LEGaTO

Low Energy Toolset for Heterogeneous Computing

D3.2 "FIRST RELEASE OF THE TASK-BASED RUNTIME"

Version 1.0

Document Information

Contract Number	780681
Project Website	https://legato-project.eu/
Contractual Deadline	31 July 2019
Dissemination Level	Public
Nature	Report
Author	Miquel Pericàs (CHALMERS)
Contributors	Konstantinos Parasyris (BSC), Mustafa Abduljabbar (CHALMERS), Xavier Martorell (BSC), Leonardo Bautista (BSC), Behzad Salami (BSC), Le Quoc Do (TUD), Gunnar Billung-Meyer (CHR), Babis Chalios (BSC)
Reviewers	Adrian Cristal (BSC), Madhavan Manivannan (CHALMERS)

The LEGaTO project has received funding from the European Union's Horizon 2020 research and innovation programme under the Grant Agreement No 780681.

Change Log

Version	Description of Change
0.1	2019-06-30. Initial draft
0.2	2019-07-09. Fault Tolerance
0.3	2019-07-16. New section layout
0.4	2019-07-17. OmpSs@FPGA and OmpSs@Linter
0.5	2019-07-18. XiTAO software topologies and Executive Summary
0.6	2019-07-19. Introduction and Conclusions
0.7	2019-07-19. XiTAO scheduler section and XiTAO release
0.8	2019-07-19. Secure Checkpointing
0.9	2019-07-23. OpenStack and RedFish
0.91	2019-07-23. OmpSs@Cluster
0.92	2019-07-24. Draft for internal review
1.0	2019-07-31. Address comments from internal review

Index

List of Figures

List of Tables

1. Executive Summary

This report details the status of the LEGaTO toolchain backend as of M20. The report is organized in three sections covering (1) the OpenStack middleware, (2) the energy-efficient runtime, implemented via its two main components Nanos and XiTAO, and (3) the fault tolerance schemes for GPU (via checkpointing), FPGA (for undervolting reliability), and CPU (ensuring secure checkpointing).

The first release of the task-based runtime supports static node composition via both OpenStack and the RedFish API. At the runtime layer, the current release now includes the first public release of the XiTAO runtime, featuring both performance and energy-aware scheduling and virtual topologies for locality-aware scheduling. The release also features support for executing OmpSs applications on cluster hardware (OmpSs@Cluster), and improvements to OmpSs@FPGA targeting novel FPGA hardware, support for scalar operands, and instrumentation for performance analysis. In addition, a tool has been developed to debug the correctness of OmpSs programs (OmpSs@Linter). Finally, this LEGaTO release also features support for advanced fault tolerance in the form of high performance GPU checkpointing, support for reliable and energy efficient FPGA undervolting, and support for secure checkpointing via the SCONE tool.

2. Introduction

Heterogeneous architectures composed of asymmetric cores, FPGAs and GPUs are instrumental to reach the levels of energy efficiency demanded by next generation IoT, Edge and HPC applications. The LEGaTO project is building a toolchain to map applications written in the OmpSs language onto two heterogeneous platforms provided by Christmann and Maxeler. This deliverable describes the first LEGaTO release of the runtime system that is being developed to support LEGaTO applications at runtime. The application development and compilation aspects are covered in the sibling deliverable D4.2.

The main goal of the runtime developed in LEGaTO is energy efficiency. In order to achieve the targeted improvement of 10× energy reduction, LEGaTO's runtime workpackage (WP3) is researching scheduling and locality-awareness techniques, the offloading of computations to FPGAs, and the undervolting of FPGAs.

The main task of the runtime is to make efficient use of the underlying hardware. This requires a good understanding of the available hardware and its configuration. The RECS hardware developed by Christmann can be statically configured and dynamically queried by the runtime via the RedFish API and an OpenStack layer. The Redfish API is described in Section 3.1. The OpenStack layer is described in Section 3.2. Work to target the platform from the runtime layer is currently underway and described in Section 3.3.

In modern platforms, performance and energy-efficiency are highly dependent on data movement. To this end, we are developing novel APIs to specify application-level task locality in a platform-independent way. The XiTAO runtime supports

a mechanism called software topologies which allows to map tasks on a virtual topology which is translated to hardware at runtime. The operation of this scheme is described in Section 4.1.1.

Scheduling techniques are being researched mainly in the context of the experimental XiTAO runtime. XiTAO decouples task parallelism from the amount of resources by specifying a resource container. Runtime-guided allocation of resource containers is a major target of our research, enabling the user to target performance-aware or energy-aware schedules. Our research on XiTAO is described in Section 4.1.2. We also describe the public release of XiTAO in Section 4.1.3.

FPGAs are recently becoming popular in HPC and in the datacenter as a way to accelerate applications while achieving high energy efficiency. In LEGaTO we are researching how to support FPGAs at runtime via the OmpSs@FPGA infrastructure which enables seamless offloading of FPGA bitstreams to FPGA accelerators, all integrated within the OmpSs compilation flow and Nanos runtime. Our research on OmpSs@FPGA is described in Section 4.2.1.

Scalability is another major goal of LEGaTO in order to support larger applications and systems. To achieve improved scalability we are researching how to execute OmpSs applications on multiple nodes with distributed memory. This approach, called OmpSs@Cluster has currently been released as part of OmpSs-2. The technologies required to execute OmpSs applications on large-scale clusters are described in Section 4.2.2.

One challenge associated with scalability is reliability. Executing an application on a large collection of nodes decreases its Mean Time Before Failure (MTBF). Checkpointing is a common technique to increase reliability by storing application snapshots to long term storage (e.g. disk). Checkpointing has been extensively researched in the context of CPUs. The LEGaTO project aims to extend this support to heterogeneous architectures including FPGA and GPU. Section 5.1 details our current research on automatically checkpointing applications running on GPUs using the FTI checkpointing library.

Further energy-efficiency with FPGAs can be achieved by using undervolting. This technique reduces the voltage of FPGA components to achieve a more energy-efficient operation mode. However, too aggressive undervolting can lead to errors. How much to undervolt and how to correct errors are two goals of our research on FPGA technology. Current results are described in Section 5.2.

Ensuring integrity and privacy is also an important goal of checkpointing. Currently LEGaTO is exploring how to generate secure checkpoints via the SCONE toolchain. This research is described in Section 5.3.

Finally, we are also developing runtime components to support the programming of OmpSs applications. In particular, we are currently developing a tool called OmpSs@Linter whose goal is to detect potential bugs in the specification of OmpSs task dependencies and missing synchronization between OmpSs parent and child tasks. These developments are described in Section 6.1.

3. Middleware and backend drivers

The RECS|Box hardware, as described in the SD1 deliverable [29], is a modular microserver system, which features the simultaneous use of different hardware architectures. All microservers and PCIe extensions are embedded into a high-speed-low-latency communication infrastructure, which allows very performant and efficient communication between all computation resources. The RECS|Box hardware also offers the possibility to build subsets of these resources and define their physical communication infrastructure during runtime to match applications and algorithms computation requirements perfectly. This mechanism of node composition is described in detail in D2.2 [3].

A suitable middleware layer is required to abstract out the complexity of these various hardware management possibilities. This will also improve the user-level experience. The software stack above the hardware infrastructure consists of multiple components, which can be seen in figure 3.1.

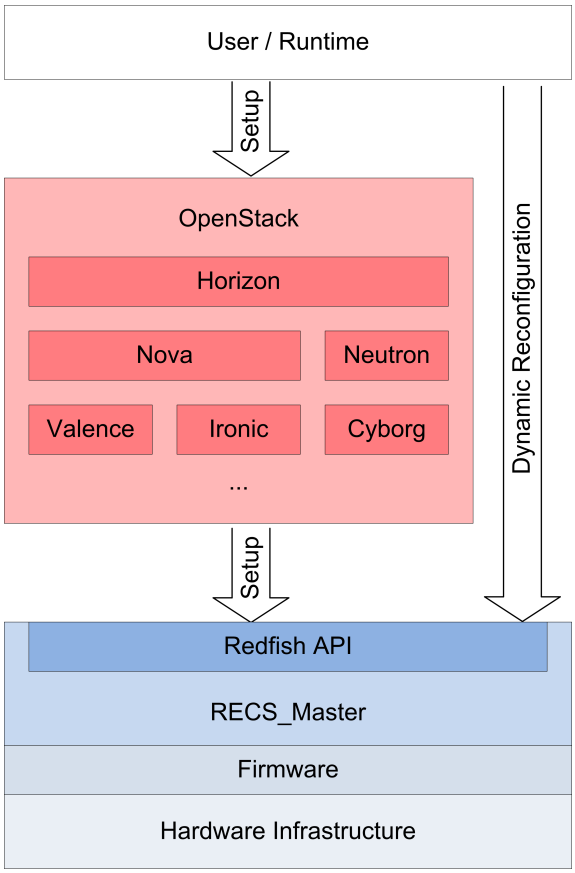


Figure 3.1. LEGaTO middleware stack for resource management and node composition

An embedded firmware is running on management CPUs within the hardware, managing, controlling and monitoring it on a low-level. The RECS_Master is the central management software within the RECS|Box, interacting closely with the firmware and providing the management and monitoring capabilities to the user through different interfaces. For regular manual administrative usage, a WebGUI is provided that allows access to all user-changeable settings and all monitoring and control functionality. In addition to that, the RECS_Master offers a Nagios Remote Plugin Executor (NRPE) interface for integration with monitoring and alerting software as well as an IPMI interface. Furthermore, there is a self-defined RESTful API, which allows retrieval of all measured sensor values and

basic control of the nodes. Finally, a Redfish API is available for managing certain special functions like node composition.

While the firmware is clearly part of WP2, the development of the RECS_Master belongs to both workpackages, as it is on the one hand part of the internal management of the underlying hardware infrastructure but on the other hand, it provides middleware functionality to the user and upper software layers through its APIs. In LEGaTO, the Redfish API is extended to allow dynamic node composition and its development status is reported in section 3.1. As already noted above, the mechanism of node composition is described in D2.2 [3].

The other main block of the LEGaTO middleware is OpenStack. This is an open-source software platform for managing cloud computing with the idea of providing infrastructure as a service. In LEGaTO, OpenStack is used to manage hardware resources and we extended it to support static node composition. The development status is reported in section 3.2. Despite the original plan, to extend OpenStack even further to also handle dynamic node composition during runtime, we decided to shift this functionality to the RECS_Master and pause the development of OpenStack in the current status. With this, we will support full-featured (re-)configuration of the underlying hardware independently of OpenStack and thus not forcing every customer to use and maintain an OpenStack infrastructure.

3.1. Redfish API

The management and composition of resources is one main task of the LEGaTO middleware layer. Therefore, it needs a comprehensive API that is capable of both providing detailed information about the hardware system and allowing operation calls for managing the available resources. This has to be provided by the RECS_Master management software of the RECS|Box, which has all required information at hand and the ability to control every part of the hardware system at the lowest level.

As already noted in the SD1 [29], we use the Redfish API of the RECS_Master for this task. The Redfish Scalable Platforms Management API from the Distributed Management Task Force, Inc (DMTF) [7] is designed to perform out-of-band management of multiple systems at once. It describes a RESTful interface on top of a data model [8], which is capable of expressing the relationships between components in modern systems. The payloads of HTTP requests to and responses from an implementation of a Redfish API are expressed in JSON [10] following OData [21] JSON conventions and can therefore easily be interpreted by clients. The standard is designed to be extensible and in the following, we describe the extended Redfish API of the RECS_Master, which matches the characteristics of the RECS|Box hardware and allows dynamic node composition. The comprehensive documentation of the Redfish API is accessible online [4].

3.1.1. Data Model

Figure 3.2 shows the adapted data model of the Redfish API, which defines multiple types of resources, that are related to each other. In addition to those relations, these resources contain information about their properties and capabilities.

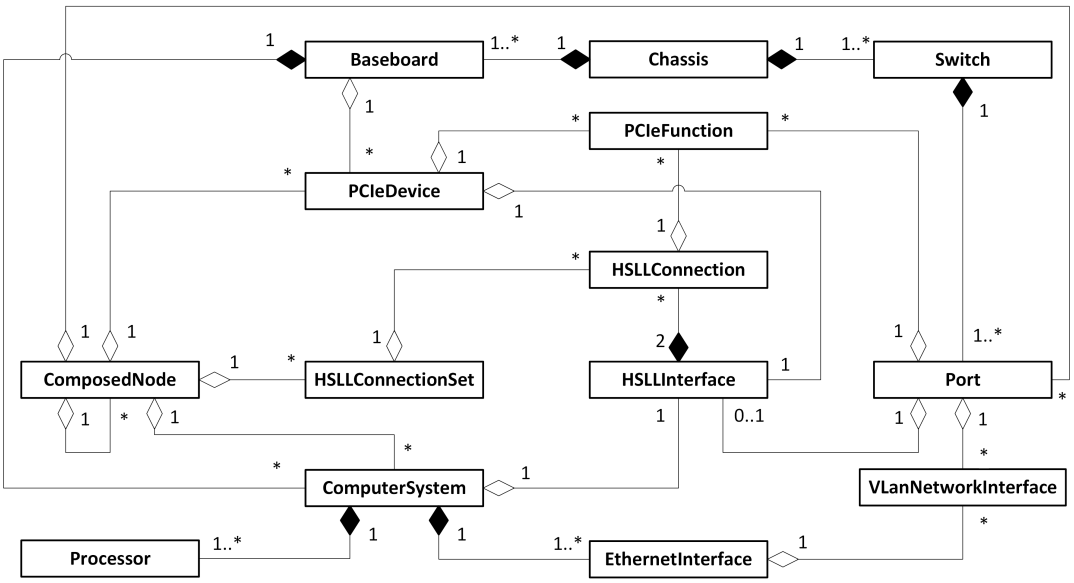


Figure 3.2. Redfish data model of the RECS_Master

A *Chassis* represents a physical RECS|Box enclosure. It holds references to the *Baseboards* it includes physically, which can host multiple *ComputerSystems*. Those are physical microservers with one or more *Processors* on it, which can be of the type CPU, FPGA or GPU and are providing additional information like their instruction set and number of cores. Each microserver has one or more *EthernetInterfaces* and can have an *HSLLInterface*. The latter represents the physical endpoint of the microserver where it connects to the high-speed-low-latency communication infrastructure. Those interfaces play an important role in the node composition process, described in D2.2 [3]. Like a microserver, a *PCIeDevice* also references to the *Baseboard*, it is attached to and has an *HSLLInterface*. In addition to that, such a PCIe extension card can provide a set of *PCIeFunctions*, which can be utilized by directly connected *ComputerSystems*. There are two types of *Switches* defined in this model. An Ethernet switch is a logical unit representing the physical 10 Gb/s Ethernet switch fabric within a *Chassis*. It provides *Ports*, the *ComputerSystems* can connect to with their *EthernetInterfaces*, optionally using VLANs, which translate to *VLanNetworkInterfaces* in this model. The other *Switch* type represents a PCIe switch fabric. Similarly to its Ethernet counterpart, it provides *Ports*. Analogue to the *ComputerSystems* and *PCIeDevices*, each PCIe *Port* has an *HSLLInterface* as an endpoint to connect to the high-speed-low-latency communication infrastructure. Additionally, it can have a set of *PCIeFunctions*, which can be utilized by computing resources connected to that *Port*. This mechanism is also a main part of node composition.

All the aforementioned resources represent a physical component of the RECS|Box microserver system. With node composition, it is possible to group computing resources together on a logical level and interconnect them by using the high-speed-low-latency communication infrastructure. A *ComposedNode*, which results from a node composition process, can comprise multiple and heterogeneous resources. Those can be *ComputerSystems*, *PCIeDevices*, *Ports* (only PCIe) and other already defined *ComposedNodes*. The way these resources are interconnected, is determined by an *HSLLConnectionSet*, which contains a set of *HSLLConnections*. Such a connection defines two *HSLLInterfaces* of resources

contained in the *ComposedNode* as endpoints and the number of physical lanes they are connected to each other. In addition to that, if one of the endpoints of an *HSLLConnection* is a *Port*, the connection can also specify a set of *PCIeFunctions* this *Port* will offer to the resource at the other endpoint of the connection. With an *HSLLConnectionSet*, it is possible to build a communication topology among the resources in a *ComposedNode* and assign virtual PCIe functions, which can be used by resources connected to the respective *Ports*. It is also possible to define more than one *HSLLConnectionSet*. Only one set can be active at a time, but it can be changed at any time within the lifetime of a *ComposedNode*. This feature enables the dynamic reconfiguration of the high-speed-low-latency communication infrastructure and the assignment virtual PCIe functions of at runtime.

3.1.2. Node Composition Process

The underlying mechanics of node composition are explained in detail in D2.2 [3]. For the sake of better comprehensibility, it is recommended to read that before continuing with this section. Here, we focus on the process of utilizing the Redfish API. All steps of this process are explained by means of a small, but comprehensive artificial example. The composed node in this example will consist of a microserver with an x86 processor, two FPGAs, an SSD storage in form of a PCIe extension card and ports of a PCIe switch.

The process of node composition through the Redfish API is loosely based on the process described by Intel's Pod Manager API specification [15] and is done in multiple steps. Figure 3.3 depicts this process as a state diagram.

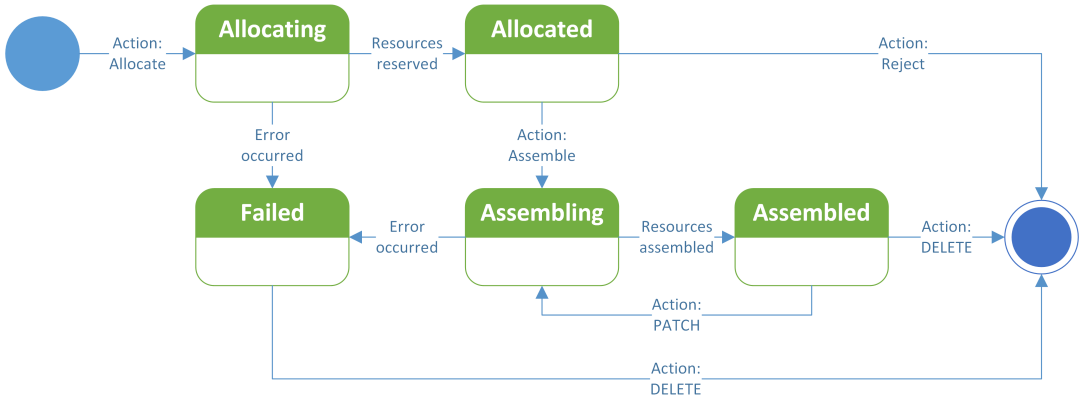


Figure 3.3. Node composition process with the Redfish API

As the first step, the user provides a set of requirements describing the node he wants to compose and orders the RECS_Master to allocate corresponding resources. These requirements can either be concrete hardware entities such as microservers or value sets specifying the desired properties of the composed node such as processor architecture, cores or communication capabilities. For this step, figure 3.4 shows an example HTTP POST request to the corresponding action URL of the Redfish API.

In the example, requirements for three microservers are defined in the *Systems* array of the JSON encoded HTTP POST body. The first has to have an x86 CPU with a minimum of four cores while the other two should be FPGAs without further

```
POST http://recs.box/redfish/v1/ComposedNodes/Actions/
    ComposedNodeCollection.Allocate
Content-Type: application/json
{
  "Name": "ExampleNode",
  "Description": "Node showing composition features",
  "Systems": [{
    "Processors": [{
      "ProcessorType": "CPU",
      "ProcessorArchitecture": "x86",
      "TotalCores": 4
    }]},{
    "Processors": [{
      "ProcessorType": "FPGA"
    }]},{
    "Processors": [{
      "ProcessorType": "FPGA"
    }]}
  }],
  "Devices": [{
    "PCIeFunctions": [{
      "FunctionType": "Virtual",
      "DeviceClass": "MassStorageController",
      "DeviceId": "0xa820",
      "VendorId": "0x144d"
    }]
  }],
  "Ports": [{
    "SystemIndex": 1
  },{
    "SystemIndex": 2
  },{
    "DeviceIndex": 0
  }]
}
```

Figure 3.4. HTTP request to allocate a composed node

constraints. Furthermore, a PCIe device is specified in the *Devices* array. It is required to have at least one virtual PCIe function with the stated device class, vendor ID and device ID. Finally, three PCIe ports are requested for the composed node, which are determined by the index within the corresponding arrays in the POST body. Here, the ports of the two FPGAs and the PCIe device are demanded.

The RECS Master then calculates a best match based on these requirements and reserves the resources by creating a composed node entity out of them. If no error occurred, a new composed node is created and put from the transfer state *Allocating* in the state *Allocated*. The URL location of the composed node, proposed by the RECS_Master, is returned in the response header *location*. It can then be reviewed by the requesting user, who can either accept the offered composition by ordering the RECS_Master to assemble it or reject it and provide adjusted requirements. For rejection, an HTTP POST with an empty body to the appropriate action URL is necessary (see figure 3.5).

```
POST http://recs.box/redfish/v1/ComposedNodes/CN_0/Actions/ComposedNode
    .Reject
Content-Type: application/json
{}
```

Figure 3.5. HTTP request to reject a composed node

The second step is then the assembling of the composed node. Here, the user has to define how the allocated resources are connected to each other and how virtual PCIe functions will be assigned, if there are any. Figures 3.6, 3.7 and 3.8 show an example HTTP POST request to assemble the allocated composed node.

```
POST http://recs.box/redfish/v1/ComposedNodes/CN_0/Actions/ComposedNode
    .Assemble
Content-Type: application/json
{
  "ConnectionSets": [{
    "Description": "FPGAs share the SSD and work independently",
    "Name": "fpgas_ssd",
    "Connections": [{
      "EndpointA": {
        "@odata.id": "/redfish/v1/Systems/RCU_167739053240_BB_1_0/
           HSLLInterface"
      },
      "EndpointB": {
        "@odata.id": "/redfish/v1/Systems/RCU_167739053240_BB_1_1/
           HSLLInterface"
      },
      "Width": 2
    },{
      "EndpointA": {
        "@odata.id": "/redfish/v1/PCIeDevices/
           RCU_167739053240_BB_1_Dev_0/HSLLInterface"
      },
      "EndpointB": {
        "@odata.id": "/redfish/v1/Switches/RCU_167739053240_SWMGR_PCI/
           Ports/RCU_167739053240_SWMGR_PCI_P_1-8/HSLLInterface"
      },
      "Width": 8
    },
```

Figure 3.6. HTTP request to assemble a composed node (Connect x86 to first FPGA and PCIe SSD to switch port)

Two configurations are specified in the JSON array *ConnectionSets* of the POST body. Both are given an explanatory description and a name for later reference. Each connection set has an array of connections that are building the set. A connection is defined by two endpoints and a width, which specifies the number of lanes connecting them. The first connection set is named "fpgas_ssd" and comprises four connections. The x86 microserver is connected to the first FPGA with two lanes and the PCIe SSD is connected to its corresponding port on the PCIe switch with eight lanes. With this, the switch has access to the virtual functions of the PCIe device and is able to assign them to other ports. (see figure 3.6)

```
    {
      "EndpointA": {
        "@odata.id": "/redfish/v1/Systems/RCU_167739053240_BB_1_1/
           HSLLInterface"
      },
      "EndpointB": {
        "@odata.id": "/redfish/v1/Switches/RCU_167739053240_SWMGR_PCI/
           Ports/RCU_167739053240_SWMGR_PCI_P_1-1/HSLLInterface"
      },
      "Width": 4,
      "PCIeFunctions": [{
        "FunctionType": "Virtual",
        "DeviceClass": "MassStorageController",
        "DeviceId": "0xa820",
        "VendorId": "0x144d"
      }]
    },{
      "EndpointA": {
        "@odata.id": "/redfish/v1/Systems/RCU_167739053240_BB_1_2/
           HSLLInterface"
      },
      "EndpointB": {
        "@odata.id": "/redfish/v1/Switches/RCU_167739053240_SWMGR_PCI/
           Ports/RCU_167739053240_SWMGR_PCI_P_1-1/HSLLInterface"
      },
      "Width": 4,
      "PCIeFunctions": [{
        "FunctionType": "Virtual",
        "DeviceClass": "MassStorageController",
        "DeviceId": "0xa820",
        "VendorId": "0x144d"
      }]
    }]
  },
```

Figure 3.7. HTTP request to assemble a composed node (Connect FPGAs to PCIe switch ports)

Both FPGAs are connected to their respective ports on the PCIe switch with four lanes each. If a port is one of the endpoints of a connection, it can additionally specify a set of PCIe functions that should be assigned to the port by the switch. Of course, the switch has to have access to those functions. Otherwise, the assembling will fail. The connections of both FPGAs to their switch ports are specifing, that the latter ones shall have such PCIe functions. Those are the virtual PCIe storage functions that are provided by the PCIe SSD connected to the switch (see figure 3.7).

