Breaking host-centric management of Task-Based Parallel Programming Models

Jaume Bosch Pons

June, 2021

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

PhD Thesis

Breaking host-centric management of Task-Based Parallel Programming Models

Jaume Bosch Pons

Supervisors Carlos Álvarez Martínez and
Daniel Jiménez González

June, 2021

Jaume Bosch Pons
Breaking host-centric management of Task-Based Parallel Programming Models
PhD Thesis, June, 2021
Supervisors: Carlos Álvarez Martínez and Daniel Jiménez González

Universitat Politècnica de Catalunya (UPC)
Departament d'Arquitectura de Computadors (DAC)
Carrer Jordi Girona, 1-3
08034, Barcelona

Barcelona Supercomputing Center (BSC-CNS)
OmpSs@FPGA Team from Programming Models Group
Carrer Jordi Girona, 31
08034, Barcelona

Abstract

Heterogeneous platforms have become popular to increase the computational power of the systems within a constrained power budget. They are present in several systems, from embedded platforms and mobile devices to high-end servers and clusters. However, the co-processors are managed following a master-slave model where the general-purpose CPU drives the rest of elements. This management limits the system possibilities as not all application parts are suitable to be executed in an accelerator. This thesis presents different proposals to enhance the usage of co-processors in task-based parallel programming models, which are a powerful tool to easily program applications for heterogeneous platforms.

The first proposal enhances the task-based systems with an asynchronous, concurrent, and parameterizable behavior. The improvements go across the full-stack, from the programming model level down to the low-level communications used between the libraries and the co-processors. The evaluation shows that the implemented improvements boost the applications' performance as they can be easily tuned for the running platform.

The second proposal adds support for task spawn and synchronization in co-processors. The offloaded tasks can create child tasks that target other architectures or remain inside the co-processor. This allows the programmers to implement applications easily and effectively. The evaluation shows the efficiency of the proposal implementation in terms of latency and power consumption. The results show that applications can increase their performance and optimize their power consumption by just moving the task spawn from the host threads to the co-processor. This is thanks to the low-latency task management inside the co-processors, which also reduces the communications between the host and the co-processor.

The third proposal extends task-based programming models with concepts of recurrent workloads. The regular task syntax has been extended with new clauses to label the recurrent tasks and provide the needed information to the runtime. The evaluation shows an application programmability increase thanks to the new syntax, which allows the specification of recurrent systems with much less code and better accuracy. Also, the direct management of task repetitions and periods in the co-processors allows an almost zero-latency management that is able to manage any task granularity.

Acknowledgement

This thesis has been possible thanks to its directors, Carlos and Dani. They provided their best with proposals, hints, and their knowledge. All of that became crucial for the proposals' definition and development. Also, it has been possible due to the institutional support of Xavier and Eduard.

The BSC colleagues helped a lot with their motivation, the discussions, the coffee breaks, and much more. The OmpSs@FPGA team has been crucial for the development of all tool-chain, which is the baseline of all proposals. Without all these BSC people, the thesis would not be the same.

This book would not exist without the constant pushing of Susanna, who persuaded me to continue during the last three years. In addition, the thesis is thanks to all my friends who probably were more excited about it than myself. Unquestionably, my family is the base of all the work presented here.

This thesis has been partially supported by Spanish Government through grant BES-2016-078046, project TIN2015-65316-P, and Severo Ochoa program SEV-2015-0493. It has also been partially supported by Generalitat de Catalunya through contracts 2014-SGR-1051, 2014-SGR-1272, 2017-SGR-1414, and 2017-SGR-1328. Finally, it has been partially supported by the European Union H2020 Research and Innovation Action through the following projects:

- AXIOM Project (Grant agreement ID: 645496).
- Mont-Blanc 3 Project (Grant agreement ID: 671697).
- HiPEAC (Grant agreement ID: 687698).
- EuroEXA Project (Grant agreement ID: 754337).
- LEGaTO Project (Grant agreement ID 780681).
- EPEEC Project (Grant agreement ID: 801051).

Contents

List of Figures

List of Tables

List of Listings

Glossary

Bitstream Configuration file that defines the programming information for an FPGA. 15, 29, 39, 43–45, 47, 56, 60, 71, 73, 81, 101, 106, 111, 112, 129, 143

Helper thread Thread that is mainly devoted to offload/synchronize tasks for co-processors. 50, 51, 71–73, 94, 95

IP block Unit of reusable logic which Intellectual Property (IP) owns to one party. 29–31, 55, 56, 58, 59, 88, 104, 106, 108, 109, 115, 116, 141

Lvalue Expression that refers to an object which occupies an identifiable memory location. 10, 11, 14

Processing Element Representation of a SMP core or a FPGA accelerator inside the Nanos++ runtime. xxiv, 50

Work Descriptor Representation of a task inside the Nanos++ runtime. It includes all information that the runtime may need to schedule and handle the task among its life. xxiv, 18, 19, 50, 93, 136, 138

Worker thread Thread that is mainly devoted to execute application tasks. 19–21, 50, 51, 70

xdma Library Linux library that exposes a high-level Application Programming Interface (API) for FPGA memory management and streaming communication. 29

xTasks Library Linux library that exposes a common API for tasks and memory management regardless of the FPGA and the underlying communication channel. 21, 28, 29, 43–45, 56–58, 93–95, 98–100, 105, 138

Acronyms

AI Artificial Intelligence. 1, 125

AIT Accelerator Integration Tool. 15, 29, 31, 41, 43, 45–47, 73, 102, 140, 142

API Application Programming Interface. xix–xxi, 15, 19, 21–27, 29, 37, 43, 45, 51–54, 57, 84–86, 88–90, 92, 93, 95–98, 132, 133, 135–139, 171

ASIC Application-Specific Integrated Circuit. 5, 125

BRAM Block Random Access Memory. 30–32, 35, 42, 43, 46, 47, 58, 59, 73, 77, 106, 108, 110, 120

CPU Central Processing Unit. v, 1–3, 125

DAST DAS Thread. xxiii, 19

DDAST Distributed DAST. 6, 19–21, 39, 50, 59, 60, 62–64, 66, 68–70, 77–79

DSP Digital Signal Processor. 1

FF Flip-Flop. 77

FPGA Field Programmable Gate Array. xiv, xv, 1–9, 12, 14–16, 18, 19, 21, 22, 26, 27, 29–32, 36–38, 40–47, 49–53, 55–64, 71–77, 79, 81, 84–90, 92–127, 129, 130, 132, 133, 135–150, 152, 153, 155–162

FPS Frames Per Second. xv, 157

GPGPU General-Purpose Graphics Processing Unit. 1, 37, 125

GPU Graphics Processing Unit. 1, 18, 19, 37, 41, 93

HLS High Level Synthesis. xx, 15, 77, 86, 88, 103, 104, 106, 108, 109, 133, 135, 136, 139–141, 172

HW HardWare. 37, 38

HWR HardWare Runtime. 45, 55–57, 85–90, 93–96, 98–100, 102–105, 109, 125, 126, 130, 135, 136, 138, 139, 141, 154, 159, 162

ID Identifier. vii, 16, 28, 58, 88, 103, 104, 106–108, 142

IP Intellectual Property. xxi, 125, 126, 162

LBP Local Binary Patterns. 155, 156

LUT LookUp Table. 77

MPI Message-Passing Interface. 36

PCI Peripheral Component Interconnect. xxiv, 125

PCIe PCI Express. 125

PE Processing Element. 50

POM Picos OmpSs Manager. 45

SMP Symmetric Multi-Processing. 3, 14, 19, 40, 41, 50, 51, 71–73, 83, 93, 115, 116, 122, 124, 125, 130, 132, 136, 144, 149, 150, 155, 156, 159

SMT Simultaneous Multi-Threading. 60

SoC System on Chip. 50, 112, 115, 125, 143, 161

SOM Smart OmpSs Manager. xiii–xv, 45, 55–59, 102, 104–106, 115, 116, 140, 141

WD Work Descriptor. 18, 19, 50, 93, 136, 138

1 Introduction

This chapter motivates the thesis research in section 1.1. Then, the main research objectives are explained in section 1.2. Section 1.3 lists and briefly summarizes the thesis contributions. Finally, section 1.4 describes the structure of the document.

1.1 Motivation

Multicore processors can be found in almost any electronic device, from small electrical appliances to huge servers. They have become popular as a way to keep increasing the computational power of general purpose processors after the end of Dennard scaling law [1]. With these processors, parallel programming models have grown as a way to easily define the applications parallelism and take advantage of these processors. Those programming models also decouple the applications from the running platform, creating a source code that can fit a wide range of hardware.

Heterogeneous platforms have also become popular to increase even more the computational power of the systems meanwhile the Moore law [2] keeps going. They allow keeping power consumption restricted and enhancing the processors' capabilities at the same time. Heterogeneity is present in several systems:

- Cell phones. They usually contain a Central Processing Unit (CPU) with small and big cores, a Graphics Processing Unit (GPU) (or General-Purpose Graphics Processing Unit (GPGPU)), Digital Signal Processors (DSPs), Artificial Intelligence (AI) accelerators.
- Embedded boards. They have the main processor and other co-processors like a GPGPU or an Field Programmable Gate Array (FPGA).
- Machines in the Top500 ranking.

New parallel programming models appeared with native support for heterogeneous systems (e.g., OpenACC [@3]). Also, some of the parallel programming models initially designed for the multicore processors introduced new features to support heterogeneous systems (e.g., OmpSs [4] and OpenMP [5]).

The master-slave management is commonly used for heterogeneous platforms, thereby the co-processors/accelerators are driven by the main general purpose CPU. This has some benefits from the programming model and runtime side. It simplifies the management of accelerators because they cannot actively interact with the runtime, and they are just passive components that operate over the memory under demand. However, the master-slave model limits the system possibilities as not all parts of an application are suitable to be executed in an accelerator.

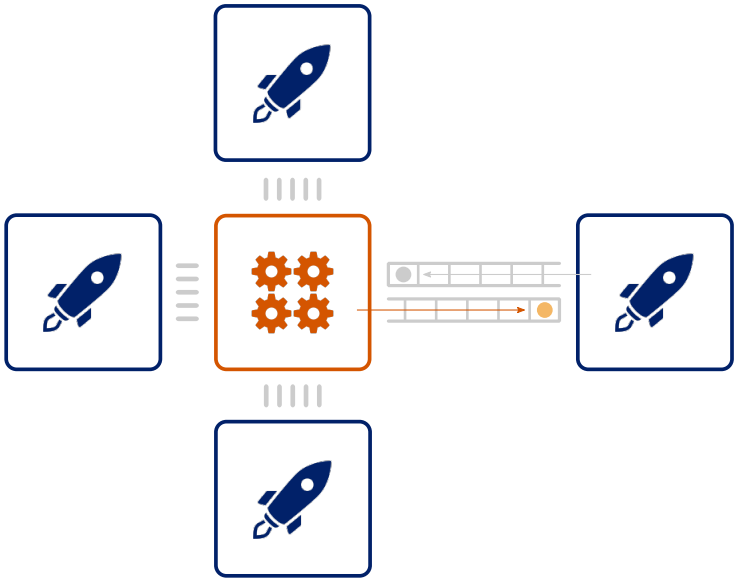


Figure 1.1: Master-slave model for co-processors management in task-based programming models

Figure 1.1 shows this master-slave model for co-processors management in task-based programming models. The orange square with gears in the middle represents the host, and the blue rockets the co-processors. The host is in the middle of the system with point-to-point connections to the four co-processors. The communication lanes between the host and the co-processors are detailed on the right side. There are two queues or channels used to offload the tasks and retrieve the executed ones.

The case of FPGA devices is a clear example of the master-slave model limiting the co-processor's capabilities. A single FPGA device may contain several accelerators that implement different operations. The fact of breaking the master-slave model implies the possibility of avoiding numerous synchronization overheads. These overheads are due to the host-centric management of accelerators and their reduction will lead to significant performance gains. In addition, the host threads are not always able to adequately feed the FPGA task accelerators in the FPGA device when dealing with fine-grained tasks. By creating and executing the tasks directly in the FPGA, the application will avoid the

host-FPGA latency allowing a faster task creation, gaining CPU time for other application activities, and increasing the resources productivity.

The programmability of tasks that become accelerators in the FPGA design is also important. There are pieces of code that cannot be executed in the FPGA (like Operating System code). Currently, if a task accelerator finishes processing some data and needs to load or store data from/to disk, some synchronization code in the host needs to monitor the task state and proceed adequately. With the proposed system, task accelerators can autonomously launch a host task which performs the disk access. Once task is completed and synchronized, the FPGA task accelerator can proceed. The same approach could be used for complex pieces of code that are called from inside the accelerated code but are executed infrequently and would consume several FPGA resources with a small performance advantage. The real-time face detection is an example as the FPGA efficiently processes the frames. However, the host CPU is responsible for coordinating the actions when a face is detected, although most of the time no faces are detected on frames. With an autonomous FPGA design, the algorithm in the FPGA can continually scan the video frames, and trigger an Symmetric Multi-Processing (SMP) task that will do the necessary actions in case of a positive detection. This approach does not need a continuous monitoring from the host CPU. Besides, systems that want to monitor a large set of signals also could benefit from an FPGA driven execution. Instead of having several threads monitoring the signals and checking if some action must be done, several monitoring FPGA task accelerators can be implemented. Those will occupy a small portion of the area and only call an SMP task when some action from the host is needed.

Last but not least, another interesting example involves the code compatibility between different systems. When programming a code to be executed in a given FPGA device, it usually is designed to operate over a given size of data related to the available resources. However, the chosen FPGA task accelerator may not fit in a new hardware requiring changing the host-FPGA communication interface. With an autonomous FPGA device, an FPGA task accelerator with the same interface that spawns the work in smaller FPGA task accelerators could be created. Even more, the new FPGA task accelerator could reverse-offload the work back to the host, losing the performance gains but maintaining the functionality. Given the vast design exploration space that FPGAs open for accelerator design, this last point is more important than it seems as it allows to refactor the FPGA task accelerators in a given FPGA after the application is compiled. The same solution also applies to situations where a new FPGA task accelerator is incorporated to the application or the application is executed concurrently with other applications that also need some FPGA space.

1.2 Objectives

This thesis aims to improve the current task-based parallel programming models extending them to enhance the use of co-processors/accelerators. The task-based parallel programming models (like OpenMP and OmpSs) are one simple but powerful way to express tasks (code regions) that can be executed in the accelerators. The main thesis objectives are briefly explained in the following points.

Asynchronous, concurrent and parameterizable task management

The first objective is to demonstrate that the efficiency of task management in task-based systems can be improved by asynchronous, concurrent, and flexible handling. The runtime activities may be decoupled from the element that initiates or requires the runtime action. Any thread must be able to perform data movements from/to co-processors, offload tasks, manage the instrumentation, etc. Applications should be able to customize the task management to fit their needs and maximize performance. This includes task definition at programming model level and task handling at runtime level.

Task spawn in co-processors

The second objective is to demonstrate the feasibility of spawning tasks and synchronizing them within the co-processors. The requirements for such capabilities are just a shared memory region between the accelerator and the host. Then, the accelerator can write there the required information to create a task in this memory region and continue. On the host side, some thread will read that information and update the runtime structures accordingly. This approach allows any co-processor to require runtime services asynchronously.

The proposed design for this objective can be extended with a smarter approach for FPGA devices that keeps the management of tasks inside the FPGA when possible. This requires a minimal runtime support in the FPGA device that handles the requests from the FPGA task accelerators and forwards them to the host runtime if needed. Otherwise, the runtime inside the FPGA may handle the request without involving the host runtime and save the communication latency.

Recurrent tasks

The third objective is extending task-based parallel programming models to support recurrent tasks and define how they interact with the other directives. Recurrent tasks process data in a dataflow manner alone or in cooperation with others. Those tasks are periodically under execution, either after some time before the last launch or immediately after the task returns from the user code. This kind of task is very valuable in real-time systems, such as image processing or sensor monitoring, and it matches the behavior of Application-Specific Integrated Circuits (ASICs) or small accelerators synthesized in an FPGA.

1.3 Thesis publications and contributions

The articles published during the thesis development are:

- Exploiting Parallelism on GPUs and FPGAs with OmpSs.
  Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell. ANDARE 2017. [6]
  The publication presents the OmpSs approach to deal with heterogeneous programming on GPU and FPGA accelerators, and it presents the performance obtained implementing the Matrix Multiplication with OmpSs in a Xilinx Ultrascale+ FPGA.

- Asynchronous Task Creation for Task-Based Parallel Programming Runtimes.
  Jaume Bosch, Xubin Tan, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé. OpenMPCon 2018. [7]
  The publication presents a general design to support the asynchronous creation and synchronization of tasks in parallel programming runtimes, and it analyzes the requirements to support those in co-processors like FPGAs.

- Application Acceleration on FPGAs with OmpSs@FPGA.
  Jaume Bosch, Xubin Tan, Antonio Filgueras, Miquel Vidal, Marc Mateu, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, Jesus Labarta. FPT 2018. [8]
  The publication presents the evaluation of the OmpSs@FPGA environment with the Matrix Multiplication, Cholesky and N-Body benchmarks, showing the internal execution details and obtained performance.

- Supporting task creation inside FPGA devices.
  Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González. BSC International Doctoral

Symposium 2019. [9]
The publication presents the implementation of a system to support the creation and synchronization of tasks inside the FPGA devices, it also presents an initial performance evaluation of the system.

- Breaking master-slave model between host and FPGAs.
  Jaume Bosch, Miquel Vidal, Antonio Filgueras, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé. PPoPP 2020. [10]
  The publication extends the initial evaluation done in [9] with real benchmarks and more insights of the implementation.

- Asynchronous Runtime with Distributed Manager for Task-based Programming Models.
  Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé. PARCO 2020. [11]
  The publication presents and evaluates the DDAST Manager which is a distributed runtime manager that reduces the task management overheads specially in many-core processors.

- Task-based programming models for heterogeneous recurrent workloads.
  Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Eduard Ayguadé.
  ARC 2021 [Accepted for publication]. [12]
  The publication presents the full implementation to support recurrent workloads in OmpSs and presents an extended evaluation with all performance results.

The thesis has been developed in the Programming Models Group, Computer Science (CS) department, Barcelona Supercomputing Center (BSC). During the development of the thesis, different collaborations with other groups have successfully used the knowledge and tools developed in this thesis. The following list (ordered by date) presents the publications related to this thesis and briefly describes its contribution:

- Picos, A Hardware Task-Dependence Manager for Task-Based Dataflow Programming Models.
  Xubin Tan, Jaume Bosch, Miquel Vidal, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, Mateo Valero. HPCS 2017. [13]
  The publication presents Picos which is a hardware task dependence manager, it is synthesized in an FPGA device and evaluated with different benchmarks.

- Hardware Heterogeneous Task Scheduling for Task-based Programming Models.
  Xubin Tan, Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé. OpenMPCon 2018. [14]

This publication presents an initial version of Picos++, which supports the data dependence management and scheduling of tasks for heterogeneous systems, and it presents a first performance evaluation.

- TaskGenX: A Hardware-Software Proposal for Accelerating Task Parallelism.
  Kallia Chronaki, Marc Casas, Miquel Moretó, Jaume Bosch, Rosa M. Badia. ISC 2018. [15]
  The publication presents the design and requirements of a hardware manager to accelerate the task management in the software runtimes.

- A Hardware Runtime for Task-Based Programming Models.
  Xubin Tan, Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, Mateo Valero. TPDS 2019. [16]
  This publication presents Picos++ which fully supports the data dependence management and scheduling of tasks for heterogeneous systems where task may be executed in different architectures.

- Adding Tightly-Integrated Task Scheduling Acceleration to a RISC-V Multi-core Processor.
  Lucas Morais, Vitor Silva, Alfredo Goldman, Carlos Álvarez, Jaume Bosch, Michael Frank, Guido Araujo. MICRO 2019. [17]
  This publication presents the integration of Picos with a RISC-V multicore processor, and it evaluates the new processor synthesized in an FPGA.

- Design and implementation of an architecture-aware hardware runtime for heterogeneous systems.
  Juan Miquel de Haro, Jaume Bosch, Daniel Jiménez-González, Carlos Álvarez. BSC International Doctoral Symposium 2020. [18]
  This publication presents the new Picos Daviu which integrates within the OmpSs@FPGA ecosystem to directly handle the data dependences of tasks spawned inside the FPGA devices.

- OmpSs@FPGA framework for high performance FPGA computing.
  Juan Miquel de Haro, Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, Jesus Labarta. TC 2021 [Accepted for publication]. [19]
  This publication presents the analysis and optimization of a different benchmarks in 2 FPGA boards using different optimizations and tweaks of OmpSs@FPGA ecosystem.

- High Performance Computing particle-pair distance algorithms, to generate X-ray spectra from 3D models.

César González, Jaume Bosch, Juan Miguel de Haro, Maurizio Paolini, Antonio Filgueras, Simone Balocco, Carlos, Álvarez, Ramon Pons. HPC 2021 [Under review]. [20]
This publication presents the analysis and optimization of a chemical application that simulates particles in an FPGA device.

1.4 Thesis Structure

The thesis is structured in different chapters that explain different parts of the developed work. However, all developments are related and the improvements in one proposal are needed, or used, in the others. Therefore, the work is presented by means of the different thesis objectives summarized in section 1.2. Then:

- Chapter 1, this one, introduces the thesis motivation, objectives and contributions.
- Chapter 2 describes the previous work related to this thesis and the baseline environment used to develop the thesis proposals.
- Chapter 3 presents the first thesis proposal that develops the asynchronous, concurrent, and parameterized concepts on task-based systems.
- Chapter 4 presents the second thesis proposal that develops the task spawn and synchronization inside FPGA devices.
- Chapter 5 presents the third thesis proposal that introduces the key concepts of recurrent systems into the task-based parallel programming models.
- Chapter 6 concludes the thesis, remarking the key contributions and summarizing the future work.

2 State of the Art

The OmpSs@FPGA ecosystem is an upgrade of the OmpSs [4] infrastructure (Mercurium source-to-source compiler and Nanos++ runtime) to incorporate FPGA support [8]. This ecosystem is the baseline of the proposals and development of this thesis. The sections 2.1 to 2.6 describe the different elements/levels that compose the OmpSs@FPGA ecosystem. Finally, section 2.7 describes the related work with the thesis components, objectives and goals.

2.1 OmpSs Programming Model

OmpSs is a task-based parallel programming model that extends the OpenMP 3.1 [@21] syntax and been an active forerunner of the current tasking capabilities available in OpenMP 5.1 [@22]. It uses an implicit parallel region, in contrast to OpenMP which requires an explicit parallel region. OmpSs supports different annotations for expressing parallelism, but the task annotation is the one used for Heterogeneous systems [23]. The following sections explain the tasking and heterogeneity model of the programming model.

2.1.1 Tasking model

The task paradigm is a widely used approach to express portions of code that are asynchronously run. OmpSs has the `task` compiler directive to explicitly annotate code regions that will be asynchronously executed by any available thread. It can be used in two ways: 1) before a code region (inline tasks) as shown in listing 2.1; 2) before a function declaration (outline tasks) as shown in listing 2.2. The body of inline tasks is asynchronously executed when encountered, and the function body of outline tasks is asynchronously executed every time the function is called.

```
1 double vec[10];
2 #pragma omp task
3 {
4   for (int i=0; i<10; i++) vec[i] = 0.0;
5 }
```

Listing 2.1: Inline OmpSs task example

```
1 #pragma omp task
2 void vec_init(double *vec);
```

Listing 2.2: Outline OmpSs task example

Task dependences

Tasks can be implicitly synchronized using task dependences. The dependences are defined for each task by means of accessed memory regions and their directionality. The following clauses are available in the task directive to specify the data and directionality of accessed data:

- `in(memory-reference-list)`. This clause defines a list of lvalues that are read by the task. The task execution is postponed until all predecessor sibling tasks with an `out`, `inout`, `concurrent` or `commutative` clause applying to some lvalue from the list are executed [@24].
- `out(memory-reference-list)`. This clause defines a list of lvalues that are written by the task. The task execution is postponed until all predecessor sibling tasks with an `in`, `out`, `inout`, `concurrent` or `commutative` clause applying to some lvalue from the list are executed [@24].
- `inout(memory-reference-list)`. This clause defines a list of lvalues that are read and written by the task. The task execution is postponed until all predecessor sibling tasks with an `in`, `out`, `inout`, `concurrent` or `commutative` clause applying to some lvalue from the list are executed [@24].
- `concurrent(memory-reference-list)`. This clause defines a list of lvalues that are read and written by the task. The task execution is postponed until all predecessor sibling tasks with an `in`, `out`, `inout`, or `commutative` clause applying to some lvalue from the list are executed. In contrast to `inout`, this clause allows tasks with same lvalue within it execute concurrently [@24].
- `commutative(memory-reference-list)`. This clause defines a list of lvalues that are read and written by the task. The task execution is postponed until all

predecessor sibling tasks with an `in`, `out`, `inout`, or `concurrent` clause applying to some lvalue from the list are executed. Moreover, the clause defines a mutex for each lvalue on it. The mutex applies to all tasks with the same lvalue on the clause, and it only allows one of those tasks to run in parallel [@24].

The format or lvalues in the memory references list can be one of those:

- Single elements. Discrete lvalue over some variable.
- Array section. Section of an array or pointed data. It can be done in two forms:
  - `a[lower : upper]`. All elements of `a` from lower to upper (both included) are referenced. If lower is not provided, it is assumed to be 0. If upper is not provided and the expression refers to an array with a known size, the last element of the array is assumed.
  - `a[lower ; size]`. All elements of `a` from lower to lower+size-1 (both included) are referenced.
- Shaping expression (`[size]a`). Reshapes a pointed data into an array of known size whose elements are referenced.

Listing 2.3 shows an example of an outline task with an output data dependence. The dependence is over the first ten elements pointed by the `vec` parameter.

```
1 #pragma omp task out(vec[0; 10])
2 void vec_init(double *vec);
```

Listing 2.3: Outline OmpSs task with data dependence example

Taskwait

The tasks can be explicitly synchronized using the `taskwait` directive. It ensures that, after the directive, the tasks spawned in the same nesting level have finished their execution. An example of this directive can be seen in listing 2.4. Before the taskwait, the state of spawned tasks is not known, so it is not possible to know if `vec` has been initialized or not. After the taskwait, the programming model ensures that tasks have been synchronized, so `vec` variable has been initialized.

```
1 double vec[10];
2 #pragma omp task
3 {
4   for (int i=0; i<10; i++) vec[i] = 0.0;
5 }
6 // vec[i] may not be initialized
7 #pragma omp taskwait
8 // vec[i] is initialized
```

Listing 2.4: Taskwait example

2.1.2 Heterogeneity support

The task directive can have a target directive associated that defines where the spawned task will be executed. The architecture is specified through the `device` clause which default value is `smp` (the one of general purpose processors). The main architectures are: `smp`, `gpu`, `opencl` [23], `fpga` [6], etc. An example of a OmpSs task with the FPGA architecture is shown in listing 2.5.

```
1 #pragma omp target device(fpga) onto(10)
2 #pragma omp task out(vec[0; 10])
3 void vec_init(double *vec) {
4   for (int i=0; i<10; i++) vec[i] = 0.0;
5 }
```

Listing 2.5: Outline OmpSs task with FPGA device example

The usage of the `device(fpga)` clause implies that the associated task will become an FPGA task accelerator. All tasks that have the clause in the source code will be placed together in the FPGA design.

Task type

One FPGA design may contain several FPGA task accelerators of different types, like an application binary may contain serveral functions. In contrast to binaries, the FPGA design cannot relay on memory pointers to determine which is the right function for a given tasks. Instead, an integer identifier is assigned to each FPGA task, so the runtime can match the created tasks with the right FPGA task accelerators.

The unique integer identifier for each FPGA task is assigned through the `onto` clause, which usage is mandatory. The clause takes one numerical value, which must be unique between all tasks with the `device(fpga)` clause in the same FPGA design. An example

of clause's usage is shown listing 2.5 (line 1) where the task is arbitrary labeled with a 10.

Task copies

The tasks may be executed in a device that does not have access to the host address space. Therefore, data movements between devices may be required before and/or after the execution of tasks with different architectures. OmpSs provides different clauses in the target directive to define the task data copies. Those copies are the data regions that must be mapped to device address space and where data must be copied before and/or after the task execution. The clauses are:

- `copy_in(shaping-expression-list)`. Defines a list of memory regions to be copied into the device address space before the task execution.
- `copy_out(shaping-expression-list)`. Defines a list of memory regions to be copied from the device address space after the task execution.
- `copy_inout(shaping-expression-list)`. Defines a list of memory regions to be copied into and from the device address space, before and after the task execution.
- `copy_deps`. Populates the copy clauses with the dependence clauses information. For each `in`, generates a `copy_in`; for each `out`, generates a `copy_out`; and for each `inout`, generates a `copy_inout`. This is the default behavior if no `copy_in`, `copy_out` or `copy_inout` clause are provided.
- `no_copy_deps`. In contrast to `copy_deps`, do not use the dependence clauses information to full-fill the copy clauses.

Taskwait

The taskwait directive supports a couple of clauses that enhance its behavior, specially in Heterogeneous systems. The clauses are intended to finely tune the behavior of the directive and maximize application performance. By default, the taskwait directive synchronizes the child tasks, and it also synchronizes their output data copies from devices address spaces to the main application address space. They are:

- `noflush`. Only synchronizes child tasks but not their output copies. Then, the data its kept in the device address space, which could improve data locality in the copies of following tasks.

- `on(memory-references-list)`. Defines a list of lvalues that must be synchronized instead of all child tasks. The taskwait is ready when all previously created child tasks with an `out`, `inout`, `concurrent` or `commutative` clause applying to the some lvalue in the list have finished.

Implements

The tasks may target more than one device and have different implementations in each one. This is achieved using the `implements` clause of `target` directive. The clause value is the name of the function task (A) which is being implemented by the former (B). Then, any call to task A may end in the execution of B. An example of an OmpSs task with the implements clause is shown in listing 2.6. The example shows the task function `matmul_block`, that has FPGA architecture, and task function `matmul_block_smp`, that has SMP architecture and implements the first using a cblas call.

```
1  #pragma omp target device(fpga)
2  #pragma omp task in([BSIZE*BSIZE]a, [BSIZE*BSIZE]b) inout([BSIZE*BSIZE]c)
3  void matmul_block(const float *a, const float *b, float *c) {
4    //...
5  }
6
7  #pragma omp target device(smp) implements(matmul_block)
8  #pragma omp task in([BSIZE*BSIZE]a, [BSIZE*BSIZE]b) inout([BSIZE*BSIZE]c)
9  void matmul_block_smp(const float *a, const float *b, float *c) {
10   cblas_gemm(ROW_MAJOR, NO_TRANS, NO_TRANS, BSIZE, BSIZE, BSIZE, 1.0, a,
11              BSIZE, b, BSIZE, 1.0, c, BSIZE);
12 }
```

Listing 2.6: OmpSs task with implements example

2.2 Mercurium

Mercurium is a C/C++/Fortran source-to-source compilation infrastructure aimed at fast prototyping developed by the Programming Models group at the Barcelona Supercomputing Center. [@25]

Mercurium is used, together with the Nanos++ Runtime Library, to implement the OmpSs programming model [@24]. Both tools provide also an implementation of OpenMP 3.1. More recently, Mercurium has been also used to implement the OmpSs-2 programming model [@26] together with the Nanos6 Runtime Library [@27]. Apart from that, since Mercurium is quite extensible it has been used to implement other programming models

or compiler transformations, examples include Cell Superscalar, Software Transactional Memory, Distributed Shared Memory or the ACOTES project. [@25]

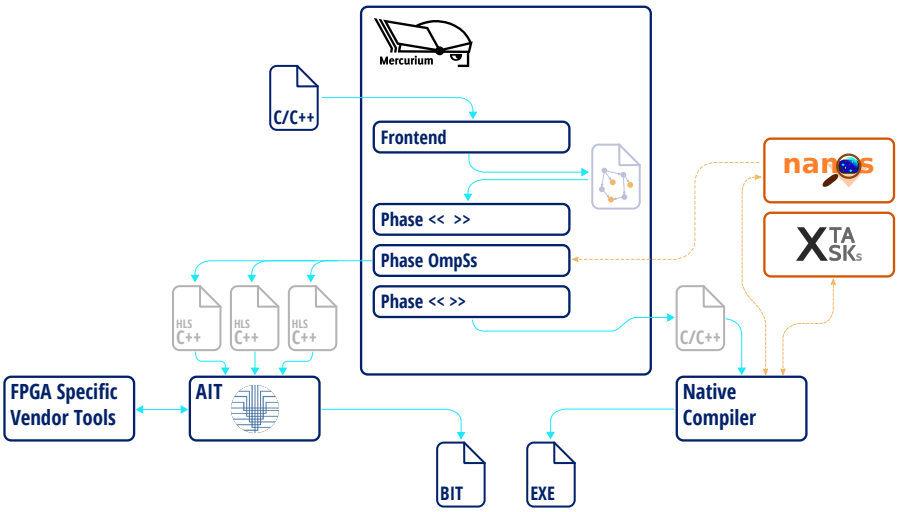


Figure 2.1: Mercurium compiler structure

Figure 2.1 shows a simplified view of an application compilation and linkage using Mercurium. The C/C++ source code files are provided to Mercurium profile, which first parses them in the frontend creating an internal common high-level representation. The following phases use this representation to analyze and modify the application code. It is finally pretty-printed to a new C/C++ source file that is compiled and linked using a native compiler. The different phases executed consecutively during the compilation handle different parts and perform code transformations accordingly to their purpose. For example, the OmpSs phase replaces the compiler directives by Nanos++ API calls. In addition, the OmpSs phase generates intermediate C++ High Level Synthesis (HLS) source code files for each FPGA task found in the source code. Those files are used by Accelerator Integration Tool (AIT) (formerly autoVivado) [28] to generate the FPGA bitstream, which includes the extra design stuff to allow the communication at runtime with the FPGA task accelerators. AIT relies on the specific FPGA vendor tools (Xilinx Vivado and Vivado HLS [@29] [30]) for the bitstream generation. Besides, the compiler also uses an underlying native compiler to generate the application binary from the new C/C++ source code. This binary is linked against the different libraries (Nanos++, xTasks) that support the execution at run time.

2.2.1 HLS Source Code

Mercurium generates a C++ HLS source code file for each FPGA task. This HLS source code includes the task function code, the symbols used in that function, and an

FPGA task accelerator wrapper. The wrapper is a function specifically generated by the compiler for each FPGA tasks that wraps the user function code and implements the communication protocol against the Task Manager. This communication is based on two streams (input and output). The input stream is used to retrieve the task information and arguments needed for launching the task function code. The output stream is used to notify the finalization of task execution.

As an example, listing 2.7 shows a simplified version of the FPGA task accelerator wrapper generated for the task shown in listing 2.5. The wrapper has a port to access the FPGA task accelerator identifier, the two AXI-Stream interfaces connected to Task Manager, and an AXI interface to access the data of `vec` parameter of the user function. The wrapper can be divided into six parts:

1. Retrievement of ready task header words (lines 22-27).
2. Retrievement of task arguments (lines 30-24).
3. Read of input data. In the example there is no input data, then this part is not generated by the compiler.
4. Execution of user function if compute flag is enabled (line 37).
5. Write of output data if the output flag is enabled (lines 40-42).
6. Submit of task Identifier (ID) in the output stream (line 49).

2.3 Nanos++

Nanos++ is a runtime library designed to serve as runtime support in parallel environments. The runtime is developed at the Barcelona Supercomputing Center within the Programming Models group, and its main use is to support the OmpSs programming model. Apart from OmpSs, Nanos++ also supports most of the OpenMP 3.1 features and includes some additional extensions (some of them also introduced in following OpenMP releases). [@31]

The runtime provides the required services to support task parallelism based on data dependences. Data parallelism is also supported by means of services mapped on top of its task support. Tasks are implemented as user-level threads when possible (currently x86, x86-64, ia64, arm, ppc32 and ppc64 are supported). It also provides support for

```
void vec_init (double * vec);

void vec_init_mcxx_hls_wrapper(
  const ap_uint<5> mcxx_acceleratorID,
  hls_axis_t mcxx_inStream, hls_axis_t mcxx_outStream,
  double *vec_port)
{
  #pragma HLS INTERFACE ap_ctrl_none port=return
  #pragma HLS INTERFACE axis port=mcxx_inStream
  #pragma HLS INTERFACE axis port=mcxx_outStream
  #pragma HLS INTERFACE m_axi port=vec_port

  float vec[10];
  unsigned long long int _params[1];
  unsigned char _paramFlags[1];
  unsigned long long int _taskID, _tmp;
  unsigned long long int _instr_timerAddr, _instr_bufferAddr;
  unsigned int _compteFlags, _destinationID;
  hls_axis_data_t _axisPkg;

  // Read the ready task header words
  _taskID = mcxx_inStream.read().data;
  _instr_timerAddr = mcxx_inStream.read().data;
  _instr_bufferAddr = mcxx_inStream.read().data;
  _tmp = mcxx_inStream.read().data;
  _compteFlags = tmp;
  _destinationID = tmp >> 32;

  // Read the task arguments
  for (unsigned int i=0; i<1; i++) {
    _tmp = mcxx_inStream.read().data;
    _paramFlags[tmp >> 32] = _tmp;
    _params[_tmp >> 32] = mcxx_inStream.read().data;
  }

  // Execute the task body
  if (_compteFlags) vec_init(vec);

  // Write the output data
  if (_paramFlags[0]&0x20 != 0x00) {
    memcpy(vec_port + _params[0], vec, 10*sizeof(double));
  }

  //Sync the task execution in mcxx_outStream
  _axisPkg.data = _taskID;
  _axisPkg.last = 1;
  _axisPkg.tid = mcxx_acceleratorID;
  _axisPkg.tdest = _destinationID;
  mcxx_outStream.write(_axisPkg);
}
```

Listing 2.7: FPGA task accelerator wrapper example

maintaining coherence across different address spaces (such as with GPUs or cluster nodes) by means of a directory/cache mechanism. [@31]

The main purpose of Nanos++ runtime library is to be used in research of parallel programming environments. The runtime tries to enable easy development of different parts, so researchers have a platform that allows them to try different mechanisms. As such it is designed to be extensible by means of plugins. The scheduling policy, the throttling policy, the dependence approach, the barrier implementations, slicers and worksharing mechanisms, the instrumentation layer, and the architectural dependant level are examples of plugins that developers may easily implement using Nanos++. [@31]

Those plugins isolate the different runtime services creating an internal interface to exchange information between. Therefore, each plugin can be transparently replaced without affecting the others. For example, the different instrumentation plugins generate different types of traces with different application' insights, or they use different underlying instrumentation infrastructure (Extrae, OMPT, etc.). The architecture plugins also provide a common interface for task management agnostic of the underlying device where the task will be executed (GPU, FPGA, or a regular processor thread).

2.3.1 Task Life Stages

The tasks spawned by the application are internally represented by Work Descriptors (WD) in Nanos++. This structure contains all information that runtime needs to correctly handle the task among its life cycle. It includes task dependences, devices where the task can be executed, task arguments, mappings of memory in device address space (if any), and more. During the task life, it goes around runtime modules to handle its requirements. The different stages that a Work Descriptor (WD) goes through during its life are:

1. Task creation. The WD had been allocated and filled with the basic task information.

2. Task submission. The task has been inserted into the task dependence graph and is waiting for its predecessor tasks to finish. This stage is skipped if the task does not have data dependences.

3. Task ready. Once the predecessor tasks finish, the task becomes ready and is forwarded to the scheduler module. The task will wait until the assigned resource for execution is available to start the task execution.

4. Task memory allocation. Once the device where the task will be executed is assigned, the runtime checks if some memory must be allocated in the device address space to handle it. It may happen that the device memory contains the data of other tasks, and the former has to wait until the previous ones finish. This stage is skipped if the task does not have data copies.

5. Task input data copy. Once the memory for task copies has been allocated, the runtime starts moving the input data to the mapped regions. These copies may be asynchronously handled, so the task has to wait until they are finished. Also, it may happen that data was moved to a device address space by a predecessor task and has to be moved/copied into another address space. This stage is skipped if the task does not have data copies.

6. Task running. Once all data is ready for task execution, the task execution starts. It may be a synchronous execution if task is run in an SMP device by a regular thread, or it may be asynchronously executed if the task is run in a GPU, FPGA, etc. device where tasks are offloaded.

7. Task blocked. During the task execution, it may become blocked as a result of some runtime API call. For example, a task becomes blocked when it has a taskwait until its child tasks finish.

8. Task finished. Once the task execution finishes, the task dependences are released, and successor tasks may become ready.

9. Task output data copy. The data generated by the task is moved back to the original memory region, the mapping is invalidated, and the device memory region is released. This stage is skipped if the task does not have data copies.

10. Task cleanup. Once the child tasks have finished, and all data copies have been invalidated, the WD is deallocated.

2.3.2 DDAST

The Distributed DAST (DDAST) Manager [32] is a distributed vesrsion of the centralized DAS Thread (DAST) [33]. The manager is responsible of executing the runtime code that manages tasks at different critical points of their life. The aim of decoupling runtime and application activities is avoiding the bottlenecks created when several threads try to concurrently update some runtime structures. Then, the worker threads send messages to the manager requesting some runtime operations instead of directly executing them.

The distributed manager does not use dedicated threads to handle the messages. Instead, any worker thread can become part of the manager and start executing only runtime code. With this approach, all threads can cooperate to execute the pending runtime operations when there are several of them. Correspondingly, all the threads can execute application tasks when the number of pending runtime operations is small. The manager design is based on general modules that can be extended to support other runtime functionalities.

The messages (request of runtime operations) sent by the worker threads to the runtime manager can be of two types: *Submit Task Message* and *Done Task Message*. The first one, the *Submit Task Message* is sent when a worker thread wants to submit a new task into the runtime structures to find out its predecessor tasks. The second one, the *Done Task Message* is sent when a worker thread finishes the execution of a task and wants to notify the successor tasks, scheduling them if they become ready.

Functionality Dispatcher

The Functionality Dispatcher is a module that mediates between different runtime parts decoupling the computational resources from specific runtime functionalities. The module allows using the idle threads to execute any runtime operation. Therefore, the runtime functionalities can be handled without having computational resources exclusively dedicated to them.

The runtime modules register a callback function in the Functionality Dispatcher during the runtime initialization. Also, the worker threads notify the module when they do not have tasks to execute, so they are idle. Therefore, the Functionality Dispatcher tries to take advantage of those idle resources and uses them to execute the registered callbacks.

DDAST Callback

The DDAST Manager is implemented using one callback function registered in the Functionality Dispatcher. Therefore, the callback is executed when a worker thread becomes idle and this thread starts handling the pending messages. An idle worker thread usually means that the pending messages must be processed to submit more tasks into the dependence graph or to schedule some new ready tasks.

The behavior of the DDAST callback is parametrized by different constants defined during the runtime initialization. Here follows a brief list and explanation of these variables::

- `MAX_DDAST_THREADS`. Maximum number of threads allowed to execute the DDAST callback concurrently.
- `MAX_SPINS`. Number of times that the thread will try to get messages without success before leaving the DDAST callback.
- `MAX_OPS_THREAD`. Maximum number of messages satisfied from the same worker thread to force changing to another worker thread.
- `MIN_READY_TASKS`. Minimum number of ready tasks available to force exiting the DDAST callback.

2.4 xTasks Library

The xTasks library is in charge of abstract Nanos++ from board dependent communication protocols. It exposes an API where the main elements are accelerators, which internally have associated a communication channel, and tasks, which can be sent to those accelerators. It also provides some memory management APIs like allocating memory in the FPGA address space and copy data from/to there. Finally, the library has an instrumentation API to retrieve the tracing events generated by the FPGA task accelerators. Those APIs are explained in section 2.4.1.

2.4.1 API definition

There are some general types and definitions which are shared across all APIs. For example, the type returned type by all APIs is `xtasks_stat` (definition shown in listing A.1). All APIs return the value `XTASKS_SUCCESS` if they successfully realized the operation, otherwise they return `XTASKS_ERROR` or a more concrete error status.

Library Initialization

There is the `xTasksInit` API to initialize the library, which must be called before any other API call; and the `xTasksFini` API to cleanup the library. Their declarations are shown in listing 2.8.

```
xtasks_stat xtasksInit();
xtasks_stat xtasksFini();
```

Listing 2.8: xTasks APIs to initialize/finalize the library

Accelerators Information

```
typedef uint32_t xtasks_acc_id;
typedef uint64_t xtasks_acc_type;
typedef const char *xtasks_acc_desc;

typedef struct {
    xtasks_acc_id id;            ///< Accelerator identifier
    float freq;                  ///< Accelerator frequency (in MHz)
    xtasks_acc_type type;        ///< Accelerator type identifier
    xtasks_acc_desc description; ///< Accelerator description
} xtasks_acc_info;

xtasks_stat xtasksGetNumAccs(size_t *count);

xtasks_stat xtasksGetAccs(
  size_t const maxCount, xtasks_acc_handle *array, size_t *count);

xtasks_stat xtasksGetAccInfo(
  xtasks_acc_handle const handle, xtasks_acc_info *info);
```

Listing 2.9: xTasks APIs to retrieve the accelerators information

The `xtasksGetNumAccs` retrieves the number of FPGA task accelerators in the currently loaded design. The API declaration is shown in listing 2.9, and its parameters are:

- `count`. Pointer to a valid `size_t` variable that will be set with the number of accelerators.

The `xtasksGetAccs` retrieves the accelerator handles for each accelerator in the FPGA design. The API declaration is shown in listing 2.9, and its parameters are:

- `maxCount`. Maximum number of elements that can be set in `array`.
- `array`. Pointer to a valid array with at least `maxCount` elements of `xtasks_acc_handle` opaque type. The first elements are set with the accelerator handles.
- `count`. Pointer to a valid `size_t` variable that will be set with the number of handles set in `array` argument.

The `xtasksGetAccInfo` retrieves the information related to an accelerator. The API declaration is shown in listing 2.9, and its parameters are:

- `handle`. Accelerator handle which information will be retrieved.
- `info`. Pointer to a valid `xtasks_acc_info` that will be set with the accelerator information.

Task management

```
#define XTASKS_ARG_FLAG_COPY_IN  0x10
#define XTASKS_ARG_FLAG_COPY_OUT 0x20

typedef void *xtasks_task_handle;
typedef uint64_t xtasks_task_id;
typedef uint64_t xtasks_arg_val;
typedef uint32_t xtasks_arg_id;
typedef uint8_t xtasks_arg_flags;
typedef enum {
  XTASKS_COMPUTE_DISABLE = 0,
  XTASKS_COMPUTE_ENABLE = 1
} xtasks_comp_flags;

xtasks_stat xtasksCreateTask(
  xtasks_task_id const id, xtasks_acc_handle const accel,
  xtasks_comp_flags const compute, xtasks_task_handle *handle);

xtasks_stat xtasksAddArg(
  xtasks_arg_id const id, xtasks_arg_flags const flags,
  xtasks_arg_val const value, xtasks_task_handle const handle);

xtasks_stat xtasksSubmitTask(xtasks_task_handle const handle);

xtasks_stat xtasksTryGetFinishedTaskAccel(
  xtasks_acc_handle const accel, xtasks_task_handle *handle,
  xtasks_task_id *id);

xtasks_stat xtasksDeleteTask(xtasks_task_handle *handle);
```

Listing 2.10: xTasks APIs for FPGA tasks management

The `xtasksCreateTask` creates a task for an accelerator with the given identifier and compute flags. The API declaration is shown in listing 2.10, and its parameters are:

- `id`. Task identifier that will be returned at finalization. An arbitrary identifier that caller can use to uniquely identify a task.
- `accel`. Accelerator handle where task will be submitted.
- `compute`. Compute flags to enable/disable the execution of the task body.

