pragma HLS PIPELINE II=1 is used in the innermost loop of the copy to maximize the throughput of the copies. We expect to achieve one innermost loop iteration completed per cycle. Copies for b and c variables are not shown in the Figure 4. Then, the kernel call and the gather code for the output stream data will follow.

```
void matrix_multiply_wrapper (hls::stream<axiData> &inStream,  hls::stream<axiData> &outStream)
{
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface axis port=inStream
#pragma HLS interface axis port=outStream
  union { float f; int i;} ftoi;
   // stream inStream first matrix
   float a[64*64];
   for (int i = 0; i < 64; i++) {
      for (int j = 0; j < 64; j++) {
#pragma HLS pipeline II=1
         ftoi.i = inStream.read().data;
         a[i*64+j] = ftoi.f;
      }
   }
   // Same for inputs b and c
   float b[64*64];
   …
   float c[64*64];
   …
   // Kernel Call
   // Gather Code
}
```

Figure 4. *Wrapper: IP scheme and scatter of the stream input data for "a" variable*

In the Figure 5 we can see three more pragma HLS at the beginning of the function. Those indicates the type of ports that should be used in the hardware generation, and the necessary control bus. Input and output streams use AXI Stream (AXIS) ports. During the second stage of the wrapper, this calls the original function as we seen in Figure 5.

```
void matrix_multiply_wrapper (hls::stream<axiData> &inStream,
hls::stream<axiData> &outStream)
{
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface axis port=inStream
#pragma HLS interface axis port=outStream

   // Scatter code
   …

   matrix_multiply(BS, a, b, c);

   …

   // Gather code
}
```

Figure 5. *Wrapper: Kernel Call*

This original function code is not currently transformed neither optimized. Therefore, it is responsibility of the programmer to optimize or annotate with pragma HLS to reduce the latency or increase the throughput of the kernel.

Finally, the last and third stage of the wrapper (Figure 6) consist on writing out the output stream data. In that case, we only have one output data coming from the kernel matrix multiply, the *c* variable.

```
void marix_multiply_wrapper (hls::stream<axiData> &inStream,
hls::stream<axiData> &outStream)
{
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface axis port=inStream
#pragma HLS interface axis port=outStream
   // Scatter code
   …
   // Kernel call
   ...
   axiData output = {0, 0, 0, 0, 0, 0, 0, 0};
   output.keep = 0xFF;
   // stream out result matrix
   for (int i = 0; i < 64; i++) {
      for (int j = 0; j < 64; j++) {
#pragma HLS pipeline II=1
         ftoi.f = c[i*64+j];
         output.data = ftoi.i;
         output.last = (i==Dim-1 && j==Dim-1);
         outStream.write(output);
      }
   }
}
```

Figure 6. *Wrapper: Output Stream data and last flag*

The *c* variable data is written in the outStream variable (output stream) using the "write(output)" method. Note also, that the output is a structure (axiData) that has a standard field "last" to flag the

last element of the output stream. This is activated when the last element of the sub-matrix *c* is written.

In addition to the wrapper code for each task with target device *fpga*, a configuration file is being completed to specify the number of instances has to be generated of this task accelerator (by default one at this moment), frequency, and target device. Figure 7 shows detailed description of those steps.

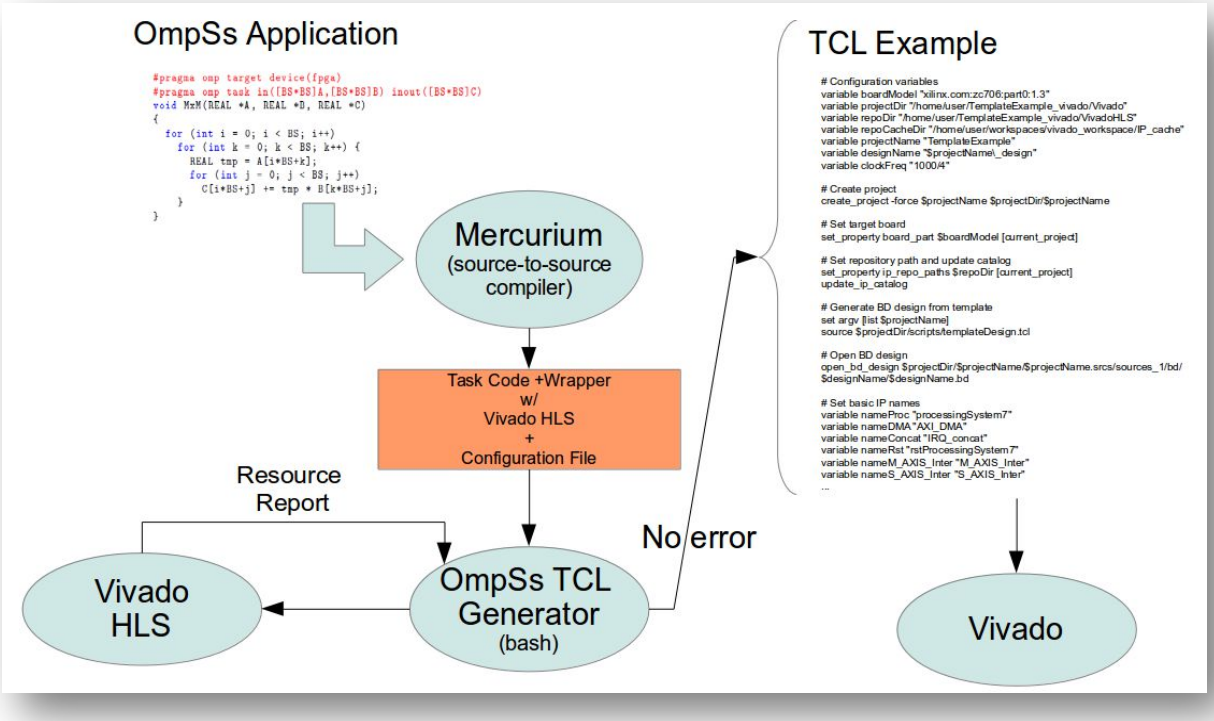


Figure 7. *Detailed Compilation Flow*

This wrapper code, the original kernel C code of all the tasks with target device of an application, the configuration file including all the information related to each of the tasks, will be passed through the Vivado HLS as shown in Figure 7 to synthesize them. The process will be successful if the overall resources of the synthesis of all the tasks likely fits in the target device indicated in the configuration file.

Finally, if no error happens, the compilation framework will create a Vivado project using TCL scripts. The IP's are properly connected using DMA engines and Interconnection IPs to the ProcessingSystem7, at the same time that the interruptions are enabled.

Thus, a bitstream will be generated for the target device with all the tasks. The FPGA should be configured with this bitstream, previously to the execution of the OmpSs application.

The compilation command line to compile a matmul_example.c, including the OmpSs annotations on tasks, and the target device fpga, and to generate the bitstream is the following:

fpgacc -k --verbose -o matmul_example --variable=bitstream_generation:ON matmul_example.c

First, the wrapper code is generated during the FPGA bitstream generation phase analysis. Figure 8 shows how it detects that the bitstream generation is active. In addition, we can see that there is a mechanism to indicate the board name, the device name and other relevant information.

```
FPGA bitstream generation phase analysis
============================================================
        Board Name                  : xilinx.com:zc702:part0:1.1
        Device Name                 : xc7z020clg484-1
        Frequency                   : 10ns
        Bitstream generation active : ON
        Vivado design path          : $PWD
        Vivado project name         : Filename
        IP cache path               : $PWD/IP_cache
        Dataflow active             : OFF
============================================================
Compiler phases pipeline executed in 0.05 seconds
Prettyprinted into file 'arm-linux-gnueabihf-fpgacc_matmul_example.c' in 0.01 seconds
Performing native compilation of 'arm-linux-gnueabihf-fpgacc_matmul_example.c' into 'matmul_example.o'
arm-linux-gnueabihf-gcc -O0 -c -o matmul_example.o arm-linux-gnueabihf-fpgacc_matmul_example.c
```

Figure 8. *Automatic compilation: FPGA Mercurium phase is activated*

Then, the process continues with the synthesis, hardware design generation and implementation, and bitstream generation. This process may last for several minutes, depending on the target device and the IPs to be generated. It will start generating the configuration file, processing each of the kernels and going through the Vivado HLS. The compilation process shows how they are being synthesized, as we seen in Figure 9.

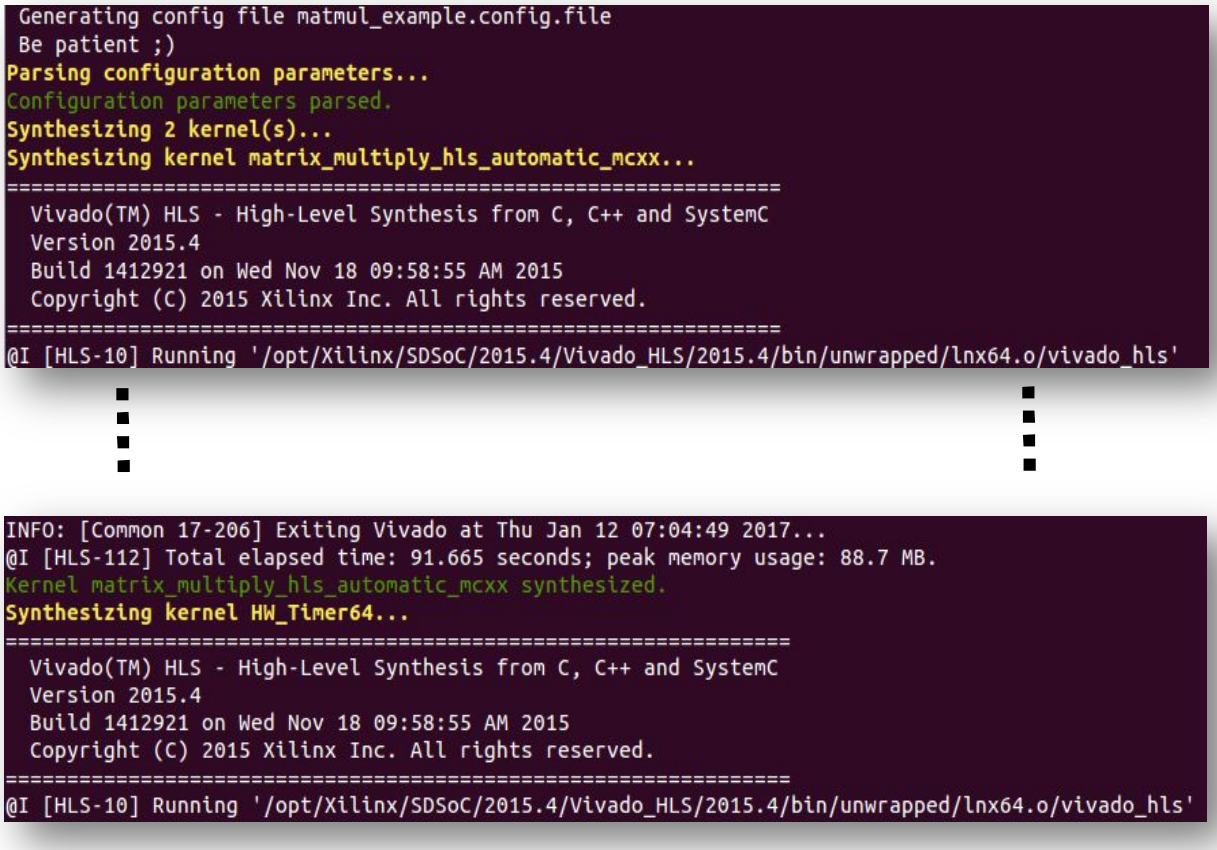


Figure 9. *Automatic compilation: Vivado HLS synthesis of the matrix_multiply kernel: at the beginning (top) and at the end (bottom)*

Once all the kernels are synthesized, the process will report the overall amount of resources necessary for those IP, as we can see in Figure 10. If the reports indicate a number larger than 100%, the process stops. Otherwise, the process continues with the generation of the Vivado project with the IP inside (Figure 10).

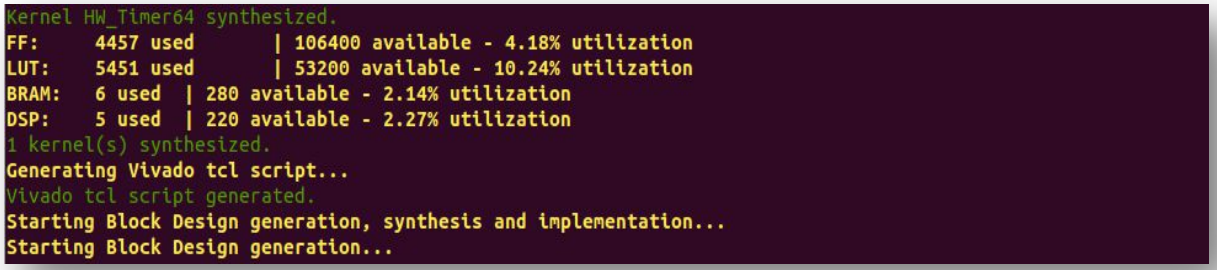


Figure 10. *Automatic compilation: Vivado HLS Report*

The Vivado project generation starts with the block design generation, then, hardware synthesis, hardware implementation, and finally, bitstream generation. Figure 11 shows timestamps of the process for an OmpSs application.

```
INFO: [Common 17-206] Exiting Vivado at Thu Jan 12 07:06:07 2017...
Block Design generated.
Starting hardware synthesis...
...
INFO: [Common 17-206] Exiting Vivado at Thu Jan 12 07:12:33 2017...
Hardware synthesized.
Starting hardware implementation...
...
INFO: [Common 17-206] Exiting Vivado at Thu Jan 12 07:15:49 2017...
Hardware implemented.
Starting bitstream generation...
...
INFO: [Common 17-206] Exiting Vivado at Thu Jan 12 07:16:41 2017...
Bitstream generated.
Hardware automatic generation finished.
```

Figure 11. *Automatic compilation: timestamps of the hardware compilation process*

This process can last for several minutes and an IP cache is automatically generated, so that part of the synthesis can be skipped. The compilation framework also allows avoiding generation of the bitstreams, if it has been already generated for that OmpSs application.