Another connection set with the name "fpga_fpga" and two connections is defined in the JSON body (see figure 3.8). Similarly to the first connection set, the first connection links the x86 microserver with two lanes to the first FPGA. The second connection links both FPGAs together with six lanes. The set with the name "fpgas_ssd" is selected as first connection set, which will be activated directly during the assembling process. When this step has successfully ended,

```
  {
    "Description": "FPGAs interconnected, SSD disconnected",
    "Name": "fpga_fpga",
    "Connections": [{
      "EndpointA": {
        "@odata.id": "/redfish/v1/Systems/RCU_167739053240_BB_1_0/
           HSLLInterface"
      },
      "EndpointB": {
        "@odata.id": "/redfish/v1/Systems/RCU_167739053240_BB_1_1/
           HSLLInterface"
      },
      "Width": 2
    },{
      "EndpointA": {
        "@odata.id": "/redfish/v1/Systems/RCU_167739053240_BB_1_1/
           HSLLInterface"
      },
      "EndpointB": {
        "@odata.id": "/redfish/v1/Systems/RCU_167739053240_BB_1_2/
           HSLLInterface"
      },
      "Width": 6
    }]
  }],
  "ActiveConnectionSet": "fpgas_ssd"
}
```

Figure 3.8. HTTP request to assemble a composed node (Connect x86 to first FPGA and both FPGAs with each other)

the user is returned the URL location of the assembled and ready to use composed node. The specified topology is now active, giving both FPGAs shared access to the PCIe SSD while the first FPGA is additionally connected to the x86 microserver. The composed node is in the state *Assembled*.

```
PATCH http://recs.box/redfish/v1/ComposedNodes/CN_0
Content-Type: application/json
{
  "ActiveConnectionSet": "fpga_fpga"
}
```

Figure 3.9. HTTP request to reconfigure a composed node

For dynamically switching to the second configuration during runtime, a HTTP PATCH request to the URL of the composed node is necessary. It changes the active connection set by setting *ActiveConnectionSet* to "fpga_fpga". The high-speed-low-latency communication infrastructure is then immediately reconfigured, discarding all configurations of the first connection set except the connection between the first FPGA and the x86 microserver and establishing a new connection between the two FPGAs with six lanes. The example of the HTTP PATCH request is shown in figure 3.9.

```
DELETE http://recs.box/redfish/v1/ComposedNodes/CN_0
```

Figure 3.10. HTTP request to delete a composed node

Finally, a composed node can be destroyed to release all reserved resources and reset all configured connections. This is done by sending an HTTP DELETE request to the URL of the composed node (see figure 3.10).

3.2. OpenStack

OpenStack has a modular architecture and clearly separates tasks to different components. Those used in LEGaTO were already briefly introduced in the SD1 deliverable [29]. Here we focus on the three main components that are responsible for node provisioning and composition.

3.2.1. Ironic

With the Ironic component, OpenStack is enabled to provision bare-metal x86 and ARM servers in contrast to the quite common handling of virtual machines with the standard component Nova. Using the hardware directly instead of applying an additional resource-consuming virtualization layer is a quite apparent decision for the RECS|Box, as this platform mainly features microservers with limited resources.

Ironic comes with a TFPT server to provide images to bare-metal nodes and for power management it interacts with the virtual IPMI interface of the RECS_Master, which was developed in the M2DC project. Virtual in this case means that there is no dedicated BMC or IPMB, but the RECS Master implements the necessary parts of the IPMI specification in software and then maps this to its internal data model that is also used for all other provided interfaces. As standard IPMI was meant to work with just one server, the "double bridging" feature was used to make selection of the node to be managed possible. There is an example in the Annex A explaining how to discover the system, reading sensors and turning on a node with IPMItool. Figure 3.11 shows a screenshot of a bare-metal node in the Ironic GUI of the OpenStack installation (Rocky release) associated with a microserver within the RECS|Box.

3.2.2. Cyborg

Cyborg is used for accelerator management and was extended in the M2DC project to handle FPGAs in the context of the RECS|Box. It interacts with the Redfish API of the RECS_Master to obtain the list of available FPGA nodes and their MAC addresses. Cyborg connects to the FPGA to read the device data and add it to its database. Bitstreams, that can be used to configure FPGAs are stored in the OpenStack Glance service to be accessed in the deployment process. Figure 3.12 shows an example HTTP request to Cyborgs REST API to actually configure an FPGA with a bitstream. In addition to the obligatory <TOKEN> for authentication, the <FPGA_ID> of the accelerator within the Cyborg database and the <IMAGE_ID> of the bitstream within Glance are used in this request.

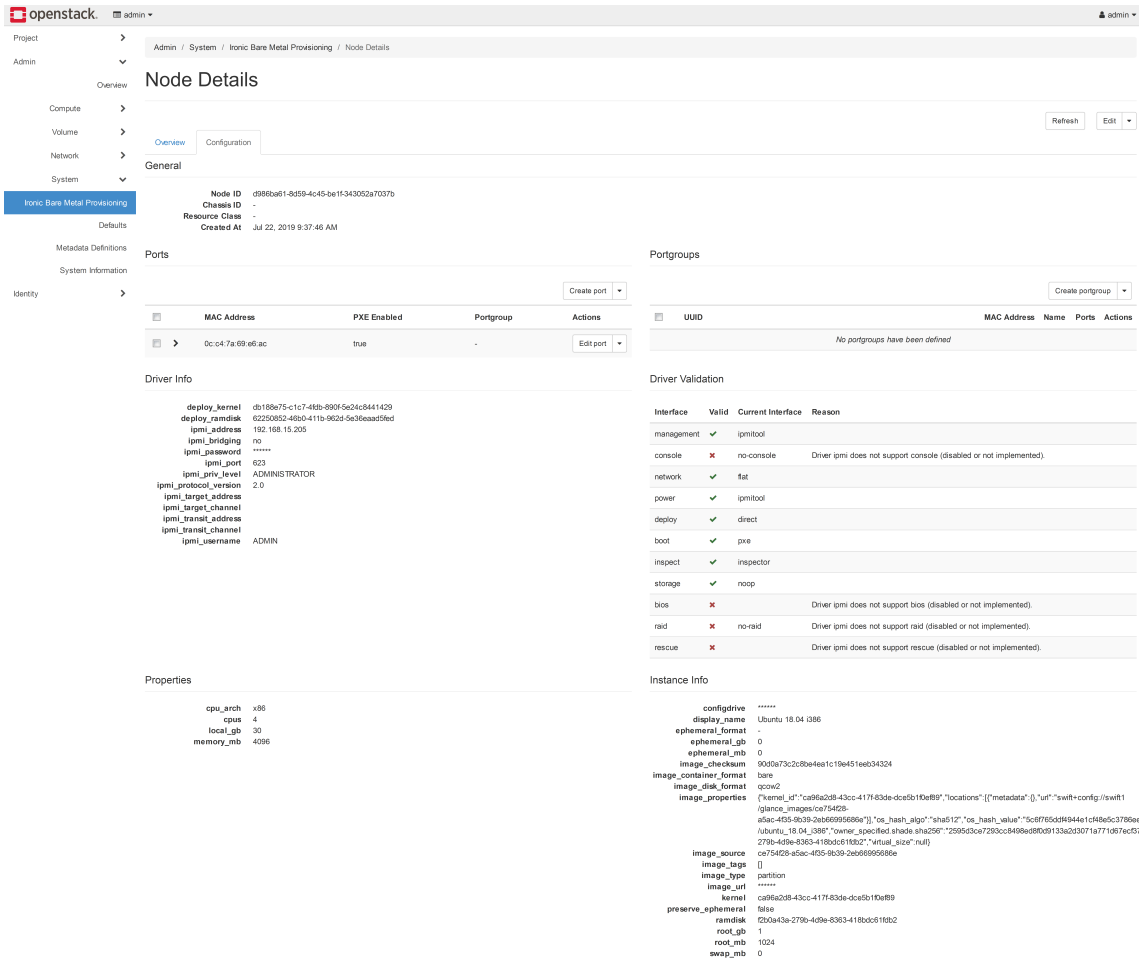


Figure 3.11. RECS|Box bare-metal node in OpenStack Ironic GUI

```
PATCH http://cyborg-api:6666/v1/accelerators/deployables/<FPGA_ID>/
    program
Content-Type: application/json
Accept: application/json
X-Auth-Token: <TOKEN>
[{
  "op": "replace",
  "path": "/program",
  "value": [{
    "image_uuid": "<IMAGE_ID>"
  }]
}]
```

Figure 3.12. HTTP request to program an FPGA with OpenStack Cyborg

3.2.3. Valence

Valence is the OpenStack component for lifecycle management of pooled bare-metal hardware infrastructure. It is intended to handle disaggregated computing, storage and networking resources and compose them together to meet various needs in data center and cloud environments. The main initiator of this component is Intel® with its Rack Scale Design [14] approach. Valence was extended, first in M2DC and then in LEGaTO, to also match the RECS|Box architec-

ture. For that, it utilizes the Redfish API of the RECS_Master, which is a perfect match, because the Redfish standard is Valence's default management protocol to communicate with hardware.

As noted in the introduction of this chapter, the Valence development is now at the state of supporting the static part of the node composition process, not allowing dynamic adaption of the high-speed-low-latency infrastructure at runtime. In favour of implementing all static and dynamic node composition features directly in the RECS_Master, the OpenStack development will be paused. Here, the process of static node composition with Valence is described. In main parts, it follows the process described in the Redfish API section 3.1, as Valence is tightly interacting with this interface.

The first step in Valence is the creation of a POD manager. In Rack Scale Design, a POD originally manages a collection of physical racks. This will be the entry point to the RECS|Box platform. See the example HTTP POST request in figure 3.13. In the request, the base URL of the Redfish API and authentication data are given. The driver "redfishv1" is selected to interact with the RECS_Master trough the Redfish API.

```
POST http://valence-api:8181/v1/pod_managers
Content-Type: application/json
Accept: application/json
{
  "url": "http://recs.box/redfish/v1",
  "name": "recs_box",
  "driver": "redfishv1",
  "authentication": [{
    "type": "basic",
    "auth_items": {
      "username": "admin",
      "password": "admin"
    }
  }]
}
```

Figure 3.13. HTTP request to create a POD manager with OpenStack Valence

The actual node composition starts, analogue to the Redfish API process, with the resource allocation, where requirements for all nodes have to be specified. Figure 3.14 shows a HTTP POST request for this first step.

In the example, a simple node consisting of two resources will be composed using the ID of the created POD manager <PODM_ID>. The requirements specify that one system has to have an x86 CPU and the other should be an FPGA. Additionally, both resources shall reside on a specific baseboard location, given by its ID within the Redfish API. The HTTP response contains the IDs of the proposed resources within the Redfish API and some additional information regarding the systems.

The next step in the node composition process is the assembling of the allocated resources by defining their high-speed-low-latency connections among each other. This is similar to the allocation step in the Redfish API process,

```
POST http://valence-api:8181/v1/nodes
Content-Type: application/json
Accept: application/json
{
  "allocate": "yes",
  "podm_id": "<PODM_ID>",
  "name": "Example Composed Node",
  "properties": {
    "Sytems": [{
      "Location": "RCU_2602448565353684_BB_1",
      "Processors": {
        "ProcessorType": "CPU",
        "ProcessorArchitecture": "x86"
      }
    },{
      "Location": "RCU_2602448565353684_BB_1",
      "Processors": {
        "ProcessorType": "FPGA"
      }
    }]
  }
}
```

Figure 3.14. HTTP request to allocate a composed node with OpenStack Valence

except that multiple connection sets and dynamic assignment of virtual PCIe functions are not allowed, as only static node composition is supported in this version of Valence. Figure 3.15 shows the example HTTP POST request for this assembling step.

```
POST http://valence-api:8181/v1/nodes
Content-Type: application/json
Accept: application/json
{
  "id": "CN_0",
  "assemble": "yes",
  "podm_id": "<PODM_ID>",
  "name": "Example Composed Node",
  "properties": {
    "connection": [{
      "components": "RCU_2602448565353684_BB_1_0,
         RCU_2602448565353684_BB_1_1",
      "width": "8"
    }]
  }
}
```

Figure 3.15. HTTP request to assemble a composed node with OpenStack Valence

Beside the obligatory POD manager ID, name and ID of the composed node are given in the assemble request. One connection is specified in this example, linking the CPU and the FPGA systems with eight lanes of the high-speed-low-latency communication infrastructure. Multiple connections can be defined in the array "connection", if more resources are available in the composed node.

In the HTTP response, the name and ID of the now assembled composed node are returned together with the URL to the node in the Redfish API and its ID in the Valence database. This concludes the node composition with OpenStack Valence.

3.3. Backend Drivers

In order to drive the platform from the runtime layer we are developing the necessary runtime system support for GPUs and FPGAs. For GPUs, CUDA and OpenCL libraries from vendors provide the necessary low level services for Nanos-6. On the Xilinx FPGAs, we provide the *xdma* and *xtasks* libraries that work on top of the vendor driver in Linux. We currently interoperate with the Xilinx and Alpha-Data drivers on the various Xilinx integrated and discrete FPGAs that we support (Zynq 7000, Zynq U+, and Virtex-7).

4. Energy-efficient task-based runtime

This chapter describes our efforts to develop runtime technologies targeting scalability and high energy efficiency.

4.1. XiTAO

This section describes the current status of the XiTAO runtime infrastructure. First we describe the concept of software topologies and its implementation in XiTAO. Next we describe XiTAO's online performance monitor and how it is used to guide scheduling decisions. Finally, we describe the first public release of the XiTAO runtime.

4.1.1. XiTAO software topologies

Software topologies is a mechanism implemented in XiTAO to achieve strict locality-aware scheduling of tasks in a portable manner. Since the execution model of XiTAO DAGs is *compile-once run-anywhere*, any information for locality aware scheduling needs to be generic and interpretable at runtime. At the task level, XiTAO implements a concept called "virtual topologies" which is converted at runtime into actual thread mappings to enforce locality aware scheduling. This mapping is *strict* in the sense that tasks that have a locality specification are no longer subject to load balancing. It is hence important to only use the locality feature when strictly necessary to avoid excessive communication. We will explore more relaxed schemes in the future.

XiTAO's virtual topologies consist of regular N-dimensional cartesian topologies. Figure 4.1 shows an example with virtual mappings of the `jacobi2D` and `copy2D` kernels as implemented in the Heat benchmark that is part of the sample benchmarks available in the public XiTAO git repository. In this example, each task in the DAG of task assembly objects (TAO-DAG) is given an address (called a *software topology address*) in a virtual topology consisting of a one-dimensional topology (a line between 0 and 1). Generically, the idea is that by measuring the virtual distance between two XiTAO tasks, the runtime obtains approximate information on the communication relationship between the two tasks. If two tasks have the same address, this is understood by the runtime as

meaning the highest amount of data reuse between the two tasks. As a consequence, the XiTAO runtime will attempt to schedule the two tasks on the same set of cores. This then optimistically results in data reuse via the caches of the cores. In the current state of XiTAO, we have implemented one-dimensional virtual topologies. As tested with the Heat diffusion simulation benchmark, this scheme can have a very positive impact on performance by avoiding unnecessary communication.

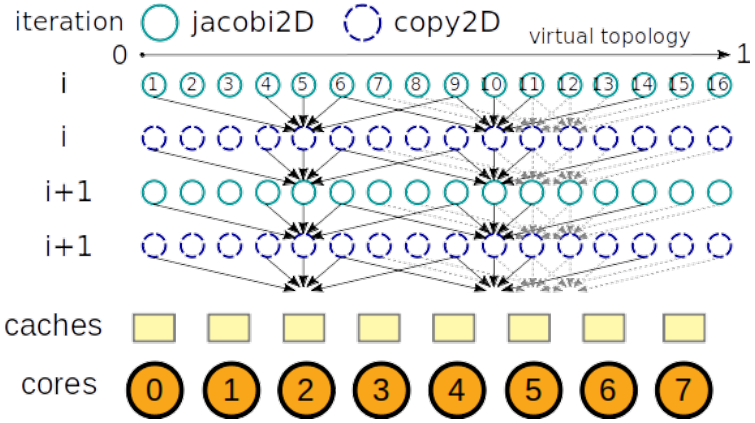


Figure 4.1. Virtual topology mapping of Jacobi2D and Copy2D kernels

XiTAO's virtual topologies can also be used to support NUMA-aware data placement. This is achieved by introducing an extra layer of tasks that takes care of data placement. Using virtual topologies, this layer of tasks can be scheduled to the same cores and NUMA nodes as the dependent tasks. This, combined with the default first touch allocation implemented in the Linux kernel[1], achieves locality-aware data placement. The overall idea is shown in Figure 4.2. The first touch policy specifies that the physical memory page is allocated on the node that first writes to the data. This is important since it means that data allocation (e.g. via malloc()) is not enough to ensure correct data placement, but in fact data placement happens when data is written to for the first time. Hence, as shown in Figure 4.2, initializing the data will take care of the proper data placement.

[1] https://queue.acm.org/detail.cfm?id=2513149

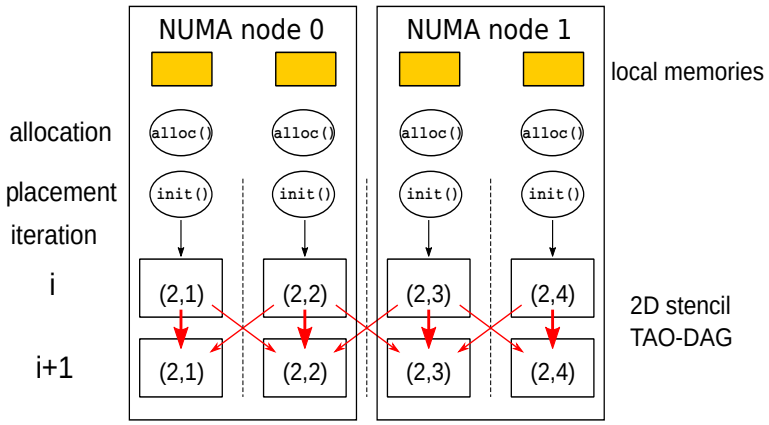


Figure 4.2. Adding an extra layer for NUMA-aware data placment

4.1.2. The XiTAO heterogeneous scheduler

With the emergence of heterogeneous hardware, it becomes important to adapt the runtime scheduling to the platform's heterogeneity. Systems may include both static and dynamic sources of heterogeneity. Static heterogeneity sources are those that are fixed at design time, for example, single-ISA cores with different levels of power-efficiency (e.g. big.LITTLE), or asymmetric ISA systems consisting of processor cores and accelerators (e.g. CPU/GPU/FPGA platforms). Dynamic sources of heterogeneity are those that arise from runtime conditions. For example, usage of Dynamic Voltage-Frequency Scaling (DVFS) [17] to tune the performance and efficiency of cores is an example of dynamic heterogeneity. Another example of is the occurrence of external applications competing for the same set of resources.

Classical schedulers that target single-threaded tasks usually perform a greedy schedule in an attempt to maximize resource usage and achieve good performance. A downside of this approach is that it leads to oversubscription of resources as all threads/cores attempt to bring their working sets into the shared cache by using the shared access to main memory. One way to address this limitation is to convert the classical 1-task to 1-core scheduling problem into a hierarchical problem in which a global level schedules M-task-groups onto N-core-places, and a local level schedules the M-tasks onto the assigned N-cores. This scheme, known as Elastic Places, targets interference-free scheduling and allows to retain the greedy property in the global scheduler [22]. The XiTAO library[2] is an embodiment of this concept. In XiTAO, parallel tasks are scheduled into resource partitions called elastic places. The method has been shown to perform efficiently on homogeneous manycore and NUMA systems. To achieve higher energy efficiency, however, it is important to make XiTAO aware of heterogeneous platforms.

In LEGaTO, we have explored and proposed schemes to automatically determine resource partitions (i.e. N-core places) at runtime. Furthermore, we have researched how this knowledge can be used to leverage modern single-ISA platforms with both static and dynamic sources of heterogeneity. To this end, we

[2] https://sites.google.com/site/mpericas/xitao

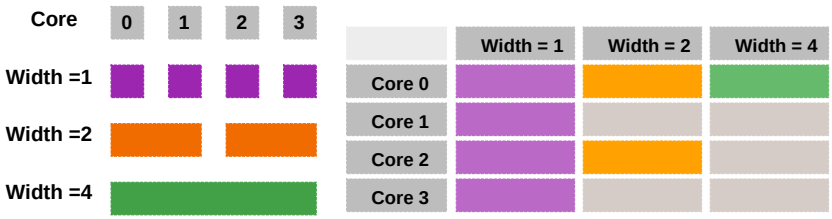


Figure 4.3. Example of a PTT with four cores. Valid resource widths are 1, 2, or 4.

have developed a scheduler inspired by Criticality-Aware Task Scheduling (CATS) [5] and extended it with a performance trace table (PTT) that monitors the system's performance characteristics at runtime. Despite its simplicity, the PTT provides enough information to implement both heterogeneity-aware and interference-free schedules at runtime at minimum cost.

4.1.2.1. Performance Trace Table

To be able to intelligently distribute tasks to the corresponding core type and to dynamically affect the scheduling decisions based on the available resources, we have introduced a runtime task performance tracer and a table (PTT) to record and predict future task execution times. The table provides an online model of the execution time for each valid combination of *leader core* and *resource width*, *(core id, resource width)*. The leader is the core with the lowest index in the assigned place (i.e. resource partition). The left part of Figure 4.3 shows all the scenarios of resource width when the total number of cores seen by the runtime is four. In this case, the resource width can be 1, 2 and 4.

The PTT is implemented in the XiTAO runtime. It is organized as shown in the right of Figure 4.3. The size of the table is $core\_number \times resource\_width\_number$. The fields of the table are initialized to 0, which models a zero execution time. This ensures that all configuration pairs will eventually be visited and trained at runtime. Due to the decentralized implementation of the scheduler, the table is organized to fit into cache lines such that each core only accesses one cache line indexed with core number, hence avoiding false sharing. For each entry, the execution time of the pair is temporarily stored. Then each entry is updated with a weighted time of 1:4 (i.e. 80% old, 20% new). This combination has been found experimentally to perform well. The performance trace table is updated always by the leader core of a task. This simplifies the implementation and reduces cache migrations. This also means that every core can have a model value of the task with $width = 1$ but only every fourth core will have a model value of the task with $width = 4$. As shown in Figure 4.3, in the case of width=1 (purple field), each core is the leader for its own partition. For the case width=2 (orange), the leading core 0 (2) handles the resource partition containing cores 0 (2) and 1 (3).