- `handle`. Pointer to a valid `xtasks_task_handle` that will be set with an opaque task handle.

The `xtasksAddArg` adds an argument to an existing task. The API declaration is shown in listing 2.10, and its parameters are:

- `id`. Argument identifier. The arguments are identified by its ordinal position in the parameters.
- `flags`. Argument flags to enable/disable the accelerator wrapper copies.
- `value`. Argument value which may be a memory pointer or an scalar variable.
- `handle`. Task handle returned by `xtasksCreateTask`.

The `xtasksSubmitTask` submits the task for the accelerator into the ready queue. The API declaration is shown in listing 2.10, and its parameter is:

- `handle`. Task handle returned by `xtasksCreateTask`.

The `xtasksTryGetFinishedTaskAccel` tries to retrieve a finished task from the finished queue of the given accelerator. The API declaration is shown in listing 2.10, and its parameters are:

- `accel`. Accelerator handle to retrieve the task from.
- `handle`. Pointer to a valid `xtasks_task_handle` that will be set with an opaque task handle.
- `id`. Pointer to a valid `xtasks_task_id` that will be set with the task identifier provided in `xtasksCreateTask`.

The `xtasksDeleteTask` cleans the task information and liberates the assigned memory. The API declaration is shown in listing 2.10, and its parameters are:

- `handle`. Pointer to a valid `xtasks_task_handle` returned by `xtasksCreateTask`. It is invalidated after the call.

Instrumentation

There is the `xtasksInitHWIns` API to initialize the instrumentation support in the library, and the `xtasksFiniHWIns` API to cleanup the instrumentation library part. The initialization function must be called before any other instrumentation API call and after `xTasksInit`. Both declarations are shown in listing 2.11. The `xtasksInitHWIns` argument is:

- `nEvents`. Number of events that each instrumentation buffer should be able to hold. There is an independent buffer for each submitted task.

```
typedef uint64_t xtasks_ins_timestamp;
typedef enum {
    XTASKS_EVENT_TYPE_BURST_OPEN = 0,
    XTASKS_EVENT_TYPE_BURST_CLOSE = 1,
    XTASKS_EVENT_TYPE_POINT = 2,
    XTASKS_EVENT_TYPE_INVALID = 0xFFFFFFFF
} xtasks_event_type;
typedef struct {
    uint64_t value;      ///< Event value
    uint64_t timestamp;  ///< Event timestamp
    uint32_t eventId;    ///< Event id
    uint32_t eventType;  ///< Event type (one of xtasks_event_type)
} xtasks_ins_event;

xtasks_stat xtasksInitHWIns(size_t const nEvents);
xtasks_stat xtasksFiniHWIns();

xtasks_stat xtasksGetAccCurrentTime(
  xtasks_acc_handle const handle, xtasks_ins_timestamp *timestamp);

xtasks_stat xtasksGetInstrumentData(
  xtasks_task_handle const handle, xtasks_ins_event *events,
  size_t maxCount);
```

Listing 2.11: xTasks APIs for accelerators instrumentation

The `xtasksGetAccCurrentTime` returns the current time of an accelerator. This API is useful to synchronize the host and accelerator times. The API declaration is shown in listing 2.11, and its parameters are:

- `handle`. Accelerator handle which timestamp will be retrieved.
- `timestamp`. Pointer to a valid `xtasks_ins_timestamp` that will be set with the accelerator timestamp.

The `xtasksGetInstrumentData` retrieves the instrumentation events generated by a task. The API declaration is shown in listing 2.11, and its parameters are:

- `handle`. Task handle which events will be retrieved.
- `events`. Pointer to a valid array with at least `maxCount` elements of `xtasks_ins_event` type. The first elements are set with the task events. The next event after last valid event has the `XTASKS_EVENT_TYPE_INVALID` type.

Memory management

```
1  typedef void *xtasks_mem_handle;
2  typedef enum {
3      XTASKS_HOST_TO_ACC,       ///< From host memory to accelerator memory
4      XTASKS_ACC_TO_HOST        ///< From accelerator memory to host memory
5  } xtasks_memcpy_kind;
6
7  xtasks_stat xtasksMalloc(size_t len, xtasks_mem_handle * handle);
8
9  xtasks_stat xtasksFree(xtasks_mem_handle handle);
10
11 xtasks_stat xtasksGetAccAddress
12   xtasks_mem_handle const handle, xtasks_arg_val * addr);
13
14 xtasks_stat xtasksMemcpy(
15   xtasks_mem_handle const handle, size_t offset, size_t len, void *usr,
16   xtasks_memcpy_kind const kind);
```

Listing 2.12: xTasks APIs for FPGA memory management

The `xtasksMalloc` allocate a memory region in the FPGA address space that is accessible by the accelerators. The API declaration is shown in listing 2.12, and its parameters are:

- `len`. Size in bytes of the region to allocate.
- `handle`. Pointer to a valid `xtasks_mem_handle` that will be set with an opaque memory region handle.

The `xtasksFree` liberates a previously allocated memory region. The API declaration is shown in listing 2.12, and its parameter is:

- `handle`. Memory region handle returned by `xtasksMalloc`.

The `xtasksGetAccAddress` returns the memory address that can be sent to accelerators. This address is only valid in the FPGA address space and allows the accelerators to access the memory region. The API declaration is shown in listing 2.12, and its parameters are:

- `handle`. Memory region handle returned by `xtasksMalloc`.
- `addr`. Pointer to a valid `xtasks_arg_val` that will be set with the address.

The `xtasksMemcpy` copies data between the user address space and FPGA address space. The API declaration is shown in listing 2.12, and its parameters are:

- `handle`. Memory region handle returned by `xtasksMalloc`.
- `offset`. Bytes to skip at the beginning of FPGA memory region. Starts to write/read after those bytes.
- `len`. Bytes of data to copy between regions.
- `usr`. Memory pointer to user space data.
- `kind`. One of `xtasks_memcpy_kind` that define the copy directionality (to/from FPGA).

2.4.2 Communication Queues

There are two communication queues to offloaded and retrieve tasks to/from the FPGA task accelerators.

Ready Queue

The Ready Queue is intended to hold a pointer to tasks sent by the host runtime to the FPGA task accelerators. The format of elements in that queue is shown in figure 2.2. The entry information is filled with task information provided in the different API calls. The arguments bitmask is always fixed to `0xFFFF` in the current implementation, and it was intended for future features. The task size is the number of words (64 bits) pointed by the task pointer.

The queue is composed of 1024 entries of 128 bits, which are divided into 32 sub-queues (one for each accelerator) of 32 entries. Each sub-queue is managed as a circular buffer with a single-producer (xTasks library) single-consumer (`Ready Task Manager`).

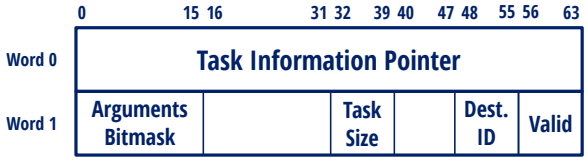


Figure 2.2: Format of elements in the ready queue

The data pointed by the task information pointer has the format shown in figure 2.3. The header words include the task identifier, instrumentation timer, instrumentation buffer addresses, compute flags, and destination ID. The instrumentation words are only set if instrumentation support is initialized. Otherwise, they contain zeros. The blue part is repeated for each task argument with the argument flags, identifier, and value.

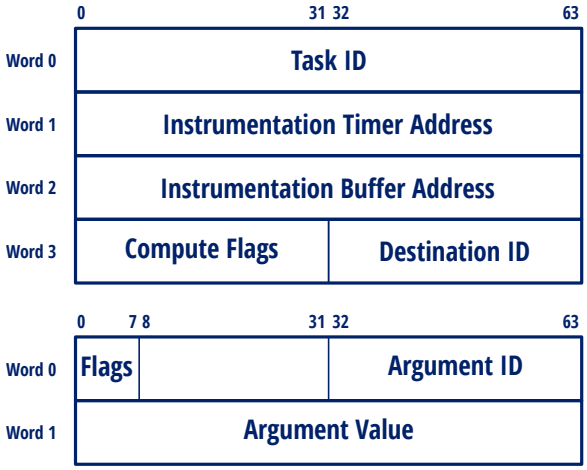


Figure 2.3: Format of ready task information

Finished Queue

The Finished Queue is intended to hold the identifiers of tasks which execution have finished. The format of finished tasks is shown in figure 2.4. The entries contain the task id sent in the ready task information and the accelerator identifier where the task has been executed.

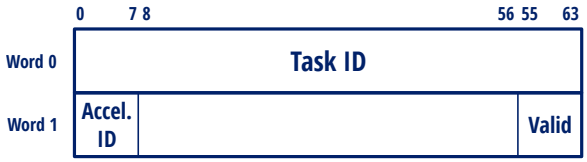


Figure 2.4: Format of elements in the finished queue

The queue is composed of 1024 entries of 128 bits divided into 32 sub-queues of 32 (one for each accelerator) entries. Each sub-queue is managed as a circular buffer with a single-producer (`Finished Task Manager`) single-consumer (xTasks library).

2.5 xdma Library

The xdma library is in charge of the low-level communication between the host and the FPGA device. It hides the underlying board communication protocol, which may be PCI transfers or memory mappings of some FPGA memory regions. On some platforms, the library uses a custom Linux kernel module that performs the operations which require elevated privileges. The main purpose is to read/write the communication queues available in the Task Manager from the xTasks library and read/write the FPGA address space. Besides, the xdma library has other APIs for stream communication, which may be an alternative to Task Manager for the task offloading. However, the stream communication showed a lower performance than Task Manager in previous works [28].

2.6 FPGA Design

The FPGA design is built by AIT tool. It includes all components needed to handle the application execution: the Task Manager, the FPGA task accelerators, and different adapters and interconnects. The following sections explain the main elements, protocols, and interconnects placed in the FPGA bitstream.

2.6.1 Task Manager

The Task Manager is the IP block in charge of managing the tasks offloaded to the FPGA task accelerators by the host. It interacts with the communication queues described in section 2.4.2, and with all FPGA task accelerators. The communication between the Task Manager and the FPGA task accelerators is done over two AXI-Stream interfaces, one for each direction. Figure 2.5 contains the main elements of the FPGA bitstream design for an example with two FPGA task accelerators.

The different IP blocks that compose the Task Manager are explained in the following points.

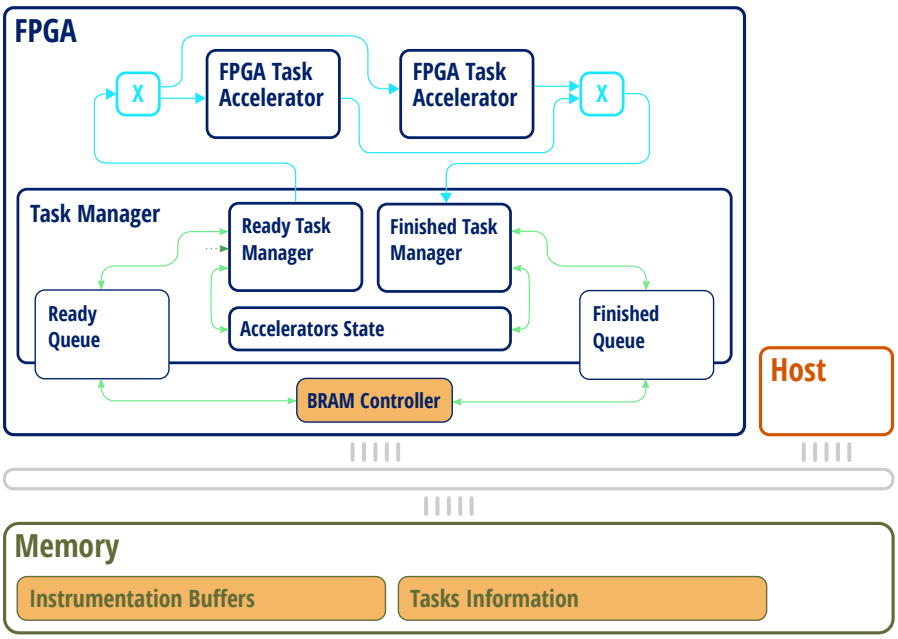


Figure 2.5: FPGA Bitstream design with the Task Manager

Ready Task Manager

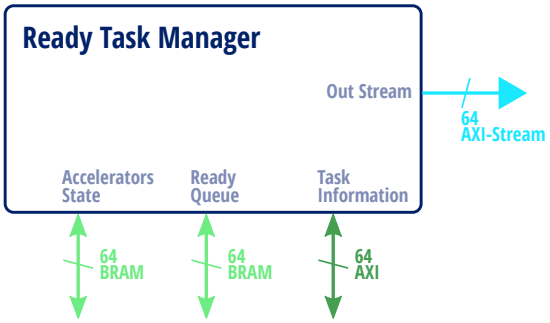


Figure 2.6: External interface of Ready Task Manager

The `Ready Task Manager` is an IP block developed in C++ using HLS tools. The module reads the tasks from the `Ready Queue` and forwards them to the different FPGA task accelerators. The module is connected as shown in figure 2.5. Its external interface is detailed in figure 2.6, where the different ports and protocols to communicate the module with the other components are shown. The ports are:

- `Accelerators State`. Block Random Access Memory (BRAM) port to access the `Accelerators State` memory, which contains the state of each FPGA task accelerator. This memory is read to check if the FPGA task accelerators can receive a new task, and it is written to set the busy state.

- `Ready Queue`. BRAM port to access the `Ready Queue`, which stores the task pointers sent by the host runtime for each FPGA task accelerator. This memory is mainly read but also written to invalidate entries.
- `Out Stream`. AXI-Stream port used to forward the ready task information to all FPGA task accelerators.

Finished Task Manager

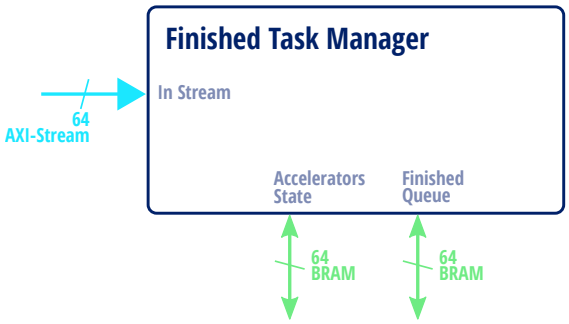


Figure 2.7: External interface of Finished Task Manager

The `Finished Task Manager` is an IP block developed in C++ using HLS tools. The module is connected as shown in figure 2.5. Its external interface is detailed in figure 2.7, where the different ports and protocols to communicate the module with the other components are shown. The ports are:

- `In Stream`. AXI-Stream port used by all FPGA task accelerators to send finished task identifiers.
- `Accelerators State`. BRAM port to access the `Accelerators State` memory, which contains the state of each FPGA task accelerator. This memory is written to set the FPGA task accelerator state to available after the task execution.
- `Finished Queue`. BRAM port to access the `Finished Queue`, which is used to send the finished tasks to the host runtime.

2.6.2 FPGA Task Accelerators

The FPGA task accelerators are IP blocks created by AIT for each C++ HLS source code that Mercurium generated. All FPGA task accelerators have two 64 bits AXI streams connected to the Task Manager (sky-blue paths in figure 2.8). These streams are used to receive ready tasks and send the task identifier of finished tasks. They also have an AXI

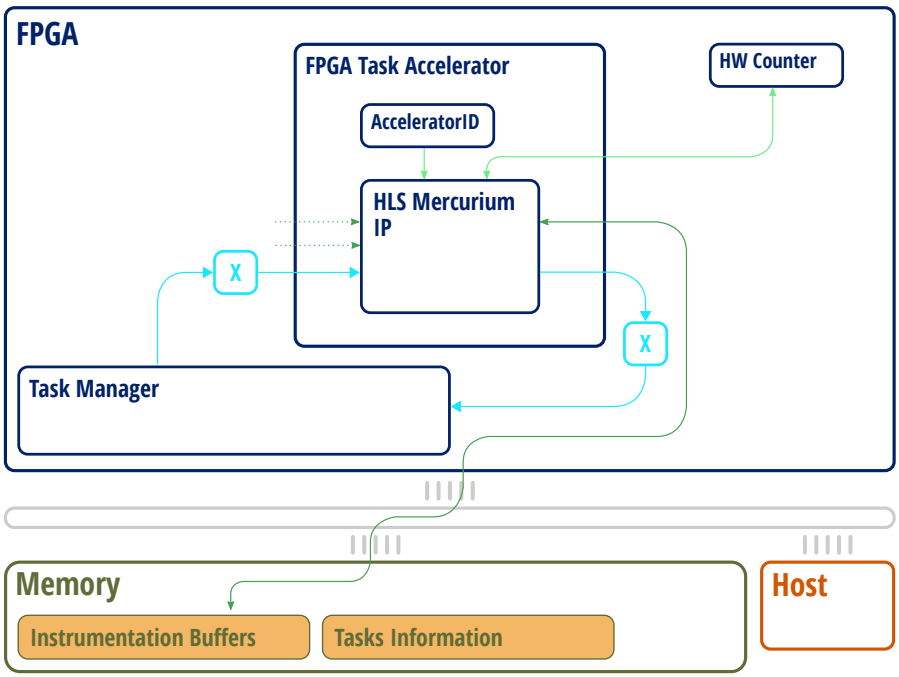


Figure 2.8: Internal structure of FPGA Task Accelerator

port for each task parameter that allows reading/writing the task data (doted dark-green arrows in figure 2.8). Moreover, they may have an additional AXI port to access the instrumentation buffer in memory and a BRAM port to read the `HW Counter`, which is a 64 bits memory that increments every clock cycle and is used as instrumentation event timestamp. All interconnections are shown in figure 2.8.

2.6.3 Interface Protocols

The FPGA design uses different protocols to communicate the elements inside the Task Manager, and communicate the FPGA task accelerators with the Task Manager. These protocols are briefly explained in the following points.

Handshake Protocol

The handshake protocol is a point to point communication within two components that synchronously wish to exchange data. The interface signals are summarized in table 2.1.

The TVALID and TREADY handshake determines when information is passed across the interface. A two-way flow control mechanism enables both the master and slave

Signal	Source	Description
TVALID	Master	Indicates that master is providing a valid package in TDATA.
TREADY	Slave	Indicates that slave can accept the package in the current cycle.
TDATA	Master	Bus that contains the package data that is being sent. The data width is an integer multiple of 8 (byte size).

Table 2.1: Signals of Handshake interface

to control the rate at which the data and control information is transmitted across the interface. For a transfer to occur, both the TVALID and TREADY signals must be asserted. Either TVALID or TREADY can be asserted first, or both can be asserted in the same ACLK cycle. A master is not permitted to wait until TREADY is asserted before asserting TVALID. Once TVALID is asserted, it must remain asserted until the handshake occurs. A slave is permitted to wait for TVALID to be asserted before asserting the corresponding TREADY. If a slave asserts TREADY, it is permitted to un-assert TREADY before TVALID is asserted. [@34]

Figure 2.9 shows a waveform of an example where a data block is transmitted.

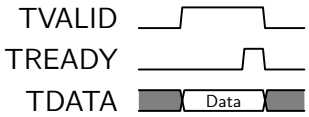


Figure 2.9: Handshake protocol example waveform

AXI-Stream Protocol

The AXI4-Stream protocol is used as a standard interface to connect components that wish to exchange data. The interface can be used to connect a single master, which generates data, to a single slave, which receives data. The protocol can also be used when connecting larger numbers of master and slave components. The protocol supports multiple data streams using the same set of shared wires, constructing a generic interconnect that can perform upsizing, downsizing, and routing operations. [@34]

The interface signals used in the OmpSs@FPGA designs are summarized in table 2.2. The protocol defines more signals which are not used, and their information can be found in [@34]. A package is defined as the set of interface signals and the data itself sent together from the master to the slave. A transaction is defined as a set of packages sent from the master to the slave which last package has the TLAST signal set to high.

The TVALID and TREADY handshake determines when information is passed across the interface. A two-way flow control mechanism enables both the master and slave

Signal	Source	Description
ACLK	Clock source	Global clock to sample other signals (on rising edge).
TVALID	Master	Indicates that master is providing a valid package in other signals (TDATA, TID, TDES).
TREADY	Slave	Indicates that slave can accept the package in the current cycle.
TDATA	Master	Bus that contains the package data that is being sent. The data width is an integer multiple of 8 (byte size).
TLAST	Master	Indicates that the package is the last one of the transfer.
TID	Master	Source identifier of the package.
TDEST	Master	Destination identifier for the package used for package routing.

Table 2.2: Signals of AXI-Stream interface

to control the rate at which the data and control information is transmitted across the interface. For a transfer to occur, both the TVALID and TREADY signals must be asserted. Either TVALID or TREADY can be asserted first, or both can be asserted in the same ACLK cycle. A master is not permitted to wait until TREADY is asserted before asserting TVALID. Once TVALID is asserted it must remain asserted until the handshake occurs. A slave is permitted to wait for TVALID to be asserted before asserting the corresponding TREADY. If a slave asserts TREADY, it is permitted to un-assert TREADY before TVALID is asserted. [@34]

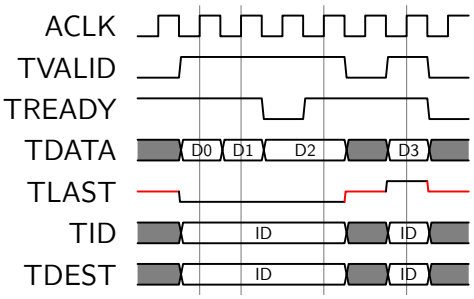


Figure 2.10: AXI-Stream protocol example waveform

Figure 2.10 shows a waveform of an example transaction with four packages that contain D0, D1, D2, and D3. The figure shows that TDATA, TLAST, TID, and TDEST signals are undefined when TVALID is low. It also shows that packages can be transmitted in consecutive cycles (packages 1 and 2) or may take longer if TVALID or TREADY signals are not high during the ACLK rising edge.

BRAM Protocol

The BRAM protocol is a point to point interface used to access the Block Memories (BRAMs). It supports read and write operations from master to slave (BRAM). The protocol requires the slave to resolve the requests from the master in a fixed latency. The interface signals are summarized in table 2.3.

Signal	Source	Description
ACLK	Clock source	Global clock to sample other signals (on rising edge).
EN	Master	Indicates that master is requesting a slave action.
WE	Master	Indicates the bytes of DIN that slave must write. The signal width is the data width divided by 8.
ADDR	Master	Address to read/write.
DIN	Master	Bus with data sent from master to slave.
DOUT	Slave	Bus with data sent from slave to master.

Table 2.3: Signals of BRAM interface

Figure 2.11 shows a waveform of an example where a read request and a write request are performed. The protocol latency of the example is one cycle. The example considers a data width of one byte. Therefore the WE signal is 1 bit wide. The first request starts at the first vertical line, and it is a read request for address `0x01`. The request response data is available in the DOUT signal at next clock cycle, which is the second vertical line. The second request is a write request, the WE signal is not zero, to address `0x02` (ADDR) of data `0xFF` (DIN). The write is assumed to be finished in the next clock cycle, which is the fourth vertical line. The slave also writes the data available in ADDR, like a read request.

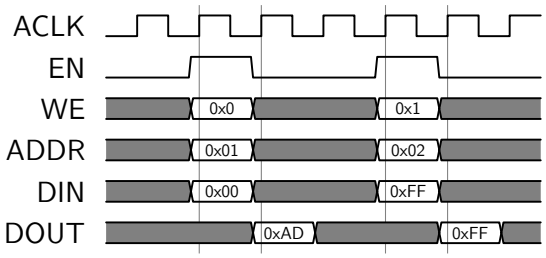


Figure 2.11: BRAM protocol example waveform

AXI Protocol

The Advanced eXtensible Interface (AXI) is a master-slave communication interface designed for high-bandwidth and low-latency data access. The key protocol features are: separate channels for address/control and data phases, support for unaligned transfers,

burst-based, and support for out-of-order transactions [@35]. The protocol is mainly used to access the memory data in the FPGA modules. The channels defined by the protocol are summarized in table 2.4.

Channel	Source	Description
Read address	Master	Address and control information of read transaction.
Read data	Slave	Data and response information to read transactions.
Write address	Master	Address and control information of write transaction.
Write data	Master	Data and strobe signal of the write transaction.
Write response	Slave	Response to the write transaction.

Table 2.4: Channels of AXI interface

Figure 2.12 shows a waveform of the AXI channels during two AXI transactions. The fist transaction (first vertical line) is a read of four data blocks starting at address `0x00`. The second transaction (second vertical line) is a write transaction of four data blocks starting at address `0x40`.

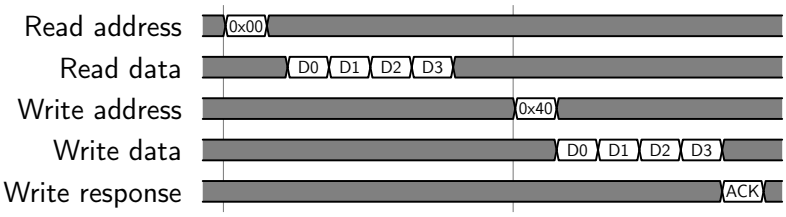


Figure 2.12: AXI protocol example waveform

2.7 Related Work

Several works exist about enhancing and improving parallel programming models. They are over different models and working at different levels or with different approaches. OmpSs is one of those, which is under constant development as several people use it as a baseline to develop different prototypes or extend its capabilities.

There are efforts similar to OmpSs@FPGA ecosystem that also try to simplify the usage of co-processors. The Vineyard project [@36] aims at facilitating heterogeneous programming, based on OpenSPL [@37], OpenCL [@38] and SDSoC [@39]. The Ecoscale project [40] targets applications written in MPI [@41] and OpenCL, to synthesize the OpenCL kernels for the FPGAs, and support distributed and heterogeneous computing. The LegUp high-level synthesis software [42] generates FPGA designs from C/C++ codes, which may also include pthreads or OpenMP parallelism [43]. All these works do

not consider extending the co-processors' capabilities, they try to facilitate their usage as slaves.

There are several related works that also deal with runtime overheads reduction. TurboBLYSK [44] is a framework that implements the OpenMP 4.0 with a custom compiler and a highly efficient runtime scheduler of tasks with explicit data-dependence annotations (requires extra information from application programmers). DAGuE framework [45] offers an architecture aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. However, it auto-parallelizes the application based on a static analysis of the application at compile-time, instead of dynamically building a task dependence graph at runtime. TaPaSCo [46] is a framework that uses a hardware/software-co-design to enable a high launch rate of FPGA-based accelerators. This co-design is similar to our runtimes cooperation, but the support is application transparent without explicitly calling the framework APIs in our approach. HPX (High Performance ParalleX) [47] and STAPL (Standard Template Adaptive Parallel Library) [48] are general purpose frameworks for parallel and distributed applications of any scale. Both use the same asynchronous philosophy of the thesis proposal for task-based systems. However, they tie the application implementation to the framework. Also, there are OpenMP implementations over HPX but with a limited heterogeneity support: hpxMP [49] and OMPX [50].

Other works propose using a hardware manager to implement some capabilities of software runtime to reduce the overheads. Nexus [51], Nexus# [52], Picos [53] are examples of dependence manager for task-based programming models. They focus on fine-grain tasks that need a smart runtime management to obtain good application performance. However, the proposed designs are host-centric as the managers are designed as slaves to accelerate parts of the host runtime.

There are also previous efforts that try to extend the capabilities of accelerators in heterogeneous systems. The most extended work is the CUDA Dynamic Parallelism [@54] introduced by Nvidia in their GPUs to support the nested execution of CUDA kernels. Vesely et al. [55] discuss the support of operating system calls in GPGPUs. In addition, Chen et al. [56] propose to use the accelerators as a host and the regular processors as accelerators for general purpose work offloading. These works propose extending some capabilities of accelerators (GPUs and Intel Many Integrated Core) to allow a more flexible programming. In contrast, this thesis proposes to add all these capabilities for FPGAs with some extra extensions (like dual-side offloading) to obtain even more functionality.

Other related efforts try to reduce the management overheads of the FPGA devices and increase their programmability. Tan et al. [16] present a HardWare (HW) manager that

supports task dependencies resolution and heterogeneous task scheduling for any parallel task-based programming model. However, it does not allow interaction of the FPGA task accelerators with the HW manager beyond the task offloading for their execution. In a similar way to what OmpSs@FPGA does, Cabrera et al. [57] and Sommer et al. [58] propose extensions in OpenMP to support the definition of a task that targets an FPGA device. In contrast to the thesis objectives, none of them consider creating more tasks within an FPGA task.

Related to the recurrent systems, Serrano et al. [59] analyze the usage of OpenMP to develop critical real-time systems. They analyze the timing constraints, which are not considered in this thesis but are fully compatible. They briefly introduce an event clause that is used to define when a recurrent task must start executing. Besides, Pop et al. [60] propose an OpenMP extension to define persistent tasks that work in a stream fashion. Those tasks are similar to the recurrent tasks proposed in the thesis but they are activated by the availability of input data instead of by a timer.

3 Proposal for Asynchronous, Concurrent and Parameterizable Task-Based Systems

The proposal objective is to demonstrate that the efficiency of task management in task-based systems can be improved by asynchronous, concurrent, and flexible handling. That kind of task management is the first step to break the host-centric systems because it uses a distributed behavior that does not rely on dedicated elements.

As a demonstration, this chapter proposes a full-stack design using asynchronous runtime operations. It goes from the programming model to the underlying libraries and tools. Besides, the design is based on the parameterization and customization of all elements. This flexibility is the baseline to allow developers to create applications that perfectly fit the heterogeneous architectures. Moreover, the more flexible the model and the tools, the more room for other proposals like the ones in chapters 4 and 5.

Section 3.1 describes the key ideas of the proposal design. Then, sections 3.2, 3.3 and 3.4 describe the extensions implemented to develop the proposal design at programming model level, binaries and bitstreams level, and execution time level. After that, section 3.5 presents the evaluation environment, the benchmarks, and the results. Finally, section 3.6 concludes the chapter with the key contributions, and section 3.7 lists the publications related to this chapter.

3.1 Proposal Design

The knowledge acquired during the design and initial development of DDAST Manager (in *Asynchronous Runtime for Task-Based Dataflow Programming Models* master thesis [32]) has been used to model the desired behavior that a task-based parallel runtime should have when dealing with co-processors. The design is based on the distributed, concurrent management of the co-processors relying on the DDAST Functionality Dispatcher [11] that asynchronously executes different runtime operations.

The first design goal is to allow any thread to interact with any co-processor. This includes the data movements from/to the co-processors address space if needed, the offloading of tasks to them, the management of tracing and instrumentation events generated inside the co-processors, and any other handling operations. With this model, the same thread may be concurrently executing regular SMP tasks and offloading others to co-processors.

The second design goal is to parameterize as many parts as possible to allow the customization and fine-tuning of applications to the underlying architecture and workload needs. This goal involves all levels of design: starting from the programming model, going through the compiler and linker, and the runtime/libraries that support the application execution.

The third design goal is to simplify the development effort. Despite the possibility of customizing the tools' behavior, they should provide reasonable default values suitable for the vast majority of workloads. Moreover, the efforts that can be automatized must be automatically handled by the tools.

3.2 Programming model extensions

This section presents the extensions developed at the programming model level. The extensions have been developed in OmpSs programming model [4] to extend its capabilities and enhance the application programmability to better fit the underlying architecture or hide undesired system requirements.

3.2.1 Automatic type identifier

The compiler has been modified to automatically generate unique type identifiers when needed. These automatic identifiers supply the `onto` clause of the `target` directive, that is no longer mandatory to uniquely identify each FPGA task accelerator type. However, the programmers can still force the desired type identifier providing it to the compiler using the `onto` clause as before.

The identifier type is defined as 64 bits wide, but the current implementation only uses the lower 35 bits. The format is shown in figure 3.1. The first 32 bits are a hash of the source code filename and the task's function name. The bits 32 to 34 are a bitmask that defines the devices that the task has support for. The current implementation only

considers three devices, but it can be easily extended using more bits (35, 36, etc.). Bit 32 is for FPGA architecture, bit 33 for SMP architecture and bit 34 for GPU architecture.

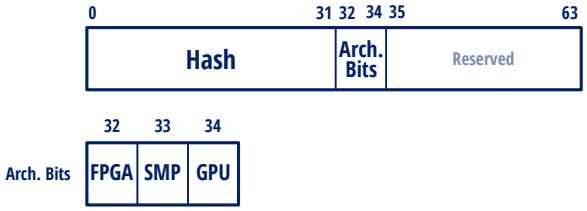


Figure 3.1: Format of task type identifier

3.2.2 Clause for Accelerator Replication

The number of task accelerators in the FPGA design to execute one task type can be modified using a new clause. The clause has been added to the OmpSs target directive. The `num_instances(N)` clause must take only one value (`N`) which must be a positive integer number. If the clause is not provided, only 1 task accelerator is created in the FPGA design. The clause syntax has been added in the Mercurium compiler to allow the FPGA phase to gather it and correctly forward the value to AIT.

```
1  #pragma omp target device(fpga) num_instances(2)
2  #pragma omp task in([1]src) out([1]dst)
3  void foo(float *dst, const float *src) {
4    *dst = *src;
5  }
6
7  int main(...) {
8    for (int i=0; i<10; i++) {
9      foo(dst+i, src+i);
10   }
11   #pragma omp taskwait
12 }
```

Listing 3.1: Example of num_instances(N) clause

An example of the `num_instances(N)` clause is shown in listing 3.1. The example creates two instances of `foo` task accelerator in the FPGA design. Then, the 10 task instances (created every function call from line 9) are concurrently ran in both FPGA task accelerators. This is because the different tasks access different `dst` indexes, so they do not have any data dependence. The different instances of the same FPGA task act like different threads that indistinctly execute tasks in the host.

3.2.3 Clauses for Data Caching in Accelerator HLS Wrapper

The OmpSs programming model promotes the task dependences to task copies by default. Besides, the task copies are cached in the FPGA task accelerator wrapper to accelerate the access. However, this behavior may be undesired due to the increasing amount of resources needed (mainly BRAMs), and performing memory access each time may be better. To allow application programmers explicitly define the desired behavior, three new clauses have been introduced in the target directive:

- `localmem_copies`. This clause requires the compiler to generate a copy of the data defined as task copies inside the FPGA task accelerator wrapper. This is the default behavior.

- `no_localmem_copies`. This clause requires the compiler not generating a copy of the data defined as task copies inside the FPGA task accelerator wrapper.

- `localmem(...)`. This clause takes a list of shaping expressions that define the data that must be copied inside the FPGA task accelerator wrapper. When this clause is used, the task copies do not have a local copy inside the FPGA task accelerator unless explicitly required with the `localmem_copies` clause. The element syntax for this clause is the same one used for the task dependences and copies. However, the region shape must be known at compile time (immediate values and constant variables are allowed). Otherwise, the data region cannot be generated in the FPGA bitstream.

The new clauses untie the copy of task data inside the FPGA task accelerator wrapper from the task copies. Therefore, the application can rely on the OmpSs runtime (Nanos++) to move the task data from the main host memory to FPGA address space memory, but avoid the further copy of those data inside the FPGA task accelerator wrapper if not desired. This approach was difficult to implement previously in the applications. It required an explicit allocation of FPGA device memory and explicit data movements to/from there.

As an example of `localmem(...)` clause usage, the listing 3.2 shows the source code of a `histogram` task that processes 500 floats and maintains an histogram of observed values. Since the number of entries in the histogram is huge (1000000), it does not make sense (or it may not fit into BRAMs) to copy those data inside the FPGA task accelerator wrapper.

```
#pragma omp target device(fpga) localmem([500]input)
#pragma omp task in([500]input) inout([1000000]counters)
void histogram(const float *input, unsigned int *counters) {
  for (unsigned int i = 0; i < 500; i++) {
    unsigned int idx = hash(input[i]);
    counters[idx] += 1;
  }
}
```

Listing 3.2: Example of localmem(...) clause

3.3 Compiler and FPGA design extensions

This section presents the extensions developed at compiler and FPGA design generation levels. The extensions have been developed in the OmpSs compilation tools: Mercurium and AIT. The extensions are designed to hide undesired management stuff, which increases ecosystem utilization and programmability. Also, they extend the current code and design generation capabilities parameterizing the behavior of the tools and allowing the application developers to better fit the underlying architecture.

3.3.1 FPGA design configuration retrievement from bitinfo

The available FPGA task accelerators in the loaded FPGA bitstream must be known during the Nanos++ initialization. This information is used by the runtime to match the types of created tasks with the FPGA task accelerators types and known where to offload the tasks. To gather this information, Nanos++ uses the `xtasksGetAccs` API from xTasks library (which declaration is shown in listing 2.9). In the baseline design, the generation of that information in the xTasks library was done through a `.xtasks.config` file that AIT creates in the design stage. However, this approach requires the usage of `XTASKS_CONFIG_FILE` environment variable or a pre-arranged file name to find the configuration file. The user is also responsible for matching the right configuration file with the loaded bitstream. Otherwise, the execution may hang.

The best approach is to include this information in the FPGA bitstream and expose them somehow to the xTasks library. This way, the system remains coherent, and users do not need to take care of this critical aspect. To this end, AIT has been modified to add a BRAM, accessible from the host through a BRAM controller, with the design information. This memory is called bitinfo, and it is encoded in 32 bits wide words with the format shown in figure 3.2. There are different fields with all relevant information. The fields are separated by a special word which content is fixed to `0xFFFFFFFF`.

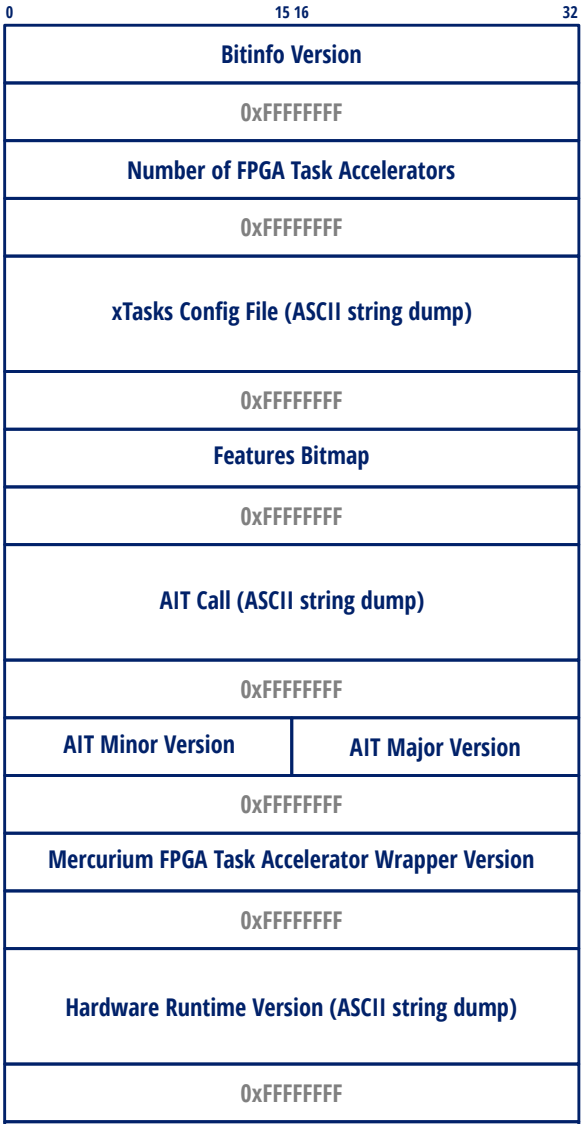


Figure 3.2: Bitinfo structure

The information contained in the bitinfo includes:

- Bitinfo version. Version of the bitinfo layout, which defines the structure of the following words.
- Number of FPGA Task Accelerators. Integer that defines the number of FPGA task accelerators in the FPGA bitstream.
- xTasks Config File. ASCII string dump of the xTasks library configuration file (internal format shown in figure 3.3).
- Features Bitmap.

  - Bit 0. FPGA instrumentation (1 if available, 0 otherwise).
  - Bit 1. Communication with FPGA task accelerators using DMA engines (1 if available, 0 otherwise).
  - Bit 2. Optimization strategy used for interconnects. (1 if maximize performance, 0 if minimize area).
  - Bits 3-4. Accelerator interconnection mode.
  - Bit 6. Hardware runtime (1 if available, 0 otherwise).
  - Bit 7. Extended capabilities of hardware runtime (1 if available, 0 otherwise).
  - Bit 8. SOM Hardware runtime (1 if available, 0 otherwise).
  - Bit 9. POM Hardware runtime (1 if available, 0 otherwise).
- AIT Version. Two integers of 16 bits that define the minor and major versions of AIT.
- Mercurium FPGA Task Accelerator Wrapper Version. Integer that defines the version of wrappers generated for each FPGA task accelerator by Mercurium.
- Hardware Runtime Version. Version of the Hardware runtime in the bitstream (in Xilinx style, "none" if no HardWare Runtime (HWR) instantiated).

The format of xTasks configuration file is detailed in figure 3.3. The words of 32 bits from bitinfo are joined into lines formed by five words (160 bits in total). Each word contains 4 ASCII characters. The first line is the header line, and its contents are always the same (shown in gray). This header line contains the headings of the four information columns. The different columns are separated by the `\t` character, either in the header and in the following rows. Following the header, there are three lines for each FPGA task accelerator type with the information of the four columns (blue part of figure 3.3). The first 152 bits encode the integer that represents the FPGA task accelerator type. The following field is the number of instances of such type codified in 24 bits. Then, there are 248 bits that contain the accelerators' name. Finally, there are 24 bits that encode the accelerators' frequency in MHz.

The xTasks library reads the information in the bitinfo during its initialization to configure its internal structures and provide the information to Nanos++ through the `xtasksGetAccs` API. The access to the bitinfo is done using different mechanisms depending on the board and how it can be interfaced. In the Xilinx Zynq boards, the access is done using a new Linux kernel module developed for that purpose.

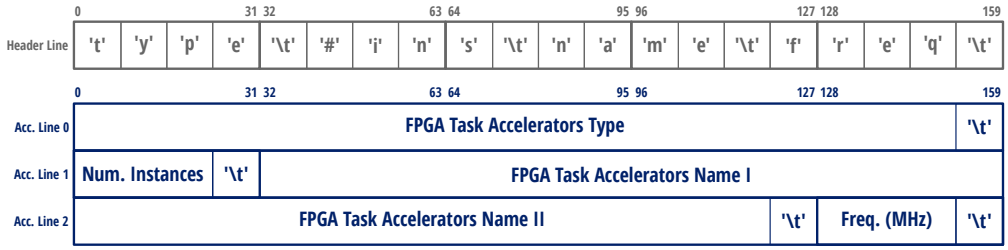


Figure 3.3: xTasks config file structure

The kernel module exposes a set of character devices in the Linux sysfs that provide the information available in the bitinfo at user level. The kernel module obtains the physical memory address to access the BRAM controller from the devicetree. Therefore, the mapping is transparent to the user and several errors (unavailable bitinfo, unsupported bitinfo version) are managed by the kernel module. The devices' structure follows the same bitinfo organization with one device (or more) for each field. For example, the xTasks configuration file can be read with the following command: `cat /dev/ompss_fpga/bitinfo/xtasks`.

3.3.2 Tuning memory interconnections

The interconnection between all data ports of the different FPGA task accelerators (generated by Mercurium) and main memory is balanced among the board's available ports. AIT tries to homogeneously distribute the ports exposed in the FPGA task accelerators to balance the load in each of the board ports.

As an example, figure 3.4 shows an FPGA design with three instances of an FPGA task accelerator that has two data ports. The interconnection between the memory ports of FPGA task accelerators and the board memory ports in the FPGA (assuming four ports and one interconnect for each port) is shown in dark green. In this case, AIT started assigning the data ports from the top FPGA task accelerator to the left interconnect in a round-robin way.

The default mapping results in two instances sharing the board memory ports and one instance using the others in standalone. This causes that the standalone FPGA task accelerator moves data faster than the others. This unbalance between the different FPGA task accelerators may draw the application performance. Therefore, a mechanism to customize the memory interconnection design could improve the application performance.

The AIT capabilities have been extended to support the definition by users of mappings between FPGA task accelerators data ports and the FPGA memory ports. To this end,

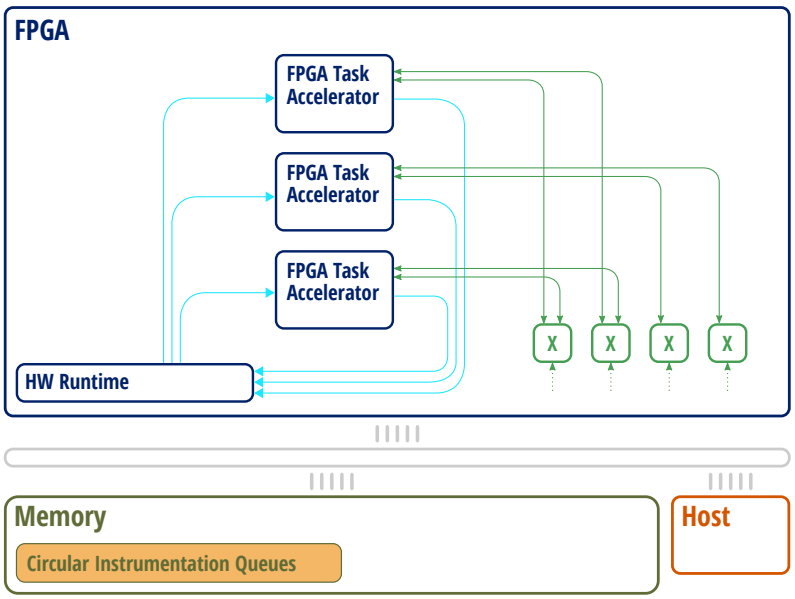


Figure 3.4: FPGA Bitstream design with the default data interconnections

a new flag (`-datainterfaces_map <file>`) that takes the path of a mappings file have been added. The file should contain a line for each data port to connect with a specific memory interface instead of one chosen by AIT. The tool checks the file mappings during the generation of FPGA task accelerators interconnections. Then, the user-defined mapping is used if the data port description is found. Otherwise, AIT selects one interconnect based on the current utilization.

3.3.3 Shared wide Memory Port

The default interconnection of an FPGA task accelerator to the main memory uses a dedicated memory port for each argument. It is needed due to some HLS limitations that do not allow sharing a memory port to access different data types. Moreover, having a dedicated port for each argument could allow concurrent data movements from main memory to BRAMs or vice-versa. All the ports in the HLS source code are created by Mercurium during the FPGA phase and wired in the FPGA design by AIT during the bitstream generation.