For the PS part, the source-to-source Mercurium compiler will generate an intermediate code, with the Nanos++ runtime calls and the creation of data structures necessaries for the proper creation and submission of the tasks of OmpSs programming model. Mainly, the necessary data structures are the input/output/inout dependence and copy input/output/inout data to be transferred.

The main two Nanos++ calls are:

1. *nanos_err_t nanos_create_wd_compact(...)*
2. *nanos_err_t nanos_submit(...)*

The first one creates the work descriptor of the task with all the necessary information. The second one introduces the task in the task dependence graph of the runtime. With this task dependence graph the runtime knows when a task is ready for the execution (is ready to be fired). Ready tasks are executed by threads. In the case of task going to a device, it may be necessary to copy data to the device and, after executing it, copy back the data. Those copies are specified by the copy clauses.

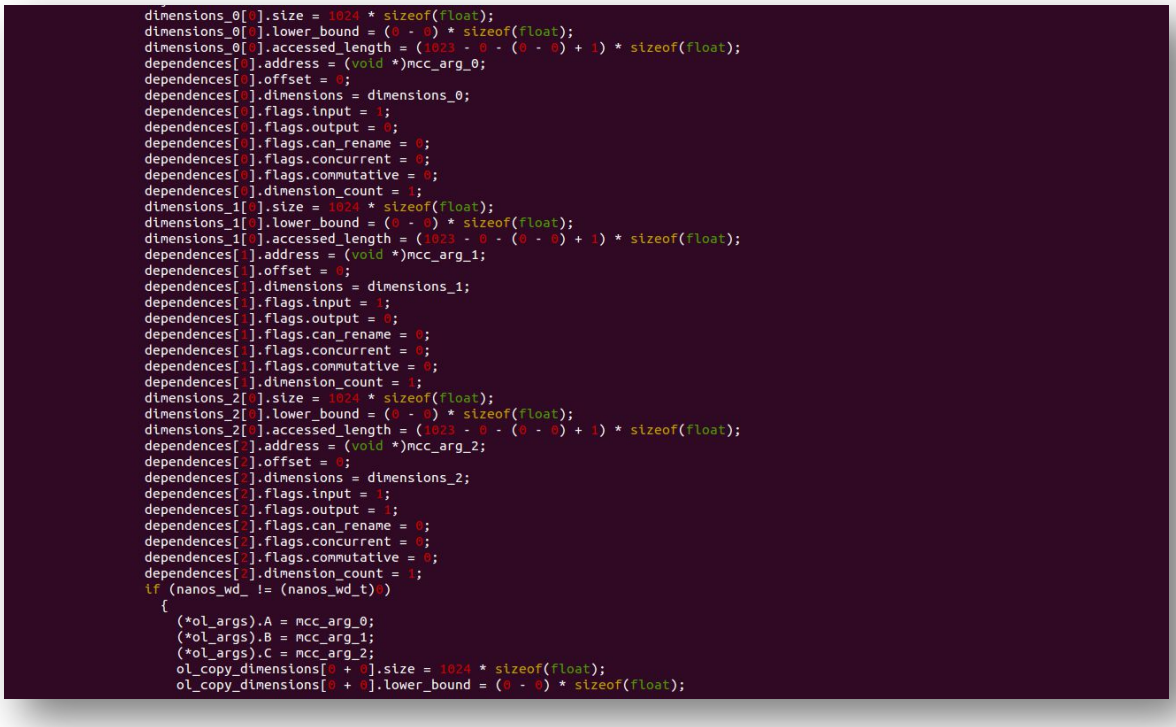
Figure 12, Figure 13, and Figure 14 show different parts of the intermediate code generated by the Mercurium compiler. This intermediate code can be obtained using the compiler option "-k" in the compilation, as shown previously.

Figure 12 shows the section code where the work descriptor of the task is created. This code is automatically generated by Mercurium for each task instance in the code.

```
void matmul(int m, int l, int n, int mDIM, int lDIM, int nDIM, float **tileA, float **tileB, float **tileC)
{
  int i;
  int j;
  int k;
  for (i = 0; i < mDIM; i++)
    {
      for (j = 0; j < nDIM; j++)
        {
          for (k = 0; k < lDIM; k++)
            {
              {
                float *mcc_arg_0 = tileA[i * lDIM + k];
                float *mcc_arg_1 = tileB[k * nDIM + j];
                float *mcc_arg_2 = tileC[i * nDIM + j];
                {
                  static nanos_fpga_args_t fpga_ol_matrix_multiply_1_args;
                  nanos_wd_dyn_props_t nanos_wd_dyn_props;
                  struct nanos_args_0_t *ol_args;
                  nanos_err_t nanos_err;
                  struct nanos_args_0_t imm_args;
                  nanos_region_dimension_t dimensions_0[1];
                  nanos_data_access_t dependences[3];
                  nanos_region_dimension_t dimensions_1[1];
                  nanos_region_dimension_t dimensions_2[1];
                   /* device argument type */
                  fpga_ol_matrix_multiply_1_args.outline = (void (*)(void *))(void (*)(struct nanos_args_0_t *))&fpga_ol_matrix_multiply_1;
                  fpga_ol_matrix_multiply_1_args.acc_num =  -1;
                  static struct nanos_const_wd_definition_1 nanos_wd_const_data = {.base = {.props = {.mandatory_creation = 0, .tied = 0, .clear_chunk = 0, .reserved0 = 0
, .reserved1 = 0, .reserved2 = 0, .reserved3 = 0, .reserved4 = 0}, .data_alignment = __alignof__(struct nanos_args_0_t), .num_copies = 3, .num_devices = 1, .num_dimension
s = 3, .description = "matrix_multiply"}, .devices = {[0] = {.factory = &nanos_fpga_factory, .arg = &fpga_ol_matrix_multiply_1_args}}};
                  nanos_wd_dyn_props.tie_to = 0;
                  nanos_wd_dyn_props.priority = 0;
                  nanos_wd_dyn_props.flags.is_final = 0;
                  nanos_wd_dyn_props.flags.is_implicit = 0;
                  ol_args = (struct nanos_args_0_t *)0;
                  nanos_wd_t nanos_wd_ = (nanos_wd_t)0;
                  nanos_copy_data_t *ol_copy_data = (nanos_copy_data_t *)0;
                  nanos_region_dimension_internal_t *ol_copy_dimensions = (nanos_region_dimension_internal_t *)0;
                  nanos_err = nanos_create_wd_compact(&nanos_wd_, &nanos_wd_const_data.base, &nanos_wd_dyn_props, sizeof(struct nanos_args_0_t), (void **)&ol_args, nanos_
current_wd(), &ol_copy_data, &ol_copy_dimensions);
```

Figure 12. *Intermediate Code: Work Descriptor Creation*

Figure 13 shows the data dependence structure creation. This structure is an array of structures, where each element of the array contains the information of the size of the data, the lower boundary index, and the direction (input/output/inout) information.



```
                  dimensions_0[0].size = 1024 * sizeof(float);
                  dimensions_0[0].lower_bound = (0 - 0) * sizeof(float);
                  dimensions_0[0].accessed_length = (1023 - 0 - (0 - 0) + 1) * sizeof(float);
                  dependences[0].address = (void *)mcc_arg_0;
                  dependences[0].offset = 0;
                  dependences[0].dimensions = dimensions_0;
                  dependences[0].flags.input = 1;
                  dependences[0].flags.output = 0;
                  dependences[0].flags.can_rename = 0;
                  dependences[0].flags.concurrent = 0;
                  dependences[0].flags.commutative = 0;
                  dependences[0].dimension_count = 1;
                  dimensions_1[0].size = 1024 * sizeof(float);
                  dimensions_1[0].lower_bound = (0 - 0) * sizeof(float);
                  dimensions_1[0].accessed_length = (1023 - 0 - (0 - 0) + 1) * sizeof(float);
                  dependences[1].address = (void *)mcc_arg_1;
                  dependences[1].offset = 0;
                  dependences[1].dimensions = dimensions_1;
                  dependences[1].flags.input = 1;
                  dependences[1].flags.output = 0;
                  dependences[1].flags.can_rename = 0;
                  dependences[1].flags.concurrent = 0;
                  dependences[1].flags.commutative = 0;
                  dependences[1].dimension_count = 1;
                  dimensions_2[0].size = 1024 * sizeof(float);
                  dimensions_2[0].lower_bound = (0 - 0) * sizeof(float);
                  dimensions_2[0].accessed_length = (1023 - 0 - (0 - 0) + 1) * sizeof(float);
                  dependences[2].address = (void *)mcc_arg_2;
                  dependences[2].offset = 0;
                  dependences[2].dimensions = dimensions_2;
                  dependences[2].flags.input = 1;
                  dependences[2].flags.output = 1;
                  dependences[2].flags.can_rename = 0;
                  dependences[2].flags.concurrent = 0;
                  dependences[2].flags.commutative = 0;
                  dependences[2].dimension_count = 1;
                  if (nanos_wd_ != (nanos_wd_t)0)
                    {
                      (*ol_args).A = mcc_arg_0;
                      (*ol_args).B = mcc_arg_1;
                      (*ol_args).C = mcc_arg_2;
                      ol_copy_dimensions[0 + 0].size = 1024 * sizeof(float);
                      ol_copy_dimensions[0 + 0].lower_bound = (0 - 0) * sizeof(float);
```

Figure 13. *Intermediate Code: Data Dependences Structure Creation and Initialization*

Finally, Figure 14 shows the section of code where the copy data structure is created. This structure, together with the data dependence structure is included into the work description, and used to submit the task with *nanos_submit(...)*.

```
if (nanos_wd_ != (nanos_wd_t)0)
  {
    (*ol_args).A = mcc_arg_0;
    (*ol_args).B = mcc_arg_1;
    (*ol_args).C = mcc_arg_2;
    ol_copy_dimensions[0 + 0].size = 1024 * sizeof(float);
    ol_copy_dimensions[0 + 0].lower_bound = (0 - 0) * sizeof(float);
    ol_copy_dimensions[0 + 0].accessed_length = (1023 - 0 - (0 - 0) + 1) * sizeof(float);
    ol_copy_data[0].sharing = NANOS_SHARED;
    ol_copy_data[0].address = (void *)mcc_arg_0;
    ol_copy_data[0].flags.input = 1;
    ol_copy_data[0].flags.output = 0;
    ol_copy_data[0].dimension_count = (short int)1;
    ol_copy_data[0].dimensions = &ol_copy_dimensions[0];
    ol_copy_data[0].offset = 0;
    ol_copy_dimensions[1 + 0].size = 1024 * sizeof(float);
    ol_copy_dimensions[1 + 0].lower_bound = (0 - 0) * sizeof(float);
    ol_copy_dimensions[1 + 0].accessed_length = (1023 - 0 - (0 - 0) + 1) * sizeof(float);
    ol_copy_data[1].sharing = NANOS_SHARED;
    ol_copy_data[1].address = (void *)mcc_arg_1;
    ol_copy_data[1].flags.input = 1;
    ol_copy_data[1].flags.output = 0;
    ol_copy_data[1].dimension_count = (short int)1;
    ol_copy_data[1].dimensions = &ol_copy_dimensions[1];
    ol_copy_data[1].offset = 0;
    ol_copy_dimensions[2 + 0].size = 1024 * sizeof(float);
    ol_copy_dimensions[2 + 0].lower_bound = (0 - 0) * sizeof(float);
    ol_copy_dimensions[2 + 0].accessed_length = (1023 - 0 - (0 - 0) + 1) * sizeof(float);
    ol_copy_data[2].sharing = NANOS_SHARED;
    ol_copy_data[2].address = (void *)mcc_arg_2;
    ol_copy_data[2].flags.input = 1;
    ol_copy_data[2].flags.output = 1;
    ol_copy_data[2].dimension_count = (short int)1;
    ol_copy_data[2].dimensions = &ol_copy_dimensions[2];
    ol_copy_data[2].offset = 0;
    nanos_err = nanos_set_translate_function(nanos_wd_, (nanos_translate_args_t)nanos_xlate_fun_matmulexamplec_0);
    if (nanos_err != NANOS_OK)
      {
        nanos_handle_error(nanos_err);
      }
    nanos_err = nanos_submit(nanos_wd_, 3, &dependences[0], (nanos_team_t)0);
```

Figure 14. *Intermediate Code: Copy Structure Creation and Initialization*

At runtime, the Nanos++ runtime will deal with the task dependence graph and the execution of the task by the worker threads. In case of having to submit a task to hardware accelerator, all the information regarding the accelerators is already available to the runtime using a configuration file and the device tree of the system.

The instrumentation of the hardware acceleration and the necessary hardware support is explained in Section 3. This novel feature will allow the programmer to analyze the performance of the memory transfers and computation of the task, and to decide new approximations to the section code computing a specified task. Therefore, the programmer may want to use the *implements* clause to have optimized code for a certain part of the code. An example of how to use the *implements* clause follows:

```
#pragma target device(smp)
#pragma task ...
software_task(){...}

#pragma target device(fpga) implements(task)
#pragma task ...
fpga_optimized_task(){...}
```

The automatic compilation will generate the bitstream for this accelerator and will be available for any exectution of the accelerated task of the *software_task*, using the synthesized code of *fpga_optimized_task* function.

2.3 OmpSs@Cluster Support Implementation

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]. Other alternatives to implement this approach were presented in Section 3 of the deliverable D4.1. The underlying communications layer was presented in Section 4 of the deliverable D4.1.

Figure 15 shows an architectural view of the various options under exploration. 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 MPI and UDP, which are the ones on which OmpSs@cluster typically works, and in addition to these, please note the new XSMLL and AXIOM 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.

For the baseline development in the AXIOM project, we are following option 1 of Figure 15. 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, GASNet has various transport layers that can be used, starting from the simple UDP layer, to MPI, Infiniband and others. The approach here is to use the re-implemented GASNet conduit API, using the Linux Kernel Driver in development in Task 5.1. This implementation takes advantage of the RDMA support provided by FORTH for the AXIOM-link interconnect, in a way similar to the Infiniband conduit. For more information about the GASNet conduit implemented on top of the AXIOM NIC API, please refer to Deliverable D5.2.

Therefore, it is not necessary to give any support from the point of view of the code generation with Mercurium, neither the Nanos++ runtime, to allow the remote execution of tasks. The runtime will call the GASNet, and the GASNet conduit will do all the work of dealing with the API of the hardware communication layer.