This implementation of tracing execution history requires very little information: only the number of cores and their distribution into core-clusters with shared caches is required. The cores simply update the corresponding index, independent of its resource type, thus creating a performance model. In other words, no matter what core-types a platform has (big, little, etc.), the performance of these cores will be reflected by PTT values. This is beneficial not only for portability

and potentially functional-heterogeneity, but also for temporally added heterogeneity such as DVFS caused by heat variations, or interference caused by other tasks and/or uncontrollable system activities such as background processes or interrupts.

4.1.2.2. Performance-based Scheduler

Based on the implementation of the performance trace table, we have developed a heterogeneous scheduler with the goal of optimizing performance. To achieve this, we followed the basic strategy of CATS [5] and extended it with our heterogeneity-agnostic methodology. This scheduler is named *performance-based scheduler*. The main feature of the performance-based scheduler is its ability to find the optimal cores and resource width by globally searching the PTT.

If a task lies on the critical path we globally search the performance trace table to find the optimal pair of core and resource width for such task. Global search means that all the entries of the PTT of the particular TAO type are checked to find the value that globally minimizes the product: $exec\_time \times resource\_width$. The goal of this operation is to find the pair of core and resource width that minimizes the system's occupation of resources, understood as the product of resources and execution time. For tasks that do not lie on the critical path, only the best width is selected according to the PTT, but the leader core is not modified. We also modified work stealing so that only non-critical task can be stolen. Critical tasks are ignored during work stealing.

4.1.2.3. Experimental Setup

We evaluated the PTT scheduler on two platforms. From the static heterogeneous family, we use a NVIDIA Jetson TX2 development board, featuring a dual-core NVIDIA Denver 2 64-bit CPU and a quad-core ARM A57 Complex (each with 2 MB L2 cache). Both the Denver 2 and the A57 cores implement the ARMv8 64-bit instruction set and are cache coherent. On the homogeneous side, an Intel 2650v3 (code-named "Haswell") based platform is used to evaluate the effect of interference while scheduling Random DAGs, and to evaluate the behavior of XiTAO when executing the VGG-16 Image Classification network [27].

Random Directed Acyclic Graph

Kernels

We generate random DAGs to evaluate the properties of the PTT scheduler. The random DAGs are based on a mix of different kernel types. When selecting the kernels, the priority is to achieve different characteristics in terms of memory-intensiveness (streaming), cache-intensiveness (i.e. data reuse) and compute-intensiveness. The following three kernels are selected for this purpose.

Matrix Multiplication. A *matrix multiplication* kernel is created for the *compute-intensive* property. We implement a matrix multiplication that achieves parallelism by ensuring that the writing of output data is done to separate cache lines for each thread while still sharing the input data.

Sort. For the *data reuse* property, a *quick sort* and *merge sort* kernel combination is selected. This kernel first splits the input array into chunks and performs

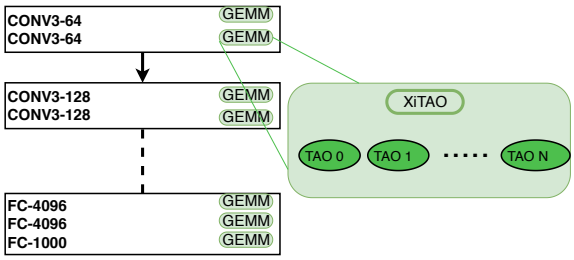


Figure 4.4. Architecture of VGG-16: CONVX-Y represents X-D filter and Y Channels of convolutional layer respectively

in-place sorting with quick sort before carrying out two levels of merge sort, effectively reusing the data within the kernel. This kernel has a maximum parallelism of four.

Copy. Finally, a *copy* kernel handling large inputs is implemented for the *streaming* property. This kernel reads and writes large portions of data to memory, effectively creating a streaming behavior where the kernel accesses main memory continuously. Each core copies a subset of the data.

For each kernel, we select the appropriate working set size corresponding to the desired behavior. For the matrix multiplication kernel, we choose a 64 $\times$ 64 matrix. For the sort kernel, we choose a 262KB input array, taking up a total space of 524KB due to double buffering, effectively fitting the L2 caches of the TX2 platform. Finally, the copy kernel uses a 16.8MB array, taking up a total space of 33.6MB, which is much larger than the space of the L2 cache.

DAG construction

To properly evaluate the performance of our scheduler, randomized DAGs composed of random selections of these three kernels are implemented. By tuning the parameters, it is possible to achieve different degrees of average parallelism and thus generate different scheduling scenarios.

To generate a suitable randomized DAG, a set of configuration parameters are used, similar to the generation of DAGs by Topcuoglu et al. [28]. The first parameter is the number of tasks of each kernel. This is used to choose which kernel should be most prominent in the DAG. The second parameter is the average width of the DAG. This is used to obtain the desired level of parallelism. The last configuration parameter, the edge rate parameter, determines the average amount of connected edges a task has, which also affects the parallelism of the DAG. A seed value is used to manipulate the randomization to recreate a different DAG several times for comparison.

Image Classification

We also ported the VGG-16 [27] image classification model from the Darknet framework [23]. This application uses a 16-layer deep convolutional neural networks (CNNs) to classify an image using a pre-trained model. Each convolutional (CONV) and fully-connected (FC) layer implements GEneral Matrix Multiply (GEMM) that takes most of the computation time. Figure 4.4 shows the XiTAO implementation of VGG-16. In VGG-16, the input size varies as the network progresses. For example, the convolutional layer iterates over a minimum 64 chan-

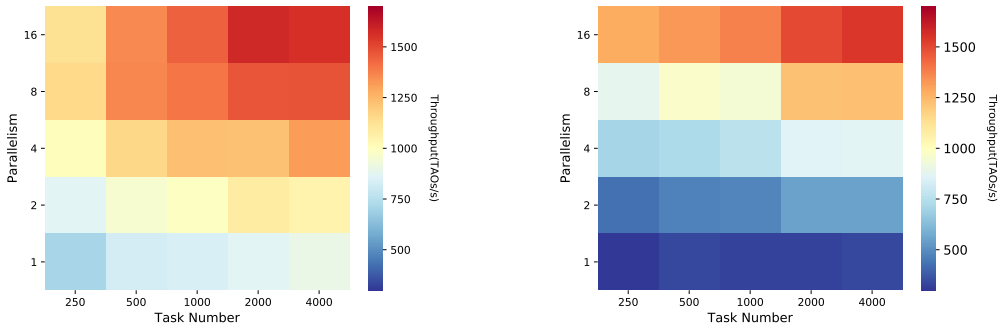
nels to a maximum of 512 channels. Therefore, in the XiTAO implementation we partition the work among TAOs. The number of TAOs in each layer depends on the number of *channels* and *block_length*. The parameter *block_length* refers to the number of channels assigned to each TAO, which is tuned at runtime. Each TAO performs parallel GEMM with the number of threads equal to the *width* of TAO. Note that the *width* is dynamically determined by the XiTAO scheduler. Since there are no loop carried dependencies inside the layer we benefit from two levels of parallelism in the XiTAO implementation. However, each layer is dependent on the previous layer, we therefore synchronize all TAOs at the end of each layer.

4.1.2.4. Performance Evaluation

We now present our evaluation of the XiTAO performance scheduler on the random DAGs and the Imagenet VGG-16 application.

Comparison with Homogeneous Scheduler

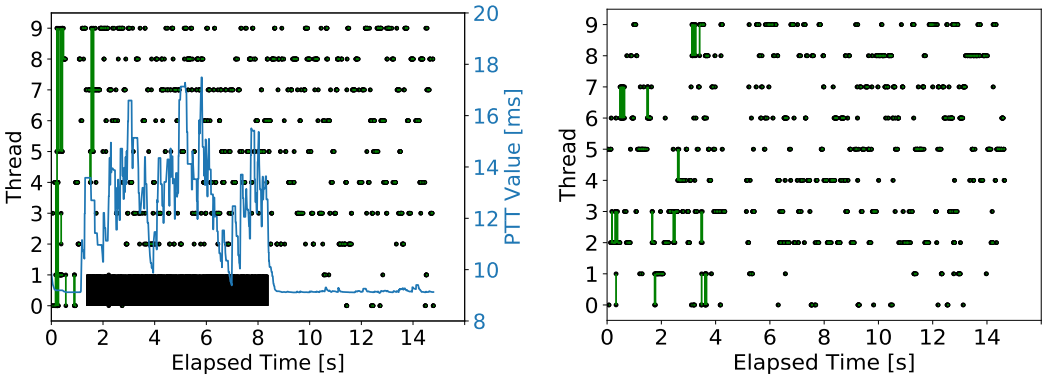
Figure 4.5 shows a throughput heatmap for two schedulers: a homogeneous scheduler (random work stealing with no task moldability) and the performance-based scheduler using the PTT. Each benchmark executes between 250-4000 tasks (X-Axis) on random DAGs with a parallelism between 1-16 (Y-Axis). The underlying random DAG is a combination of the aforementioned kernels with equal proportions. In the most challenging case with low task count and parallelism (tasks=250, par=1), we observe that the temperature of the performance-based scheduler (depicted by Figure 4.5a) is at least twice higher (meaning twice the throughput). The additional ingredient of scheduling critical tasks on the high performing cores (mainly *Denver* cores in this case) and the ability to dynamically tune the resource width renders this scheduler superior even with no external task-DAG parallelism.



(a) Performance-based Scheduler *(b) Homogeneous Scheduler*

Figure 4.5. The performance impact over parallelism and number of TAOs and the performance comparison between performance-based scheduler and homogeneous scheduler.

The throughput is higher across the table except for a few cases of very high parallelism that pose almost no challenge on scheduling decisions. Another interesting yet expected observation is that the number of tasks plays a negligible role on the performance of the homogeneous scheduler, whereas the throughput of the counterpart is a factor of both axes. A twofold increase in the number



(a) Dynamic migration of processes in response to PTT spikes during interference. *(b) The scheduler's behavior when there is no interference*

Figure 4.6. The effect of interference on PTT scheduling of critical tasks.

of tasks provides twice the amount of PTT training data. This directly reflects on the performance by improving the quality of the dynamic, PTT-based choices. In addition, a higher degree of parallelism (on the Y-Axis) permits a better utilization of the resources.

Process Interference

One of the remarkable advantages of using PTT is maximizing performance via minimizing the side effects of interference. This feature is especially important since it is often the case that user or kernel level resources are shared. Figure 4.6a depicts the response of the XiTAO performance-based scheduler to a running background parallel process, in this case a chain of MatMul DAGs, alongside a highly parallel random DAG. The black dots represent the time-stamp at which the threads start executing critical-path TAOs. A vertical green line shows the resource partition used to execute the TAO, the absence of a green line indicates that a single core is used for this critical task. While bootstrapping the PTT, a few width choices are attempted. At the point of interference (i.e. in Figure 4.6a), we show the PTT value at (width=1,core=1). Other relevant values are dropped for brevity. Due to the jitters in PTT values, the scheduler automatically selects cores from (2-9) for executing the critical tasks. Cores (0-1) are still selected under typical circumstances according to Figure 4.6b. Shortly after the interference event, the scheduler recovers to normal operation yielding a marginal wall time difference across the two experiments. Note that non-critical task continue to be executed on cores with interference, as long as these "slower" cores succeed in stealing tasks. This is important so that the PTT is continuously updated to reflect the status of the system.

ImageNET Classification

Figure 4.7 depicts a strong scalability study of the performance of the XiTAO version of the VGG-16 code for predicting a predefined image class by multiple convolutions of a crop layer (1024 x 1024) converted to matrix (512 x 512 x 3). The study is to assess the scheduling performance of the conventional fork-join application class with minimal effort. It is carried out on a dual-socket Intel Haswell platform. XiTAO, in combination with the PTT, exhibits 69% parallel efficiency compared to linear scaling. In this experiment there is no criticality

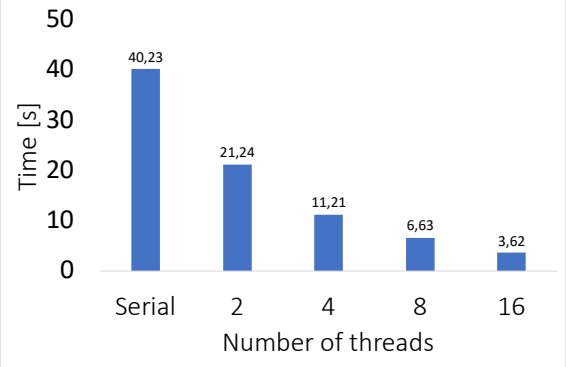


Figure 4.7. Performance of CPU GEMM on XiTAO VGG-16 with variable number of threads

notion, i.e., all tasks are marked non-critical. Figure 4.8 shows the number of TAOs scheduled with corresponding widths. During execution, the PTT chooses the best width to schedule a TAO. For example, in the case of running VGG-16 with 8 threads, *67%* of TAOs are scheduled with *width = 1* and *30%* TAOs are scheduled with *width = 8*, indicating that these widths led to the best speed-up during execution.

4.1.3. The XiTAO Public Release

The XiTAO runtime's code has been documented and released as open source (available on https://github.com/mpericas/xitao) under the BSD 3-Clause License. In order to facilitate future collaboration, the code follows the `Doxygen` formatting standard. Using a simple parallel dot-product example, the documentation describes the necessary classes and functions used to generate the TAO-DAG, trigger and finalize the runtime. The following benchmarks are available as experimental use cases for XiTAO:

- HEAT: an iterative blocked Jacobi heat diffusion solver that validates the XiTAO's mapping of software to hardware topologies. The benchmark exposes the decomposed domain and the XiTAO runtime maps it according to the NUMA nodes of the platform. The domain is decomposed in two dimensions x,y, both internally and externally. The internal decomposition refers to the DAG inside the TAOs, while the external decomposition refers to the TAO-DAG [22].
- Random DAGs: a parametric graph generator that is designed to generate weighted directed acyclic graphs with various characteristics, adopted from [28]. The DAG nodes can be either matrix multiply nodes (compute-intensive), streaming nodes (memory intensive), or sort nodes (cache-intensive)
- Dot product: a parallel dot product example that provides a simple usage of the basic XiTAO features (see Listing 4.1). The implementation of `VecMulDyn` shown in Listing 4.2 represents an internal dynamic scheduler for the dot product TAO that overdecomposes the input to create small work units to be fetched dynamically. `VecMulSta` is an alternative static scheduler but is dropped for brevity.
- VGG-16: an external image classifier that is forked from the Darknet repository.

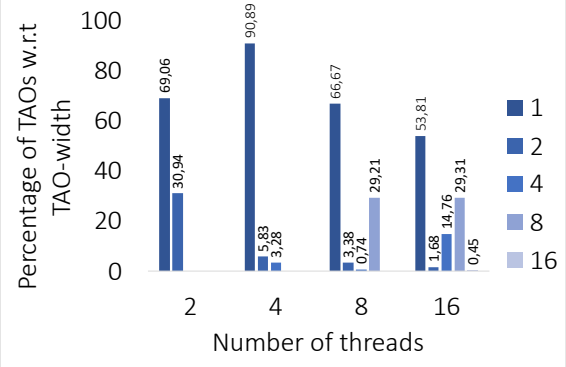


Figure 4.8. Percentage of TAOs scheduled with corresponding TAO width by PTT

```
// init XiTAO runtime
gotao_init();

// create numvm number of TAOs
int numvm = len / block;

// static or dynamic internal TAO scheduler
#ifdef STATIC
VecMulSta *vm[numvm];
#else
VecMulDyn *vm[numvm];
#endif
VecAdd *va = new VecAdd(C, &D, len, width);

// Create the TAODAG
for(int j = 0; j < numvm; j++){
#ifdef STATIC
  vm[j] = new VecMulSta(A+j*block, B+j*block, C+j*block, block, width);
#else
  vm[j] = new VecMulDyn(A+j*block, B+j*block, C+j*block, block, width);
#endif
  //Create an edge
  vm[j]->make_edge(va);
  //Push current root to assigned queue
  gotao_push(vm[j], j % gotao_nthreads);
}
//Start the TAODAG exeuction
gotao_start();

//Finalize and claim resources back
gotao_fini();
```

Listing 4.1. A simple parallel dot product example using the XiTAO API.

4.2. OmpSs

This section describes the status of the OmpSs runtime work that has been performed in the LEGaTO project. We start by describing the OmpSs@FPGA infrastructure to program hybrid CPU-FPGA systems, and proceed with a description of the OmpSs@Cluster runtime that targets scalable execution of OmpSs applications on cluster hardware.

4.2.1. OmpSs@FPGA

In this section, we describe the current implementation of OmpSs@FPGA, and we present the study and evaluation of the benchmarks matrix multiplication, cholesky decomposition and Nbody.

4.2.1.1. OmpSs@FPGA Ecosystem

The OmpSs [25, 9] programming model allows to express parallelism that will be executed in the available resources among the host SMP cores, or integrated/discrete GPUs and/or FPGAs. OmpSs is based on task parallelism, and very

```
/*! this TAO will take two vectors and multiply them.
This TAO implements internal dynamic scheduling.*/
class VecMulDyn : public AssemblyTask
{
public:
#if defined(CRIT_PERF_SCHED)
  //! A public static class variable.
  /*!
    It holds the performance trace table for the corresponding TAO.
  */
  static float time_table[][GOTAO_NTHREADS];
#endif
  //! VecMulDyn TAO constructor.
  /*!
    \param _A is the A vector
    \param _B is the B vector
    \param _C is the Result vector
    \param _len is the length of the vector
    \param width is the number of resources used by this TAO
    The constructor computes the number of elements per thread and overdecomposes the domain using PSLACK parameter
    In this simple example, we do not instatiate a dynamic scheduler (yet)
  */
  VecMulDyn(double *_A, double  *_B, double *_C, int _len,
      int width) : A(_A), B(_B), C(_C), len(_len), AssemblyTask(width)
  {
     if(len % (width)) std::cout <<  "Warning: blocklength is not a multiple of TAO width\n";
     blocksize = len / (width*PSLACK);
     if(!blocksize) std::cout << "Block Length needs to be bigger than " << (width*PSLACK) << std::endl;
     blocks = len / blocksize;
     next = 0;
  }

  //! Inherited pure virtual function that is called by the runtime to cleanup any resources (if any), held by a TAO.
  int cleanup(){
  }

  //! Inherited pure virtual function that is called by the runtime upon executing the TAO.
  /*!
    \param threadid logical thread id that executes the TAO
    This assembly can work totally asynchronously
  */
  int execute(int threadid)
  {
   //  int tid = threadid - leader;
    while(1){
      int blockid = next++;
      if(blockid > blocks) return 0;
      for(int i = blockid*blocksize; (i < len) && (i < (blockid+1)*blocksize); i++)
          C[i] = A[i] * B[i];
    }
  }
#if defined(CRIT_PERF_SCHED)
  //! Inherited pure virtual function that is called by the performance based scheduler to set an entry in PTT
  /*!
    \param threadid logical thread id that executes the TAO
    \param ticks the number of elapsed ticks
    \param index the index of the width type
    \sa time_table()
  */
  int set_timetable(int threadid, float ticks, int index)
  {
    time_table[index][threadid] = ticks;
  }

  //! Inherited pure virtual function that is called by the performance based scheduler to get an entry in PTT
  /*!
    \param threadid logical thread id that executes the TAO
    \param index the index of the width type
    \sa time_table()
  */
  float get_timetable(int threadid, int index)
  {
    float time=0;
    time = time_table[index][threadid];
    return time;
  }
#endif
  int blocks;    /*!< TAO implementation specific integer that holds the number of blocks per TAO */
  int blocksize; /*!< TAO implementation specific integer that holds the number of elements per block */
  int len;       /*!< TAO implementation specific integer that holds the vector length */
  double *A;     /*!< TAO implementation specific double array that holds the A vector */
  double *B;     /*!< TAO implementation specific double array that holds the B vector */
  double *C;     /*!< TAO implementation specific double array that holds the result vector */
  atomic<int> next; /*!< TAO implementation specific atomic variable to provide thread safe tracker of the number of
      processed blocks */
};
```

Listing 4.2. TAO's dot product class using an internal dynamic scheduler.

similar to OpenMP tasking. It is being used as a forerunner prototyping environment for future OpenMP features. On GPUs, both CUDA and OpenCL kernels are supported. For FPGAs, OmpSs uses the vendor IP generation tools (Xilinx Vivado and Vivado HLS [19, 30], or Altera Quartus [13]), to generate the hardware configuration from high-level code. OmpSs@FPGA can also leverage existing IP cores, provided they adhere to the same interface with our software platform.

OmpSs@FPGA is a significant upgrade of the OmpSs infrastructure (Mercurium source-to-source compiler and Nanos++ runtime) to incorporate FPGA support. Figure 4.9 shows an example of an OmpSs application. In particular, function *matrix_multiply* is defined as a task with input dependencies *a* and *b* and input/output dependency *c*. Each call to this function will be converted in a task that will be run when its dependencies are ready. This task has also been defined to be potentially executed in two target devices: any of the cores of the *smp* running the application and three instances of an accelerator that will be built to do this task in the FPGA. The accelerator has been tuned by the programmer to exploit the parallelism of the FPGA by using some additional directives (*#pragma HLS*) not related to OmpSs programming model. In the following sections, we will describe how the OmpSs compilation and runtime ecosystem helps programmability, heterogeneity, memory transfers and tracing support, and finally, mechanisms to develop blocking techniques from inside the FPGA.

4.2.1.2. Programming Productivity

Figure 4.10 shows the toolchain flow. In particular, it currently supports Xilinx FPGAs using the Vivado HLS and Vivado tools through our *autoVivado* tool.

At the compilation level, the OmpSs application is split in two parts according to the OmpSs directives. All functions annotated with the *target device(fpga)* directive are defined as tasks that will be transfered to the Vivado HLS tool for compilation to IP cores. Additionally, the Mercurium compiler generates a stub/wrapper function for each task, used to invoke the corresponding IP core from our Nanos++ runtime system, adapting the parameter passing. *autoVivado* tool invokes Vivado HLS to transform the wrapper functions and the FPGA-annotated functions into IP cores. Then, *autoVivado* connects them to the rest of the system using Vivado and generates the bitstream with the accelerators. Also, a configuration file (*xtasks.config*) with accelerator metadata is generated. This is necessary for the Nanos++ runtime in order to know the type and number of accelerators in the FPGA. This compilation process is automatically done by the compiler avoiding hand made code errors and speeding up all the process of hardware generation for the supported platforms (Zynq 7000 and Ultrascale+ families).

On the other hand, Nanos++ is the OmpSs runtime system. It takes care of executing tasks annotated by the programmer in the available resources. The high-level view of the execution environment is presented in Figure 4.11.