The data width of each memory port is the same as the data type width. The AXI protocol, used in the memory ports, only reads/writes one element in each memory access. However, the width of the physical memory port to the DDR module is usually wider. Therefore, the default interconnection is overloading the memory due to the generation of small accesses instead of wide ones. Besides, the application performance could be improved thanks to the faster data movements.

```
void histogram(const float *input, unsigned int *counters);

void histogram_mcxx_hls_wrapper(
  hls_axis_t mcxx_inStream, hls_axis_t mcxx_outStream,
  float *input_port, unsigned int *counters_port)
{
  float input[500];
  unsigned long long int _params[2];

  //Read arguments from mcxx_inStream into _params
  //...
  memcpy(input, input_port + _params[0], 500*sizeof(float));

  histogram(input, counters_port + _params[1]/sizeof(unsigned int));

  //Sync the task execution in mcxx_outStream
  //...
}
```

Listing 3.3: FPGA task accelerator wrapper example with original memory ports

As an example, listing 3.3 shows a simplified version of the wrapper that Mercurium generated around the task function (same of listing 3.2) in the HLS source code. The wrapper has the two synchronization streams (`mcxx_inStream` and `mcxx_outStream`) and two memory ports (`input_port` and `counters_port`), one memory port for each argument. After reading the function parameters from the input stream, the wrapper reads the elements of `input` array into a local wrapper variable. Those local copies are used to call the `histogram` function. In contrast, the `counters` parameter is not copied, and the `histogram` function gets a reference to the memory region through the dedicated memory port.

```
void histogram(const float *input, unsigned int *counters);

void histogram_mcxx_hls_wrapper(
  hls_axis_t mcxx_inStream, hls_axis_t mcxx_outStream,
  ap_uint<128> *wrapper_mem_port, unsigned int *counters_port)
{
  float input[500];
  unsigned long long int _params[2];

  //Read arguments from mcxx_inStream into _params
  //...
  unsigned int _n_lines = 500*sizeof(float)/sizeof(ap_uint<128>);
  unsigned int _n_elems_line = sizeof(ap_uint<128>)/sizeof(float);
  for (unsigned int _line=0; _line < _n_lines; _line++) {
    unsigned int _off_line = _params[0]/sizeof(ap_uint<128>) + _line;
    ap_uint<128> _tmp_line = wrapper_mem_port[_off_line];
    for (unsigned int _elem=0; _elem < _n_elems_line; _elem++) {
```

```
18          input[_line*_n_elems_line + _elem] = _tmp_line.range(
19            _elem*sizeof(float)*8 + sizeof(float)*8 - 1,
20            _elem*sizeof(float)*8);
21        }
22      }
23
24      histogram(input, counters_port + __params[1]);
25
26      //Sync the task execution in mcxx_outStream
27      //...
28  }
```

Listing 3.4: FPGA task accelerator wrapper example with new shared memory port

A new option in Mercurium compiler changes how the data is read/write in the FPGA task accelerator wrapper. The option is `fpga_memory_port_width`, and it takes a positive integer value that defines the desired bit-width for the shared memory port created in the FPGA wrapper. This option creates a shared memory port with the desired width in the FPGA task accelerator wrapper for all task parameters that explicitly or implicitly appear in the `localmem` clause. The shared memory port is not used for the parameters without a copy inside the wrapper due to the HLS limitations mentioned before. Those limitations do not apply to the parameters in `localmem` clause because the wrapper reads chunks of a fixed size and forwards those bits to the corresponding local variable regardless of their type. The width must be multiple of 8 and all task parameters widths.

Listing 3.4 shows the same example of listing 3.3 but with the new option enabled. The new wrapper version has the memory port `wrapper_mem_port`, which replaces the previous `input_port`, with a `ap_uint<128>` type, which basically is an unsigned integer value of the specified width. The new port is read in line 17, and the retrieved data is stored in the `input` array in line 19.

3.4 Execution model extensions

This section presents the extensions developed at the execution level. The extensions have been developed in the OmpSs runtime environment. The extensions are designed to boost the performance of the applications, reducing the runtime overheads through an asynchronous behavior. Moreover, the extensions remove undesired limitations among different execution tools that prevent performance and limit the other proposal extensions.

3.4.1 Concurrent Offloading to Accelerators

The offloading of OmpSs tasks to the devices (GPU, FPGA, OpenCL, etc.) is managed by a dedicated special thread. The Nanos++ runtime structures and logic were developed with this restriction in mind. Therefore, the runtime was only allowing one helper thread per accelerator, which only was used to send and retrieve tasks to its assigned device. This model has different problems:

- The helper thread may be underutilized when device tasks are a few and long.
- The helper thread may not be fast enough to feed the device when device tasks are a lot and fast.
- The processor may not have enough physical cores to run in parallel all helper threads and regular worker threads. This is not a problem in big HPC nodes with several physical cores, but it is in a SoC boards with a few cores.

The model was replaced by a flexible one, which allows any thread to deal with device tasks. This modification required re-engineering the Nanos++ core because it was designed in a way that one thread only can match one Processing Element (PE). The new design keeps the relation of a thread with one PE to match the compatible WD but allows any thread to change its PE and start dealing with tasks for those. This change allows the concurrent offloading of tasks to the same accelerator from different threads.

The number of helper threads that offload tasks to accelerators could be parameterized after the modifications. Several helper threads share the same PEs to concurrently handle the set of accelerators. Moreover, the avoid the underutilization of the FPGA helper threads, the possibility of executing SMP tasks directly by them has been added using the PE switch capability. The behavior of FPGA helper threads is handling FPGA tasks until there are no actions to perform, spin a time, and if no actions to perform are found, the thread tries to execute one SMP task. After that, the helper thread starts the loop again, searching for FPGA tasks to offload and/or retrieve. This behavior makes the thread a hybrid between an FPGA helper thread and an SMP worker thread, which prioritizes the FPGA actions.

The integration of the DDAST core in the Nanos++ runtime, allows the use of idle SMP worker threads to offload tasks to devices. During the runtime initialization, the architecture plugins may register a callback in the Functionality Dispatcher. Those callbacks may switch the PE of the caller thread to the accelerator one, try to find a suitable WD, and offload the task if any is found. This capability was not possible before due to the strong relation between PEs and threads. It was also not possible due to the

lack of a mechanism to take profit of idle worker threads. This makes worker threads to have an hybrid behaviour between regular worker threads and regular helper threads but always prioritizing SMP tasks.

The list of runtime options added to Nanos++ with the modifications explained in this section is:

- 'NX_FPGA_HELPER_THREADS' and '–fpga-helper-threads' to adjust the number of FPGA helper threads to use in the execution. By default, it is 1.
- 'NX_FPGA_HYBRID_WORKER', '–fpga-hybrid-worker' and '–no-fpga-hybrid-worker' to adjust whether the FPGA helper threads also may run SMP tasks. By default, it is enabled.
- '–fpga-idle-callback' and '–no-fpga-idle-callback' to adjust whether the FPGA callback is registered in the Functionality Dispatcher or not. By default, it is enabled.

3.4.2 Extrae Support for Device Instrumentation

The instrumentation of the applications is crucial for understanding the obtained behavior/performance and improving their implementation. The Nanos++ runtime has different instrumentation plugins that gather different kinds of information from the application and the runtime internals. Moreover, the support for tracing FPGA task accelerator was introduced in [61] and extended in [62]. The FPGA tracing was developed using the OMPT interface, which was implemented in the Extrae library [@63].

The regular Extrae API is not suitable for device instrumentation due to two drawbacks. First, the API assumes that the caller thread has generated the events. This assumption is not true when instrumenting the FPGA task accelerators because they cannot directly interact with the Extrae API. They rely on some host thread who forwards the events to Extrae. Therefore, the system will require one thread per FPGA task accelerator, exclusively dedicated to the forwarding, in order to obtain the desired schematic. The dedicated helper thread per FPGA task accelerator has been removed in section 3.4.1 due to its performance drawbacks. Second, the Extrae API does not receive timestamps for the events but gets the event timestamp during the API call. This is also a problem because the events generated by FPGA task accelerators are forwarded to Extrae after they happen, so the event timestamp must be different to the time when calling the API.

The OMPT specification has a specific part for tracing the devices, the target directive related events. This specification is tied to the OpenMP model and has limited possibilities compared to the flexible Extrae API. For instance, user-defined events are not feasible, and the events are mainly tied to task offloading and execution. However, the tracing of FPGA task accelerators presented in [61] and [62] was based on a preliminary implementation of OMPT specification in Nanos++ and Extrae.

The Paraver [64] traces with FPGA device events generated by the OMPT implementation are shown in figure 3.5 and figure 3.6). Each figure row represents a different computational or communication component of the system. The colored regions along the x-axis represent the different events in that component [61] among time. They were useful to see the events inside the FPGA, but they present some limitations. On the one hand, the implementation required using several trace threads for the same FPGA task accelerator, one for each event type. Instead, using only one trace thread to join all events happening in the same FPGA task accelerator would be more clear. On the other hand, the limited set of OMPT event types limits the amount of information extracted from the executions, in contrast to the wide range of statistics gathered when using the Extrae API directly.

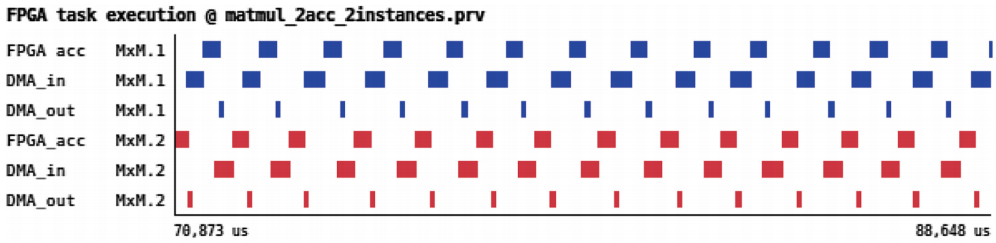


Figure 3.5: OMPT execution trace of Matrix Multiply using two FPGA task accelerators (figure 6.a from [61])

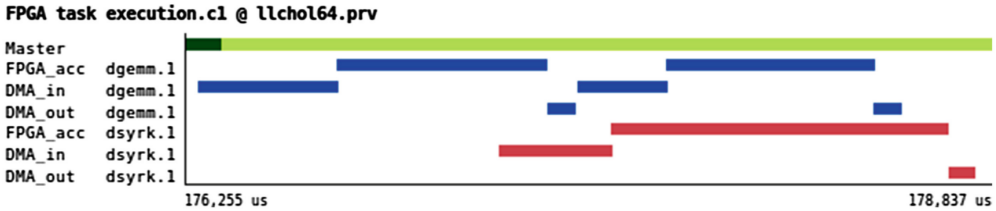


Figure 3.6: OMPT execution trace of Cholesky with overlap of host tasks and `dgemm` and `syrk` FPGA tasks (figure 10 from [61])

An extension of the Extrae API has been designed to extend its capabilities and allow a proper device tracing. Supporting these new APIs in Nanos++ runtime overcomes the drawbacks of previous FPGA instrumentation through OMPT API. The key points of new APIs are introducing the concept of device and allow the external definition of timestamps for device events. For Extrae library, a device is just an extension of the

thread concept with only two extra properties: latency (delta of time between device clock and host clock) and a lock (to coordinate different host threads emitting events for the same device).

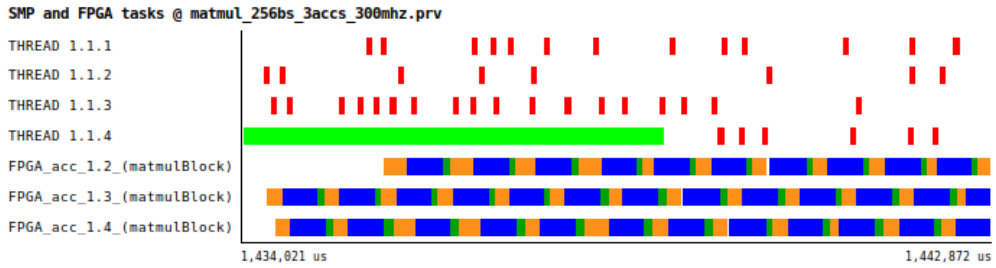


Figure 3.7: Extrae execution trace of Matrix Multiply using three FPGA task accelerators

Figure 3.7 shows how the traces look after the Extrae extension. All events generated in the same FPGA task accelerator are displayed in the same line of the Paraver trace, and their color recognizes the different events. The events emitted in the host threads are synchronized and merged with the events emitted in the FPGA device. In the example, the color meanings are:

- Light-green. Execution of main task.
- Red. Task offload to an FPGA task accelerator.
- Blue. Execution of a task into an FPGA task accelerator.
- Orange. Input data movement between the main memory and the local data memories inside an FPGA task accelerator.
- Dark-green. Output data movement between the local data memories inside an FPGA task accelerator and main memory.

New API definition

This section briefly describes the new Extrae APIs and types.

```
1 typedef unsigned long long extrae_time_t;
```

Listing 3.5: Type definitions for new Extrae APIs

The new `extrae_time_t` type is the type used to specify the timestamp of the events. The value must be in nanoseconds. Therefore the caller must translate the device time representation to nanoseconds if needed.

```
1  void Extrae_register_device(
2    const char *description,
3    extrae_time_t (*get_device_time_fn)(void *),
4    void *get_device_time_arg);
5
6  void Extrae_nevent_device(
7    int device_id, unsigned count,
8    extrae_type_t *types,
9    extrae_value_t *values,
10   extrae_time_t *times);
```

Listing 3.6: Function declarations of new Extrae APIs

The `Extrae_register_device` function registers the `N` device in the trace to allow emitting its events where `N` is the number of previously registered devices and `N` becomes the `device_id` of the new device. From the trace point of view, a device is just an extra thread being traced. During the registration, Extrae computes the delta between the host and device times. This delta will be applied to all timestamps emitted within the device events. The function parameters are:

- `description`. Char array with the device description text that will be shown in the trace files.
- `get_device_time_fn`. Pointer to function that returns the device time in `extrae_time_t` format (nanoseconds) and takes 1 argument (`get_device_time_arg`). It will be used during the registration to compute the delta between the host and device times.
- `get_device_time_arg`. Argument for the `get_device_time_fn` function.

The `Extrae_nevent_device` function emits an array of events defined by the provided types, values, and times. It is analogous to `Extrae_nevent` API but emits the events in the trace thread representing the requested device, instead of the caller thread. The function parameters are:

- `device_id`. Identifier of the device that generated the events. It must be an integer within `0` and `N-1` where `N` is the number of registered devices through `Extrae_register_device` API.
- `count`. Number of events to emit.
- `types`. Array of `extrae_type_t` with `count` elements that contain the event types.

- `values`. Array of `extrae_value_t` with `count` elements that contain the event values.
- `times`. Array of `extrae_time_t` with `count` elements that contain the event timestamps.

3.4.3 Task Manager replacement by Hardware Runtime

The Task Manager is intended to manage the tasks offloaded by the host runtime to the different FPGA task accelerators. Considering the objectives of this thesis, the Task Manager approach becomes a limit for the development as it is tied to the task management concept. Therefore, a new design has been developed to keep in mind the modularity and the possibility of extending the capabilities in the future.

The new design interface has been called Hardware Runtime (HWR), and this interface has been implemented in the Smart OmpSs Manager (SOM). The interface and the implementation have been distinguished into two separate parts to allow different implementations, which may have more or less features making each one more suitable for different scenarios.

The new design is based on commands distinguished between them by an 8 bits command code, which also defines the command length and format. The same format is used in the communication queues and the stream messages sent between the HWR and the FPGA task accelerators. This simplifies the management of IP blocks which only need to generate a stream message for each command word. All commands are divided in words of 64 bits, so the communication queues and the stream messages are always 64 bits wide.

Figure 3.8: Format of command head

All commands share the same structure in the head (first word, 64 bits) as shown in figure 3.8. The head has the lower 8 bits reserved to encode the command code, and the upper 8 bits reserved to encode the validity of the command (only meaningful in the communication queues). The odd command codes are reserved for commands that make the FPGA task accelerators become busy and do not accept further commands until some message is sent back. In contrast, the even command codes do not block the FPGA task accelerator, and other commands can be sent to it immediately after the current one.

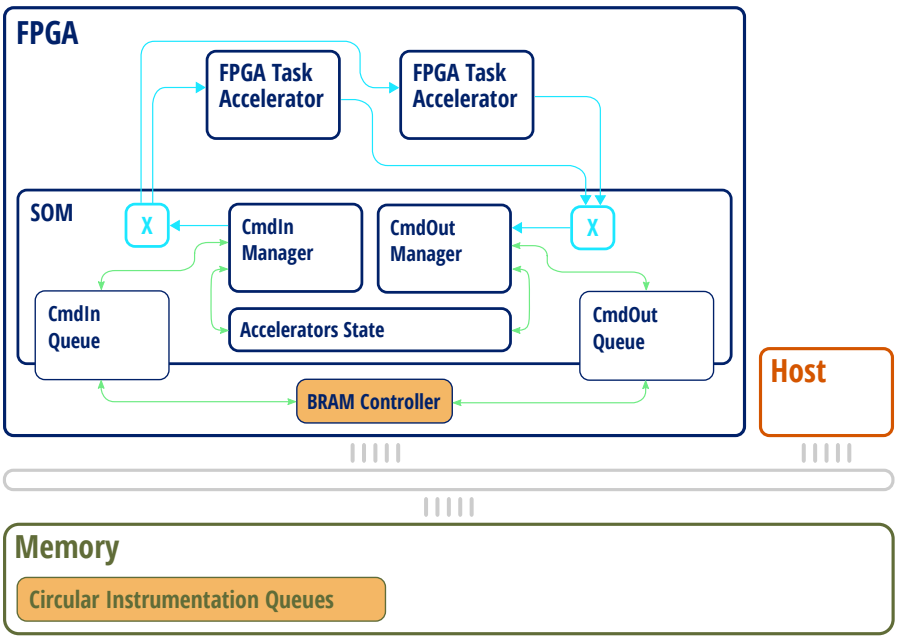


Figure 3.9: FPGA Bitstream design with the SOM Hardware Runtime

Figure 3.9 shows the main elements in the FPGA bitstream design with the Smart OmpSs Manager (SOM) implementation of the new HWR. The design is very similar to the previous one (shown in figure 2.5) as the available features are the same but using a new flexible design. The main differences are the queue and sub-queues lengths that have been adjusted, and the tasks information memory region, which is no longer needed as all task information is encoded in the commands directly.

The following points briefly describe the new queues and IP blocks.

CmdIn Queue

The `CmdIn Queue` is intended to hold the commands sent by the host runtime to the FPGA task accelerators.

The queue is composed of 1024 words of 64 bits, which are divided into 16 sub-queues (the maximum number of FPGA task accelerators supported in an FPGA design) of 64 elements. Each sub-queue is managed as a circular buffer with a single-producer (xTasks library) single-consumer (`CmdIn Manager` in SOM implementation). The elements stored in the queue may not have a fixed length as each command may have a different format with more or less information. The execute task command does not have a fixed length as it depends on the number of task arguments. Therefore, the elements can start at any

word of the queue, and their length is determined by the information in the first word (head).

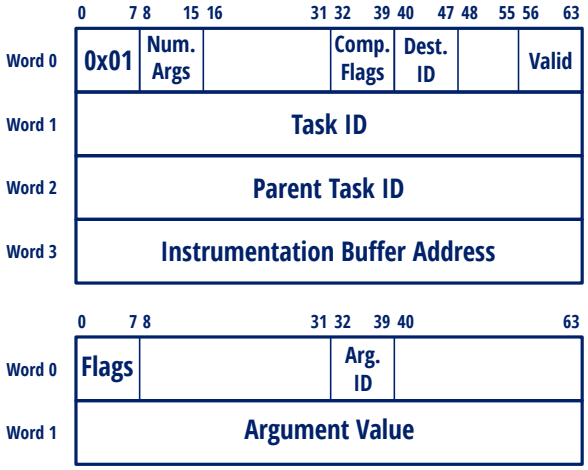


Figure 3.10: Format of execute task command

The format of the execute task command is shown in figure 3.10. This command is sent by xTasks library when the `xtasksSubmitTask` API is called, the command is equivalent to the tasks sent to the Task Manager. The first four words of the command are the command header. Then, there are `nArgs` groups of two words, each one with the information of one argument (in blue). The command code of execute task command is `0x01`, which is written in the bits 0-7 of command head word. The remaining command data is the same as the encoded in the tasks wrote into the `Ready Queue`.

CmdOut Queue

The `CmdOut Queue` is intended to hold the commands sent to the host runtime from the HWR.

The queue is composed of 1024 words of 64 bits, which are divided into 16 sub-queues (the maximum number of FPGA task accelerators supported in an FPGA design) of 64 elements. Each sub-queue is managed as a circular buffer with a single-producer (`CmdOut Manager` in the SOM implementation) single-consumer (xTasks library). The elements stored in the queue may not have a fixed length as each command may have a different format with more or less information. However, all finished execute task commands have a fixed length of two words.

The format of the finished execute task command is shown in figure 3.11. This command is sent by the HWR to the xTasks library when the execution of a task offloaded by the host runtime with an execute task command finishes. The head word of the command

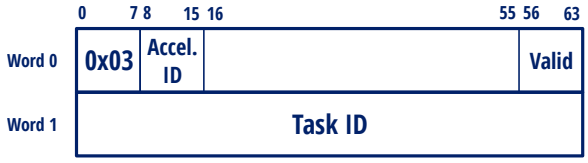


Figure 3.11: Format of finished execute task command

contains the command code, which is 0x03, the accelerator ID that executed the task, and the valid bits. The second word encodes the ID of the task whose execution finished (the identifier is the one provided by xTasks library in the execute task command).

CmdIn Manager

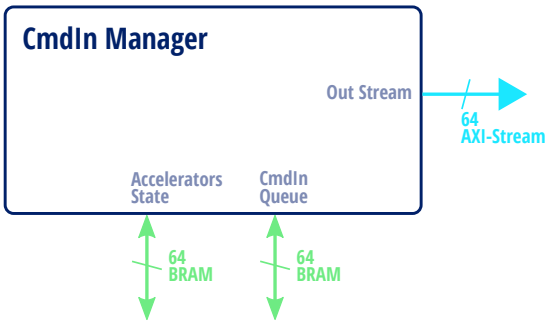


Figure 3.12: External interface of CmdIn Manager

The `CmdIn Manager` is an IP block of SOM implementation. It is developed in C++ using HLS tools. The module is connected as shown in figure 3.9. Its external interface is shown in figure 3.12, where the different ports and protocols to communicate the module with the other components are detailed. The ports are:

- `Accelerators State`. BRAM port to access the `Accelerators State` memory, which stores the state of each FPGA task accelerator. This memory is read to check if the FPGA task accelerators can receive a new command, and it is written to update the state after sending a new command.

- `CmdIn Queue`. BRAM port to access the `CmdIn Queue`, which stores the commands sent by the host runtime for each FPGA task accelerator. This memory is mainly read but also written to invalidate entries.

- `Out Stream`. AXI-Stream port used to forward the commands to all FPGA task accelerators.

CmdOut Manager

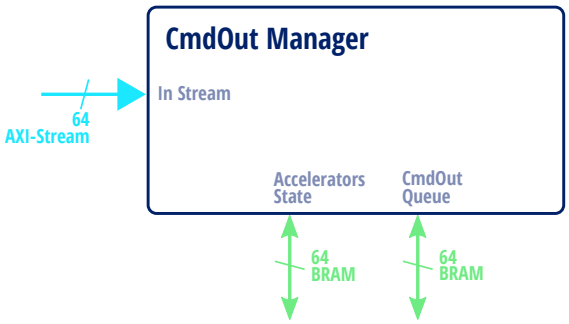


Figure 3.13: External interface of CmdOut Manager

The `CmdOut Manager` is an IP block of SOM implementation. It is developed in C++ using HLS tools. The module is connected as shown in figure 3.9. Its external interface is shown in figure 3.13, where the different ports and protocols to communicate the module with the other components are detailed. The ports are:

- `In Stream`. AXI-Stream port used by all FPGA task accelerators to send finished execute task messages.
- `Accelerators State`. BRAM port to access the `Accelerators State` memory, which stores the state of each FPGA task accelerator. This memory is written to update the FPGA task accelerator state after the command execution.
- `CmdOut Queue`. BRAM port to access the `CmdOut Queue`, which is used to send the finished execute task commands to host runtime.

3.5 Evaluation

This section presents the evaluation results of the proposed design and enhancements. The results are presented for a set of the implementation modifications explained in this chapter.

First, sections 3.5.1 and 3.5.2 present the environments and benchmarks used among the evaluation. Then, sections 3.5.3 and 3.5.4 present the parameters tuning and performance evaluation of DDAST Manager in a new architecture. Together with the previous ones, these results result in a new dynamic parameter auto-tuning, which is architecture aware. The core of DDAST Manager is the basis for developing some of the asynchronous and concurrent operations in the other improvements, and its optimization is critical for the overall runtime performance. Section 3.5.5 presents the results for the concurrent

offloading to accelerators extension described in section 3.4.1. Section 3.5.6 presents the results for the tuning memory interconnections extension described in section 3.3.2. Finally, section 3.5.7 presents the results for the shared wide memory port extension described in section 3.3.3.

3.5.1 Environment

Two different environments have been used in the proposal evaluation. Both are described in the following points.

Power9

The evaluation of DDAST Manager has been extended to a new many-core machine with an IBM Power9 architecture [65]. The nodes used in the evaluation have 2 IBM Power9 8335-GTG processors with 20 cores each. The executions only use 1 thread per core because more than one thread does not benefit the evaluated benchmarks' performance due to the Simultaneous Multi-Threading (SMT) core architecture [66]. The processors work at 3 Ghz of frequency and have 512 GiB of main memory available. The compiler used to compile the applications and the runtimes natively is the GNU C Compiler Collection (GCC) version 8.1.0.

ZCU102

The evaluation of extensions involving FPGA devices have been done in a Xilinx Zynq UltraScale+ MPSoC ZCU102 board [@67], although the different extensions have been used on different boards and in different projects. The system on chip (SoC) is composed of 4 ARM Cortex-A53 cores, which run at 1.1 GHz, a Xilinx ZU9EG FPGA, and a main DDR4 memory of 4 GiB. The board is booted using the Ubuntu Linux 16.04 operating system. The tools used to generate the application bitstreams and binaries are Vivado Design Suite 2019.2, GNU C/C++ Compiler 6.2.0, and PetaLinux Tools 2019.1.

3.5.2 Benchmarks

The used benchmarks are described in the following points. For each one, its execution arguments are explained and provided with the number of created tasks in each configuration and any other remarks that may be valuable for reproducibility. In all of them, some timing instructions are added after the sequential initialization and after the final

global taskwait. The elapsed time between these two points is defined as the execution time. Moreover, the times provided in this evaluation are the average value obtained from different executions.

In some tests, different sets of execution parameters are used to create different task granularities. Besides, the benchmark execution parameters are selected considering the following:

- Problem size. Have a big enough problem size to gather significant results.
- Coarse grain (CG) task size. Smallest task size that has enough parallelism to feed all computation units, delivering almost the best performance, and hiding the runtime overheads.
- Fine grain (FG) task size. Solve the same problem with tasks that use half the coarse-grain value.

Matrix Multiply

The Matrix Multiply (Matmul) benchmark [@68] computes the product of two blocked matrices in parallel. The application takes two main arguments: the matrix dimension (`MSIZE`) and the block dimension (`BSIZE`). Therefore, the matrices with $MSIZE * MSIZE$ elements are divided into sub-matrices with $BSIZE * BSIZE$ elements. Consequently, each task uses three of these sub-matrices to compute the corresponding multiplication. The task kernel implementation is based on OpenBLAS library [@69] when they are executed in the host processor, and a simple C implementation when they are executed in the FPGA. The task dependences follow a regular pattern with several independent chains that group all tasks working with the same output block.

The used values for `MSIZE` and `BSIZE` arguments are summarized in table 3.1.

Environment	Task granularity	Matrix Size	Block Size	Num. Tasks
Power9	Coarse Grain	8 192	512	4 096
	Fine Grain	8 192	256	32 768
ZCU102	Coarse Grain	4 096	256	4 096
	Fine Grain	4 096	128	32 768

Table 3.1: Matrix Multiply execution arguments

In the FPGA device evaluations, the kernel implementation has been done using a simple C implementation where the loops in the kernel task follow the k-i-j order as shown in pseudo-code of listing 3.7. The k-i-j order has a better access pattern to the BRAMs

where the matrices are stored and results in a better performance. The instances number of `matmulBlock` and the initiation interval (II) of the innermost loop are shown in table 3.2 for the task granularities. In these evaluations, the kernel task accelerators always run at 300 MHz. Also, different configurations are considered depending on where tasks can be executed:

- `SMP`. Tasks are only run in the host processor.
- `FPGA`. Tasks are only run in the FPGA task accelerators.
- `SMP+FPGA`. Tasks are both run in the host processor and FPGA task accelerators.

```
1  #pragma omp target device(fpga)
2  #pragma omp task in([BSIZE*BSZIE]a, [BSIZE*BSIZE]b) inout([BSIZE*BSIZE]c)
3  void matmulBlock(const float *a, const float *b, float *c) {
4    for (unsigned int k=0; k<BSIZE; ++k) {
5      for (unsigned int i=0; i<BSIZE; ++i) {
6        for (unsigned int j=0; j<BSIZE; ++j) {
7          #pragma HLS pipeline II=LOOP_II
8          c[i*BSIZE + j] += a[i*BSIZE + k] * b[k*BSIZE + j];
9        }
10     }
11   }
12 }
```

Listing 3.7: Matrix Multiply pseudo-code

Task granularity	Num. kernel accels	Loop II
Coarse Grain	3	2
Fine Grain	3	1

Table 3.2: FPGA configurations for Matrix Multiply benchmark

N-Body

N-Body is a simulation among time of N physical bodies (particles) in a space that attract between them as a result of their mass [@70]. The application takes three arguments: the number of particles, the number of time steps to be simulated, and the number of particles per block (`BSIZE`). Therefore, the particles are spread into blocks with `BSIZE` particles, which are used as task input/output. The tasks follow a regular chained pattern similar to the Matrix Multiply one, but this benchmark has nested tasks.

The task nesting makes more critical some of the requests to the DDAST Manager because they may block the application parallelism until they are processed. The values for the arguments used in the DDAST Manager evaluations are summarized in table 3.3.

Task granularity	Particles	Timesteps	Block Size	Tasks
Coarse Grain	16 384	16	256	65 568
Fine Grain	16 384	16	128	262 176

Table 3.3: N-Body execution arguments

In the FPGA device evaluations, the kernel implementations (`calculate_forces` and `update_positions`) use the same C implementation like in the host. However, they have some HLS directives to control the parallelism created in each FPGA task accelerators to balance the resources-parallelism ratio. The instances number of kernel tasks and the parallelism (`Parallel particles`) of the innermost loop are shown in table 3.4 for the configurations. In this evaluations, the kernel task accelerators always run at 250 MHz. Also, two configurations are considered depending on where tasks can be executed:

- `SMP`. Tasks are only run in the host processor.
- `FPGA`. Tasks are only run in the FPGA task accelerators.

Task granularity	Num. calculate forces accels	Parallel particles	Num. update positions accels
Coarse Grain	3	10	1

Table 3.4: FPGA configuration for N-Body benchmark

Sparse LU

The Sparse LU benchmark [@68] computes the Lower Upper (LU) decomposition of a sparse matrix in parallel. The application takes two arguments: the matrix dimension (`MSIZE`) and the block dimension (`BSIZE`). Therefore, the matrix with `MSIZE*MSIZE` elements is divided into sub-matrices with `BSIZE*BSIZE` elements. The task dependences follow a much more complex and irregular pattern than the Matrix Multiply and N-Body benchmarks.

The used values for `MSIZE` and `BSIZE` arguments in the DDAST Manager evaluations are summarized in table 3.5.

Task granularity	Matrix Size	Block Size	Num. Tasks
Coarse Grain	8 192	128	11 472
Fine Grain	8 192	64	89 504

Table 3.5: Sparse LU execution arguments

Cholesky Factorization

The benchmark implements the cholesky matrix factorization using a blocked algorithm. The application takes two arguments: the matrix dimension (`MSIZE`) and the block dimension (`BSIZE`). Therefore, the matrix with `MSIZE*MSIZE` elements is divided into sub-matrices with `BSIZE*BSIZE` elements. The task dependences have a non-regular pattern like in SparseLU factorization. The task kernel implementation is based on OpenBLAS library when they are executed in the host processor. The values for the arguments used in the evaluations are summarized in table 3.6.

Task granularity	Matrix Size	Block Size	Num. Tasks
Fine Grain	2 048	64	5 984

Table 3.6: Cholesky execution arguments

The implementation has 4 kernel tasks: `potrf`, `trsm`, `gemm` and `syrk`. The number of instances of each FPGA task accelerator is shown in table 3.7. The `potrf` has an execution pattern that makes the FPGA task accelerator in the FPGA device either consume a lot of resources or be very slow. Therefore, this has been considered in the configurations:

- `SMP`. Tasks are only run in the host processor.
- `FPGA`. Tasks are only run in the FPGA task accelerators.
- `MIX`. `potrf` kernel tasks are run in the host processor and other kernel task in FPGA task accelerators.

Task granularity	Num. `potrf` accs	Num. `trsm` accs	Num. `gemm` accels	Num. `syrk` accs
Fine Grain	1	1	3	1

Table 3.7: FPGA configuration for Cholesky benchmark

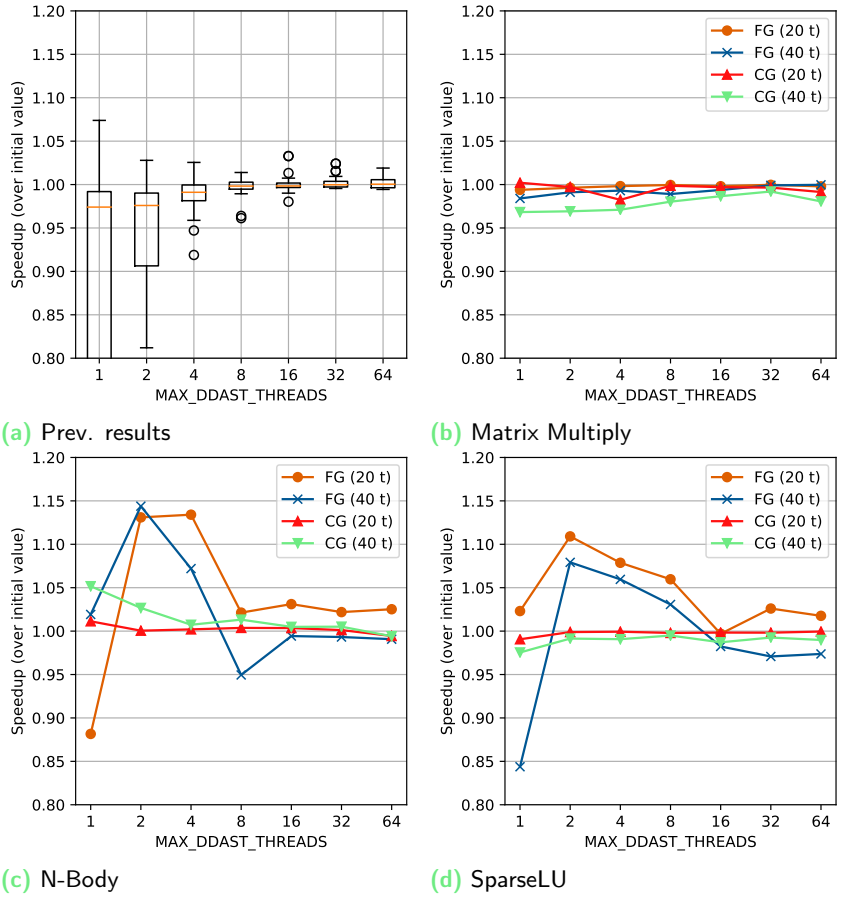
3.5.3 DDAST Tuning

The tuning analysis to find good default values for DDAST parameters has been repeated in the Power9 architecture like was done in the previous work [32]. This way, the previously established default values can be validated for the new architecture, or better ones could be found. Table 3.8 shows the initial values, the previous tuned values, and the new tuned values.

Parameter	Initial Value	Prev. Tuned Value	Tuned Value
MAX_DDAST_THREADS	∞	∞	$\lceil num\_threads/8 \rceil$
MAX_SPINS	20	4	1
MAX_OPS_THREAD	6	8	8
MIN_READY_TASKS	4	4	4

Table 3.8: DDAST parameters values

The speedup over the initial parameter value is shown in all plots of figure 3.14, figure 3.15, figure 3.16 and figure 3.17 (y-axis). The first chart of each figure shows the aggregated speedup results from the previous tuning, which was realized on other architectures. These charts are of boxplot type as the aggregation of different benchmarks and architectures may create a huge variability that is relevant for the parameter tuning. The other charts on each figure show the speedup for the three benchmarks with two amounts of threads and the two task granularities. These charts are simple line charts as results are not aggregated and they do not have a big variability.



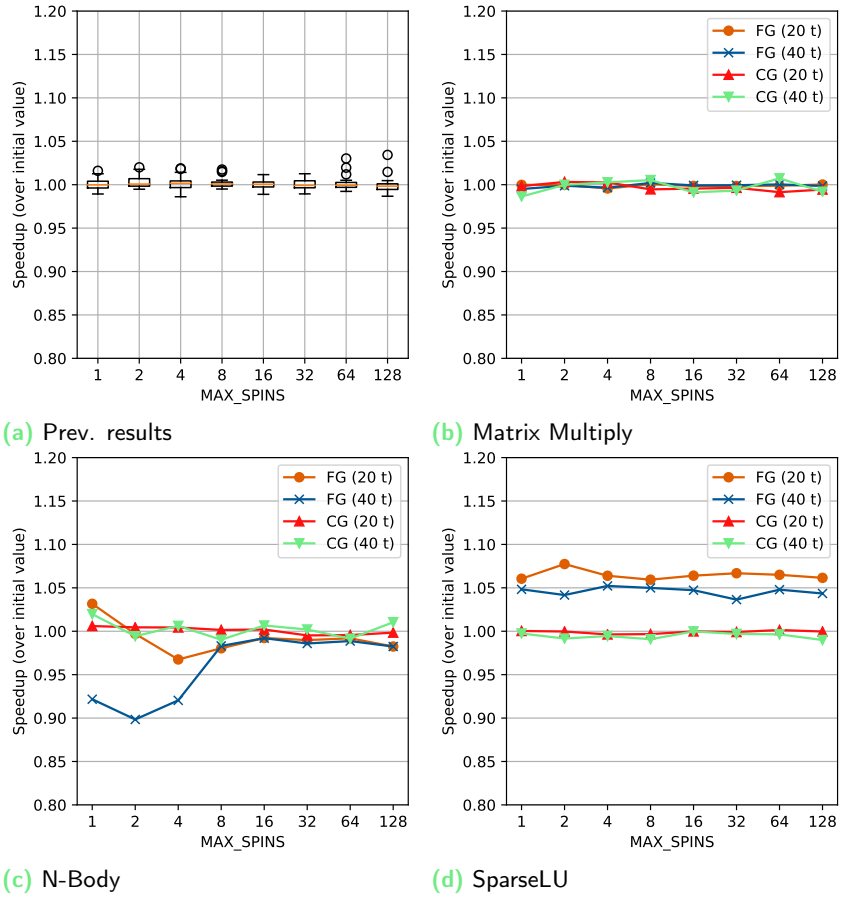
(a) Prev. results (b) Matrix Multiply

(c) N-Body (d) SparseLU

Figure 3.14: Speedup changing the MAX_DDAST_THREADS

Figure 3.14 shows the speedup evolution when changing the `MAX_DDAST_THREADS` value. The previous results (shown in figure 3.14a) do not report a clear benefit of limiting the amount of threads, so the previous tuned value was established at ∞. In contrast, figures 3.14c and 3.14d show an improvement of around a 10 % when reducing the value of `MAX_DDAST_THREADS`. The better results come from a better data locality achieved when the runtime activity is restricted to some threads.

The tuned value has been updated to $\lceil num\_threads/8 \rceil$. This amount guarantees enough threads to manage all runtime requests without losing the data locality benefits found in the new architecture. In fact, correlating the parameter value to the number of threads in the execution seems reasonable as it determines the DDAST pressure during the executions.



(a) Prev. results (b) Matrix Multiply

(c) N-Body (d) SparseLU

Figure 3.15: Speedup changing the `MAX_SPINS`

Figure 3.15 shows the speedup evolution when changing the `MAX_SPINS` value. The previous results (shown in figure 3.15a) report that the parameter value does not significantly impact the execution time, then the tuned parameter value was set to 4. The motivation was to retain the idle threads as little as possible in the DDAST callback,

considering the future scenario where the Functionality Dispatcher is used for several services. However, the same motivation can be used to set the new tuned value to 1 instead of 4.

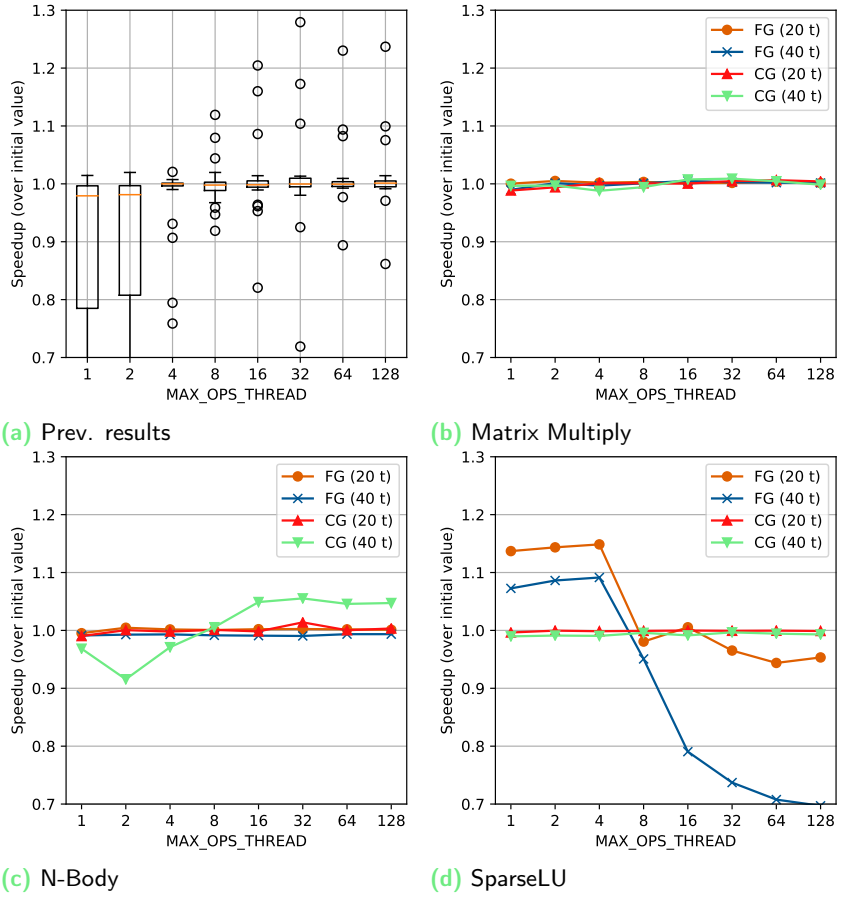
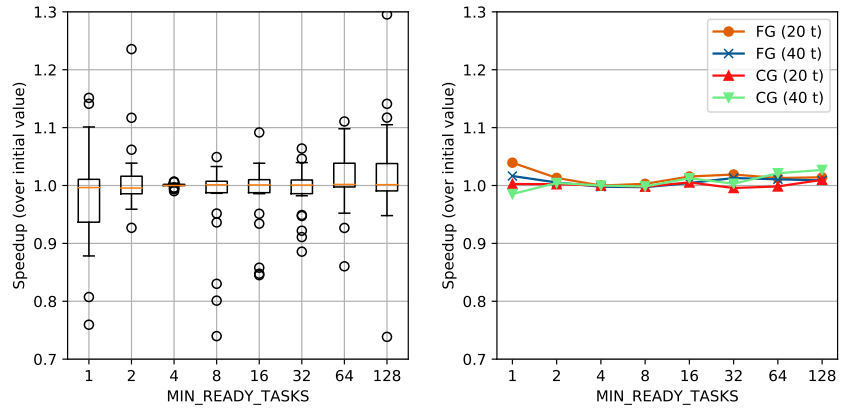


Figure 3.16: Speedup changing the `MAX_OPS_THREAD`

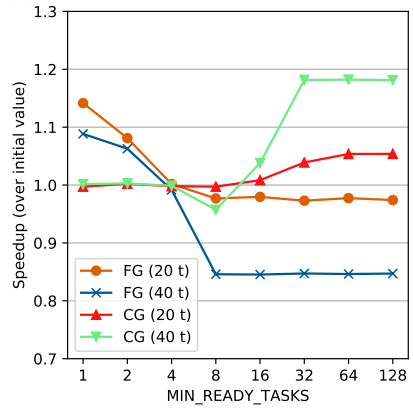
Figure 3.16 shows the speedup evolution when changing the `MAX_OPS_THREAD` value. The previous results (shown in figure 3.16a) report that the parameter value significantly impacts the execution time, but not in the same way for all the benchmarks. This behavior is also observed in the new results where the good values for N-Body are larger than 16, and the ones for SparseLU are smaller than 4. Therefore, the predefined tuned value has been kept to 8 because it seems to be a reasonable intermediate point.

Figure 3.17 shows the speedup evolution when changing the `MIN_READY_TASKS` value. The previous results (shown in figure 3.17a) report that the parameter value significantly impacts the execution time, but not in the same way for all the benchmarks. This behavior is also observed in the new results where the best values are not compatible between benchmarks or granularities in the same benchmark. Therefore, the predefined tuned value has been kept to 4 because it seems to be a reasonable intermediate point.

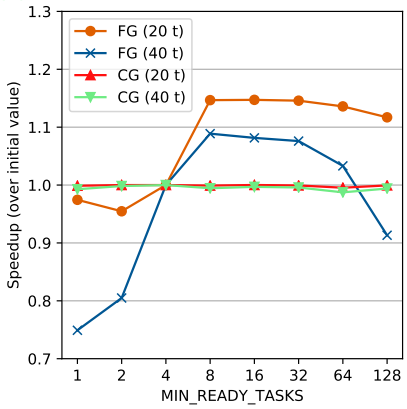


(a) Prev. results

(b) Matrix Multiply



(c) N-Body



(d) SparseLU

Figure 3.17: Speedup changing the `MIN_READY_TASKS`

3.5.4 DDAST Performance Comparison

The performance comparison of the DDAST runtime against the baseline runtime has been repeated for the Power9 architecture like it was done in the previous work. The results are shown for different runtime versions/configurations:

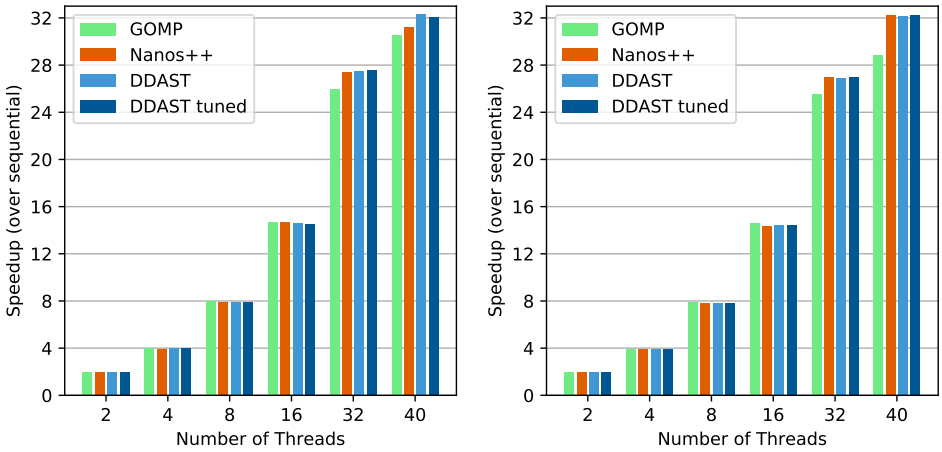
- *GOMP*. OpenMP implementation with the same task structure using the GNU Compiler runtime. This is a production runtime which performance can be used as a reference of the potential of our approach.