Under the compiler point of view, the only special thing that the programmer has to do is to specify copy_deps in the clause of the target device. With this, the OmpSs@Cluster runtime will take care of doing the copies to the remote node where a task is going to be executed.

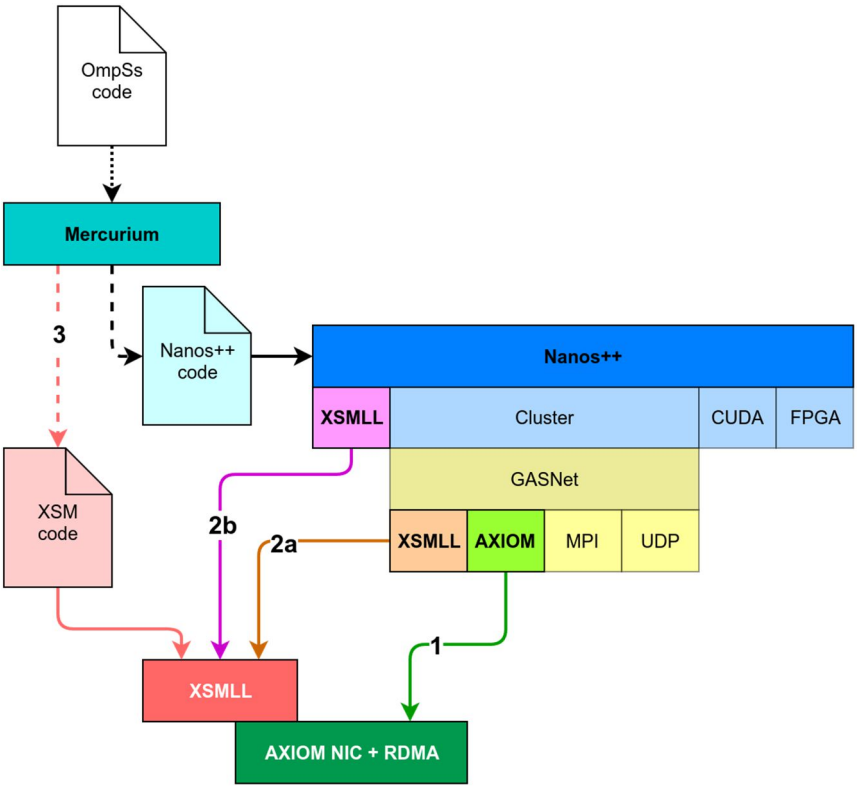


Figure 15. *Architectural view related to the support of distributed environments.*

2.4 Exploration of the Mercurium Integration with XSMLL

The feasibility of the integration of the Mercurium compiler code generation on top of the XSMLL runtime has been evaluated as a cooperation between BSC and UNISI. This option for code generation was initially introduced in Deliverable D4.1, as the Option 3 for the AXIOM support for distributed environments. With it, we are considering the possibility that the XSMLL execution model could be directly integrated into Mercurium, to provide direct XSMLL code generation from OmpSs code.

This task has consisted on examining the existing XSMLL examples running on the COTSon [8,9] simulation environment. As a result, we have first derived an OmpSs version for those benchmarks, and later established the rules to transform the OmpSs tasks towards code calling the XSMLL runtime. Along with these transformations, we have also determined which will be the restrictions set to the OmpSs programs for the transformation to the XSMLL environment to be possible. The resulting model is slightly different from the original OmpSs semantics, in such a way that the task graph is built at compile time, and the dependences among tasks are considered to have a global scope.

2.4.1 OmpSs Simple Application: Matrix Multiply

We have coded a matrix multiplication example in OmpSs with the same structure as the matrix multiply *(mmx)* example provided by the XSMLL runtime system. In the version of the code running on the XSMLL runtime, at the top level of task creation, we observe the code shown in Figure 16. As it can be observed, during code generation, the compiler will need to have the XSMLL identifier, represented by the type *xtid_t*, of matrix_multiply to be provided to the previous task Fill_matrix, before executing the latter. In the same way, the compiler needs the XSMLL identifier of Report_results to be provided to matrix_multiply. As a result, this means that both source and sink tasks of the dependences need to be created before the compiler can issue the code to establish the link between them.

This is different from the OmpSs model, on which tasks can start execution before the sink of the dependence has been created.

There is another difference between the OmpSs approach and XSMLL. Observe the creation of the task Re-port_results. It is given a number of dependences of N/BLOCKSZ. For the Mercurium compiler to provide this value, it is needed that the compiler sees all the tasks created that are the source of these N/BLOCKSZ number of dependences. With the Mercurium compiler infrastructure this is only possible if all tasks are created on the same compilation unit, being this a function or a subroutine. To do it, otherwise, we will need the compiler to perform interprocedural analysis, to account for those tasks present on different functions, called from the current context, with source dependences on the same data.

```
// create the three main tasks
//  1. Fill_matrices
xtid_t xfill = xschedulez(&fill_matrix, 1, sizeof(fill_matrix_s));
//  2. Matrix_multiply
xtid_t xmmx = xschedulez(&owm_matrix_mul, 1, sizeof(owm_matrix_mul_s));
//  3. Report_results
xtid_t xreport = xschedulez(&report, N / BLOCKSZ, sizeof(report_s));
...

// link Fill_matrices and Matrix_multiply, so that
//  when Fill_matrices finishes, it fires Matrix_multiply
fill_matrix_s *xf_fp = xpoststor(xfill);
xf_fp->xmmx = xmmx; // Fill in xtid_t of frame struct.

// link Matrix_multiply and Report_results, so that
//  when Matrix_multiply finishes, it fires Report_results
owm_matrix_mul_s *xm_fp = xpoststor(xmmx);
xm_fp->xreport = xreport; // Fill in xtid_t of frame struct.
...
```

Figure 16. *Code snipped to create and link tasks together in the XSMLL mmx sample.*

After this analysis, the OmpSs code derived from the *mmx* XSMLL sample is presented in Figure 17. On the OmpSs code, all tasks are found in the same function. There are 3 high level tasks, Fill_matrix is the first one, which initializes the 3 matrices (*A*, *B*, and *C*) and computes a checksum (*scs*), which is later used for the verification of the obtained results. The second task, Matrix_multiply, takes the 3 matrices, and creates the inner tasks (as many as N/BLOCKSZ), that will actually perform the matrix multiplication in parallel (*bmmul*). The third and final top-level task is Report_results, which takes C, and checks that the result is correct.

Within the Mercurium compiler, this kind of source codes is analyzed in two steps. In the first step, the internal compiler data structures is built based on the source code and the task annotations. In the second step, the code is analyzed to determine which tasks depend on data generated by other tasks. For example, from the code shown in Figure 17, the task Matrix_multiply will be determined to depend on the task Fill_matrix, because the three A, B, and C variables are written by Fill_matrix, and at least read by Matrix_multiply (A and B are read, and C is read and written).

Determining the dependences between a task and previous tasks with inner level tasks, as it is the case of Report_results and Matrix_multiply, will be implemented based on the symbols listed in the de-

pend clauses of the tasks involved. In this example, dependences for Report_results will account the N/BLOCKSZ iterations of the loop creating tasks in Matrix_multiply. Although computing the amount of dependences may not be always possible, we think that we can support tasks created within loop nests on which the number of iterations can be computed with an expression.

```
int main ()
{
    int i, j, nb;
#pragma omp task depend(out: A[N][N], B[N][N], C[N][N], scs) label(Fill_matrix) ...
   {
    // C = 0
    // Initialization of C to 0
    ...
    // B
    for (i = 0; i < N; i++) {
        cr = 0;
        for (j = 0; j < N-1; j++) {
            int val1 = rand()&0xFF;
            cr = (cr + val1) &0xFF;
            B[i*N+j] = (DATA)val1;
        }
        B[i*N+N-1] = (DATA)cr;
    }
    // A
    for (j = 0; j < N; j++) {
        // initialize A, similarly to B
    }
    // Calculate the SuperCheckSum
    for (i = 0; i < N; i++) {
        superchecksum = (superchecksum + (uint64_t)(A[(N-1)*N+i]) *
                                         (uint64_t)(B[i*N+N-1]))&0xFF;
    }
    *scs = (superchecksum<<2)&0xFF;
   }

#pragma omp task depend(in:    A[N][N], B[N][N]) \
                 depend(inout: C[N][N]) private(i,j) label(Matrix_multiply)
   {
    for (j=nb=0; j<N; j+=BLOCKSZ,++nb) {
       #pragma omp task depend (in:    A[j:j+BLOCKSZ], B[N][N]) \
                        depend (inout: C[j:j+BLOCKSZ]) label(bmmul)
       {
         for (j = 0; j < BLOCKSZ; j++) {
            for (i = 0; i < N; i++) {
               DATA t = 0;
               for (k = 0; k < N; k++) {
                  t += A[j * N + k] * B[k * N + i];
               }
               C[i + (j * N)] = t;
            }
         }
       } // end of task
    }
   }
#pragma omp task depend(in: C[N][N], scs) private(i,j) label(Report_results)
   {
    int ok = 1;
    print_matrix (C, N, N);
      // Verify the SuperCheckSum
    uint64_t xchecksum = 0L;
    for (i = 0; i < N; i++) {   // i = row pointer
        for (j = 0; j < N; j++) { // j = column pointer
            xchecksum = (xchecksum + (int)(C[i*N + j])) &0xFF;
        }
    }
    if (scs==xchecksum) ok = 1; else ok = 0;
    printf("CHECKSUM=%lu vs %lu   OK=%d\n", xchecksum, scs, ok);
    printf("\n*** %s ***\n", ok ? "SUCCESS" : "FAILURE");
   }

#pragma omp taskwait
}
```

Figure 17. *OmpSs version of the mmx XSMLL sample.*

While data participating in task depend clauses will be included in the task frame context, variables used internally will be considered as *firstprivate*. This is the same default behavior that OmpSs (and OpenMP) define for variables not listed on task data sharing clauses. Variables can also be established to be private to the task, by using the private clause in the OmpSs code. Such private variables will be declared inside the function defined for the task.

2.4.2 Code transformations

After the analysis of the sample code, the following code transformations are planned on the Mercurium compiler. On the first place, the code of the main function needs to be transformed to implement task creations through calls to the XSMLL runtime. Secondly, each task will be outlined in its own function, in order to give the possibility to the runtime system to execute it on another thread.

The code generated in the function spawning the tasks has to deal with task creation and data dependences from tasks. As a first example, in Figure 18 we show the creation of the Fill_matrix task, and how its data references are set. Data is laid out internally by the XSMLL runtime system following the offsets and sizes (second and third parameters in *xsubscribe*). The resulting pointer to the starting location of the data is then assigned to the task frame, in the field indicated by the XDST macro. The last parameter, the read/write mode is used by the XSMLL runtime to implement the data coherency. The runtime systems tracks the use of this data, in order to forward the data to the destination tasks.

```
xtid_t xfill = xschedulez(&fill_matrix, 1, sizeof(fill_matrix_s));
xtid_t xmmx = xschedulez(&owm_matrix_mul, 1, sizeof(owm_matrix_mul_s));
...
// subscribe Fill_matrix to (in and) out data
xsubscribe(XDST(xfill, fill_matrix, A), 2*SIZE(N, N), SIZE(N, N), _OWM_MODE_W);
xsubscribe(XDST(xfill, fill_matrix, B),   SIZE(N, N), SIZE(N, N), _OWM_MODE_W);
xsubscribe(XDST(xfill, fill_matrix, C),             0, SIZE(N, N), _OWM_MODE_W);
xsubscribe(XDST(xfill, fill_matrix, scs),
                                  3*SIZE(N, N), sizeof(uint64_t), _OWM_MODE_W);

// link Fill_matrices and Matrix_multiply, so that
//  when Fill_matrices finishes, it fires Matrix_multiply
fill_matrix_s *xf_fp = xpoststor(xfill);
xf_fp->xmmx = xmmx; // Fill in xtid_t of frame struct.
...
```

Figure 18. *Code creating the Fill_matrix task from the main function*

As a second example of task creation, we show the code that creates the inner tasks of Matrix_multiply. In this case, such inner tasks are created inside the *for* loop present in the source code, to implement the task spawning. Figure 19 shows the transformed code. For each iteration of the loop, one additional task is created, and linked to the final Report_results task. This way, we use a global dependence name space, on which tasks at different levels can interact to each other. The link to the Report_results task can only be done if such task is created earlier. This is implemented correctly in the current example, as it will be created by the main function in advance to executing the Matrix_multiply task. Matrix_multiply is also linked to Report_results, in such a way that the Matrix_multiply function can access its frame context to get the reference to Report_results and assign it to the *bmmul* tasks.

Each *bmmul* task is also subscribed to its input and output data, and, in this way, the XSMLL runtime ensures that before their execution, the data will be accessible.

```
void Matrix_multiply()
{
 const Matric_multiply_s * fp = xpreload();
 xtid_t xreport = fp->xreport;
 for (j=nb=0; j<N; j+=BLOCKSZ,++nb) {

   xtid_t bm = xschedulez(&bmmul, 1, sizeof(bmmul_s));
   /* Region A (size N*BLOCKSZ) stores the block of matrix A  */
   /* Region B (size N*N)       stores the entire matrix B */
   /* Region C (size N*BLOCKSZ) stores the computed result.  */
   xsubscribe(XDST(bm, bmmul, A), (N*N+N*j)*sizeof(DATA),
                                    N*BLOCKSZ*sizeof(DATA), _OWM_MODE_R);
   xsubscribe(XDST(bm, bmmul, B), (N*N)*sizeof(DATA),
                                      (N*N)*sizeof(DATA), _OWM_MODE_R);
   xsubscribe(XDST(bm, bmmul, C), (N*j)*sizeof(DATA),
                                    N*BLOCKSZ*sizeof(DATA), _OWM_MODE_W);

   // finish barrier
   bmmul_s* bm_fp = xpoststor(bm);
   bm_fp->xreport = xreport;
   xdecrease(bm,1);
 }
}
```

Figure 19. *Code creating the bmmul tasks from Matrix_multiply*

On the second transformation needed for the Mercurium compiler, each task will be outlined on its own function, including the declaration of the data type that will represent its frame context. Figure 20 shows this code, describing how the parameters are accessed. In particular, each task gets access to its frame context using the *xpreload* XSMLL service, and uses the data structure defined to use its input and output data. At the end of the task, the XSMLL *xdestroy* service is used to destroy the underlying thread.

```
// structure representing the frame context of Report_result
typedef struct { DATA * C; uint64_t *scsp; } Report_results_s;
void Report_results()
{
   const Report_results_s *crr = xpreload();
   const DATA *C = crr->C;
   const uint64_t *scsp = crr->scsp;
   uint64_t scs = *scsp;
   int i, j;  // i, j variables are listed as private in the OmpSs code
   int ok = 1;
   print_matrix(C, N, N);
    /// code is the same as in the OmpSs task
   ...
   xdestroy();   // all tasks end with the call to the XSMLL xdestroy service
}
```

Figure 20. *Code of the function outlined for the Report_results task*

2.4.3 OmpSs to XSMLL Source Code Transformation Restrictions

Given the example shown on the previous section, and the possibilities shown to transform it to XSMLL code, the following are the restrictions we have determined on the input OmpSs source code to be able to be transformed:

- All annotated code must be visible within the same function or subroutine. Either automatic or hand-coded inlining can be used to achieve this.
- All code must be enclosed in tasks. There can be no code in between tasks, because its execution would not be done in the proper order.
- Each task accesses data provided on in/out depend clauses, having effects outside of the task, or it accesses internal local variables. The task must not have other side effects on data in addition to the dependent data.
- Functions called from the transformed compilation unit should not access the data used in tasks depend clauses, as they will not be properly transformed. Instead, for that code to work properly, it should be inlined within the compilation unit with the annotations.
- The code under transformation must have a single *taskwait* directive at the end. *Taskwait* is not supported in any other context, as there is no way to stop a running thread in XSMLL, to wait for the previously created tasks.

