LEGaTO

Low Energy Toolset for
Heterogeneous Computing

D3.3 "FINAL RELEASE OF THE TASK-BASED RUNTIME"

Version 1.0

Document Information

Contract Number	780681
Project Website	https://legato-project.eu/
Contractual Deadline	31 May 2020
Dissemination Level	Public
Nature	Report
Author	Miquel Pericàs (CHALMERS)
Contributors	Do Le Quoc (TUD), Xavier Martorell (BSC), Leonardo Bautista-Gomez (BSC), Behzad Salami (BSC), Mustafa Abduljabbar (CHALMERS), Gunnar Billung-Meyer (CHR), Omar Shaaban Ibrahim (BSC), Jimmy Aguilar Mena (BSC), Paul Carpenter (BSC), Tobias Becker (MAX)
Reviewers	Micha vor dem Berge (CHR)

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
1239	2020-04-01, File created
1245	2020-04-12, Deliverable outline
1276	2020-04-28, Middleware
1280	2020-05-03, FPGA Undervolting
1292	2020-05-04, XiTAO energy-aware scheduler, topologies, pipeline parallelism and extrae support
1306	2020-05-08, Trusted Key Management
1312	2020-05-11, FPGA Checkpointing
1313	2020-05-11, OmpSs@Cluster
1317	2020-05-12, GPU Checkpointing
1324	2020-05-13, OmpS@FPGA, OmpSs@linter, backend drivers
1332	2020-05-18, Conclusion
1343	2020-05-21, CHR reviews applied
1364	2020-05-28, SLiC API
1367	2020-05-31, Release Candidate RC1

This log reflects actual revision numbers from SVN (version control software used).

Index

List of Figures

List of Tables

1. Executive Summary

This report describes the final release of the LEGaTO toolchain backend. It corresponds to the backend state as reached at M30. The report is organized in four sections covering:

1. The middleware and drivers to run on the LEGaTO hardware.
2. The energy-efficient runtime, with a focus on the two main components OmpSs/Nanos and XiTAO.
3. The runtime support for fault tolerance and security, focusing on fault tolerance for GPU/FPGA (via checkpointing), FPGA (undervolting reliability), and CPU security (trusted key management).
4. The runtime tools that have been introduced to support the development of applications written using with the LEGaTO toolchain.

This deliverable D3.3 extends and supersedes deliverable D3.2 "First release of the task-based runtime". To avoid overlap between the two documents, we choose to highlight only those components that are novel or have been considerably updated since D3.2. Overall, this deliverable covers work that has been done during the past 10 months, from M20 until M30. Throughout the deliverable, we attempt to highlight how this "final release" deliverable differs from D3.2 (the "first release").

This final runtime release includes many highlights. Dynamic node composition is now supported via the Redfish API and web GUI of the RECS|Box management software. At the runtime layer, the release includes a more advanced implementation of the XiTAO runtime, featuring (1) performance, interference and energy-aware scheduling, (2) virtual topologies for locality-aware scheduling, and (3) novel support for pipeline parallelism. The release also features advanced support for executing OmpSs applications on cluster hardware (OmpSs@Cluster), and improvements to OmpSs@FPGA targeting novel FPGA hardware, such as support for scalar operands, and instrumentation for performance analysis. To support program development, the OmpSs@Linter tool has been considerably extended, and the XiTAO runtime is now integrated with the Extrae/Paraver toolchain from BSC. Finally, this final release also features support for advanced fault tolerance in the form of high performance GPU/FPGA checkpointing, support for reliable and energy efficient FPGA undervolting, and trusted key management.

The various components of the LEGaTO backend and how they related to the major four goals of LEGaTO (energy efficiency, fault tolerance, security, and programmability) are shown in Figure 1.1.

Several components developed in the LEGaTO work package 3 "Tool-Chain Back-End" tasks (mainly in T3.4 "Energy-efficient Task-based Runtime") are tightly coupled with the toolchain front-end developed in work package 4. The description of the runtime components developed by Maxeler (MaxJ), Technion (DFiant) and University of Neuchatel (HEATS) is deferred to Deliverable D4.3 ("Final release

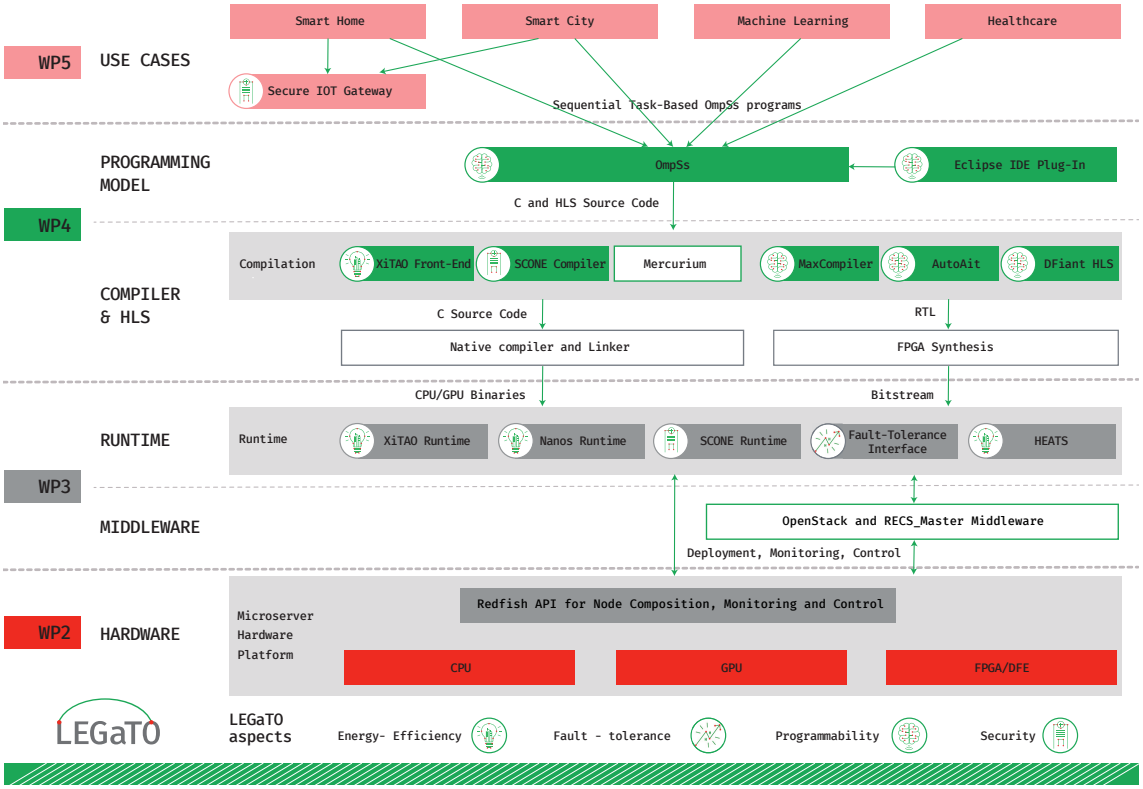


Figure 1.1. The LEGaTO stack and relation of components with LEGaTO goals

of energy-efficient, secure, resilient task-based programming model and compiler extensions, including FPGA toolchain"). D4.3 describes these components together with the rest of the front-end toolchain infrastructure.

2. Introduction

Optimizing application execution to utilize heterogeneous systems composed of asymmetric cores, FPGAs and GPUs is instrumental to reach the levels of energy efficiency demanded by next generation IoT, Edge and HPC applications. Over the past 2.5 years, the LEGaTO project has been building a toolchain to map applications written in the OmpSs language onto two heterogeneous platforms provided by Christmann and Maxeler. This deliverable (D3.3) describes the final LEGaTO release of the runtime system that has been developed to support LEGaTO applications via intelligent execution-time mechanisms. The main goal of the runtime stack developed is energy efficiency, with additional goals being productivity, fault tolerance and security.

This deliverable extends and supersedes Deliverable D3.2 which was submitted at M20. The focus of the deliverable is on novel and/or heavily updated components since D3.2. The application development and compilation aspects of the LEGaTO toolchain are covered in the sibling deliverable D4.3.

In order to achieve the targeted improvement of 10× energy reduction, LEGaTO's runtime work package (WP3) has been researching scheduling and locality-awareness techniques, the offloading of computations to FPGAs, and the undervolting of FPGAs. The main goal 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 and UBI can be statically configured, and dynamically queried and and configured by the runtime via the Redfish API and an optional OpenStack layer. The Redfish API is described in Section 3.1, along with the Web GUI designed to configure the RECS hardware which is described in 3.2. The work to target the RECS platform from the runtime layer is described in Section 3.3. On the other hand, the specific components to target the Maxeler DFE platform are described in Section 3.4.

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 the hardware topology at runtime. The operation of this scheme is described in Section 4.1.1 with a focus on virtual places mapping, a novel feature introduced in this release.

Several scheduling techniques have been researched 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 how to exploit moldability and task criticality to achieve reduced energy consumption is described in Section 4.1.2. We also describe a novel implementation of pipeline parallelism within XiTAO in Section 4.1.3.

FPGAs are becoming popular in HPC and in the datacenter as a way to accelerate applications with high energy efficiency. In LEGaTO, we have researched 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 have researched how to execute OmpSs applications on multiple nodes with distributed memory. This approach, called OmpSs@Cluster, has been released as part of OmpSs-2[1]. 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). Previously, checkpointing has been extensively researched in the context of CPUs. The LEGaTO project has researched how to extend this support to heterogeneous architectures including FPGA and GPU. Section 5.1 details our work on automatically checkpointing applications running on GPUs using the FTI checkpointing library. This D3.3 deliverable also includes a description of the novel FPGA checkpointing support in Section 5.2

[1] https://pm.bsc.es/ompss-2

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 that have been researched in LEGaTO. The results of this research are described in Section 5.3, with a focus on accelerators for convolutional neural networks (CNN).

Ensuring integrity and privacy is also an important goal of LEGaTO. Via LEGaTO's PALAEMON framwork it is now possible to transfer secrets in a trusted manner to applications running inside of enclaves such as Intel SGX. This work is described in Section 5.4.

Finally, we are also developing runtime components to support the programming of LEGaTO applications. The main focus is the tool 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. This development is described in Section 6.1. This deliverable concludes with a description of novel performance debugging support integrated in XiTAO via the extrae/paraver toolchain 6.2.

3. Middleware and backend drivers

The hardware utilized in the LEGaTO project is of a very special kind. It features the usage of heterogeneous computing resources such as x86, ARM, GPUs, FPGAs and FPGA-based Dataflow Engines (DFEs) and allows to compose those resources statically and dynamically. This characteristic, called node composition, is made possible by the flexible high-speed low-latency communication infrastructure. It can be configured to connect resources with each other and can also be reconfigured during runtime to adapt to specific characteristics of computing algorithms. In addition to that, the hardware allows the dynamic distribution of PCIe functions among the connected resources. Further details and examples of node composition can be found in D2.2 [7].

The middleware layer of the LEGaTO software stack is developed within this project to handle those complex characteristics of the underlying hardware. It enables the upper software layers to interact with it and make use of its flexibility to enhance the efficiency of computation algorithms.

As already stated in chapter 3 of D3.2 [36], the focus of the middleware has been shifted from OpenStack development to the extension of the firmware, Redfish API and Web GUI embedded in the RECS|Box server itself. Those software parts were enhanced to take over the management tasks of OpenStack to allow easy and full-featured (re-)configuration of the underlying hardware, without OpenStack. Anyhow, the static part of the node composition process is still supported with OpenStack. This part was already described in section 3.2 in D3.2 [36].

3.1. Redfish API

The hardware inventory of the RECS|Box can be accessed through the Redfish API, which was described in D3.2 [36]. Its documentation can be accessed online at Github [12]. This RESTful API is implemented in the embedded RECS_Master

management software of the RECS|Box. The API makes the capabilities of the underlying hardware visible and one can navigate through it to gather all required information about the hardware model like e.g. microservers and PCIe extension cards.

Because it is an essential feature to have a picture of the overall system health status, the API was enhanced within the LEGaTO project to now also contain detailed monitoring data regarding the thermal and power state of the hardware components. The figures 3.1 and 3.2 in this section show parts of the JSON output after querying the Redfish API.

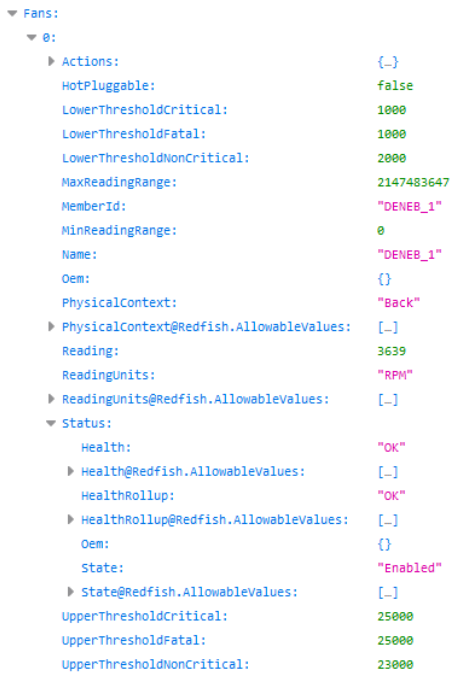


Figure 3.1. Redfish API output of chassis fan monitoring data

(a) Infrastructure power data *(b) Current data*

Figure 3.2. Redfish API output of baseboard power and current monitoring data

Furthermore, the RECS_Master management software was enhanced to expose more detailed information about PCIe functions through the Redfish API. This includes technical information such as bus, domain and instance numbers as well as vendor and device identifiers. Those identifiers, commonly known as VID

and DID, are then resolved to their names within the RECS_Master to administer additional human readable information to the user.

3.2. Web GUI

The RECS_Master management software provides not only the Redfish API but also a comprehensive Web GUI to manage the hardware. As part of the RECS|Box server management system, it was developed in previous EU-funded projects and completely reworked and enhanced within the LEGaTO project.

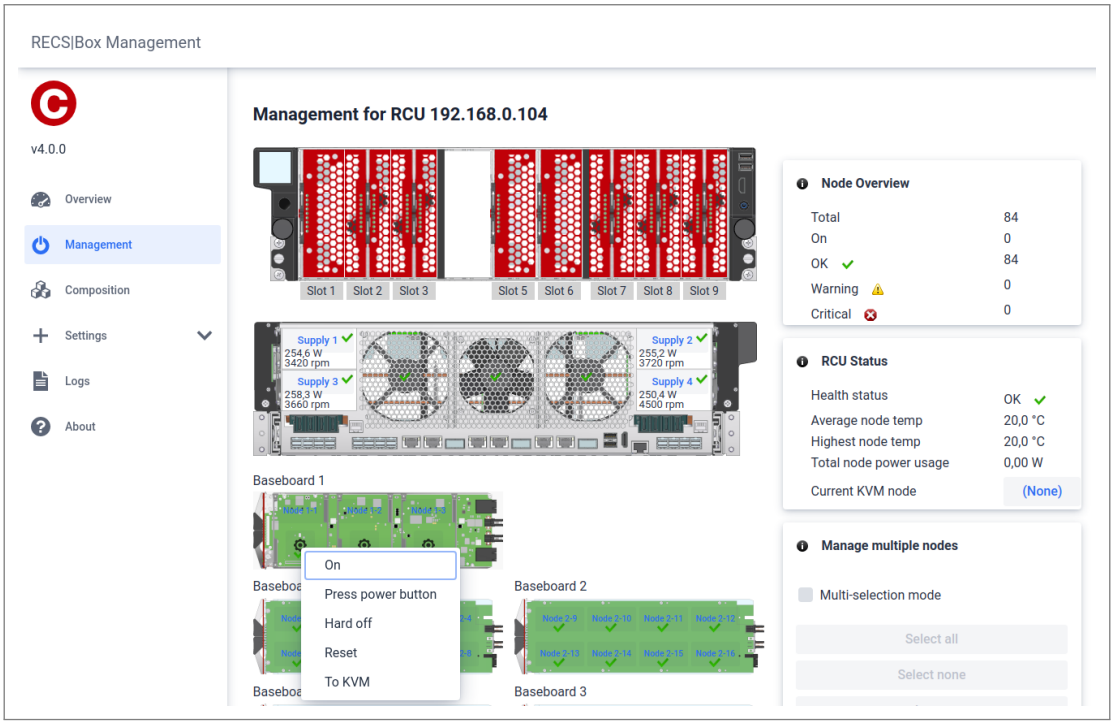


Figure 3.3. RECS|Box server Web GUI showing the management overview

Figure 3.3 shows a screenshot of the management overview of a RECS|Box server to get a picture of its capabilities.