- *Nanos++*. Baseline OmpSs runtime (version 0.11a).

- *DDAST*. Runtime with the DDAST Manager and using the new tuned values for the DDAST parameters. The values are summarized in table 3.8 and are the same for all the runs. This version is implemented on top of Nanos++ runtime (version 0.11a).

- *DDAST tuned*. Same runtime as *DDAST* but with the best values of the DDAST parameters found during the tuning for each combination of benchmark, task granularity, and architecture.

The speedup over the sequential version of each benchmark is shown in all plots of figure 3.18, figure 3.19 and figure 3.20 (y-axis). All of them show strong scalability of Nanos++ and DDAST runtimes for Matrix Multiply, Sparse LU, and N-Body benchmarks. Therefore, the performance evolution when the runtimes must manage more computational resources can be seen. Each plot's label describes the architecture and the task granularity (fine-grain, FG, or coarse-grain, CG) of those results. *DDAST tuned* results are included because they show the potential of the proposal. Also, although it is out of the scope of this work, DDAST Manager parameters may be dynamically tuned at runtime to fit each application as shown in [71].

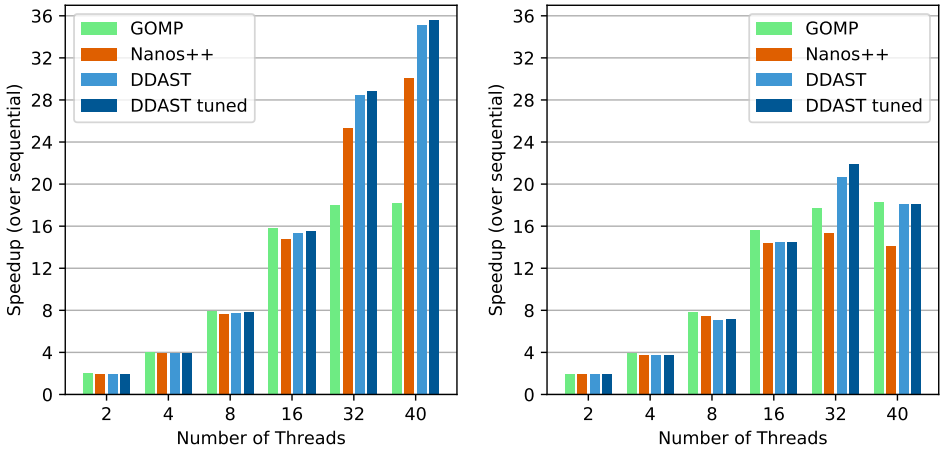


(a) Coarse Grain (b) Fine Grain

Figure 3.18: Matrix Multiply scalability

Figure 3.18 shows the Matrix Multiply scalability for the different runtime versions. In this benchmark, all runtimes behave very close (including *GOMP*) in both task granularities because *Nanos++* does not suffer from management contention in this configuration. Then, the results show that the DDAST design is also capable under circumstances that are not its design target.

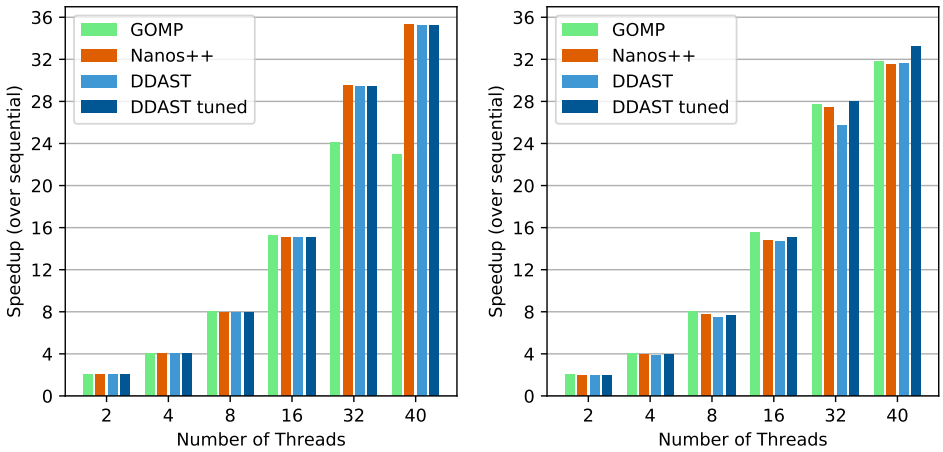
Figure 3.19 shows the N-Body scalability for the different runtime versions. The coarse-grain results (figure 3.19a) show that all Nanos++ based runtimes scale up to the maximum amount of threads. However, the DDAST versions improve the baseline runtime performance when the number of threads is larger than 16. The fine-grain results (figure 3.19b) show that *Nanos++* scales up to 16 threads, and then it stalls. *DDAST* keeps increasing the performance up to 32 threads, and then also stalls. The difference between both runtimes is the cost of task submission, which is smaller in



(a) Coarse Grain (b) Fine Grain

Figure 3.19: N-Body scalability

DDAST due to its asynchronous approach. This allows the application to create a huge amount of tasks faster in the new runtime model than in the baseline implementation. In both granularities, *GOMP* creates tasks faster than Nanos++ based runtimes for small amounts of worker threads (up to 16 threads) but suffers great contention from the idle worker threads when tasks are executed faster than created, which happens with 32-40 threads.



(a) Coarse Grain (b) Fine Grain

Figure 3.20: SparseLU scalability

Figure 3.20 shows the SparseLU scalability for the different runtime versions. Regardless of the task granularity, all runtimes provide very good scalability. The data dependences in this benchmark create an irregular task graph that usually requires processing multiple requests from different worker threads to discover a single ready task. This creates a challenging situation for the DDAST Manager where all possible ready tasks depend

on a message hidden by several other requests in a queue. However, the results show that even with this type of applications *DDAST* can achieve a performance similar to *Nanos++* and similar, or even better, to *GOMP*.

3.5.5 Concurrent Offloading to Accelerators

Figures 3.21, 3.22 and 3.23 show the performance (x-axis) of Matrix Multiply, Cholesky and N-Body when changing the ratio between SMP workers and FPGA helper threads (x-axis). For each ratio, different configurations are considered to see the performance impact of the different improvements. The configurations labeled with `w_cb` have the FPGA idle callback enabled, and the ones with `n_cb` have it disabled. Also, the configurations with `w_hb` have the FPGA hybrid worker enabled, and the one with `n_hb` have it disabled. All executions are done in the ZCU102 board, which has 4 ARM cores.

Figure 3.21 shows the Matrix Multiply performance in GFLOPS. The results are shown for different configurations described in the legend. In these configurations, `SMP`, `FPGA` and `SMP+FPGA` label the available architectures of tasks. All configurations use the same bitstream with 3 fine-grain FPGA task accelerators.

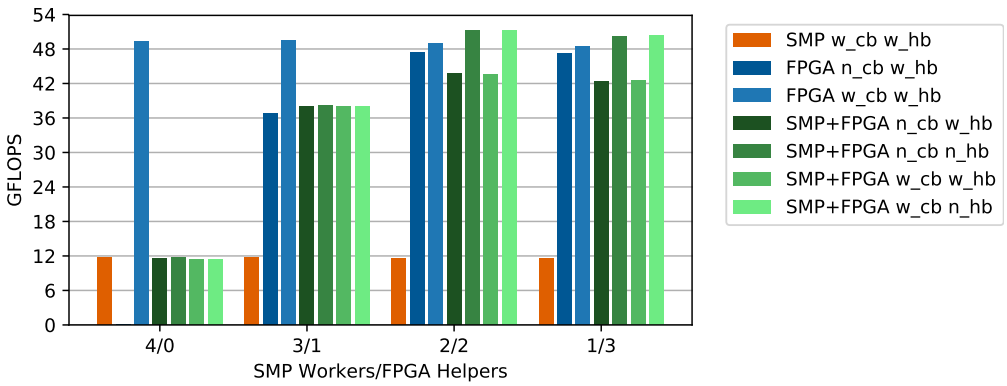


Figure 3.21: Matrix Multiply performance comparison with concurrent offloading

The results in figure 3.21 show that the performance with only `SMP` architecture is constant regardless the workers/helpers ratio. This means that the hybrid behavior of FPGA helper threads successfully helps to execute SMP tasks, as they pick SMP tasks when no FPGA tasks are ready for offloading. The same happens for `FPGA` architecture when the idle callback is enabled, as the SMP workers help to offload the FPGA tasks. In contrast, two or more FPGA helper threads are needed to effectively feed the FPGA task accelerators without the idle callback. Finally, the performance changes proportionally to the number of tasks executed in the SMP workers and the FPGA task accelerators when both are available. The used implementation depends on the workers/helpers ratio

and their configuration. Again, at least two FPGA helper threads are needed to feed the FPGA task accelerators correctly. Moreover, the hybrid behavior of FPGA helper threads causes a performance drop as the FPGA helper threads start executing SMP tasks after offloading the maximum number of tasks to the different FPGA task accelerators. Those SMP tasks take so long to finish that FPGA task accelerators consume all their work and become idle. This behavior can also be modified by increasing the maximum number of offloaded FPGA tasks, but it is not shown as it goes out of scope.

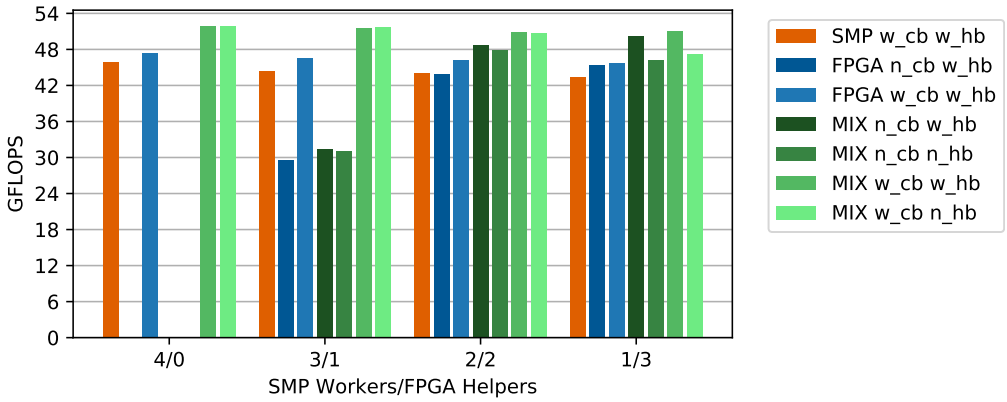


Figure 3.22: Cholesky performance comparison with concurrent offloading

Figure 3.22 shows the Cholesky performance in GFLOPS. The results are shown for different configurations described in the legend. In these configurations, `SMP`, `FPGA` and `MIX` label the architectures of tasks. In contrast to Matrix Multiply, the tasks do not have more than one available architecture. Therefore, the tasks of `MIX` configuration are run in the FPGA except `portf` tasks which are run in the host.

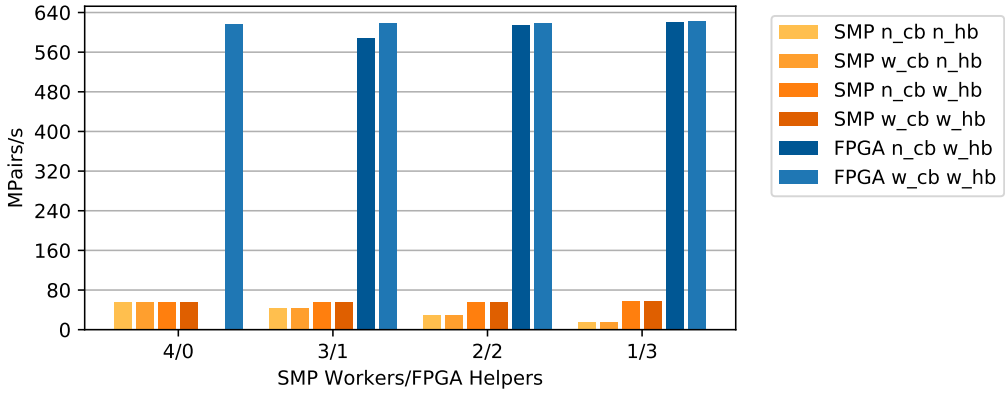


Figure 3.23: N-Body performance comparison with concurrent offloading

The results in figure 3.23 show that the idle callback helps to sustain the application performance regardless of the workers/helpers ratio. They also show that one helper thread without the idle callback is not enough to get good performance with FPGA tasks. In addition, the hybrid FPGA helper thread helps to improve the application performance

when most of the threads are FPGA helpers. In this case, the SMP tasks are executed in SMP workers or FPGA helper threads.

Figure 3.23 shows the N-Body performance in MPairs/s. The results are shown for different configurations described in the legend. In these configurations, `SMP` and `FPGA` label the architectures of tasks. In this case, multiple task architectures are not considered as the performance gap between them is quite large, as results show.

3.5.6 Tuning memory interconnections

Figures 3.24 and 3.25 show the execution traces of Matrix Multiply with 3 coarse-grain FPGA task accelerators and with a 128-bit memory port. The traces show the different operations realized, with different colors, by the FPGA task accelerators among time (x-axes). The orange color means that data is being read from memory into the local BRAMs, the blue color means that the task code is being executed, and green means that data is being written to memory. Both traces show the same duration to facilitate their comparison.

Figure 3.24 is from a bitstream that uses the default memory interconnections generated by AIT. It has two memory ports per FPGA task accelerators, and they are connected as shown in figure 3.4. The default memory interconnection unbalances the performance of the different FPGA task accelerators. The two that share the memory interconnect spent 293 and 95 microseconds reading and writing the task data. Meanwhile, the FPGA task accelerator with a non-shared memory interconnect only requires 147 and 71 microseconds on average for the same actions. However, the task execution is homogeneously between all FPGA task accelerators, and it elapses 437 microseconds on average.

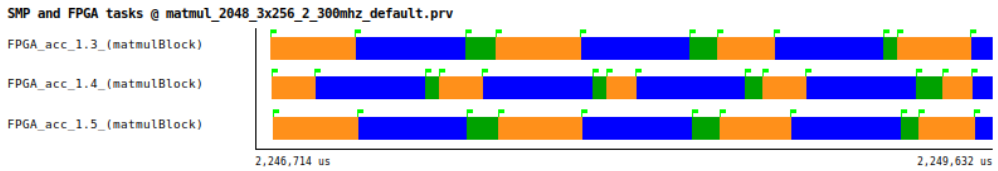


Figure 3.24: Execution trace of 3 FPGA task accelerators with the default memory interconnection

Figure 3.25 is from a bitstream that uses the new AIT capabilities to allow users to customize the design and define the interconnection of memory ports. It also has two memory ports per FPGA task accelerator, but each pair are connected to the same memory port interconnect, which are only used by one FPGA task accelerator. This way, each one uses a different memory interconnect. In this second execution, all FPGA task accelerators perform the memory movements more homogeneously. On average, they

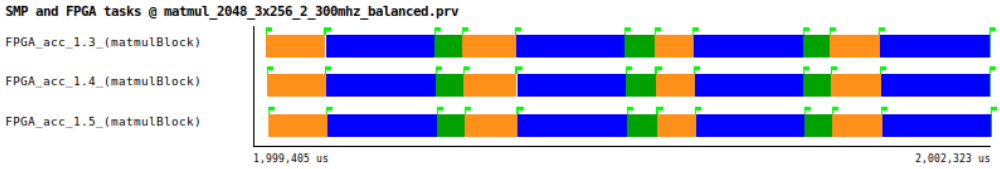


Figure 3.25: Execution trace of 3 FPGA task accelerators with a balanced memory interconnection

spent 192, 137, and 111 microseconds reading the data, executing the task, and writing the data, respectively.

The possibility of tuning memory mappings is essential to avoid unbalance and performance degradation due to the creation of a bottleneck. The default round-robin interconnection mechanism may be good enough for a wide range of applications. However, some applications, or some configurations of those, may have a memory access pattern that, combined with the number of memory ports, require the user handling to optimize performance.

3.5.7 Shared wide Memory Port

Figure 3.26 shows the speedup over the default memory interconnection mode (y-axis) of FPGA task accelerators. The results are for different bit widths (x-axis) of the shared memory port explained in section 3.3.3. The different series correspond to different benchmarks and task granularities (labeled in the legend). The Matrix Multiply results are for the `FPGA` configuration. The Cholesky results are for the fine-grain tasks and the `MIX` configuration.

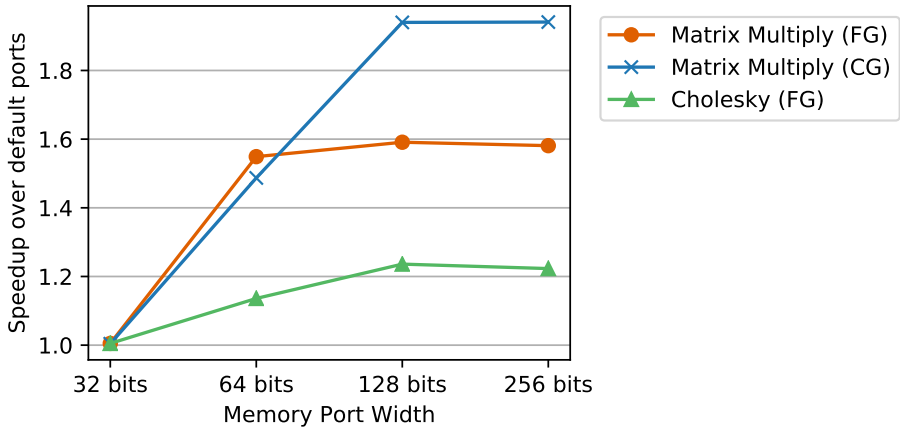


Figure 3.26: Performance comparison with different memory port widths

The results in figure 3.26 show that the best performance is achieved when the shared memory port width is near to the physical width of the memory port (128 bits in the ZCU102 board). The smallest width (32 bits) corresponds to the data type width. Then, the shared memory port is equivalent to the default memory port; therefore, the speedup is one. After that, the speedup increases with the memory port width. The different improvement values are due to:

- The percentage of execution time spent in data transfers. The major the percentage, the more room for improvement.
- The possibility of executing more tasks in the saved time. If there are no ready tasks for execution in the FPGA task accelerators, the fastest data transfers will not benefit the overall execution time.

Figures 3.27 and 3.28 show different execution traces of Matrix Multiply in the FPGA task accelerators. The different color regions (orange, blue, green) represent different activities (copying data from memory, executing task code, copying data to memory).

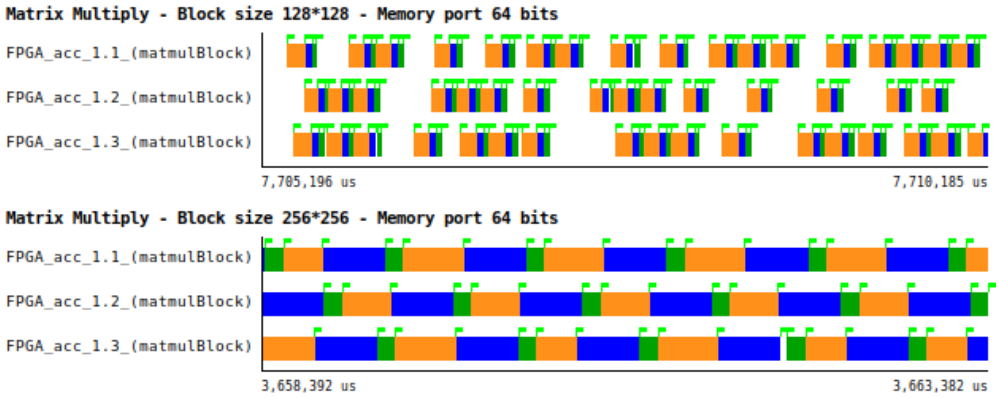


Figure 3.27: Task density comparison of Matrix Multiply execution traces with same memory port (64 bits) and different block sizes

Figure 3.28 shows two Matrix Multiply execution traces. Both use the same 64 bits memory port but for different task granularities. The traces only show a portion of the benchmark execution, but both have the same duration. In the fine-grain trace, a 42 % of trace time is spent on data movements (orange and green regions), an 18 % in task execution (blue regions), and a 40 % of the time the FPGA task accelerators are IDLE (white regions). In the coarse grain trace, a 52 % of trace time is spent on data movements, a 47.5 % in task execution, and a 0.5 % of the time the FPGA task accelerators are IDLE (white region). Although both traces spent most of the time in data transfers, the improvement when using a larger memory port is only noticed in the coarse grain due to the lack of ready tasks for execution in the smaller block size. This

behavior is noticed in figure 3.26 where the 128-bit memory port has a larger speedup for coarse-grain than fine-grain tasks.

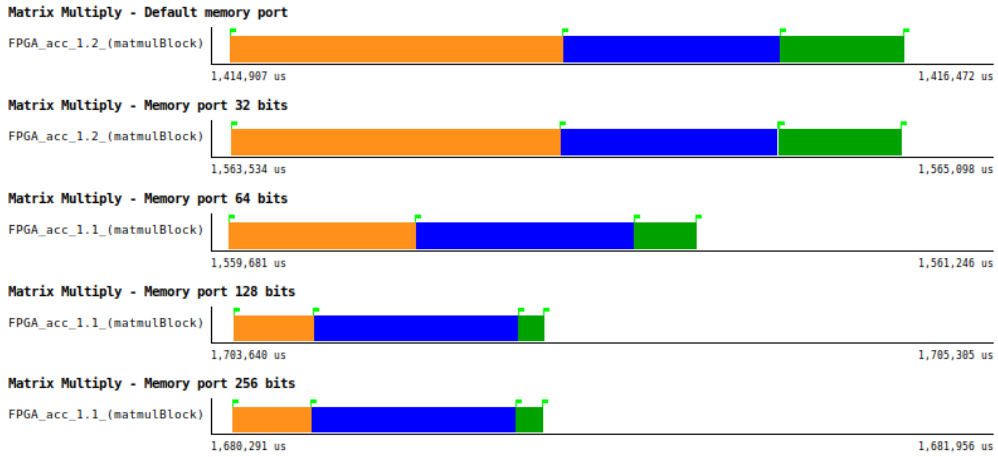


Figure 3.28: Execution traces of one Matrix Multiply coarse grain task with different memory ports

Figure 3.28 shows five execution traces of one Matrix Multiply task in one FPGA task accelerator. Each trace uses a different memory port, but all of them have represented the same duration (x-axis is time). From top to bottom, the configurations are: default memory ports, 32-bit shared memory port, 64-bit shared memory port, 128-bit shared memory port, and 256-bit shared memory port. The traces clearly show the reduction of time spent in data movements (orange and green regions) when the memory port width increases. They also show that the task execution time (blue regions) is not affected by the new logic.

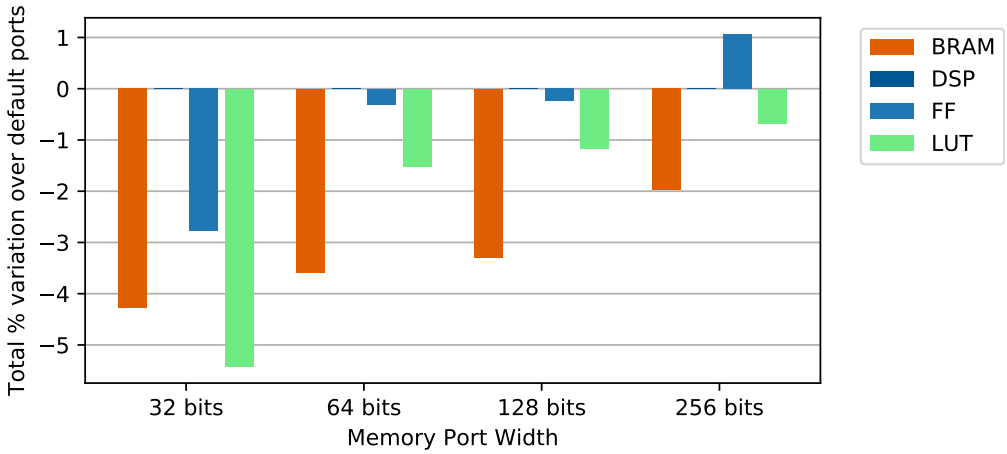


Figure 3.29: Matrix Multiply resources utilization variation with different memory port widths

Figure 3.29 shows the variation of total resources utilization percentage (y-axis) over the utilization in the default ports. The variation is shown for different widths of the shared

memory port width. Also, the different resources available in the FPGA are considered: BRAM, DSP, Flip-Flop (FF), and LookUp Table (LUT).

The results show that the utilization of resources decreases with the shared memory port. The first reason is the possibility of joining all memory ports into a single one, which is used to read the data of different task parameters. This is not possible when the parameters have different data types due to a HLS limitation. The single shared port simplifies the FPGA design interconnection and requires less logic. In the Matrix Multiply, the tasks have three parameters; thereby, the new design saves two of the three memory ports. On the other hand, the wider the memory port, the more logic needed to split the retrieved data into the different parameter type elements. This split logic is reflected in figure 3.29, because of the resource utilization increase with the memory port width. However, the task performance benefits of the wide memory port, as it increases the data read/write throughput.

3.6 Conclusion

This chapter presents several enhancements to achieve asynchronous, concurrent, and parameterized management for task-based systems. The runtime infrastructure developed for the DDAST Manager has been used to concurrently manage the co-processors in the Nanos++ runtime. The management has also been developed by means of asynchronous operations, which present a better resource utilization. On the one hand, the `num_instances`, `localmem`, `localmem_copies` and `no_localmem_copies` clauses have been introduced into the programming model to better define the desired behavior and improve resource utilization. On the other hand, some programming model clauses (e.g., `onto`) and usage constraints (e.g., FPGA configuration retrievement) have been updated to become transparent to application programmers. Moreover, other compiler and runtime options have been added to parameterize the internal behavior of the tools. For example, a compiler option to aggregate memory operations into wider ones or a runtime option to define the number of host threads that manage the FPGA task accelerators.

The proposal evaluation demonstrates the better efficiency of systems with asynchronous, concurrent, and flexible task management. Moreover, that approach allows a better expression of application requirements, leading to a performance increase. The results show how the parameterizable behavior, which may be tuned for each application, drove to a more efficient task management. Besides, the application tuning is easier thanks to the enhancements in the co-processors instrumentation, which allow clearly see the

different activities. All these improvements have been incorporated (or at least influenced) into the new versions of Nanos++ runtime, especially for the OmpSs@FPGA version.

The main lesson learned is that performance-oriented runtimes should be developed as a set of simple interacting actors. The complex behavior arises from the interaction of simple actors, and not from the interface that a single module exposes. This leads to a system that is simple to maintain and that provides, at the same time, great flexibility to adapt to different scenarios. A second lesson, one that is common knowledge though, is that memory accesses and their correct management are key to accelerators performance.

3.7 Publications

The list of thesis publications related to the explained in this chapter is:

- Exploiting Parallelism on GPUs and FPGAs with OmpSs.
  Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell. ANDARE 2017. [6]
  In this work, some of the OmpSs improvements are proposed and evaluated.

- Application Acceleration on FPGAs with OmpSs@FPGA.
  Jaume Bosch, Xubin Tan, Antonio Filgueras, Miquel Vidal, Marc Mateu, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, Jesus Labarta. FPT 2018. [8]
  In this work, more OmpSs improvements are proposed, and the evaluation is extended to more benchmarks.

- Asynchronous Runtime with Distributed Manager for Task-based Programming Models.
  Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé. PARCO 2020. [11]
  In this work, the complete DDAST proposal and evaluation are discussed.

The list of publications related to collaborations with the work presented in this chapter is:

- Picos, A Hardware Task-Dependence Manager for Task-Based Dataflow Programming Models.
  Xubin Tan, Jaume Bosch, Miquel Vidal, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, Mateo Valero. HPCS 2017. [13]

In this work, the DDAST implementation has been used as a baseline to develop the software runtime that communicates with Picos to resolve the task data dependences.

- Hardware Heterogeneous Task Scheduling for Task-based Programming Models.
  Xubin Tan, Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé. OpenMPCon 2018. [14]
  In this work, the DDAST implementation has been used as a baseline to develop the software runtime that communicates with Picos using different communication queues. Also, the OmpSs enhancements have been used in the FPGA task accelerators.

- TaskGenX: A Hardware-Software Proposal for Accelerating Task Parallelism.
  Kallia Chronaki, Marc Casas, Miquel Moretó, Jaume Bosch, Rosa M. Badia. ISC 2018. [15]
  In this work, the DDAST implementation has been used to characterize the software runtime overheads and define the hardware manager requirements.

- A Hardware Runtime for Task-Based Programming Models.
  Xubin Tan, Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, Mateo Valero. TPDS 2019. [16]
  In this work, the DDAST implementation has been used as a baseline to develop the software runtime that communicates with Picos using different communication queues. Also, the OmpSs enhancements have been used in the FPGA task accelerators.

- Adding Tightly-Integrated Task Scheduling Acceleration to a RISC-V Multi-core Processor.
  Lucas Morais, Vitor Silva, Alfredo Goldman, Carlos Álvarez, Jaume Bosch, Michael Frank, Guido Araujo. MICRO 2019. [17]
  In this work, the DDAST implementation has been used as a baseline to develop the software runtime that communicates with Picos using RISC-V instructions.

- OmpSs@FPGA framework for high performance FPGA computing.
  Juan Miquel de Haro, Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, Jesus Labarta. TC 2021 [Accepted for publication]. [19]
  In this work, the OmpSs@FPGA ecosystem enhancements have been used to program the applications and tune their performance to each system.

- High Performance Computing particle-pair distance algorithms, to generate X-ray spectra from 3D models.

César González, Jaume Bosch, Juan Miguel de Haro, Maurizio Paolini, Antonio Filgueras, Simone Balocco, Carlos, Álvarez, Ramon Pons. HPC 2021 [Under review]. [20]
In this work, the OmpSs@FPGA ecosystem enhancements have been used to program the application and tune its performance.

4 Proposal for Task Spawn in Co-processors

The proposal objective is to demonstrate the feasibility of spawning tasks and synchronizing them within the co-processors. The baseline OmpSs programming model considers the non-SMP tasks as leaf tasks. That means those tasks do not create child tasks, and they can only have sibling tasks that depend on them. Therefore, the tasks offloaded to devices should take long enough to overcome the device communication latency, which may change depending on the interconnection. Moreover, that limitation makes the devices slaves of the master host, and it can become a bottleneck due to the host-centric approach.

This chapter proposes an extension of OmpSs programming model to demonstrate that task-based parallel programming models can support the creation and synchronization (either implicit or explicit) of child tasks within co-processors. The extension design is implemented over baseline OmpSs tool-chain, but the design could be used for other task-based programming models. Moreover, the flexibility of the proposal allows directly managing tasks within co-processors without involving the host runtime, saving communication time and increasing application performance.

Section 4.1 describes the key ideas of the proposal design. Then, sections 4.2 to 4.6 describe the modifications done starting from the OmpSs programming model, up to the FPGA bitstream design. After them, section 4.7 shows an evaluation of different benchmarks using the new capabilities. Finally, section 4.8 concludes the chapter with the key contributions, and section 4.9 lists the publications related to this chapter.

4.1 Proposal Design

The proposed design is a system architecture that allows the co-processors to interact with the runtime to create tasks and synchronize them. The design only requires an accelerator-runtime communication channel and a unique identifier to determine in which context the tasks are being created (usually, it is the parent task identifier). The co-processor can then put the required information in the communication channel to spawn

the task and continue. On the runtime side, some manager will read that information and update the runtime structures accordingly. This approach allows the co-processors to create tasks asynchronously.

A way to allow the co-processors synchronizing the spawned tasks is also considered, which is the main functionality of the taskwait directive. The co-processor has only to send the taskwait information in the communication channel and wait until the runtime fulfills the request. On the runtime side, the runtime manager notifies the co-processor when all required tasks have finished.

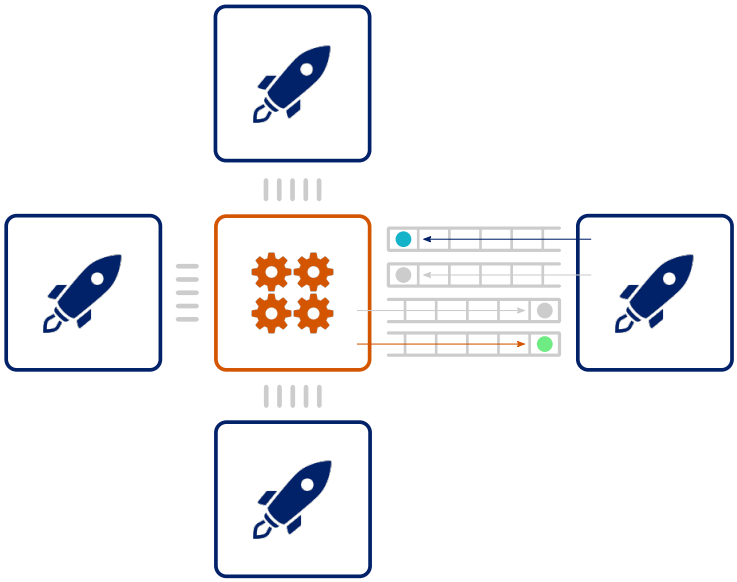


Figure 4.1: Proposal design model for co-processors management with task spawn capabilities

Figure 4.1 shows the structure of the proposal design. Over figure 1.1, two new communication channels have been added between the host and the co-processors. Those channels handle the tasks spawned by the co-processors or the synchronization requests to the runtime.

This design allows some optimization on the co-processor side if the spawned tasks can be retained inside the co-processor, and the host does not need to be aware of them. Therefore, minimal runtime support can be introduced in the co-processor to handle those tasks without involving the host runtime (as shown in figure 4.2). In the case of a task spawn, the tasks without data dependences could be directly forwarded into the input queue, as if the host submitted them. In the case of a task synchronization, the co-processor has to store the number of tasks executed in each task context (parent task identifier used in the task spawn) and the number of tasks internally created for that identifier. Note that the amount of executed tasks must count for tasks directly handled in the co-processors and tasks forwarded to the host runtime.

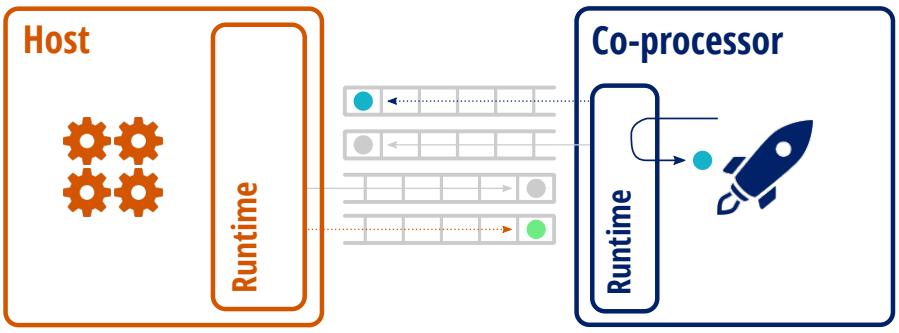


Figure 4.2: Proposal design model for co-processors management with task spawn capabilities and distributed runtime support

The proposed design is flexible enough to move some key functionalities from co-processors to host or vice-versa. It just requires replacing some components. For example, the co-processor may internally handle the spawned tasks without involving the host runtime or just forward the information to the host runtime. This flexibility enables the option to fit the proposal in a wide range of co-processors, from small ones in embedded systems to big ones in HPC servers or clouds.

4.2 Programming model extension

The extension of the OmpSs programming model [4] defines the application behavior when a programming model directive is found in the body of a task annotated with the `device(fpga)` clause of a `target` directive. By default, the current behavior is kept (like when those programming model directives are found inside an SMP task). However, the scope of the extension has been limited to outline tasks and not considered inline tasks. The outline tasks have a more clear data-scope, as all data must be accessed through the function parameters.

```
1  #pragma omp target device(fpga) copy_inout([BSIZE]array) \
2    num_instances(3)
3  #pragma omp task
4  void update_array_fpga(int *array, const int val) {
5    for (int i=0; i<BSIZE; ++i) array[i] += val;
6  }
7
8  #pragma omp target device(fpga) copy_inout([SIZE]array)
9  #pragma omp task
10 void update_array_blocked(int *array, const int SIZE) {
11   for (int i=0; i<SIZE; i+=BSIZE) {
12     update_array_fpga(array+i, 2020);
13   }
```

```
14    #pragma omp taskwait
15  }
16
17  int main(...) {
18    int array[NUM_BLOCKS*BSIZE];
19    update_array_blocked(array, NUM_BLOCKS*BSIZE);
20    #pragma omp taskwait
21  }
```

Listing 4.1: OmpSs example with FPGA nested tasks

Listing 4.1 shows an example of a synthetic benchmark implemented with nested FPGA tasks. It just updates an array of integers with a fixed value using a blocking approach to obtain task parallelism. The code shows the possibility of calling a task and synchronizing the created tasks with a taskwait, all of it inside an FPGA task.

4.3 Mercurium Compiler Support

The support of new OmpSs capabilities has been integrated into Mercurium compiler. The changes involved the OmpSs and the FPGA device translation level phases and changes in the compiler core to extend some representations.

The first major change involved the compiler core in order to store the creation context where a task is being created. A new member has been added in the `task_environment` elements called `creation_ctx`. It contains the information of parent `task_environment` if known. The task statements are walked to fill this new member, and the current environment is forwarded to inner task calls found during the walk. This way, the compiler decides whether the programming model directives must be handled regularly or handled depending on the enclosing device phase.

Using the specific device phase to handle the programming model directives allows them to decide whether the default transformation suits the device or a device-specific transformation must be used. Those transformations usually mean placing a runtime API call, therefore the device decides whether to support the default runtime API call or use a device custom API. Indeed, the device can ignore some information in the programming model directives to fit the device's capabilities.

4.3.1 Task Directive

The task directive is the one that represents the task spawn in the programming model. The support of such directive has been reduced to a subset of all available clauses when

found inside an FPGA task. However, future implementations could extend those support. The list of supported features is:

- Dependences clauses: `in`, `out` and `inout`.
- Extension through `target` directive.
  - Device clauses: `device`
  - Copies clauses: `copy_in`, `copy_out`, `copy_inout`, `copy_deps` and `no_copy_deps`.
  - Localmem clauses: `localmem`, `localmem_copies` and `no_localmem_copies`.

The default transformation into a runtime API call has not been used due to its large number of arguments and non-desired features. A new Nanos++ API has been added with a compact format that better suits the FPGA device needs. The new API is `nanos_fpga_create_wd_async` which is further detailed in section 4.4.1 and has been declared as shown in listing 4.4.

The unique task type identifier allows the HWR and Nanos++ to identify the task being spawned uniquely. This identifier can be compared with the struct `nanos_const_wd_definition_t` which is a parameter of default task spawn API. The HWR uses the type identifier to schedule the task in the FPGA device, whether choosing which accelerator in the FPGA will execute the task, or deciding to reverse-offloading the task to the host runtime. When the task is reverse-offloaded, the host runtime uses the type identifier value to retrieve the `nanos_const_wd_definition_t` information.

The association between a task type identifier and the `nanos_const_wd_definition_t` information is done during the runtime initialization. The compiler uses the `nanos_post_init` functionality of Nanos++ runtime, which allows defining a set of functions to be executed after the runtime initialization. This functionality is used to execute a set of functions from the application binary that call a new runtime API to provide the information associated with all existing types in the application. The new Nanos++ API is `nanos_fpga_register_wd_info` which is explained in section 4.4.1.

4.3.2 Taskwait Directive

The explicit task synchronization is done using the taskwait directive. The support of such directive has been implemented using a new API which has a similar declaration to the default API but with different parameter types. The new API is `nanos_fpga_wg_wait_completion` which is further detailed in section 4.4.1.

4.3.3 HLS Source Code

The intermediate HLS source code files will have calls to new Nanos++ APIs when the FPGA task wants to spawn or synchronize child tasks. Therefore, the wrapper generated by Mercurium must include an implementation of those APIs when the FPGA task may call them. The main goal of those APIs in the FPGA task accelerator is to generate a message for the HWR runtime, which contains the needed information. Also, retrieve from the HWR the needed information (for example, when the child tasks have finished).

The existing input and output streams are declared as arguments of the top-level function in the HLS source code. In the baseline, they are only used in the body of the top-level function. However, they may be needed at any point of the source code to spawn/synchronize tasks with the new capabilities. Therefore, those streams must be available at any point of the source code, which may be possible in two ways:

- Argument forwarding. This mechanism consists on adding two arguments (input and output streams) to any function of the HLS source file. It includes user-defined functions like the function annotated with the target FPGA directive and any other function called in its body. This is a recursive walk of the functions to alter the user code, which is not easy to handle and may end in side effects.
- Global variable declaration. Instead of declaring the input and output stream as arguments of the top level function, they could be declared as global variables which scope includes all the HLS source file.

```
1  hls_axis_t mcxx_inStream;
2  hls_axis_t mcxx_outStream;
3
4  void histogram(const float *input, unsigned int *counters)
5  {
6    //...
7  }
8
9  void histogram_mcxx_hls_wrapper(
10   float *input_port, unsigned int *counters_port)
11 {
12   #pragma HLS INTERFACE ap_ctrl_none port=return
13   #pragma HLS INTERFACE axis port=mcxx_inStream
14   #pragma HLS INTERFACE axis port=mcxx_outStream
15
16   //...
17 }
```

Listing 4.2: FPGA task accelerator wrapper example with global streams (non-valid design)

The argument forwarding was discarded due to the complex handling in the compiler and the possibility of side effect that may result in invalid code. The global variable declaration was tried as shown in listing 4.2 but the source code is invalid due to the HLS limitations that do not allow the declaration of AXI-Stream interfaces on global variables.

```
ap_uint<8>  mcxx_eInPort;
ap_uint<70> mcxx_eOutPort;

void mcxx_write_eout_port(
  const unsigned long long int data, const unsigned short dest,
  const unsigned char last)
{
  #pragma HLS INTERFACE ap_hs port=mcxx_eOutPort register
  ap_uint<72> tmp = data;
  tmp = (tmp << 6) | ((dest & 0x1F) << 1) | (last & 0x1);
  mcxx_eOutPort = tmp;
}

ap_uint<8> mcxx_read_ein_port()
{
  #pragma HLS INTERFACE ap_hs port=mcxx_eInPort
  ap_uint<8> data = mcxx_eInPort;
  return data;
}

void histogram(const float *input, unsigned int *counters)
{
  //...
}

void histogram_mcxx_hls_wrapper(
  hls_axis_t mcxx_inStream, hls_axis_t mcxx_outStream,
  float *input_port, unsigned int *counters_port)
{
  #pragma HLS INTERFACE ap_ctrl_none port=return
  #pragma HLS INTERFACE axis port=mcxx_inStream
  #pragma HLS INTERFACE axis port=mcxx_outStream
  #pragma HLS INTERFACE hs port=mcxx_eInPort
  #pragma HLS INTERFACE hs port=mcxx_eOutPort

  //...
}
```

Listing 4.3: FPGA task accelerator wrapper example with global handshake ports

The solution has been the declaration of new input and output ports that use the handshake protocol instead of the AXI-Stream protocol (show in listing 4.3). The input port has been declared 8 bits wide as it is enough for the type of data that HWR sends to the FPGA task accelerators. The output port has been declared 70 bits wide to

compress the 64 data bits, 5 destination ID bits, and the last word bit. Those ports can be easily converted from/to AXI-Stream outside the HLS IP block and routed like the regular streams. The new ports are only added to FPGA task accelerators that need the new functionalities. The others are kept with the baseline interface. Then, the `mcxx_inStream` and `mcxx_outStream` AXI-Stream ports are always created.

For convenience, two auxiliary functions have been defined to write `mcxx_eOutPort` (`mcxx_write_eout_port`) and read `mcxx_eInPort` (`mcxx_read_ein_port`). The write function takes as arguments the three values that are concatenated in a word and wrote using the handshake port.

nanos_fpga_create_wd_async

The wrapper generated in the HLS intermediate file implements the `nanos_fpga_create_wd_async` API to support calling it from the FPGA task accelerator code. The implementation forwards the information of the new task to the HWR, using the `mcxx_eOutPort` that communicates all FPGA task accelerators with the HWR.

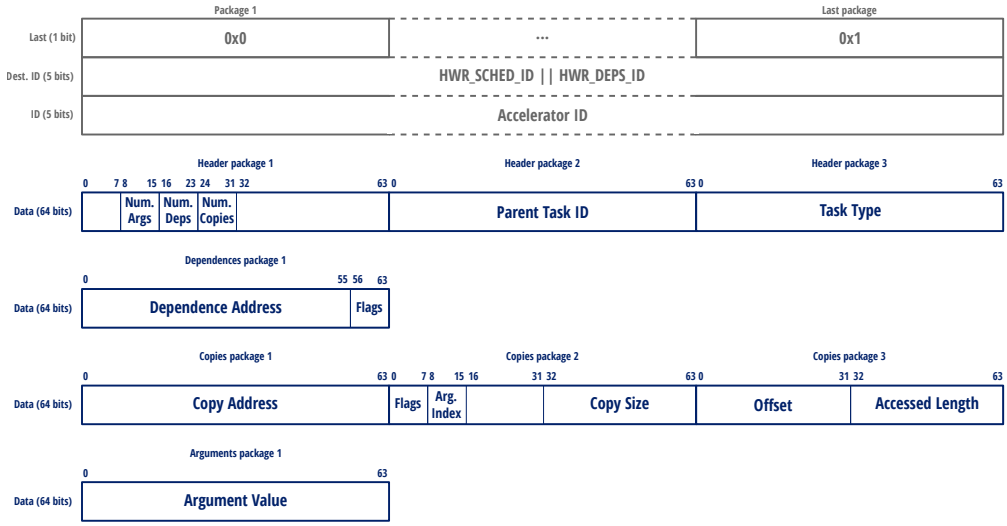


Figure 4.3: Format of new task message for HWR

The format of the AXI-Stream message sent to the HWR can be seen in figure 4.3. The gray part shows the protocol information that includes 1 bit to define if the package is the last of the message and two identifiers (5 bits wide) for the destination and the source. The destination changes between `HWR_SCHED_ID`, if the task does not have dependences and is ready for execution, and `HWR_DEPS_ID`, if the task has data dependences. Then, the blue part shows the data part of the packages. The data information is spitted into four groups: the header, the dependences, the copies, and the arguments (the last three

groups are repeated according to the number of dependences, copies, and arguments, respectively). Three packages compose the header: the first contains the number of task arguments, dependences, and copies; the second package contains the identifier of the task that is spawning the child task; and the third package contains the type identifier of the child task. Each dependence uses one package containing the dependence memory address in the lower 55 bits and the dependence flags in the upper 8 bits. Each copy uses three packages: the first contains the memory address of the copy region; the second contains the copy flags, the argument index that the copy refers to, and the size of the region to copy; the third package contains the offset not accessed at the beginning of the copy and the accessed length after the copy. Finally, each argument uses one package that contains the argument value, which may be a memory pointer or a scalar value.