3 Instrumentation Support

In this section, we present the instrumentation support for both OmpSs@FPGA to be able to instrument FPGA acceleration, and for distributed environments. For the latter, we present the OmpSs@Cluster instrumentation support over Ethernet and the GASNet conduit implemented by Evidence to support the High Throughput Interconnection link of FORTH.

3.1 FPGA Instrumentation Support

Having different types of hardware accelerators available, each with their own specific low-level APIs to program them, there is not yet a clear consensus on a standard way to retrieve information about the accelerator's performance. To improve this scenario, OMPT is a novel performance-monitoring interface that is being considered for integration into the OpenMP standard. OMPT allows analysis tools to monitor the execution of parallel OpenMP applications, by providing detailed information about the activity of the runtime through a standard API. For accelerated devices, OMPT also facilitates the exchange of performance information between the runtime and the analysis tool. We have implemented part of the OMPT specification, that refers to the use of accelerators, both in the Nanos++ parallel runtime system and the Extrae tracing framework, obtaining detailed performance information about the execution of the tasks issued to the accelerated devices to later conduct insightful analysis.

In this section, we discuss the extensions developed for the Nanos++ runtime library and the Extrae instrumentation package to comply with the OMPT standard performance-monitoring interface [11].

To support the OMPT interface for tools, the runtime has to maintain information about the state of the execution, and provide a set of callbacks to notify a tool of various runtime events during the run, such as thread begin/end, parallel region begin/end, and task region begin/end, among others. Meanwhile the tool must implement these callbacks to retrieve the information emitted by the runtime, and process and store it as required.

Regarding accelerators, OMPT proposes two mechanisms to pass information to the tool. On the one hand, the Native Record Types interface (see Section 6.2 of the OMPT API[11] enables to invoke native control functions directly on the accelerator, binding the implementation to the architecture. On the other hand, the OMPT Record Types are a set of standard events that express the activity of the accelerator. These events are a generic abstraction of the activity of the device that unifies different types of hardware accelerators. We have opted to rely on the use of OMPT Record Types, so the tool is always agnostic of the underlying devices, with the consequent advantages of reducing software dependencies.

The sections below describe the modifications applied both to the Nanos++ runtime as well as to the Extrae instrumentation library. The extensions in Nanos++ include the addition of new query services to instrument the FPGA device, a reshaping phase of this information into Nanos++ internal events and the extension of the OMPT plugin. First, to capture and handle these new device events and then completing the callback interface established with the tool. On the tool side, the extensions in Extrae include support for the tracing buffer management during the program's execution, as well as considerations about the data representation for analysis.

3.1.1 Compiler Support

To generate a bitstream in a platform with hardware accelerators with instrumentation support, a compiler option should be provided. In this case, the wrapper will have additional ports to access to the hardware support for profiling. This port provides a mechanism to read the timing information to measure the DMA memory transfers and the computation time inside the FPGA.

Figure 21 shows a skeleton of an IP with profiling support.

```
void matrix_multiply_wrapper (hls::stream<axiData> &inStream,  hls::stream<axiData> &outStream. counter_t __data[4])
{
#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=__data
 ...
  counter_t  __counter_reg[4]={0x0,0x0,0x0,0x0};

  __counter_reg[0] = read_hw_timer(__data);
   // Read data
  __counter_reg[1] = read_hw_timer(__data);
    // Kernel Call
  __counter_reg[2] = read_hw_timer(__data);
    // Write data
  __counter_reg[3] = read_hw_timer(__data);

   write_counters_memory(__data, __counter_reg);
}
```

Figure 21. *Wrapper: IP scheme with profiling information*

The IP will read the timing information through the *m_axi* port (read_hw_timer functions) before and after each DMA transfer, and before and after the kernel execution. Finally, this information is copy back to the PS Memory using again the *m_axi* port. The address of the timing information and where to copy back the hardware counters are provided at execution time by the runtime, when invoking the task acceleration. All the mechanism to keep this instrumentation is explained in next Section.

3.1.2 Nanos++ Runtime Support

Nanos++ is a library designed to serve as runtime support for parallel environments. It is mainly used to support OpenMP-like programming models by providing services to exploit task parallelism synchronized by means of data-dependencies. Tasks are run by user-level threads when their data-dependencies are satisfied. The runtime also provides support for maintaining coherence across different address spaces (such as GPUs, cluster nodes or FPGAs).

One of the principles of design of the Nanos++ Runtime Library is modularity. Figure 22 shows a simplified schema of the Nanos++ modules involved in the management of the accelerated devices and the instrumentation mechanisms, and how these modules interact. For each supported hardware device, Nanos++ provides a specific plug-in that implements all the necessary logic to execute a task in the target architecture.

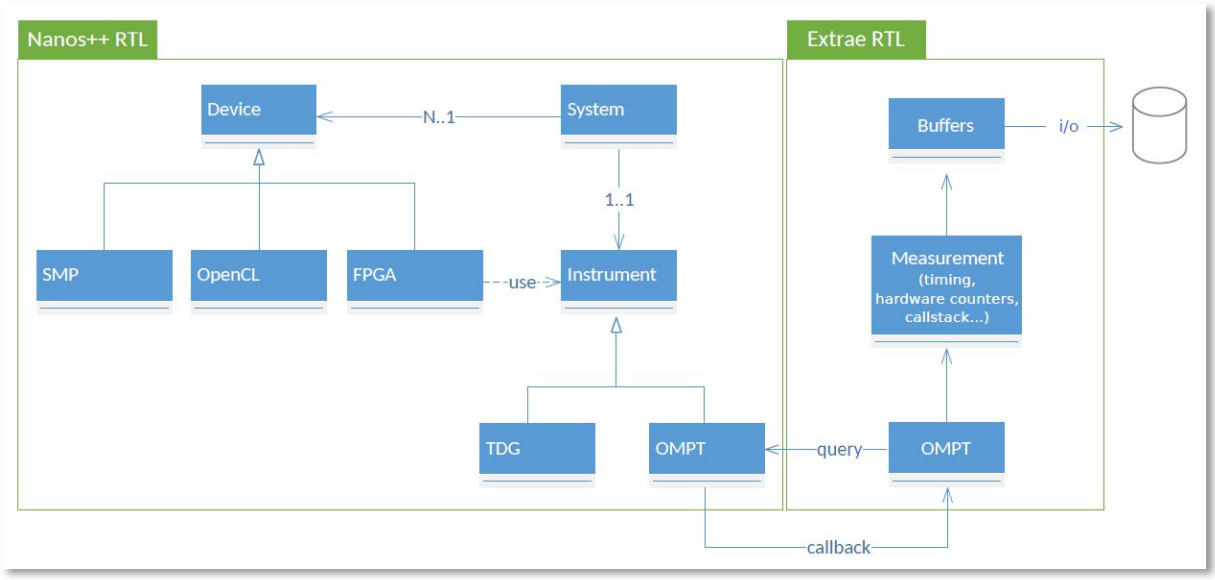


Figure 22. *Nanos++ Runtime Library partial class diagram: device components and instrumentation*

The execution of a task on a device that works with a different memory address space, such as the case of FPGAs, involves several steps. In this case, the device plug-in is responsible for the allocation and copy of input data to the device memory, issue the task for execution, and deallocate and copy results back to the host.

The instrumentation support for the executed tasks is also provided by plug-ins as TDG and OMPT. TDG is an example of monitoring tool that provides a graphical representation of the program's tasks dependencies. In this work, we added the new instrumentation plug-in to support OMPT, which will be used by the FPGA device to provide performance information regarding the execution of the tasks on the accelerator.

The device plug-in notifies about the activity of the hardware using a set of internal events representing the state of the accelerator (i.e. copying data and running a task) to the OMPT plug-in. In turn, the OMPT plug-in stores these events in separate memory buffers for each active device. This requires to develop mechanisms to manage the creation of event buffers, as well as query services (e.g. register a new device, number of devices, device identifier, target identifier) to associate the devices with their corresponding buffers.

The device plug-in is also responsible for retrieving the time-stamps in which the hardware produces the events, and provides this information to the OMPT plug-in. The FPGA hardware timings may not be based on a real-time clock, but on internal clock cycles counters. For this reason, the device plug-in needs to provide a mechanism to query and translate the hardware timings into unified time-stamps, according to the OMPT standard.

All the events stored in the OMPT plug-in from the devices are assembled into OMPT Record Types, a set of standard events designed to exchange data between the runtime and the tool. When the buffers are full, or on demand by the tool, the runtime will provide the event information to the performance tool through several callbacks set during the initialization phase. Then, the tool will parse the information through a set of iterator routines for the OMPT Record Types, which the instrumentation plug-in provides. This process is explained in detail in the next Section.

3.1.3 Extrae Instrumentation Tool Support

The Extrae instrumentation framework has been accordingly extended to implement the OMPT standard in order to monitor the activity of the parallel runtime, enabling to capture performance information about the work offloaded to hardware accelerators. The information collected will help the analyst to understand which application's tasks are executed on which device, as well as their duration.

Integrating the OMPT interface on the tool side involves two main design aspects; the first one concerns the data storage and management. Most accelerators do not have a local memory to allocate instrumentation buffers in which to store the tracing events, neither access to the I/O subsystem to store them. To circumvent this limitation, our solution relies on hosting the tracing buffers for the accelerators on the host processor's main memory. The tool is then responsible for allocating the memory for the events that the runtime will produce and storing the data to disk, while the emission of the events is delegated to the runtime.

The second issue refers to the data representation for the analysis. It is important that the tool presents a clear view of which task ran on which accelerator. There are two main representations, the first assign one timeline for each host H and one timeline for each accelerator A (for a total of H+A time-lines). The second show one timeline of each accelerator for each host (resulting in HxA time-lines). Thereby, each accelerator timeline only contains the activities that originate on the according host. We chose the latter because it allows visualizing more clearly the interactions between host and accelerator. Furthermore, we split each accelerator timeline into its main logical components (kernel computation, input and output memory transfers) to highlight the chain of execution and the tasks' data dependencies.

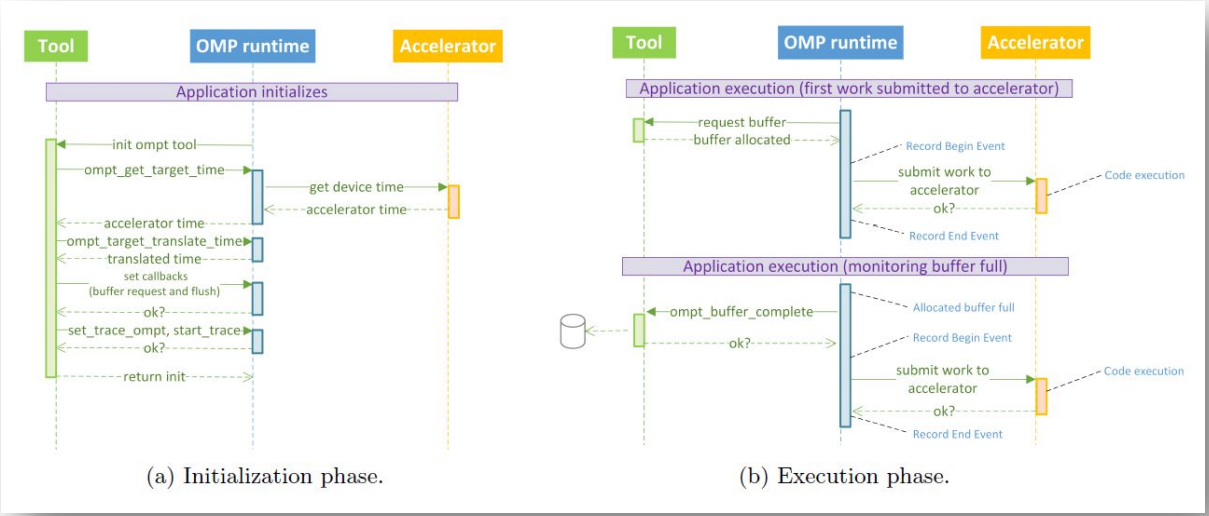


Figure 23. *Simplified call sequence between the monitoring tool, runtime and accelerator device*

From the implementation standpoint, a simplified call sequence interaction between the tracing tool, the runtime and the application through the OMPT API is shown in Figure 23. In the initialization phase (see Figure 23.a), the tool needs to correct the time latency between the clocks on the host and the target accelerator (using ompt_target_get_time and ompt_target_translate_time), and also assign

synthetic thread identifiers. Those thread identifiers are going to represent the different logical components of each accelerator for each host, as explained above.