Nanos++ environment has a *thread team* created by default, the *dependence graph* used to organize tasks that still have pending data dependences to be resolved, and the *task pool* representing all the ready tasks. Running threads create tasks and insert them into the dependence graph. When data dependences have been fulfilled, the thread detecting this situation moves the tasks

```
1  #pragma omp target device(fpga,smp) copy_deps num_instances(3)
2  #pragma omp task in([BS]a,[BS]b) inout([BS]c)
3  void matrix_multiply(float a[BS][BS],
4       float b[BS][BS],float c[BS][BS]) {
5  #pragma HLS inline
6  #pragma HLS array_partition variable=a \
7          block factor=BS/2 dim=2
8  #pragma HLS array_partition variable=b \
9          block factor=BS/2 dim=1
10   for (int ia = 0; ia < BS; ++ia)
11     for (int ib = 0; ib < BS; ++ib) {
12 #pragma HLS PIPELINE II=1
13       float sum = 0;
14       for (int id = 0; id < BS; ++id)
15         sum += a[ia][id] * b[id][ib];
16       c[ia][ib] += sum;
17 } }
18 ...
19 for (i=0; i<NBI; i++)
20   for (j=0; j<NBJ; j++)
21     for (k=0; k<NBK; k++)
22       matrix_multiply(AA[i][k], BB[k][j], CC[i][j]);
23 #pragma omp taskwait
24 ...
25 }
```

Figure 4.9. First version of FPGA Matrix Multiply code

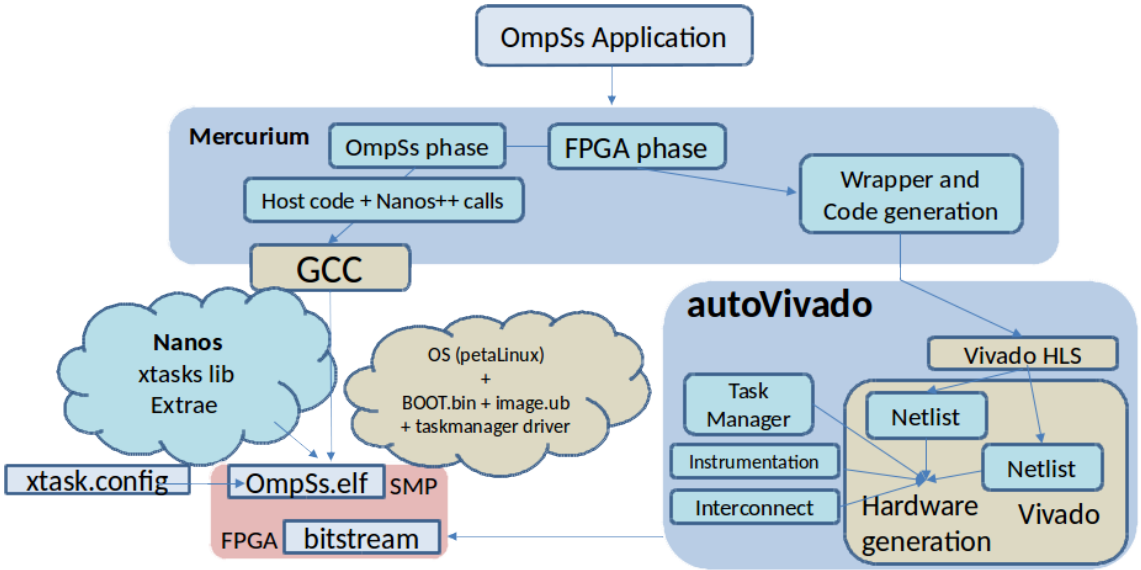


Figure 4.10. OmpSs compilation env. with FPGA support

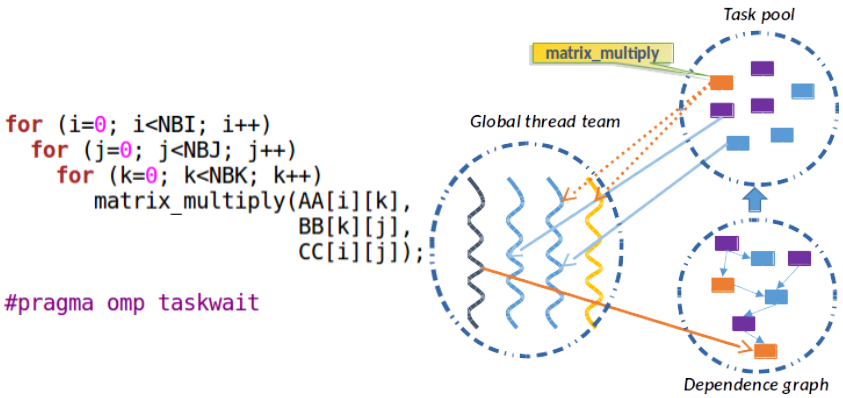


Figure 4.11. High-level representation of the Nanos++ environment

now free of dependences to the task pool. When a thread finishes the execution of a task, it becomes idle and it looks for work in the task pool. The Nanos++ runtime will also take care of the possible heterogeneity expressed by the programmer and the necessary memory transfers (copies).

On the FPGA side, an special IP, called Task Manager, is in charge of the management of the accelerator executions and finalizations. It will provide the accelerator with the information of a task, written by the Nanos++ runtime in shared memory. Then, once the accelerator finalizes, the Task Manager will be signaled by the output stream port of the accelerator to indicate the end of the task. Finally, the Task Manager will notify the Nanos++ runtime of the finalization of the task.

4.2.1.3. Heterogeneity Support

Figure 4.9 example code includes a ***target device*** directive that indicates two target devices for the defined task: *smp* and *fpga*. This means that any invocation to this task can be run either in the *smp* or the *fpga*, transparently to the programmer. The code to be run is the same, one accelerated in the FPGA, and the another executed in the SMP. In the case of the *fpga* device, the ***num_instances*** clause is used to express how many instances of the given IP core (*fpga* task) the programmer decides to generate; in this case three instances. Those tasks, of the same type, may potentially run in parallel at runtime in both *smp* and *fpga*. The runtime system will know through the configuration file (*xtasks.config*) the number of instances available of each accelerator type defined by the programmer.

On heterogeneous environments, Nanos++ has a specific subset of threads that represent each of the heterogeneous devices. We call these threads *helper threads*. Figure 4.11 shows, on the left side, the code invoking the heterogeneous task *matrix_multiply* and, on the right side, the overview of threads and task pool in the runtime. The orange thread (thread number 4, on the right hand side of the Global thread team) in the figure is one of those helper threads. In this particular example, it may represent one FPGA accelerator.

Tasks can be also annotated with the ***implements(funcname)*** clause, indicating that such task is a different implementation of the same algorithm that *funcname* implements. This allows the runtime system to select the *best* version to

```
1  #pragma omp target device(smp) copy_inout([BS]c) \
2              implements(matrix_multiply)
3  #pragma omp task in([BS]a, [BS]b) inout([BS]c)
4  void matmulBlockSmp(float a[BS][BS],
5     float b[BS][BS], float c[BS][BS]) {
6    const float alpha = 1.0;
7    const float beta = 1.0;
8    cblas_gemm(CblasRM, CblasNT, CblasNT, \
9               BS, BS, BS, alpha, a, BS, b, BS, beta, c, BS);
10 }
```

Figure 4.12. Implements version of SMP Matrix Multiply code (no castings done)

run at any given point in time. This is done by applying a scheduling policy that takes these alternative implementations into account.

Tasks annotated with the *implements* clause implement the same functionality as other tasks but with a different code. At compile time, two (or more) versions of the task are built targeting different computing units. At runtime, those tasks can be executed on an SMP core or more devices. This means that when the runtime system finds one of these tasks in the ready queue, it can be grabbed by a regular worker thread, that will execute the SMP version of the task in a SMP core. Or the task can be grabbed by one of the *helper threads*, and then the device version of that task will be executed in the device represented by the thread, transparently to the programmer (as shown in Figure 4.11 for the Matrix Multiply). Figure 4.12 shows the code that implements the function listed in Figure 4.9 in SMP by using OpenBLAS gemm.

4.2.1.4. Memory Transfers Support

Tasks executing in devices, with their own local memory, may need copy data from/to the device. In particular, the device memory space is main memory in the SMP, accessible from the accelerators and the SMP cores, physically contiguous, pinned and non-cacheable. With ***copy_deps*** clause the programmer indicates that all the dependences will require, at runtime, device memory space for copies between host memory and accelerators local memory. Alternatively, ***copy_in/out/inout(list-of-variables-with-size)*** clauses indicate the list of parameters of the task that needs to be copied to/from the accelerator and deactivate the by default *copy_deps* clause. In both cases, the runtime takes care of allocating device memory and copies between user and device memory for the list of parameters labeled as copies. Task parameters that are not indicated to be copied by the programmer should have been previously allocated in device memory (using our Nanos++ runtime API) so that the accelerator can access to that memory. If any of the task parameters is a scalar it is not needed to specify any copy, neither the user has to pre-allocate device memory; it is directly passed to the accelerator.

As mentioned, a wrapper is generated for each task so that accelerators and runtime can communicate with each other. The work to be done by the wrapper is to get information of the task (address of each parameter) and output

final signal, and optionally (based on some compilation flags), get/write tracing information, reserve local memory for some of the parameters, copy the data from/to device memory to/from local memory, and do timing instrumentation. In detail, the wrapper reads the address of each parameter of the accelerator using an input stream port of the wrapper IP, which is connected to our Task Manager IP (see Figure 4.10). Then, it maps the parameter address to an IP port connected to the external memory, using the AXI protocol, and copies the parameter data from/to device memory to/from local memory if required. In the case that copies are not requested, the kernel code of the programmer can access to the device memory without any change. This makes programming easy and can be useful to apply blocking techniques in an application from inside the FPGA, as shown in the N-body application below.

4.2.1.5. Instrumentation Support

The OmpSs@FPGA ecosystem allows to trace the execution of the runtime threads (running in cores) and accelerators. For the threads, it provides information at application and runtime levels so that the programmer can analyze both the application and runtime internals as the creation of tasks, task executions, taskwaits, etc. For the accelerators, the current support provides the user with the information of execution time of the data movements done by the wrapper and the computational time of the kernel. Figure 4.18 shows an execution trace where this information is shown for three accelerators of the Matrix Multiply application.

This tracing feature implies hardware support for timing within the bitstream and is transparently done to the programmer, which only has to activate the corresponding compilation flag. The execution trace we generate is done using an internal tracing library and it can be visualized by Paraver tool [1].

4.2.1.6. Experimental Environment

The communication logic and all the hardware accelerators are coded in C with Vivado HLS directives. The final system designs are synthesized with Vivado Design Suite 2016.3.

The hardware platform contains a Zynq Ultrascale+ MPSoC Chip XCZU9EG-FFVC900 [31]. It includes the Application Processing Unit (APU) with 4 ARM Cortex-A53 cores (that operate at 1.1GHz) and a FPGA. It has a 4GB DDR4 as main memory.

Sequential and parallel execution time of OmpSs applications are obtained in the system which operates on Ubuntu Linux 16.04. We also use performance tools Extrae and Paraver[1] to analyze the application behavior in our system.

4.2.1.7. Applications

Three applications have been analyzed with our current workflow: Matrix Multiply, Cholesky and N-Body. Table 4.1 summarizes the characteristics of each application.

Following subsections explain how each application has been implemented in the heterogeneous system through successive High Level Optimizations using the OmpSs@FPGA workflow.

Application	Description	Parameters
Matrix Multiply	Blocked matrix multiply in square blocks	Matrix size, Block size
Cholesky	Blocked Cholesky decomposition of a matrix	Matrix size, Block size
N-Body	Blocked N-body simulation	Number of particles, particles in a block, time-steps

Table 4.1. Summary of applications' characteristics

Matrix Multiplication

The first application ported to the OmpSs@FPGA framework was the Matrix Multiplication. Despite its apparent simplicity it is a key application as it includes several of the properties that are found in common HPC problems as a regular dependence pattern and a blocked implementation that involves moving several times the same data to and from the accelerators. Figure 4.9 presents the initial code of a matrix multiply algorithm for a block of $BS \times BS$ size that can be used to implement a blocked matrix multiply.

In order to port this code to FPGA and use a good part of its resources, some directives should be added to take advantage of the parallelism in the innermost loop. Better performance would be achieved when all the multiplications and additions of this loop are performed in parallel. Figure 4.9 shows the code with the High Level Synthesis (HLS) pragmas used to obtain the parallel version of the loop. The key pragma in the code is *PIPELINE II=1* that says that an iteration of the second loop should start each cycle ($II = 1$). To obtain this performance, the innermost loop should be completely unrolled. To accomplish this goal, all the elements in a row of matrix a and all the elements in a column of matrix b should be read each cycle. Pragmas *array_partition* make the compiler to distribute a by columns (second dimension $dim = 2$) or b by rows (first dimension $dim = 1$) in different block RAM (BRAM) memories by a factor that is half the side size of such matrices (as each BRAM has 2 read ports).

The first option to create a good accelerator for the matrix multiplication in the FPGA was to try to create an accelerator as big as it would fit in the FPGA. After some tests, we found that an accelerator with $BS = 128$ fits really well (even three of them) in the used FPGA but with $BS = 256$ it was also possible to obtain a successful compilation.

Cholesky decomposition

Cholesky Factorization is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. It computes $A = LL'$, with A an $n \times n$ matrix and L lower-triangular. The side of the matrix A, n is the first parameter of the application. The decomposition is made by blocks of any size (the side of the blocks is the second parameter of the application) which results in four different kernels. Figure 4.13 shows the code

```
1  void Cholesky(float **A) {
2    int i,j,k;
3    for (k=0; k<BS;k++){
4      spotrf(A[k*BS+k]);
5      for (i=k+1;i<BS;i++)
6        strsm(A[k*BS+k], A[k*BS+i]);
7      for (i=k+1;i<BS;i++) {
8        for (j=k+1;j<i;j++)
9          sgemm(A[k*BS+i], A[k*BS+j], A[j*BS+i]);
10       ssyrk(A[k*BS+i], A[i*BS+i]);
11 } } }
```

Figure 4.13. Cholesky application with its four composing kernels

of the Cholesky decomposition as it is decomposed in its four kernels: spotrf, strsm, sgemm and ssyrk. The SMP version is computed using the OpenBLAS version of the kernels while the FPGA version implements these kernels in C and compiles them through the HLS tool.

N-Body simulation

The N-Body simulation computes the problem of predicting the individual motions of a group of objects interacting with each other. Figure 4.14 shows the main loop (*nbody*) of the application. All the forces are computed, in blocks, for each particle against all the remaining particles. Then, the particles velocity and positions are updated with the forces previously computed. This is done for as many time steps as desired.

In order to decompose the problem in simpler tasks, a routine that computes the interaction of BS objects against a set of other BS objects is used. The computation of the forces for a block of particles is done considering all other blocks of particles. Each call to `calc_forces_BLOCK` function is defined as a task, which allows the OmpSs programming model to execute them in parallel when possible.

4.2.1.8. Possibilities for performance improvements

With the help of OmpSs@FPGA, different techniques can be used to improve the performance obtained with the different applications analyzed. In this section, some of these techniques will be reviewed.

Blocking Performance impact

There may be applications with a significant amount of work to be done that may not require any SMP computation, and then, could be completely executed in the accelerators. However, the limitation of the local memory inside the accelerators (due to the available FPGA resources) and also, the communication overhead associated, usually reduces the possibility of performing all the computation in the FPGA or the performance that can be achieved. A common approach in task-based programming models is to define tasks that can operate with a limited block size and then, perform the overall computation by blocks.

```
1  #pragma omp target device(fpga) \
2     copy_in([PARTICLE_SIZE*BS]block1, \
3             [PARTICLE_SIZE*BS]block2) \
4     copy_inout([FORCE_SIZE*BS]forces)
5  #pragma omp task
6  void calculate_forces_BLOCK (for_ptr_t forces, \
7    part_ptr_t block1,part_ptr_t block2,char safe);
8
9  void calc_forces(force_t *forces,
10       part_t *bl, int n_blocks) {
11   for (int i = 0; i < n_blocks; i++) {
12     for (int j = 0; j < n_blocks; j++) {
13       for_ptr_t fo = (for_ptr_t)(forces+i);
14       part_ptr_t b1 = (part_ptr_t)(bl+i);
15       part_ptr_t b2 = (part_ptr_t)(bl+j);
16       char safe = (b1 == b2);
17       calculate_forces_BLOCK(fo, b1, b2, safe);
18 } } }
19 ...
20 void nbody(part_t *part,force_t *forces,
21      int n_blocks, int timesteps) {
22   for(int t = 0; t < timesteps; t++) {
23     calc_forces(forces, part, n_blocks);
24     update_part(n_blocks, part, forces);
25   }
26 }
```

Figure 4.14. N-body main loop and blocking version of the calculate_forces

Name	B_18Kb	DSP48E	FFs	LUTs
2 BLOCK	88 (9.7%)	1014 (40.2%)	236888 (43.2%)	171200 (62.5%)
ALL FPGA	44 (4.9%)	507 (20.1%)	123446 (22.5%)	88481 (32.3%)
2 SUBBLOCK	107 (11.7%)	1014 (40.2%)	242864 (44.3%)	177116 (64.6%)

Table 4.2. Resources used by N-Body kernels in XCZU9EG-FFVC900

In the case of FPGA accelerators, applying blocking in the code running in the SMP and using the accelerators to perform the block processing may imply several synchronizations and communications, in addition to several copies of task parameters between user and device memory. However, applying blocking from the accelerator can help to decouple SMP and accelerator task executions and reduce the total number of synchronizations between Nanos++ runtime and accelerators.

N-Body simulation is the selected application to demonstrate how to use blocking to improve the performance when using OmpSs@FPGA. The first implemented version simply puts the kernel `calc_forces_BLOCK` into the FPGA trying to make the computation as fast as possible. It was possible to fit 2 instances of the function that computes the problem for 128 particles in the FPGA. Table 4.2, row **2 BLOCK** shows the resources used by this and all the other implementations presented in this section. The working frequency for all of them is 200MHz.

With the OmpSs@FPGA ecosystem it is easy to compare the performance obtained by the version that executes the tasks in the FPGA against the parallel version that uses the SMP cores to compute the tasks. Simply by changing the target device in the first line in Figure 4.14 from *smp* to *fpga* and back, the same code can be executed using the different resources in the system (using Nanos arguments to adjust the number of resources used in each execution). Figure 4.15 shows the time in logarithmic scale used by these OmpSs implementations when using a different number of resources to compute a 16384 particles problem with 8 time steps. As it can be seen in the figure, the runtime is able to obtain a near perfect parallelism when executing on the 4 cores available in the system. However, it can also be seen that the FPGA implementation is several times (15x) faster than the SMP implementation, making this problem a good fit for FPGA execution in the analyzed system. On the other hand, it can easily be observed by the programmer that the 2 accelerators version (**2 FPGA**) has almost the same performance as the 1 accelerator version (**1 FPGA**). The fact that Nanos helps testing all the versions by simply changing the execution command line, allowing the programmer to see that there is a problem with this accelerator. In this case, the accelerators are so fast that the capacity of the threads to create tasks is the performance limiting factor.

To further improve the performance, a new accelerator was programmed that took care of executing the whole `calc_forces` function inside the FPGA (**ALL FPGA** column in Figure 4.15 and **ALL FPGA** row in Table 4.2). Figure 4.16 shows this *calculate_forces* blocking version from inside the FPGA. In this case the *calculate_forces_BLOCK* is not defined as a task. The parameters *forces, block1, block2* are specified to be copied. However, we have specified, at compile time, that the wrapper does not reserve local memory for the parameters (neither

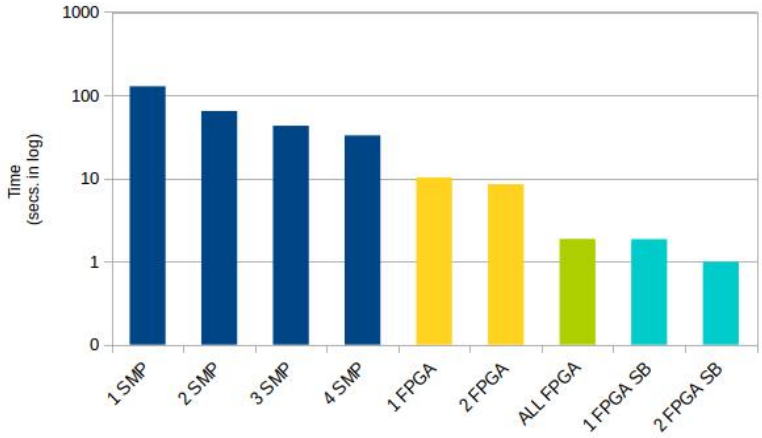


Figure 4.15. Time of N-Body execution with different blocking.

perform copies) but, connects the parameter variables of the task to external memory ports of the IP. That means that each access to the data of the parameters is actually accessing the external device memory transparently to the user. The programmer can perform copies to local memory (local variables in the code) and process the local copy (in BRAM) to avoid continuously accessing external memory. In the code of the figure the programmer uses *memcpy* (actually this *memcpy* is interpreted and optimized by Vivado HLS) to perform copies to local variables. These local variables are usually mapped to BRAM of the FPGA. As it can be seen this accelerator is several times faster than the previous one, although it doesn't use even half the resources available in the FPGA fabric, it doesn't make sense to fit two of them in it because there is no parallelism available. To obtain some parallelism over this last version, a new FPGA accelerator was developed. This new version receives the list of blocks to compute and iterates over them. With this approach (**FPGA SB** columns in Figure 4.15 and **SUBBLOCK** row in Table 4.2) two instances of the accelerator fit in the FPGA and were able to obtain a 1.87x over the previous version and a 128x over the version using 1 SMP core.

Implements and Dataflow Performance impact

In order to obtain the best possible performance out of heterogeneous systems it is crucial to use all the available resources whenever it is possible. In addition to making easy the programmability of FPGA accelerators and taking care of the necessary data transfers, OmpSs@FPGA also presents a *implements* clause that is really useful for heterogeneous systems.

Along with the implements and the Matrix Multiply presented above, different matrix multiply accelerator sizes were tested in the FPGA fabric available in the system. Table 4.3 shows in rows **1 128 Acc**, **3 128 Acc** and **1 256 Acc** how many resources it took to synthesize one or three accelerators of **BS** size 128×128 (128), or one accelerator of **BS** size 256×256 (256). All the accelerators listed worked at 300MHz and as it can be deduced from the reported sizes it is not possible to fit four 128 size accelerators or two 256 size accelerators in the FPGA.