A major progression of the Web GUI within LEGaTO was its extension to support the complete node composition process in order to have this important feature embedded within the RECS|Box system without the need for an additional OpenStack installation. Now all resources and connections between them as well as the utilization of PCIe functions can be configured easily by the user through this Web GUI. The following figures show an example node composition process with the "Node Composition Wizard" of the RECS|Box Web GUI.

After giving the new composed node a name and description in the first step (figure 3.4), the user gets a detailed list of all available microserver resources within the RECS|Box cluster server. He can now select nodes, that will be part of the composition (figure 3.5). In this example, two Intel Stratix 10 SX 2800 FPGAs from baseboard 1 and an Intel Xeon microserver from baseboard 5 are selected.

In the next step (figure 3.6) an NVIDIA Tesla V100 is selected from the available PCIe extension cards to also be part of the composed node to be created. Like the Intel Xeon node, it is connected to baseboard 5 and provides a physical and

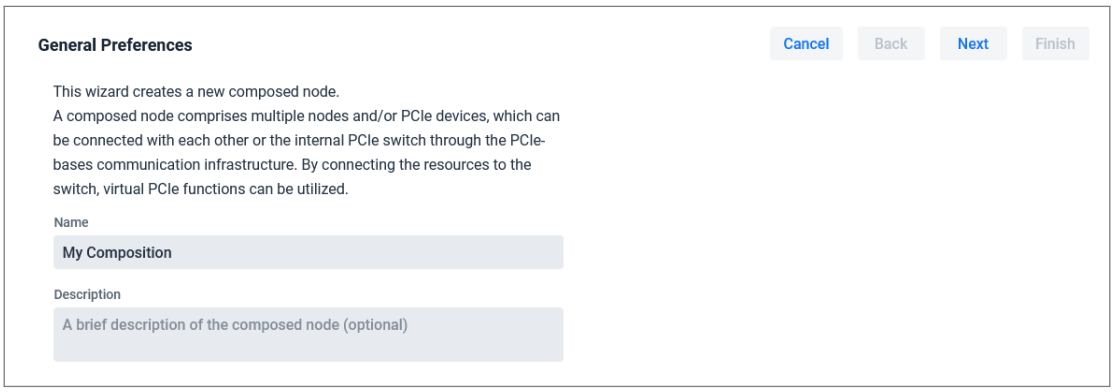


Figure 3.4. Node Composition Wizard - General preferences view

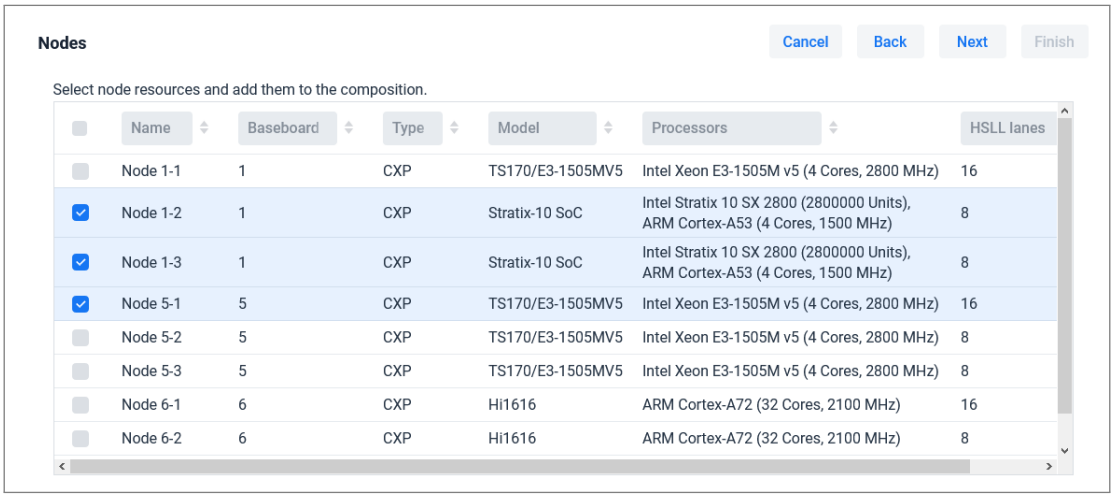


Figure 3.5. Node Composition Wizard – Node selection view

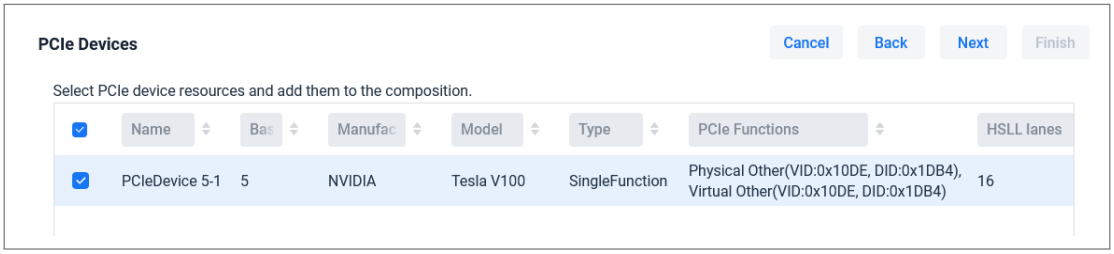


Figure 3.6. Node Composition Wizard – PCIe device selection view

a virtual PCIe function.

After that, all physical resources of the composed node to be created are selected. In the following, they will be connected and PCIe functions will be assigned.

An essential part of dynamic node composition is the reconfiguration of the physical high-speed low-latency connections and PCIe functions during runtime. To achieve this, one can now specify certain independent sets of connections between the physical resources of the composed node. In this example (figure 3.7), three connection sets named "initialisation", "algorithm 1st phase" and "algorithm 2nd phase" are defined by the user within this wizard. Only one of these sets can be active at a time and each one consists of multiple connec-

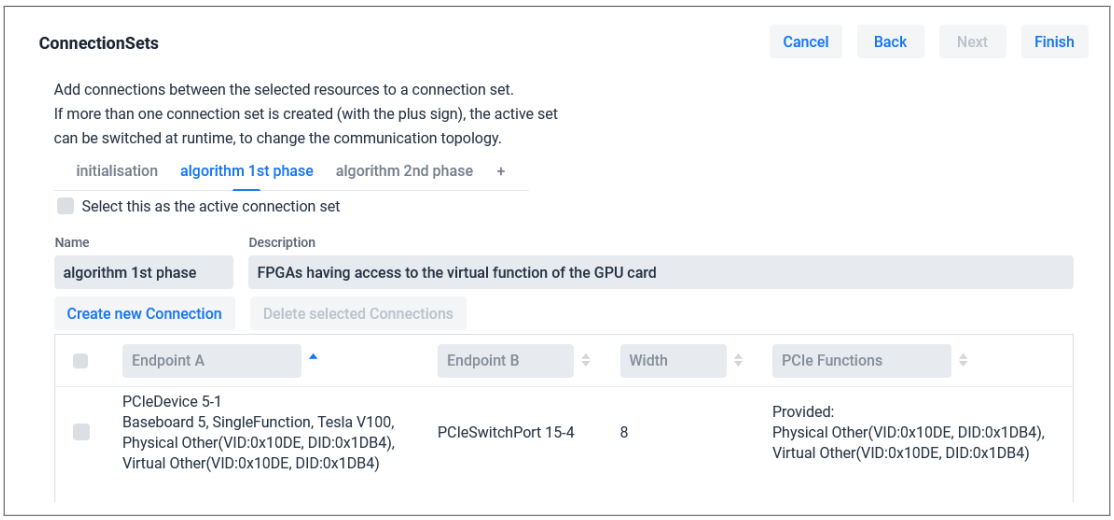


Figure 3.7. Node Composition Wizard – Connection sets view

tions between the resources selected before. During runtime, the active set can be changed by calling the Redfish API (figure 3.9 in D3.2 [36]). This then results in an immediate reconfiguration of the physical high-speed low-latency infrastrcture of the RECS|Box. If one of the connection endpoints is a port of a PCIe switch, a connection can also define one or more PCIe functions, which will be shown in the following.

In this case (figure 3.7), one connection with 8 lanes will be established between the NVIDIA Tesla V100 extension card and an embedded PCIe switch port. By doing this, the switch can then extend the single-root (SR-IOV) to multi-root virtualisation (MR-IOV) and thus, expand the virtual function of the GPU extension card to be used concurrently by multiple resources connected to the switch.

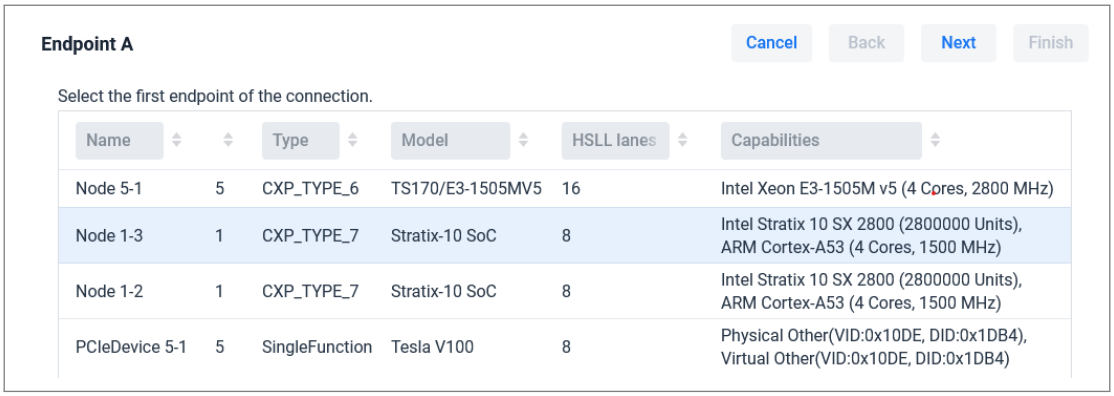


Figure 3.8. Connection Wizard – Endpoint A selection view

Now, further connections can be added to the connection set, using the virtual function provided by the NVIDIA Tesla V100 and distributed by the PCIe switch. For that matter, the "Connection Wizard" can be started to define such connection. In the first step of this wizard, the first endpoint of the connection has to be specified. In this example (figure 3.8), one of the FPGA resources is selected.

In the next step (figure 3.9), the user has to define the second endpoint of the connection. This can either be one of the other resources within the composed node or a port of PCIe switch like in our case.

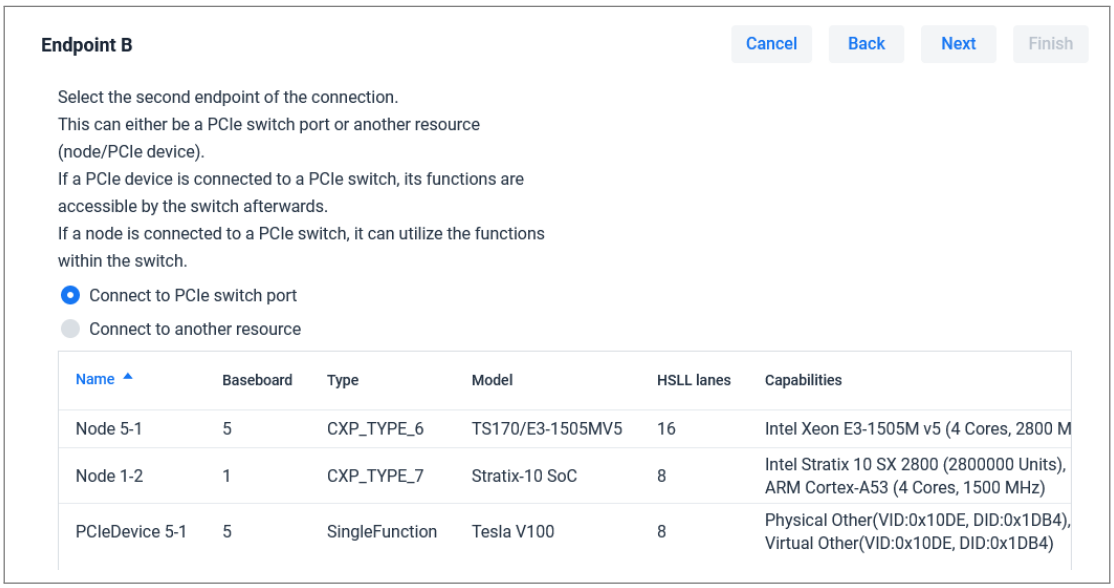


Figure 3.9. Connection Wizard – Endpoint B selection view

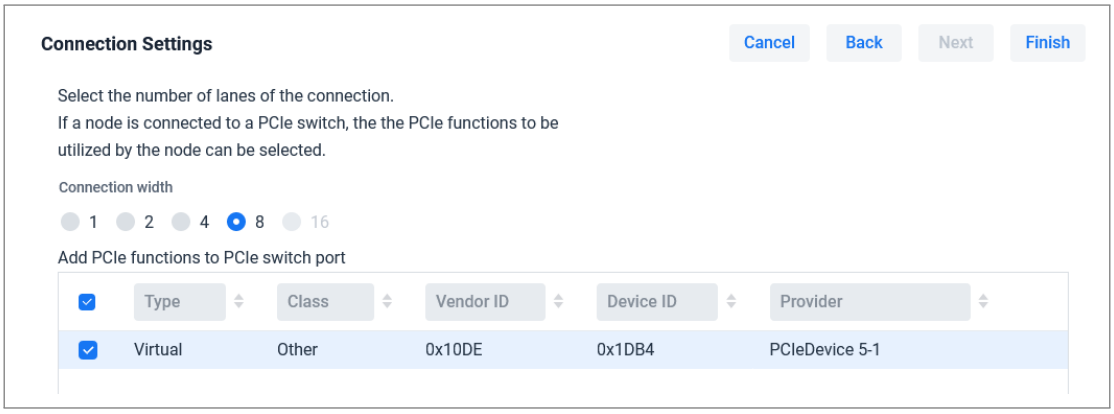


Figure 3.10. Connection Wizard – Connection settings view

By connecting the FPGA to the switch port, the number of lanes has to be specified this connection will consist of. In addition to that, all virtual PCIe functions available at the switch are listed and can be chosen from. In our case, the virtual function of the PCIe device is shown and selected. It will be added to the PCIe switch port when the connection is physically configured by activating the connection set, this connection belongs to.