Then, the tool needs to provide two callbacks to the runtime for the tracing buffer management during the program's execution, one will handle memory allocation requests, and the other will process a buffer of events when it is full. The first callback allocates a buffer for a target accelerator within the host address space, which will be returned to the runtime on demand to have it filled with the monitored tracing events (see Figure 23.b). The second callback receives a full buffer from the runtime that contains the traced events of a target accelerator, and the tool is the one responsible for storing the data into disk. As mentioned earlier, the runtime records the traced events into OMPT Record Types, which the tool can parse with provided iterators (ompt_target_buffer_get_record_ompt, etc.) to serialize the data into the final trace. The events that we are currently monitoring are ompt_event_task_begin, ompt_event_task_switch and ompt_event_task_end, which enables us to keep track of the tasks offloaded to the accelerators and their duration.

More precisely, the ompt_event_task_begin event notifies about the current active task in the accelerator, and provides information about the task identifier and the outlined function. The event ompt_event_task_switch notifies when a task is *scheduled in* and *out* of the device, which can be used to mark in the trace the real execution life span of the task in the accelerator. In order to record the actual time-stamps of these events, it is necessary to synchronize the target device's clock with the host's, applying the previously calculated time corrections. In turn, the ompt_event_task_end event indicates the finalization of the given task, but some runtimes may omit this event and only emit a last ompt_event_task_switch marking the task as *scheduled out*.

Lastly, the tool notifies the runtime to start monitoring the host and the accelerators activity (ompt_target_set_trace_ompt and ompt_target_start_trace). Once the execution finishes, the tool still needs to store the remaining events in the tracing buffers that may have not been flushed yet. To this end, the tool calls to ompt_target_stop_trace for each accelerator device, which implicitly requests the runtime to flush the associated allocated buffer for the given accelerator through the corresponding callback provided by the tool.

3.1.4 Hardware Support

To support the hardware instrumentation, a timer has been implemented. Therefore, both IP accelerators and the OmpSs application, running in the ARM, can easily and transparently access it. This can have been basically done using a BRAM memory that can be accessible using through a BRAM controller. The BRAM controller is mapped to a physical memory accessible from anywhere through the PS system and the IP accelerators.

Figure 24 shows a picture of the Vivado hardware design. This hardware design shows an IP accelerator with the input and output Stream, ant the data port for the profiling support.

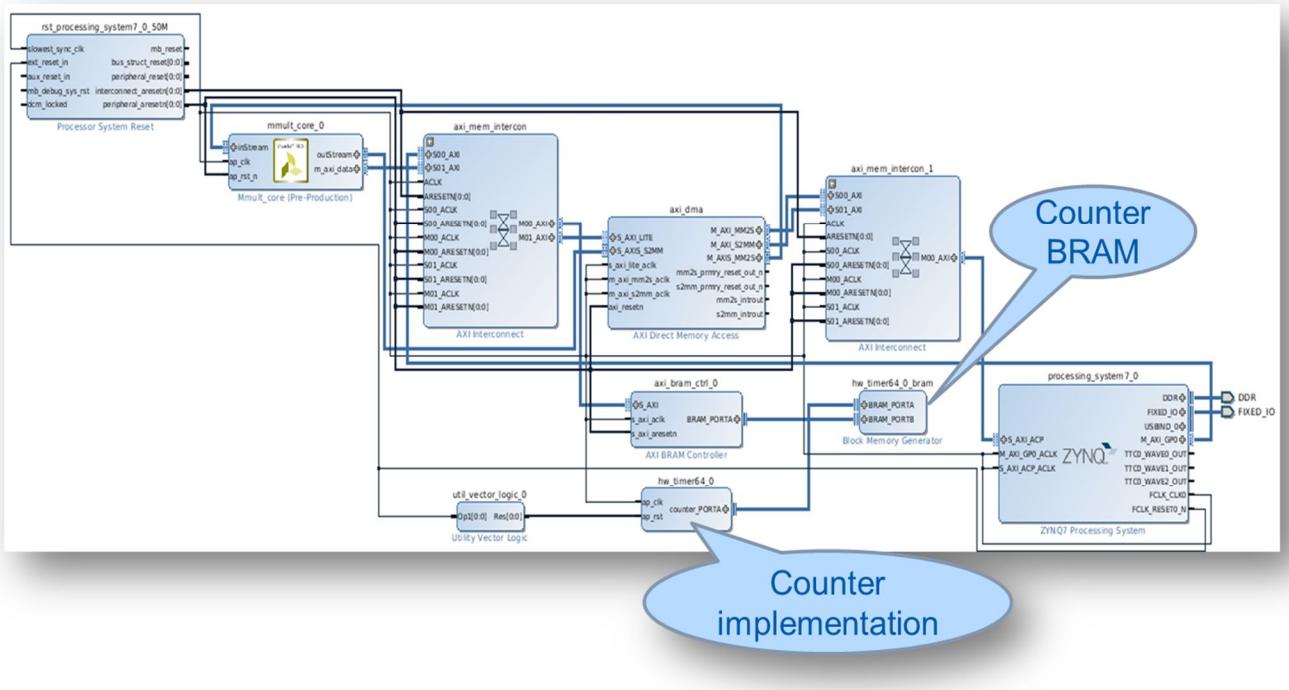


Figure 24. *Hardware design of an IP accelerator with hardware instrumentation support.*

3.2 OmpSs@Cluster Instrumentation Support

Deliverable D5.2 provides a comprehensive description on the design choices that were done in to develop a GASNet conduit, which directly supports the AXIOM user Libraries and the AXIOM kernel driver running on top of the AXIOM NIC registers. In order to provide a comprehensive trace infrastructure, that is able to trace all actions performed in the GASNet conduit, we added support for the Extrae logging facilities directly inside the implementation of the GASNet conduit. In particular, we added the following trace points:

- **IOCTL execution**. Added mainly because the current interaction of the AXIOM user libraries with the kernel driver uses IOCTL calls. In this way, it is possible to trace the main interactions between the application and the message buffer implementation. The instrumentation method used in this case is the Extrae support for wrapping existing functions.

- **All the AXIOM API** (send/receive of raw messages, RDMA write/read, send/receive long messages). The instrumentation method used in this case is the insertion of explicit Extrae trace points inside the AXIOM User library.

- **The main high-level functions of the AXIOM allocator** (see D5.2). The instrumentation method used in this case is the insertion of explicit Extrae trace points inside the AXIOM User library.

To provide a seamless integration with the instrumented libraries of the existing Nanos++ runtime, the compilation of the AXIOM User Library is now implemented in a way to produce two separate libraries. The first one, without instrumentation mechanisms, and the second one, with Extrae instrumentation, which is automatically copied in the $PREFIX/lib/instrumentation directory in order to be automatically used by the system when Extrae is enabled.

The Extrae traces are then produced after the execution on each node of the AXIOM cluster (currently emulated on QEMU machines, see Section 4.2); a script is then responsible for collecting and merging the traces in a single trace, that is later visualized with Paraver tool.

4 Results

This Section shows the evaluation of the automatic generated OmpSs@FPGA, using the instrumentation of the hardware acceleration of those accelerated applications in a Zynq-7000 SoC. In addition, we present some preliminary results of OmpSs@Cluster on a QEMU-emulated AXIOM cluster, using the GASNet conduit, implemented by Evidence over the API of the interconnection network implemented by FORTH. This way, we have been able to advance on the porting of the software infrastructure to run on two (emulated) Zynq Ultrascale+ boards, which is ready to be ported to the actual hardware environment, when available.

4.1 OmpSs@FPGA Instrumentation Analysis

In this section, we present different execution scenarios varying the number and type of accelerators, with the objective of showing the significant insight that trace-based performance analysis of the accelerators activity provides to the user.

Results in the next subsection have been obtained on a Zynq SoC 702 board. This platform integrates a SMP dual core ARM Cortex A9 processor running at 666 MHz plus a programmable logic (FPGA) region based on Xilinx's Artix 7 FPGA [12]. The OmpSs ecosystem for FPGA/SMP heterogeneous execution used to obtain these results is based on the Mercurium compiler 1.99.9, the Nanos++ runtime 0.10a, and Extrae tracing framework 3.3.0. In order to generate the FPGA bitstream that implements the accelerator/s logic for the OmpSs task/s with target device fpga, Vivado and Vivado HLS, Xilinx's proprietary tools, both version 2015.4, have been used. In the case of floating-point applications, Vivado HLS synthesizes code compliant with the IEEE-754 standard. All the applications and libraries have been cross-compiled using arm-linux-gnueabihf-gcc 4.8.4 (Ubuntu/Linaro 4.8.4-2ubuntu1 14.04.1).

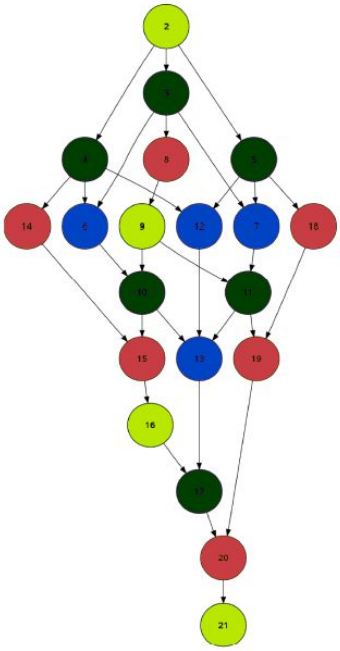
We show trace execution results for two tiled applications: matrix multiply and Cholesky decomposition, using the following fpga task granularities for the tiles (blocks): 64x64 -block and 32x32 single precision floating point matrix multiply, and 64x64 -block double-precision floating point Cholesky decomposition. Matrix multiply (Figure 25) is a well-known and common scientific computation kernel that provides a reasonably simple scenario to illustrate how our framework is able to display the activity of the accelerated system.

```
#pragma omp target device(fpga)
#pragma omp task in([BS*BS]A,[BS*BS]B) inout([BS*BS]C)
void MxM(REAL *A, REAL *B, REAL *C)
{
  for (int i = 0; i < BS; i++)
    for (int k = 0; k < BS; k++) {
      REAL tmp = A[i*BS+k];
      for (int j = 0; j < BS; j++)
        C[i*BS+j] += tmp * B[k*BS+j];
    }
}

void matmul(REAL **AA, REAL **BB, REAL **CC, int NB)
{
  for (int k = 0; k < NB; k++)
    for(int i = 0; i < NB; i++)
      for (int j = 0; j < NB; j++)
        MxM(AA[i*NB+k], BB[k*NB+j], CC[i*NB+j]);
}
```

Figure 25. *Matrix multiplication annotated with OmpSs directives. Matmul is the blocking matrix multiplication function, and MxM performs the matrix multiplication of a block.*

Cholesky decomposition (Figure 26.a) is a more complex scientific computation kernel due to the task data dependencies, as it can be seen in Figure 26.b. In this case, two out of four of the kernels are annotated to be able to run in the FPGA (using the directive target device (fpga). The other two have not been considered to be mapped to the FPGA by the programmer. This approach has been selected with the purpose of showing the potential of the proposed tracing although is not the configuration that achieves the best performance [13].

```
#pragma omp target device(fpga)
#pragma omp task in([BS*BS]A) inout([BS*BS]C)
void dsyrk(double *A, double *C, int BS);

#pragma omp task inout([BS*BS]A)
void dpotrf(double *A, int t, int BS );

#pragma omp task in([BS*BS]A) inout([BS*BS]B)
void dtrsm(double *A, double *B, int t, int BS);

#pragma omp target device(fpga)
#pragma omp task in([BS*BS]A, [BS*BS]B) inout([BS*BS]C)
void dgemm(double *A,double *B,double *C,int t,int BS);

void chol_ll(double **AA, int t, int NB, int BS)
{
  for (int k = 0; k < NB; k++ ) {
    for (int j=0; j<k; j++)
      dsyrk(AA[j*NB+k], AA[k*NB+k], BS);

    dpotrf(AA[k*NB+k], t, BS );

    for (int i = k+1; i < NB; i++)
      for (int j=0; j<k; j++)
        dgemm(AA[j*NB+i],AA[j*NB+k], AA[k*NB+i],t,BS);

    for (int i = k+1; i < NB; i++)
      dtrsm(AA[k*NB+k], AA[k*NB+i], t, BS );
  }
}
```



(a) Code annotated with OmpSs directives. Each function call will be a task instance (**dgemm, dsyrk**: target device (**fpga**); **dtrsm, dpotrf**: no target device is defined, which is (**smp**) by default.

(b) Task dependency graph for number of blocks equal to 4. Light green, red, blue, dark green nodes correspond to tasks **dpotrf, dsyrk, dgemm, dtrsm**, respectively.

Figure 26. *Cholesky decomposition.*

.

4.1.1 Cholesky Application Analysis

In the case of the tiled Cholesky decomposition, we have evaluated one of the possible smp /fpga target device assignments of the application's task kernels.

In order to simplify the example, the problem size is fixed (256x256), the task granularity is fixed (64x64 blocks), and the combination of software/hardware partition analyzed is one hardware accelerator for each dgemm and dsyrk tasks, and dpotrf and dtrsm tasks running in the SMP. Figure 27 shows the trace execution of the Cholesky application using this software/hardware partition. The begin/end of the smp tasks (in Master), the DMA transfers, and the fpga tasks, are marked with green flags to help distinguish consecutive instances of the smp tasks. This trace shows how the proposed OMPT-based tracing combines the accelerator and SMP partial traces into one single trace that helps to gain overall understanding of the whole application. While dpotrf and dtrsm tasks run in the Master thread (first row, light and dark green colors respectively), dgemm and dsyrk tasks (blue and red color respectively) are offloaded to the accelerator devices. We can observe that the task execution order matches the task data dependency graph in Figure 26.b, where the tasks are represented by the same colors.