Figure 4.17 displays the performance obtained by the three different approaches. Columns labeled **0 SMP** show the performance obtained by executing the ap-

```
1  #pragma omp target device(fpga) \
2          copy_in([PART_BSIZE*n_blocks]block1) \
3          copy_in([PART_BSIZE*n_blocks]block2) \
4          copy_inout([FORCE_BSIZE*n_blocks]forces)
5  #pragma omp task
6  static void calculate_forces(for_ptr_t forces,
7          part_ptr_t block1, part_ptr_t block2, int n_blocks) {
8     const int pbs = sizeof(float)*PART_BSIZE;
9     const int fbs = sizeof(float)*FORCE_BSIZE;
10    for (int i = 0; i < n_blocks; i++) {
11       for_ptr_t  lforces[FORCE_BSIZE];
12       part_ptr_t lblock1[PART_BSIZE];
13       memcpy(lforces, forces + i*FORCE_BSIZE, fbs);
14       memcpy(lblock1, block1 + i*PART_BSIZE , pbs);
15       for (int j = 0; j < n_blocks; j++) {
16          float lblock2[PART_BSIZE];
17          memcpy(lblock2, block2 + j*PART_BSIZE, pbs);
18          calculate_forces_BLOCK(lforces,
19                                 lblock1, lblock2, (i == j));
20       }
21       memcpy(forces + i*FORCE_BSIZE, lforces, fbs);
22    }
23 }
```

Figure 4.16. FPGA Blocking version of the calculate_forces function of N-body

Name	B_18Kb	DSP48E	FFs	LUTs
1 128 Acc	287 (15.7%)	642 (25.5%)	76147 (13.9%)	54462 (19.9%)
1 256 Acc	648 (35.5%)	1280 (50.8%)	183646 (33.5%)	107207 (39.1%)
3 128 Acc	537 (58.9%)	1920 (76.2%)	311271 (56.8%)	169670 (61.9%)
3 256 DF	644 (70.6%)	1925 (76.4%)	341902 (62.4%)	208906 (76.2%)

Table 4.3. Resources used by Matrix Multiply kernels in XCZU9EG-FFVC900

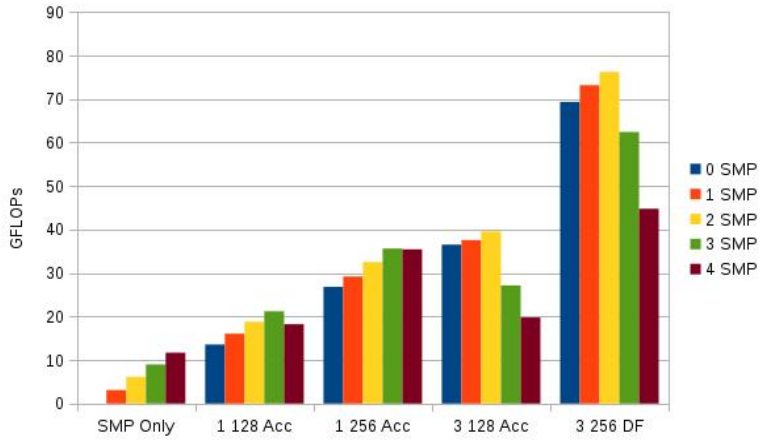


Figure 4.17. GFLOPs for Matrix Multiply with different FPGA accelerators

plication in the FPGA accelerators alone, while columns **1 SMP** to **4 SMP** display the result of using from 1 up to 4 threads to perform tasks with the code in Figure 4.12 in parallel with the FPGA. As it can be seen, the best approach in terms of performance is not to fit a single large accelerator (**1 256 Acc**) but to use three smaller accelerators in parallel. Also, note that for any possible solution, the use of the SMP to compute matrix multiplication blocks in parallel always improves the resulting performance. When using four threads with 1 accelerator or three or more threads with 3 accelerators the performance drops from the maximum. This is due to the fact that there is over-subscription. Effectively, with 1 accelerator, 1 thread is used to send tasks to the FPGA. With three accelerators, 2 threads should be used in order to feed the accelerators properly. However, the runtime is intelligent enough to always use the best approach in the default configuration given the maximum performance for every configuration. Also, it is important to note that the exploration of all these possibilities is done with the same code, changing only the *BS* size and the number of instances of the accelerators, so reducing the programmability effort to a minimum.

In order to further improve the performance results, the trace of the execution with 3 128 Matrix Multiply accelerators was extracted. Figure 4.18 shows the trace of this execution. The four top lines in blue represent the threads and do not show any information. The three bottom lines show in blue when the corresponding FPGA accelerator is not working and in yellow when it is reading data. In brown it can be seen the computation time and in green the writing of the output matrix back to the memory. As it can be seen from the trace, the computation time is around 4 times shorter than the data movement time. Furthermore, the data copies are not overlapped with the computation. From this observation a way to improve the accelerators was devised. The idea is that doubling the accelerator size increases the data size by four times but the computation size by eight times. Therefore, if the number of cycles per operation is not significantly increased this will result in a better balance in the accelerator design while not incrementing the DSP usage.

Following this idea to design better balanced accelerators, the number of computations were limited by setting the initiation interval of the innermost loop to 2 cycles (so making the same effective computations in the 256 DF accelerator

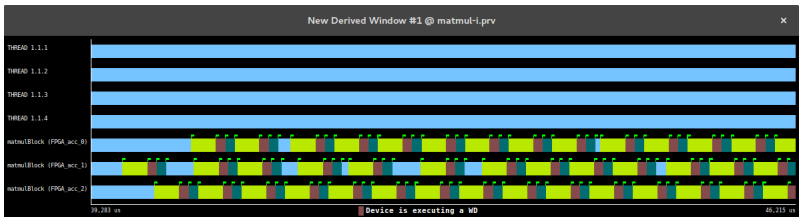


Figure 4.18. Trace of FPGA execution with 3 128 Matrix Multiply accelerators

as in the 128 one). Also the *DATAFLOW* pragma was used in order to overlap the data copies with the computation. The result of these changes can be observed in Table 4.3 row **3 256 DF** and Figure 4.17 column **3 256 DF**. These new accelerators fit in the FPGA available in the system while nearly doubling the performance of the previous 3 128 accelerators. They take the same time to compute the results but transfer half the blocks and overlap part of these transfers with the computation. In addition the *implements* clause still adds some performance to the FPGA by using the SMP achieving 76 GFLOPs with little more that 5 Watts consumption.

Heterogeneity and Programmability impact

Another common problem when dealing with complex applications composed by different kernels in heterogeneous environments is how to distribute such kernels over all the different resources. The OmpSs@FPGA environment can help with this distribution by allowing different possible mappings to be tested easily and without burden to the programmer.

Cholesky decomposition application is composed of four different kernel tasks that present a complex dependence pattern that grows exponentially with the size of the problem. Table 4.4 shows the FPGA resources used by each kernel of the Cholesky application when implemented in the FPGA when different blocking sizes were used in the application. As it can be extrapolated from the results, it is impossible to fit all the 4 kernels with a blocking size of 128 in the FPGA due to the limited resources available. However, from previous results it is known that the bigger the accelerator, the better the obtained performance.

One of the first implications of using one accelerator for each kernel is that the execution in the FPGA would be a sequential one. However, using the *implements* clause explained in the previous section would help us to improve the performance significantly by allowing threads to also execute kernel functions. Figure 4.19 shows the time used when executing 6 different versions of the same Cholesky problem to solve a 2k equation matrix. Two implementations with different block sizes (32 and 64) were tested using the SMP cores to solve the problem (using OpenBLAS implementations of the kernels), using the FPGA accelerators to solve the same problem and also using both (through the implements clause).

As it can be seen in Figure 4.19 the initial FPGA alternatives (Columns FPGA and SMP+FPGA) of the code do not lead to good performance results compared against the SMP only alternative. Even the version that uses both SMP and FPGA (SMP+FPGA) to compute the result is slower than the SMP only version. On one hand, the reason behind this behavior is that including four different accelera-

Name	B_18Kb	DSP48E	FFs	LUTs
fgemm32	68 (3.7%)	160 (6.4%)	19771 (3.6%)	15559 (5.7%)
fsyrk32	36 (2.0%)	160 (6.4%)	19822 (3.6%)	16149 (5.9%)
ftrsm32	36 (2.0%)	104 (4.1%)	11482 (2.1%)	10875 (4.0%)
fpotrf32	10 (0.6%)	22 (0.9%)	3487 (0.6)%	3302 (1.2%)
fgemm64	74 (4.1%)	160 (6.4%)	23887 (4.4%)	30032 (11.0%)
fsyrk64	42 (2.3%)	160 (6.4%)	23849 (4.4%)	30727 (11.2%)
ftrsm64	42 (2.3%)	250 (9.9%)	28734 (5.2%)	25753 (9.4%)
fpotrf64	28 (1.5%)	22 (0.9%)	3514 (0.6%)	3350 (1.2%)

Table 4.4. Resources used by Cholesky kernels in XCZU9EG-FFVC900

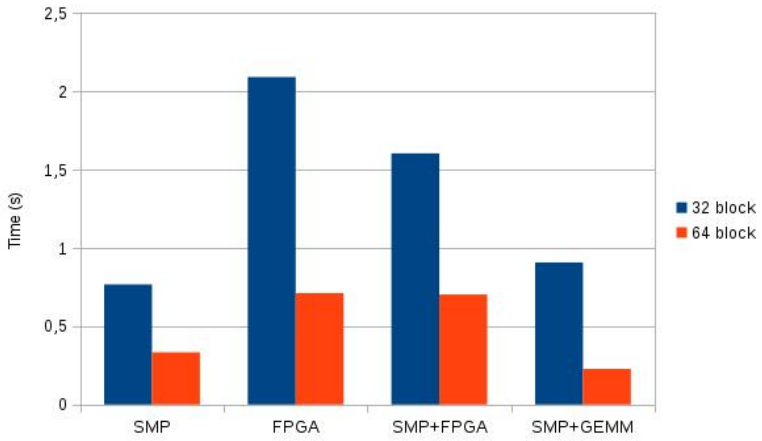


Figure 4.19. Time of Cholesky execution with different task mappings

Size	Case	#SMP	#HWACC0	#HWACC1	#HWACC2	#HWACC3
(2k, 32)	4a	0	41664	2016	2016	64
	4a+4t	17992	23839	2015	1850	64
	4g+4t	4096	10300	10371	10451	10542
(2K, 64)	4a	0	4963	496	496	32
	4a+4t	2344	2667	493	448	32
	4g+4t	1024	1209	1235	1258	1258

Table 4.5. Number of tasks executed in different hardware units

tors in the FPGA limits the performance of each accelerator. On the other hand, the Cholesky algorithm is not well-balanced among all its kernels. Table 4.5, rows **4a** show the number of tasks executed in each accelerator when using only the FPGA accelerators to execute all the kernels. Obviously, this number of tasks corresponds to the number of tasks of each type that these blocking versions have. It can be seen that there are several more *gemm* tasks than any other type, so when the *implements* clause is used and the tasks can be executed in both SMP cores and FPGA (rows **4a+4t**), mainly all the tasks executed in SMP cores in parallel with the accelerators are *gemm* tasks.

To solve the aforementioned unbalance, a second FPGA accelerated version of Cholesky (**4g+4t**) is implemented with 4 *gemm* accelerators on the FPGA. The remaining kernels were implemented using the SMP cores. From the point of view of the programmer, this new version only implies incrementing the number of instances of the *gemm* FPGA accelerator and not including the instances of the other accelerators, a change that can easily be done in the source code. The rest of the whole program remains exactly the same. As it can be seen in table 4.5, rows **4g+4t**, the balance is significantly better with this approach. Figure 4.19, columns **SMP+GEMM** show the performance results obtained by this last version of the code that uses FPGA accelerators to compute only the *gemm* tasks. This last version outperforms the initial SMP only version and illustrates how using the OmpSs@FPGA framework simplifies the accelerator space exploration keeping the necessary changes made by the programmer to a minimum.

4.2.2. OmpSs@Cluster

In the context of LEGaTO, we modify OmpSs-2@Cluster in order to allow integration for supporting accelerators, i.e. OmpSs-2@CUDA, OmpSs-2@FPGAs. Accordingly, transparent to the user, we introduce conceptual model of the steps required to execute a task which, in Nanos6 terminology is called the Execution Workflow. In order to execute a task, once a scheduling decision has been made, Nanos6 creates an Execution Workflow for that task, which is a dependency graph that consists of a number of Execution Steps forming a small dependency graph.

The execution workflow is the mechanism that formalizes the execution of a Task in Nanos6. It organizes the steps we need to perform for the execution of a Task, from the moment it becomes ready (all its dependencies are satisfied) until it completes its execution. The steps of the execution workflow form a workflow graph. A step of the workflow can start only after all its predecessors

are completed.

Figure 4.20 presents the main components of the Task Execution Workflow in Nanos. In general, in order to execute a Task on a device we need to perform the following:

1. Allocate and Pin Memory. For every symbol (variable) the Task is accessing we need to allocate memory on the device's address space and for some devices we need to pin that memory, so it does not get evicted (swapped out).
2. Data Copy. For every data access of the Task we might need to copy the latest data in the memory allocated for the symbol the data access belongs in. This copy can be synchronous or asynchronous, depending on the capabilities of the devices that participate in the copy operation.
3. Execute the Task. Once all the data copies are completed we can execute the body of the task
4. Release Data Accesses. After the execution of the task we can release its data accesses. This means that that we can release the dependencies of subsequent tasks fast, taking advantage of OmpSs early release of dependencies feature.
5. Notification. We need to inform the dependency system that the Task has completed.
6. Unpin. Once the task is completed we can unpin the memory it is using. In reality, here we probably reduce a reference counter and let the memory subsystem decide when to unpin/deallocate the memory on the device.

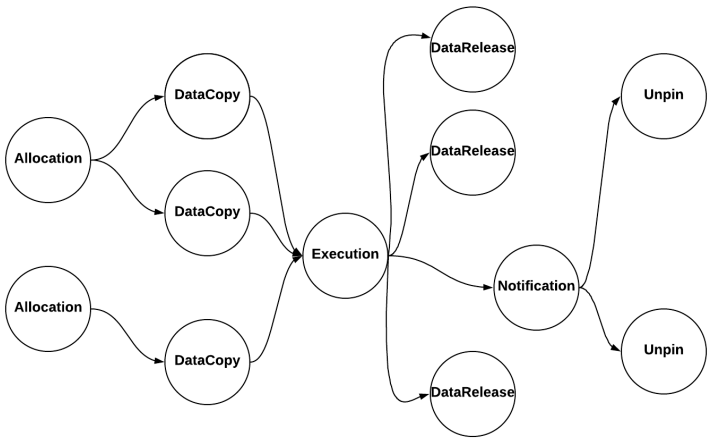


Figure 4.20. Task Execution Workflow in Nanos6

The need for the Execution Workflow, stems from our desire to support execution of Tasks in multiple types of devices, e.g. Cluster, FPGA, CUDA, SMP, etc. As a result, the Step objects of the the Execution Workflow are specialized depending on what type of device we execute the Task on.

Each device implementation, defines the way to allocate memory on the device, copy data from/to the device towards the host (SMP). It knows how to offload-/execute code on that device and finally it determines what sort of actions we need to perform in regards with releasing dependencies after the task is finished.

An `ExecutionWorkflow` object is created for a Task, once the scheduler decides on which device to execute it. The worker thread that handles the Task, creates and executes the `ExecutionWorkflow` and starts its execution.

The workflow creation method takes as arguments the compute device that will execute the Task and a memory device of that compute device, which will be used to store the data accesses of the Task. The type of the Step objects to be created depends on the type of these two devices. For example, if the target is a CUDA device Nanos will create a CUDA`ExecutionStep` for executing the Task.

Once the Execution Workflow object is created, the worker thread will start its execution. The execution starts from the *root* Step objects, i.e. the Step objects without predecessors. Keep in mind, that some target devices might not have use for certain types of Steps. For example, the Execution Workflow to offload a Task to a remote Cluster node, does not need to perform the `Allocation` steps, because that part is handled by the cluster-related code, implicitly.

The execution of a Step might be synchronous or asynchronous. Synchronous Steps perform their operation and release the next Steps. Asynchronous Step implementations, setup the asynchronous operation and program a callback which will release the subsequent steps, once the operation is completed.

4.2.2.1. Host Execution Workflow

The simplest form of the Execution Workflow is the one we create for executing a Task on an SMP core. In this case, we do not need to allocate memory for the accesses, executing the Task is as simple as calling a function and notifying the dependency system for the Task's completion is straightforward, since we do not need to communicate with a remote instance of Nanos.

The only Step that might perform an action is the `DataCopy` Step, which might need to bring data from a remote address space, e.g. a remote Cluster node. We determine if this is the case, when we create the ExecutionWorkflow object for the Task. At that point, we iterate through the data accesses of the Task and if one of them is located currently on an address space other than the local memory, we setup the corresponding (CUDA, Cluster, FPGA, etc.) `DataCopy` Step.

4.2.2.2. Cluster Execution Workflow

The Cluster Execution Workflow slightly differs that that of other devices, due to the fact that Cluster devices are able to create subtasks. This means that, when we offload a task to a remote cluster device, we offload part of the task-graph of the application. The Execution Workflow needs to be aware of this fact and perform all the operations needed, in order to connect the two parts of the dependency graph, i.e. exchange dependency and location information between cluster nodes. This also means, we need to create two Execution Workflows one for offloading the Task from the original to the remote node and on for actu-

ally running the Task on the remote node. Figure 4.21 describes graphically the process of offloading a Task to a remote node.

On the original node (left side), we create an Execution Workflow which consists of two special types of `DataCopy` Steps, namely `DataLink`, one `Execution` Step and one `Notification` Step. The `DataLink` Steps are responsible for propagating location information on the remote task. In OmpSs@Cluster we only *fetch* data in when we want to execute a Task, so at this point the `DataLink` Steps will only propagate the location information about the Task data accesses. The cluster `Execution` Step is the responsible for actually offloading the Task to the remote node. It creates and sets up and sends the cluster command message for offloading the Task to the remote node. Finally, the `Notification` Step, on this cluster Node will interact with the local part of the dependency system, once it receives a command message declaring that the remote Task is completed.

On the remote node (right side), things are different. Here, we will actually execute the Task so, from the point of view of this node, this is a Host Execution Workflow, with the sole exception of the `DataRelease` Steps. The `DataRelease` Step, is responsible for informing the original node that a data access has been released (once the Task finishes execution). The `DataLink` and `DataRelease` Steps are the Steps that link the two parts of the dependency graph for cluster execution. The former sends information from the original node to the remote task and the latter, sends information back to the original node.

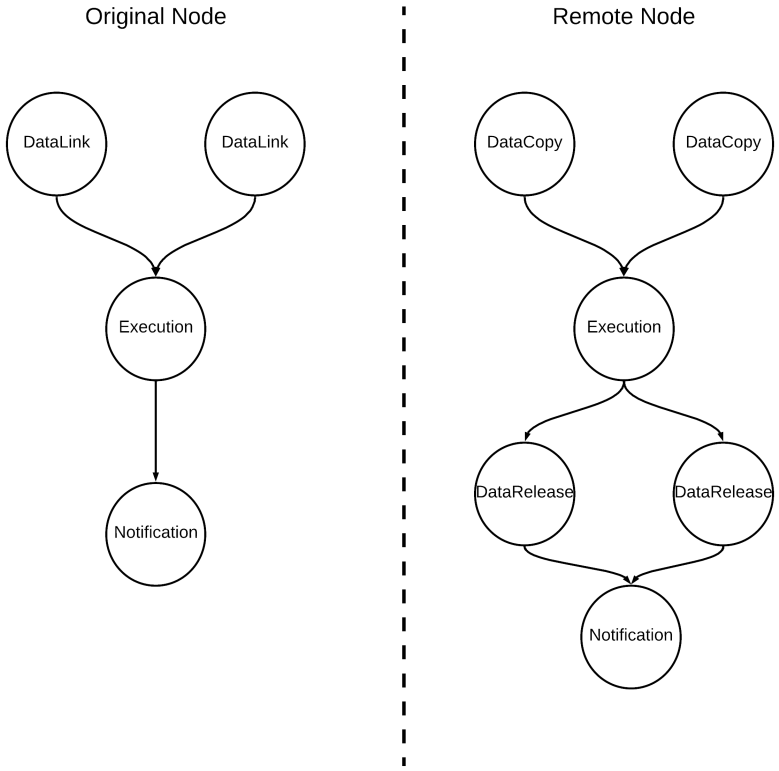


Figure 4.21. Using the Execution Workflow to execute an OmpSs@Cluster Task

These Steps are currently used only for executing cluster Tasks, but if in the future we have another device implementation, e.g. FPGA, which allows creat-

ing subtasks on the device, this would mean that the device workflow needs to implement `DataLink` and `DataRelease` Steps as well.

5. Runtime support for Fault Tolerance and Security

This chapter describes the current fault tolerance support in the LEGaTO runtime to ensure reliable operation of the scalable and energy efficient toolchain.

5.1. GPU Checkpointing

In this chapter we present the contributions of our current work in respect to fault tolerance. In brief the following additions were performed:

1. We have extended the Fault Tolerance Interface (FTI) [2] to support transparent checkpointing of data in multi-GPU/multi-node systems.
2. We optimize the checkpointing procedure.
3. We have added support for differential checkpoint (dCP) for GPU aplications to reduce the amount of data stored during the checkpointing procedure.
4. We have added support for incremental checkpoint (iCP) for GPU aplications to incrementally add data to the checkpoint files.
5. We thoroughly evaluate our approach using different applications on a multi-node multi-GPU system.

5.1.1. FTI implementation

FTI is a library that provides an API to the developer to efficiently perform multi-level checkpointing. It is implemented in C/MPI and provides also a Fortran interface to the user. The developer uses library function calls to define which data needs to be checkpointed as well as at which execution points a checkpoint can be taken. At execution time the library is controlled using a configuration file which defines several parameters. This allows the user to compile once the application and select different parameters prior executing the application, for example select the file format of the C/R files. In Figure 5.1 we present a toy example of the FTI API.

Depending on the configuration file FTI may spawn extra MPI processes per node to perform asynchronous checkpoint and hide latencies of the file system. To guarantee that the library will not cause any damage to the application communication channels, FTI has a function call, *FTI_Init()* (line 6) that will perform all the necessary actions before the application starts the real execution. FTI_Init() will start by reading the configuration file that should be correctly filled by the user before the execution. Once the configuration has been checked, FTI will detect in which node each process resides and will write this topology in a file. Then, it will delegate a user-defined number of processes per node as FT-managers and will create two MPI communicators, one for all FT-managers and another for the application processes. The MPI communicator created by

```
1  int main(int argc, char *argv[]){
2    int rank, nbProcs;
3    double *h,*g;
4    int i;
5    MPI_Init(&argc, &argv);
6    FTI_Init(argv[1], MPI_COMM_WORLD);
7    MPI_Comm_size(FTI_COMM_WORLD, &nbProcs);
8    MPI_Comm_rank(FTI_COMM_WORLD, &rank);
9    h = (double*) malloc (sizeof(double)*nElements);
10   g = (double*) malloc (sizeof(double)*nElements);
11   initData(&h,&g);
12   FTI_Protect(0, &i, 1, FTI_INTG);
13   FTI_Protect(1, h, nElements, FTI_DBLE);
14   FTI_Protect(2, g, nElements, FTI_DBLE);
15   for(i = 0; i < N; i++){
16     FTI_Snapshot();
17     performComputations(h,g,i);
18   }
19   FTI_Finalize();
20   MPI_Finalize();
21 }
```

```
1  int main(int argc, char *argv[]){
2    int rank, nbProcs;
3    double *h,*g;
4    int i;
5    MPI_Init(&argc, &argv);
6    FTI_Init(argv[1], MPI_COMM_WORLD);
7    MPI_Comm_size(FTI_COMM_WORLD, &nbProcs);
8    MPI_Comm_rank(FTI_COMM_WORLD, &rank);
9    cudaMallocManaged(&h,sizeof(double)*nElements,
         flags);
10   cudaMalloc(&g,sizeof(double)*nElements);
11   initData(&h,&g);
12   FTI_Protect(0, &i, 1, FTI_INTG);
13   FTI_Protect(1, h, nElements, FTI_DBLE);
14   FTI_Protect(2, g, nElements, FTI_DBLE);
15   for(i = 0; i < N; i++){
16     FTI_Snapshot();
17     performComputations(h,g,i);
18   }
19   FTI_Finalize();
20   MPI_Finalize();
21 }
```

Figure 5.1. Source code using FTI. FTI API calls and variables are marked as red. On the left side we demonstrate the original FTI and on the extended one.

FTI for the application processes is called FTI_COMM_WORLD and replaces the global communicator used in the application (MPI_COMM_WORLD). By simply replacing the global communicator in such a way, FTI guarantees that no message will ever be exchanged between application processes and FT-managers within the application. During a checkpoint the application process performs a level 1 checkpoint, thus, the data is stored in a local storage device (SSD, NVMe), after all data is copied to the kernels I/O layer (in memory checkpoint) the application process execution resumes. The FTI-managers are responsible to wait for the data to be stored on the local storage device and afterwards to perform the user specified checkpoint level. For example for a level 4 checkpoint the FTI managers copy the level 1 checkpoint to the global parallel file system (GPFS).