After sending the message, the FPGA task accelerator waits for an acknowledge message in the `mcxx_inStream`. If the data message is a `0x00`, the HWR could not handle the task spawn, and the message must be resent. In contrast, if it is a `0x01`, the task has been successfully created. This acknowledge is the key to avoid deadlocks between several FPGA task accelerators creating tasks concurrently. The acknowledge ensures liveness in the interconnection between the FPGA task accelerators and the HWR because no message will remain in that channel waiting for being processed by the HWR. Instead, it will loop between the FPGA task accelerator and the HWR but allowing others to insert their messages in the middle.

nanos_fpga_wg_wait_completion

The wrapper generated in the HLS intermediate file implements the `nanos_fpga_wg_wait_completion` API to support calling it from the FPGA task accelerator code. The implementation forwards the information of the blocking task and the number of child tasks that must be synchronized to the HWR. The communication is done over `mcxx_eOutPort` that communicates all FPGA task accelerators with the HWR.

The AXI-Stream format of the message sent to HWR can be seen in figure 4.4. The gray part shows the protocol information that includes 1 bit to define if the package is the last of the message and two identifiers (5 bits wide) for the destination and the source. The data part of the message is shown in blue and it encodes: 32 bits for the number of child tasks to synchronize, a `0x1` in the 32th bit that defines that the task wants to block until the children finish, and 64 bits for the task identifier that is blocking.

After sending the message, the FPGA task accelerator waits until the HWR sends a response through `mcxx_inStream`. This response notifies that the tasks spawned in the requested context have finished. Therefore, the taskwait has been accomplished.

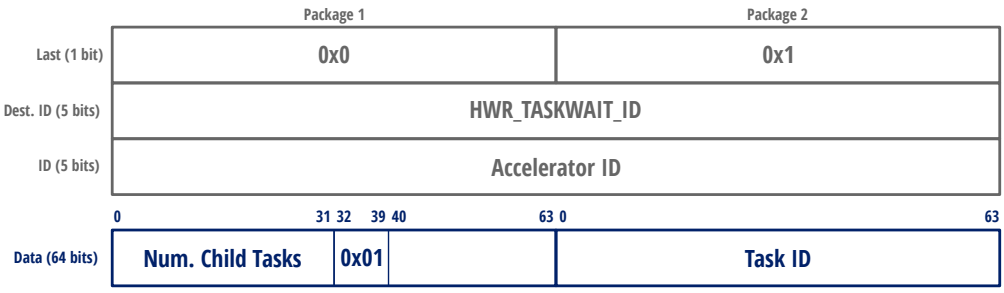


Figure 4.4: Format of block message for Taskwait manager

4.4 Nanos++ Runtime Support

The support for task spawn in FPGA devices requires coordination between Nanos++ (host runtime) and the HWR in the FPGA device. Both runtimes need to cooperate and exchange information when needed to guarantee the correctness of the application execution. This communication can be restricted as much as possible to avoid non-necessary round-trips. The goal is to keep the management near the action as much as possible. Sections 4.4.2 and 4.4.3 explain the new two callbacks registered in the Functionality Dispatcher [11] to poll the communication queues with the HWR. Before, section 4.4.1 describes the new APIs introduced in the runtime to support the new capabilities.

4.4.1 New APIs

The Nanos++ API has been extended with four new APIs. `nanos_fpga_current_wd`, `nanos_fpga_wg_wait_completion` and `nanos_fpga_create_wd_async` can only be called from the FPGA task accelerators, they will throw an error if called in the host. `nanos_fpga_register_wd_info` is the other new API which is used to provide the host runtime with information about tasks that FPGA may create.

nanos_fpga_create_wd_async

The `nanos_fpga_create_wd_async` API asynchronously creates a new task with the provided arguments, data copies and dependences. The API can only be called from the FPGA task accelerators, and it throws an error if called in the host. The API declaration is shown in listing 4.4 and its parameters are:

- `type`. Unique identifier of task type (64 bits wide).

- `numArgs`. Number of task arguments pointed by `args`.
- `args`. Pointer to the `numArgs` arguments of the task. Each argument (64 bits wide) may be a pointer or a scalar.
- `numDeps`. Number of task dependences pointed by `deps`.
- `deps`. Pointer to the `numDeps` dependences of the task. Each dependence (64 bits wide) is a memory pointer.
- `depsFlags`. Pointer to the `numDeps` flags for the task dependences. Each flag (8 bits wide) defines the directionality of the corresponding dependence with the codes defined by `nanos_fpga_argflag_t`.
- `numCopies`. Number of task copies pointed by `copies`.
- `copies`. Pointer to the `numCopies` copies of the task. Each copy (196 bits wide) packages the following information:
  - `address`. Base memory address of the copy region (64 bits wide).
  - `flags`. Copy flags that define the copy directionlity using the `nanos_fpga_argflag_t` codes (8 bits wide).
  - `ard_idx`. Index of the argument that the copy refers to (64 bits wide).
  - `size`. Amount of bytes to be copied (32 bits wide).
  - `offset`. Amount of bytes not accessed at the beginning of the region (32 bits wide).
  - `accessed_length`. Amount of bytes accessed after `offset` (32 bits wide).

```
typedef enum {
  NANOS_ARGFLAG_DEP_OUT  = 0x08,
  NANOS_ARGFLAG_DEP_IN   = 0x04,
  NANOS_ARGFLAG_COPY_OUT = 0x02,
  NANOS_ARGFLAG_COPY_IN  = 0x01,
  NANOS_ARGFLAG_NONE     = 0x00
} nanos_fpga_argflag_t;

typedef struct __attribute__ ((__packed__)) {
  unsigned long long int address;
  unsigned char flags;
  unsigned char arg_idx;
  unsigned short _padding;
  unsigned int size;
  unsigned int offset;
  unsigned int accessed_length;
} nanos_fpga_copyinfo_t;

void nanos_fpga_create_wd_async(
  const unsigned long long int type,
  const unsigned char numArgs, const unsigned long long int * args,
  const unsigned char numDeps, const unsigned long long int * deps,
  const unsigned char * depsFlags,
  const unsigned char numCopies, const nanos_fpga_copyinfo_t * copies);
```

Listing 4.4: Nanos++ FPGA API for task spawn

nanos_fpga_current_wd

The `nanos_fpga_current_wd` API returns a 64 bits long integer with the identifier of the task being executed in the FPGA task accelerator. The API can only be called from the FPGA task accelerators, and it throws an error if called in the host. The API declaration is shown in listing 4.5.

```
unsigned long long int nanos_fpga_current_wd();
```

Listing 4.5: Nanos++ FPGA API to retrieve current task information

nanos_fpga_wg_wait_completion

The `nanos_fpga_wg_wait_completion` API synchronizes the child tasks, blocking the caller until the execution of child tasks have finished. The API can only be called from the FPGA task accelerators, and it throws an error if called in the host. The API declaration is shown in listing 4.6 and its parameters are:

- `uwg`. Identifier of the task which child tasks will be synchronized (64 bits wide).
- `avoidFlush`. Boolean that requires skipping the task data flush into the parent task address space.

```
nanos_err_t nanos_fpga_wg_wait_completion(
  unsigned long long int uwg, unsigned char avoidFlush);
```

Listing 4.6: Nanos++ FPGA API for task synchronization

nanos_fpga_register_wd_info

The `nanos_fpga_register_wd_info` API stores the information that the runtime will need to create the WD for a given task type code. Those task types are sent by the HWR to identify the type of task being reverse offloaded. The API declaration is shown in listing 4.7 and its parameters are:

- `type`. Unique identifier of task type (64 bits wide).
- `numDevices`. Number of devices pointed by `devices`.
- `devices`. Pointer to the `numDevices` devices information that points to the used code function.
- `translate`. Function pointer for task arguments translation between host and device address spaces.

```
nanos_err_t nanos_fpga_register_wd_info(
  uint64_t type,
  size_t num_devices, nanos_device_t * devices,
  nanos_translate_args_t translate);
```

Listing 4.7: Nanos++ FPGA API for task information registration

4.4.2 FPGA Create WD Listener

The FPGA task accelerators can spawn tasks that cannot be directly executed in the FPGA device, due to a different architecture (SMP, GPU, etc.) or due to data dependences which HWR may not be able to solve. This tasks are retrieved from the FPGA device through the `xtasksTryGetNewTask` xTasks library API. The API returns a package with the information of one FPGA spawned task, if any. Then, the runtime can create a WD with the retrieved information and the extra information provided to the runtime through `nanos_fpga_register_wd_info`.

The poll of `xtasksTryGetNewTask` is done at two points: 1) From a new callback registered in the Functionality Dispatcher. 2) From the yield epilogue in the loop of the FPGA helper threads. All points are coordinated by an exclusion lock, which ensures only one thread retrieves tasks concurrently, and others skip the handling. This is needed as far as FPGA device can send tasks with dependences which must be handled in order. Besides, a thread starts retrieving tasks until the xTasks library returns an invalid task information or until the throttling policy does not allow the creation of more tasks.

4.4.3 FPGA Instrumentation Listener

The new task spawn capabilities in the FPGA devices required re-engineering the FPGA instrumentation support. The baseline support was done using an event buffer per FPGA task, which means every task offloaded to an FPGA accelerator has its own buffer. This approach allows each task to safely write its buffer, which will remain untouched until the host synchronizes the task when the events in the buffer are handled (forwarded to the instrumentation library). However, keeping this approach has two main drawbacks: 1) The need to reserve a memory region for instrumentation buffers used by FPGA spawned tasks, which also has to be managed by the HWR. 2) The requirement of a hard synchronization to flush the buffers in the host instrumentation library. Therefore, the FPGA instrumentation has been re-implemented to use an instrumentation buffer per FPGA task accelerator, which has the following benefits:

- The FPGA task accelerators can generate instrumentation events regardless of the origin of the running task.
- Instrumentation setup is not sent for every task. The baseline implementation sends the instrumentation buffer address with every task, which is only sent once with the new implementation. This modification is managed by xTasks library, and it is further explained in section 4.5.1.
- Each buffer can be larger. The number of FPGA task accelerators is smaller than the number of alive tasks, so the instrumentation buffer size can be larger using the same amount of memory.

The structure of the components involved in the FPGA device instrumentation is shown in figure 4.5. The components are the same as the baseline structure, only replacing the instrumentation buffer per-task by circular queues per-accelerator. The change in the queue format is transparent to the runtime because the underlying xTasks library manages it. Nanos++ uses the `xtasksGetInstrumentData` (explained in section 2.4) which copies into a user-level buffer the number of required events.

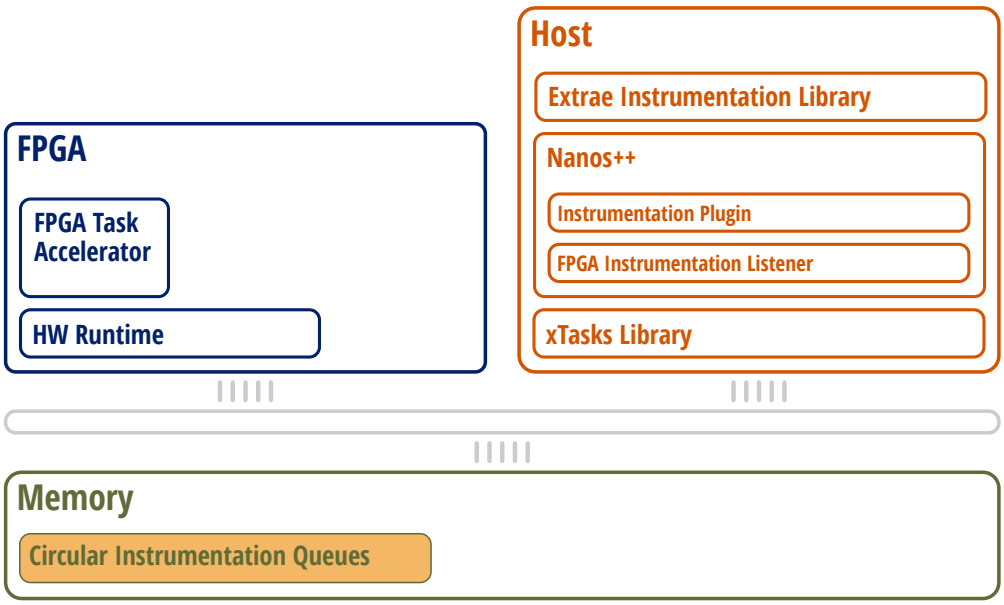


Figure 4.5: New instrumentation structure

The changes in the Nanos++ runtime to support the new instrumentation are about when the instrumentation events are retrieved from the instrumentation buffers. The baseline implementation retrieves the events after synchronizing each task executed in an FPGA task accelerator. The new implementation periodically polls the xTasks API to check if new events from any FPGA task accelerator are available. The poll has been implemented registering a new callback in the Functionality Dispatcher and introducing it in the loop of FPGA helper threads. Moreover, a final check has been added in the FPGA plugin cleanup to ensure that all events remaining in the buffers are handled. Only one thread can handle the same accelerator at any time. This is intended to avoid the contention at xTasks API when several threads try to retrieve events for the same FPGA task accelerator.

4.5 xTasks Library Support

The communication between the host runtime (Nanos++) and the FPGA device runtime is done through the xTasks library. To support the new functionalities, the library added new APIs available for Nanos++ runtime (described in section 4.5.1) which interface the new communication queues read by the HWR (described in section 4.5.2). Also, the instrumentation API have been updated to support the new instrumentation, which provides an instrumentation buffer for each FPGA task accelerator instead of for each task.

4.5.1 New APIs

The xTasks API has been extended with two new APIs and one API has been updated. The following points detail those APIs.

xtasksTryGetNewTask

The `xtasksTryGetNewTask` API tries to retrieve a task that FPGA spawned and it has been offloaded back to the host. The caller must ensure that tasks with dependences are handled in order. Otherwise, their execution order may be reversed. The API declaration is shown in listing 4.8, and its parameters are:

- `task`. Pointer to a valid `xtasks_newtask` pointer that will be set if a task is retrieved. If the pointer already points a valid task information, it can be reallocated to ensure enough space for all task information (header, arguments, dependences, and copies). Although the pointer is to a `xtasks_newtask` struct, the allocated memory region will be large enough to contain all task information. The caller is responsible for freeing the pointer memory region after using the data.
- Returns `XTASKS_SUCCESS` if `task` has been updated and points to a valid task, and returns `XTASKS_PENDING` if not task has been retrieved.

Each call to `xtasksTryGetNewTask` checks if the word in the reading head has the valid bits set. If it has, the reading head is set to the next head word. Then the `task` pointer is set to a large enough memory region, and the data is copied there from the communication queue. After reading each word from the queue, they are set to zero, leaving them available again for another task. The head word is not set to zero after reading, but its valid bits are clean after the whole task has been processed.

The retrieved task identifiers (`taskId` and `parentId`) in the `xtasks_newtask` structure must be provided to `xtasksNotifyFinishedTask` after the task finished its execution. The `taskId` is an identifier generated by the HWR once the new task message is received. The `parentId` may be an identifier of a task sent by xTasks to the FPGA device, if the host offloaded the task that spawned the new task; or it may be another task identifier generated by the HWR, if there are more nesting levels.

```
typedef uint64_t xtasks_newtask_arg;

typedef struct {
  uint64_t address; ///< Dependence address
  uint8_t  flags;   ///< Dependence flags
} xtasks_newtask_dep;

typedef struct {
  uint8_t flags;       ///< Copy flags
  void   *address;     ///< Copy address
  size_t  size;        ///< Size of the region (in bytes)
  size_t  offset;      ///< Offset not accessed (in bytes)
  size_t  accessedLen; ///< Accessed length (in bytes)
} xtasks_newtask_copy;

typedef struct {
  tasks_task_id        taskId;    ///< Task identifier inside HWR
  xtasks_task_id       parentId;  ///< Parent task identifier
  uint64_t             typeInfo;  ///< Identifier of the task type
  size_t               numArgs;   ///< Number of arguments
  xtasks_newtask_arg  *args;      ///< Arguments array
  size_t               numDeps;   ///< Number of dependences
  xtasks_newtask_dep  *deps;      ///< Dependences array
  size_t               numCopies; ///< Number of copies
  xtasks_newtask_copy *copies;    ///< Copies array
} xtasks_newtask;

xtasks_stat xtasksTryGetNewTask(xtasks_newtask **task);
```

Listing 4.8: xTasks API for FPGA spawned tasks retrieval

xtasksNotifyFinishedTask

The `xtasksNotifyFinishedTask` API notifies the finalization of a task offloaded from the FPGA device to the host runtime. The API declaration is shown in listing 4.9, and its parameters are:

- `parent`. Identifier of the parent task which child task execution has finished. The identifier must be the one returned by `xtasksTryGetNewTask`.
- `id`. Identifier of the task which execution has finished. The identifier must be the one returned by `xtasksTryGetNewTask`.
- Returns `XTASKS_SUCCESS` if the notification has been sent, and returns `XTASKS_ENOENTRY` if the queue to send the notification is full.

Each call to `xtasksNotifyFinishedTask` tries to reserve the needed slots in the SpawnIn queue and write the task information if the space is available.

```
1 xtasks_stat xtasksNotifyFinishedTask(
2   xtasks_task_id const parent, xtasks_task_id const id);
```

Listing 4.9: xTasks API notify the finalization of tasks to HWR

xtasksGetInstrumentData

The `xtasksGetInstrumentData` API has been updated to support the new instrumentation approach based on a buffer for each FPGA task accelerator. The new API declaration can be seen in listing 4.10. The new version takes a `xtasks_acc_handle` instead of a `xtasks_task_handle` (as shown in baseline declaration, listing 2.11). The semantics of the API and the other arguments remain as before.

```
1 xtasks_stat xtasksGetInstrumentData(
2   xtasks_acc_handle const accel,
3   xtasks_ins_event *events,
4   size_t const maxCount);
```

Listing 4.10: New declaration of xTasks API for instrumentation events retrieval

In the new implementation, the xTasks library sends a command to each FPGA task accelerator during the `xtasksInitHWIns` (only if the feature is available). The command format is shown in figure 4.6 and its main goal is to provide the FPGA task accelerators the circular buffer information. The first word (the command header word) contains the command code in the lower 8 bits, and the following 24 bits define the size (in events) of the circular queue for the FPGA task accelerator. The second word contains the memory address of the circular queue.

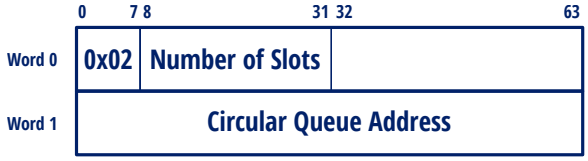


Figure 4.6: Format of setup instrumentation command

The new instrumentation approach reduces the length of the execute task command sent to HWR on calls to `xtasksSubmitTask`. The updated command format is shown in figure 4.7. The format is the same as before (shown in figure 3.10) but removing the word with the instrumentation buffer address in the command header.

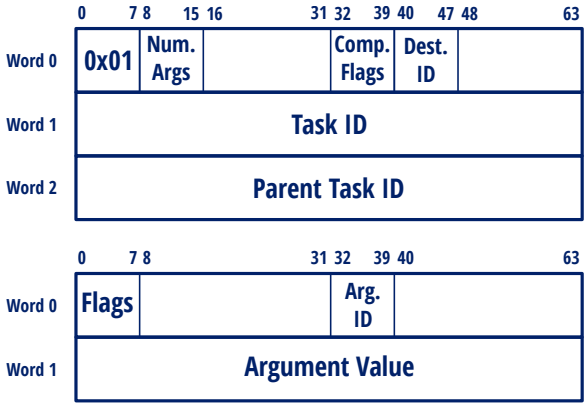


Figure 4.7: Format of execute task command (v2)

4.5.2 New Queues

The communication between the xTasks library and the HWR is done using different communication queues accessible by both parts. This section describes the format and management of the added or modified queues to support the new functionalities. All queues are implemented like the previously available to communicate commands to the FPGA device.

SpawnOut Queue

The SpawnOut queue is intended to hold tasks offloaded from the FPGA device to the host runtime. They are offloaded if the task execution cannot directly be handled by the HWR. It may be due to a non-FPGA architecture task, non-availability of the FPGA task accelerator type, or because the task has data dependences and they cannot be managed by the HWR.

The queue is composed of 1024 words of 64 bits, and it is managed as a circular buffer with a single-producer (HWR) single-consumer (xTasks library). The elements stored in the queue do not have a fixed length as it depends on the number of task arguments, dependences, and copies. Therefore, an element can start at any word of the queue and its length is determined by the information in the first word (head).

The format of an element is shown in figure 4.8. The first four words are the task header. Then, there are `nDeps` words with the dependences information. After that, there are `nCopies` groups of 3 words, each one with the information of one task copy. Finally, there are `nArgs` words with the task arguments.

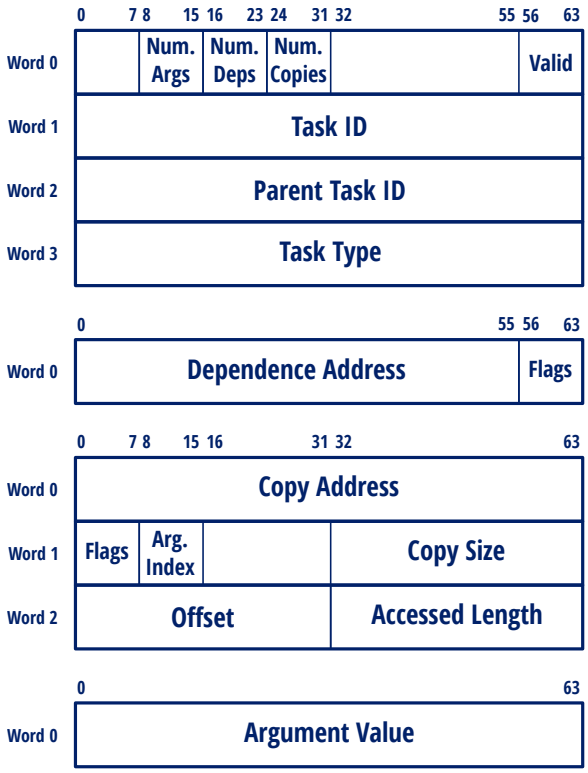


Figure 4.8: Format of elements in the SpawnOut Queue

The elements format is almost the same as the one shown for the new task message (figure 4.3). The differences are the valid bits, which are added in the top bits of the head word, and the `Task ID`, which is the word number 1 of the header. The valid bits mean that the following words in the queue are formatted according to the head information.

SpawnIn Queue

The SpawnIn queue is intended to hold task finalization notifications from host runtime to FPGA device. The queue is composed of 1024 words of 64 bits, and it is managed as a circular buffer with a single-producer (xTasks library) single-consumer (HWR). The elements stored in the queue always use three words: the header, the finished task identifier, and the parent identifier of the finished task. The format of an element is shown in figure 4.9.

The library keeps a count of available slots in the queue. It is decreased in every write to make it coherent. Then, if the number of available words is not enough, it tries to increase the number of available slots looking at the oldest wrote entries and see if they have been invalidated by the HWR.

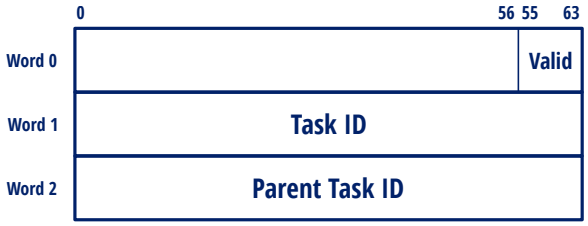


Figure 4.9: Format of elements in the SpawnIn Queue

Instrumentation Buffers

The instrumentation buffers are a set of memory buffers allocated during the `xtasksInitHWIns` with the requested size as it was previously done but with task-level buffers. The requested size defines the number of instrumentation events that the queue of each FPGA task accelerator can hold. Each event occupies 192 bits with the format shown in figure 4.10.

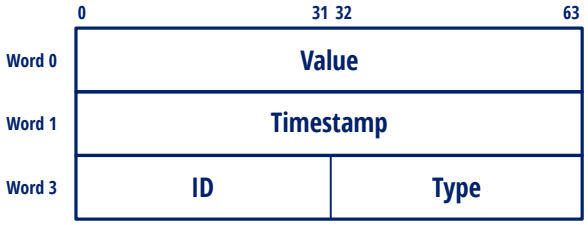


Figure 4.10: Format of instrumentation events in the circular instrumentation buffers

The circular buffer management has been implemented considering that it is a single consumer and single producer buffer. This allows a distributed state of the buffer, where the writer and the reader are implicitly coordinated, and they only need to share the data container. During the initialization, all buffer slots are set to the invalid type (as defined in listing 2.11). Each call to `xtasksGetInstrumentData` copies the number of requested events (or the number of remaining events until the sub-buffer end) from the circular buffer to the buffer provided by the caller. Then, it starts iterating the retrieved events, and if they are valid, the corresponding entry in the circular buffer is set to invalid, which makes it available for the FPGA task accelerator again.

4.6 FPGA Design Support

The support of the new features directly involves the design of the FPGA bitstream. The goal is to extend the capabilities of FPGA devices. Therefore, the components instantiated in the FPGA design must change to handle the new functionalities. The main

parts in the FPGA design are the FPGA task accelerators and the Hardware Runtime. The FPGA task accelerators have been modified as explained in section 4.6.1 to add support for spawning tasks. Also, the HWR has been extended as explained in section 4.6.2 to handle the new messages sent by the FPGA task accelerators.

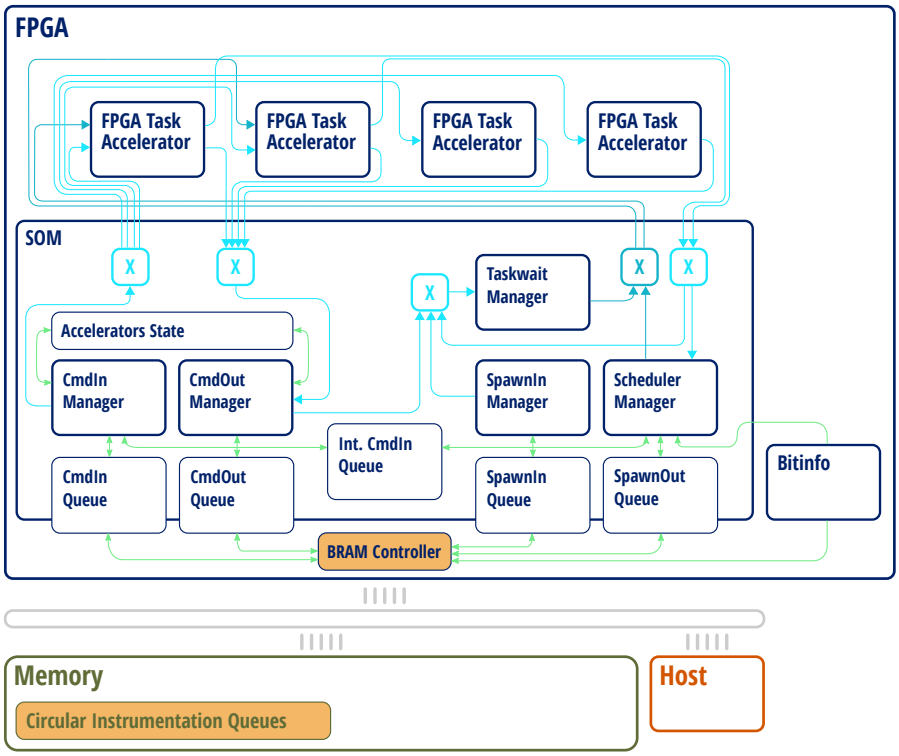


Figure 4.11: FPGA Bitstream design with the extended SOM Hardware Runtime

Figure 4.11 shows the elements in the system, focused in the FPGA part. The figure extends the baseline design shown in figure 3.9. The elements shown in the figure and their interconnections are explained in the following sections.

4.6.1 FPGA Task Accelerators

The FPGA task accelerators have been extended, if needed, to support the new capabilities. AIT checks if the HLS source code generated by Mercurium has the additional ports (`mcxx_eOutPort` and `mcxx_eInPort`) used to communicate the FPGA task accelerator with the HWR. Due to the HLS restrictions, the regular input and output streams cannot be used to send/retrieve the new messages. When the new ports are found, AIT adds the needed adapters to convert the AXI-Stream protocol used in the interconnections between the HWR and the FPGA task accelerators to the handshake protocol used by the new ports. Those adapters are briefly explained in the following points.

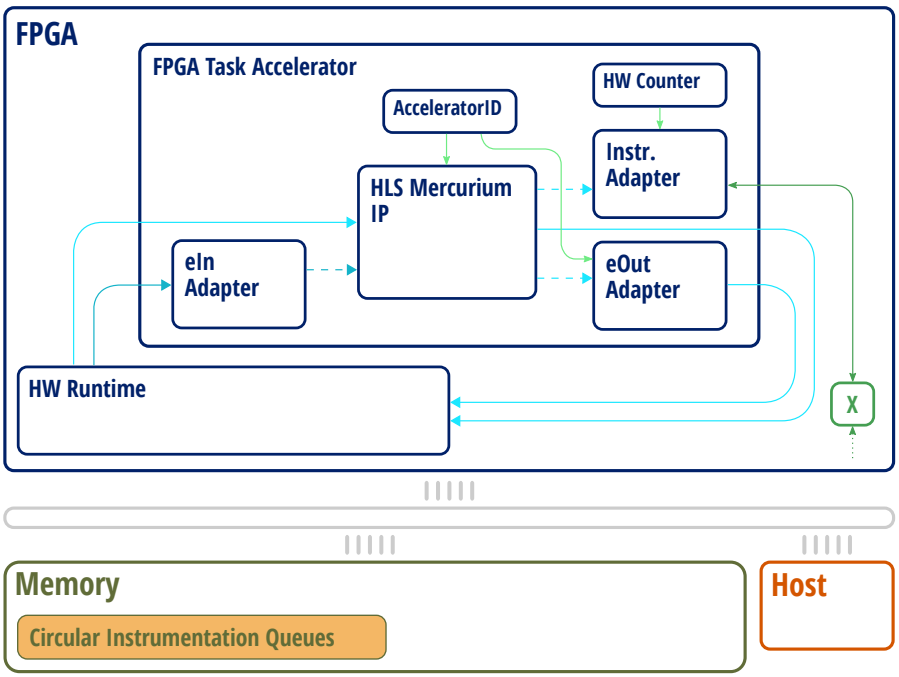


Figure 4.12: Internal structure of FPGA Task Accelerator with task spawn support

The update of FPGA task accelerators using adapters has been used to improve the FPGA instrumentation capabilities (which also were affected as explained in section 4.4.3). An extra adapter has been created to handle the instrumentation events generated by each FPGA task accelerator and write them into the circular instrumentation queue.

eOut Adapter

The eOut Adapter converts the `mcxx_eOutPort` that uses the handshake protocol to a AXI-Stream port. The handshake port has 70 data bits, 1 valid bit, and 1 ready bit. These 70 data bits compress the following AXI-Stream bits: 64 data bits, 5 destination ID bits, and 1 last bit. The valid and ready bits from both protocols can be directly mapped. Finally, the source ID bits of the AXI-Stream are filled with the data read from the accelerator ID constant. Listing A.2 shows the HLS implementation of the adapter.

eIn Adapter

The eIn Adapter converts the AXI-Steam protocol that arrives to the FPGA task accelerator from the HWR into the handshake protocol used by the `mcxx_eInPort` generated by Mercurium. The adapter forwards the 8 bits of data, the valid signal, and the ready signal from one protocol to another. The other signals of AXI-Stream protocol (destination

ID, source ID and last signal) are not used, so they are discarded. Listing A.3 shows the HLS implementation of the adapter.

Instrumentation Adapter

The instrumentation Adapter reads the `mcxx_instrPort` that uses the handshake protocol and writes the received instrumentation events in the circular instrumentation queue. The handshake port has 104 data bits, and 1 type bit. If the type bit is 1, the data bits contain an instrumentation event generated by the FPGA task accelerator. Then, the data is composed by: the instrumentation event value (64 bits), the instrumentation event id (32 bits), and the instrumentation event type (8 bits). If the type bit is 0, the data bits contain the memory address of the instrumentation circular queue and the number of words (64 bits wide) that the queue has. The data is then composed of the memory address (64 bits), the number of elements (24 bits), and other unset bits.

The adapter is the single producer of data in the circular queue. This allows it to know the number of available slots in the circular queue at any time. When there is only one available slot, the adapter writes there a runtime event with a predefined identifier, and the value is the number of events lost. The event value starts at one, and it is incremented every time the adapter reads an event if only one slot is available. Before writing this special event, the adapter checks if the oldest events have been read and increases the number of available slots. In the regular case, when there are more than one available slots, the adapter writes the read event information in the circular buffer with the format shown in figure 4.10.

4.6.2 Hardware Runtime

The SOM implementation of the HWR has been extended with new streams, new modules, and new communication queues to support the new features. The new design is shown in figure 4.11 which extends the design shown in figure 3.9. The figure contains two extra FPGA task accelerators with task spawn capabilities (the two at left) and two regular FPGA task accelerators (the two at right), which do not have creation capabilities.

The interconnection between the HWR and the FPGA task accelerators have been extended with two new AXI-Stream interconnects, one in each direction. The stream from the HWR is only 8 bits wide and is used to acknowledge or respond the messages sent by the FPGA task accelerators. It is connected to the `eIn_Adapter` as shown in figure 4.12 and by the HLS IP block after the protocol conversion. The stream to the HWR is 64 bits wide and is used to communicate the messages from the FPGA task

accelerator when it spawns or synchronizes child tasks. It is connected to `eOut_Adapter` as shown in figure 4.12. They have been kept aside from regular communication streams due to their different purposes/destinations and avoiding starvation between different FPGA task accelerators.

Two new communication queues between the HWR and xTasks library have been added to support the spawn of tasks to the host runtime (`SpawnOut Queue`) and synchronize their finalization (`SpawnIn Queue`). Their format is already explained in section 4.5.2. They are managed as circular queues with a non-fixed number of words for each element. The HWR writes the `SpawnOut Queue` and polls it for invalidated elements when there is not enough space. Also, the HWR polls the `SpawnIn Queue` searching for finished task notifications sent by the host.

Three new modules and one internal queue have been added in the SOM implementation to handle the FPGA spawned tasks (all shown in figure 4.12). The new internal queue is called `Internal CmdIn Queue`. The new modules are called: `Scheduler Manager`, `SpawnIn Manager` and `Taskwait Manager`. Moreover, some updates have been done in already available modules: `CmdIn Manager` and `CmdOut Manager`. They are explained in the following points.

Internal Command In Queue

The internal command in queue, abbreviated as `Int. CmdIn Queue` in figure 4.12, is intended to hold execute task commands for tasks spawned inside the FPGA.

The queue is composed of 1024 words of 64 bits, which are divided into 16 sub-queues (the maximum number of FPGA task accelerators supported in an FPGA design) of 64 elements. Each sub-queue is managed as a circular buffer with a single-producer (`Scheduler Manager`) single-consumer (`CmdIn Manager`). The elements stored in the queue do not have a fixed length as it depends on the number of task arguments. Therefore, an element can start at any word of the queue and its length is determined by the information in the first word (head).

The format of an element is the same used in the execute task command of `CmdIn Queue`. The format of the execute task command has been updated with the new capabilities, and its format is shown in figure 4.7.

Scheduler Manager

The `Scheduler Manager` is an IP block developed in C++ using HLS tools. Its external interface is shown in figure 4.13, where the different ports and protocols to communicate the module with the other components are detailed. Those ports are connected as shown in figure 4.11 and they are:

- `In Stream`. AXI-Stream port used by all FPGA task accelerators to send new task messages.
- `Bitinfo`. BRAM port to access the Bitinfo memory, which stores the configuration of FPGA task accelerators in the bitstream. This port is only read.
- `Int. CmdIn Queue`. BRAM port to access the `Int. CmdIn Queue`, which buffers the execute task commands sent to the `CmdIn Manager`. This memory is mainly written but also read to look for invalidated entries.
- `SpawnOut Queue`. BRAM port to access the `SpawnOut Queue`, which is used to offload tasks to the host runtime. This memory is mainly written but also read to look for invalidated entries.
- `Out Stream`. AXI-Stream port to send the acknowledge messages to FPGA task accelerators.

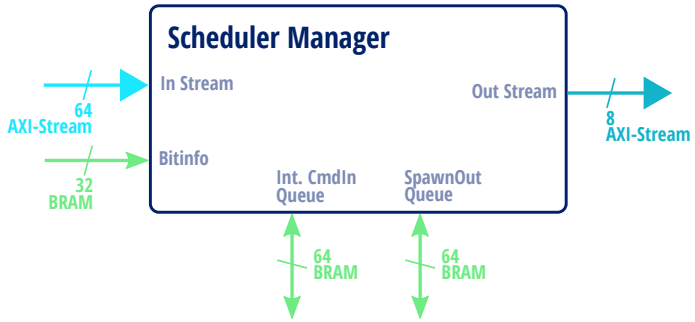


Figure 4.13: External interface of Scheduler Manager

The purpose of the `Scheduler Manager` is scheduling the tasks spawned by the FPGA task accelerators. The SOM implementation routes all new task messages to this IP block, regardless they have or not data dependences (destination ID of the AXI-Stream will be `HWR_DEPS_ID` or `HWR_SCHED_ID`, respectively). Then, the `Scheduler Manager` forwards the task information, with the data dependences, to the host runtime for its handling.

The available types and instances of FPGA task accelerators are read during the module initialization, triggered by the reset signal. The module starts reading words of the Bitinfo

memory (which is formatted as explained in section 3.3.1) until the end of xtasks.config file is reached. The retrieved information is used to match the tasks with the different FPGA task accelerators based on their type word and then decide where a given task will be executed. For each FPGA task accelerator type, the module stores: the type, the first ID of such type, the number of IDs of such type, and the last ID where a task has been sent. With this information, the `Scheduler Manager` implements a round-robin policy. However, the policy could be updated as all this information is packed in a struct, and the stage of deciding where the task will be executed is isolated. Then, both parts could be easily modified in the source code.

The task information arrives through `in stream` encoded as explained in section 4.3.3 and shown in figure 4.3. First, the `Scheduler Manager` assigns a new identifier to the task, which is just an increasing odd number. This identifier is needed to keep track of the task among its life, and it is an odd number to distinguish tasks spawned in the host against the ones in the FPGA. Following, the module checks if the new task has any data dependences or if it is not an FPGA task. If it is, the task information is written into the `SpawnOut Queue`. Intently, this queue has almost the same format of the data in a new stream message, which allows an easy forward of the data. On the other hand, if the task does not have data dependences and is an FPGA task, the module searches the received type in the types list (fulfilled during the initialization). Then, it selects the FPGA task accelerator where the task will be executed and updates the last ID information of the entry. Once the task has been scheduled to an FPGA task accelerator, the task information is written into the `Int. CmdIn Queue` following the queue format. Finally, the module sends the acknowledge message to the FPGA task accelerator that sent the message.

The acknowledge message purpose is to allow the manager rejecting the messages to ensure progress when several FPGA task accelerators create tasks concurrently. The rejection of messages causes the sender to re-send the message. However, other FPGA task accelerator may send their own messages between one rejected messages and its re-sending. This mechanism avoids blocking the manager when processing a message if the output queue, where the current task has to be written, is full. Blocking the `In Stream` until the output queue has enough space may end in deadlocks when different FPGA task accelerators create tasks concurrently.

The format differences between a new task message and an element of the `Int. CmdIn Queue` are managed by the `Scheduler Manager`. The main differences are:

- The command code (bits 0-7 of header word) are fixed to `0x01`, which is the command code for an execute task command.

- The computation flags are fixed to `0x01`, which enables the kernel execution.
- The destination ID is fixed to `HWR_CMDOUT_ID`, which makes the FPGA task accelerator notify the `CmdOut Manager` the finalization of the task.
- The task ID is the new identifier generated by the IP block.
- The parent task ID is the identifier received in the new task message.
- The argument flags are set to `0x31` by default (in and out wrapper copies enabled, and private mode). However, the IP block checks the copies directionality received in the new task message and updates the flags of the argument that the copy refers too.

SpawnIn Manager

The `SpawnIn Manager` is an IP block developed in C++ using HLS tools. Its external interface is shown in figure 4.14, where the different ports and protocols to communicate the module with the other components are detailed. Those ports are connected as shown in figure 4.11 and they are:

- `SpawnIn Queue`. BRAM port to access the SpawnIn Queue, which is used to notify the finalization of tasks offloaded using the `SpawnOut Queue`. This memory is mainly read but also written to invalidate entries.
- `Out Stream`. AXI-Stream port to send the finalization notification to other modules.

Figure 4.14: External interface of SpawnIn Manager

The purpose of the `SpawnIn Manager` is to poll the `SpawnIn Queue` for new elements, read them and invalidate them. The module generates an AXI-Stream message for the `Taskwait Manager` for every element found in the queue. The format of those messages is shown in figure 4.15. The current implementation always sets the number of child tasks to 1 (bits 0-31 of the first package) because the host runtime inserts an element for each finished task. However, the message syntax allows merging some of those messages in the future. The finalization code is `0x10`, and it is set in bits 32-39 of the first package.

Finally, the second package contains the task identifier of the parent which child task has finished.

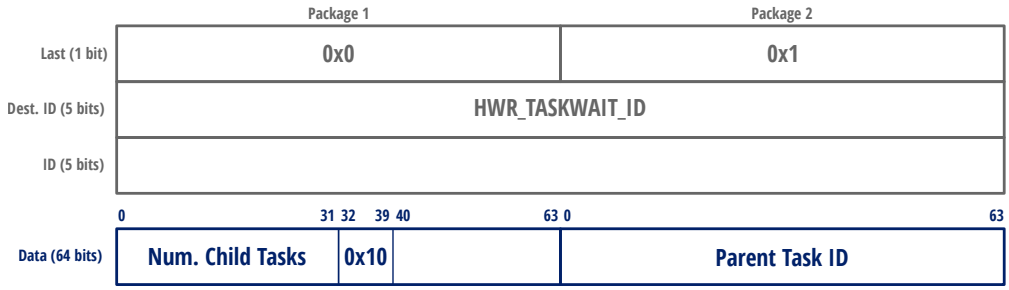


Figure 4.15: Format of finish message for Taskwait Manager

Taskwait Manager

The `Taskwait Manager` is an IP block developed in C++ using HLS tools. Its external interface is shown in figure 4.16, where the different ports and protocols to communicate the module with the other components are detailed. Those ports are connected as shown in figure 4.11 and they are:

- `In Stream`. AXI-Stream port used by all FPGA task accelerators to send block messages, and by other HWR modules to send finish messages.
- `Out Stream`. AXI-Stream port to send the acknowledge of block messages to FPGA task accelerators.

Figure 4.16: External interface of Taskwait Manager

The purpose of the `Taskwait Manager` is keeping track of the number of child tasks per parent, and sending a wake-up message when all child tasks have finished. To this end, the module has an internal table with the number of tasks for each identifier. The module can receive two types of messages in the input stream: block messages (format shown in figure 4.4) and finish messages (format shown in figure 4.15). Whenever it receives a new message, it looks for the task identifier in the internal table and increases (block messages) or decreases (finish messages) the balance with the notified number of tasks. The balance may be positive or negative, depending on the messages ordering. If the identifier is not found, the IP block initializes a new entry with balance zero in the

table and proceeds with the regular increment/decrement. If the balance becomes 0 after the update, the module sends a wake-up message to the FPGA task accelerator that sent the block message for that identifier, and it invalidates the entry in the internal table.

CmdIn Manager

The `CmdIn Manager` has been extended to retrieve commands from the `Int. CmdIn Queue` and forward them to the FPGA task accelerators. The new module version tries to retrieve a command as usual from the `CmdIn Queue` and, if there is no available command, it looks into the `Int. CmdIn Queue`. Besides, the stat written into `Accelerators State` includes the information about the origin of the running command, that is later used by the `CmdOut Manager`. The new external interface is shown in figure 4.17, where the different ports and protocols to communicate the module with the other components are detailed. The new port is connected as shown in figure 4.11 and it is:

- `Int. CmdIn Queue`. BRAM port to access the Int. CmdIn Queue, which buffers the execute task commands sent by the `Scheduler Manager`. This memory is mainly read but also written to invalidate entries.

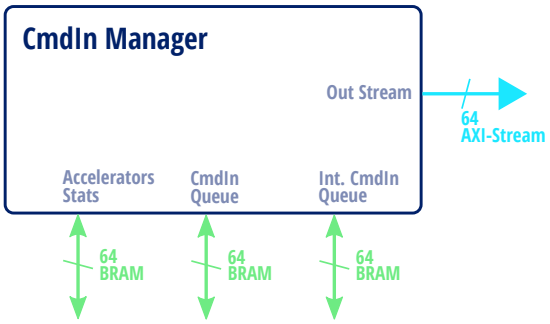


Figure 4.17: External interface of CmdIn Manager (v2)

CmdOut Manager

The `CmdOut Manager` has been extended to notify the finalization of tasks to the `Taskwait Manager`. The new module version gets the parent task identifier received in the input stream and sends it in the finish message (with the format shown in figure 4.15) if the `Accelerators State` reports that the command was send from the `Int. CmdIn Queue` The new external interface is shown in figure 4.18, where the different ports and protocols to communicate the module with the other components are detailed. The new port is connected as shown in figure 4.11 and it is:

- `Out Stream`. AXI-Stream port to send finish messages to `Taskwait Manager`.

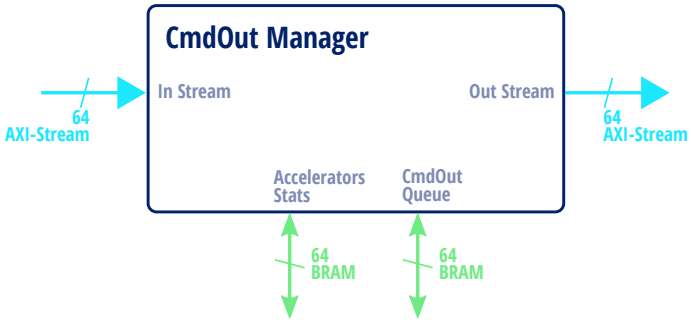


Figure 4.18: External interface of CmdOut Manager (v2)

4.7 Evaluation

This section presents the evaluation results of the new capabilities. It has been done in a Xilinx Zynq ZCU102 board and considering different benchmarks. For each benchmark, different approaches have been developed to manage the kernel tasks that really compute the solution. Some of those approaches do not use the proposed extensions to obtain the baseline potential and compare it with the extended mode.

Section 4.7.1 describes the experimental setup used in the evaluation, section 4.7.2 describes the FPGA resources and power consumption, section 4.7.3 describes the performance results of the synthetic benchmark used to see the limits and overheads of the extension, and section 4.7.4 describes the performance of real benchmarks.

4.7.1 Experimental Setup

The evaluation of the modifications has been done in a real environment and with real executions. The tools used to generate the application bitstreams and binaries are: Vivado Design Suite 2017.3, GNU C/C++ Compiler 6.2.0, and PetaLinux Tools 2019.1. The modifications have been developed on top of OmpSs@FPGA release 1.4.0.

The following points detail the machine and benchmarks used during the evaluation. The benchmarks have been chosen by availability and by their different characteristics (computation/memory intensity, regular or heterogeneous task parallelism). Therefore, the evaluation is done over different scenarios that better shown the overall extension capabilities.