After repeating this wizard with the other FPGA resource of the composed node, the connection set "algorithm 1st phase" is complete, now containing all three specified connections, which can be seen in figure 3.11.

If all connection sets are defined, the "Node Composition Wizard" can be finished. This triggers the allocation and assembling of the composed node. The underlying hardware is then configured according to the connection set marked as active (e.g. "initialisation"). After the active connection set is then switched to "algorithm 1st phase", both FPGAs will be able to concurrently use the virtual PCIe function provided by the NVIDIA Tesla V100 through the PCIe switch. When the algorithm reaches a certain point, the runtime can trigger the last connection set "algorithm 2nd phase" to have a new communication topology for a

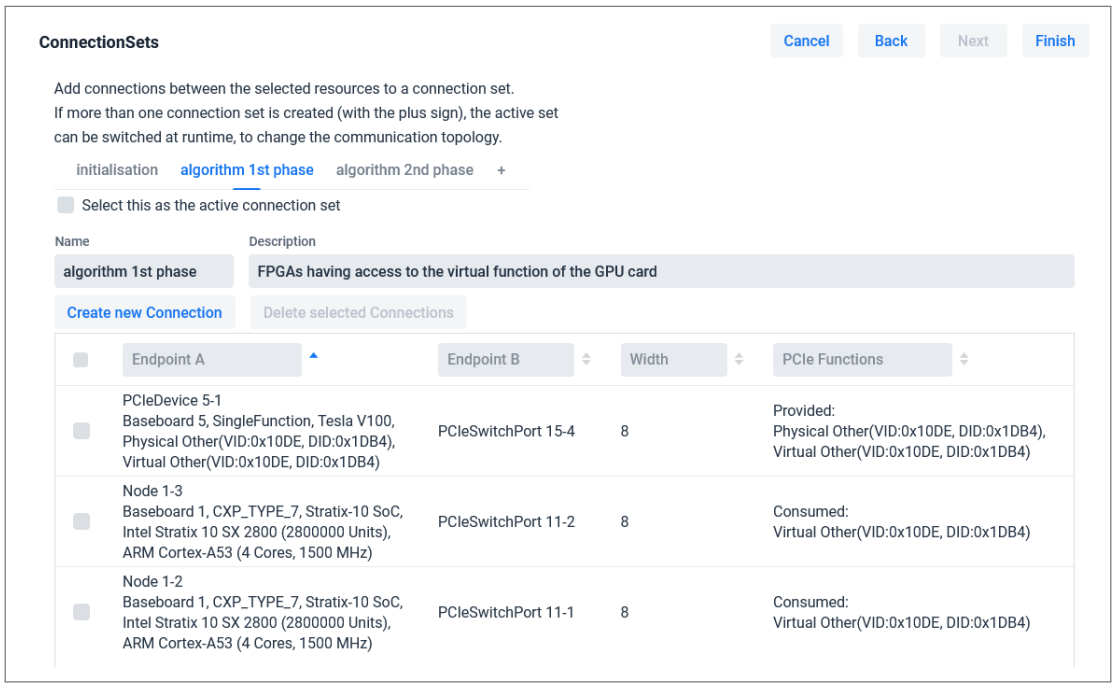


Figure 3.11. Node Composition Wizard – Connection sets view (complete)

more efficient computation in that phase.

After the computation has finished, the composed node can be deleted directly from within the runtime by calling the Redfish API or by the user utilizing the Web GUI. This deletion will roll back all configurations and frees the assigned resources. S

3.3. RECS Platform Drivers

In order to use the described node composition features from LEGaTO's runtime layer, we are developing the runtime system support for GPUs and FPGAs. For GPUs, CUDA and OpenCL already include libraries with the necessary low level services for OmpSs. 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 supported by the LEGaTO runtime (Zynq 7000, Zynq U+, and Virtex-7).

Since mid-2019, we have improved the quality of the information published by the bitstreams generated, and accessible on the Linux devices directory `/dev/ompss_fpga`. This is a summary of the information:

- `bit_info/wrapper_version`, includes the version of the wrapper code genersted for this bitstream. It is used to determine if the software side complies with the proper arguments and protocol to connect with the IP core.
- `bit_info/ait_version`, indicates that the new release of the AIT (Accelerator Integration Tool) was used to generate the bitstream instead of the former autoVivado.
- `bit_info/ait_call`, includes the command invocation and arguments

provided to AIT. As an example, the call used in the matrix multiplication example is:

```
ait.pyc --name=matmul --board=zcu102 -c=300 --hwruntime=som
--interconnection_opt=performance --to_step=bitstream
--wrapper_version=6
```

- `bit_info/xtasks`, includes the information on the generation and invocation of the IP cores. E.g., hash_key #instances name frequency

With the bitstream information available, the xtasks library checks the compatibility of the application binary generated, with the bitstream configured in the FPGA, and the it has the features required by the application.

We have also fixed an issue related to loading and unloading the kernel module several times, that was corrupting the Linux internal data structures.

3.4. SLiC API

Maxeler DFEs are accessed through the Simple Live CPU interface (SLiC) API and the MaxelerOS runtime. The SLiC API provides a shared library for host code integration. It provides the relevant function calls to interact with a DFE, such as configuring the device, setting registers, accessing memory, and streaming data. MaxelerOS provides the runtime components that include a daemon to interface with DFEs, low-level devices drivers and utilities for managing and monitoring DFEs. Both SLiC and MaxelerOS are components that are part of Maxeler's commercial toolset and they were already described in detail in Deliverable D2.1 [47].

Traditionally, Maxeler has targeted its own in-house developed DFEs with its toolset. During the LEGaTO project, Maxeler extended support to also include Xilinx Alveo U200 and U250 data center cards as compiler targets that are system-level compatible to Maxeler MAX5 DFEs. The SLiC API and MaxelerOS runtime were extended accordingly to support these devices. This expands the number of FPGA devices that are supported through the LEGaTO toolstack.

4. Energy-efficient task-based runtime

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

4.1. XiTAO

XiTAO is a lightweight layer that provides a task-parallel and data-parallel interface using modern C++ features. The design goals of XiTAO are to be low-overhead and to serve as a development platform for testing scheduling and resource management algorithms.

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). Generally, 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.

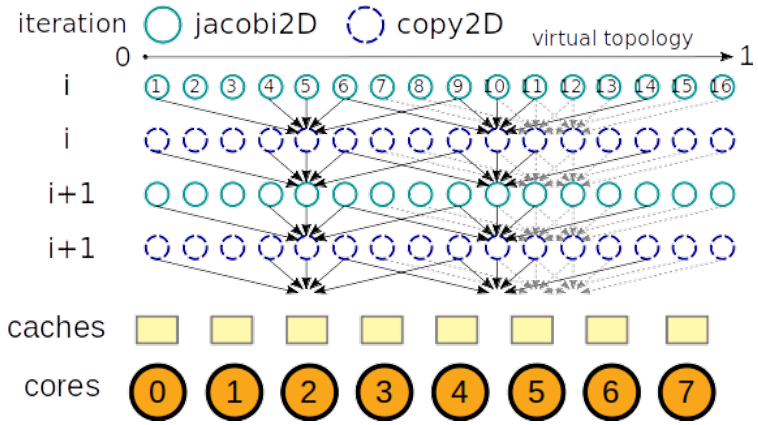


Figure 4.1. Virtual topology mapping of Jacobi2D and Copy2D kernels

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

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

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.

4.1.1.1. XiTAO Virtual Places Mapping

The granularity of the software topology mapping is managed by XiTAO at runtime, so rather than mapping to a single core, the runtime may decide to mold the task on more than one core within the specified locality. To aid this mapping, the hardware layout may be optionally passed to the runtime. Figure 4.3 shows a possible way to express an dual socket 8-core system. Width 4 has two place options: C0-C3 and C4-7. Such places are also candidates for STA mapping if the STA lies in either range. The runtime scheduler is designed to look for the optimal place for a task depending on an online performance model discussed in Section 4.1.2

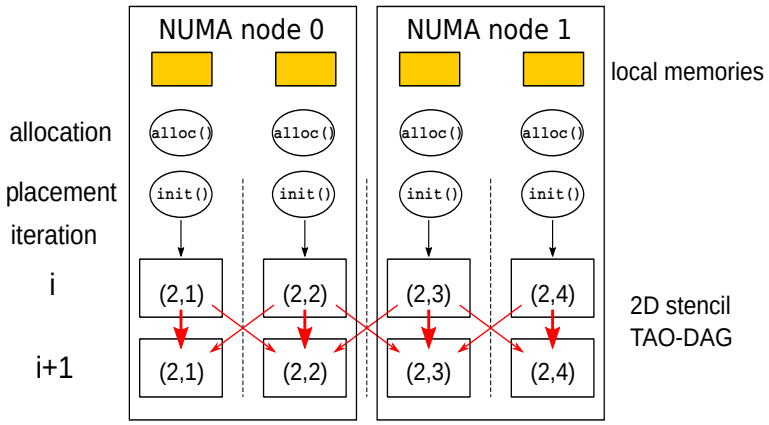


Figure 4.2. Adding an extra layer for NUMA-aware data placement

Figure 4.3. A sample view of the hardware places uncovered by the software topology address

4.1.2. The XiTAO heterogeneous scheduler

Modern multicore systems feature performance asymmetric cores to enable energy efficient execution of a variety of applications. Dynamic scheduling, such as random work stealing scheduler (RWSS), is widely used by many task-based runtime systems as the underlying scheduling strategy. RWSS greedily assigns tasks to any available resources and employs work stealing to ensure load balancing. On asymmetric platforms with more general DAGs, applying RWSS can result in suboptimal performance and inefficient utilization of resources, for example, by poorly mapping non-critical tasks to the fastest resources and vice-versa.

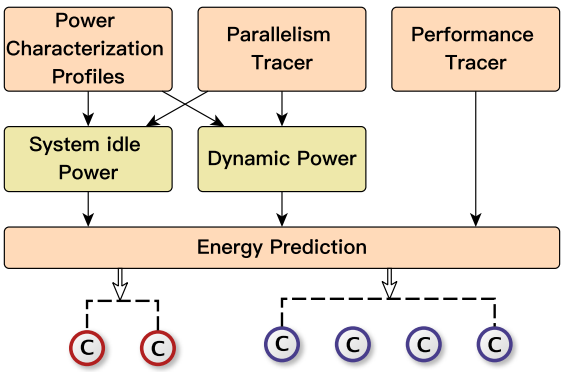


Figure 4.4. Energy efficient task scheduler runtime overview.

Some recent proposals have specifically targeted the issue of energy efficient task scheduling. However, these works suffer from several shortcomings. Firstly, they do not consider per-task power characteristics, where we show that the choice of core(s) to execute a task in an energy efficient manner is primarily affected by the kernel type, for example, compute-bound, memory-bound, cache-sensitive, etc. Our analysis also shows that runtime idle energy due to running the work stealing loop is significant, especially when the parallel slackness is low. Secondly, these works cannot handle intra-task (or nested) parallelism but only inter-task parallelism and instead only allow a task to run on a single core. Thirdly, these proposals rely on the assumption of fine-grained per-core DVFS control for achieving energy efficiency. However, most systems only feature cluster-level DVFS where DVFS settings can only be controlled for all the cores in the cluster but not individually. Another issue is the fact that modern power management monitoring and control introduces significant timing overheads. It was shown that DVFS transition time can last from tens of microseconds to over a millisecond [11]. However, with fine-grained tasks sized of microseconds level, the assumption of fine-grained per-task DVFS scheduling is unrealistic any more.

In LEGaTO, we propose an energy efficient task scheduler that reduces energy consumption by determining energy aware mappings for all tasks, which addresses the problem of scheduling of task-DAGs on asymmetric systems where frequencies are either fixed or managed by the OS power governors.

4.1.2.1. Energy Efficient Scheduler Overview

The task scheduler features several components: power characterization profiles, a performance tracer, a global parallelism tracer and a local task mapping algorithm. Figure 4.4 highlights the essential components in the proposed scheduler. These components perform the following functions: The power characterization of the platform allows the scheduler to understand CPU power consumption trends with respect to type of tasks, number/type of cores and frequencies. The performance tracer consistently tracks the history of task execution time on different cores types and counts and allows to predict the performance of future tasks given a set of resources, which is the performance trace table described in section 4.1.2 in deliveriable 3.2. The parallelism tracer gives the information of real-time parallel slackness of cores, which allows the algorithm to attribute system idle power and dynamic power to running tasks, even when some cores in the system are idle (not running any tasks). Finally, the lo-

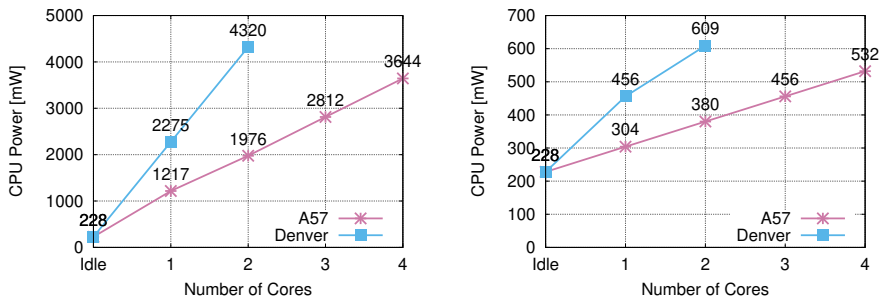
cal task mapping algorithm takes the system idle energy and per-task dynamic energy into account to fulfill the energy prediction for each task.

Power Characterization Profiles. The goal of this component is to estimate power consumption when mapping a task to an execution place. Here, we adopt offline characterization method because of the limitation of the power sensor. We consider tasks to be broadly grouped into one of the following categories: compute-bound, memory-bound and cache-sensitive. For characterization, we utilize one micro-benchmark as a representative for each category. We construct a simple power model that can provide an estimate for the power consumption of a task and is a function of execution place (leader core (LC), resource width (RW)) and task category (TC). The total power consumption (P_{tot}) of a task when running at a specific frequency (F) is given by Equation 4.1:

$$P_{tot}(LC, RW, TC, F) = P_{idl} + (P_{one} + P_{oth} \cdot (RW - 1)) \tag{4.1}$$

Here P_{idl} represents system idle power, which CPUs consume even when no application is running. P_{one} represents dynamic power consumption when using only one core in a cluster (resource width of one). P_{oth} represents the increase in dynamic power consumption when more than one core in the cluster is used. We carry out offline characterization on NVIDIA TX2 using two frequency levels: MAX (2,035,200 Hz) and MIN (345,600 Hz), although it can be easily extended to a broad range of other frequencies.

For example, Figure 4.5 shows the CPU power measurement when running the compute-bound microbenchmark with different number of cores and frequencies. When turning on all six cores while keeping them in idle state, the system idle power is found to be approximately 228 mW. We turn off the Denver cluster to measure P_{idl} of the A57 cluster, and observe it to be 152 mW. The `Tegra` Linux kernel does not permit powering off core zero (A57 in this case). Therefore, we can only turn off the Denver cluster. We infer that the idle power consumption of the Denver cluster is 76 mW (228 mW-152 mW). It is obvious that irrespective of the frequency, CPU dynamic power consumption linearly increases when additional cores from same cluster are used. Based on the measurements shown in Figure 4.5, we obtain the CPU power consumption profile for the compute-bound task category as shown in Table 4.1.