Using the function *FTI_Protect* (line 10,11,12) the user can define a continuous memory region which will be stored in the C/R file upon the checkpoint procedure. The function can be called multiple times to protect different memory regions. After defining the memory regions the developer can call *FTI_Snapshot* (line 14) to instruct the library that a checkpoint can be taken. Whether a checkpoint will be actually taken depends on the user-specified checkpoint frequency defined in the configuration file. On recovery *FTI_Snapshot* will perform the actual recovery procedure. Finally, *FTI_Finalize* checks that all the FT-managers have finished their job and free all the resources.

5.1.2. GPU Support for FTI

Our target is to provide a single API to support checkpointing of different memory regions regardless of their actual physical location. An example using the GPU/CPU checkpoint API is presented in Figure 5.1. Noticeably, in line 10 the developer allocates memory space using a unified virtual memory (UVM) address , thus this address is accessible in the host code, whereas in line 12 the developer allocates a device pointer, which is not directly accessible through host code. In lines 14,15,16 the developer protects three different memory address, a host address , a UVM address and a device address however there are no API ex-

tensions. Interestingly, the checkpointing API has no extensions and is identical with the one described in Section 5.1.1. In *FTI_Protect* the developer specifies a single address which can be either a host-memory address, a device memory address or a UVM address and the FTI runtime library will handle accordingly each different address type.

5.1.2.1. FTI GPU/CPU implementation

FTI handles checkpoints in three different phases. The first, called initialization-phase, initializes the library and defines the protected memory regions. The second C/R-phase, corresponds to the actual C/R procedure where it moves the data from the device and host memory to the stable local storage device (SSD, NVMe) and vice versa in the case of recovery. When the checkpoint-phase is terminated, the application resumes normal execution, and the async-phase starts. In the async-phase the FTI managers perform the necessary actions for the various checkpoint levels.

To support Hybrid GPU/CPU support to FTI we extend the initialization-phase and the C/R-phase. In the initialization-phase we identify the physical location of the address, upon a *FTI_Protect* call the physical location of the data is internally determined. This is done through the CUDA driver API support, namely the function *cudaPointerGetAttributes(&attributes, address)*. The function raises an error when called with a host address, whereas it returns normally with a device or a UVM address. In the second case we further check the values of *attributes* field which provide information whether the address is UVM or not. In the end we tag each address as *CPU,GPU,MANAGED*.

When the checkpointing phase takes place, depending on the tag of each address we perform a different action. In the case of CPU or UVM addresses, we invoke the normal FTI C/R procedure. In the case of UVM addresses we use the CUDA driver to fetch the data from the GPU and move them to the stable local storage. Finally, in the case of *GPU* addresses, we overlap the writing of the file with the data movement from the GPU side to the CPU side. This is done through streams and asynchronous memory copies of chunks from GPU memory to host pinned memory. The procedure of transferring data from the GPU memory to the CPU is depicted in Figure 5.2. Each protected memory region is divided into smaller blocks. The size of the block, from now on called communication block (*cBlock*), is controlled through a configuration option. The CPU requests from the Host to Device (H2D) engine an asynchronous transfer of the first *cBlock*, when the *cBlock* is copied to the host memory, the CPU requests the next *cBlock* and starts performing the necessary actions with the current cBlock. The main actions are the following: i) Update the checkpoint integrity checksum ii) Copy the *cBlock* to the I/O layer through the respective I/O library call.

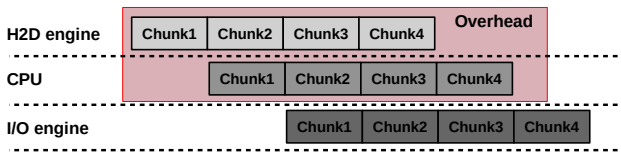


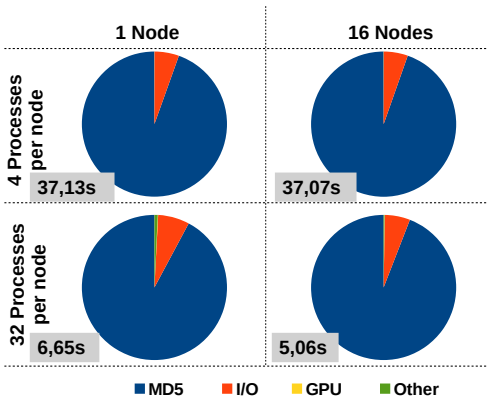
Figure 5.2. The device to memory transfer protocol. Ideally, all the data movements are overlapped. The user-application is delayed until all data is copied to the I/O layer.

When the data is copied to the I/O layer the CPU starts processing the next chunk, which ideally should already be copied in the host memory by the H2D engine. The application process does not wait for the I/O operations to finalize, when all data is moved to the I/O layer it informs the FTI-managers to start the background actions and resumes execution. The described scheme is optimal only if the time spent to compute the integrity checksum and copy the *cBlock* to the I/O layer is equal to the time spent to copy the data from the device to the host. In any other case, either the H2D engine or the CPU is idle.

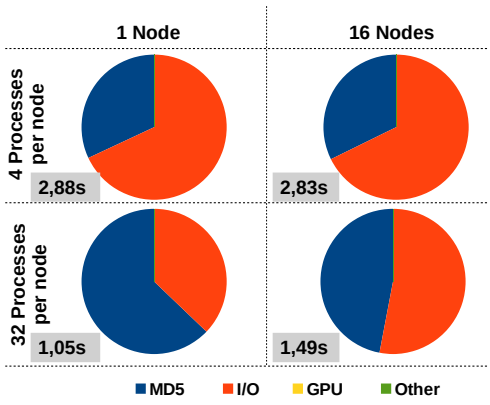
5.1.3. FTI Analysis and Optimization

In this section we analyze and optimize the FTI GPU checkpoint scheme. We use two micro-benchmarks for profiling and analysis purposes. The micro-benchmarks check the strong/weak scaling of our approach using different mixtures of device/host memory allocations. The first micro-benchmark allocates two memory buffers, the first buffer, called *hBuff*, is allocated on the host memory, whereas the second one, called *dBuff*, is allocated on the device memory. The size of each memory buffer is user defined. The application protects these two buffers and performs a checkpoint every 5 minutes. The second micro-benchmark is identical, but the device memory is allocated using a unified memory address space.

To test both the weak and the strong scaling of our approach we execute each benchmark on 4 different node/process/GPU mappings. Specifically, we executed experiments using 1 and 16 nodes, with 4 and 32 processes in each node. When executing the application with 4 processes, the GPU devices are not shared among the processes. In the case of 32 processes, each GPU is shared by 8 processes in round-robin fashion. The checkpoint size of each node is 48 Gb regardless of the number of processes within the node.



(a) Percentage of time spent to perform different actions during the checkpoint procedure.



(b) Percentage of time spent to perform different actions during the checkpoint procedure, using the optimized GPU MD5 impelementation.

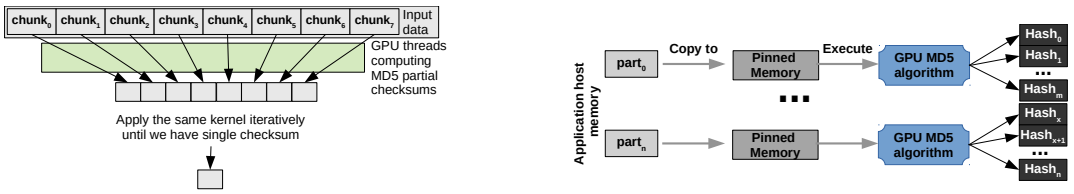
Figure 5.3. Execution time breakdown of checkpoints before optimization and after optimization.

In Figure 5.3a we depict the results of the executed experiments when using the non UVM micro-benchmark. The implementation demonstrates nice weak and strong scaling, since when we compare in pairs the case of 4 processes with 32

processes and 64 with 512 processes we decrease the problem size by a factor of 8 and we observe a 8x reduction on the overhead. Interestingly, even though the GPU devices in the case of 32 and 512 processes are shared, we do not observe any extra overheads. In the case of weak scaling when we compare in pairs the 4 processes with the 64 and the 32 with the 512 we observe that in pairs the executions present the same overhead. Figure 5.3a also presents the execution time spent in 4 categories, the execution time to compute the integrity checksum, the execution time to copy the data from the GPU, the execution time spent to copy the data to the I/O layer and the execution time spent to perform other minor actions during the checkpoint (e.g create directories, metadata etc.). The communication between the GPU and the CPU is completely overlapped, therefore there is almost no-overhead to move the data from the GPU to the CPU. Interestingly, the execution time is mainly spent in computing the integrity checksum of the checkpoint file. In other words, the CPU performs an increased amount of work before storing the data. The results of the remaining memory schemes and the UVM benchmark were omitted for brevity reasons as they presented similar results.

5.1.3.1. Optimization of Integrity checksum

To reduce the execution time overhead of the checkpoint procedure we need to reduce the execution time spent to compute the integrity checksum. FTI uses the MD5 algorithm [24] as an integrity checksum. The first step is to implement a GPU CUDA version of the MD5 algorithm presented in [12]. The algorithm is represented in Figure 5.4a. The input data is splitted into smaller chunks, cuda threads compute, in parallel, the MD5 hash of each chunk, after computing the hash values the MD5 algorithm is applied again on the computed hashes.



(a) Data flow of MD5 GPU computation.

(b) Data flow when applying the MD5 GPU algorithm to data that reside in the host memory.

Figure 5.4. MD5 computation of GPU and CPU data

Although, the parallel MD5 algorithm on FTI is easily applicable in the case of data accessible from the device (*GPU, managed*), this is not the case for the data that are allocated in the host memory. HPC applications have a huge memory footprint, hence copying the data from the host side to the device is not an option, as the device could not have enough free memory space to store temporalily the data and apply the algorithm. On the other hand, we could pin the user defined host memory regions and define them as zero-copy memory regions. Consequently, the memory would reside on the host side but it would be accessible from the device memory. Pinning large regions of memory reduces the available physical RAM on the demand-paging system. Therefore, we would heavily reduce the performance of the code executing on the host side.

Taking into account all the previous reasons, we implement the following communication scheme presented in Figure 5.4b. We split the host-memory to parts, each part is of equal size, and we process each part in sequential order. The process is as follows, each part is copied to a pinned memory location accessible from the device (zero-copy), we perform the GPU MD5 algorithm and we gather the computed hashes on a GPU memory location. When all parts are processed, the computed intermediate hash values are stored in the device memory, and therefore, we can execute the GPU-MD5 algorithm in the intermediate results to compute iteratively the final single checksum. Splitting the memory of the host into parts allows us to control the amount of memory overhead. The larger the size of each part, the larger the amount of memory overhead. Applying the correct algorithm within FTI is trivial, since the variables are already tagged as *CPU, GPU, managed*. Consequently, for protected variables tagged as *CPU* we use the *CPU-GPU* implementation, whereas for the *GPU, managed* we use the *GPU* implementation. In Figure 5.3b we present the results of checkpointing the micro-benchmark. On the bottom left corner of each of the pies we present the total overhead to perform the checkpoint. As expected the total time to perform a checkpoint is heavily reduced.

5.1.3.2. Task-Based Implementation

As discussed, the CPU is waiting for the GPU to compute the MD5 checksums and then it continues with the writing of the chunks. However, the computation of the MD5 checksum is completely independent from the writing of the file. Both tasks, writing of the file and the MD5 computation, only read the data and do not change any value, therefore both of them can execute in parallel. We exploit this observation and we assign a task to apply the MD5 computation on the data, and a second task to perform the storing of the data. The tasks are executed in parallel. The total overhead of this scheme is equal to the maximum execution time of this procedure. Noticeably, both tasks transfer data simultaneously through the *NVLINK*, this could result into latencies of the I/O task. However, copying data to the I/O layer is actually slower than copying data through the *NVLINK*, therefore this scheme should not result to extra overheads of the I/O operations as the bottleneck is the I/O layer and not the GPU-CPU transfers.

5.1.4. Evaluation

In Section 5.1.3 we have analyzed and optimized our implementation using synthetic benchmarks. In this section we will use real mini-apps to evaluate different checkpoint scenarios of our multi-node/multi-GPU checkpoint scheme.

5.1.4.1. HPC applications

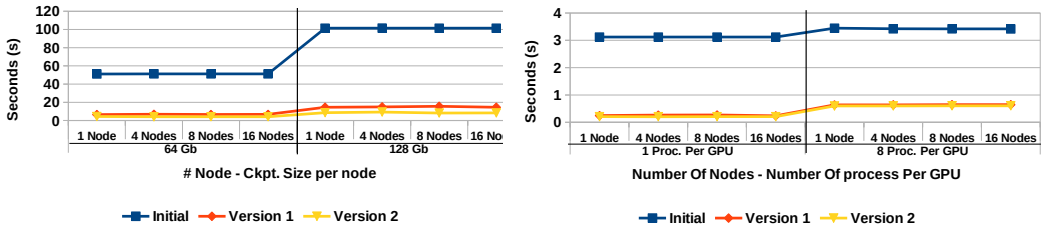
In this section we describe the applications used to evaluate our heterogeneous checkpoint methodology. 1. **Heat2D** is a 2D heat distribution simulation using a 1D domain decomposition. It simulates the transition from a non-equilibrium heat distribution to the equilibrium state. The large majority of memory used is allocated using the UVM by Heat2D. This allowed us to increase the tested data sets beyond the GPU main memory. 2. **Jacobi solver** is a real world example that iteratively solves the Poisson equation on a rectangle with Dirichlet boundary conditions. The algorithm uses second-order central differences to approximate the Laplacian operator on the discrete grid. The benchmark allocates memory

using exclusively *device* memory, and therefore, the majority of the checkpoint data is stored in the GPU side. 3. **Hydro** is a multi-node multi GPU benchmark [6] which implements a 2D Eulerian scheme using a Godunov method [11]. The application's checkpoint data is distributed almost evenly among the CPU and the GPU device, more precisely we simulate a problem with grid size equal to 25000×25000, the total checkpoint size is equal to 24Gb.

5.1.4.2. Experimental Results

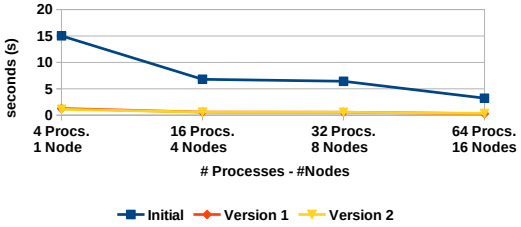
During the evaluation all checkpoints request a level 4 checkpoint (the final checkpoint file will be stored by the FT-managers to the GPFS). For every 4 user processes we assign an FT-manager. Consequently, when we execute an application with 4 application processes, we use an extra (fifth) process to transfer the local checkpoint to the GPFS system. In the evaluation we present only the overhead of the application process, as the FT-managers present minimum extra overhead [2].

We test the 3 different methods implemented in this work. The *initial* corresponds to the implementation without any optimization presented in subsection 5.1.2.1. The *version 1* corresponds to the version which utilizes the GPU to compute the MD5 checksum presented in Section 5.1.3.1 and the *version 2* corresponds to the task parallel version presented in Section 5.1.3.2.



(a) Execution time spent to checkpoint Heat2D.

(b) Execution time spent to checkpoint Jacobi.



(c) Execution time spent to checkpoint Hydro.

Figure 5.5. Execution Time spent to checkpoint different applications

Checkpoint UVM Address Space

We use *Head2D* to test the behavior of our multi-gpu/multi-node checkpoint methodology when the application is using UVM memory allocations. We checkpoint *Heat2D* for two different problem sizes, namely in the first problem we checkpoint 16Gb per process whereas in the second we checkpoint 32Gb per-process. The first problem size barely fits in the GPU main memory at once, the second problem size does not fit at once in the GPU main memory. In both cases, the CUDA-driver is responsible to transfer and manage the UVM allocated data.

We test different number of nodes, in each node we execute 4 processes, one per GPU device, therefore the GPU devices are not shared among the processes. Finally, the problem size is weakly scaled as the number of nodes increases. When we use 16 nodes the total size of the problem size and thus the total size of the checkpointed data is equal to 1Tb and 2Tb respectively.

Figure 5.5a depicts the results of our experiments for the different methods. The *x-axis* corresponds to the different problem sizes and the different node configurations, whereas the *y-axis* corresponds to the execution time overhead in seconds. Interestingly, the checkpoint overhead does not increase as we increase the number of nodes for the two different problem sizes. During the checkpoint each application process stores temporarily the data into the local NVMe device and the FT-managers are responsible to move the data to the GPFS, therefore the application overhead remains constant, regardless of the number of nodes. As expected, the overhead decreases as we apply our optimized methods. When we compare the first version with the initial version we obtain a $7.2X$ reduction in the overhead, whereas when we compare the second version with the initial one the overhead is reduced by $12.05X$. The same amount of reduction is observed in both problem sizes, consequently our implementation strongly scales.

Shared GPU among Processes

In *Jacobi* we test the behavior of our implementation, when multiple processes share the same GPU device. We execute two different sets of experiments, in the first set there is a 1:1 ratio between the number of processes to the number of GPUs. In other words, each GPU is used only by a specific CPU. On the second set we use a 8:1 ratio, 8 processes share the same GPU. Once more, we evaluate the weak scaling of our method, each application rank solves a local domain size of 8192×8192 elements which corresponds to per process checkpoint size of 1Gb. We execute both ratio configurations on different number of nodes. The results are depicted in Figure 5.5b. The *Y-axis* presents the overhead in seconds, whereas the *X axis* represents, the different node/GPU per process configurations. Recall, that each node consists of 4 GPUs, therefore on the 1:1 ratio we execute 4 user processes per node, whereas on the 8:1 configuration we execute 32 processes per node.

When we compare the different versions with the initial one, for the 1:1 CPU to GPU ratio, the speed up of the first version executes 12 times faster than the initial one while the second ones executes 15 times faster. In the case of the 8:1 ratio the speed up is dropped to $5.3x$ and $5.7x$ for the two versions respectively. The slow down in comparison with the 1:1 version is due to the latency of the I/O layer and not due to the latency of the transfers from the GPU device to the CPU device. The processes among the same layer share the I/O layer, increasing the amount of data stored in this layer should also increase the execution time of the procedure.

Finally, both ratio configurations correspond to the best case scenario of the version 1 implementation, as almost all the checkpoint data is stored in the GPU side, therefore the MD5 GPU algorithm is executed without any intermediate copy steps, and therefore, the computation is completely offloaded asynchronously to the GPU device and the main process spends time to execute the

streaming computations.

Strong Scaling of multi-GPU/multi-node checkpoint

In the case of *Hydro* we test the strong scaling of our method. In *Hydro* the data is almost evenly distributed among the GPU memory and the CPU. Data in the GPU device does not use UVM address space, consequently it is not accessible directly from the Host. Figure 5.5c depicts the results of our experiments. When comparing the *version 1* with the initial one, the overhead is reduced by a factor of 11x, regardless of the node configuration. However, in contrast to the previous applications, we do not observe extra benefits from the second optimization (*version 2*). This is because, *Hydro* checkpoints 50 different memory regions, except the smallest node configuration, in all the remaining cases these regions fit in intermediate pinned buffer of the first implementation. Consequently, the CPU only needs to offload the computation to the GPU and synchronize with the device at the end of checkpoint procedure. Therefore the *version 1* with *version 2* are almost identical in terms of performance.

5.1.5. Differential Checkpoint Support for GPU data

Differential checkpoint (dCP) is defined as the procedure that stores in the CR file only the data that have changed a value in comparison with the previous checkpoint. The dCP implementation in FTI is performed in the following steps, each *protected* variable is divided into chunks, for each chunk we compute a MD5 checksum[1] as described in 5.1.3.1, the checksum of the data residing currently in the memory of the system (device or host) are compared with the checksum of the data of the previous checkpoint file. If the checksums differ this data is written in the file. Writing dCP updates of a checkpoint file is supported via two different checkpoint file formats. The one called *FTIFF* is described in detail in [16] and updates the *dirty* data in-place, whereas the second one, called *dcpPosix*, always appends the *dirty* data to the end of the checkpoint file. Therefore, during a checkpoint *dcpPosix* scales linearly to the amount of data to be updated, but always increases the file size. This observation leads to prolonged recovery time, as we need to search in the file the latest correct data. On the other hand, *ftiff* is slower during the checkpoint, since it seek to the correct file position to update in place the data but does not add any extra overheads during recovery. The support for GPU data is identical to the one described in Section 5.1.2.

5.1.6. Incremental Checkpoint Support for GPU data

We define incremental checkpointing as the procedure where the data is incrementally written to the checkpoint file. This technique serves primarily to avoid overhead caused by oversaturated network channels. It can be used to store data that is produced at different timings. The partial data can be written to file the moment they are produced. A typical use case of such a mechanism is mutlithreaded applications (for example OpenMP ones) with each thread computing a subset of the entire data set. After the data is computed the same thread can incrementally add the partial data to the checkpoint file.

[1]The MD5 checksums are computed already for the integrity checksum, therefore we do not add any extra execution time overhead for the dCP

```
1 int FTI_InitICP(int id, int level, bool activate);
2 int FTI_AddVarICP( int varID );
3 int FTI_FinalizeICP();
4 int FTI_RecoverVar(int varID);
```

Listing 5.1. API for incremental checkpoint.

The API of FTI to perform incremental checkpoint is presented in Listing 5.1. After protecting the various memory spaces (device or host), the user can start the incremental checkpoint procedure using the *FTI_InitICP* function call. From the point on during the execution time of the application the user can update data of the checkpoint file using *FTI_AddVarICP*, function call. Finally, when all necessary data are updated, the user needs to call the *FTI_FinalizeICP* function call. Respectively, upon recovery the user can use the *FTI_RecoverVar* function to recover specific variables from the checkpoint file.

5.1.6.1. Fault tolerance design for the Nanos++ runtime system

The support for fault tolerance in the Nanos++ runtime system will be based on the differential and incremental checkpoint methodologies of the FTI library and the execution workflow of the Nanos++. During the application execution the runtime will exploit the execution workflow of the application and will inform FTI about the memory regions that have changed their values. In other words, when the runtime system identifies a consistent state to checkpoint the application, it will perform the following steps:

1. Call *FTI_InitICP* from all the nodes,
2. Update the checkpoint file using the *FTI_AddVarICP* all the memory locations that have already changed their values. This can be tracked by the runtime system using the information of the task creation arguments (*out* or *inout*). Noticeably, memory regions that are used only as *in* arguments, will be pushed to the checkpoint file only during the first checkpoint.
3. Add to the checkpoint file the state of the runtime itself.
4. Call the *FTI_FinalizeICP* function to terminate the checkpoint procedure.

In Nanos the tasks operate on data copies, and not the original data, the original data is updated by explicitly copy-in, and copy-out memory regions. We will experiment with this feature to identify opportunities to overlap the checkpoint of the application with the execution of the application.

5.2. FPGA Unvervolting

Aggressive undervolting, *i.e.,* supply voltage underscaling below the nominal/default level, is an effective technique to improve the energy efficiency of the circuits; however, with the cost of the reliability issues due to the timing delay increase. In LEGaTO, we utilize this technique to improve the energy efficiency of FPGAs and also to improve their resilience. In brief, our main contributions listed below:

1. The effect of the process variation and environmental temperature on the resilience behavior of FPGA on-chip memories under aggressively low-voltage operations.

2. Experimentally modeling the voltage behavior of commercial FPGAs and the subsequent energy-resilience trade-off for a Neural Network (NN) use-case application.