ZCU102

All executions have been done in a Xilinx Zynq UltraScale+ MPSoC ZCU102 [@67]. The System on Chip (SoC) is composed of 4 ARM Cortex-A53 cores, that run at 1.1 GHz, a Xilinx ZU9EG FPGA and a main DDR4 memory of 4 GiB. The board is booted using the Ubuntu Linux 16.04 operating system.

Synthetic Benchmark

Listing 4.11 shows the pseudo-code of the synthetic benchmark. It updates all elements of an array in parallel using a block decomposition. The benchmark execution time can be analyzed when changing the task size but keeping the total work constant, as the array length and the block length are parameterized. Different configurations with 1, 2, 4, 8, and 15 instances of `update_array_fpga` accelerator inside the FPGA have been used. This pattern allows an exploration of the runtime limits (it is embarrassingly parallel) with different task granularities. Although the board supports higher frequencies, all bitstreams have been generated with FPGA task accelerators running at 100 MHz for the sake of comparison.

```
1  #pragma omp target device(fpga) num_instances(15) copy_inout([BSIZE]array)
2  #pragma omp task
3  void update_array_fpga(int *array, int BSIZE, int val) {
4    for (int i=0; i<BSIZE; ++i) array[i] += val;
5  }
6  #pragma omp target device(fpga) copy_inout([SIZE]array)
7  #pragma omp task
8  void update_array_blocked(int *array, int BSIZE, int SIZE) {
9    for (int i=0; i<SIZE; i+=BSIZE) {
10     update_array_fpga(array+i, 2020);
11   }
12   #pragma omp taskwait
13 }
14 int main(...) {
15   int array[NUM_BLOCKS*BSIZE];
16   update_array_blocked(array, BSIZE, NUM_BLOCKS*BSIZE);
17   #pragma omp taskwait
18 }
```

Listing 4.11: Synthetic benchmark pseudo-code

Matrix Multiply

Matrix Multiply implements the multiplication of two matrices using a blocked algorithm with the accumulation of the result in a third matrix. The implementation has been done using a simple C implementation where the loops in the kernel task follow the k-i-j order as shown in pseudo-code of listing 4.12. The k-i-j order has a better access pattern to the BRAMs where the matrices are stored and results in a better performance. To test the extensions, an FPGA task accelerator that spawns all tasks for the kernel task accelerators has been added. Then, the host has to offload a single task into the FPGA device to launch the multiplication of the two matrices regardless their size. Three different configurations have been used depending on the block size of the kernel task (`BSIZE`). The instances number of `matmulBlock` and the initiation interval (II) of the innermost loop are shown in table 4.1 for the 3 configurations. The kernel task accelerators always run at 300 MHz, and the matrices used in all executions contain 4096x4096 elements.

```
1  #pragma omp target device(fpga) copy_in([BSIZE*BSZIE]a, [BSIZE*BSIZE]b) \
2    copy_inout([BSIZE*BSIZE]c)
3  #pragma omp task
4  void matmulBlock(const float *a, const float *b, float *c) {
5    for (int k=0; k<BSIZE; ++k) {
6      for (int i=0; i<BSIZE; ++i) {
7        for (int j=0; j<BSIZE; ++j) {
8          #pragma HLS pipeline II=LOOP_II
9          c[i*BSIZE + j] += a[i*BSIZE + k] * b[k*BSIZE + j];
10       }
11     }
12   }
13 }
```

Listing 4.12: Matrix Multiply pseudo-code

Block Size	Num. kernel accels	Loop II
64x64	7	1
128x128	3	1
256x256	3	2

Table 4.1: FPGA configurations for Matrix Multiply benchmark

N-Body

N-Body is a simulation among time of N physical bodies (particles) in a space that attract between them as a result of their mass. The implementation uses two main kernel

tasks that define the kernel task accelerators in the FPGA device: `calculate_forces_block`, which updates a block of particles considering the particles of another block, and `update_positions_block`, which updates the position of the particles in a block based on the calculated forces. Using these two kernel tasks, two more tasks that also can become FPGA task accelerators have been defined: `calculate_all_forces`, which iterates over all combinations of particle blocks and calls to `calculate_forces_block`, and `update_all_positions`, which iterates over all particle blocks and calls `update_positions_block`. Finally, a top level task called `solve_nbody` have beed defined. It spawns `calculate_all_forces` and `update_all_positions` tasks and it also can become an FPGA task accelerator. The pseudo-code of the tasks organization defined in top of kernel tasks is shown in listing 4.13.

```
1  void calculate_forces_block(forceBlock1, partBlock1, partBlock2);
2  void update_positions_block(forceBlock, partBlock);
3
4  void calculate_all_forces(forces, particles, numBlocks) {
5    for i in {0..numBlocks}
6      for i in {0..numBlocks}
7        calculate_forces_block(forces[j*BS],
8            particles[j*BS], particles[i*BS])
9  }
10
11 void update_all_positions(forces, particles, numBlks) {
12   for i in {0..numBlocks}
13     update_positions_block(forces[j*BS], particles[j*BS])
14 }
15
16 void solve_nbody(forces, particles, numBlocks, numTimesteps) {
17   for t in {0..numTimesteps} {
18     calculate_all_forces(forces, particles, numBlocks)
19     update_all_positions(forces, particles, numBlocks)
20   }
21 }
```

Listing 4.13: N-Body pseudo-code

Table 4.2 3 different configurations depending on the block size of kernel tasks. The FPGA task accelerators run at 250 MHz, and all executions are simulating 32 768 particles during 16 timesteps.

Block Size	Num. calculate forces accels	Parallel particles	Num. update positions accels
128	4	8	1
256	3	10	1
512	1	10	1

Table 4.2: FPGA configurations for N-Body benchmark

Cholesky Factorization

Cholesky implements the cholesky factorization of a matrix using a blocked algorithm. The implementation has 4 kernel tasks: `potrf`, `trsm`, `gemm` and `syrk`. The first one, `potrf`, has an execution pattern that makes the FPGA task accelerator in the FPGA device either consume a lot of resources or be very slow. Therefore, two approaches have been defined: *full*, which contains all kernel task accelerators in the FPGA design, and *mixed*, which implements the `potrf` kernel task in the SMP threads and the rest of kernel tasks in the FPGA design as task accelerators. The *full* approaches contain a non-optimized `potrf` accelerator that consumes few FPGA resources and the *mixed* approaches use the OpenBLAS library to implement the `potrf` task in the host processor. For both approaches and for the three considered block sizes, the `num_instances` of each kernel task accelerators are shown in table 4.3. In all cases, the FPGA task accelerators run at 250 MHz.

As can be seen in table 4.3, the FPGA device resources freed by the `potrf` accelerator in *mixed* approach are used to increase the number of instances of remaining kernel task accelerators in comparison to *full*. For the 64x64 and 128x128 block sizes, the number of `gemm` instances, which implements the Matrix Multiply, has been increased as it is the most used. For the 32x32 block size, the number of `trsm` instances has been increased as there are already 6 `gemm` accelerators and the performance gain was better with an extra `trsm` accelerator.

Block Size	Approach	Num. potrf accels	Num. trsm accels	Num. gemm accels	Num. syrk accels
32x32	*full*	1	2	6	1
	mixed	0	3	6	1
64x64	*full*	1	1	3	1
	mixed	0	1	4	1
128x128	*full*	1	1	1	1
	mixed	0	1	2	1

Table 4.3: FPGA configurations for Cholesky benchmark with *full* and *mixed* approaches

4.7.2 Resources Utilization and Power Consumption

Table 4.4 shows a summary of available resources in the FPGA chip of ZCU102 SoC, the power consumption of each A53 core reported by Vivado, and the resources utilization and power consumption of the implemented IP blocks as reported by Vivado. The resource utilization values of SOM show that the manager uses a tiny part of the FPGA

resources. The same report for SOM with the proposed extensions in this chapter shows that the new IP block uses some more resources, which are needed to support the new features. Nevertheless, it still uses a small portion of total resources. In contrast, the power estimation of the extended SOM is 5x the baseline SOM power due to the 3 extra queues ($\sim$ 20 mW for each), the new IP blocks inside the manager ($\sim$ 50 mW) and the extra interconnections ($\sim$ 50 mW). Finally, resource usage and power consumption of the new FPGA task accelerators with task spawn capabilities (*Acc. Spawn*) are also a small portion of the total budget. This leaves the bulk of FPGA resources available to implement application kernels.

Name	BRAM	DSP	FF	LUT	Power
ZCU102	912	2 520	548 160	274 080	-
ARM A53 Core	-	-	-	-	800 mW
SOM	10 (1 %)	0 (0 %)	1 350 (0.2 %)	3 929 (1.4 %)	40 mW
Ext. SOM	25 (2.7 %)	0 (0 %)	2 967 (0.5 %)	8 796 (3.2 %)	200 mW
Acc. Spawn	3 (0.3 %)	0 (0 %)	2 225 (0.4 %)	4 008 (1.4 %)	30 mW

Table 4.4: Resources utilization and power estimation in ZCU102

Host SMP threads are free when the FPGA device creates the tasks directly targeting itself. In contrast, they are busy when the host SMP threads create the FPGA tasks, offloading the tasks and retrieving the finalization messages to synchronize the executions. With the new capabilities, a small task accelerator in the FPGA that consumes $\sim$ 30mW replaces the general-purpose ARM cores that consume 800 mW each (3.2 W in total). This reduces the power consumption of the application or increases the computational capabilities because host cores can be used to execute useful application code instead of runtime management code. Based on power estimation reported by Vivado and shown in table 4.4, the implementation of an application with the new features reduces the power consumption of the baseline version by 2.21 W (switching off 3 host threads and considering the extra power used in the FPGA design).

4.7.3 Scalability limits and overheads

Figure 4.19 shows the average execution time of the synthetic benchmark (y-axis) when increasing the number of instances of `update_array_fpga` task accelerator (x-axis). The evolution is shown for two task spawn approaches (*cHOST* and *cFPGA*) and different block sizes (values are shown in each label). The total amount of application work remains constant for all the executions. Thereby, the number of array blocks is increased

when the block size is decreased. In contrast, the number of array blocks is constant regardless of the number of FPGA task accelerators. The cHOST and cFPGA approaches are equivalent in application terms, and the only difference between them is where the tasks are spawned.

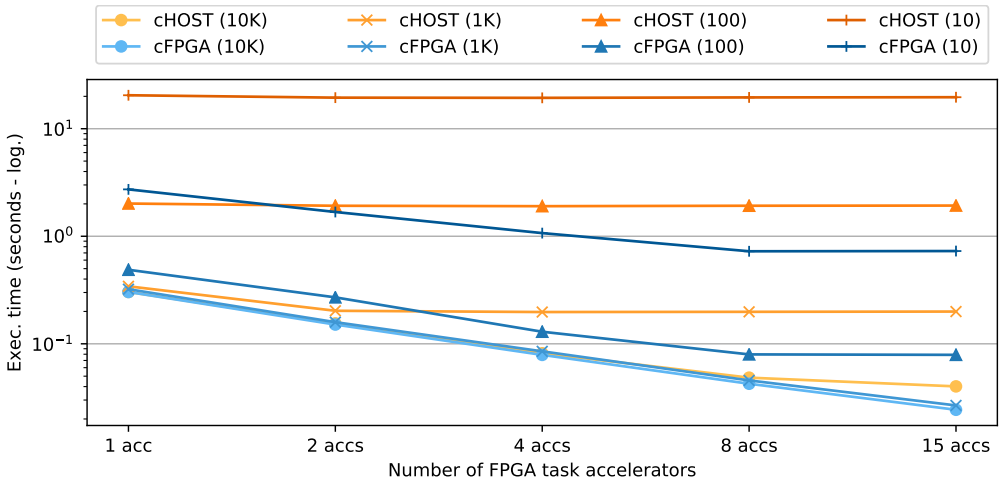


Figure 4.19: Synthetic benchmark execution time with different configurations

The ideal behavior would be that the execution time proportionally decreases when the number of FPGA task accelerators increases. This happen in *cFPGA (10K)*, *cFPGA (1K)*, and almost in *cHOST (10K)*. In contrast, the execution time remain constant in *cHOST (1K)*, *cHOST (100)* and *cHOST (10)*. In those task granularities, the host runtime cannot feed the FPGA device due to the runtime overheads in the task spawn and the communication latency of offload tasks. Also, *cFPGA (100)* and *cFPGA (10)* only scale up to 8 FPGA task accelerators. After that, the execution time does not improve with the larger amount of execution resources. For both task spawn approaches, the tasks are executed faster than created when the time stops decreasing.

Results show that the creation and management of FPGA tasks directly inside the FPGA are faster than when done from the host. Moreover, performance gain increases with small task sizes and/or large numbers of FPGA task accelerators. The better execution time is because of the lower task spawn and task managing overheads in *cFPGA* compared to *cHost*. In fact, *cFPGA* is able to obtain the best absolute performance using two different block sizes, while the *cHost* approach needs a careful size tuning to obtain the sweet point that delivers a performance that approaches but not reaches the ideal one. Consequently, FPGA creation can discover more parallelism, increase the resources utilization, and reduce overall execution time while simplifying programmability.

Figure 4.20 shows the same results of figure 4.19 but dividing the execution time between the number of FPGA tasks created at each block size. Therefore it shows the wall clock

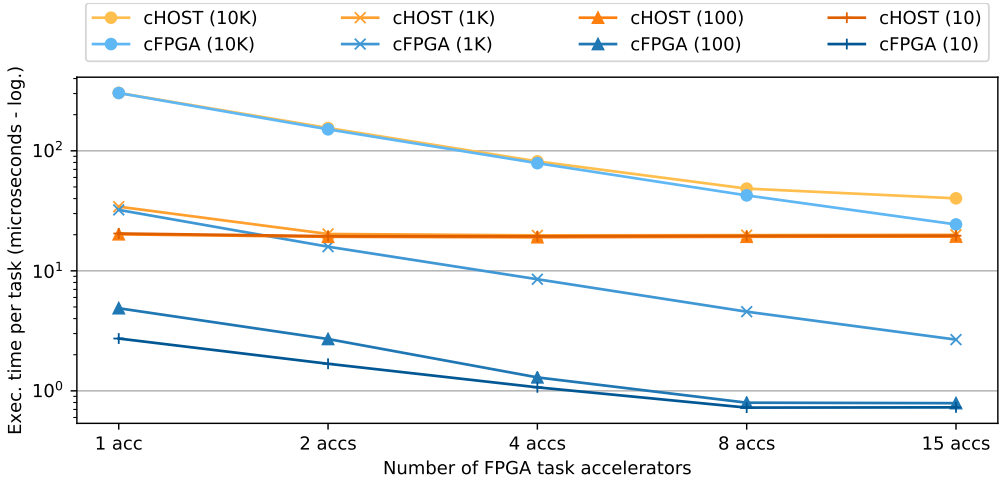


Figure 4.20: Synthetic benchmark time per task with different configurations

time per task in the y-axis. This way, the minimum task size that *cHost* and *cFPGA* can manage is clearly shown.

In the largest block size (10K), *cHost* and *cFPGA* behave almost identically except when a large number of FPGA task accelerators is used. In this block size, the time per task decreases with the major number of FPGA task accelerators. In the others, *cHost* stands at $\sim$ 19.5 μs per task meanwhile *cFPGA* keep decreasing (at least until 8 FPGA task accelerators). As time remains constant regardless of the number of executors, the entire execution time is spent creating and managing tasks. Therefore, $\sim$ 19.5 μs approximates the mean task creation and management time for a task in the host runtime. The same approximation can be made for *cFPGA* with the smallest block size (10) and 8-15 FPGA task accelerators. In this case, the mean task creation and management time is $\sim$ 0.73 μs, which is a 26.7x speedup. In addition, the *cFPGA* times will decrease proportionally with a higher frequency, and the *cHost* times will remain almost constant (due to the runtime management overheads). This will make a higher dent in the potential performance.

4.7.4 Real benchmarks

Matrix Multiply

Figure 4.21 shows the average GFLOPS (y-axis) obtained by different approaches and with three different block sizes of kernel tasks (x-axis). The considered approaches are:

- *OpenBLAS*. Same application taskification as the other approaches, but using the OpenBLAS [@69] library to compute the block-by-block kernel task.

- *cHost*. Creates the tasks for the kernel task accelerators in the host. This approach is the normal master-slave approach previously used.
- *cFPGA*. Creates the tasks for the kernel task accelerators using one extra FPGA accelerator. This approach breaks the master-slave model using the proposed extensions.
- *cHost (warm-up)*. Same as *cHost* but with a previous warm-up execution that pre-fetches the data into the FPGA memory space.
- *cFPGA (warm-up)*. Same as *cFPGA* but with a previous warm-up execution that pre-fetches the data into the FPGA memory space.

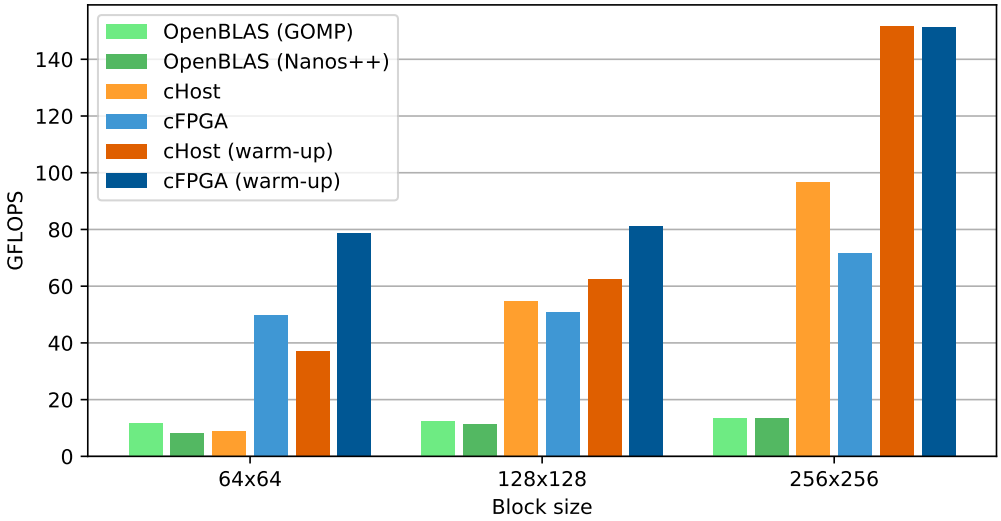
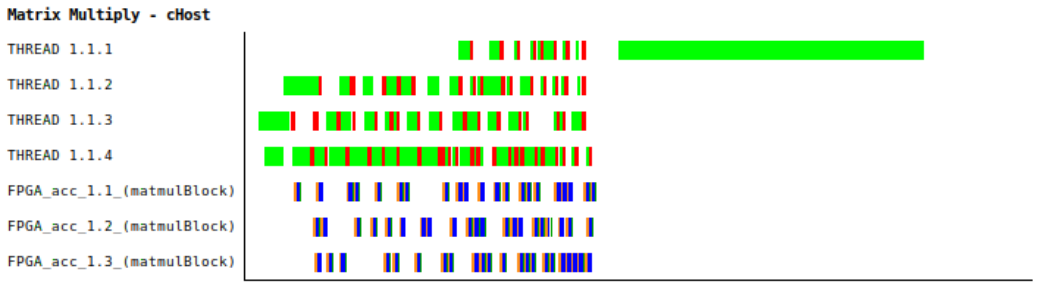


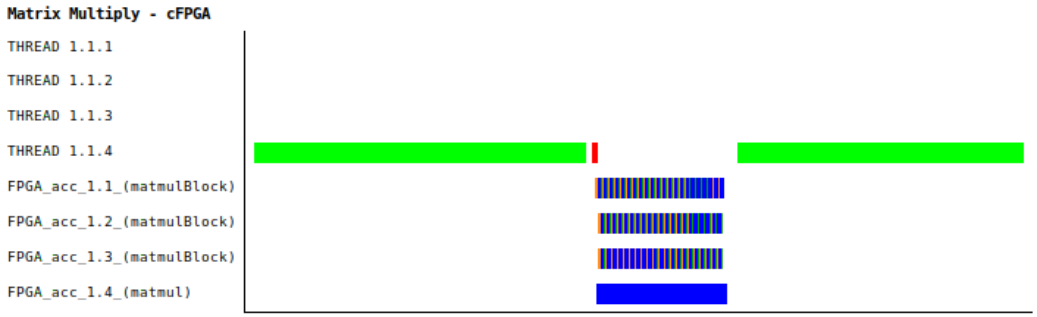
Figure 4.21: Matrix Multiply GFLOPS with different approaches and block sizes

Performance of fine-grain tasks (64x64 block size) in figure 4.21 shows that *cFPGA* outperforms the *cHost*. The considerable performance gap is due to the smaller overheads in the task creation like in the synthetic benchmark. However, the data movements between the host address space and the FPGA address space also enlarge the overheads of *cHost* in this case. The runtime has to check and satisfy the task data requirements before offloading it into the FPGA. This check has only to be done once in the application implementation used by *cFPGA* as just one task is offloaded to the FPGA. The runtime can pre-fetch the data into the FPGA address space to avoid those overheads. This can be achieved using previous tasks that already require the movement of the data into the FPGA address space or some hint for the runtime places in the application code. The same performance of *cHost* and *cFPGA* but using previous warm-up tasks is shown in *cHost (warm-up)* and *cFPGA (warm-up)*. As can be seen, both approaches improve the performance, but *cHost (warm-up)* is more benefited than *cFPGA (warm-up)* because the first has more tasks to handle.

The performance of coarse-grain tasks (256x256 block size) in figure 4.21 shows the opposite behavior of fine-grain tasks and is *cHost* that gets better performance than *cFPGA*. Despite the small overheads of *cFPGA*, the approach used to implement the Matrix Multiply requires a large data movement for the full matrices before the unique FPGA task can be offloaded. In contrast, the *cHost* approach allows doing the data movements block by block and offloading tasks to the FPGA in the middle. These movements of data between the address spaces are not relevant in *cHost (warm-up)* and *cFPGA (warm-up)*, and both approaches obtain the same performance. The results for the coarse-grain tasks with the warm-up tasks show that *cFPGA (warm-up)* can achieve the same performance of the baseline under configurations that use huge accelerators.



(a) cHost



(b) cFPGA

Figure 4.22: Execution traces of Matrix Multiply with 3 accels of 128x128 block size and 512x512 matrix size

Figure 4.22 shows two execution traces of Matrix Multiply for the 128x128 block size, figure 4.22a for *cHost* and figure 4.22b for *cFPGA*. The x-axes represent the time, and the y-axes the different computational resources (each line). The different colors in each line represent what is happening in that resource at that time: light-green is for data movements between address spaces, red is for FPGA task offloading, dark blue is for the execution of tasks in the FPGA task accelerators, and orange/dark-green are for input/output data movements between main memory and FPGA BRAMs. Both traces are scaled to represent the same period of time in the x-axis. Note that figure 4.22b has

one more compute place (one line) than figure 4.22a at the bottom, which represents the FPGA accelerator implementing the full Matrix Multiply task. This pair of executions has the same performance gap as described for *cHost* and *cFPGA* in the 256x256 block size and it can be seen in figure 4.21.

Traces in figure 4.22 clearly show that:

1. The offloading of tasks into the FPGA is delayed in *cFPGA* due to the sequential large data movement.
2. The data movements in *cHost* are done in parallel with the execution of tasks in the FPGA accelerators.
3. The *cFPGA* approach can occupy the FPGA accelerators much better than *cHost*.
4. The *cHost* is not able to feed the FPGA accelerators fast enough to make the most of them.
5. The *cFPGA* only uses one host thread to do the data movements and offload the FPGA task. The rest are idle all the execution time and could be switch off with a power-saving around 2W.

N-Body

Figure 4.23 shows the average execution time of 3 executions (y-axis, truncated at 100 s) obtained by each approach in the 3 different block sizes of kernel tasks (x-axis). The considered approaches are:

- *cHost*. Creates the tasks for the base kernel accelerators in the host. `solve_nbody`, `calculate_all_forces` and `update_all_positions` are executed in the host. This is the baseline approach without using the proposed extensions.
- *cFPGA calc*. Creates the tasks for `calculate_forces_blocks` using an extra FPGA accelerator. `calculate_all_forces` is an FPGA accelerator but `solve_nbody` and `update_all_positions` are executed in the host.
- *cFPGA calc + updt*. Creates the tasks for the base kernel accelerators using an extra FPGA accelerator. `calculate_all_forces` and `update_all_positions` are FPGA accelerators but `solve_nbody` is executed in the host.
- *cFPGA solve*. Same as *cFPGA calc + updt* but also uses an extra FPGA task accelerator for `solve_nbody` task.

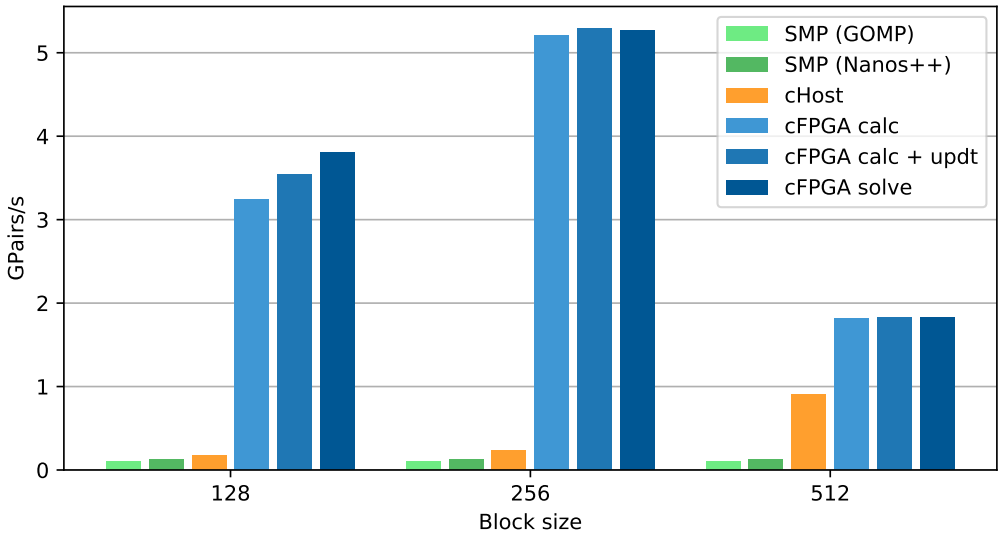


Figure 4.23: N-Body GPairs/s with different approaches and block sizes

The *cHost* times in figure 4.23 show the complexity of managing such small FPGA tasks from the host size. The only possible block size to get a reasonable overall execution time is 512, which has long kernel task accelerators ($\sim$ 140 μs for `calculate_forces_block` accelerators and $\sim$ 188 μs for `update_positions_block` accelerators). When using a block size of 256 or 128 particles, the *cHost* time increases as there is a major number of tasks to manage and they are executed faster. The `calculate_forces_block` tasks take $\sim$ 36 μs and $\sim$ 77 μs for `update_positions_block`, with 256 block size; and $\sim$ 17 μs and $\sim$ 27 μs for both tasks with 128 block size, respectively. Note that the total amount of work is the same in the three configurations, and the only difference is the task granularity used to execute the application. Despite the bad performance of *cHost*, the execution time is even larger when the kernel tasks target the SMP host threads instead of the FPGA device, and therefore the results are not shown in figure 4.23.

The shorter execution times of all *cFPGA* approaches are due to the better utilization of kernel task accelerators thanks to the smaller task creation overheads. Moreover, performing synchronization between stages inside the FPGA device removes host-FPGA latencies. Consequently, the *cFPGA solve* approach reduces the execution time of *cFPGA calc* and *cFPGA calc + updt* as more task synchronizations are done inside the FPGA device. In addition, *cFPGA solve* shows the capability of nesting multiple levels of FPGA tasks with the programming model extension.

The best overall configuration is the 256 block size with the *cFPGA solve* approach. Due to a high resources consumption, the 512 configuration can only fit 1 `calculate_forces_block` accelerator. Despite the better data locality, the execution of a task in the 512 configuration is not faster than the execution in three task accelerators of

the 256 configuration, which has a better ratio between execution time and consumed FPGA resources. In the 128 block size, four `calculate_forces_block` accelerators fit into the design, but the time spent doing memory accesses by task accelerators in the FPGA increases too much, which decreases the overall performance. In all the configurations, the `calculate_all_forces` and `update_all_positions` accelerators can feed the kernel task accelerators without problems. Indeed, the fastest kernel task accelerator, `calculate_forces_block` with 128 block size lasts $\sim$ 17 μs meanwhile the creation FPGA accelerators can spawn one task every $\sim$ 0.45 μs.

Cholesky Factorization

The same *cHost* and *cFPGA* approaches as in the other benchmarks are considered. Those approaches result in 4 combinations that mix *mixed* and *full* FPGA target devices and the device where tasks are created (*cHost* and *cFPGA*). The *cFPGA mixed* approach shows the programmability and performance improvements of the proposal when creating tasks from the FPGA device but using the general-purpose processor to execute some application code not suitable for the FPGA device. Also, results contain the *OpenBLAS* approach that uses the same application taskification as the other approaches but using the OpenBLAS [@69] library to compute the block-by-block kernel task in the host.

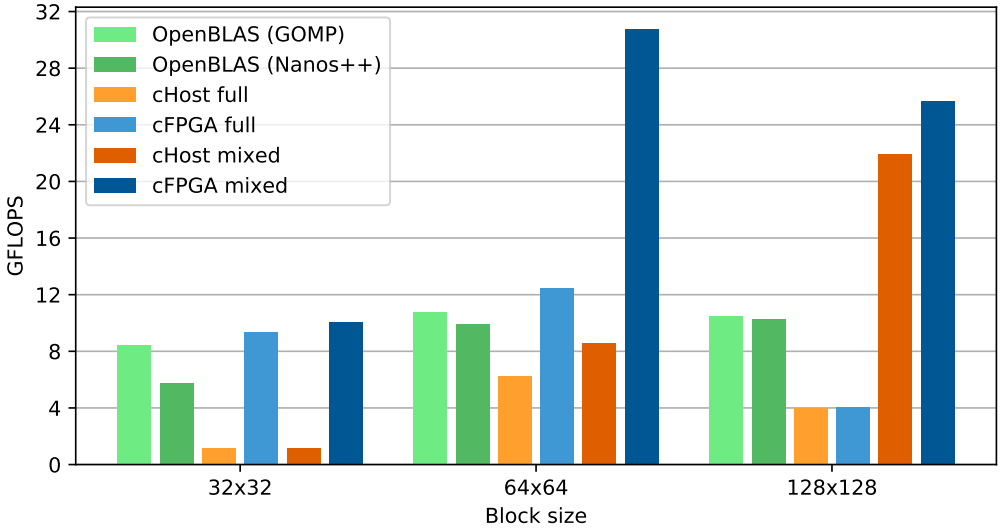


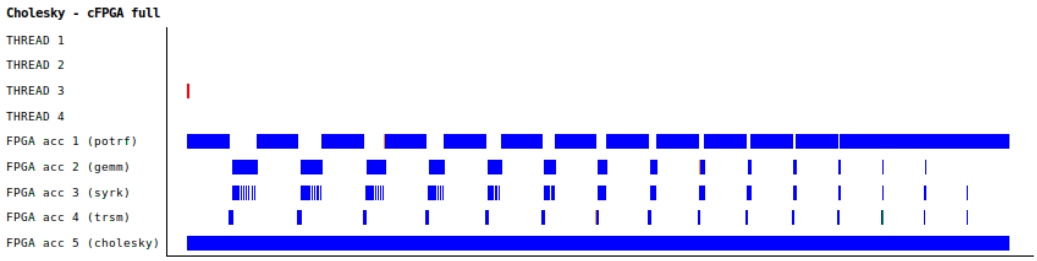
Figure 4.24: Cholesky GFLOPS with different approaches and block sizes

Figure 4.24 shows the average execution time of 3 executions (x-axis) for the different approaches with 3 blocks sizes. The size of the input matrix is 2048x2048 in all the results. The *cFPGA* approach always has the same or better execution time than *cHost*. Even for the larger task size the *cFPGA* approach is faster (around a 20 %) than the *cHost* alternative. Also, note that the absolute fastest solution is the 64x64 size *cFPGA*

approach (around a 40 % faster than the fastest *cHost*) due to the better resource balance that the smaller kernel sizes offer. This solution is unreachable without the proposed extensions.

The 128x128 block size results also show that the *full* approach is worse than *OpenBLAS*. The execution of the `potrf` tasks in the slow task accelerator enlarges the execution time. As it can be seen, the combination of the other kernel task inside the FPGA device with the execution of `potrf` tasks in the host processor boost the performance by a 5x. This improvement is due to the faster execution of `potrf` tasks and the increase of 1 `gemm` accelerator in the FPGA device. Figure 4.25 shows execution traces of this case. Traces show the state of the different resources (y-axes) among the execution time (x-axis). Different colors represent different states: red, offloading a task into the FPGA device; light-green, executing a `potrf` task in the SMP; and blue, executing a task in FPGA task accelerator.

(a) cFPGA mixed



(b) cFPGA full

Figure 4.25: Cholesky execution traces of *cFPGA* approach (128x128 block size and 2048x2048 matrix size)

The same execution time trend of `potrf` 128x128 block size can be seen in the 64x64 block size results of figure 4.24, but with smaller differences. This is because the `potrf` kernel tasks take a smaller percentage of execution time, making it less critical. On the other hand, the 32x32 block size results show that *cHost* cannot efficiently handle the fine-grain kernel tasks. The runtime overheads in the task creation and a large number of required host-FPGA synchronizations harm the performance. Despite that, *cFPGA* can manage a large number of tasks and occupy all the task accelerators.

4.8 Conclusion

This chapter presents an extension of the OmpSs@FPGA ecosystem to support task creation and synchronization directly inside the FPGA devices, breaking the master-slave model and demonstrating the feasibility of such model. The proposed design and implementation includes novel modules in the HWR, like the `Scheduler Manager` and the `Taskwait Manager`. These modules cooperate with the host runtime to avoid useless host-FPGA round trips. The optimizations are transparent to the programmers, increasing programmability and productivity. The modifications allow FPGA task accelerators to interact with the runtime and create children tasks and synchronize them regardless of the target architecture.

This extension enables a new dimension of possibilities for application programmers as they can mix tasks for different devices and nest them without restrictions. For example, a system call can be done inside the FPGA wrapping it inside an SMP task. This child task will be reverse offloaded to the host where the task will execute.

The evaluation results show that the new distributed model reaches higher performance than the master-slave model. Moreover, the peak is achieved using fine-grained tasks which deliver a smaller performance in the baseline model. The improvement is obtained reducing the runtime overheads and eliminating non-needed host-FPGA synchronizations. This allows a faster task creation with less energy consumption. Moreover, the creation and synchronization of FPGA tasks directly inside the FPGA device allows the use of host CPUs to execute real application code instead of runtime code.

The proposal has been firstly implemented in a SoC with an integrated FPGA device. However, the same design has been successfully used in discrete FPGAs connected through PCI Express (PCIe). Thanks to the modular design of the software stack and the FPGA IP blocks, the support for this platforms only requires minimal changes in the low level communication libraries and IPs.

FPGAs are an excellent platform to develop concepts due to their flexibility to incorporate new hardware. However, the proposal design can be easily extended to other heterogeneous systems like GPGPUs or AI specific accelerators. Small ASIC modules dedicated to runtime tasks can help all these systems to have better programmability and obtain better performance at a very low cost. These benefits will even increase for large systems where several accelerators are directly connected between them through the network. Indeed, the proposed design provides a distributed management that avoids bottlenecks in the host and enables applications to scale up.

4.9 Publications

The list of thesis publications related to the work explained in this chapter is:

- Asynchronous Task Creation for Task-Based Parallel Programming Runtimes.
  Jaume Bosch, Xubin Tan, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé. OpenMPCon 2018. [7]
  This publication analyzes the initial requirements to support the task spawn inside co-processors and how to integrate the capabilities in the baseline system.

- Supporting task creation inside FPGA devices.
  Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González. BSC International Doctoral Symposium 2019. [9]
  This publication proposes a first implementation to support the task spawn inside FPGA devices and presents the initial results of the synthetic benchmark.

- Breaking master-slave model between host and FPGAs.
  Jaume Bosch, Miquel Vidal, Antonio Filgueras, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé. PPoPP 2020. [10]
  This publication presents the full implementation to allow the task spawn inside the FPGA devices over OmpSs. It also presents an extended evaluation with all performance results.

The list of publications related to collaborations with the work presented in this chapter is:

- Design and implementation of an architecture-aware hardware runtime for heterogeneous systems.
  Juan Miquel de Haro, Jaume Bosch, Daniel Jiménez-González, Carlos Álvarez. BSC International Doctoral Symposium 2020. [18]
  In this work, the extension of OmpSs@FPGA ecosystem to support the task spawn inside FPGA devices has been used. Moreover, the modular design of HWR has been used to integrate the Picos Daviu IP module creating a new HWR called POM.

- OmpSs@FPGA framework for high performance FPGA computing.
  Juan Miquel de Haro, Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, Jesus Labarta. TC 2021 [Accepted for publication]. [19]
  In this work, the extension of OmpSs@FPGA ecosystem to support the task spawn inside FPGA devices has been used.

- High Performance Computing particle-pair distance algorithms, to generate X-ray spectra from 3D models.
  César González, Jaume Bosch, Juan Miguel de Haro, Maurizio Paolini, Antonio Filgueras, Simone Balocco, Carlos, Álvarez, Ramon Pons. HPC 2021 [Under review]. [20]
  In this work, the extension of OmpSs@FPGA ecosystem to support the task spawn inside FPGA devices has been used.

Proposal for Recurrent Tasks

5

Periodic systems (i.e., recurrent workloads) are a common workload in several industry environments and real-time systems. Those workloads use the task concept to define the different activities that must be executed periodically (after some amount of time). Thereby, task-based parallel programming models are a great candidate to support recurrent workloads that may afterward be parallelized and handled with the current programming model features. The proposal aims to demonstrate that task-based parallel programming models can be easily extended to efficiently support recurrent workloads.

Section 5.1 describes the key ideas of the proposal design. Then, sections 5.2 to 5.6 describe the modifications done starting from the OmpSs programming model, up to the FPGA bitstream design. After them, section 5.7 shows an evaluation of different benchmarks using the proposal enhancements. Finally, section 5.8 concludes the chapter with the key contributions, and section 5.9 lists the publications related to this chapter.

5.1 Proposal Design

The following points summarize the design goals of the proposal:

- Integrate the concept of a recurrent task in the OmpSs programming model, as an example of how the task-based parallel programming models can express a recurrent system.
- Use the low-power and high-throughput capabilities of FPGA task accelerators to implement the recurrent tasks management.
- Support new synchronization mechanisms between independent FPGA task accelerators.

The extension of the baseline task-based parallel programming model to allow the expression of recurrent tasks is the first design goal. The point is to extend the current task syntax to allow the definition of a recurrent system by means of a period and a repetitions number. The task starts executing every time the defined period is accomplished, not before. However, the start may be delayed if the previous repetition

has not finished. The recurrent tasks do not become accomplished until a number of repetitions are executed or aborted due to some conditions explicitly programmed in the application code. The proposal does not consider some of the features that a real-time recurrent system may need, like task deadlines, but it does not limit their future support or their explicit handling in the application code. In this sense, it is a first step open to future extensions.

The proposal allows a compact, easy and intuitive expression of a recurrent system. Although a recurrent system could be implemented at application level with the baseline programming model, the programming effort is much smaller thanks to the proposed extensions.

The handling of application recurrent tasks requires some runtime support to continuously monitor them and schedule the task repetitions after the different periods expire. This continuous monitoring is very suitable for FPGA devices. As demonstrated in the previous chapter, they have task creation capabilities with a lower-power consumption and a high throughput. In both metrics, FPGA tasks outperform the equivalent SMP tasks in the host.

As a complementary enhancement, a new synchronization mechanism has been developed for FPGA tasks. The synchronization between tasks is usually performed through data dependences or explicitly using the taskwait directive. However, this approach requires a common task context that performs the synchronization. This may not be suitable for the recurrent tasks that are executed periodically without coordination. The critical clause can perform such coordination. Therefore, the proposal design includes critical regions to coordinate different FPGA task accelerators. The critical regions are supported by the HWR using the infrastructure of the previous proposal.

5.2 Programming model extension

The extension of the OmpSs programming model [4] defines two new clauses in the task directive used to label a task as a recurrent one. The new clauses are optional. If they are not used, the created task is a regular task; and if any of both are present, the task becomes a recurrent task. The clauses are:

- `period(N)`. This clause defines the minimum amount of time between the beginning of two task executions. By default, the time is expressed in microseconds, but this may be changed at compile time (currently, using a compiler option). The value is 32 bits wide, and it is evaluated at runtime, so it may be unknown at

compile time. When the `period` clause is not provided in a recurrent task, the default (implicit) value is 0, which makes the task body execution restarts just after finishing. The same happens when the task execution takes longer than the period: the next repetition starts just after the former. Therefore, the task will not be under execution twice. However, this behavior could be changed in the future with extra clauses or by a runtime option. Other approaches could be: aborting the current repetition, aborting the next repetition, schedule both repetitions in parallel.

- `num_repetitions(N)`. This clause defines the maximum number of times that the task body will be executed. The value is 32 bits wide, and it is evaluated at runtime, so it may be unknown at compile time. When the `num_repetitions` clause is not provided, the default (implicit) clause value in a recurrent task is an unlimited number of repetitions. Since 0 repetitions may be a valid amount of repetitions, the unlimited number of repetitions is represented with the largest representable value in an unsigned 32 bits number (`0xFFFFFFFF`, or 4 294 967 295 in base 10).

The semantics of other task clauses in a recurrent task remain equal to a regular task. The same criteria apply to the interaction of a recurrent task with other programming model directives (like the taskwait). The recurrent tasks will not become finished until all repetitions have been run. Therefore, a taskwait after a recurrent task will not be accomplished until all the recurrent task repetitions have been run. Also, the data dependences in a recurrent task only postpone the execution of the first repetition but not the others, and the successor tasks of a recurrent task are not ready until all repetitions are executed.

```
1  #pragma omp task inout([10]array) num_repetitions(reps) period(1000000)
2  void recurrent_task(int *array, const int reps);
3
4  #pragma omp task inout([10]array)
5  void regular_task(int *array);
6
7  int main(...) {
8    int array[10];
9    regular_task(array);
10   recurrent_task(array, 100);
11   regular_task(array);
12   #pragma omp taskwait
13 }
```

Listing 5.1: OmpSs example with a recurrent task

Listing 5.1 shows an example of a recurrent task (`recurrent_task`) and a regular task (`regular_task`). The main function calls the regular task, then the periodic task, and finally the regular, creating a chain of three task instances due to its data dependence. The recurrent task has the `num_repetitions` clause, which defines that the task body will be executed `reps` times (in this case, the argument value is 100), and the `period` clause, which defines that the task will begin the execution every 1 second (1000000 microseconds). The first regular task becomes ready when created as its data dependences are satisfied. In contrast, the other two are postponed. The recurrent task is postponed until the first regular task finishes, and the second regular task is postponed until the 100 repetitions of the recurrent task have been executed.

5.3 Mercurium Compiler Support

The support of new OmpSs capabilities has been integrated into Mercurium compiler. The changes involved the OmpSs and the FPGA device translation level phases but also changes in the compiler core to extend some representations.

The first change involved the compiler core in order to store the expressions contained in the new clauses. This information is checked during the replacement of OmpSs task directives into runtime API calls. If the task directive does not contain neither `period` clause nor `num_repetitions`, the compiler emits the regular API calls and generates the user function code as before. In contrast, the compiler forwards the period and number of repetitions information to the runtime during the task creation if any of the clauses appear. The information is provided to the runtime using the new `nanos_set_wd_recurrent` API.

The support of critical regions inside the FPGA task accelerator uses the already existing runtime API for the regular SMP threads. Therefore, the compiler performs the already existing code transformations regardless of the architecture of the enclosing task. The transformation is based on a mutual exclusion lock acquired at the beginning of the critical region and released after the region body.

5.3.1 HLS Source Code

The wrapper generated by the compiler must support a new FPGA command for the execution of a recurrent task. The command format is similar to the regular execute task command (figure 4.7) but with an extra word that contains the period and number of

repetitions information. The format of the new command is shown in figure 5.1. The new word is shown in blue, and the others are in gray.

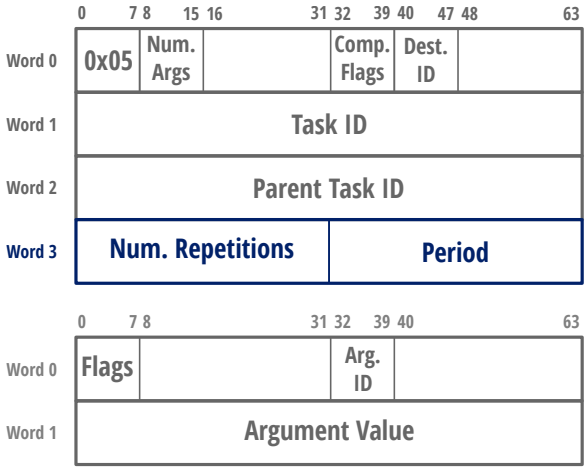


Figure 5.1: Format of execute recurrent task command

The proposal just requires a loop around the call to the user function in the wrapper source code. This loop iterates as many times as the number of repetitions the task has, and it adds a delay after the user function call. The delay implementation depends on the time unit of `period` clause, which is microseconds by default. A simplified pseudo-code version of this wrapper logic is shown in listing 5.2 for the example of listing 5.1.

The intermediate HLS source code files may have calls to new Nanos++ APIs when the FPGA task is recurrent. Therefore, the wrapper generated by Mercurium must include an implementation of those APIs when the FPGA task may call them. The main goal of those APIs in the FPGA task accelerator is to control the repetitions of recurrent tasks. They are listed following and further detailed in section 5.4:

- `nanos_get_periodic_task_repetition_num`. The implementation returns the current repetition number which is being executed. Note that the first repetition is the number 1, since 0 is returned when the task is non-recurrent.
- `nanos_cancel_periodic_task`. The implementation cancels the remaining repetitions of the current recurrent task. This does not cancel the remaining task code after the API call.

The support of critical regions inside FPGA task accelerators uses the already defined Nanos++ APIs in the intermediate HLS source code. The compiler analyzes the task body looking for critical regions and adapts the intermediate source code generation according to the task needs. The following points describe the new supported APIs.

```
extern const ap_uint<5> mcxx_acceleratorID;
extern const ap_uint<64> mcxx_hwCounter;
extern const ap_uint<10> mcxx_acceleratorFreq;
unsigned int mcxx_period;
unsigned int mcxx_numReps;

void recurrent_task(int *array, const int reps)
{
  //...
}

void recurrent_task_mcxx_hls_wrapper(
  hls_axis_t mcxx_inStream, hls_axis_t mcxx_outStream,
  int *array_port)
{
  #pragma HLS INTERFACE ap_ctrl_none port=return
  #pragma HLS INTERFACE axis port=mcxx_inStream
  #pragma HLS INTERFACE axis port=mcxx_outStream
  #pragma HLS INTERFACE m_axi port=array_port

  unsigned long long int _tmp;
  //...

  _tmp = mcxx_inStream.read().data;
  mcxx_period = _tmp;
  mcxx_numReps = _tmp >> 32;

  //...

  for (unsigned int rep=0; rep<numReps; rep++) {
    _tmp = mcxx_get_current_time();
    recurrent_task(...);
    _tmp = mcxx_get_current_time() - _tmp;
    if (_tmp < mcxx_period) usleep(mcxx_period - _tmp);
  }

  //...
}
```

Listing 5.2: FPGA task accelerator wrapper example with recurrent loop

nanos_set_lock

The wrapper generated in the HLS intermediate file implements the `nanos_set_lock` API to support calling it from the FPGA task accelerator code. The implementation forwards the information of the lock that will be acquired to the HWR. The communication is done over `mcxx_eOutPort` that communicates all FPGA task accelerators with the HWR.