(a) MAX *(b) MIN*

Figure 4.5. CPU power consumption of compute-bound microbenchmark on two clusters in MAX and MIN frequencies.

Parallelism Tracer. Our analysis indicates that idle threads consume considerable energy throughout execution. For example, it shows that running the

Table 4.1. Power consumption profile of compute-bound tasks

Cluster (C)	Frequency (F)	P_{idl}	P_{one},P_{oth}
A57	MAX	152 mW	854 mW / core
	MIN	152 mW	76 mW / core
Denver	MAX	76 mW	2046 mW / core
	MIN	76 mW	190.5 mW / core

worker loop on the Denver core roughly consumes 60% of the power of running a compute-bound task. While on MIN frequency, the worker loop roughly consumes up to 75% of the power of running a compute-bound task. To reduce the total energy waste, the scheduler puts cores to sleep after they make several unsuccessful steal attempts. We use an exponential backoff strategy when selecting the sleep duration for idle cores. For each sleep decision, the runtime checks a threshold parameter $\mathbf{N}$, that is how many times the core has been unsuccessfully trying to steal tasks. Upon wake up, if the core finds a ready task or makes a successful steal, it resets the backoff parameter. Otherwise, it will sleep for an exponentially increasing time if it can not find ready tasks. Listing 4.1 highlights the basic implementation of this approach.

```
1  int idle_tries = 0, backoff_param = 0;
2  while(true){
3    //check local queue or try to steal
4    if(no_available_tasks){
5      idle_tries++;
6      if(idle_tries==N){
7        sleep(1<<backoff_param);
8        idle_tries = 0;
9        backoff_param++;
10     }
11   } else {
12     backoff_param = 0;
13     idle_tries = 0;
14     // execute task
15   }
16 }
```

Listing 4.1 The exponential backoff sleep approach.

Task Energy Prediction. The prediction scheme of the energy efficient task placement for a single task is shown in Algorithm 1. The inputs include *status* that stores the core states (sleeping 0 or active 1), incoming tasks. The output is the predicted optimal task placement for each task. Line 5 shows the leader core belongs to cluster A. Through accumulating the corresponding set bits, we can obtain the number of active cores per cluster (Line 6-7). If there are any active cores on cluster B, the parallel tasks running on the cluster A share total idle power of the cluster, which can be obtained by powering off cluster B (Line 8-9). Otherwise, they share the entire board's idle power since there is no active core on cluster B (Line 10-11). Line 19 presents the resource occupation of task j over all running tasks on cluster A. Line 20 demonstrates the estimation of idle power for task j. The total dynamic power consumption of all tasks running on cluster A can be obtained by plugging the values from the corresponding power profile in the formula shown in Line 21. Line 22 involves the estimation of dynamic power by task j. Finally, we could obtain the energy consumption estimation of the task on specific configuration shown in Line 23. Line 24-27 iterate each possible

configuration until getting the optimal energy efficient placement for the task.

Algorithm 1 Energy Efficient Task Placement Selection(for the case of two clusters)

```
Input: status, incoming task j
Output: optimal task placement for task j
Minimum = infinite
for each possible configuration (leader core, width) do
    Leader core belongs to clusterA
    NumActiveCores_clusterA = accumulate(status[x-y])
    NumActiveCores_clusterB = accumulate(status[y-z])
    if NumActiveCores_clusterB > 0 then
        IdleP_temp-clusterA = IdleP_tot-clusterA
    else
        IdleP_temp-clusterA = IdleP_tot
    Check if cores [leader, leader+width) are active
    for i in leader,leader+width do
        if status[i] is 0 then
            NumActiveCores_clusterA++
    Task j Resource Occupation: RO_j= width_j / NumActiveCores_clusterA
    IdleP[j] = IdleP_temp-clusterA × RO_j
    DynaP_tot = P_one + P_oth × ( NumActiveCores_clusterA - 1 )
    DynaP[j] =DynaP_tot × RO_j
    Energy = ( IdleP[j] + DynaP[j] ) × Execution Time
    if Energy < Minimum then
        Minimum = Energy
        Update optimal placement with new (leader, width)
```

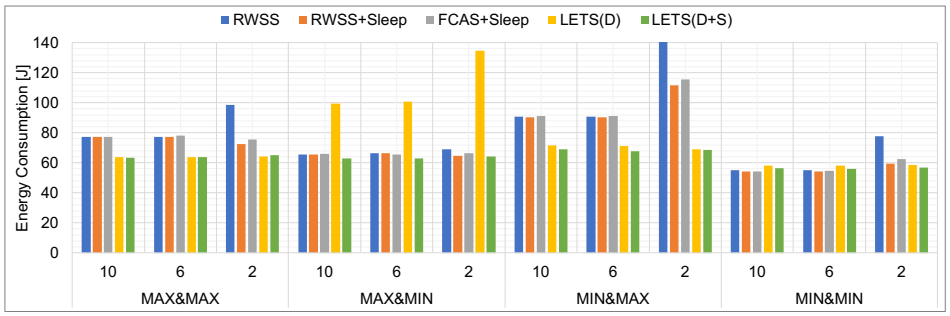
4.1.2.2. Energy Savings Evaluation

Table 4.2. Summary of evaluated schedulers.

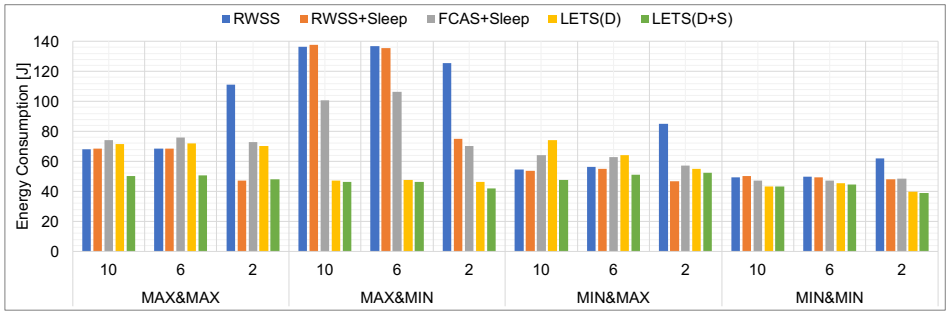
Name	Acronym	Notion
Random Work Stealing Scheduler	RWSS	Typical greedy scheduler
Random Work Stealing Scheduler with Sleep	RWSS+Sleep	RWSS enhanced with Sleep
Fast Core Always Scheduler with Sleep	FCAS+Sleep	Performance-oriented criticality scheduler enhanced with Sleep
Low Energy Task Scheduler (D)	LETS(D)	The proposed scheduler prediction only using dynamic power
Low Energy Task Scheduler (D+S)	LETS(D+S)	The proposed scheduler prediction using dynamic and system idle power

We use an NVIDIA Jetson TX2 development board for our evaluation. The board is set to `MAX-N nvpmodel mode`. The benchmarks are the same like in deliverable 3.2 section 4.1.2. All evaluated schedulers in this section are described in Table 4.2.

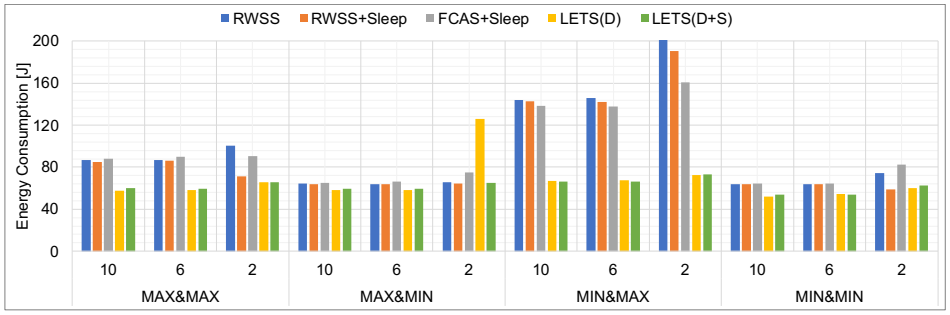
Figure 4.6 shows the total energy consumption comparison of these scheduling runtimes with or without Sleep. The results of "FCAS" without Sleep are dropped in favor of reducing clutter in the figures, as the latter delivers a relative advantage similar to "RWSS+Sleep". It is obvious that LETS(D+S) achieves the best energy savings over the other two schedulers in most of cases. Taking matrix multiplication as an example, using LETS(D+S) achieves up to approximately 25% energy savings compared to RWSS and FCAS+Sleep when the parallelism of the task DAG is high (10 and 6). With a lower parallelism of 2, LETS(D+S) achieves more energy savings which range from 6% to 60% compared to RWSS variants, and from around 4% to 40% compared to FCAS+Sleep. In the case of minimum cluster frequencies and high parallel slackness, i.e., MIN&MIN and DAG parallelism=10 or 6, the energy consumption of LETS(D+S) does not add a notable benefit (if any) since the two clusters operate at the lowest frequency, which renders the scheduling decision of no vital impact for a compute-bound kernel.



(a) Matrix Multiplication - Energy Consumption



(b) Copy - Energy Consumption



(c) Stencil - Energy Consumption

Figure 4.6. The energy consumption comparison results of three benchmarks. The x axis includes frequency combinations and task DAG parallelism. MIN&MAX means Denver is set to MIN and A57 is set to MAX, and so on.

An important fact is that the energy consumption of LETS(D+S) is relatively stable throughout all frequency combinations as opposed to others, which means that our proposed scheduler could achieve the most energy efficient task scheduling no matter what cluster frequencies are, which implies that it is agnostic of the OS DVFS events.

4.1.3. Support for Pipeline Parallelism

A significant amount of scientific applications execute the same workflow on multiple input units of same size. Such applications can be expressed as parallel pipelines by identifying the computational tasks of applications as pipeline stages. Commonly, applications expressed as a chain graphs (DAGs) are ported into parallel pipelines for efficient scheduling on multi-core computing platforms.

Pipeline paralleism is applied when a chain of tasks operate on a single input unit to produce output and there is a data dependency from task t_i to t_{i+1}. All the tasks are executed in parallel but on different input units. We implemented

pipeline parallelism on the top of the existing XiTAO's programming model.In XiTAO, application is expressed as a DAG where each node is a standalone Task Assembly Object(TAO). A TAO can be serial or parallel. Parallel TAOs contain it's on local scheduler, the mode of parallelism inside a TAO is Data parallel. Thus XiTAO has a mixed-mode programming model. We implemented pipeline stages as a self-calling TAO (making an edge to itself).

4.1.3.1. Pipeline Parallelism implemented in XiTAO

For the sake of understanding let's consider a motivating example. An application is divided into two tasks t_1 and t_2 such that, t_1 processes an input token and produce an output. The output from t_1 is fed into t_2 which further processes and provides a final output. The application works on a stream of input tokens of same size. Let's define t_1 and t_2 as TAOs in XiTAO implementation. Ideally there is an edge from t_1 to t_2 but since we are implementing a pipeline, we make these TAOs a self-calling TAO. Whenever there is a ready input for pipeline stage it should execute itself and notify the successive pipeline stage when its input is ready. The mechanism of notifying successive stages about the ready input is implemented in the form of a data structure which holds the number of input units ready to be processed by corresponding pipeline stage. The mechanism is explained in Figure 4.7.

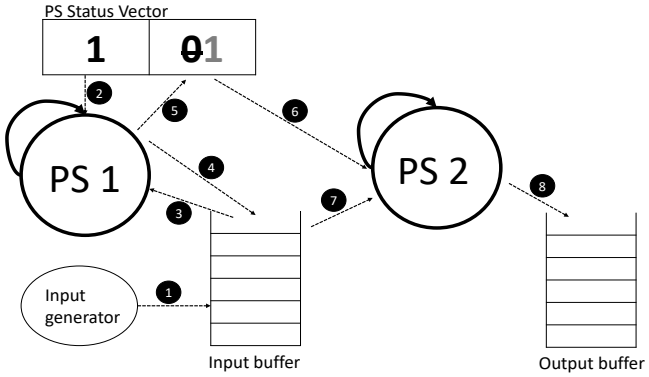


Figure 4.7. Step-by-step depiction of parallel pipeline stages in XiTAO runtime

Listing 4.2 highlight the necessary steps to implement XiTAO pipelines. Basic functionality of each pipeline stage is abstracted out from line 2 - 15. Each pipeline stage execute respective computations on input data (line 10-12) in parallel by the number of threads determined by the runtime. Once processing is done, last thread of the TAO exiting the process_stage() computations does some boilerplate work: Notifying next stage and calling same pipeline stage again for next available input unit (Line 17 - 25). Finally in $main()$, we declare pipeline stages and launch only once. Note that each pipeline stage will remain active in the system until last input unit is processed by last pipeline stage. However Pipeline moldability is achieved by changing the resource width of the TAO at every call to $xitao\_push()$.

```
#define MAX_INPUTS 100
#define NUMBER_OF_PIPELINE_STAGES 2
int PS_status[NUMBER_OF_PIPELINE_STAGES];
// Define pipeline stage
 class PipelineStage : public Assemblytask{
 public: PipelineStage(int stage_number, int input_number, int batchSize){}

  void execute(){
  if(stage_number == 1)
    process_stage1();
  else
    process_stage2();
  if(is_last_thread) // Last thread finishing the work should do the job of notifying
    notify_and_call();
  }

  void notify_and_call(){
    //work Forwarding
    if(stage_number != STAGES-1) // if it is not the last Pipeline stage
      PS_Status[stage_number+1]+=1; //activate next stage for current input unit

    //Self_calling
    input_number++ // pick next work item
    if(input_number <= batchsize)
      xitao_push(new PipelineStage(stage_number, input_number, batchSize));
  }
 }

//execuet Pipeline

void main(){
  PipelineStage PS1 = new PipelineStage(1, 1, MAX_INPUTS);
  PipelineStage PS2 = new PipelineStage(2, 1, MAX_INPUTS);
  generate_input_stream();
  PS_Status[0] = 1; //First stage is activated manually
  PS_Status[1] = -1; //second stage is in active by default

  xitao_push(PS1); // Launch Pipeline stage 1
  xitao_push(PS2); // Launch Pipeline stage 2

}
```

Listing 4.2 Pseudo code for implementing parallel pipeline stages in XiTAO

4.1.3.2. Template based Tensor-Expression Language for generating pipelined XiTAO code

For the sake of usability and expressivity, we designed a simple template based language targeting deep neural network applications. Convolutional Neural Networks (CNN) in particular consists of several convolutional layers in combination with computationally light layers such as maxpooling and softmax layers. Each layer processes on the output of previous layer and produces output for successive layer. The structure of CNNs resembles to chain graphs. Moreover inference phase of CNNs is applied on a stream of input units for example object identification in video stream. Thus CNNs are a good case for experimenting pipeline parallelism under XiTAO environment.