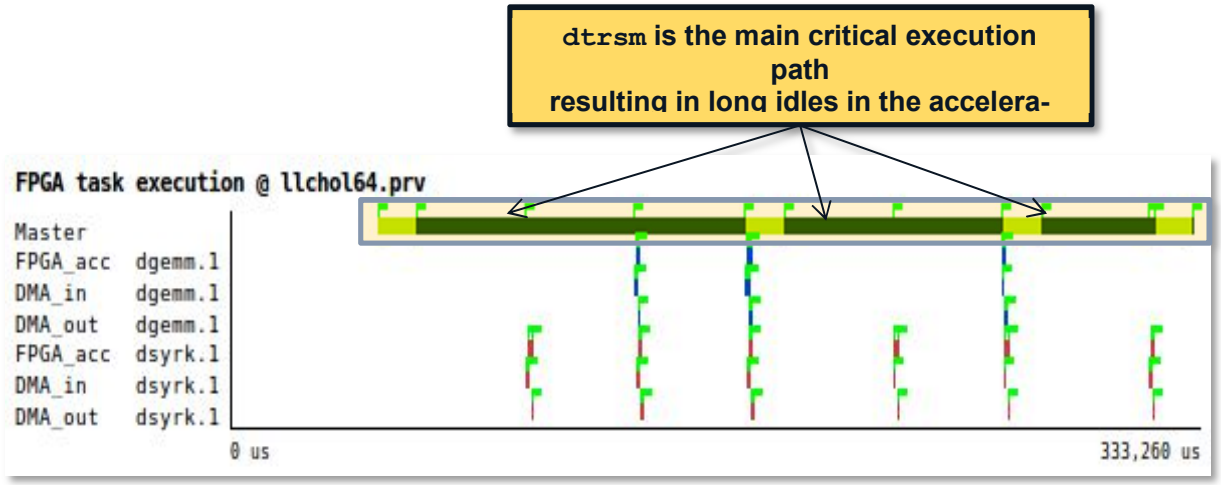
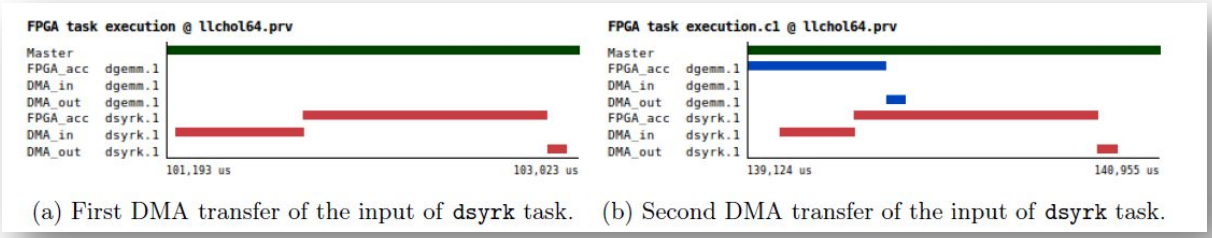


Figure 27. *Cholesky execution using 2 fpga accelerators for dgemm and dsyrk. Dpotrf and dtrsm run in the SMP.*

Figure 27 also provides significant insight on the task acceleration in the FPGA. For instance, the first dsyrk task presents an unexpected input/output data transfer duration that is significantly higher than the rest of dsyrk tasks (see Figure 28). Figure 28.a shows the very first DMA memory transfer (DMA_in) in the application, which costs twice as much as any other DMA_in transfer of the same kernel (see Figure 28.b). This likely results from the DMA driver initialization during the very first fpga task acceleration.



(a) First DMA transfer of the input of `dsyrk` task. (b) Second DMA transfer of the input of `dsyrk` task.

Figure 28. *Comparison of two different runtime configurations for hardware acceleration.*

Furthermore, Figure 27 shows that the dtrsm tasks (dark green phases in Master preceding the accelerated kernels) are in the critical execution path of the current application implementation. The dtrsm tasks have a relatively low performance compared to the FPGA acceleration, and only few dgemm and dsyrk instances can be ready to be executed after a dtrsm task execution. In addition, the dgemm and dsyrk tasks are executed much faster, leaving the hardware accelerators idle for a long time (e.g. the dtrsm task latency). With this information, the programmer may decide to accelerate the dtrsm tasks in hardware. Another possible solution is to execute the slow SMP tasks (dpotrf and dtrsm) in more than one core of the SMP, so that two or more dtrsm tasks can be executed in parallel. However, this would incur in over-subscription with the current environment: The Zynq system (with two Cortex-A9 SMP) and the current implementation of OmpSs, which uses one internal thread to submit the hardware computations.

Either way, once the programmer solves this application design issue, the dgemm and dsyrk tasks may turn to be in the critical execution path. The execution pattern of dgemm and dsyrk tasks is always the

same for each iteration of the algorithm. All the dgemm tasks are executed first, and then the dsyrk task execution follows, although there are no task dependencies between them. This behavior is attributed to the runtime scheduling policy, which results in a low degree of overlapping between these two kernels, as only the last dgemm task is able to overlap execution with the dsyrk task of the same iteration, as shown in Figure 29. A deeper analysis shows that the hardware computations of both kernels cannot concur more than 12% of the execution time. Therefore, improving the application design issue that was initially observed, will likely uncover this second issue regarding the runtime scheduling policy.

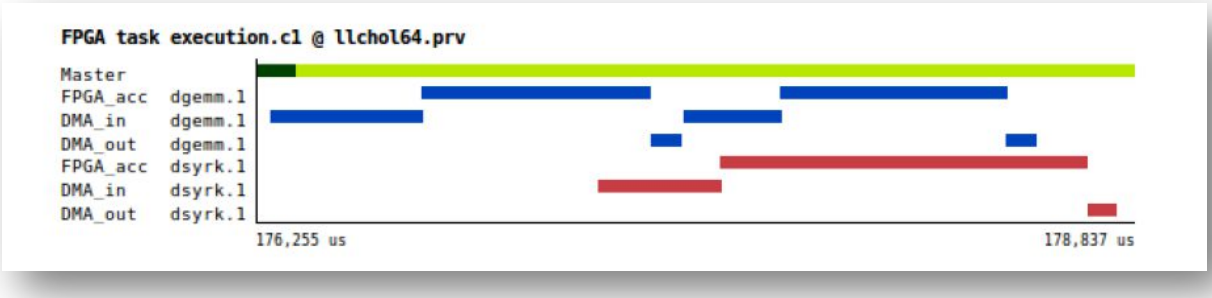


Figure 29. *Execution overlap of dgemm and dsyrk tasks. Dpotrf and dtrsm tasks run in SMP.*

A possible solution to reduce the potential scheduling policy impact is to enable one internal runtime thread per accelerator device, so that each internal runtime thread will take care of one type of accelerator. This solution would decouple the execution order of ready tasks in their corresponding hardware accelerators. However, this approach of using one internal runtime thread per accelerator is not the ideal solution, because that may entail over-subscription in systems with a small number of cores. Therefore, other solutions should be explored to parallelize task offloading using a single internal runtime thread. The current runtime implementation has only one internal runtime thread for all the hardware accelerators.

4.1.2 Matrix Multiply Analysis

Tiled matrix multiplication is analyzed varying the number of accelerators, the level of optimization of those hardware accelerators, and the number of MxM instances in the code (unroll degree). The problem size is a 256x256 matrix, divided into smaller blocks of 64x64 tiles, which are automatically offloaded to the accelerators, programmed in the FPGA, by the Nanos++ runtime of OmpSs. In addition, we present some results for 32x32 tiled matrix multiplication, which show some special characteristics due to the fine-grain granularity of the tasks.

Figure 30 (top) shows an execution trace of the application when using one single fpga accelerator device. Rows represent the different computational and communication components of the system. From top to bottom: the master thread (Master), the kernel computations (FPGA acc MxM.1), and the DMA memory transfer copies from main memory to the accelerator (DMA in MxM.1) and from the accelerator to main memory (DMA out MxM.1). We can observe that (1) all tasks are offloaded to the fpga accelerator device since there is not any task execution in the Master thread (this corresponds to the MxM target device specification), and (2) there are two MxM different tasks. In particular, the accelerator device is only one (FPGA acc MxM.1), but two different colors appear because two different

MxM instances in the OmpSs program are called (the innermost loop of matrix multiplication function in Figure 25 has been unrolled by two).

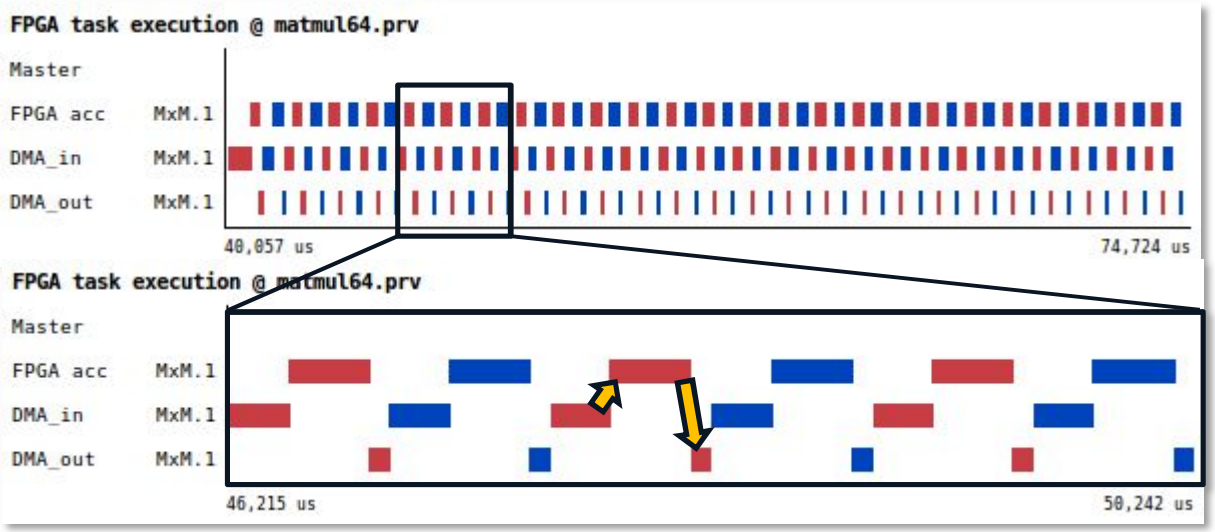


Figure 30. *Execution trace of a 256x256 MxM using unrolling 2 (two colors) and one FPGA accelerator of 64x64 size*

Figure 30 (bottom) shows a detailed view of the computation of 6 tiles (3 for each multiplication), where the reader can clearly see that they execute alternately. We can observe a clear dependence chain between computations and memory transfers. First, the data has to be copied from the main memory to the accelerator, which is shown in the DMA in row. As soon as the data has been copied, the computation of the task can start, displayed in the FPGA acc. Once the computation of the kernel has finished, the data is copied back to the main memory, as shown in the DMA out. We can observe that the next iteration does not start until the previous one has finished copying the data back to the host. Looking at the depicted execution pattern, we can also infer a potential improvement for the runtime, which could consider overlapping the input/output memory transfers and hardware computation between iterations since the DMA channels are independent.

Table 1 shows the average execution time of each of the stages of the task execution in an accelerator: input DMA transfer (DMA in), acceleration execution (FPGA acc) and output DMA transfer (DMA out). These measurements have been obtained using the Paraver profiling feature and validate that the MxM computation latency matches the High Level Synthesis tool estimation, and the DMA transfer times are very similar to the expected times.

	Task #1 (Red)	Task#2 (Blue)
FPGA acc MxM.1	334 us	337 us
DMA_in MxM.1	260 us	246 us
DMA_out MxM.1	82 us	82 us
Total	**674 us**	**665 us**

Table 1: Average time per computation and transfer for a 64x64 MxM.

On the one hand, the input/output DMA transfer time ratio is close to 3, and corresponds to the three input matrices and one output matrix needed by the hardware accelerator. However, this ratio may change for other task granularities. For instance, we have analyzed the trace execution for 32x32 block size (see Table 2) and the input/output DMA transfer execution times are very similar to the 64x64 block size, being larger than the expected for a finer granularity. In general, the DMA transfer performance may vary due to two main reasons: (1) the different DMA input/output bandwidth [13] and (2) the different waiting time for the corresponding DMA submit (i.e. the runtime programming the DMA), that can be significant for fine-grain task granularities. Note that a DMA transfer is not started until the corresponding submit is done. For instance, the task granularity in the 64x64 case is large enough to allow the runtime perform the DMA submit before the execution of the MxM task concludes. Therefore, the DMA out transfer can start immediately after the hardware computation, and the listed time accounts for actual transfer time. On the contrary, in the finer-granularity case of 32x32 MxM, with 8 less computation latency, the hardware computation is completed so fast that the runtime does not arrive on time to issue the DMA submit beforehand. Therefore, the DMA out transfer time of the 32x32 case also includes the waiting time for the DMA submit to arrive. All the above explains why the 32x32 MxM DMA out transfer time is higher than expected and similar to the 64x64 case.

On the other hand, the hardware computation and the input DMA transfer times are similar for the 64x64 MxM, but they may vary depending on the task granularity, the computation complexity and the hardware optimizations applied, which may be more or less aggressive depending on the FPGA resources availability. Thus, the execution time for two different approaches of the same 64x64 hardware accelerator may vary from 0.17ms to 26.34ms, having the same input and output memory transfer times. For an optimized version of a 32x32 tiled matrix multiplication, the input DMA transfer/FPGA execution time ratio goes up to 5 (Table 2), being the hardware computation time significantly lower than the data transfer times.

	Task #1 (Red)	Task#2 (Blue)
FPGA acc MxM.1	47 us	45 us
DMA_in MxM.1	229 us	256 us
DMA_out MxM.1	81 us	82 us
Total	**357 us**	**383 us**

Table 2: Average time per computation and transfer for a 32x32 MxM

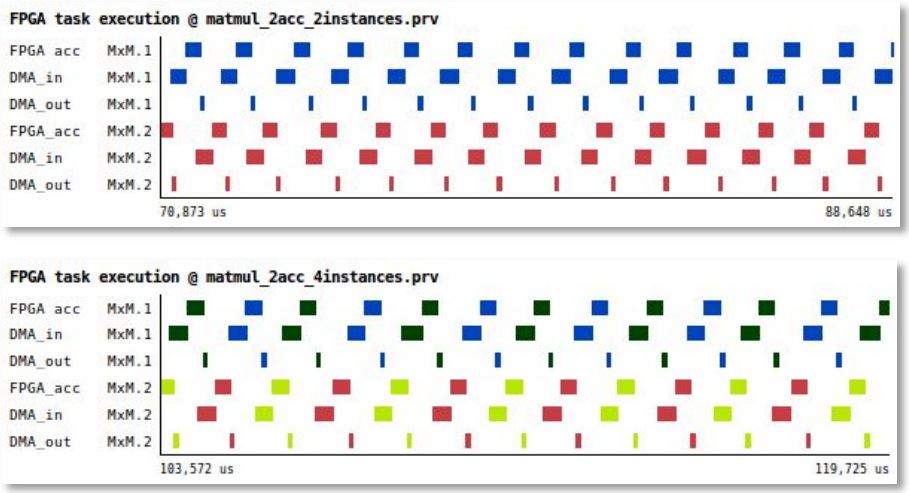


Figure 31. *MxM execution trace using two MxM hardware accelerators and two(top) and four(bottom) MxM task instances*

Figure 31 shows detailed views of the execution of the same problem using two accelerators and two and four matrix multiplication (MxM) instances respectively. In these cases, each MxM instance may be assigned to any of the two accelerator devices. It can be seen that tasks overlap in time, increasing the occupation of resources and the parallelism. Table 3 shows the execution time percentage of overlapping DMA memory transfers and hardware computations between the two accelerators. As the reader can see, there are overlaps between input and output DMA transfers, and between input DMA transfers and hardware computations. However, both input and output DMA channels are never active simultaneously due to the runtime's task scheduling pattern. Likewise, the hardware computations from both accelerators neither overlap due to their small execution times. For slower or more time-consuming accelerators, the hardware computations can overlap in time, as we can observe in Figure 32. This view presents a detailed zoom of the execution trace of a matrix multiplication, using two accelerators and four MxM instances, where the MxM tile is computed by a no optimized accelerator. In this case, the execution overlap between FPGA accelerators is above 90%.