The AXI-Stream format of message sent to HWR can be seen in figure 5.2. The gray part shows the protocol information that includes 1 bit for the last package information and two identifiers (5 bits wide) for the destination and the source. The data part of the message is shown in blue, and it encodes: 8 bits fixed to the code `0x4` that define that the task wants to acquire the lock, and 24 bits for the identifier, or hash, of the lock to be acquired.

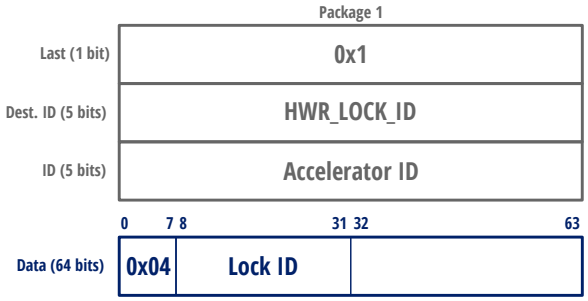


Figure 5.2: Format of set lock message for Lock manager

After sending the message, the FPGA task accelerator waits until the HWR sends a response through `mcxx_inStream`. This response notifies whether the lock has been acquired or not. If the response is successful, the lock has been acquired, and the critical region can be executed. Otherwise, the acquire is retried as the API is blocking.

nanos_unset_lock

The wrapper generated in the HLS intermediate file implements the `nanos_unset_lock` API to support calling it from the FPGA task accelerator code. The implementation forwards the information of the lock that will be released to the HWR. The communication is done over `mcxx_eOutPort` that communicates all FPGA task accelerators with the HWR.

The AXI-Stream format of message sent to HWR can be seen in figure 5.3. The gray part shows the protocol information that includes 1 bit for the last package information and two identifiers (5 bits wide) for the destination and the source. The data part of the message is shown in blue, and it encodes: 8 bits fixed to the code `0x6` that define that

the task wants to release the lock, and 24 bits for the identifier, or hash, of the lock to be released.

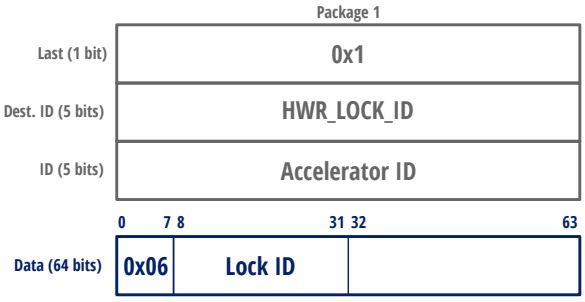


Figure 5.3: Format of unset lock message for Lock manager

In contrast to `nanos_set_lock`, this API is non-blocking and it does not wait for any response. Then, the lock release is performed asynchronously as the FPGA task accelerator sends the release message and keeps executing the task body.

nanos_try_lock

The wrapper generated in the HLS intermediate file implements the `nanos_try_lock` API to support calling it from the FPGA task accelerator code. The implementation forwards the information of the lock that will be acquired without blocking. The communication is done over `mcxx_eOutPort` that communicates all FPGA task accelerators with the HWR.

The AXI-Stream format of message sent to HWR is the same of `nanos_set_lock` and can be seen in figure 5.2. However, the FPGA task accelerator does not retry acquiring the lock if the HWR response to the lock message is non-successful. The result is returned to the caller context in order to allow the application code to decide what to do, considering if the lock has been acquired or not.

5.4 Nanos++ Runtime Support

The support for recurrent tasks requires some support by Nanos++ runtime. The WD has been extended to store the period and number of repetitions information if needed. Then, the runtime uses that information to implement the desired behavior. In the case of recurrent SMP tasks, the runtime handles the recurrent task execution with the needed delay in the midtime. In the case of recurrent FPGA tasks, the runtime offloads the task

to the FPGA device using the new xTasks API (explained in section 5.5). Section 5.4.1 describes the new APIs introduced in the runtime to support the new capabilities.

5.4.1 New APIs

The Nanos++ API has been extended with three new APIs. `nanos_get_periodic_task_repetition_num` and `nanos_cancel_periodic_task` can be called at any device and from any task. `nanos_set_wd_recurrent` is the other new API which is used to set the recurrent information of a newly created task before its submission.

nanos_get_periodic_task_repetition_num

The `nanos_get_periodic_task_repetition_num` API returns an unsigned integer with the current repetition number of the recurrent task being executed. The first repetition number is 1, and the highest repetition number is 4 294 967 295 (`0xFFFFFFFF`). After the highest number, the count will overflow and start again from 0. In addition, the repetition number 0 is returned when the API is called from a non-recurrent task. The API declaration is shown in listing 5.3.

```
1 unsigned int nanos_get_periodic_task_repetition_num();
```

Listing 5.3: Nanos++ API to retrieve current task repetition

nanos_cancel_periodic_task

The `nanos_cancel_periodic_task` API cancels the remaining repetitions (if any) of the current recurrent task being executed by the caller. The API call does not abort the remaining user code after the API call for the current task repetition. The API will not have any effect when called outside a recurrent task. The API declaration is shown in listing 5.4.

```
1 void nanos_cancel_periodic_task();
```

Listing 5.4: Nanos++ API to cancel remaining task repetitions

nanos_set_wd_recurrent

The `nanos_set_wd_recurrent` API synchronizes the child tasks, blocking the caller until the execution of child tasks have finished. The API can only be called from the FPGA task accelerators, and it throws an error if called in the host. The API declaration is shown in listing 5.5 and its parameters are:

- `wd`. WD which recurrent information will be set.
- `period`. Minimum period (in microseconds) between task beginnings.
- `repetitions`. Number of times that task body will be executed before the task becomes finished.

```
1 nanos_err_t nanos_set_wd_recurrent(
2   nanos_wd_t wd, unsigned int period, unsigned int repetitions);
```

Listing 5.5: Nanos++ FPGA API to set recurrent task information

5.5 xTasks Library Support

The offload of recurrent tasks from both runtimes, Nanos++ and HWR, is done through xTasks library. To support the new functionalities, the library added new APIs available for Nanos++ runtime (described in section 5.5.1) which interface the already existent communication queues. The new API is used to create the recurrent task, while the other task operations (delete, submit, etc.) can be done through the already existent API.

5.5.1 New APIs

The xTasks API has been extended with one new API which is detailed in the following point.

xtasksCreatePeriodicTask

The `xtasksCreatePeriodicTask` creates a recurrent task for an accelerator with the given identifier, compute flags, period and number of repetitions. Most of the parameters and types are shared with `xtasksCreateTask`, which is explained in section 2.4.1. The API declaration is shown in listing 5.6, and its parameters are:

- `id`. Task identifier that will be returned at finalization. Arbitrary identifier that caller can use to identify a task uniquely.
- `accel`. Accelerator handle where task will be submitted.
- `compute`. Compute flags to enable/disable the execution of the task body.
- `numReps`. Number of task body repetitions to execute.
- `period`. Time between the execution beginning of two repetitions.
- `handle`. Pointer to a valid `xtasks_task_handle` that will be set with an opaque task handle.

```
1 xtasks_stat xtasksCreatePeriodicTask(
2   xtasks_task_id const id, xtasks_acc_handle const accel,
3   xtasks_comp_flags const compute, unsigned int const numReps,
4   unsigned int const period, xtasks_task_handle *handle);
```

Listing 5.6: xTasks APIs for FPGA recurrent task creation

5.6 FPGA Design Support

The support of the new features only requires minor changes in the FPGA task accelerators and HWR. The main change is the new command that must be supported by the HLS wrapper, which is generated by Mercurium, and by the HWR that must understand the new command and correctly move it from queues to the streams. Section 5.6.1 describes the changes done in the FPGA task accelerators and section 5.6.2 the ones in HWR.

Figure 5.4 shows the elements in the system, focused in the FPGA part. The figure extends the chapter 4 proposal design shown in figure 4.11. The new elements shown in the figure are explained in the following sections.

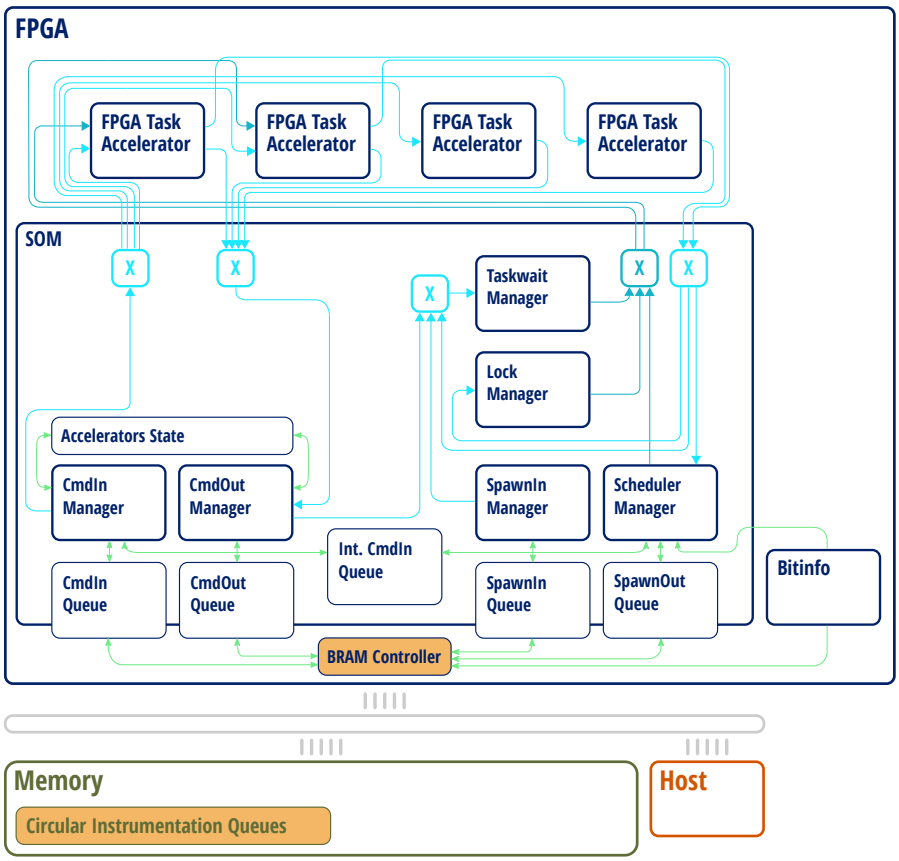


Figure 5.4: FPGA Bitstream design with the extended SOM Hardware Runtime (v2)

5.6.1 FPGA Task Accelerators

The FPGA task accelerators have been extended, if needed, to support the new recurrent capabilities. AIT checks if the HLS source code generated by Mercurium has the additional ports (`mcxx_hwCounter` and `mcxx_acceleratorFreq`). They are used to delay the requested time between repetitions by accounting for the number of cycles based on the FPGA task accelerator frequency. When the new ports are found, AIT adds and connects an instance of a hardware counter (`HW Counter`) and a constant variable with the accelerator frequency (`Frequency`). The interconnection and new elements of a recurrent FPGA task accelerator are shown in figure 5.5 together with the previous elements.

The acquire and release messages sent to support the critical regions on the FPGA task accelerators are forwarded through the existing interconnections. These interconnections are the ones that go through the `eIn Adapter` and `eOut Adapter` as shown in figure 5.5. They were introduced and handled in the previous proposal.

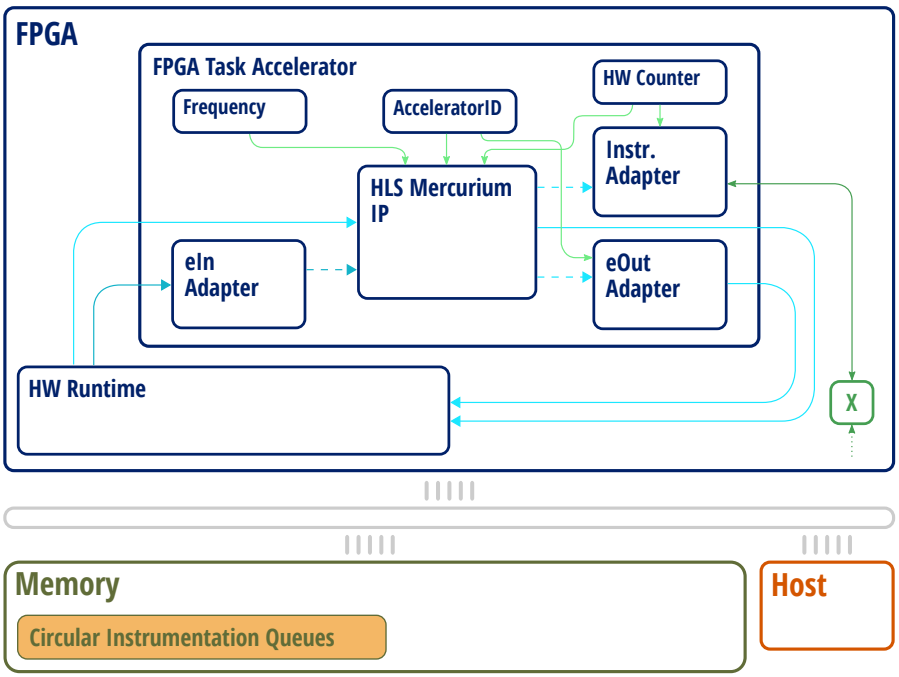


Figure 5.5: Internal structure of FPGA Task Accelerator with task spawn support

5.6.2 Hardware Runtime

The SOM implementation of the HWR has been extended to support the new execute recurrent task command. The update only involved the CommandIn manager, which needs to recognize the new command to correctly compute the number of words following the command header. The format of those commands is shown in figure 5.1.

One new module has been added in the SOM implementation to handle the acquire and release messages for locks. The new module is the `Lock Manager`, and it uses the HWR interconnections added in the proposal of chapter 4.

Lock Manager

The `Lock Manager` is an IP block developed in C++ using HLS tools. Its external interface is shown in figure 5.6, where the different ports and protocols to communicate the module with the other components are detailed. Those ports are connected as shown in figure 5.4 and they are:

- `In Stream`. AXI-Stream port used by all FPGA task accelerators to send the acquire/release messages.

- `Out Stream`. AXI-Stream port to send the acknowledge of acquire messages to FPGA task accelerators.

Figure 5.6: External interface of Lock Manager

The purpose of the `Lock Manager` is keeping the state of different locks that may be concurrently acquired and released by the FPGA task accelerators. To this end, the module has an internal table with the state of each lock. The module can receive two types of messages in the input stream: acquire messages (format shown in figure 5.2) and release messages (format shown in figure 5.3). Whenever it receives a new message, it looks for the lock identifier in the internal table and sets or clears the lock state. The response is successful in the acquire messages when the lock was not previously set and fail when the state is unchanged.

The number of entries in the internal lookup table is fixed during the AIT design stage. The current implementation has a direct mapping in the table for each lock ID. However, this could be changed and implement an N-way table. There must be enough entries in the table to ensure that nested locks do not map to the same entry. Otherwise, the system will enter in a deadlock state due to the impossibility of acquiring the inner lock.

5.7 Evaluation

The evaluation of the proposed design and implementation of recurrent tasks has been done in terms of programmability and productivity, limitations of tasks management, and power savings. Section 5.7.1 presents the experimental setup used among the evaluation. Section 5.7.2 presents the evaluation of the limitations and management overheads of the proposed design using a synthetic benchmark. Section 5.7.3 shows the results for a sensor monitoring use case in embedded systems. Finally, section 5.7.4 presents the evaluation of the proposal for a face detection application. This face detection application is made over a stream of images.

5.7.1 Experimental Setup

The evaluation of the proposal modifications has been done in an embedded board and with real executions. The tools used to generate the application bitstreams and binaries are: Vivado Design Suite 2020.1, GNU C/C++ Compiler 6.2.0, and PetaLinux Tools 2019.2. The modifications have been developed on top of OmpSs@FPGA release 2.3.0.

Zedboard

All executions have been run in a Zedboard, which contains a Xilinx Zynq-7000 All Programmable SoC [@72]. The board is commonly used in embedded industrial systems as it offers high versatility with reduced power consumption and budget. The SoC is composed of 2 ARM Cortex-A9 cores, that run at 667 MHz, a Xilinx Zynq-7000 FPGA and a main DDR3 memory of 512 MB. The board is booted using the Ubuntu Linux 16.04 operating system. All FPGA bitstreams have been generated and executed at 100 MHz, or 200 MHz if mentioned.

5.7.2 Synthetic benchmark

Listing 5.7 shows the pseudo-code of the synthetic benchmark using the proposal extensions. The benchmark does nothing but its execution time can be analyzed when changing the task size, task period and number of repetitions. This pattern allows an exploration of the runtime limits to manage fine-grain periodic tasks. The benchmark executes a task called `foo` which is a periodic task that gets executed `num_reps` times every `period` microseconds. The task lasts for `duration` microseconds.

```
1  #pragma omp target device(fpga) \
2    num_repetitions(num_reps) period(period)
3  #pragma omp task
4  void foo(int duration) {
5    usleep(duration);
6  }
7
8  int main(...) {
9    foo(duration);
10   #pragma omp taskwait
11 }
```

Listing 5.7: Recurrent synthetic benchmark pseudo-code

The same benchmark behavior can be achieved without using the proposal extensions. However, it requires an extra effort from the programmers side and additional code to manage the periodic task. Listing 5.8 shows the pseudo-code for the same synthetic benchmarks but without the proposal extensions. The benchmark requires an extra task (`foo_manager`) that manages all executions of recurrent task (`foo`). The manager task runs in the SMP host threads, and it has to launch and synchronize the recurrent task every `period` microseconds. During the wait, the taskyield directive [@73] (line 16) is used to avoid blocking the host thread that executes the manager task.

```
1  #pragma omp target device(fpga)
2  #pragma omp task
3  void foo(int duration) {
4    usleep(duration);
5  }
6
7  #pragma omp task
8  void foo_manager(int duration, int num_reps, int period) {
9    for (unsigned int rep=0; rep<num_reps; rep++) {
10     const double t_ini = wall_time_us();
11     foo(duration);
12     #pragma omp taskwait
13     while ((wall_time_us() - t_ini) < (double)period &&
14            rep < (num_reps - 1))
15     {
16       #pragma omp taskyield
17     }
18   }
19 }
20
21 int main(...) {
22   foo_manager(duration, num_reps, period);
23   #pragma omp taskwait
24 }
```

Listing 5.8: Recurrent synthetic benchmark pseudo-code without proposal extensions

The comparison of both synthetic benchmark implementations shows the programmability enhancement that the proposal adds. The baseline system requires one extra task of 13 lines for each recurrent task in the application as shown in listing 5.8. Moreover, the extra manager task consumes resources in the host threads and adds pressure to the host runtime, which may create a new bottleneck. In contrast, the proposal moves the management complexity in the FPGA task accelerator that only needs to receive the task to execute with the number of repetitions and the period. It has a minimal footprint in the resources used by the FPGA task accelerator as shown in table 5.1 (for a 100 MHz build). In terms of power consumption, the proposal removes the need to use an A9 core for recurrent task management, which consumes 277 mW. Indeed, it only increases the

FPGA task accelerator power in 1 mW doing the same work. Both powers are reported by Vivado in the post-implementation power summary.

Name	BRAM	DSP	FF	LUT	Power
Zedboard	280	220	106 400	53 200	-
ARM A9 Core	-	-	-	-	277 mW
`foo` baseline	0 (0 %)	2 (0.9 %)	593 (0.6 %)	478 (0.9 %)	7 mW
`foo` proposal	0 (0 %)	4 (1.8 %)	921 (0.9 %)	792 (1.5 %)	8 mW

Table 5.1: Vivado resources utilization and power report for Zedboard (100 MHz)

The time devoted to each repetition is `period`, or `duration` if the task execution lasts more than task's period (concurrent repetitions are not allowed). Then, the optimal execution time is defined by `num_reps` per the time devoted to each repetition but the last one which only needs to consider the task execution (`duration`) as the benchmark can immediately finish. Equation 5.1 defines the optimal execution time given the `duration`, `num_reps` and `period` parameters.

$$Time_{optimal} = (num\_reps - 1) * max(duration, period) + duration \tag{5.1}$$

Then, the effectiveness of an execution is defined by the ratio between the optimal execution time and the real execution time. The resulting value is a percentage that ideally would always be a 100 %. Equation 5.2 shows the formula to compute the effectiveness of an execution.

$$Effectiveness_{exec} = Time_{optimal}/Time_{exec} \tag{5.2}$$

Figure 5.7 shows the effectiveness (y-axis) of the synthetic benchmark when changing the task period (x-axis). The chart has different series that correspond to different task durations. The task durations are labeled in the legend and are only shown for periods equal or larger than them. The number of repetitions is constant in all the results, and its value is 10 000. The FPGA device is configured at 100 MHz. The blue series show the effectiveness of synthetic benchmark implemented with the proposal enhancements and having the recurrent task management in the FPGA task accelerator. On the other hand, the orange series show the effectiveness of baseline implementation using a manager task in the host threads. These baseline series are labeled in the legend with *(host)*.

The effectiveness results in figure 5.7 show that the proposal implementation (blue) always obtains higher effectiveness than the baseline implementation (orange). The host management of the recurrent task always has the same pattern. First, the effectiveness

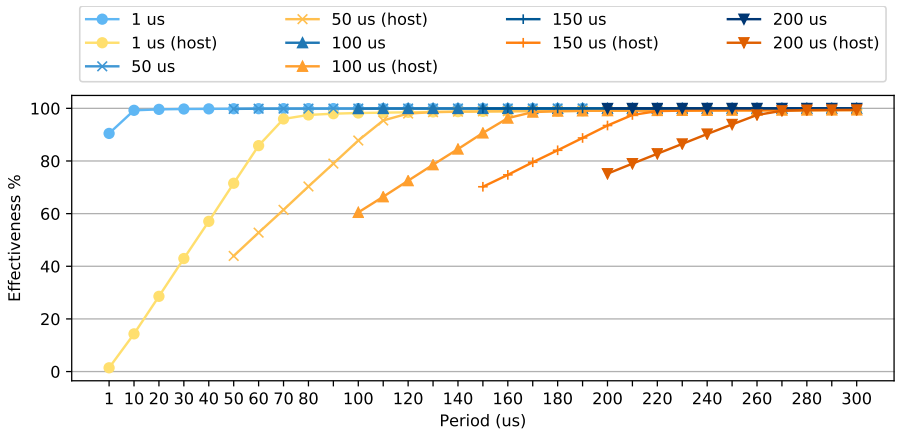


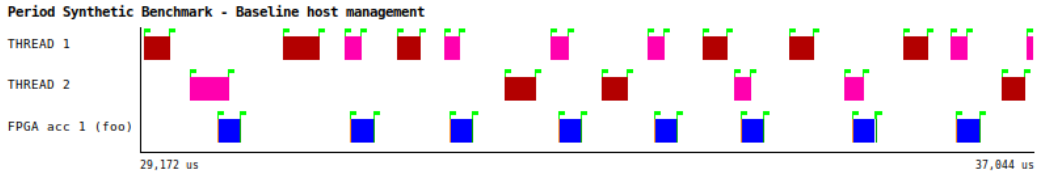
Figure 5.7: Effectiveness of synthetic benchmark for different task durations and periods (microseconds scale)

is poor when the period is equal to the task duration. Then, the effectiveness starts increasing with the higher periods until the difference between the period and the task duration is at least 70 microseconds. After that, the effectiveness is stable near to 99.4 %. These 70 microseconds' need to effectively handle the recurrent task are due to the communication round-trip between the host and the FPGA task accelerator. In contrast, the proposal implementation is able to effectively handle all task durations and periods as the management is done inside the FPGA task accelerator itself. Therefore, the communication between the host and the FPGA task accelerator is only needed twice: at the beginning of the recurrent task and after all task repetitions.

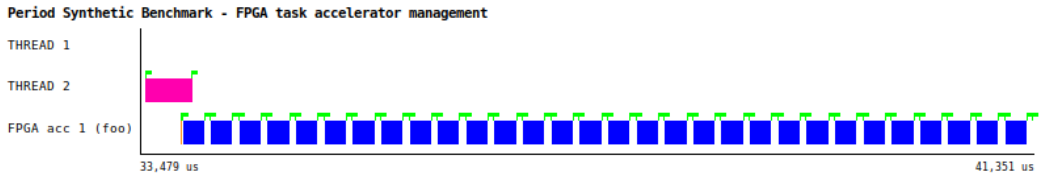
Figure 5.8 shows two execution traces for a portion of benchmark execution (same duration in both). The recurrent task parameters are 200 microseconds for task duration and 250 microseconds for task period. Figure 5.8a contains the trace for the baseline host management and figure 5.8b contains the equivalent one with the proposal enhancements. In both traces, the different colors represent the activities being done in the different computational elements among time (x-axes). Pink regions represent the offload of `foo` task to the FPGA. Brown regions represent the execution of `foo_manager` task. Blue regions represent the execution of one `foo` repetition in the FPGA task accelerator.

The elapsed time shown in traces is about 7 900 microseconds, which could fit ∼31 repetitions of `foo` task. Although, the baseline host trace only contains 8 repetitions of the recurrent task due to the task communication and instrumentation overheads. In contrast, the proposal trace contains 30 repetitions of the recurrent task launched every 250 microseconds.

The performance gap between both managements is 4x in figure 5.8, however the real performance gap is 1.1x (as shown in figure 5.7). The difference comes from the



(a) Baseline host management



(b) Proposal management

Figure 5.8: Execution traces of synthetic benchmark with 250 us period and 200 us duration

instrumentation overheads that are huge for those fine-grain tasks. In the proposal management, the host threads are not doing any activity after the initial offload which puts less stress in the instrumentation system. Considering the 90 % effectiveness shown in figure 5.7 the baseline host management should contain 28 repetitions of the recurrent tasks. The difference between the expected trace and the real obtained trace increases the importance of efficient task management like the proposal one.

The power savings of the proposal can be estimated based on the traces of figure 5.8. Assuming a large number of task repetitions, which is the expected behavior in a real-time system, the proposal allows freeing the ARM cores 100 % of the execution time. In contrast, one host thread is at least needed in the baseline host management. This management difference makes the new proposal design consume 276 mW less than the baseline implementation.

To further analyze the proposal limits and overheads, figure 5.9 shows the effectiveness of the synthetic benchmark like figure 5.7 but with task durations and periods at nanoseconds scale. The number of repetitions is constant in all the results to 10 000 000, which is proportionally equivalent to the 10 000 value of microsecond results. The FPGA device is also configured at 100 MHz. The host management effectiveness (yellow line) is only shown for one nanosecond task duration as it remains around 1 % in all periods. For all periods shown in figure 5.9, any task duration larger than one nanosecond results in an effectiveness below the shown line for the host management.

The effectiveness results in figure 5.9 confirm that the proposal implementation (blue) is the only option to effectively manage fine-grain periodic tasks. The FPGA management results show a pattern similar to the previously seen in host management results in figure 5.7. For a given duration, the effectiveness notably increases with the higher

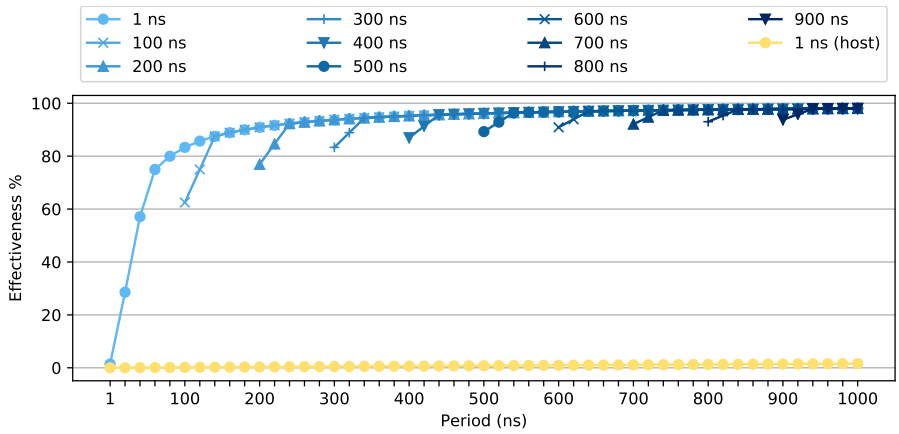


Figure 5.9: Effectiveness of synthetic benchmark for different task durations and periods (nanoseconds scale)

periods until the difference between the period and the task duration is at least 60 nanoseconds. After that, the effectiveness stabilizes and slowly increases with higher periods. The need for these 60 nanoseconds to effectively handle the recurrent task is due to the minimal management in the FPGA task accelerator to check the number of remaining repetitions and whether the period has elapsed or not. Despite that, the results with a duration above 900 nanoseconds show that the proposal management is very efficient without matter if the task period is equal to the task duration. This is thanks to the thin overheads of the proposal, which are 1167x smaller than the ones in the baseline approach.

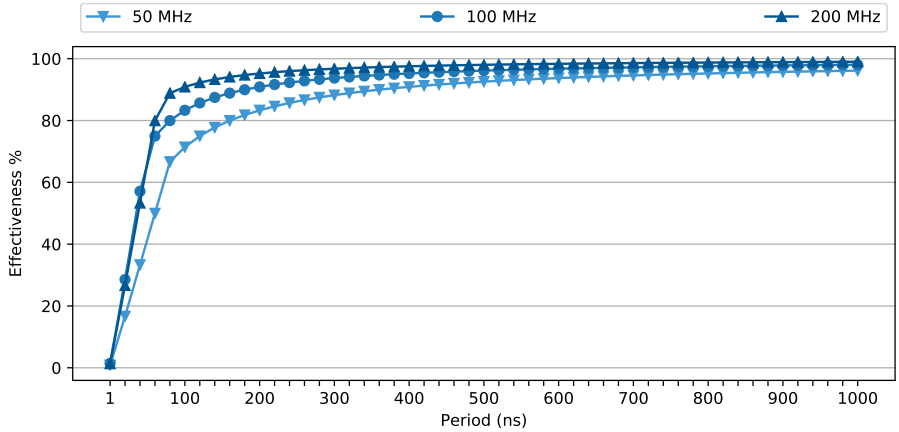


Figure 5.10: Effectiveness of synthetic benchmark for different FPGA frequencies and periods with 1 nanosecond task duration

Figure 5.10 shows the effectiveness of the synthetic benchmark (y-axis) among different periods (x-axis) but for different FPGA frequencies. The number of repetitions is 10 000 000, like in figure 5.9 as both use the nanoseconds scale. In all results, the task duration is fixed to 1 nanosecond. The results show that the frequency of FPGA device

also affects the effectiveness. The major the frequency, the fastest the execution of recurrent tasks in the FPGA task accelerators, and the lower the management overheads. All the results in this section have been computed at 100 MHz, which is pretty conservative for a modern FPGA (that can easily reach 300 or more MHz). The hardware management is too effective that it does not even need to work at these higher frequencies to cover nearly all the possible working periods.

5.7.3 Sensors Monitoring

The sensors monitoring is a common workload in industrial environments. Its purpose is collecting information from any kind of sensors with a given frequency to know the state of the elements and keeping track of it, or even reacting if some irregularity is detected. The workload is a perfect example of a recurrent system with tasks that correspond to the different actions performed every some time. A task may read the sensor's input value, check the latest registered value for a sensor, and (maybe) trigger some action, analyze a bunch of collected data, write the latest values to disk, etc.

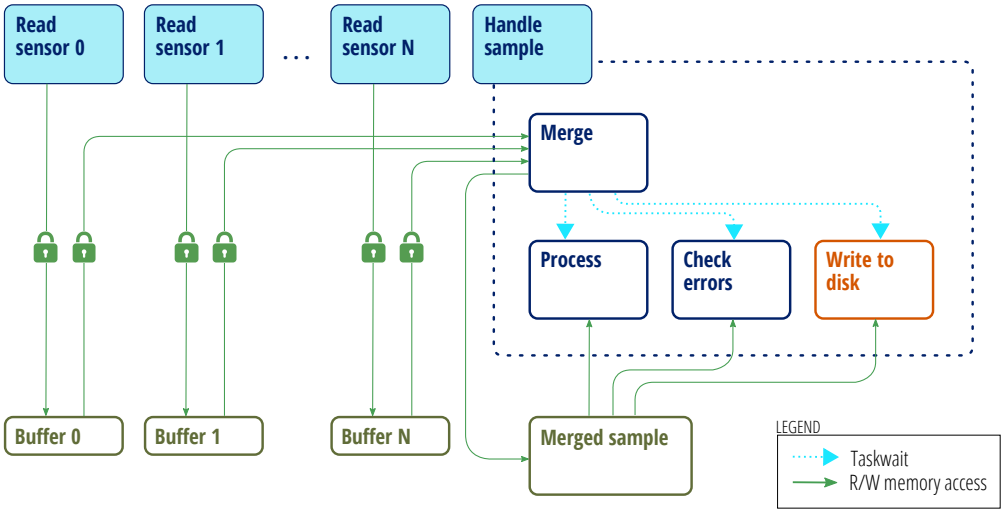


Figure 5.11: Tasks organization and memory regions of sensors monitoring benchmark (`cFPGA cri` configuration)

Figure 5.11 shows the tasks and memory regions used to implement the benchmark with the proposal enhancements. The blue boxes represent FPGA tasks and the orange box the SMP task (`Write to disk`). The blue boxes with a blue background are the recurrent tasks, the other (white background) are regular tasks. Moreover, the recurrent `Handle sample` task invokes different child tasks, which are grouped into a dashed box. The execution of those child tasks is ordered by a taskwait directive, which is represented by a doted sky-blue arrows in figure 5.11. The olive-green boxes represent the main

memory regions accessed by the tasks, and the green arrows the information flow between tasks through the regions. The green arrows with a key on top of them denote that this access is done using a critical region to ensure the atomicity of read/write operations.

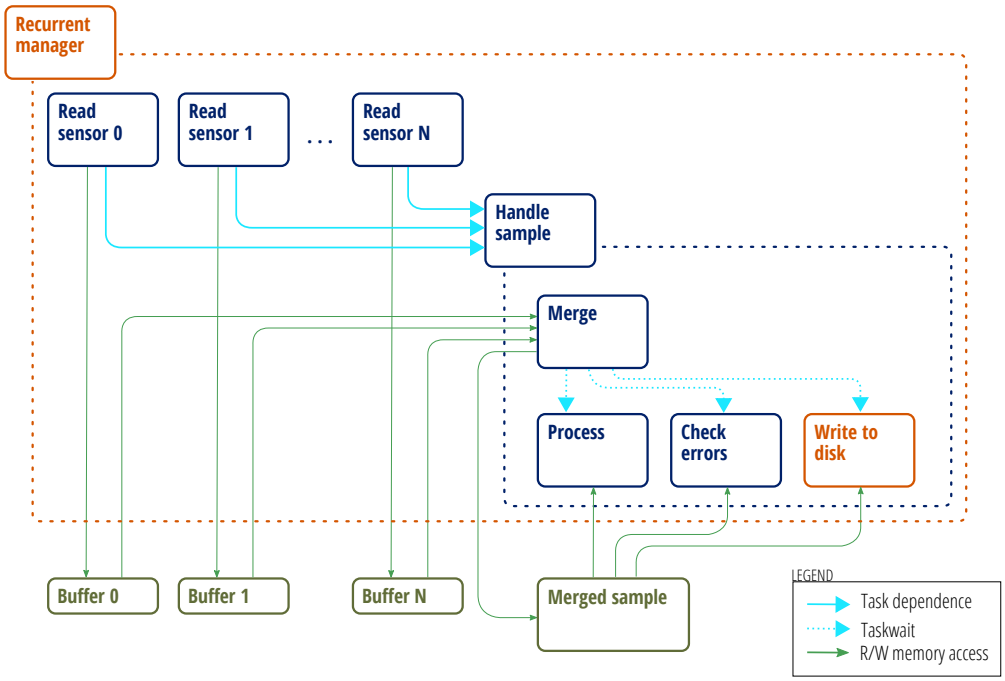


Figure 5.12: Tasks organization and memory regions of sensors monitoring benchmark (`cHost ncri` configuration)

The same benchmark has been implemented without using the proposal enhancements: the support for recurrent tasks and the critical regions in FPGA task accelerators. The support of recurrent tasks has been replaced by one additional task (either SMP task or FPGA task) that monitors the recurrent tasks and spawns them periodically (`Recurrent manager`). The critical regions that ensure the atomic access to shared buffers (`Buffer 0`, `Buffer 1`, etc.) have been replaced by task dependences in the child tasks (`Read sensor` and `Handle sample`) or explicit taskwaits in the `Recurrent manager` task. The synchronization between the recurrent tasks requires a single complex monitor task, in contrast to the several monitor tasks used in the synthetic benchmark. The pseudo-code of the SMP manager task is shown in listing 5.9. Also, the tasks and memory regions for the implementation with an SMP manager task and task dependences (`cHost ncri` configuration) are shown in figure 5.12. The figure uses the same color meanings of figure 5.11, but it adds the solid sky-blue arrows that represent the ordering of tasks due to data dependences.

```
#pragma omp task
void Recurrent_manager(buffer_sensor[N+1], merged_sample,
  num_reps[N+1], periods[N+1])
{
  double last_start[N+1];
  int reps_count[N+1];
  int num_finished = 0;
  for (unsigned int idx=0; idx<(N+1); idx++) {
    last_start[idx] = 0;
    reps_count[idx] = 0;
  }
  while (num_finished < (N+1)) {
    const double now = wall_time_us();
    if ((now - last_start[0]) > periods[0] &&
        reps_count[0] < num_reps[0])
    {
      Read_sensor0_task(buffer_sensor[0]);
      last_start[0] = now;
      reps_count[0]++;
      num_finished += reps_count[0] >= num_reps[0] ? 1 : 0;
    }
    if ((now - last_start[1]) > periods[1] &&
        reps_count[1] < num_reps[1])
    {
      Read_sensor1_task(buffer_sensor[1]);
      last_start[1] = now;
      reps_count[1]++;
      num_finished += reps_count[1] >= num_reps[1] ? 1 : 0;
    }

    // The if code-block is repeated for the N sensors.
    // ...

    if ((now - last_start[N]) > periods[N] &&
        reps_count[N] < num_reps[N])
    {
      Handle_sample(buffer_sensor[0] ... buffer_sensor[N-1], merged_sample);
      last_start[N] = now;
      reps_count[N]++;
      num_finished += reps_count[N] >= num_reps[N] ? 1 : 0;
    }
    #pragma omp taskwait
    #pragma omp taskyield
  }
}
```

Listing 5.9: Sensors monitoring benchmark pseudo-code without proposal extensions

The manager to implement the recurrent support without the proposal extensions requires some significant extra logic, as shown in listing 5.9. The `Recurrent manager` task has as parameters all other task parameters and two extra arrays with the number of repetitions (if they apply) and the different periods for each recurrent task. Then, the manager has to keep track of the last submit timestamp and the number of repetitions executed for each recurrent task. Also, a finalization condition (`num_finished`) has to be maintained to know when all recurrent tasks have finished and then break the manager loop. The loop iterations check if the current timestamp is behind the last start timestamp plus the specified period for each recurrent task. If so, the task is launched, and their state inside the manager is updated. Once all recurrent tasks have been checked, the manager waits for them and yields until more tasks have to be submitted.

The management without the proposed capabilities assumes that the duration of recurrent tasks is smaller than the smallest period. This is due to the taskwait after the task spawns (line 39 of listing 5.9). For durations larger than the smallest period, the management will delay some task spawns. In contrast, the proposal implementation does not have such limitation as the recurrent task are managed independently. Also, the host manager can only ensure the period between task spawns but not task starts, which may be delayed by the runtime when no executors are available or due to runtime overheads. The recurrent tasks management implemented in the proposal has a good period precision as the repetitions management is kept together with the executor (the FPGA task accelerator).

The different benchmark configurations considered in the comparison are:

- `cHost ncri`. Recurrent tasks management in host threads using the `Recurrent manager` task with data dependences to synchronize the child tasks. The `ncri` label stands for non-critical.

- `cHost cri`. Recurrent tasks management in host threads using the `Recurrent manager` task (like `cHost ncri`), but using the proposal critical regions instead of task data dependences to synchronize the child tasks.

- `cFPGA ncri`. Recurrent tasks management centralized in an FPGA task similar to `Recurrent manager`, but with an additional taskwait before `Handle samples` that orders the child tasks execution. This configuration requires the proposal enhancements to account time in the FPGA task accelerators.

- `cFPGA cri`. Recurrent tasks management implemented in each FPGA task accelerator and tasks synchronization through critical regions.

The sensors data is simulated in all executions to avoid the complexity of underlying communication protocols that are out of the thesis and proposal scope. Also, the period of `handle_sample` is fixed to one second, which is enough to handle the write to disk of sensor traces. The other periods are parameterized, and they take values between a base period and two times this base period.

Due to the different period precision of each configuration and the unconstrained duration of `Handle sample`, the effectiveness of an execution ($Effectiveness_{exec}$) is computed against the real execution time of the same configuration but with a period of 1 second for all `Read sensor` tasks ($Time_{ref}$). In the optimal case, both executions of the configuration should have the same execution time, as it is mainly determined by the one second period of `Handle sample` task that remains constant. Equation 5.3 shows how the effectiveness of an execution is computed.

$$Effectiveness_{exec} = Time_{ref}/Time_{exec} \tag{5.3}$$

Figure 5.13 shows the effectiveness (y-axis) for the four configurations among different base periods (x-axis). The number of repetitions for `Handle sample` task is fixed to 10, and there are two sensors in all executions. The results show two groups the `cFPGA` (blue lines) and the `cHost` (orange lines). The host configurations only maintain the effectiveness above 1 millisecond base period. In contrast, the FPGA configurations maintain the effectiveness above 10 microseconds base period, which is 100 times smaller. Moreover, the host configurations fail to execute with base periods below 100 microseconds due to an excessive memory consumption generated by the huge number of created tasks in Nanos++ runtime.

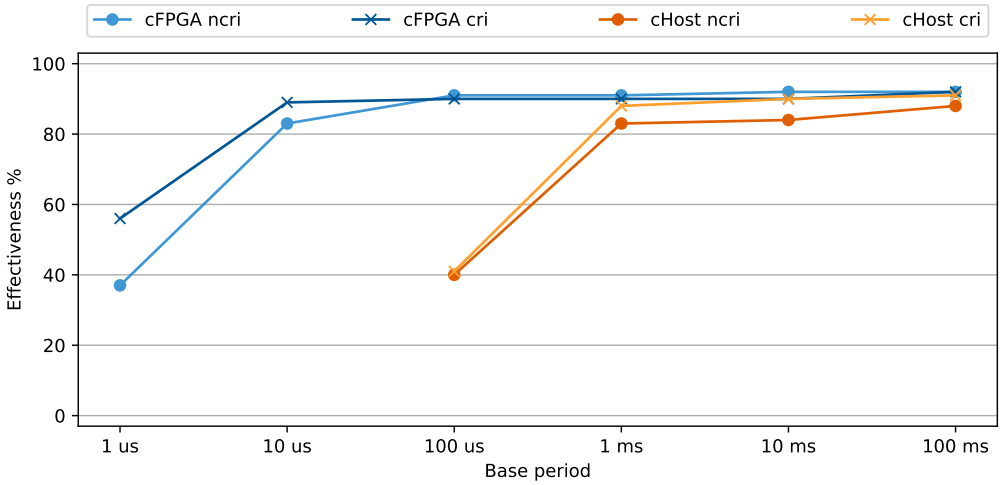


Figure 5.13: Effectiveness of sensors monitoring benchmark among base periods

The configurations that use the critical regions show better effectiveness in figure 5.13 than the configurations without it. The use of critical regions creates a fine-grain synchronization between the tasks, which has a low overhead thanks to the proposal support in the HWR. In addition, it can be seen that the support of the critical region adds more effectiveness the smaller the base period. This emphasizes their usefulness for the HWR approach.

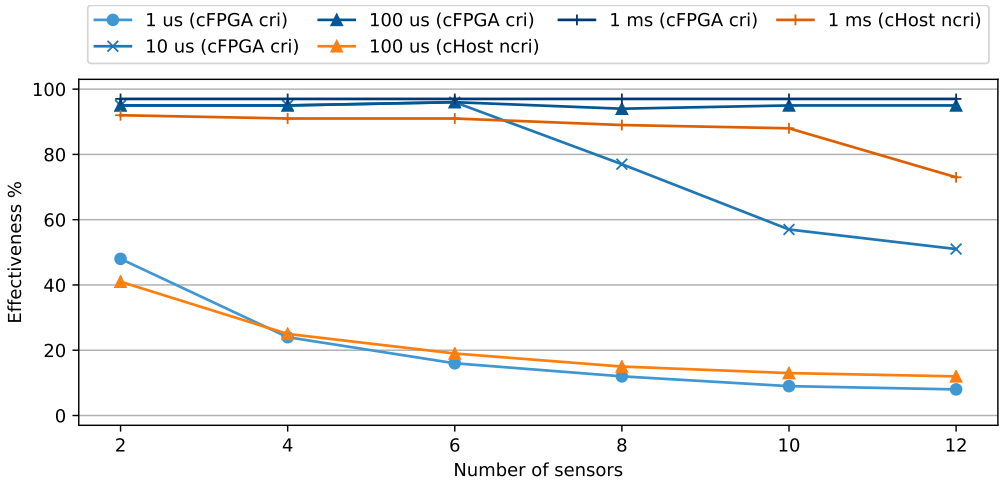


Figure 5.14: Effectiveness of sensors monitoring benchmark (only read) among number of FPGA task accelerators

Figure 5.14 shows the effectiveness (y-axis) for the small base periods of figure 5.13, where the configurations start to lose effectiveness, among different amounts of read sensor tasks. Figure 5.14 helps to see how the proposal and the baseline behave with different amounts of recurrent tasks (the read sensors in this case). The results confirm the 100x difference between both configurations seen in previous results. The host results are shown for 1 ms and 100 us base periods as the smaller ones fail to execute.

The proposal implementation effectively handles all amounts of read sensors for 1 ms and 100 us base periods. Meanwhile, the baseline host management loses some effectiveness with 12 recurrent tasks for 1 ms base period and suffers with the 100 us base period. The behavior observed for `cHost ncri` with base period 100 us is similar to the `cFPGA cri` with base period 1 us.

5.7.4 Face Detection

The detection of faces on pictures is a common workload in several scenarios and with different approaches. It is done offline in millions of smartphones to analyze the photos taken by users and cluster them by means of detected faces. Also, it is done in real-time

(in-place or off-place in data centers) to analyze and process a stream of pictures. This real-time approach is related to the proposal enhancements and could benefit from the new FPGA capabilities.