A three layered CNN [Conv1, Conv2, FC] is implemented using template based tensor expression language in Listing 4.3. layer parameters and input tensors are initialized from lines 4 to 26. Network structure is defined in lines 29 to 31. The last argument of each layer specifies the pointer to preceding layer. Once network structure is defined, we define pipeline stages for each layer at the back end. Sequence of pipeline stages is determined by the last argument of layer definitions. For example in line 30 conv1 is set as preceding layer of conv2. Line 34 launches the whole network which means pipeline stages corresponding to each layer are launched under the XiTAO runtime. Figure 4.8 shows preliminary results of the discussed approach executed on a homogeneous platform, a cluster of dual socket Intel Heswell nodes. We executed a 3-layer-CNN network. The

```
#include "PipelineStages.h"
#include "TensorExpressions.h"

const int batchSize = 10;
const int W  =  512;
const int H  =  512;
const int C  =  1;
const int R1 =  3;
const int S1 =  3;
const int C1 =  3;
const int K1 =  64;
const int R2 =  3;
const int S2 =  4;
const int C2 =  K1;
const int K2 =  128;
const int R  =  2;
const int M  =  1000;
const int stride = 1;
const int padding = 1;

int main() {
  Tensor<4, float> inputTensor({batchSize, H, W, C});
  auto input = TensorExpr<4, float>(inputTensor);
  auto w1 = Tensor<4, float>({R1, S1, C1, K1});
  auto w2 = Tensor<4, float>({R2, S2, C2, K2});
  auto m  = Tensor<2, float>({M, R*R});
  xitao_init();

  auto conv1 = Conv2D<float>(batchSize, R1, S1, C1, K1, H, W, C1, stride, padding, w1, input);
  auto conv2 = Conv2D<float>(batchSize, R2, S2, C2, K2, H, W, C2, stride, padding, W2, conv1);
  auto result = FC<float>(batchSize, {M, R*R}, {R*R}, m, conv2);

  xitao_start();
  result.run();
  xitao_finish();
  return 0;
}
```

Listing 4.3 Code for a small pipelined-XiTAO CNN

workload distribution is such that $PS2 > PS1 > PS3$. Since pipeline stage 2 is heavier than the other two stages, a configuration in which more cores are given to PS2 yields better execution time.

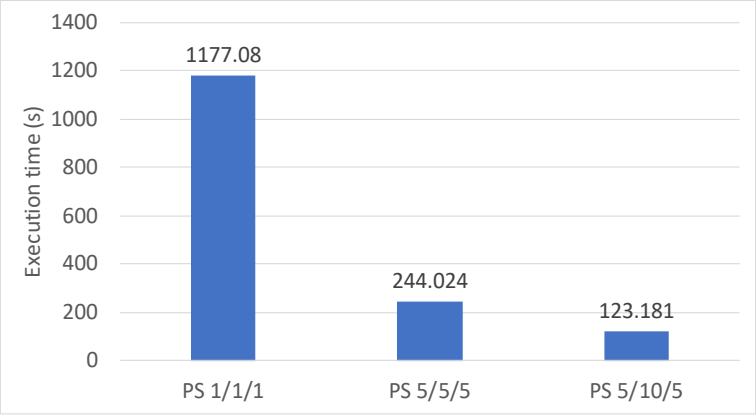


Figure 4.8. Execution time of 15 input units with different resource widths given to each pipeline stage.

4.2. OmpSs

4.2.1. OmpSs@FPGA

OmpSs@FPGA is the OmpSs programming model extension towards the support of accelerators in FPGAs. AutoVivado / AIT has gone through versions 1.4.0 to 2.0.0, and 2.2.0 in the last 9 months of the project. In this deliverable we

present OmpSs@FPGA, summarize its status presented in the previous deliverable [36], and the changes done to the runtime and compiler infrastructure related to OmpSs@FPGA in this last period of the LEGaTO project.

4.2.1.1. OmpSs@FPGA description and status by mid-2019

The OmpSs [40, 15] 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 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 [33, 50], or Altera Quartus [24]), 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.

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. On het-

```
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. OmpSs@FPGA Matrix Multiply benchmark

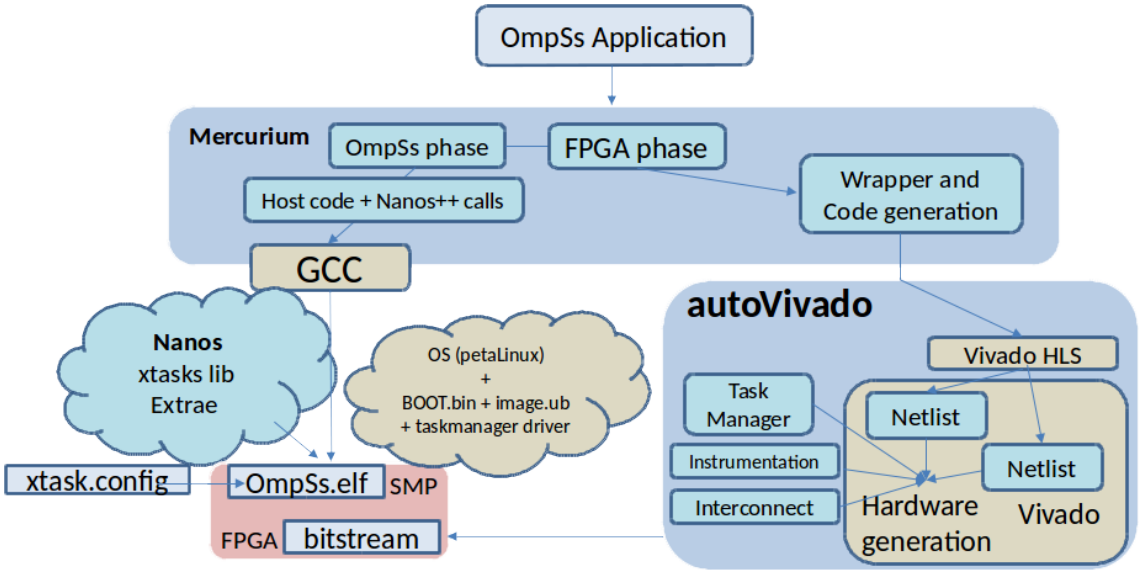


Figure 4.10. OmpSs compilation env. with FPGA support

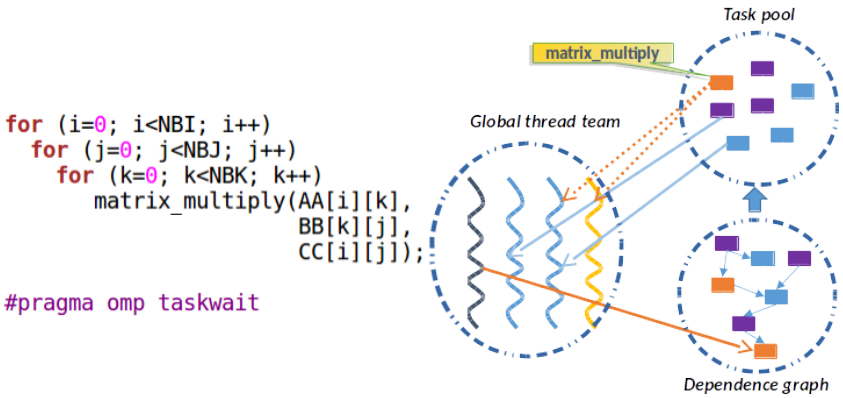


Figure 4.11. High-level representation of the Nanos++ environment

erogeneous 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 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).

4.2.1.2. Task Manager - SOM hardware runtime

The task manager is refactored to be more flexible and is renamed to SOM - Smart OmpSs Manager. The new version of the manager is able to receive control messages, to drive the execution of tasks, and the data transfers.

Additionally, the SOM IP now incorporates a round-robin scheduler to use as much as possible all the different instances of an accelerator.

The xtasks backend libraries is improved to use the commands interface of the SOM.

4.2.1.3. Boards support and minor fixes

Minor issues have been fixed in Zynq7000, COM Express, Zynq U+, and Alpha-Data environments, regarding the runtime and the instrumentation infrastructures. Regarding the instrumentation, the interconnection is reorganized to reduce the consumption of resources.

4.2.1.4. Compiler

Compilation problems are fixed in functions that are intrinsics in Vivado HLS. Also, the compilation process is enhanced to take into account that there are functions - like memcpy and math functions, that are interpreted by Vivado HLS in a specific way. For those cases, we do not want that Mercurium generates any prototype function definition. If it does, the function is not recognized as intrinsic anymore by Vivado HLS. For those cases, and when the target of the code is the FPGA, now Mercurium does not generate the function prototype.

Thanks to an improved support for moving data variables to the FPGA, it is now possible to allow applying the target directive "#pragma omp target device(fpga)" to constant variable declarations.

The Mercurium compiler for the C language now automatically searches and moves the functions called from inside FPGA tasks, to the HLS intermediate file, for HLS compilation to the target. This avoids to have to annotate all functions used with the target directive. Also we have included the support to be able to call one function from several tasks.

The task wrappers are now labeled with the AIT version information to be able to check it against the underlying framework.

In addition, *wide* data transfers, of 128 bits in Zynq U+ boards, and 512-bits in the PCIe-based AlphaData boards is now supported. This option provided additional performance on the applications that can exploit it. This is the case of matrix multiplication and, in general, of applications doing large transfers. The performance comparison is done in section 4.2.1.6.

4.2.1.5. autoVivado / AIT - Accelerator Interation Tool

The autoVivado tool is refactored to abstract itself from the vendor backend, with the goal to be able to use backend toolchains from different vendors in the future. We have also renamed the tool to AIT - Accelerator Integration Tool.

Up to Vivado HLS release 2018.3 is found to work. Vivado versions starting in 2018 required an update on the code related to data transfers, to do properly type conversions.

4.2.1.6. Evaluation

We evaluate the matrix multiplication benchmark on the current OmpSs@FPGA infrastructure. Results show that we have improved the performance of the application till around 155 Gflop/s, when using wide 128-bit accesses.

The experiment has been done on the Xilinx Development Kit ZCU102, incorporating the Zynq Ultrascale+ XCZU9EG chip, with 4 ARM Cortex A53 cores (1.3 GHz), and the programmable logic, running at 300 MHz. The application places 3 in-

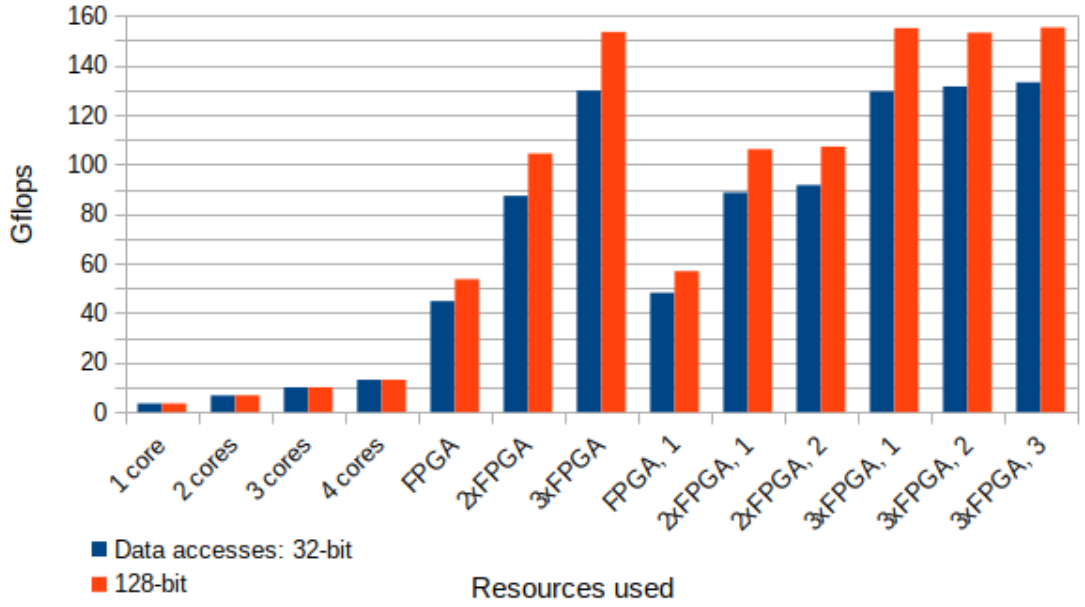


Figure 4.12. Evaluation of matrix multiplication on OmpSs@FPGA.

stances of the matrix multiplication kernel with block sizes of 256x256 single precision elements.

Figure 4.12 shows the results obtained. From left to right it shows the Gflop/s obtained on 1 to 4 cores, the FPGA with 1 instance of the block by block multiplication (FPGA), 2 instances (2xFPGA), and 3 instances (3xFPGA); and the combinations of 1, 2, 3 FPGA block instances with 1, 2 and 3 SMP cores. Blue bars represent the performance obtained when using 32-bit based data transfers, and the red bars when using 128-bit based data transfers.

On the current implementation, the high performance obtained from the FPGA makes the additional 1-2 Gflops provided by each additional core, nearly unnoticed. I can be better appreciated in the 32-bit based data transfers version, on experiments 3xFPGA, with 1, 2, and 3 additional cores.

4.2.2. OmpSs@Cluster

OmpSs@cluster is the distributed memory variant of the OmpSs programming model. It is based on the latest iteration of the OmpSs programming model, i.e. OmpSs-2 [5], which supports efficient task nesting through weak dependencies and early release of dependencies [35]. These features help open parallelism and avoid centralized task creation and scheduling, leading to higher scalability and greater use of the available parallelism in the underlying platform. OmpSs@cluster was originally developed in the ExaNoDe project [39], and is compatible with the SMP version of OmpSs-2, so that the same program and compiled binary can be executed on either an SMP or a cluster. OmpSs@cluster has been part of the OmpSs-2 public release since June 2019.

OmpSs@cluster is implemented via the Mercurium source-to-source compiler and Nanos6 task-based runtime system. Only the node feature, described below, requires support in Mercurium. Apart from this, no changes are required in the

compiler beyond those for the SMP variant of OmpSs-2.[2] Most of the support for OmpSs@cluster is implemented in Nanos6, through a new memory allocation API for local and distributed arrays, a cluster-aware task scheduler and support for inter-node offloading of tasks over MPI.

Work in the LEGaTO project in OmpSs@cluster has concentrated on improving the overall stability of the cluster runtime, porting a sparse matrix–vector kernel to OmpSs@cluster and supporting the execution of irregular applications through more efficient tracking of location information in the Nanos6 dependency system. The work on improved tracking of location information is ongoing at the time of writing this deliverable.

4.2.2.1. Basic Concepts

OmpSs-2 [5] is the latest generation of the OmpSs parallel programming model developed at BSC, which combines the design principles of two programming models: Star Superscalar (StarSs) [4], and OpenMP [44], hence the name OmpSs. OmpSs incorporates the main principles from StarSs, of tasking, dependencies, and heterogeneity, and integrates them with OpenMP's compiler directives (pragmas), which extend the C/C++ and FORTRAN programming languages. A sequential code is annotated with pragmas to produce a parallelizable version. A non-OmpSs compiler will ignore the OmpSs pragmas, but the program will still execute correctly. This ease of use constitutes OmpSs's philosophy as a productive high performance programming model, with no need to redesign the code for a parallel version.

OmpSs supports multiple parallel architectures including symmetric multiprocessing (SMP) of multicores and heterogeneous architectures (CPU + GPU), distributed-memory cluster architectures, and Field Programmable Gate Arrays (FPGAs).

Several features introduced in OmpSs have been included into the OpenMP standard: tasking in OpenMP 3.0, dependencies in OpenMP 4.0, and the taskloop construct in OpenMP 4.5.

Execution Model

OmpSs-2 uses a thread–pool execution model in which parallelism is expressed via tasks. In cluster mode, multiple instances of the application will be instantiated, one per node on which the application will run. On each node, OmpSs-2 starts by initiating a pool of threads that will be used to execute the tasks across the cores on that node.

The main function is wrapped as a task (main task). At the beginning of the program's execution, the first node (master node) enqueues the main into its task queue. In this way, the main function is executed on just one node, which is consistent with the sequential version of the application. One of the threads on the master node will pick the main task and start executing the main thread. All remaining threads will wait for a ready tasks that would be created later by the main task or any other task.

[2] An existing binary of an OmpSs-2 program can benefit from cluster execution merely by enabling the cluster-aware Nanos6 library.