3. Detailed undervolting fault characterization and efficient mitigation techniques for NN accelerator.

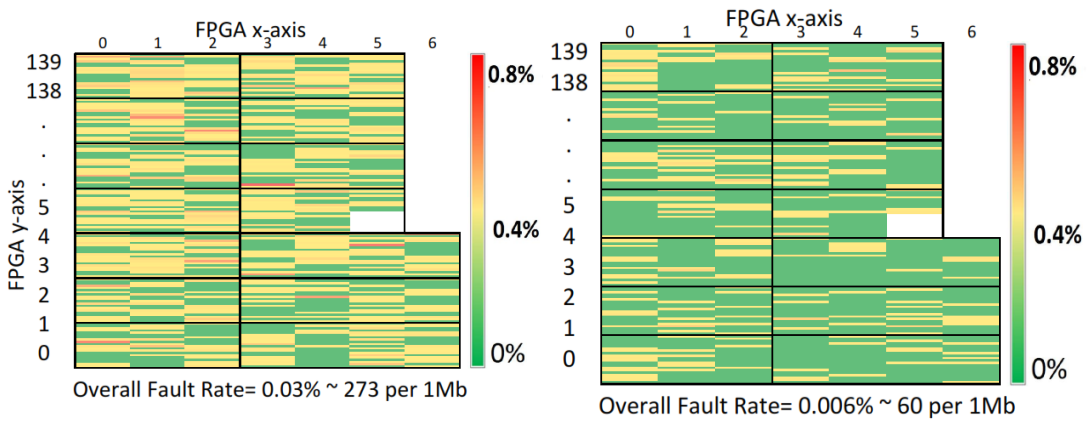
5.2.1. Introduction

FPGAs are continually obtaining more attention to accelerate NNs, thanks to the massively parallel architecture, data-flow execution model, and reconfigurability feature of FPGAs as well as the recent advances on High-Level Synthesis (HLS) tools. However, the energy efficiency of such accelerators is still a key concern, reported to be at least one order of magnitude less than customized ASIC-based models [20]. To bridge this gap, several design-, compile-, and application-level techniques can be applied to FPGAs. As an orthogonal architectural-level approach, we propose to utilize aggressive supply voltage underscaling. This technique can significantly improve the energy efficiency of the underlying hardware; however, as a downside, it may cause timing faults and in lower voltage levels, it may cause a system crash. In the NN applications, these timing faults can, in turn, degrade the accuracy. We experimentally evaluate the energy-accuracy trade-off of a typical FPGA-based NN under extremely low-voltage operations. We implement and demonstrate the low-voltage FPGA-based NN on four representative FPGAs from Xilinx, a main vendor, to consider the FPGA-to-FPGA variation in the results.

5.2.2. Experimental Methodology

We perform our undervolting experiments on several Xilinx FPGA platforms with 28nm technology, *i.e.*, VC707, ZC702, and two identical samples of KC705 (A & B), representing performance-oriented, sw/hw, and power-optimized designs, respectively. Also, note that among different FPGA components, we concentrate on on-chip memories or Block RAMs (BRAMs), since first, they play a key role in the structure of the accelerator to locate the NN weights on-chip; second, unlike other FPGA components, they have an independent voltage rail in the studied FPGA platforms, *i.e.*, V_{CCBRAM}. BRAMs are small memory blocks, which are distributed over the chip, and each basic BRAM block is a matrix of bitcells composed of rows and columns. In studied platforms, the size of each basic setup BRAM is 18 Kbits with 1024 rows and 18 columns. The default/nominal voltage of BRAMs, *i.e.*, V_{nom} is 1V for all of the studied platforms, set by the vendor. Note that for the voltage scaling, we use Power Management Bus (PMBus) standard to access V_{CCBRAM}. We underscale the supply voltage by the scale of 10mV. Finally, we report the total power consumption, including dynamic and static parts, measured using PMBus.

5.2.3. Effect of Process Variation and Environmental Temperature



(a) KC705-A. *(b) KC705-B.*

Figure 5.6. Fault Variation Maps (FVM) for two identical samples of KC705 at V_{crash}. Totally different fault rates and fault locations (FVM) are experimentally observed.

5.2.3.1. Process Variation

We perform an analysis to understand the effect of the voltage underscaling on two identical samples of the same platform, *i.e.* KC705-A and KC705-B. This experiment can show the impact of the die-to-die process variation. We observed that KC705-A shows a 4X higher fault rate. Furthermore, by extracting their Fault Variation Maps (FVMs), *i.e.*, per-BRAM fault rate associated with the physical location of BRAMs on the chip, we observed a significant difference in the fault map among BRAMs, see Figure 5.6 that shows FVM of these platforms at the lowest voltage level that we practically could underscale, i.e., V_{crash}. For instance, BRAM#(116,1) has high-vulnerability in KC705-A; however, it has low-vulnerability in KC705-B. The consequence is that we observed a significant impact of the die-to-die process variation in the reliability behavior of FPGA BRAMs under aggressively reduced voltage levels.

5.2.3.2. Environmental Temperature

We perform an experiment to study the effect of the environmental temperature on the behavior of faults when V_{CCBRAM} is lowered below the minimum safe voltage level, i.e, V_{min}. Note that V_{min} is observed to be significantly lower than the nominal level, on average of 39% for all platforms. Toward this goal, we place the FPGA platforms inside a heat chamber where we regulate the temperature using a heater. We monitor the on-board temperature using PMBus that reads an on-board temperature sensor. BRAMs fault rates are extracted and shown in Figure 5.7 under the on-board temperatures of 50°C (default temperature), 60°C, 70°C, and 80°C. As can be seen, with heating up, the fault rate constantly reduces; for instance, by more than 3X in VC707, with the temperature increasing from 50°C to 80°C. This phenomenon is the consequence of the Inverse Thermal Independence (ITD) property [18]. ITD is a thermal property of digital devices with nano-scale technology nodes; and states that under ultra low-voltage operations, the circuit delay reduces at higher temperatures. The reason is that as the technology node scales down, the supply voltage approaches the threshold voltage. Hence, at low-voltage regimes, increasing the temperature reduces the threshold voltage and allows the device to switch faster. In turn, with the circuit

delay decreasing, the number of critical paths, and subsequently, the fault rate reduces. This property is experimentally verified in our case, for commercial FPGAs. Also, as can be seen, the fault rate in VC707 is reduced more aggressively than KC705-A. For instance, the relatively 156% more fault rate in 50°C is reduced to 11.6% less fault rate in 80°C, for VC707 vs. KC705-A. This behavior can be due to the architectural and technological difference between these platforms, since their design goal is different, *i.e.,* performance (VC707) vs. power (KC705-A). Also, as seen, by heating up, the fault rate is significantly lower for VC705-B than KC705-A, as the consequence of the process variation.

5.2.4. Energy-resilience Trade-off on FPGA-based NN

We study the effect of the aggressive undervolting on a state-of-the-art NN application. Our experiments explained below, are based on a typical NN implementation on the FPGA [26].

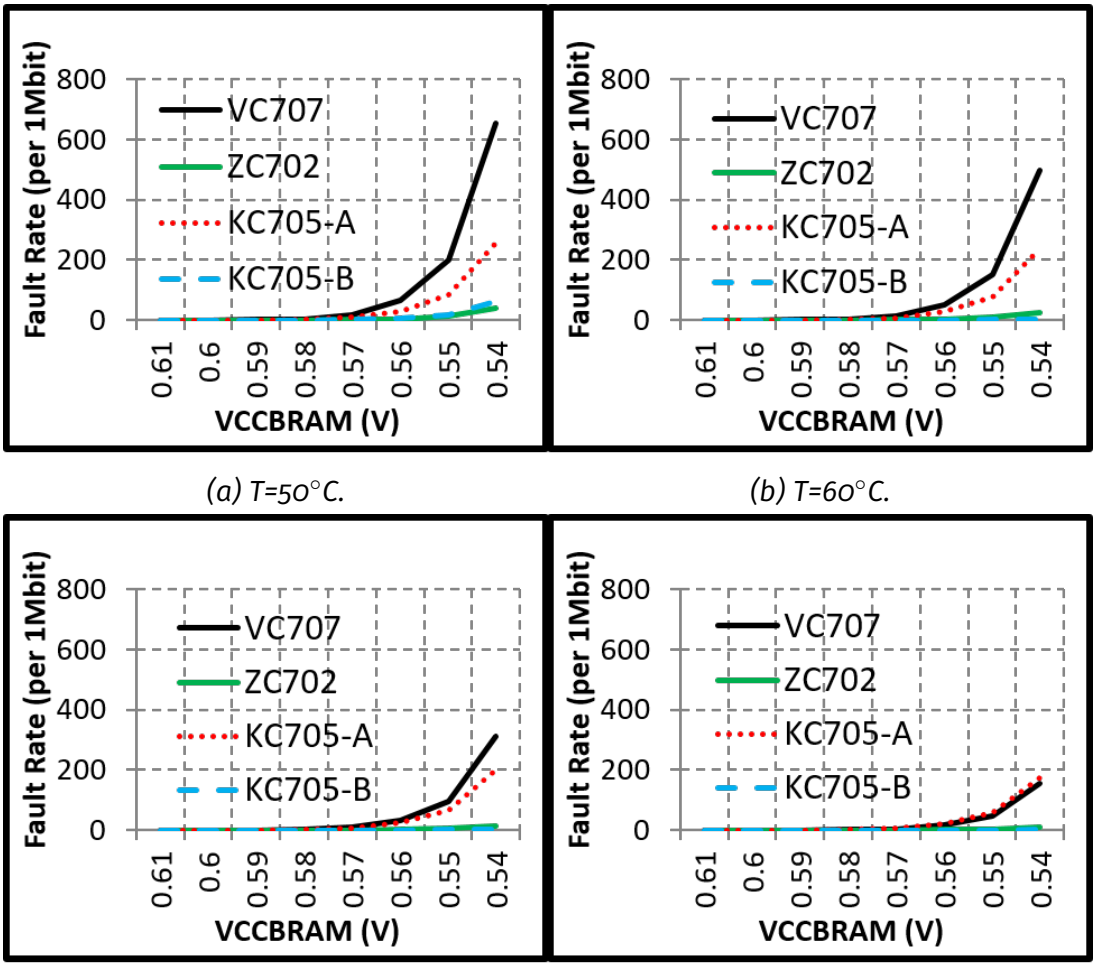
5.2.4.1. Overall Resilience Behavior

The overall voltage behavior observed is illustrated in Figure 5.8. As seen, by voltage underscaling below the default level *i.e.,* V_{nom}, there is a **Guardband** region. The voltage guardband is set by vendors to ensure the correct functionality of the device with worst-case process and environmental conditions. In this region, there is energy efficiency improvement without compromising the NN accuracy or performance since no fault appears. By further undervolting below the guardband and due to the circuit delay path increase, faults start to appear at $V_{1st-fault}$. However, until a minimum safe voltage level, *i.e.,* V_{min}, relatively low fault rate is automatically covered by NN and thus, there is no accuracy loss, *i.e.,* **Masked** region. By further voltage underscaling below V_{min}, the NN accuracy starts to being degraded, *i.e.,* **Critical** region. For instance, we observe that decreasing the voltage by 50mV leads to an NN accuracy loss of up to 3.46%. To prevent it, we propose effective fault mitigation techniques that can significantly prevent the accuracy loss up to 0.1%. Also, thanks to our fault mitigation techniques, V_{min} decreases, and thus, the NN accuracy starts to be degraded in lower voltages of up to 30mV. Finally, by further voltage underscaling, the FPGA system crashes with no response at V_{crash}, *i.e.,* **Crash** region. Note that by experimenting on several representative FPGAs, we evaluate the FPGA-to-FPGA variation and observe that the different voltages, *i.e.,* $V_{1st-fault}$, V_{min}, and V_{crash} have slight variability; however, the fault rate and in turn, the NN accuracy loss in the **Critical** region is significant, which can be the consequence of the process variation, architectural differences, or aging.

5.2.4.2. Detailed Resilience Analysis

As described earlier in Section 5.2.4.1, we observe four voltage regions in the accelerator. Among studied platforms, there is a slight variability in the size of these regions, as detailed in Figure 5.9. However, as seen, the fault rate in the **Critical** region and the subsequent impact on the NN accuracy is significantly different among the studied platforms. This variability can be the result of the process variation, aging, or architectural differences among them. Below, we describe them in detail:

1. **Guardband Region:** By voltage underscaling of V_{CCBRAM} below V_{nom}= 1V,



(a) T=50°C.

(b) T=60°C.

(c) T=70°C.

(d) T=80°C.

Figure 5.7. The correlation among on-board temperature, supply voltage, architectural technology, and fault rate for FPGA BRAMs. x-axis represents V_{CCBRAM} from V_{min} to V_{crash} and y-axis shows the fault rate per 1Mbit.

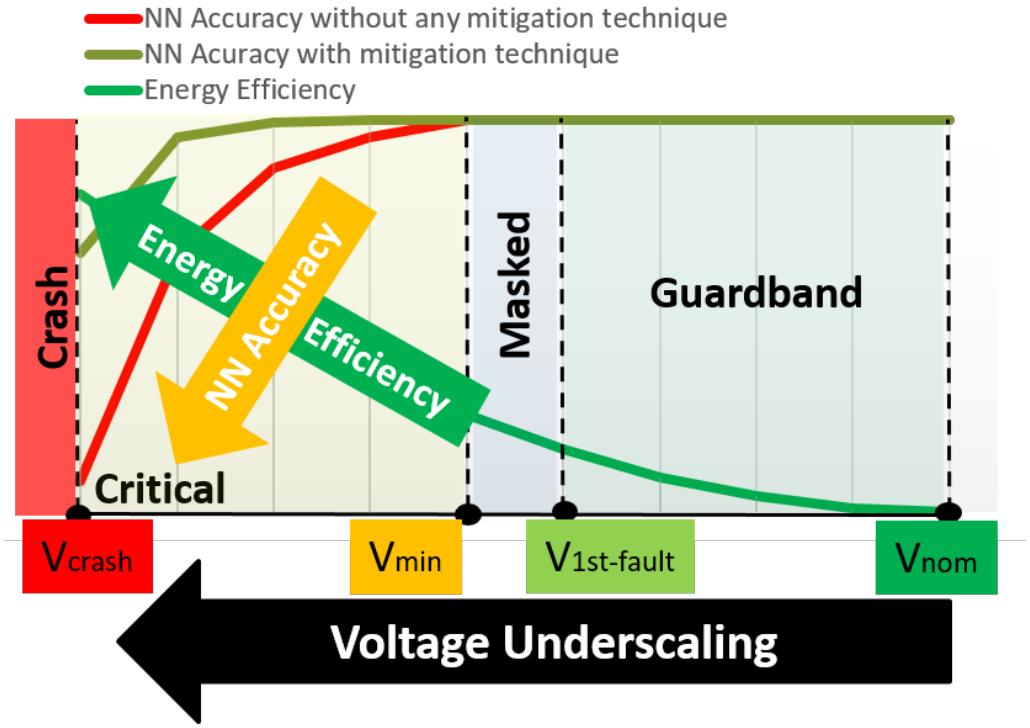


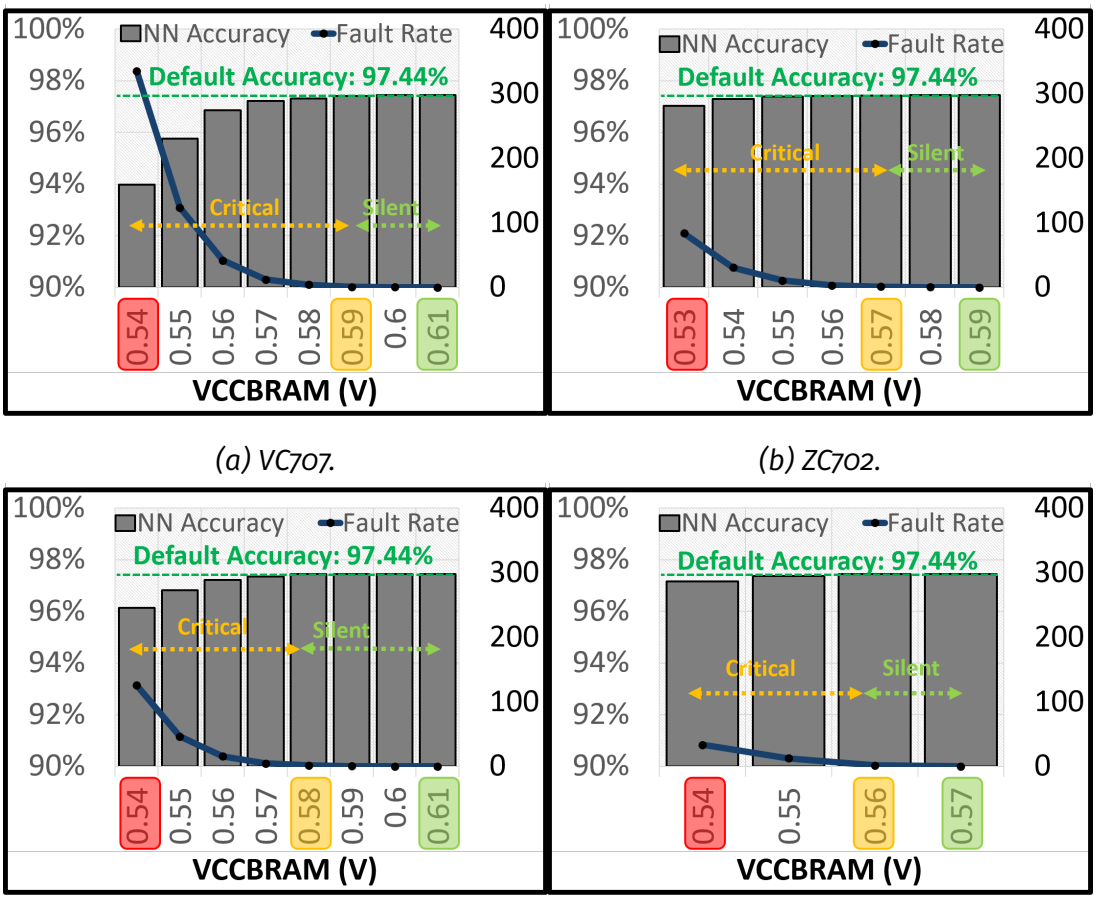
Figure 5.8. The overall energy/accuracy trade-off of the FPGA-based NN.
V_{nom}: The default voltage level.
$V_{1st-fault}$: The voltage level that the first fault appears.
V_{min}: Below this voltage level there is NN accuracy loss.
V_{crash}: Below this voltage level FPGA crashes.

we observe a large voltage guardband in [V_{nom} and $V_{1st-fault}$) for all platforms. The size of the Guardband area is measured to be 405mV on average. Guardbands are usually set by vendors to guarantee the worst-case process and environmental conditions. In this voltage region, there is no fault in BRAMs, and subsequently, there is no NN accuracy loss.

2. **Masked Region:** By further voltage underscaling below $V_{1st-fault}$ and until V_{min}, faults start to appear in BRAMs; however, the NN accuracy is not affected, which can be due to the inherent robustness of the NN for low fault rates. In other words, faults in this region are masked by the NN. The size of this area is measured to be 20mV on average. We observe that the accelerator is inherently robust to 1.4 faults/Mbit that occurs at V_{min}= 575mV, on average for all studied platforms.

3. **Critical Region:** By further voltage underscaling below V_{min}, the fault rate exponentially increases and subsequently, the NN starts to lose the accuracy. As shown in Figure 5.9, there is a significant variation of the fault rate and thus, accuracy loss among platforms. For instance, as the best/-worst platform, the voltage underscaling from V_{min}= 0.59V/0.56V to V_{crash}= 0.54V/0.54V in VC707/KC705-B causes 334.7/36.9 faults per 1Mbit and 3.46%/0.28% NN accuracy loss. To prevent this accuracy loss, the accelerator is equipped with effective mitigation techniques that are discussed in Section 5.2.5.

4. **Crash Region:** Finally, the system crashes below the V_{crash}, and there is no response from FPGA platforms. V_{crash} is the lowest voltage level that we



(a) VC707. *(b) ZC702.*

(c) KC705-A. *(d) KC705-B.*

*Figure 5.9. Resilience behavior of the accelerator on four studied FPGAs (x-axis: V_{CCBRAM} (V), y-axisL: NN inference error rate (percentage), y-axisR: BRAMs fault rate (per 1Mb), shown for **Masked** $[V_{1st-fault}, V_{min})$ and **Critical** $[V_{min}, V_{crash})$ regions.*
+ $V_{1st-fault}$, V_{min}, and V_{crash} are highlighted with different colors.
*+ Among different platforms, slight variation of the voltage regions and the subsequent significant impact on the fault rate and NN accuracy in the **Critical** region can be seen.*

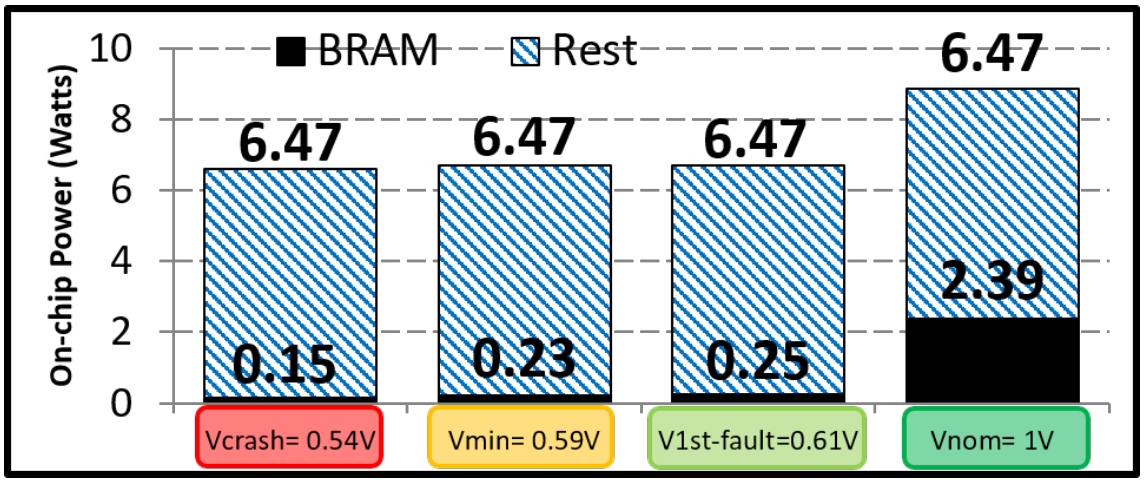


Figure 5.10. Power saving of the accelerator at different voltage regions, shown for VC707 (similar for other platforms).

could practically underscale. It is measured to be on average of 535mV with a slight variability among platforms.

5.2.4.3. Energy Saving

By voltage underscaling, the power consumption and in turn, the energy dissipation of the accelerator also gradually decrease, as shown in detail in Figure 5.10 for VC707. We achieve an average of more than 90% of BRAMs power dissipation savings at V_{crash} in comparison to the same design at V_{nom}.

5.2.5. Fault Mitigation Techniques

As mentioned earlier, there is a significant accuracy loss in the accelerator when V_{CCBRAM} is underscaled below the V_{min}. Relying on the behavior of the undervolting faults, we present techniques to mitigate the undervolting faults. These techniques, *i)* prevents the NN accuracy loss, and *ii)* decreases the V_{min} where NN accuracy starts to be degraded. For instance, on VC707, our best technique can decrease the V_{min}, for 30mV; also, it can limit the NN accuracy loss to up to 0.1% at V_{crash}.