Several algorithms exist to detect faces on pictures using a wide range of techniques. The implementation used in this evaluation is based on Local Binary Patterns (LBP) [74], which searches local patterns in the picture and allows discarding regions very fast. The baseline implementation with OmpSs tasks was developed in the context of the AXIOM project [75] by one of the project partners. The implemented algorithm is iterative and realizes up to 1000 filters to each pixel. During the first 90 filters, an early prune is performed, and all pixels that do not have enough score after each filter are discarded. All pixels that pass the first prune go through the other 910 filters and obtain a final score, which determines if the pixel is the base coordinates of a face or not.

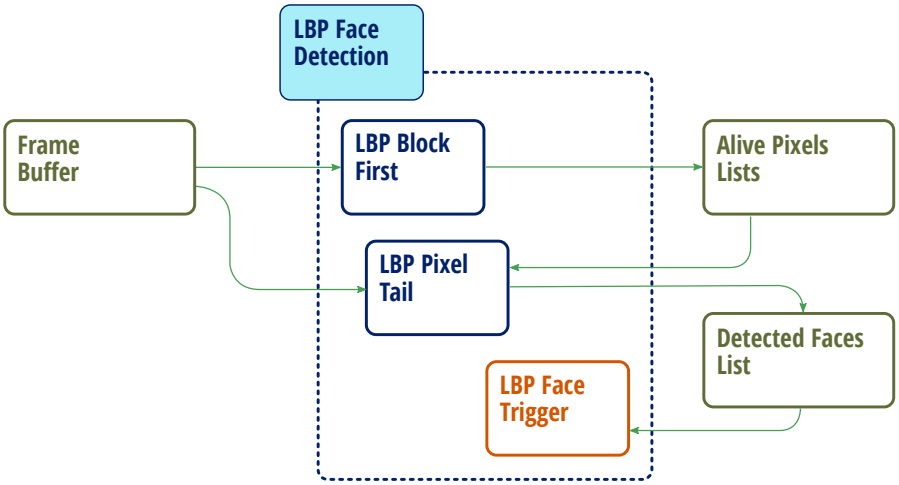


Figure 5.15: Tasks organization and memory regions of LBP Face Detection

Figure 5.15 shows an overview of task organization for the implementation developed with recurrent tasking (a portion of the source code is shown in listing A.4). The blue boxes represent the FPGA architecture tasks, the orange box (`LBP Face Trigger`) represents the host SMP architecture task. Moreover, the blue box with a light-blue background (`LBP Face Detection`) represents the recurrent task, which is periodically executed and spawns the other regular tasks. The olive-green boxes represent the main memory regions where tasks exchange information. `Frame Buffer` contains the color information for each pixel of the frame, and it is updated autonomously simulating a physical camera directly connected to the FPGA board. `Alive Pixels Lists` are a set of lists that contain the coordinates and the relevant information for each pixel that potentially contains a face. `Detected Faces List` is a list of face coordinates and scores that passed all LBP filters and obtained a minimum score.

Regarding tasks, the implementation is block-based and uses a wave approach where N `LBP Block First` tasks are spawned. Meanwhile, the results of the previous wave are check to launch the needed `LBP Pixel Tail` tasks. After a wave of task spawns, a taskwait is used to synchronize the executions and ensure that the next wave can be spawned without overwriting the data. Both tasks operate at different granularities (block of pixels and pixel) as it is more efficient in terms of performance per FPGA resource due to the sparsity of tail tasks. At the end of all LBP processing, a host SMP task is spawned if some faces are detected in the frame. The provided information contains the face coordinates and can be used to generate images like the frame shown in figure 5.16 (the faces have been pixelized to preserve people's privacy).

Figure 5.16: Output frame example with squares around detected faces

In all executions, the block size has been fixed to 96x96 effective pixels, which is increased to 144x144 pixels due to the padding needed by some filters. Also, the number of parallel blocks in each wave has been fixed to 4 blocks that may spawn up to 2304 alive pixels in the tail pass. More parallel blocks do not increase the performance but require more memory for the intermediate information in `Alive Pixels Lists`. The number of frames processed in each execution is 30, and they have been extracted from an input video that contains three people walking in a row (as can be seen in figure 5.16). This amount of frames provides long enough executions to gather significant executions for the performance analysis. Finally, 2 instances of `LBP Block First` could be placed together with 1 instance of `LBP Pixel Tail` and 1 instance of `LBP Face Detection`. In contrast to all previous benchmarks that also spawn task in the FPGA device, the recurrent `LBP Face Detection` does conditional task spawn based on the information read from memory (generated by the previous wave tasks).

Figure 5.17 shows the Frames Per Second (FPS) (y-axis) handled by the proposed recurrent implementation (`cFPGA`) and by an equivalent implementation without the FPGA task spawn support (`cHost`). The values are shown for different period values of `LBP Face Detection` task (x-axis). The maximum FPS value for each period ($FPS_{max}(period) = 1/period$) is also shown by the `Maximum` series.

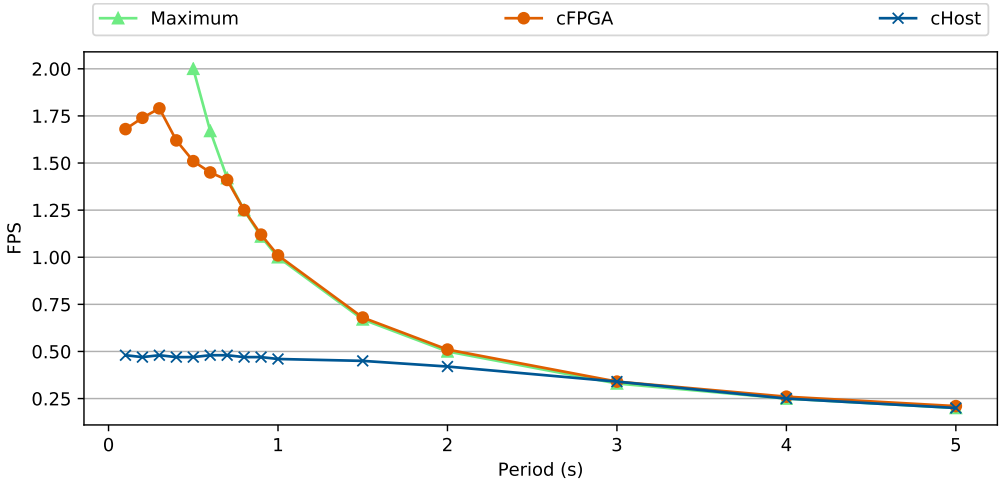
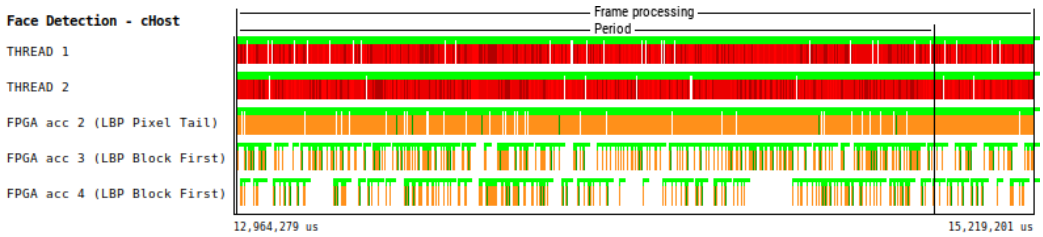


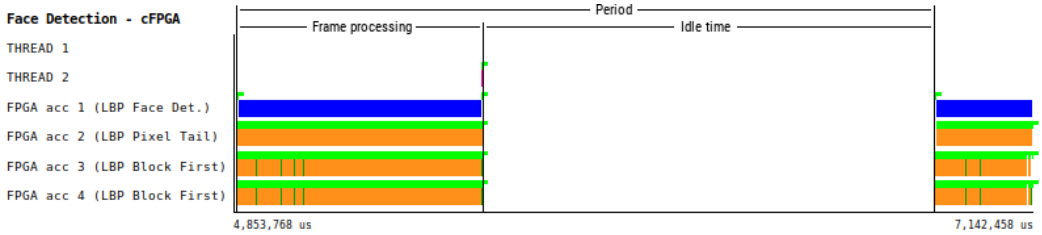
Figure 5.17: FPS of Face Detection

The results show that the `cHost` is not a performant option to handle the frames as the best rate does not arrive at 0.5 FPS. This means that `LBP Face Detection` lasts for 2 seconds on average, which is due to the data movements required to spawn the tail tasks. In contrast, the `cFPGA` implementation reaches up to 1.79 FPS. The peak is achieved with a 0.3 seconds period, which is approximately the duration of `LBP Face Detection` when there are no active pixels for the tail pass. The `cFPGA` performance is the maximum value for periods above 0.7 seconds, which is the average duration of `LBP Face Detection`. Between 0.3 and 0.7 seconds for the period, some frames are processed within the period time and some are not. Therefore, the performance scales but not at the maximum values.

Figure 5.18 shows two execution traces of Face Detection for `cHost` (figure 5.18a) and `cFPGA` (figure 5.18b) for a 2 seconds period. Both have the same elapsed time (x-axes) and show the activities in the different computing elements. The brown and red regions represent the offload of `LBP Block First` and `LBP Pixel Tail` tasks to the FPGA device correspondingly. The pink regions represent the execution of `LBP Face Trigger` task. Finally, the orange, blue and green regions represent the following activities in the FPGA trace lines: copying data into the FPGA task accelerator wrapper, executing the task statements and copying data back to FPGA memory.



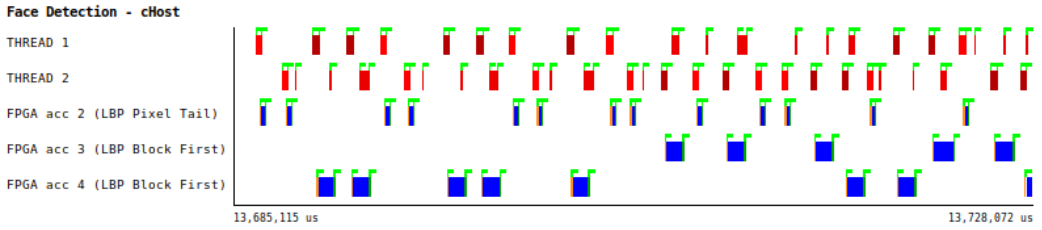
(a) Baseline host management



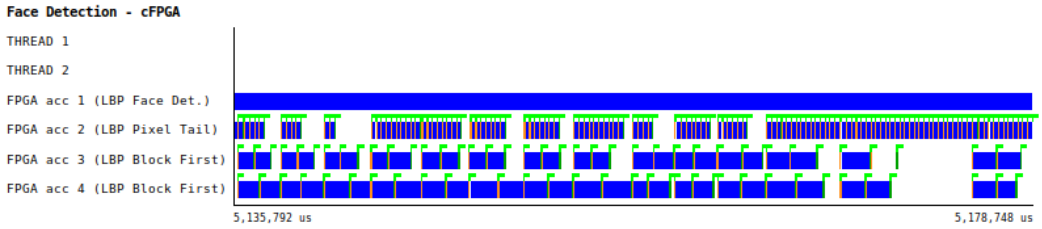
(b) Proposal management

Figure 5.18: Execution traces of Face Detection with 2 seconds period

Traces in figure 5.18 clearly show the sparsity of FPGA tasks in `cHost` and that they are compacted in `cFPGA`. The huge overheads in the case of host management result in an under-utilization of FPGA task accelerators and a poor frame rate. The `cHost` requires all the trace time (approximately 2.25 seconds) to handle the frame which is above the desired period. In contrast, the `cFPGA` lasts for 700 milliseconds, and the system remains idle until the next repetition is launched. Another relevant fact is that the host threads are not needed almost all the time. This optimizes the application power budget without impacting the overall performance.



(a) Baseline host management



(b) Proposal management

Figure 5.19: Execution traces of Face Detection with 2 seconds period (43 ms zoom)

Figure 5.19 shows the same two execution traces of figure 5.18 with the same color meanings but for a small portion of elapsed time (43 milliseconds). At that scale, the rendering interpolation does not hide the real activities at each computing element and allows to see the duration of each color region. In both cases (`cHost` and `cFPGA`), the tasks in the FPGA task accelerators last the same, but they are compacted in the case of proposal management and sparse in the baseline approach. Also, figure 5.19a shows the usage of 2 SMP threads to offload the FPGA tasks.

5.8 Conclusion

This chapter discuses an extension of task-based programming models with recurrent workloads concepts. The proposal introduces two clauses (`period` and `num_repetitions`) in the task directive to efficiently model recurrent workloads. Those clauses allow the applications to be developed with less effort, increase their maintainability and their accuracy. The FPGA task accelerators have been extended to internally handle the repetitions of recurrent tasks with minimal latency and maximum precision. In addition, the HWR and the host libraries have been updated to support the new features.

The proposal implementation includes the novel `Lock Manager` in the modular HWR to support the critical regions inside the FPGA tasks. That support provides a fine grain synchronization between independent tasks, which is very useful for recurrent workloads. Moreover, the locking capabilities led to a whole new set of possible programming features and optimizations. Although not presented here, internal FPGA locks have successfully been used to synchronize memory accesses in systems that require coordinated memory accesses for better throughput.

The evaluation shows the performance enhancements in terms of code lines, application throughput, and power efficiency. The evaluation of different workloads shows that applications can be easily programmed with the proposal extensions, and it shows the huge reduction of application complexity compared to the equivalent implementation with baseline capabilities. The direct management of repetitions in the FPGA task accelerators allows recurrent tasks of nanoseconds duration, which are not possible in the baseline, and accurate timing between repetitions. Besides, the proposal increases the power efficiency as management is kept near to the action, optimizing the execution time and reducing the communications.

5.9 Publications

The list of thesis publications related to the work explained in this chapter is:

- Towards recurrent tasks in OmpSs@FPGA.
  Jaume Bosch. HiPEAC CSW Autumn 2020. [76]
  This publication proposes a first implementation to support recurrent workloads in OmpSs and presents the initial results of the synthetic benchmark.

- Task-based programming models for heterogeneous recurrent workloads.
  Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Eduard Ayguadé.
  ARC 2021 [Accepted for publication]. [12]
  This publication presents the full implementation to support recurrent workloads in OmpSs and presents an extended evaluation with all performance results.

The list of publications related to collaborations with the work presented in this chapter is:

- OmpSs@FPGA framework for high performance FPGA computing.
  Juan Miquel de Haro, Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, Jesus Labarta.
  TC 2021 [Accepted for publication]. [19]
  In this work, the support for critical regions in the FPGA task accelerators has been used to tune some applications' performance.

Conclusion and Future Work

6

This chapter concludes the thesis with a final remark of its main contributions in section 6.1 and the future work in section 6.2.

6.1 Thesis Contributions

Heterogeneous platforms have become a key part of several computing environments to increase performance without increasing the power budget. Most of them are based on accelerators used by the main computation unit to offload specific parts of code. Task-based parallel programming models are a powerful tool to develop high-performance applications on top of such systems without the need to deal with the underlying details. This thesis goes beyond by enhancing the capabilities of heterogeneous systems and breaking some of the constraints established in the baseline task-based parallel programming models.

The overall thesis contribution is the enhancement of task-based parallel programming models capabilities. The thesis proposes breaking the master-slave model for co-processors management. This improves the applications' programmability, which can nest tasks regardless of the target architectures of parent and children tasks. Also, the programming model extension enlarges the range of supported workloads. Although the work is evaluated on SoC platforms, the contributions are suitable for other environments like discrete or network connected FPGAs. The proposals have been successfully tested and deployed in other environments for different thesis collaborations.

The first thesis proposal develops and analyzes several improvements to achieve an asynchronous, concurrent, and parameterizable behavior in task-based systems. On the one hand, the improvements allow the simpler and better expression of application requirements leading to a performance increment. For example, the `localmem` clause effortlessly defines the data cached in the FPGA task accelerators and the wide shared memory port feature in Mercurium compiler transparently increases the memory bandwidth. On the other hand, the proposal improvements allow more efficient management at runtime and a parameterizable behavior that may be tuned to each application. This has reduced the task management overheads increasing the accelerators utilization and

decreasing the importance of task granularity. All improvements could be analyzed in detail thanks to the new instrumentation capabilities implemented in the proposal.

The second thesis proposal extends the task-based parallel programming models with task spawn and synchronization support in co-processors. The implementation introduces the `Scheduler Manager` and `Taskwait Manager` IP blocks in the HWR. They are coordinated with the host runtime to ensure executions' correctness but optimizing the communications between runtimes to avoid the host-FPGA latency. The results show that the task spawn in FPGA devices can boost the applications' performance and reduce the power consumption as the host is exempt from management activities. Even more, the low-latency task spawn in FPGAs opens the possibility to use fine-grain tasks in the applications without hurting performance.

Finally, the third thesis proposal extends the task-based programming models with recurrent workloads concepts. The proposal introduces two clauses in the task directive (`period` and `num_repetitions`) to efficiently model recurrent workloads. Moreover, the implementation has been extended to support critical regions between FPGA task accelerators to synchronize independent recurrent tasks. The benchmarks evaluation shows a huge gap in the application programmability against an equivalent implementation without the novel capabilities. Besides, the evaluation shows the efficient management of recurrent tasks when performed in FPGA devices. This management perfectly suits the necessities of new cyber-physical and embedded devices in terms of power consumption and high throughput.

6.2 Future Work

All the work done in the thesis has been contributed to the main OmpSs@FPGA ecosystem, and it will be used in different European projects. Then, a future work line is porting and tuning the applications available in the European projects. Those applications could benefit from the new FPGA task accelerators capabilities and obtain a performance that was not possible before. Moreover, the new workloads with new task patterns could reveal bottlenecks in the proposed designs that must be addressed.

The baseline OmpSs@FPGA ecosystem has support for multi-node executions using a network communication layer based on GASNet [77]. Although, it is a host-centric implementation as all data transferred between nodes is managed in the host runtime. The new capabilities added to the FPGAs could be further extended to manage network data transfers between nodes. Then the FPGAs could exchange tasks and offload tasks between them without involving the host runtimes. This FPGA to FPGA communication

will also benefit the data exchange on step-based benchmarks, like N-Body. In general, all benchmarks that need to synchronize the data scattered among all nodes periodically.

The proposal for recurrent workloads does not develop the timing constraints for the model. The real-time systems, a sub-set of recurrent workloads, have strong timing constraints that must be accomplished to assure that the systems will behave correctly. Those constraints have been analyzed and developed in different related works, but this thesis focused on recurrent task management instead. Therefore, the current support could be enhanced with new capabilities that improve the real-time constraints and robustness. As an example, the programming model could be further extended with deadline information and how the system must behave in those cases.

The proposals in this thesis have been evaluated over the OmpSs programming model, which is an OpenMP forerunner. The mid-term goal is to use all the developed infrastructure for the OmpSs-2 programming model which is replacing OmpSs. However, the long-term goal is to contribute the ideas and the knowledge of this thesis to the OpenMP programming model. This would benefit a wider range of people as the proposals will be in an established and standard programming model. The end goal is developing an implementation of those extensions over some of the open-source OpenMP implementations. Therefore, the proposals could be extensively and intensively tested by the community in a broader range of applications and systems.

Bibliography

[1] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. "Design of Ion-Implanted MOSFET'S with very small physical dimensions". In: *Solid-State Circuits, IEEE Journal of* 9 (Nov. 1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[2] Gordon Moore. "Cramming More Components Onto Integrated Circuits". In: *Electronics* 38 (Apr. 1965).

[4] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, et al. "Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures." In: *Parallel Processing Letters* 21 (June 2011), pp. 173–193. DOI: 10.1142/S0129626411000151.

[5] Leonardo Dagum and Ramesh Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *Computational Science & Engineering, IEEE* 5 (Feb. 1998), pp. 46–55. DOI: 10.1109/99.660313.

[6] Jaume Bosch, Antonio Filgueras, Miquel Vidal Piñol, et al. "Exploiting Parallelism on GPUs and FPGAs with OmpSs". In: *Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*. Sept. 2017, pp. 1–5. ISBN: 978-1-4503-5363-2. DOI: 10.1145/3152821.3152880.

[7] Jaume Bosch, Xubin Tan, Carlos Álvarez, et al. "Asynchronous Task Creation for Task-Based Parallel Programming Runtimes". In: *OpenMP Developers Conference*. 2018.

[8] Jaume Bosch, Xubin Tan, Antonio Filgueras, et al. "Application Acceleration on FPGAs with OmpSs@FPGA". In: *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE. Dec. 2018. DOI: 10.1109/FPT.2018.00021.

[9] Jaume Bosch Pons, Carlos Álvarez Martínez, and Daniel Jiménez-González. "Supporting task creation inside FPGA devices". In: *Book of abstracts*. Barcelona Supercomputing Center. 2019, pp. 34–35.

[10] Jaume Bosch, Miquel Vidal Piñol, Antonio Filgueras, et al. "Breaking master-slave model between host and FPGAs". In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Feb. 2020, pp. 419–420. DOI: 10.1145/3332466.3374545.

[11] Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, and Eduard Ayguadé. "Asynchronous Runtime with Distributed Manager for Task-based Programming Models". In: *Parallel Computing* (2020). DOI: 10.1016/j.parco.2020.102664.

[12] Jaume Bosch, Antonio Filgueras, Miquel Vidal Piñol, et al. "Task-based programming models for heterogeneous recurrent workloads". In: *Proceedings of the 2021 International Symposium on Applied Reconfigurable Computing [Accepted for publication]*. June 2021. ISBN: 978-3-030-79024-0.

[13]Xubin Tan, Jaume Bosch, Miquel Vidal Piñol, et al. "Picos, A Hardware Task-Dependence Manager for Task-Based Dataflow Programming Models". In: *International Conference on High Performance Computing & Simulation (HPCS)*. July 2017, pp. 878–880. DOI: 10.1109/HPCS.2017.134.

[14]Xubin Tan, Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, and Eduard Ayguade. "Hardware Heterogeneous Task Scheduling for Task-based Programming Models". In: *OpenMP Developers Conference*. 2018.

[15]Kallia Chronaki, Marc Casas, Miquel Moreto, Jaume Bosch, and Rosa M. Badia. "TaskGenX: A Hardware-Software Proposal for Accelerating Task Parallelism". In: *International Conference on High Performance Computing*. Jan. 2018, pp. 389–409. ISBN: 978-3-319-92039-9. DOI: 10.1007/978-3-319-92040-5_20.

[16]Xubin Tan, Jaume Bosch, Carlos Álvarez, et al. "A Hardware Runtime for Task-Based Programming Models". In: *IEEE Transactions on Parallel and Distributed Systems* 30 (Mar. 2019), pp. 1932–1946. DOI: 10.1109/TPDS.2019.2907493.

[17]Lucas Morais, Vitor Silva, Alfredo Goldman, et al. "Adding Tightly-Integrated Task Scheduling Acceleration to a RISC-V Multi-core Processor". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Oct. 2019, pp. 861–872. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358271.

[18]Juan Miguel de Haro, Jaume Bosch, Daniel Jiménez-González, and Carlos Álvarez. "Design and implementation of an architecture-aware hardware runtime for heterogeneous systems". In: *Book of abstracts*. Barcelona Supercomputing Center. 2020, pp. 58–59.

[19]Juan Miguel de Haro, Jaume Bosch, Antonio Filgueras, et al. "OmpSs@FPGA framework for high performance FPGA computing". In: *IEEE Transactions on Computers [Accepted for publication]* Compiler Optimizations for FPGA-Based Systems (2021). DOI: 10.1109/TC.2021.3086106.

[20]Cesar González, Jaume Bosch, Juan Miquel de Haro, et al. "High Performance Computing particle-pair distance algorithms, to generate X-ray spectra from 3D models". In: *International Journal of High Performance Computing Applications [Under review]* (2021).

[23]Florentino Sainz, Sergi Mateo, Vicenc Beltran, et al. "Leveraging OmpSs to Exploit Hardware Accelerators". In: *Proceedings - Symposium on Computer Architecture and High Performance Computing* (Dec. 2014), pp. 112–119. DOI: 10.1109/SBAC-PAD.2014.26.

[28]Miquel Vidal-Piñol. "Synchronization/communication techniques for OmpSs@FPGA". 2017.

[30]Stephen Neuendorffer and Fernando Martinez-Vallina. "Building zynq® accelerators with Vivado® high level synthesis". In: *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. Feb. 2013, pp. 1–2. DOI: 10.1145/2435264.2435266.

[32]Jaume Bosch. "Asynchronous Runtime for Task-Based Dataflow Programming Models". 2017.

[33]Jaume Bosch, Xubin Tan, Carlos Álvarez, et al. "Characterizing and Improving the Performance of Many-Core Task-Based Parallel Programming Runtimes". In: *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2017, pp. 1285–1292. DOI: 10.1109/IPDPSW.2017.32.

[40]Iakovos Mavroidis, Ioannis Papaefstathiou, Luciano Lavagno, et al. "ECOSCALE: Reconfigurable Computing and Runtime System for Future Exascale Systems". In: *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Jan. 2016, pp. 696–701. DOI: 10.3850/9783981537079_1021.

[42]Andrew Canis, Jongsok Choi, Mark Aldham, et al. "LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems". In: *ACM Transactions on Embedded Computing Systems (TECS)* 13 (Sept. 2013). DOI: 10.1145/2514740.

[43]Jongsok Choi, Stephen Brown, and Jason Anderson. "From Pthreads to Multicore Hardware Systems in LegUp High-Level Synthesis for FPGAs". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25 (Aug. 2017), pp. 1–14. DOI: 10.1109/TVLSI.2017.2720623.

[44]Artur Podobas, Mats Brorsson, and Vladimir Vlassov. "TurboBŁYSK: scheduling for improved data-driven task performance with fast dependency resolution". In: *Using and Improving OpenMP for Devices, Tasks, and More*. Springer, 2014, pp. 45–57. ISBN: 978-3-319-11454-5. DOI: 10.1007/978-3-319-11454-5_4.

[45]George Bosilca, Aurelien Bouteiller, Anthony Danalis, et al. "DAGuE: A generic distributed DAG engine for High Performance Computing". In: *Parallel Computing* 38 (May 2011), pp. 1151–1158. DOI: 10.1016/j.parco.2011.10.003.

[46]Carsten Heinz, Jaco Hofmann, Lukas Sommer, and Andreas Koch. "Improving Job Launch Rates in the TaPaSCo FPGA Middleware by Hardware/Software-Co-Design". In: *Proceedings of 2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*. Nov. 2020, pp. 22–30. DOI: 10.1109/ROSS51935.2020.00008.

[47]Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. "HPX: A Task Based Programming Model in a Global Address Space". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. Oct. 2014, p. 6. ISBN: 978-1-4503-3247-7. DOI: 10.1145/2676870.2676883.

[48]Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, et al. "Asynchronous Nested Parallelism for Dynamic Applications in Distributed Memory". In: *Languages and Compilers for Parallel Computing*. Feb. 2016, pp. 106–121. ISBN: 978-3-319-29777-4. DOI: 10.1007/978-3-319-29778-1_7.

[49]Tianyi Zhang, Shahrzad Shirzad, Patrick Diehl, et al. "An Introduction to hpxMP: A Modern OpenMP Implementation Leveraging HPX, An Asynchronous Many-Task System". In: *Proceedings of the International Workshop on OpenCL*. May 2019, pp. 1–10. ISBN: 978-1-4503-6230-6. DOI: 10.1145/3318170.3318191.

[50]Jeremy Kemp and Barbara Chapman. "Mapping OpenMP to a Distributed Tasking Runtime". In: *Evolving OpenMP for Evolving Architectures*. Jan. 2018, pp. 222–235. ISBN: 978-3-319-98520-6. DOI: 10.1007/978-3-319-98521-3_15.

[51]Cor Meenderinck and Ben Juurlink. "A Case for Hardware Task Management Support for the StarSS Programming Model". In: *Proceedings of the 13th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*. Sept. 2010, pp. 347–354. DOI: 10.1109/DSD.2010.63.

[52]Tamer Dallou, Nina Engelhardt, Ahmed Elhossini, and Ben Juurlink. "Nexus#: A Distributed Hardware Task Manager for Task-Based Programming Models". In: *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2015, pp. 1129–1138. DOI: 10.1109/IPDPS.2015.79.

[53]Xubin Tan, Jaume Bosch, Miquel Vidal Piñol, et al. "General Purpose Task-Dependence Management Hardware for Task-Based Dataflow Programming Models". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2017, pp. 244–253. DOI: 10.1109/IPDPS.2017.48.

[55]Jan Vesely, Arkaprava Basu, Abhishek Bhattacharjee, et al. "Generic System Calls for GPUs". In: *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. June 2018, pp. 843–856. DOI: 10.1109/ISCA.2018.00075.

[56]Cheng Chen, Wenxiang Yang, Fang Wang, et al. "Reverse Offload Programming on Heterogeneous Systems". In: *IEEE Access* 7 (Jan. 2019), pp. 10787–10797. DOI: 10.1109/ACCESS.2019.2891740.

[57]Daniel Cabrera, Xavier Martorell, Georgi Gaydadjiev, Eduard Ayguade, and Daniel Jiménez-González. "OpenMP extensions for FPGA accelerators". In: *International Symposium on Systems, Architectures, Modeling, and Simulation*. Aug. 2009, pp. 17–24. DOI: 10.1109/ICSAMOS.2009.5289237.

[58]Lukas Sommer, Jens Korinth, and Andreas Koch. "OpenMP device offloading to FPGA accelerators". In: *IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. July 2017, pp. 201–205. DOI: 10.1109/ASAP.2017.7995280.

[59]Maria A. Serrano, Sara Royuela, and Eduardo Quiñones. "Towards an OpenMP Specification for Critical Real-Time Systems". In: *Evolving OpenMP for Evolving Architecture*. Jan. 2018, pp. 143–159. ISBN: 978-3-319-98520-6. DOI: 10.1007/978-3-319-98521-3_10.

[60]Antoniu Pop and Albert Cohen. "A Stream-Comptuting Extension to OpenMP". In: *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC '11)*. Jan. 2011, pp. 5–14. DOI: 10.1145/1944862.1944867.

[61]Germán Llort, Antonio Filgueras, Daniel Jiménez-González, et al. "The Secrets of the Accelerators Unveiled: Tracing Heterogeneous Executions Through OMPT". In: *International Workshop on OpenMP*. Oct. 2016, pp. 217–236. ISBN: 978-3-319-45549-5. DOI: 10.1007/978-3-319-45550-1_16.

[62]Michael Wagner, Germán Llort, Antonio Filgueras, et al. "Monitoring Heterogeneous Applications with the OpenMP Tools Interface". In: *Tools for High Performance Computing 2016*. July 2017. DOI: 10.1007/978-3-319-56702-0_3.

[64]Departament Computadors, Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. "PARAVER: A tool to visualize and analyze parallel code". In: *World occam and Transputer User Group Technical Meeting (WoTUG-18)* 44 (Mar. 1995).

[65]Satish Kumar Sadasivam, Brian Thompto, Ron Kalla, and William Starke. "IBM Power9 Processor Architecture". In: *IEEE Micro* 37 (Mar. 2017), pp. 40–51. DOI: 10.1109/MM.2017.40.

[66]Fabio Banchelli, Marta Garcia-Gasulla, Guillaume Houzeaux, and Filippo Mantovani. "Benchmarking of State-of-the-Art HPC Clusters with a Production CFD Code". In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC '20. June 2020, pp. 1–11. ISBN: 9781450379939. DOI: 10.1145/3394277.3401847.

[71]Xiaoming Li, María Garzarán, and David Padua. "A Dynamically Tuned Sorting Library." In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* Mar. 2004, pp. 111–122. DOI: 10.1109/CGO.2004.1281668.

[74]Timo Ojala, Matti Pietikäinen, and Topi Mäenpää. "Gray Scale and Rotation Invariant Texture Classification with Local Binary Patterns". In: *Computer Vision - ECCV 2000*. June 2000, pp. 404–420. ISBN: 978-3-540-67685-0. DOI: 10.1007/3-540-45054-8_27.

[75]Antonio Filgueras, Paolo Gai, Stefano Garzarella, et al. "The AXIOM Project: IoT on Heterogeneous Embedded Platforms". In: *IEEE Design & Test* PP (Nov. 2019), pp. 1–1. DOI: 10.1109/MDAT.2019.2952335.

[76]Jaume Bosch. "Towards recurrent tasks in OmpSs@FPGA". In: *HiPEAC CSW Autumn 2020*. 2020.

[77]Dan Bonachea and Jaein Jeong. "GASNet: A portable high-performance communication layer for global address-space languages". In: *CS258 Parallel Computer Architecture Project, Spring* (2002).

Webpages

[@3]OpenACC Board. *OpenACC Application Program Interface*. 2011. URL: https://www.openacc.org/sites/default/files/inline-files/OpenACC_1_0_specification.pdf (visited on June 1, 2020).

[@21]OpenMP Architecture Review Board. *OpenMP Application Program Interface - Version 3.1*. 2011. URL: https://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf (visited on Feb. 20, 2021).

[@22]OpenMP Architecture Review Board. *OpenMP API Specification: Version 5.1*. 2020. URL: https://www.openmp.org/spec-html/5.1/openmp.html (visited on Feb. 20, 2021).

[@24]Programming Models Group BSC. *OmpSs Specification*. 2018. URL: https://pm.bsc.es/ftp/ompss/doc/spec/ (visited on June 1, 2020).

[@25]Programming Models Group BSC. *Mercurium C/C++/Fortran source-to-source compiler*. 2018. URL: https://github.com/bsc-pm/mcxx (visited on Apr. 2, 2020).

[@26]Programming Models Group BSC. *OmpSs-2 Specification*. 2020. URL: https://pm.bsc.es/ftp/ompss-2/doc/spec/ (visited on May 4, 2021).

[@27]Programming Models Group BSC. *Nanos6 Runtime*. 2020. URL: https://github.com/bsc-pm/nanos6 (visited on May 4, 2021).

[@29]Inc. Xilinx. *Vivado High-Level Synthesis*. 2017. URL: https://www.xilinx.com/hls (visited on May 20, 2020).

[@31]Programming Models Group BSC. *Nanos++ Runtime Library*. 2018. URL: https://github.com/bsc-pm/nanox (visited on Apr. 2, 2020).

[@34]ARM. *AMBA® 4 AXI4-Stream Protocol*. 2010. URL: https://static.docs.arm.com/ihi0051/a/IHI0051A_amba4_axi4_stream_v1_0_protocol_spec.pdf (visited on May 20, 2020).

[@35]ARM. *AMBA® AXI™ and ACE™ Protocol Specification*. 2011. URL: https://static.docs.arm.com/ihi0022/d/IHI0022D_amba_axi_protocol_spec.pdf (visited on May 20, 2020).

[@36]Vineyard Consortium. *Objectives and Rationales of the Project*. 2017. URL: http://www.vineyard-h2020.eu/en/project/objectives-and-rationale-of-the-project/ (visited on May 20, 2020).

[@37]Inc. Maxeler. *The Open Spatial Programming Language*. 2014. URL: https://openspl.org (visited on May 20, 2020).

[@38]Inc. Khronos Group. *OpenCL*. 2018. URL: https://www.khronos.org/opencl (visited on May 20, 2020).

[@39]Inc. Xilinx. *SDSoC Development Environment*. 2020. URL: https://www.xilinx.com/sdsoc (visited on May 20, 2020).

[@41]MPI Forum. *MPI 4.0*. 2020. URL: https://www.mpi-forum.org/mpi-40/ (visited on Dec. 2, 2020).

[@54]NVIDIA. *CUDA Dynamic Parallelism Programming Guide*. 2019. URL: https://docs.nvidia.com/cuda/pdf/CUDA_Dynamic_Parallelism_Programming_Guide.pdf (visited on June 15, 2020).

[@63]BSC Tools Group. *Extrae*. 2020. URL: https://tools.bsc.es/extrae (visited on Jan. 13, 2021).

[@67]Xilinx, Inc. *ZYNQ UltraScale+ MPSoC Overview*. 2019. URL: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html (visited on June 1, 2020).

[@68]Programming Models Group BSC. *BSC Application Repository*. 2017. URL: https://pm.bsc.es/projects/bar/ (visited on Apr. 28, 2018).

[@69]Zhang Xianyi, Wang Qian, and Werner Saar. *OpenBLAS: An optimized BLAS library*. 2020. URL: https://www.openblas.net/ (visited on June 1, 2020).

[@70]Programming Models Group BSC. *BAR-Benchmarks [at] INTERTWinE*. 2017. URL: https://pm.bsc.es/gitlab/ompss/bar-benchmarks/ (visited on Apr. 28, 2018).

[@72]Avnet. *ZedBoard Technical Specifications*. 2020. URL: http://zedboard.org/content/zedboard-0 (visited on Sept. 25, 2020).

[@73]OpenMP Architecture Review Board. *OpenMP API Specification: Version 5.0 | taskyield Construct*. 2018. URL: https://www.openmp.org/spec-html/5.0/openmpsu49.html (visited on Sept. 29, 2020).

Appendix

A

Code	Description	Size (words)
0x01	Execute task command	3 + 2 x NUM_ARGS
0x02	Setup instrumentation command	2
0x03	Finished execution task command	2
0x05	Execute recurrent task command	4 + 2 x NUM_ARGS

Table A.1: FPGA commands information

```
typedef enum {
  XTASKS_SUCCESS = 0,   ///< Operation finished successfully
  XTASKS_ENOSYS,        ///< Function not implemented
  XTASKS_EINVAL,        ///< Invalid operation arguments
  XTASKS_ENOMEM,        ///< Not enough memory to execute the operation
  XTASKS_EFILE,         ///< Operation finished after fail a file operation
  XTASKS_ENOENTRY,      ///< Operation failed as no entry could be reserved
  XTASKS_PENDING,       ///< Operation not finished yet
  XTASKS_ENOAV,         ///< Function/operation not available
  XTASKS_ERROR          ///< Operation finished with some error
} xtasks_stat;
```

Listing A.1: xTasks general definition for all APIs of xtasks_stat type

```
typedef ap_uint<72> portData_t;
typedef ap_axis<64,1,8,5> axiData_t;
typedef hls::stream<axiData_t> axiStream_t;

extern const unsigned char accID;

void eOut_Adapter(volatile portData_t& in, axiStream_t& out) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_hs port=in bundle=in
#pragma HLS INTERFACE axis port=out
#pragma HLS PROTOCOL fixed
   portData_t inTmp = in;
   axiData_t outTmp = {0, 0, 0, 0, 0, 0, 0};
   outTmp.keep = 0xFF;
   outTmp.id = accID;
   outTmp.last = inTmp & 0x3;
   inTmp = inTmp >> 2;
   outTmp.dest = inTmp & 0x3F;
   inTmp = inTmp >> 6;
   outTmp.data = inTmp;
   out.write(outTmp);
}
```

Listing A.2: HLS implementation for eOut Adapter

```
typedef ap_uint<8> portData_t;
typedef ap_axis<8,1,1,5> axiData_t;
typedef hls::stream<axiData_t> axiStream_t;

void eIn_Adapter(axiStream_t& in, portData_t& out) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=in
#pragma HLS INTERFACE ap_hs port=out
   portData_t inTmp = in.read().data;
   out = inTmp;
}
```

Listing A.3: HLS implementation for eIn Adapter

```
#pragma omp target device(fpga) num_instances(2) \
copy_in( \
  [4*CONST_LBP_NUMSTAGES_FIRST_PASS]lbp_filters, \
  [CONST_LBP_NUMSTAGES_FIRST_PASS*CONST_LBP_NUMALPHAS]lbp_alphas, \
  [CONST_LBP_NUMSTAGES_FIRST_PASS]lbp_ths \
) \
copy_out( \
  [CONST_MAX_DET_ALIVEPX]alive_xx, \
  [CONST_MAX_DET_ALIVEPX]alive_yy, \
  [CONST_MAX_DET_ALIVEPX]alive_score, \
  [1]alive_count \
) \
localmem( \
  [4*CONST_LBP_NUMSTAGES_FIRST_PASS]lbp_filters, \
  [CONST_LBP_NUMSTAGES_FIRST_PASS]lbp_ths, \
  [CONST_MAX_DET_ALIVEPX]alive_xx, \
  [CONST_MAX_DET_ALIVEPX]alive_yy, \
  [CONST_MAX_DET_ALIVEPX]alive_score, \
  [1]alive_count \
)
#pragma omp task label(lbp_face_detection_first)
void lbp_face_detection_first(unsigned char* frame,
  const int x, const int y, const int width, const int height,
  const char* lbp_filters, const float* lbp_alphas, const float* lbp_ths,
  unsigned char* alive_xx, unsigned char* alive_yy, float* alive_score,
  int* alive_count);


#pragma omp target device(fpga) num_instances(1) \
copy_in( \
  [1]score, \
  [4*CONST_LBP_NUMSTAGES_TAIL_PASS]lbp_filters, \
  [CONST_LBP_NUMSTAGES_TAIL_PASS*CONST_LBP_NUMALPHAS]lbp_alphas \
) \
copy_inout( \
  [CONST_MAX_DET_FACES]f_list_faces, [1]f_list_size \
) \
localmem( \
  [1]score, \
  [4*CONST_LBP_NUMSTAGES_TAIL_PASS]lbp_filters \
)
#pragma omp task label(lbp_face_detection_tail)
void lbp_face_detection_tail_px(unsigned char* frame,
  const int x, const int y, const int width, const int height,
  float* score, const char* lbp_filters, const float* lbp_alphas,
  const float threshold, struct face* f_list_faces, int* f_list_size);


#pragma omp target device(smp) \
copy_in( \
  [CONST_MAX_DET_FACES]f_list_faces, \
  [1]f_list_size \
)
#pragma omp task
```

```
void lbp_facedetection_trigger(struct face* f_list_faces, int* f_list_size);


void lbp_face_detection_tail_block(unsigned char* frame,
  const int x, const int y, const int width, const int height,
  const char* lbp_filters, const float* lbp_alphas, const float threshold,
  unsigned char* alive_xx, unsigned char* alive_yy, float* alive_score,
  int* alive_count, struct face* f_list_faces, int* f_list_size)
{
  #pragma HLS INLINE

  //Iterate all alive pixels
  int xx, yy, idx, num_px;
  num_px = *alive_count;
  //NOTE: Sanity check that ensures there are no more px than possible
  num_px = num_px <= MAX_DET_ALIVEPX_FIRST_PASS ? num_px : 0;
  for (idx = 0; idx < num_px; idx++) {
    xx = alive_xx[idx];
    yy = alive_yy[idx];
    lbp_face_detection_tail_px(frame, x + xx, y + yy, width, height,
      alive_score + idx, lbp_filters, lbp_alphas,
      threshold, f_list_faces, f_list_size);
  }
}


#pragma omp target device(fpga) period(PERIOD) num_repetitions(30) \
copy_in( \
  [4*CONST_LBP_NUMSTAGES]lbp_filters, \
  [CONST_LBP_NUMSTAGES*CONST_LBP_NUMALPHAS]lbp_alphas, \
  [CONST_LBP_NUMSTAGES_FIRST_PASS]lbp_ths, \
  [CONST_LBP_PAR_BLOCKS*CONST_MAX_DET_ALIVEPX]alive_xx, \
  [CONST_LBP_PAR_BLOCKS*CONST_MAX_DET_ALIVEPX]alive_yy, \
  [CONST_LBP_PAR_BLOCKS*CONST_MAX_DET_ALIVEPX]alive_score, \
  [CONST_LBP_PAR_BLOCKS]alive_count, \
  [CONST_MAX_DET_FACES]f_list_faces, [1]f_list_size \
)
#pragma omp task label(lbp_face_detection_task)
void lbp_face_detection_task(unsigned char* frame, int width, int height,
  const char* lbp_filters, const float* lbp_alphas, const float* lbp_ths,
  const float threshold,
  unsigned char* alive_xx, unsigned char* alive_yy, float* alive_score,
  int* alive_count,
  struct face* f_list_faces, int* f_list_size)
{
  int x2, x, y2, y, n;
  unsigned int first_pass = 1;
  int n_x[LBP_PAR_BLOCKS];
  int n_y[LBP_PAR_BLOCKS];

  f_list_size[0] = 0;
  n = 0;
  //Iterate the blocks with jumps to avoid peaks of alive pixels
  for(x2 = 0; x2 < (width-LBP_RESOLUTION)/4; x2 += LBP_BSIZE)
  {
```

```
110   for(x = x2; x < width-LBP_RESOLUTION; x += (width-LBP_RESOLUTION)/4)
111   {
112     for(y2 = 0; y2 < (height-LBP_RESOLUTION)/2; y2 += LBP_BSIZE)
113     {
114     for(y = y2; y < height-LBP_RESOLUTION; y += (height-LBP_RESOLUTION)/2)
115     {
116       unsigned char* n_alive_xx = alive_xx + n*MAX_DET_ALIVEPX;
117       unsigned char* n_alive_yy = alive_yy + n*MAX_DET_ALIVEPX;
118       float* n_alive_score = alive_score + n*MAX_DET_ALIVEPX;
119       int* n_alive_count = alive_count + n;
120
121       lbp_face_detection_first(frame, x, y, width, height,
122         lbp_filters, lbp_alphas, lbp_ths,
123         n_alive_xx, n_alive_yy, n_alive_score, n_alive_count);
124
125       if ((n+1) == CONST_LBP_PAR_BLOCKS/2 ||
126           (n+1) == CONST_LBP_PAR_BLOCKS)
127       {
128         if (first_pass) {
129           first_pass = 0;
130         } else {
131           //Iterate the half previous block chunk which can be handled in
132           //parallel with current half block chunk
133           int nn;
134           const int offset_nn = (n+1) == CONST_LBP_PAR_BLOCKS ?
135             0 : CONST_LBP_PAR_BLOCKS/2;
136           for (nn = 0; nn < CONST_LBP_PAR_BLOCKS/2; nn++) {
137             //Get the score and alive arrays for the block
138             unsigned char* nn_alive_xx = alive_xx +
139               (offset_nn + nn)*MAX_DET_ALIVEPX;
140             unsigned char* nn_alive_yy = alive_yy +
141               (offset_nn + nn)*MAX_DET_ALIVEPX;
142             float* nn_alive_score = alive_score +
143               (offset_nn + nn)*MAX_DET_ALIVEPX;
144             int* nn_alive_count = alive_count + (offset_nn + nn);
145
146             lbp_face_detection_tail_block(frame,
147               n_x[offset_nn + nn], n_y[offset_nn + nn], width, height,
148               lbp_filters + 4*LBP_NUMSTAGES_FIRST_PASS,
149               lbp_alphas + LBP_NUMALPHAS*LBP_NUMSTAGES_FIRST_PASS,
150               threshold, nn_alive_xx, nn_alive_yy, nn_alive_score,
151               nn_alive_count, f_list_faces, f_list_size);
152           }
153           #pragma omp taskwait
154         }
155       }
156       n_x[n] = x;
157       n_y[n] = y;
158       n = (n+1) == CONST_LBP_PAR_BLOCKS ? 0 : (n+1);
159     }
160     }
161   }
162   }
163   if (n != 0 && n != (CONST_LBP_PAR_BLOCKS/2)) {
164     //Launch lbp_face_detection_tail_block for the half previous block
```

```
      //...
      #pragma omp taskwait
    }
    //Launch lbp_face_detection_tail_block for the last blocks
    //...
    #pragma omp taskwait

    if (f_list_size[0]) {
      lbp_facedetection_trigger(f_list_faces, f_list_size);
      #pragma omp taskwait
    }
  }
```

Listing A.4: Main tasks of Face Detection benchmark

Colophon

This thesis was typeset with LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.