```
#pragma oss task in(A[0;N]) in(B[0;N]) out(C[0;N])
foo(A,B,C);
```

Figure 4.13. Example of an OmpSs task with input dependencies over A and B arrays, and output dependency over C array.

If a thread encounters a task construct, it will explicitly generate a new task region, and when it is ready to execute, i.e. all dependencies have been satisfied, it will be assigned to one of the worker threads in the thread-pool. Also, each time a thread encounters a taskloop construct, the iteration space will be divided into sub-tasks. All sub-tasks will collaborate to execute the loop, and all sub-tasks will wait for other sub-tasks to finish executing their own portion.

The key features of OmpSs-2@Cluster are:

- Enforcing the expression of data dependencies between tasks to by making sure that the developer includes all memory access in the dependency list in the task clause. This assure the correct dependency order between tasks, provide the necessary information to the runtime in order to perform data fetching for offloaded remote tasks, and finally present a useful information of the scheduler to bring data were the task is offloaded.
- Distributed memory model and API that used by distributed computations. It is allocated within a task context and required to be deallocated by the same task, and can only be accessed by subtasks of the task that allocates it.

Dependency Model

OmpSs-2 follows an asynchronous data-flow model, in which the order of task execution is constrained by a dependency graph. The dependency graph is generated by the Nanos6 runtime using the addresses of the data dependencies attached to the tasks. A task may have a combination of the following consistencies, Read-after-Write (RaW), Write-after-Write (WaW), or Write-after-Read (WaR). Dependencies are defined by the user in the task construct either using the OpenMP depend clause or the OmpSs-style short notation. In addition, the semantic of the depend clause is extended with the keywords in, out, or inout, which correspond to defining input, output, or input–output access on the data dependency range. Figure 4.13 shows an example of how to define tasks with task dependences. In this example, all dependencies are defined over the N elements of the A, B, and C arrays starting at the first element indicated by the zeros in the example. At compile time, the compiler translates this dependency information and aggregates it into a specific data structure that defines the task and is used later by the runtime. When all the data dependencies have been satisfied, e.g. all the task's predecessors have finished executing, then the task becomes ready for execution.

The scheduler is responsible for assigning an available CPU and binding it to a thread. Alternatively, in OmpSs@cluster, the scheduler can decide to offload

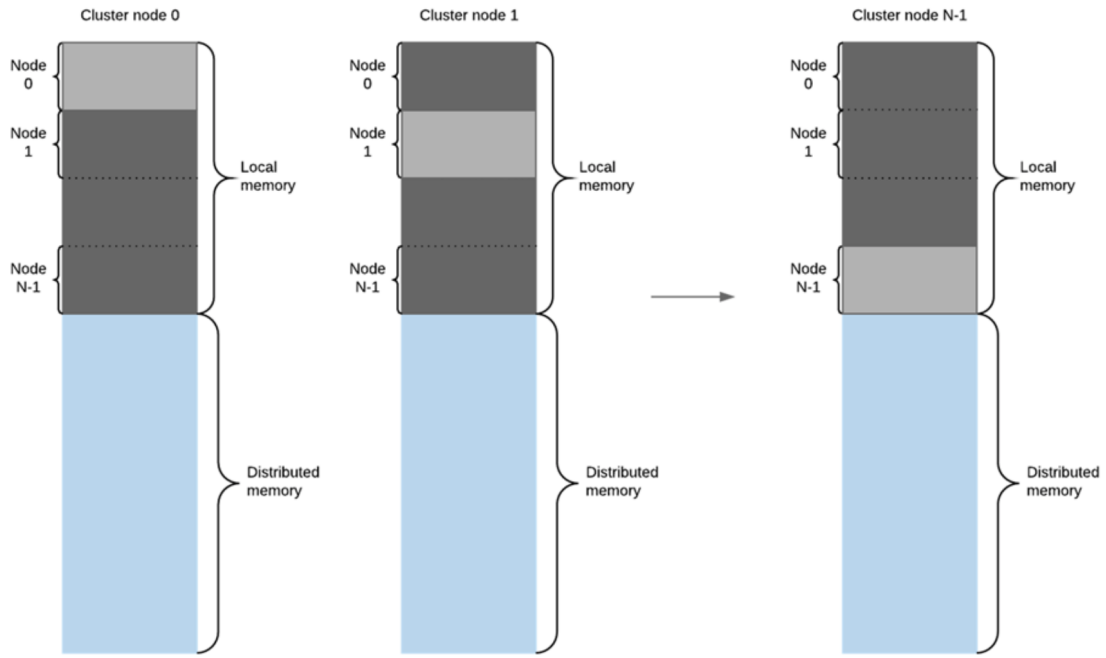


Figure 4.14. Memory model of Nanos6. The address space is divided between local and distributed memory. The local memory is further divided in equal chunks between the cluster nodes. Each cluster node allocates local memory from its personal chunk of the local memory space.

the task onto a remote node, using a push model. Hence, the task graph of the entire application is distributed across all nodes.

If a task reaches a taskwait, it will be blocked and the associated CPU will be freed to be used by another task. When a task finishes execution, all of its dependencies will be released. OmpSs-2 also allows nested dependencies, i.e. tasks inside tasks, giving the developer more control and gaining more flexibility to reflect the underlying problem pattern. To gain even more fine grain tasking and increase scalability, OmpSs-2 allows a parent task that has finishes execution and will not create further subtasks, to early release its own dependencies. Hence, it will not need to wait for all children subtasks to finish executing, and the worker thread can proceed to perform other computations.

OmpSs-2@Cluster Memory Model

All instances of the OmpSs@cluster application execute with the same virtual address space. This allows a ready task to be executed by any node, without address translation, so long as the necessary data copies are performed by the runtime system. This common address map is set up by the Nanos6 runtime at the start of execution, using the mmap system call. There are two types of the virtual memory that are visible to the developer: local, and distributed memory. Local memory is intended to be used by all computations within a single task scope, and by input/output arguments of subtasks to the current allocator task. Distributed memory is used for tasks that operate on distributed data, however it can only be accessed by substasks of allocator tasks, and not the allocator task itself. The local region is further subdivided into regions as many as cluster nodes, and associate each region with each of the nodes. This allows to offload

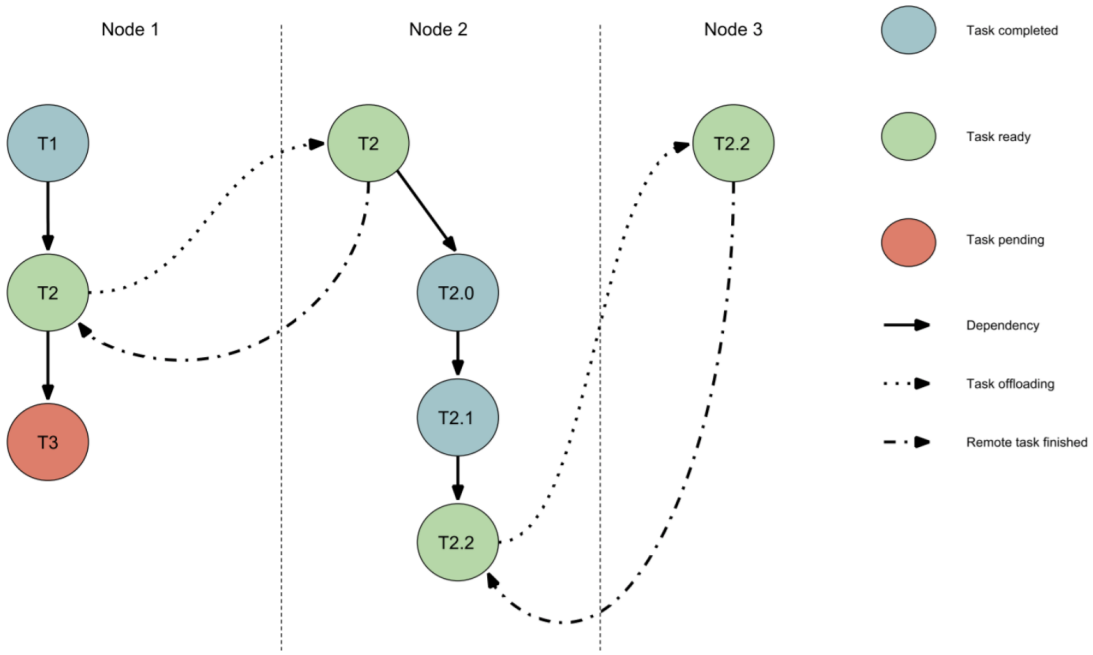


Figure 4.15. Nanos 6 task offloading model

local tasks to remote node into the same locale address as the homenode (node offloaded the task) without the need to translate address space between the two nodes.

The distributed memory allocation API splits the distributed memory (Figure 4.14) and assigns each node a region as in the local memory. However, the split occurs according to a distribution policy parameter that is provided to the API, which also guides the scheduler later. For tasks using standard memory allocators such as the malloc libc function, they still will be managed by OmpSs-2@Cluster, however the tasks will only run on the node that allocates the memory and will not be offloaded into other nodes.

OmpSs-2@Cluster Task distribution

One of the main functions of a cluster-runtime implementation of a programming model is to handle computation's (tasks in our case) distribution between cluster nodes. For this to be true, a task has to satisfy all of its dependencies and be in a ready state, then the scheduler will decide where to offload it. A restriction must be taken into consideration, that the developer must define all data dependencies and include all memory access in the dependency list. This is an essential step for the scheduler to know the memory location and size of the data associated with a task. Then it will later be able to send the correct data when offloading the task into another node or when copying data back to the homemade.

Figure 4.15 depicts how tasks are offloaded. The scheduler decides to offload task T2 from Node 1 to Node 2, hence it creates a copy of the task and send it to Node2, the original task T2 is kept on Node 1 to preserve the task graph order on Node 1. T2 then can start execution and generate more subtasks (T2.0, T2.1, and T2.2). The creation of these subtasks is completely transparent from

Node 1 point of view that allows the distribution of the task graph of the entire application among all nodes and avoid extra synchronizations. The scheduler also decides to offload subtask T2.3 to Node 3. After T2.2 finishes it reports back to the original T2.2 on Node 2, then now T2 reports back to the original T2 on Node 1. Note that T3 will be in pending until T2 finishes execution and release all of its dependencies.

4.2.2.2. Irregular Problems

The terms regular and irregular applying to code were originally coined by compiler designers. In regular code, there is no data dependency between memory references. Dense Matrix–Vector multiplication is a good example of such regular flow. Without knowing any of the matrix or vector elements, and with the knowledge of only the data input size, and the starting memory reference of the matrix and vectors, the program behavior can be easily predicted.

Irregularity can emerge as one of three forms. Irregular control flow, which emerges from conditional expressions. Irregular data-structure such as unbalanced tree, unstructured gird, or sparse matrix representation that might be dynamic and change its structure at each iteration of the solution space. The third source of irregularity is irregular communication and memory access patterns that are characterized by a sparse data structures in which accesses are made through a level of indirection or nonlinear array subscript patterns that are inherently inconsistent with cache-based architecture which eventually hides memory latency by memory reusing. The latter form of irregularity can be directly related to either control or data irregularity. The terms irregular algorithms and irregular data-structures are used interchangeably in literature. Irregular programs can be found in domains such as Sparse Direct Methods, data mining, decisions problems that use Boolean satisfiability, optimization theory, social networks, system modeling, compilers, discrete-event simulation, meshing, and n-body simulation. A major problem occurs when code has a large amount of data dependency. The input values determine the runtime execution behavior, hence the prediction of the program flow is a difficult task and cannot be elucidated statically. Exploiting the irregularity dependency requires runtime strategies such as speculative or optimistic parallel execution methods. For example, in an implementation of a binary search tree, the values and the order in which they are processed affect the control flow and memory references. Processing the values in sorted order will generate a tree with only right children whereas the reverse order will generate a tree with only left children, thus exercising a different control flow path. Even with unsorted inputs, the order of the values determines the shape of the tree as well as the order in which the tree is built, thus affecting the memory reference stream. Graphs are similar to trees regarding their inherently irregular nature. The memory access patterns generally have a data-dependent based on the fact that the connectivity of the graph and the values on nodes and edges determine which graph elements are accessed by a compute element. However, the connectivity and values are unknown prior the input is available and may change dynamically.

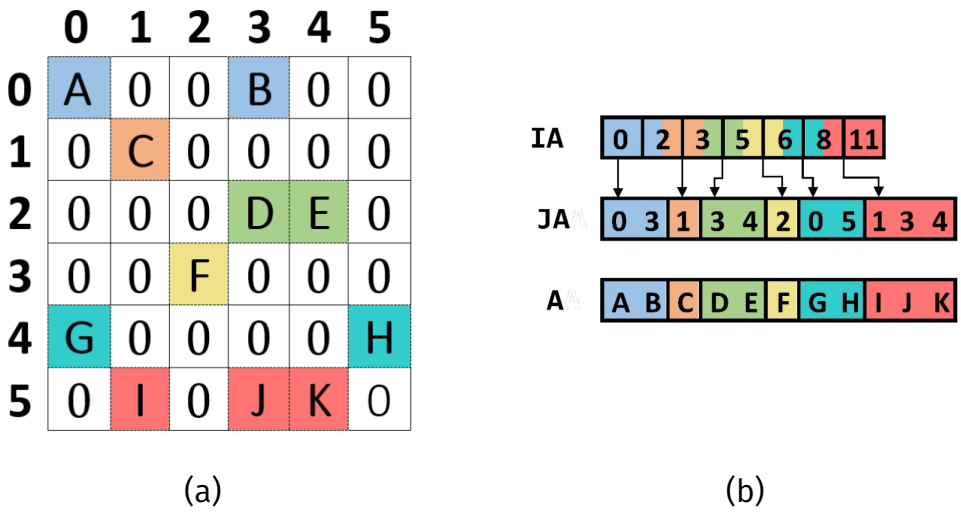


Figure 4.16. Compressed Sparse Row (CSR) sparse matrix format representation. In (a) a 2D dense matrix, and (b) is the CSR representation of (a).

4.2.2.3. Case study: Sparse matrix–vector multiplation

This section discusses SpMV as a case study of problem with irregular data structure. This is an initial performance evaluation of the current Nanos6 runtime on distributed memory system. We begin by first explain scaler SpMV implementation, and then OmpSs-2@Cluster parallel implementation. The OmpSs-2@Cluster is then compared against pure MPI implantation of the same algorithm.

Scalar CSR SpMV

Sparse matrix by dense vector multiplication is considered the cornerstone kernel in direct solvers for sparse linear systems and iterative sparse eigenvalue methods. Sparse matrix is any 2D storage with enough zeros that can be taken advantage of by many storage formats. Compressed Sparse Row (CSR) or Yale format considered the de facto format used in HPC applications. CSR represents a 2D sparse matrix M by three 1-dimensional arrays A, JA, and IA, and organize the nonzero elements by row order. Array A stores the nonzero elements, JA is the corresponding column index of each nonzero element, and the IA array represents the row offsets of each row inside A. Figure 4.16 illustrates an example of CSR representation of a 2D dense matrix.

Sparse matrix–vector multiplication (SpMV) performs sparse matrix by dense vector multiplication, defined by the equation: $y = Ax$, where x and y are dense vectors, and A is an $m \times n$ sparse matrix. Figure 4.17 shows a pseudocode of a scaler SpMV based on CSR format that we used as a reference implementation. As observed CSR requires expansion and indirect indexing to the column entries ($x[JA[j]]$), unlike the dense matrix representation where elements are stored contiguously in memory. Essentially this implies irregular memory access pattern with a naïve implementation.