	DMA_in MxM.1	DMA_out MxM.1	FPGA acc MxM.1
DMA_in MxM.2	0%	21.4%	17.0%
DMA_out MxM.2	20.1%	0%	0%
FPGA acc MxM.2	11.2%	0%	0%

Table 3: Percentage of time overlap between DMA transfers and MxM FPGA accelerator computation

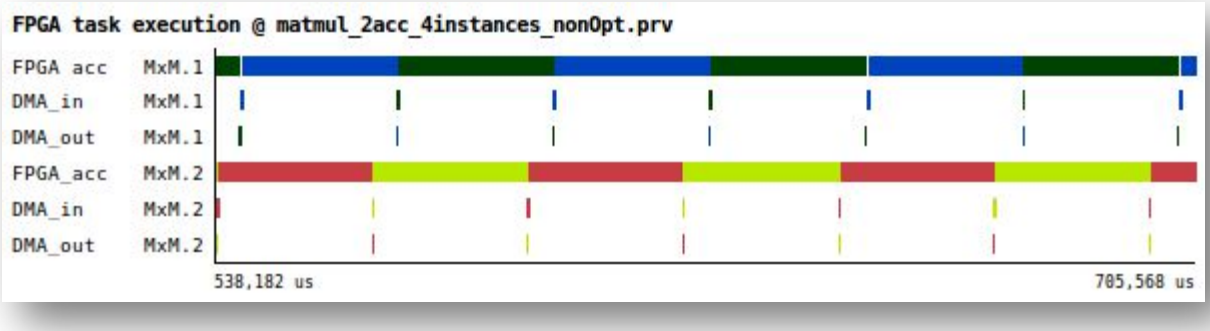


Figure 32. *Execution of 4 tasks using 2 time-consuming accelerators and 4 task instances*

Therefore, tracing the accelerator activity provides insight about DMA memory transfers and hardware computation overlap and their real latency information, which is not provided by any High Level Synthesis tool to the best of our knowledge. This analysis can help to improve the runtime memory management and scheduling policy.

4.1.3 OmpSs@FPGA Runtime Improvement Analysis

As commented, one of the reasons that makes the DMA transfer performance vary is the waiting time for each DMA submits of the DMA transfers to be done in a FPGA task execution. This difference is mainly due to the FPGA task communication model used in the current version of OmpSs@FPGA. In this model, one different DMA submit is required per task argument copy, in or out, before the corresponding DMA transfer starts. In order to reduce this difference, one possible improvement is to make the OmpSs runtime provide the necessary information of the copies (in and out's) to the accelerator, in just one DMA submit. With this unique submit, the FPGA accelerator can start all the necessary DMA copies without having to wait for each DMA submits.

A partial implementation of this new FPGA task communication model has been implemented in the current OmpSs@FPGA, with promising results for the full implementation at the end of the project. Figure 33 (a) and Figure 33 (b) are time scaled and show several FPGA task executions (64×64 tiles), with the same computation time (FPGA acc time) for two different versions of the OmpSs runtime where: (1) one DMA submit per task argument copy is necessary (top), and (2) just one DMA submit for all the task argument copies of a task execution is necessary (bottom). As it can be seen, meanwhile five full tasks can be run in the original version and six full tasks can be run in the improved runtime. This improvement is due to shortening the waiting time in the DMA transfers, as shown for the first task execution of the Figures. The DMA in of the first task in the original version is much

longer than the first DMA in of the first task of the improved version. Indeed, it also seems that a larger number of DMA submits, with the corresponding synchronization overheads, in the original runtime impacts the average DMA transfer time (DMA in and DMA out's); doubling the DMA in transfer time of the improved version.

More improvements are expected once the full implementation of the new task communication model is finished. Figure 33 (b) shows some idle time between the DMA out and DMA in of consecutive task executions. That is due to there are some removable runtime overheads of the previous communication model that are still in this partial implementation that provokes that idle time between two consecutive task executions.

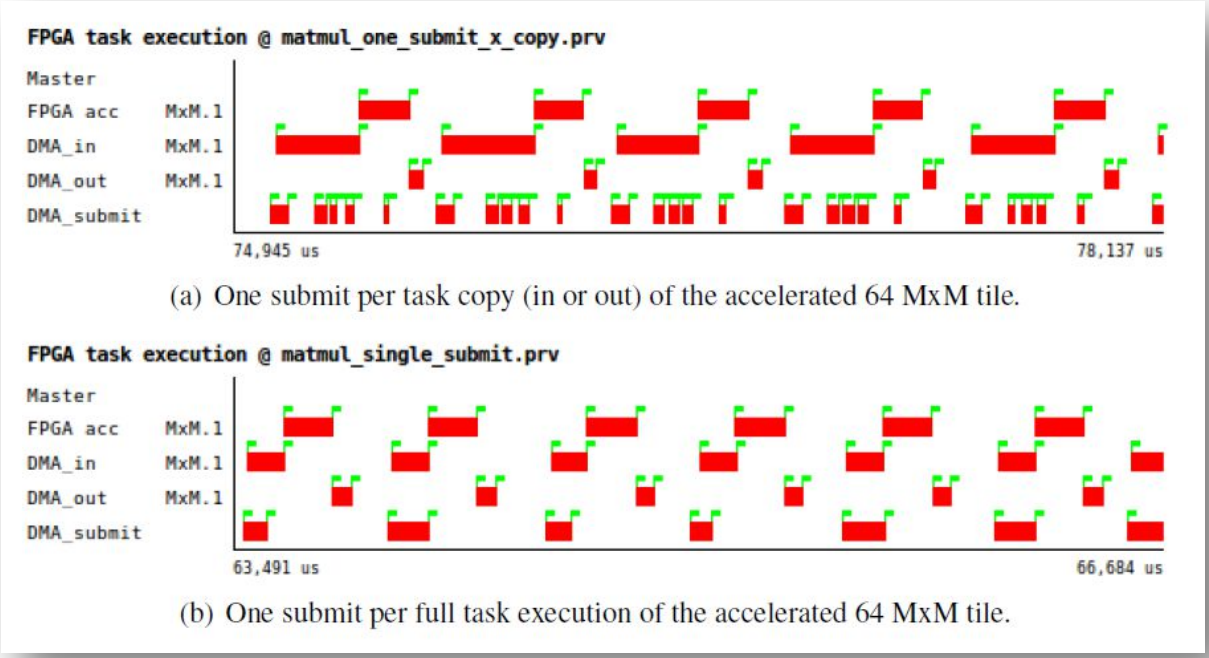


Figure 33. *Original (top) and improved (bottom) FPGA task communication model. Only one helper thread is used.*

4.2 OmpSs@Cluster on AXIOM Cluster (emulated w/ QEMU)

The physical Zynq Ultrascale+ based platform developed in the AXIOM project will be only available during the 3rd year of the project. During the 2nd year, thanks to the initial availability of a QEMU device tree directly from Xilinx, we have been able to emulate a complete AXIOM cluster, including the cluster setup (wired connections of the various boards), and the AXIOM NIC FPGA (we emulated the functionality of the AXIOM NIC register set following the specification agreed with FORTH). On top of the emulated environment, we have been able to run the complete OmpSs@Cluster stack, including the Extrae instrumentation presented in the previous Section.

Figure 34 shows the architecture implemented to emulate a cluster based on Xilinx Ultrascale+ machines representing an AXIOM cluster. The main architectural information is the following:

- **AXIOM Switch.**
  The AXIOM Switch is a separate host process that emulates the connections between the boards. It receives the packets to be sent from the various QEMU instances and it delivers the packets directly to the recipient following the routing tables (there is not an emulation of the routing of the packet through the various nodes). The AXIOM switch process is able to print diagnostic messages about the messages being delivered. It can be started with a regular (ring/2D mesh) topology, or with a custom topology specified in a text file.

- **QEMU Socket backend.**
  The QEMU implementation takes advantage of a Socket backend, which is a standard communication mechanism available in QEMU. At startup, each QEMU instance is connected using the socket backend to the AXIOM Switch process in order to simulate the connection of a board to the other boards.

- **QEMU Frontend.**
  The QEMU Frontend implements the AXIOM NIC Datasheet registers, thus emulating the real registers that will be available once the AXIOM board integrated with the AXIOM NIC IP will be available.

- **Virtual machine.**
  Each QEMU instance is programmed with a proper Device Tree Source (DTS) instantiating an amount of memory as the real AXIOM board, plus the AXIOM NIC Frontend. In this way, it is possible to run a Linux distribution on top of the virtual machine. Each Virtual machine will emulate a single board.

- **Linux distribution.**
  In order to emulate a set of Linux instances on a normal PC, we limited the Linux distribution to a minimal buildroot installation, which will include not only the basic buildroot but also all the precompiled binaries and dynamic libraries needed to start an OmpSs@Cluster application.

- **Cross-compilation of OmpSs@Cluster.**
  In order to be able to prepare the buildroot image, we prepared on the host a version of OmpSs, which is able to be cross-compiled for the ARM64 target.

- **Startup scripts.**
  We wrote a set of startup scripts, which are able to automatically start a number of QEMU emulator all connected to an AXIOM switch process. The AXIOM switch process is started with a customizable network layout. The scripts also take care of merging the Extrae data produced by the execution of the application.

- **VirtualBox Virtual Machine**
  To ease the deployment and distribution of the QEMU configuration, all the above has been packaged inside a VirtualBox x64 Virtual machine. The virtual machine has been released as part of the public content of deliverable D5.3.

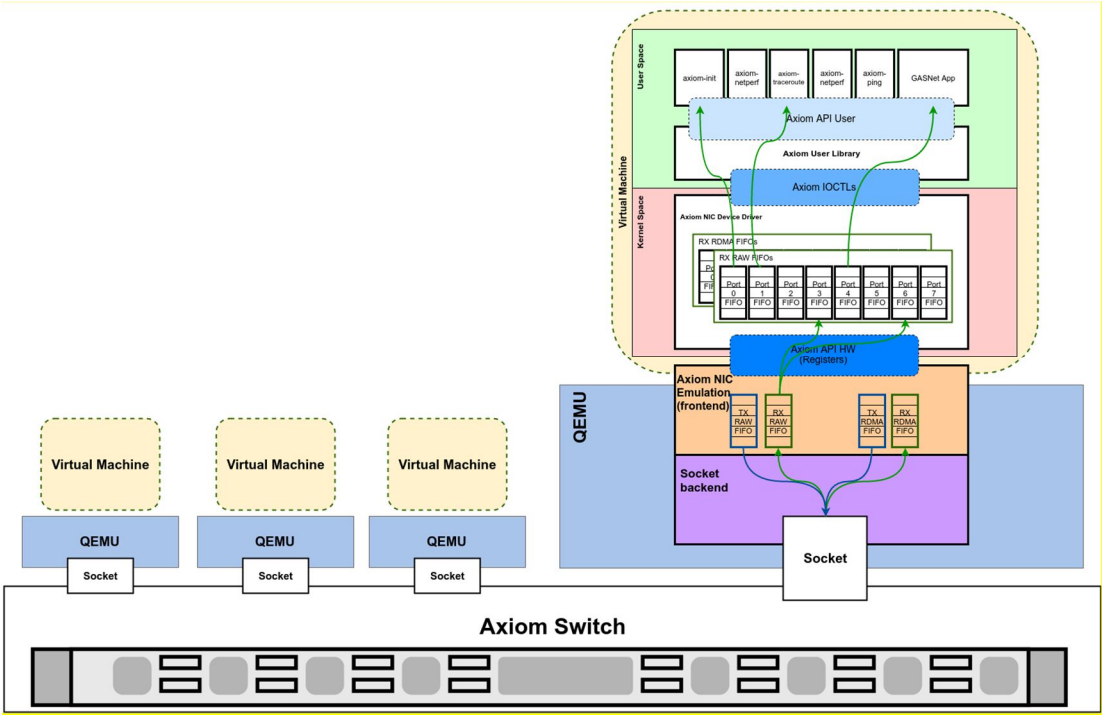


Figure 34. *The QEMU emulation infrastructure that is able to emulate an AXIOM cluster based on the Zynq Ultrascale+.*

By using the virtual machine, it is then possible to compile and execute OmpSs@Cluster applications. Figure 35 and Figure 36 show the Extrae trace results of the execution of a Matrix Multiply and of an Nbody example, respectively. The matrix multiplication experiment is done with a matrix size of 300x300 floating-point values, and a block size of 25x25 elements. The N-body experiment uses a problem with 256 particles, a block size of 128 particles, and it computes for 5 timesteps. Both experiments are done with 2 QEMU nodes, and running with 3 worker threads per node, and an additional communications thread in each node.

As expected, the traces are collected correctly, but their timings are neither synchronized nor reliable (this is normal in a QEMU emulation framework, as QEMU does not provide an accurate guest timing: it can only be useful to us for the software development but not for the performance evaluation).

However, despite the lack of timing reliability, the setup shows an initial promising integration of the complete toolset. Additional analysis will be performed during the third year once the AXIOM board will be delivered.