5.2.5.1. Intelligently-constraint Memory Mapping (IMM)

By characterizing the undervolting faults, we observe that the faults are **fully non-uniformly** distributed among different BRAMs. For instance, as shown in Figure 5.11 for VC707 at V_{crash}, only 1.8% of BRAMs, tagged as High-vulnerable, experience a vast majority (>90%) of faults. Keeping this point in the mind, Intelligently-constraint Memory Mapping (IMM) aims to eliminate High-vulnerable BRAMs. Toward this goal, IMM adds additional constraints for the Placement stage of the design compile using Physical Blocks (Pblocks) facility of Vivado, compile tool for Xilinx FPGAs. Note that due to a small percentage of High-vulnerable BRAMs, the timing slack overhead of the IMM is negligible. IMM shows significant efficiency to prevent the NN accuracy loss; although, faults in non-High-vulnerable BRAMs still cause some accuracy loss of up to 0.85% at V_{crash}, see Figure 5.13.

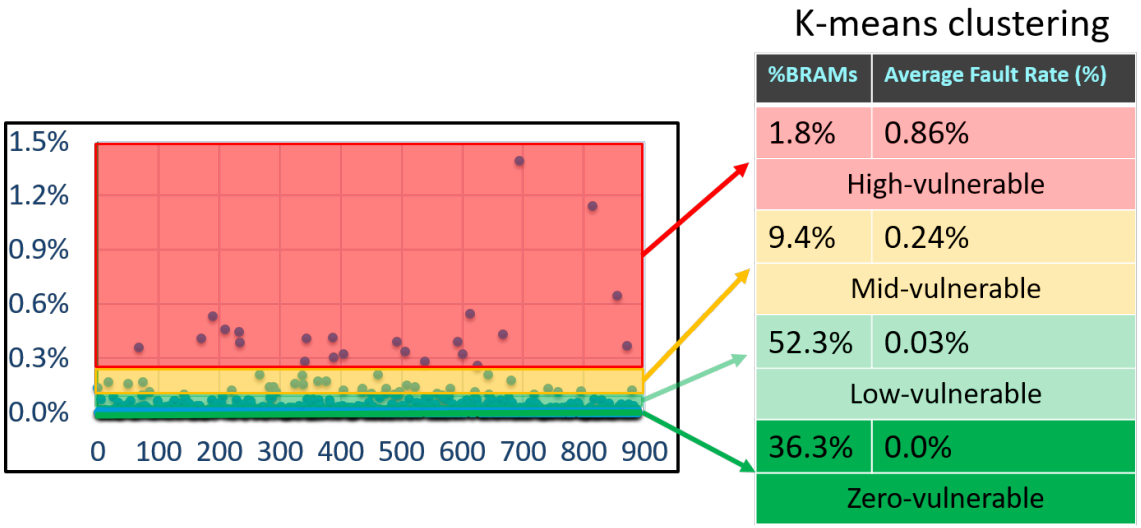


Figure 5.11. Non-uniform fault distribution among BRAMs for VC707 with 2030 BRAMs, classified using the K-means clustering in terms of the fault rate at V_{crash} (similar for other platforms).

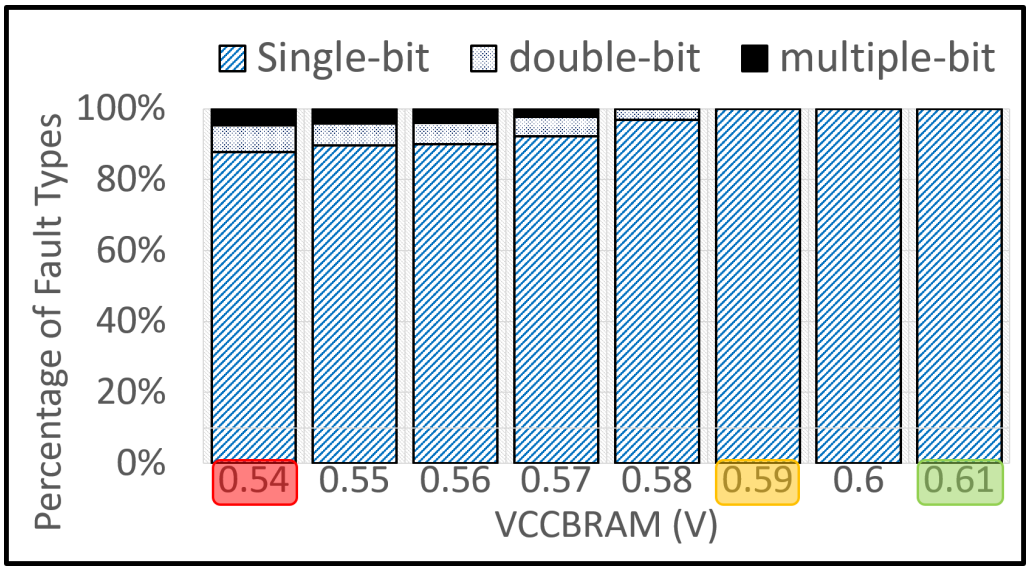


Figure 5.12. Different types of undervolting faults, shown for VC707 (similar for other platforms).

5.2.5.2. Error Correction Code (ECC)

As another fault mitigation technique, we evaluate the built-in ECC of BRAMs. It is based on Hamming code with the type of Single-Error Correction and Double-Error Detection (SECDED), which can correct single-bit faults and detect (but not correct) double-bit faults. By an off-line fault characterization, we found that a vast majority ($\sim$ 90% at V_{crash} and even more in the higher voltage levels) of undervolting faults are **single-bit**, see Figure 5.12. The built-in SECDED-type ECC of BRAMs can efficiently mitigate most of these faults. Hence, we utilize ECC of BRAMs. As can be seen in Figure 5.13, the NN accuracy loss is significantly prevented, thanks to the fault coverage by ECC. Consequently, the NN accuracy loss is significantly prevented, and by voltage underscaling until 0.57V there is

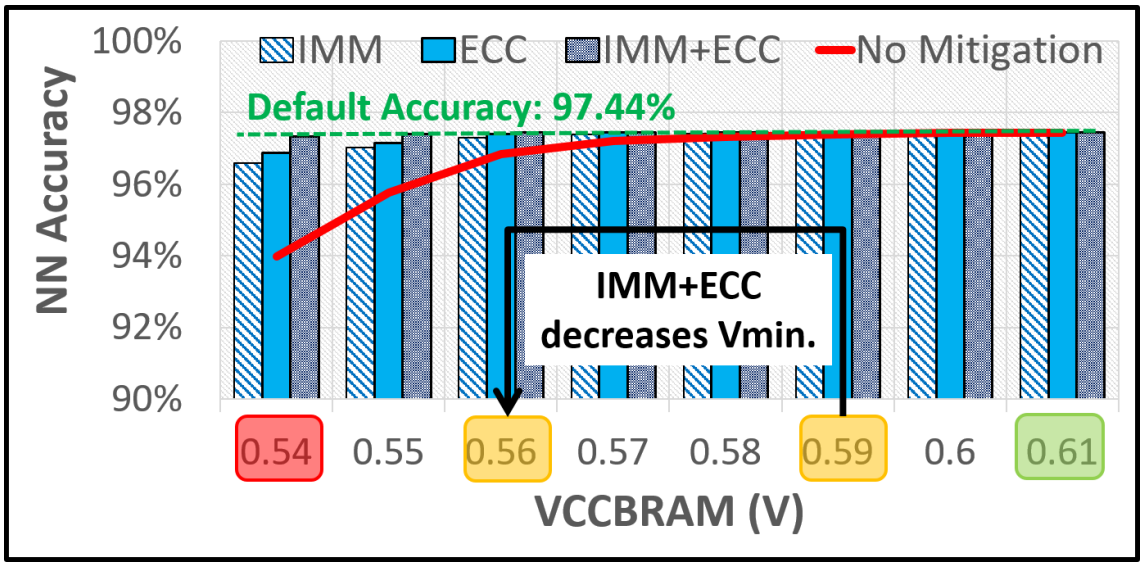


Figure 5.13. Fault mitigation in the accelerator, shown for VC707.

no effect on the NN (without any mitigation, the V_{min} is 0.59V). However, due to those faults that ECC could not correct, *i.e.*, double-bit, multiple-bit, and ECC-module corrupted faults, there is still some accuracy loss of up to 0.57% at V_{crash}.

5.2.5.3. IMM+ECC

As mentioned earlier, IMM eliminates the High-vulnerable BRAMs; however, faults remaining in other BRAMs can affect the NN accuracy, as seen in Figure 5.13. On the other side, we observed that a vast majority of faults in non-High-vulnerable BRAMs are single-bit; thus, the built-in ECC can effectively cover them. In other words, we found ECC a useful complementary for IMM technique; *i.e.*, ECC can cover those faults that are not covered by IMM. The combined IMM+ECC mitigation technique has a remarkable performance to cover the undervolting faults and in as shown in Figure 5.13, the V_{min} is decreased for 30mV (from 0.59V to 0.56V) and the NN accuracy loss is limited to up to 0.1% at V_{crash}= 0.54V on VC707, *i.e.*, the least-robust FPGA platform that we studied.

5.3. Secure Checkpointing

In the LEGaTo project, we want to achieve both goals security and fault tolerance at the same time. We protect the confidentiality and integrity of applications against strong attackers with privileged accesses using trusted execution environments (TEEs) such as Intel SGX. This approach enables applications to run inside enclaves - protected memory areas which cannot be accessed even by adversaries with a privileged/root access. In addition, to provide fault tolerance or robustness of applications, we make use of checkpoint-based mechanisms to ensure the applications can rollback to previous states in the presence of faults. However, the problem is that a checkpoint of an enclave might leak confidential data since it needs to export all data to the outside of an enclave. To handle this issue, we design and implement a secure checkpointing mechanism that protects data by ensuring that it is encrypted before sending to the outside and only an enclave with the correct MrEnclave can receive the data.

Typically, to perform a checkpoint inside an enclave, it requires to support fork()

inside the enclave. The fork system call clones the calling process/thread with the result being an identical process image at the point and time of calling the system call. In order to create a secure fork system call, there are various security and performance factors that must thoroughly be considered. These factors include security, performance, reliability, secrets sharing, and consistency. Security is the overriding factor and challenged. We need to guarantee that the state of the forking application will be replicated securely and all other operations involved must not compromise the SGX security. This requires the state of the enclave to be encrypted before being transferred. In addition, it must be authenticated, and its integrity is verified. The reliability of the fork system call is guaranteed by requiring that the state of parent enclave be consistent prior to the start of encryption and transfer of forking enclave's state. Also, the forked enclave must be in a consistent state at the time of restoration of the state of the encrypted parent state.

As mentioned above, the transfer of inconsistent state might risk leaking of secrets from the enclave. To securely share secrets during enclave creation or with secure fork child processes, a possible solution is that we establish TLS connections between enclave processes. However, a wholesome view and consideration of all the requirements to be satisfied during fork eliminate the TLS handshake protocol for the exchange of secrets. The problem is that we need to maintain consistency of state between the parent and the child at the time the child is restoring the enclave state. The use of a TLS connection implies the necessity to initialize the receiving enclave to a point where it is capable of communicating over TLS with the parent enclave. This state used for TLS communication should be maintained until all necessary communications are complete, and this violates the consistency requirements we established earlier. This is because the normal initialization scheme of the enclave will use the memory that overlaps with that of the state of the parent and will be overwritten while restoring parent state. As a result, the communication channel will break and lead to failures. Thus, we need a different approach to handle the issue. Fortunately, Intel SGX provides enclave sealing keys based on the MrEnclave of an enclave. We use this feature to provide the security guarantee and enable secrets sharing.

In addition, we need to ensure the secure checkpointing mechanism will not incur high overhead. While process creation is an expensive timely operation in comparison to the creation of threads, this operation is further impacted by the physical process of creation of enclaves. Further, to protect and transfer the enclave state, it has to be encrypted and transferred to the child enclave. This also adds significantly to the observable latency in executing the system call. To take this challenge, we make use of secure shared memory for communication between parent and child processes instead of using pipes as a state transfer mechanism used in other frameworks. Thus, our proposed secure fork() mechanism achieves much better performance compared to other frameworks[2].

6. Runtime Support for Application Development

[2] https://sconedocs.github.io/

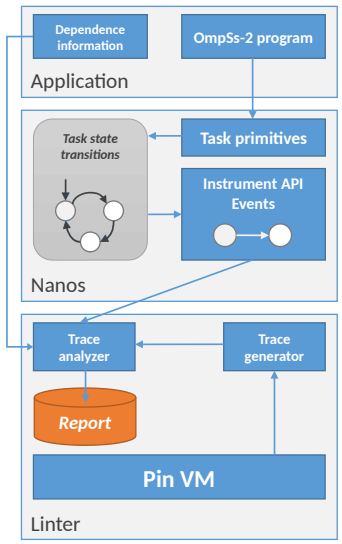


Figure 6.1. Logical flow of the OmpSs@Linter tool. The Pin VM trace (bottom) is compared with the actual dependencies observed by Nanos during runtime (top right). High level debug information (top left) is used to generate human readable feedback.

This chapter describes the tools deployed at runtime to support the development of OmpSs applications.

6.1. OmpSs@Linter as a debug tool

We have started the development of a debugging tool to trace the correctness of the task dependencies. This tool is based on the Pin instrumentation tool from Intel. Pin captures the memory accesses performed by application tasks, and our tool compares them with the task-declared input and output memory regions. Then the tool can detect mismatches among this information, and the actual memory accesses. It also detects race conditions among tasks. Figure 6.1 shows the logical flow of the OmpSs@Linter tool.

In the Figure, we can observe how applications compiled with the OmpSs Mercurium compiler have the dependence information embedded (top level). Task creation and management primitives are used by Nanos to implement the task state transitions (medium level). These are connected to the instrumentation facilities in order to generate events about task execution and data access hints provided by the programmer. This information is fed into the trace analyzer. At the same time (lower level), the binary instructions have been instrumented with Intel Pin, in such a way that a load/store data accesses are also generated and fed into the trace analyzer.

The trace analyzer compares both, the events generated about data access hints from tasks, with the actual data accesses done by tasks. As a result, the tool detects data races that occur during execution, such as:

- Missing data hints in tasks: tasks that access data not specified in depend clauses
- Unfulfilled/unnecessary data hints: data provided by the depend clauses, not actually accessed by the task

```
int z[5][10];
#pragma oss task in(z[0;3][0;4])
{
     for (int i = 0; i < 5; ++i)
          for (int j = 0; j < 10; ++j)
               z[i][j] = 2;
}
```

Supports array sections, the detection of "out-of-section" accesses, and the detection of accesses using a wrong access mode

```
[ WARNING ] [T0005 t0002] In task testcases_slides.c:39:9:
...
[ WARNING ] [T0005 t0002] WRITE  access MOV          to object <'z[[0L:2L:1L]][0L:3L]'
(0x555c9fc822a0[36:40])> in <'slides:43' (+0x3716)> DOES NOT MATCH SECTION in normal
strong dependency '   in       z[[0L:2L:1L]][0L:3L]'
[ WARNING ] [T0005 t0002] WRITE  access MOV          to object <'z[[0L:2L:1L]][0L:3L]'
(0x555c9fc822a0[40:44])> in <'slides:43' (+0x3716)> DOES NOT MATCH ACCESSMODE in normal
strong dependency '   in       z[[0L:2L:1L]][0L:3L]'
...
```

Figure 6.2. Report generated by OmpSs@linter when missing a data access hint clause.

- Missing taskwait directive: caused by read/write accesses across parent and child tasks

Current status of OmpSs@Linter implementation

We currently have a partial implementation of the OmpSs@linter tool. It detects the missing data depend clauses on tasks, and missing taskwait directives.

Figure 6.2 shows the report generated when detecting a missing depend clause, after the data has been actually accessed by the task.

Figure 6.3 shows the report generated when detecting a missing taskwait due to a data race between parent and children tasks.

In the context of the LEGaTO project, we are continuing with the development of this tool.

7. Conclusion

This document describes the first version of the LEGaTO toolchain backend (LEGaTO runtime system). The first release consists of several components: an open-stack middleware, novel backend drivers to execute on the platform FPGAs, an initial public version of the XiTAO runtime including energy-aware schedulers and platform-independent locality hints, an improved version of OmpSsFPGA, a first implementation of the OmpSs@Cluster runtime, support for GPU check-pointing, support for reliable operation of undervolted FPGAs, and novel tools supporting the development of correct OmpSs applications. Future steps in the development plan consist of integrating these components into a consolidated infrastructure. Initial steps have already been performed to enable side-by-side operation of the Nanos and XiTAO runtimes (some details are shown in Deliverable D4.2). In addition, we plan to develop back-end drivers to run the

```
int x = 10;
#pragma oss task inout(x)
{
     int y;
     x = 5;

     #pragma oss task inout(x)
          x *= 6;

     y = x + 6;
}
```

```
[ WARNING ] [T0008 t0006] In task testcases_slides.c:90:9:
[ WARNING ] [T0008 t0006] READ   access MOV          to object <'x'
(0x7fd407ffdbd4[0:4])> in <'slides:98' (+0x38de)> CAN RACE WITH dependency 'x' in child
task testcases_slides.c:95:9
```

Figure 6.3. Report generated by OmpSs@linter when missing a taskwait.

XiTAO/Nanos runtime systems on the Christmann platform. Furthermore, we plan to extend the GPU checkpointing mechanism to support also FPGAs, and enable to control of FPGA undervolting directly from the LEGaTO runtime. We will also continue the development of the OmpSs@Cluster platform targeting improved scalability, and further improve the capabilities of the OmpSs@Linter tool. These achievements will be reported in an internal release of the LEGaTO runtime scheduled for December 2019, ahead of the final release planned for May 2020.

8. References

[1] Barcelona Supercomputing Center. Performance tools. [online], 2016. http://www.bsc.es/computer-sciences/performance-tools.

[2] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2011.

[3] Gunnar Billung-Meyer. First release of hardware architecture and firmware. Technical Report D2.2, July 2019.

[4] christmann informationstechnik + medien GmbH & Co.KG. Extended Redfish API documentation. https://christmann.github.io/recs-redfish-api/index.html, 2019. (Online; Last access: 23.07.2019).

[5] Kallia Chronaki, Alejandro Rico, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Criticality-aware dynamic task scheduling for heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 329–338, New York, NY, USA, 2015. ACM.

[6] Guillaume Colin de Verdière. Hydro benchmark. Technical report.

[7] Distributed Management Task Force. Redfish Scalable Platforms Management API Specification. http://redfish.dmtf.org/schemas/DSP0266_1.1.html, 2016. (Online; Last access: 23.07.2019).

[8] Distributed Management Task Force. Redfish Schema Index | Redfish(TM) Developer Hub. http://redfish.dmtf.org/redfish/schema_index, 2017. (Online; Last access: 23.07.2019).

[9] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[10] ecma International. Introducing JSON. http://www.json.org/json-en.html. (Online; Last access: 23.07.2019).

[11] S. K. Godunov. A difference scheme for numerical solution of discontinuous solution of hydrodynamic equations. *Math. Sbornik*, 47:271–306, 1959.

[12] G. Hu, J. Ma, and B. Huang. High throughput implementation of md5 algorithm on gpu. In *Proceedings of the 4th International Conference on Ubiquitous Information Technologies Applications*, pages 1–5, Dec 2009.

[13] Intel Corp. Quartus Prime, 2017.

[14] Intel Corporation. Intel Rack Scale Design. https://www.intel.de/content/www/de/de/architecture-and-technology/rack-scale-design-overview.html. (Online; Last access: 23.07.2019).

[15] Intel Corporation. Intel Rack Scale Design Pod Manager API Specification 2.4. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design/podm-api-spec-v2-4.html, 2019. (Online; Last access: 23.07.2019).

[16] Kai Keller and Leonardo Bautista-Gomez. Application-level differential checkpointing for HPC applications with dynamic datasets. *CoRR*, abs/1906.05038, 2019.

[17] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[18] Katayoun Neshatpour, Wayne Burleson, Amin Khajeh, and Houman Homayoun. Enhancing power, performance, and energy efficiency in chip multiprocessors exploiting inverse thermal dependence. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(4):778–791, 2018.

[19] 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*, FPGA '13, pages 1–2, New York, NY, USA, 2013. ACM.

[20] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84. IEEE, 2016.

[21] OData. OData - the Best Way to REST. http://www.odata.org/. (Online; Last access: 23.07.2019).

[22] Miquel Pericas. Elastic places: an adaptive resource manager for scalable and portable performance. *ACM Transactions on Architecture and Code Optimization*, 15(2), June 2018.

[23] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016.

[24] R. Rivest. "the md5 message-digest algorithm". Technical report, April 1992.

[25] Florentino Sainz, Sergi Mateo, Vicenç Beltran, José Luis Bosque, Xavier Martorell, and Eduard Ayguadé. Leveraging ompss to exploit hardware accelerators. In *26th IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2014, Paris, France, October 22-24, 2014*, pages 112–119, 2014.

[26] Behzad Salami, Osman S Unsal, and Adrian Cristal Kestelman. Comprehensive evaluation of supply voltage underscaling in fpga on-chip memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 724–736. IEEE, 2018.

[27] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[28] H. Topcuoglu and S. Hariri and. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[29] Osman Unsal. Architecture definition and evaluation plan for legato's hardware, toolbox and applications. Technical Report SD1, August 2018.

[30] Xilinx, Inc. Vivado High-Level Synthesis, September 2017. `http://www.xilinx.com/hls`.

[31] Xilinx, Inc. Zynq ultrascale+ mpsoc overview. [online], 2017. `https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf`.

A. IPMItool command example

Getting available baseboards and power supplies:

```
ipmitool -I lanplus -H 192.168.0.100 -U admin -P admin sdr list all
Chassis          | Dynamic MC @ 20h  | ok
Baseboard 1      | Dynamic MC @ 80h  | ok
Baseboard 2      | Dynamic MC @ 82h  | ok
Baseboard 3      | Dynamic MC @ 84h  | ok
Baseboard 4      | Dynamic MC @ 86h  | ok
Baseboard 5      | Dynamic MC @ 88h  | ok
Baseboard 6      | Dynamic MC @ 8Ah  | ok
Powersupply 1    | Dynamic MC @ 50h  | ok
Powersupply 2    | Dynamic MC @ 52h  | ok
```

Figure A.1. IPMItool command to list available baseboards and power supplies

Getting available nodes on baseboard 6:

```
ipmitool -I lanplus -H 192.168.0.100 -U admin -P admin -t 0x8a sdr list all
Baseboard 6      | Dynamic MC @ 8Ah  | ok
Node 2           | Dynamic MC @ 72h  | ok
Node 3           | Dynamic MC @ 74h  | ok
Node 4           | Dynamic MC @ 76h  | ok
```

Figure A.2. IPMItool command to list available nodes on baseboard 6

Getting sensors of node 3 on baseboard 6:

```
ipmitool -I lanplus -H 192.168.0.100 -U admin -P admin -t 0x74 -T 0x8a sdr list all
+12 V            | 11.90 Volts       | ok
Node power       | 0 Watts           | ok
Inlet temp.      | 20 degrees C      | ok
Outlet temp.     | 20 degrees C      | ok
Node 3           | Dynamic MC @ 74h  | ok
```

Figure A.3. IPMItool command to get sensors of node 3 on baseboard 6

Getting power status of node 3 on baseboard 6:

```
ipmitool -I lanplus -H 192.168.0.100 -U admin -P admin -t 0x74 -T 0x8a chassis power
    status
Chassis Power is off
```

Figure A.4. IPMItool command to get power status of node 3 on baseboard 6

Turning node 3 on baseboard 6 on:

```
ipmitool -I lanplus -H 192.168.0.100 -U admin -P admin -t 0x74 -T 0x8a chassis power on
Chassis Power Control: Up/On
```

Figure A.5. IPMItool command to turn on node 3 on baseboard 6