In each figure, the set of traces show the actual execution events that happen in each thread (see labels at the right-hand side, THREAD x.y.x), across time (X axis). The communication thread is number 2 in each node. The different views, from top to bottom, represent:

- Tasks and dependences: the execution of the tasks across time, linked with yellow lines representing the task dependences.
- Tasks and control dependences: the same task view with the creation links.

- Task number: it shows a color gradient based on the task identifier assigned by Nanos++. Tasks with low Id numbers are shown as light green, and tasks with a high Id number are shown in dark blue.
- AXIOM NIC API: it shows the calls being issued to the AXIOM NIC. Those calls are issued from the GASNet conduit implemented in T5.3, and triggered by the data movements generated from OmpSs@cluster. There are four main functions that get represented in these views:
    - Axiom_recv_*_avail(), in gray color, indicating that the conduit is checking whether there are messages available, of a specific type.
    - Axiom_send_long(), in light green color, to send a long message to a remote node.
    - Axiom_send_raw(), in yellow color, to send a raw message to a remote node.
    - Axiom_rdma_write(), in blue color, indicating a data transfer between the NIC memory and the host memory.

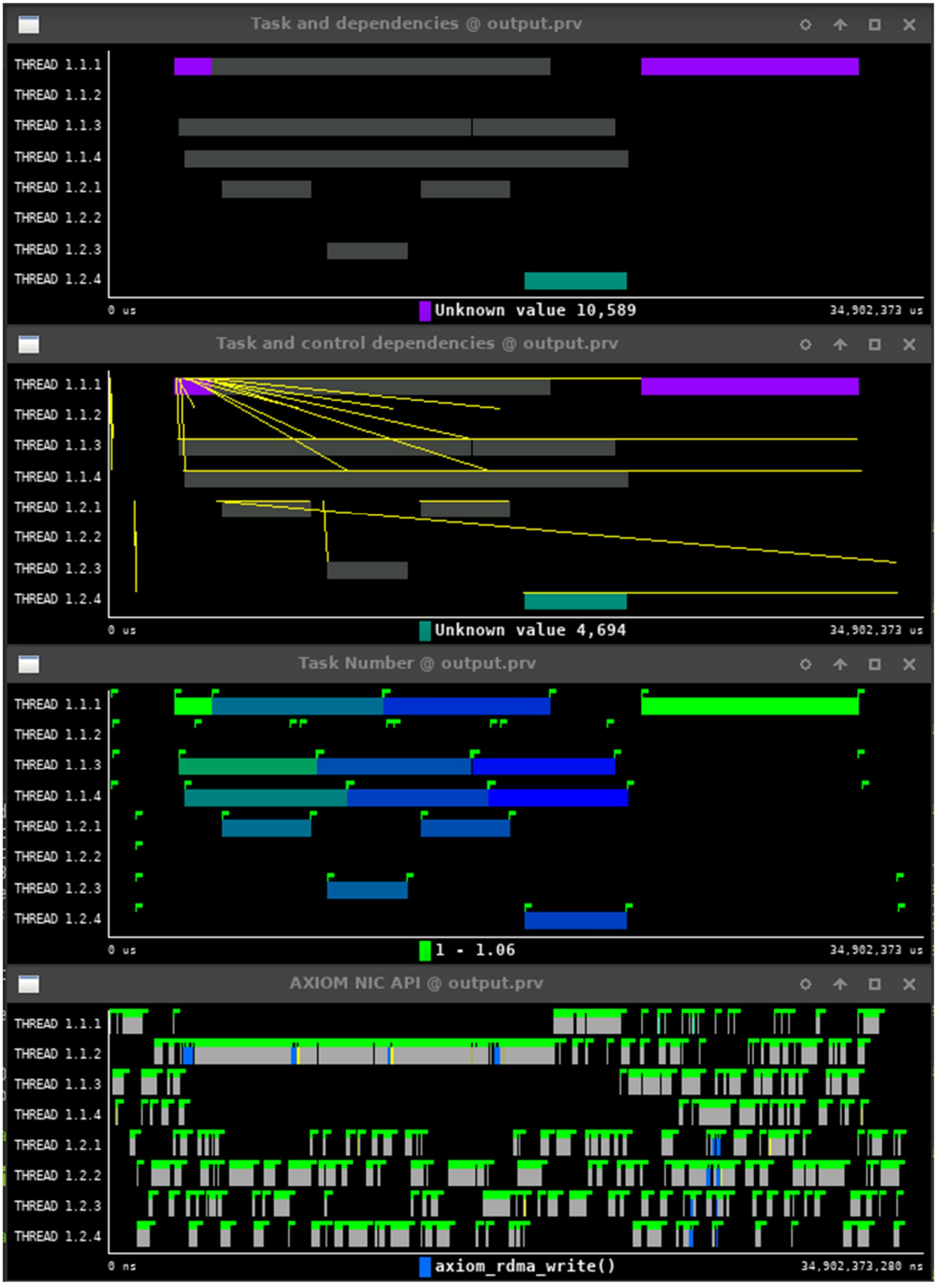


Figure 35. *PARAVER screenshot of an execution of a Matrix Multiplication on top of QEMU emulation.*

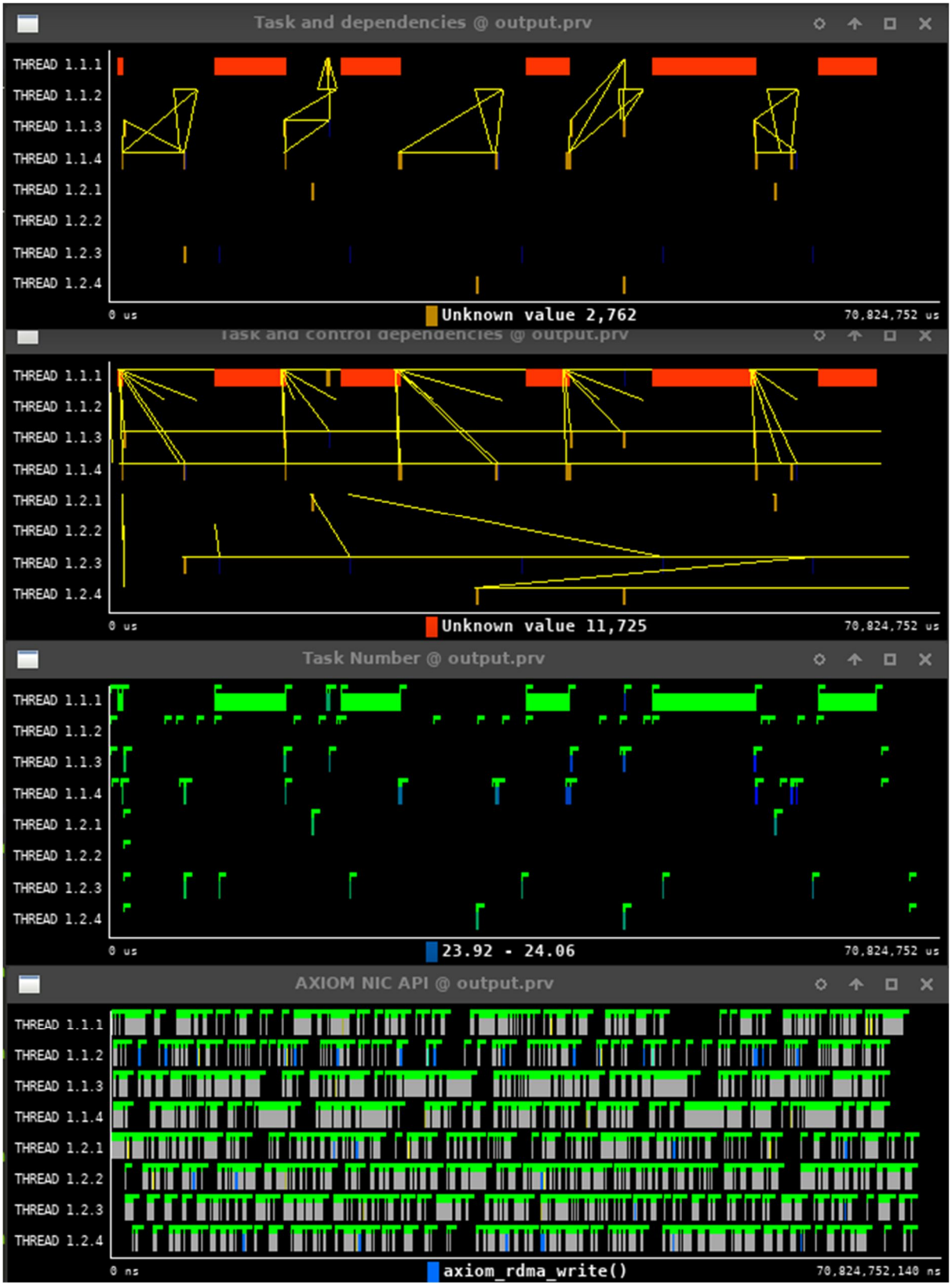


Figure 36. *PARAVER screenshot of an execution of an N-Body example on top of QEMU emulation.*

5 Confirmation of DoA objectives

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

PLANNED	DELIVERED
DELIVERABLE:	
• Documentation for the prototype compiler and first comparative research results. AXIOM code generation and instrumentation	The topics are described in this deliverable.

6 Conclusions

In this AXIOM deliverable, we have presented: (1) the programming model extensions proposed for programming the AXIOM boards with OmpSs; (2) the design of the support for the FPGA devices; (3) the automatic hardware generation framework; (4) the design exploration of the possible support of XSMLL from OmpSs annotated applications; and (5) the OmpSs@Cluster results for a QEMU environment emulating the Xilinx Zynq Ultrascale++ boards. The latter includes the GASNet conduit over the emulated AXIOM NIC.

Programming the AXIOM environment is currently much easier for the programmer with the automatic hardware generation, by using annotations in the OmpSs applications. The compilation process is almost transparent to the programmers so they can focus in the performance of the application. The programmers will program its task decomposition with the input/output/inout dependences with some of the task with target device FPGA. The compiler will automatically generate all the calls to the Nanos++ runtime in the SMP code to deal with the tasks, and the Nanos++ runtime will use the DMA library design presented in deliverable D4.1, to take care of data transfers between the host memory and the FPGA devices. Indeed, the programmers may decide to instrument to do a deeper analysis of the code, even inside the FPGA. The current compiler/runtime infrastructure transparently allows the programmers to obtain a Paraver trace and do this analysis. In this deliverable we have presented some of those possible analysis that has helped to have future research lines to tune the runtime and the applications.

Supporting distributed environments will be based on the communication layer (AXIOM-link) provided by FORTH to exchange data between AXIOM boards. We are also considering the use of common tools, like MPI or GASNet. In this deliverable we have presented OmpSs@Cluster on an AXIOM cluster, emulated with QEMU, using the GASNet conduit over the API of the future high interconnection link.

The next steps that will be taken in the project, related to this deliverable will be to fully implement a new mechanism of DMA transfers to improve the overall task executions, and develop mechanism to reduce the overall amount of DMA transfers. Also, having a mechanism to automatically generate hardware can help to evaluate the best software-hardware codesign to accelerate an OmpSs application. 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.

Other publications of the project [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] are reported in the reference list.

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. DMA driver for AXIOM: https://git.axiom-project.eu/?p=axiom-dma

6. XSMLL API for AXIOM: https://git.axiom-project.eu/?p=XSMLL

7. OmpSs website: http://pm.bsc.es/ompss

8. COTSon website: http://cotson.sourceforge.net/

9. 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

10. Xilinx, Inc. Vivado Design Suite – HLx Edition, http://www.xilinx.com/products/design-tools/vivado.html

11. 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

12. Zynq-7000 All Programmable SoC Overview. http://www.xilinx.com/support/documentation/data sheets/ds190-Zynq-7000-Overview.pdf.

13. Daniel Jiménez-González, Carlos Álvarez, Antonio Filgueras, Xavier Martorell, Jan Langer, Juanjo Noguera, and Kees A. Vissers. "Coarse-grain performance estimator for heterogeneous parallel computing architectures like Zynq All-programmable SoC". CoRR, abs/1508.06830, 2015.

14. R. Giorgi, "Transactional memory on a dataflow architecture for accelerating Haskell," WSEAS Trans. Computers, vol. 14, pp. 794–805, 2015.

15. R. Giorgi and A. Scionti, "A scalable thread scheduling co-processor based on data-flow principles," ELSEVIER Future Generation Computer Systems, vol. 53, pp. 100–108, July 2015.

16. D. Theodoropoulos et al., “The AXIOM project (agile, extensible, fast I/O module),” in IEEE Proc. 15th Int.l Conf. on Embedded Computer Systems: Architecture, MOdeling and Simulation, July 2015.

17. R. Giorgi, "Scalable Embedded Systems: Towards the Convergence of High-Performance and Em-bedded Computing", Proc. 13th IEEE/IFIP Int.l Conf. on Embedded and Ubiquitous Computing (EUC 2015), Oct. 2015.

18. 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.

19. A. Rizzo, G. Burresi, F. Montefoschi, M. Caporali, R. Giorgi, "Making IoT with UDOO", Interac-tion Design and Architecture(s), vol. 1, no. 30, Dec. 2016, pp. 95-112.

20. 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.

21. 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. Theodoropou-los, R. Giorgi, "AXIOM: A Hardware-Software Platform for Cyber Physical Systems", 2016 Eu-romicro Conf. on Digital System Design (DSD), Aug 2016, pp. 539-546.

22. 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.

23. P. Burgio, C. Alvarez, E. Ayguade, A. Filgueras, D. Jimenez-Gonzalez, X. Martorell, N. Navarro, R. Giorgi, "Simulating next-generation cyber-physical computing platforms", Ada User Journal, vol. 37, no. 1, Mar. 2016, pp. 59-63.

24. Jimenez-Gonzalez, Daniel; Alvarez-Martinez, Carlos; Filgueras, Antonio; Martorell, Xavier; Langer, Jan; Noguera, Juanjo; Vissers, Kees, “Coarse-Grain Performance Estimator for Heterogeneous Parallel Computing Architectures like Zynq All-Programmable SoC” (Journal Article) Second International Workshop on FPGAs for Software Programmers FSP 2015, CoRR , 2015.

25. R. Giorgi, S. Mazumdar, S. Viola, P. Gai, S. Garzarella, B. Morelli, D. Pnevmatikatos Dionisios and Theodoropoulos, C. Alvarez, E. Ayguade, J. Bueno, D. Filgueras Antonio and 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.