```
void csr_matvec(int rows, double *A, int *JA, int *IA, double *x, double *Y)
{
    for(int i = 0; i < rows; ++i)
    {
        double sum = 0.0;
        for (int j = IA[i]; j < IA[i+1]; ++j)
        {
            sum += A[j] * x[JA[j]];
        }
        Y[i] = sum;
    }
}
```

Figure 4.17. Scalar Compressed Sparse Row (CSR) Sparse Matrix Vector Multiplication (SpMV) Routine.

OmpSs-2 CSR SpMV

The SpMV OmpSs-2 implementation is a straightforward taskified version of the scaler SpMV presented in code snippet in Figure 4.17. For simplification and better understanding we explain the taskfication on the dense representation first then we later map it to the CSR representation. A sparse matrix M of size m (rows)$\times$ n (cols) is subdivided into tasks, at which each task will be assigned TS number rows, thus total number of rows/TS tasks with each task work on $TS \times n$ independent chunks of the matrix M. Reflecting the dense representation on the CSR representation requires a simple mapping of each chunk in the original matrix to the CSR arrays A, IA, and JA. This can be seen in the code snippet in Figure 4.18.

4.2.2.4. Results

Figure 4.19 shows the strong scalability of the CSR-SpMV kernel from Subsection 4.2.2.4, in comparison with an optimized MPI implementation. OmpSs-2 was tested with the all available schedulers, which are Locality and Random. The benchmark was tested on the MareNostrum 4 supercomputer and the code written in C/C++. The sparse matrices used were generated with 70% of the matrix are zeros and stored as space-separated values file in CSR format. Figure 4.19 shows strong scalability of CSR-SpMV against MPI running on 1 up to 8 nodes with 24,000 $\times$ 24,000 matrix.

Figure 4.20 provides an Extrae performance tool's trace visualized via Paraver tool showing the CSR-SpMV MPI execution on 2 nodes. The yellow lines show the communications and synchronization points between threads of execution. Each MPI process (rank) receives sub array copy of IA, JA, and A, and compute (denoted by the blue stripes) its portion of the matrix (TS rows) and store it on local output array. When all ranks finish, rank number 0 performs a reduction operation on these local output arrays to produce the final array output.

The OmpSs-2 execution flow trace is shown in Figure 4.21(a) running on 2 computing nodes. Figure 4.21(b) shows the same trace with the communication lines (yellow lines) between tasks. The green stripes are CSR-SpMV kernel tasks. The red strips mark tasks that offloaded remotely from node 1 to node 2 in this case.

```
for (int k = 0; k < rows / TS; ++k)
{
    int chunkIndex = IA[k*TS];
    int chunkSize = IA[TS*(k+1)] - IA[TS*k];
    int idx = k*TS;

    #pragma oss task                \
    in(A[chunkIndex; chunkSize])    \
    in(JA[chunkIndex; chunkSize])   \
    in(IA[idx; TS + 1])             \
    in(X[0; cols])                  \
    out(Y[idx; TS]) label(csr_matvec_task)
    csr_matvec(
        TS,
        &A[chunkIndex],
        &JA[chunkIndex],
        &IA[idx],
        X,
        &Y[idx]);
}
```

Figure 4.18. OmpSs-2 Compressed Sparse Row (CSR) Sparse Matrix Vector Multiplication (SpMV) of the scalar SpMV code from Figure 4.17

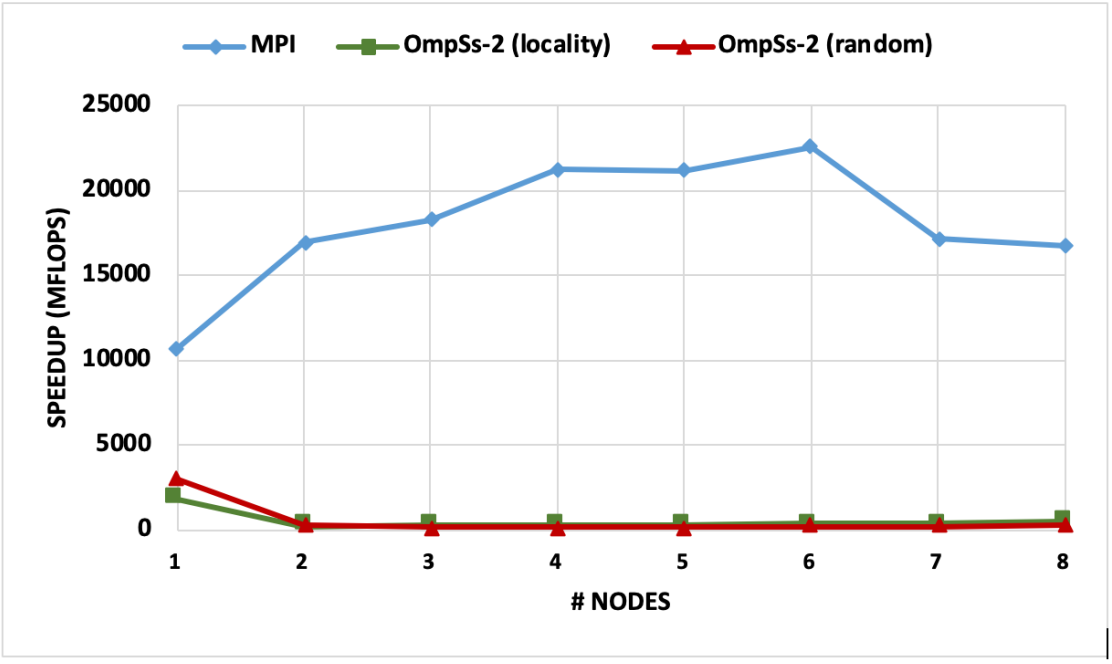


Figure 4.19. CSR-SpMV MPI vs. OmpSs-2 strong scalability of 24k x 24k sparse matrix on MareNostrum 4.

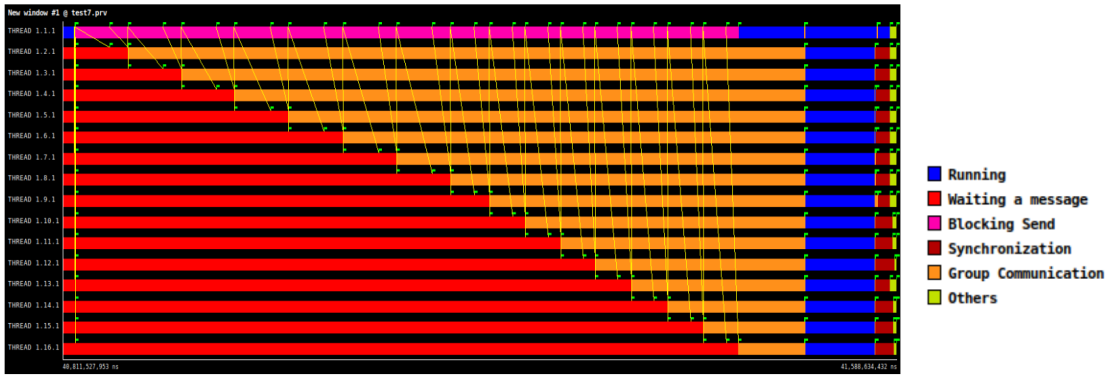


Figure 4.20. CSR-SpMV MPI (of 24k x 24k matrix) execution trace showing communications between threads (vertical yellow lines) and execution flow of concurrent threads shown by the horizontal blue lines.

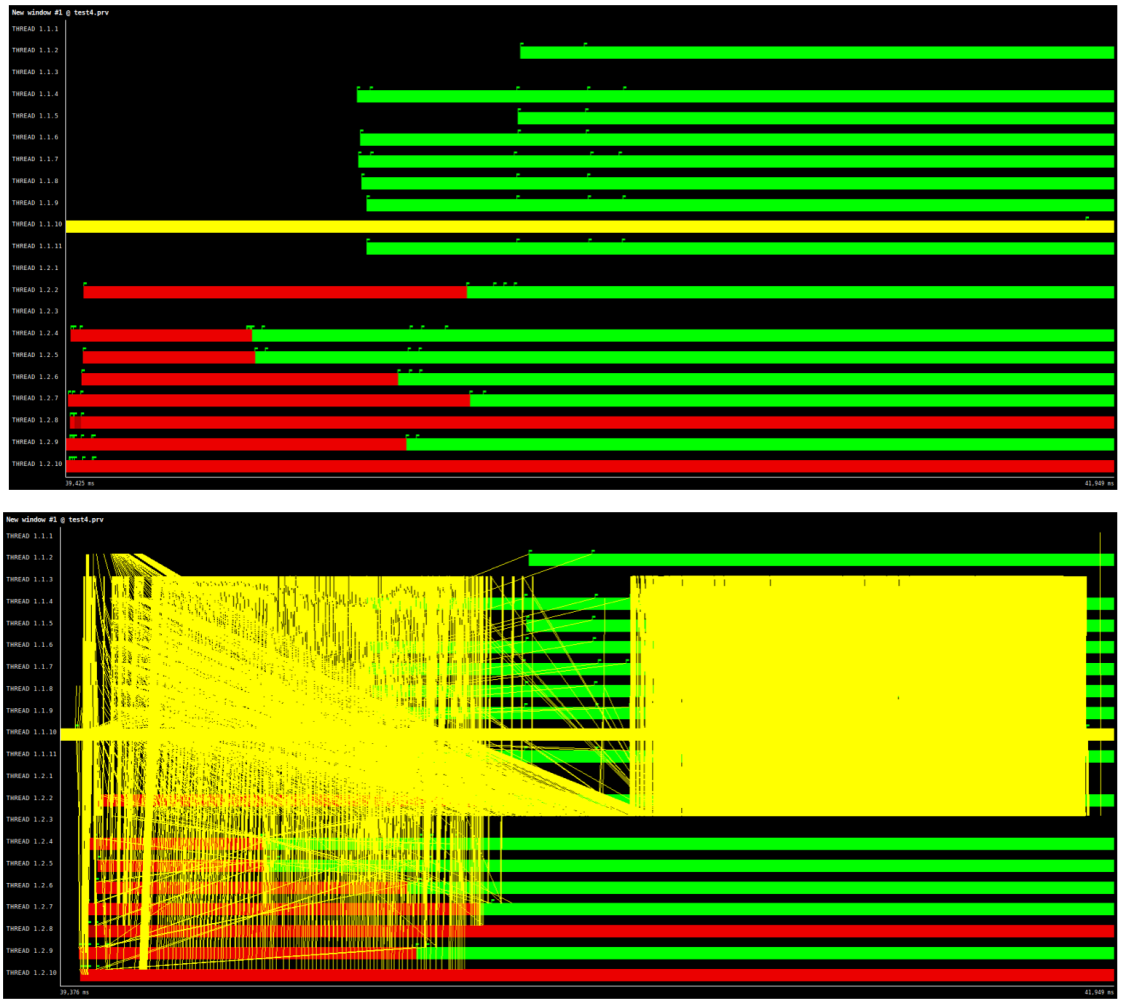


Figure 4.21. CSR SpMV OmpSs-2 execution trace of 24k x 24k sparse matrix on two nodes. (a) showing trace with no communication lines. (b) with communication line.

```
#pragma oss task in(A[0;N]) in(B[0;N]) out(C[0;N]) node(1)
foo(A,B,C);
```

Figure 4.22. Example of an OmpSs task that must execute on node 1.

4.2.2.5. Streams and node clause

A large part of the overhead of multi-node execution (seen in Figure 4.19) especially for irregular problems, has been determined to be due to messages sent by the Nanos6 cluster runtime to track location information. As remarked in Subsection 4.2.2.1, before executing any ready task, it is necessary to ensure that the node has an up-to-date copy of any data that is accessed through an in or an inout dependency. This requires control messages to (a) track the location information and, if necessary (b) to copy the data between nodes.

Tracking of location information is done through the dependency system, alongside messages that indicate that data is ready to be accessed, with read or write permission, as appropriate. In the original implementation, such messages respect the nested hierarchy of tasks in the program source code, which often requires several hops up and down the hierarchy via the common parent of two tasks. We have repurposed the streams feature of Nanos6 in order to enable a direct path for the common case where dependent tasks are running on the same node.

In addition, we introduced the node clause to the task annotation, which causes the task to always be scheduled to the same node. An example is given in Figure 4.22. This clause has been useful in experimentation and evaluation of the scheduler. It is also useful for applications such as the Smart mirror use case at the University of Bielefeld, which can be executed over multiple nodes (two Xavier nodes) with the constraint that calls to the DarkNet neural network framework are always made on the same node.

5. Runtime support for Fault Tolerance and Security

5.1. GPU Checkpointing

In the previous deliverable 3.2, we have discussed all the technical work carried on to allow for GPU checkpointing in FTI. This work has been tested with multiple applications and the evaluation has shown great performance results as reported in D3.2. Afterwards, we focused on pushing this research line into concrete outputs for the project.

5.1.1. Code Release

The prototype for GPU checkpointing was tested for correctness and high performance, but significant changes were necessary in order to cleanly integrate it into the FTI library. For this, a significant effort was done to provide a transparent API easy to use by scientific developers. We produced a new release of FTI, release Heraklion v1.3.

This release includes full support for GPU checkpointing, a new version of differ-

ential checkpointing, a complete implementation of incremental checkpointing, and full support for HDF5 checkpointing. The GPU checkpointing features presented in the previous deliverable D3.2 have been also integrated in this release. They can detailed as follows.

- New major feature allowing users to checkpoint data allocated in the GPU device memory.
- New major feature allowing users to use incremental checkpointing for CPU and GPU data by adding one by one the variables to the checkpoint file.
- New examples in the examples/GPU directory that checkpoint GPU data.
- New unitary tests for the new features.
- New configurable/flexible local test structure.
- Complete and full code documentation generated with Doxygen.

This software release is one of the important outputs of the Legato project and it is public and open source, it can be downloaded and cloned from our github repository `https://github.com/legato-project/fti`.

5.1.2. Paper Publication

In addition to the FTI release, we agregated all the design choices and experiments into a scientific article explaining the novelty of our approach and the high performance results observed on our results. The paper was accepted and published in the proceedigns at the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing.

5.2. FPGA Checkpointing

To provide Checkpoint Restart support for FPGAs in the LEGATO project we need to take into account several aspects. Firstly, checkpoint restart is a valid approach in case of large-scale applications which execute on a distributed system in which multiple nodes execute in parallel a part of the application. The norm to develop applications on top of such systems is MPI. Consequently, our fault tolerance library called Fault Tolerance Interface (FTI), main support targets such an application design. LEGATOs main purpose is to provide a system in which the developer has a clear overview of the system and a clear, well defined, path to develop applications. Towards this direction, in the core of the Legato project there is OMPSS, a programming model which provides support for development of complex heterogenous devices for energy efficiency and performance. OMPSS is accompanied by a runtime system Nanos++. Nanos++ is designed to execute the bulk of the runtime code on a CPU and offload any heterogenous tasks to the different accelerator available in the system (GPU, FPGA etc). The Nanos++ runtime system also takes care to transfer data efficiently from and to the CPU before/after a task executes. Consequently, when using the OMPSS programming model, the data are initially allocated on the CPU accessible memory by the developer, and the runtime transfers the data to and from the accelerator

device. To support C/R on an FPGA system FTI exploits the previous observation, in which data are going to be accessible from the CPU before or after the termination of a task. Therefore, it is sufficient to store data to the checkpoint only when they reside on the memory accessible directly from the CPU. Consequently, FTI already provides such a support, however it is not straightforward on how to compile an MPI library for FPGA systems, since typically it is required to perform cross compiling of both MPI and FTI for the ARM architecture of the processing system (PS) of the FPGA. To make it easier for developers to use our fault tolerance library we provide an extension on our build script which semi-automatically builds the library for such a system. To use the script one needs to just update the paths of the cmake extension script called "fpga.cmake". Afterwards the developer needs to invoke cmake as follows:

```
cmake -C ../fpga.cmake -DCMAKE_INSTALL_PREFIX:PATH=/path/to/fti
..
```

After building has finished, the user needs to copy all the created libraries into the sd card of the FPGA system.

Currently, we have experimented with an implementation of K-Means which executes on 4 nodes each node consists of a zynq system (PS + PL), the actual computation takes part on the PL using OMPSS@fpga tasks, whereas the checkpoint is taking place on the PS on each node. Although, the C/R is taking place correctly, the timings and the respective overhead are quite large, since the data are written on the FPGA SD-Card which has slow write times. We plan to further measure the overhead on more efficient system setup.

5.3. FPGA Unvervolting for CNN Accelerators

We empirically evaluate an undervolting technique, *i.e.,* supply voltage underscaling below the nominal voltage level, to improve the power-efficiency of Convolutional Neural Network (CNN) accelerators mapped to Field Programmable Gate Arrays (FPGAs). Undervolting below a safe voltage level can lead to timing faults due to excessive circuit latency increase. We evaluate the reliability-power trade-off for such accelerators. We perform experiments on three identical samples of modern Xilinx ZCU102 FPGA platforms with five state-of-the-art image classification CNN benchmarks. This approach allows us to study the effects of undervolting technique for both software and hardware variability. We achieve a total of more than 3X power-efficiency ($GOPs/W$) gain via undervolting. 2.6X of this gain is the result of eliminating the voltage guardband region, namely, a safe voltage region below the nominal level that is set by FPGA vendor to ensure the correct functionality in the worst-case environmental and process scenarios. 43% of the power-efficiency gain is due to the further undervolting below the guardband. However, this gain comes at the cost of accuracy loss in the CNN accelerator. To alleviate this issue, we evaluate an effective frequency underscaling technique.

5.3.1. Introduction

Deep Neural Networks (DNNs) and specifically Convolutional Neural Networks (CNNs) have recently attained significant success in image and video classification tasks. They are fundamental for state-of-the-art real-world applications

running on embedded systems as well as data centers. These neural networks learn a model from a dataset in their training phase and make predictions on new, previously-unseen data in their classification phase. However, their power-efficiency is inherently the primary concern due to the massive amount of data movement and computational power required. Thus, the scalability of CNNs for enterprise applications and deployment in battery-limited scenarios, such as in drones and mobile devices, is a crucial concern.

Typically, hardware acceleration using GPUs [52], FPGAs [43], or ASICs [28] leads to a significant reduction in CNN power consumption. Among them, FPGAs are rapidly becoming popular. This increase in the popularity of FPGAs is attributed to their power-efficiency compared to GPUs, their flexibility compared to ASICs, and recent advances in High-Level Synthesis (HLS) tools that significantly facilitate easier development of FPGA-based designs. Hence, major companies, such as Microsoft Brainwave project [18], have made large investments in FPGA-based CNN accelerators. However, recent studies show that FPGA-based accelerators are at least 10X less power-efficient compared to ASIC-based ones. We aim to bridge this power-efficiency gap by empirically understanding and leveraging an effective undervolting technique for FPGA-based CNN accelerators.

FPGA hardware vendors usually add a voltage guardband to ensure the correct operation of FPGAs under the worst-case circuit and environmental conditions. However, these guardbands can be very conservative and unnecessary for state-of-the-art applications. We extends undervolting studies to real FPGAs running CNNs. Specifically, we study the classification phase of FPGA-based CNN accelerators, as this phase can be repeatedly used in power-limited edge devices (unlike the training phase, which is invoked much less frequently). Unlike simulation-based approaches that may not be accurate-enough [41], our study is based on real off-the-shelf FPGA devices.

Reducing the supply voltage in the voltage guardband region does *not* lead to reliability issue under normal operating conditions, and thus, eliminating this guardband can result in a significant power reduction for real-world applications. We experimentally demonstrate a large voltage guardband for modern FPGAs: an average of 31.5% with a slight variation across hardware platforms and software benchmarks. Eliminating this guardband leads to significant power-efficiency ($GOPs/W$) improvement, on average, 2.6X, without any performance or reliability overheads. With further undervolting, the power-efficiency improves by an extra 43%, leading to a total improvement of more than 3X. This additional gain does not come for free, as we observe exponentially-increasing CNN accuracy loss below the guardband region. With further undervolting below this guardband, our experiments indicate that the minimum supply voltage at which the internal FPGA components could be functional (V_{crash}) is equal to, on average, 65% of V_{nom}. Further reducing the supply voltage results in system crash.

We evaluate our undervolting technique on three identical samples of the Zynq-based ZCU102 platform, a representative modern commercial FPGA from Xilinx. We experimentally evaluate the effects of reduced-voltage operation in on-chip components of the FPGA platform, including Block RAMs (BRAMs) and internal FPGA components, including Look-Up Tables (LUTs), Digital Signal Processors

(DSPs), buffers, and routing resources.[1]. We perform our experiments on five state-of-the-art CNN image classification benchmarks, including VGGNet [13], GoogleNet [10], AlexNet [48], ResNet [17], and Inception [16]. This enables us to experimentally study the workload-to-workload variation on the power-reliability trade-offs of FPGA-based CNN accelerators. Specifically, we extensively characterize the reliability behavior of the studied benchmarks below the guardband level and evaluate a frequency underscaling technique to prevent the accuracy loss in this voltage region.

5.3.2. Experimental Results: Voltage Behavior Analysis

We present and analyze our experimental results from reduced-voltage operation on FPGA boards. These results are collected at ambient temperature. Each result presented is the average of 10 experiments account for any variation between different experiments; although, the variation we observed was negligible.

5.3.2.1. Power Analysis of FPGA-based CNN Accelerators at the Nominal Voltage Level (V_{nom})

We measure the total on-chip power consumption of the baseline configuration to be an average of $12.59W$ for benchmarks, at the nominal voltage level (V_{nom}) and ambient temperature. This value includes the power consumption at on-chip voltage rails, including V_{CCBRAM} and V_{CCINT}. We observe that internal FPGA components on the V_{CCINT} rail dissipate more than 99.9% of this on-chip power. We believe this observation is due to power-efficient BRAM designs, using techniques like dynamic power gating, in modern Ultrascale+ FPGA platforms, including in the studied ZCU102 FPGA. Older generations of Xilinx FPGAs like the 7-series are not equipped with this capability. Thus, for such older devices, BRAM power consumption was the main source of FPGA power consumption, as shown in our previous studies [42]. To this end, as we study the power-reliability trade-off, we concentrate on V_{CCINT} due to its dominance in FPGA power consumption.

5.3.2.2. Overall Voltage Behavior

Our experiments reveal that a large voltage guardband below V_{nom} exists for V_{CCINT}, as shown in Figure 5.1 for three hardware platforms and five CNN benchmarks. In the voltage guardband region, as we reduce voltage there is no performance or reliability degradation, and thus, under normal conditions, eliminating this voltage guardband can lead to significant power savings without any overhead. As Figure 5.1 shows, we measure the average guardband amount to be $850mV - 570mV = 280mV$, with a slight variation across different benchmarks. In other words, we observe that $V_{min} = 570mV$ (on average) is the minimum safe voltage level of the accelerator, where there is no accuracy loss. As we further undervolt below V_{min}, we enter a region called the *critical region* in which the reliability of the hardware and, in turn, the accuracy of the CNN starts to decrease significantly. As Figure 5.1 depicts, we measure the average critical voltage region size, to be $570mV - 540mv = 30mV$, with a slight variation across different benchmarks. As we further undervolt below V_{min}, we reach a point at which the FPGA does not respond to requests and it is not functional. This point is called

[1]These internal FPGA components share a single voltage rail in the studied FPGA platform. To our knowledge, such voltage rail sharing is a typical case for most modern FPGA platforms

V_{crash}. We find that $V_{crash} = 540mV$ on average, with a slight variation across different benchmarks.

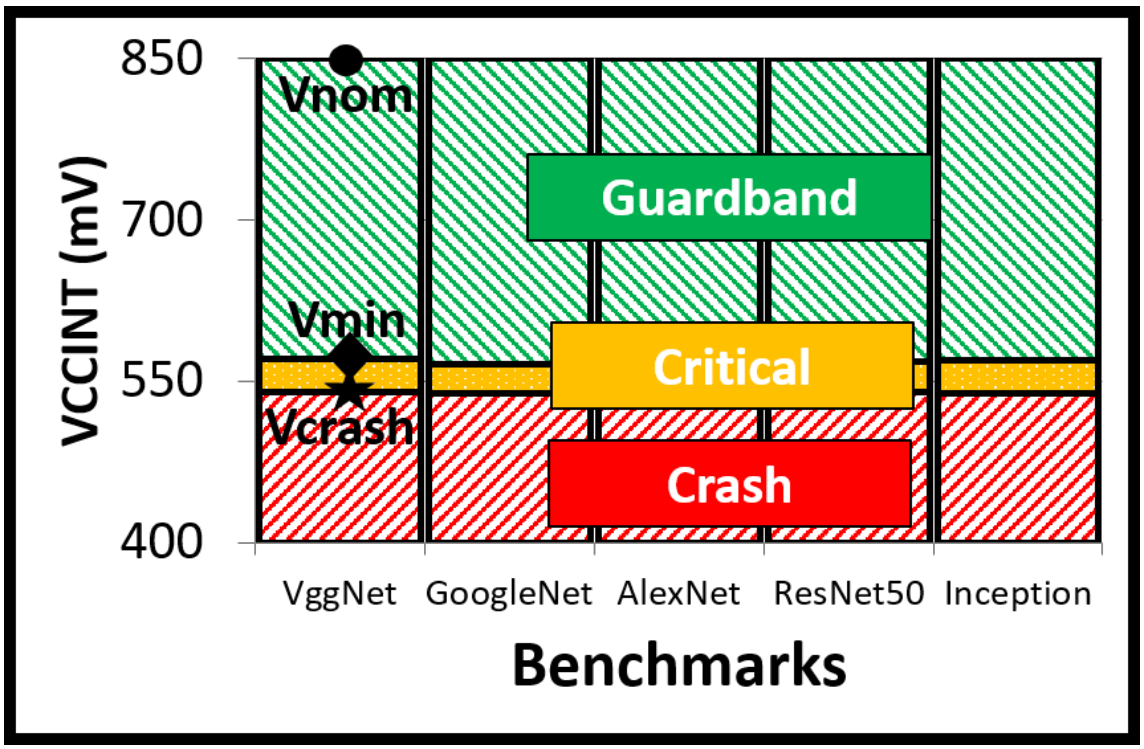


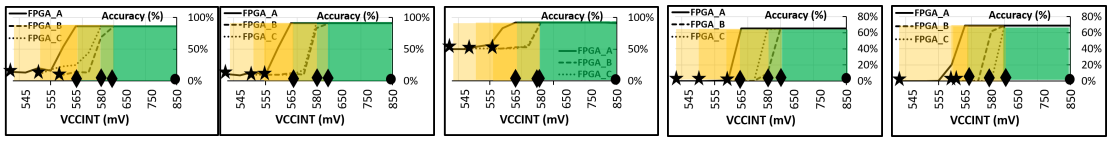
Figure 5.1. Voltage regions with a slight workload-to-workload variation (averaged across three hardware platforms).

5.3.3. Power-reliability Trade-off for Reduced-voltage FPGA-based CNN Accelerators

In this section, we elaborate the power-reliability trade-off for the FPGA-based CNN Accelerators under reduced-voltage operation.

5.3.3.1. Power-Efficiency Analysis

Figure 5.2 presents the power-efficiency results ($GOPs/W$) for five different CNN workloads. As expected, reduced power consumption with further undervolting results in constantly improved power-efficiency in both guardband and critical voltage regions. Consequently, the power-efficiency at V_{crash} is more than 3X of that at V_{nom}, for the same design of the given CNN accelerator. 2.6X of the gain in power-efficiency is the result of eliminating the voltage guardband without any CNN accuracy loss. 43% further power-efficiency gain is due to further undervolting in the critical region that comes at the cost of CNN accuracy loss.



(a) VGGNet. *(b) GoogleNet.* *(c) AlexNet.* *(d) ResNet.* *(e) Inception.*

Figure 5.3. Effect of reduced supply voltage on the accuracy of CNN workloads (separately for three hardware platforms).

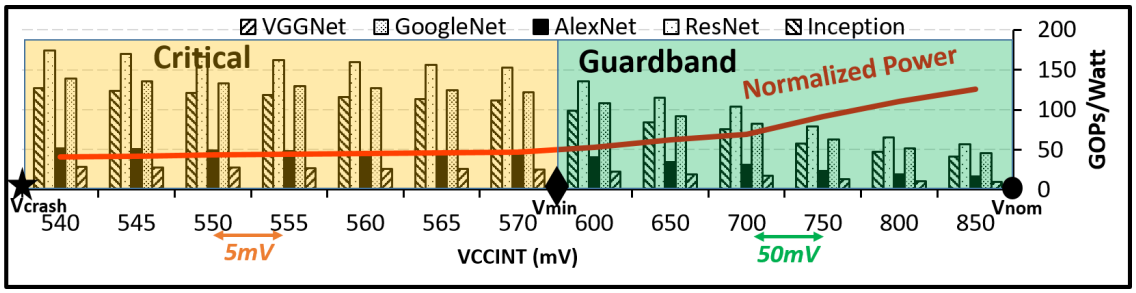


Figure 5.2. Power-efficiency ($GOPs/W$) improvement via undervolting with INT8 quantized and without pruning at ambient temperature (averaged across three hardware platforms).

5.3.3.2. Reliability Analysis

As we undervolt until V_{min}, there is no reliability overhead. However, as we further undervolt below V_{min}, the reliability of the hardware is significantly affected due to the further increase in datapath delay. The effect of the reliability loss is fully application-dependent due to different resilience levels of different applications. We study this effect on several CNN workloads. Figure 5.3 depicts our experimental results. As shown before, as we reduce the supply voltage, power-efficiency improves. When we reduce the supply voltage below V_{min}, we observe that the accuracy of all benchmarks gradually reduces. With further undervolting, when the supply voltage reaches an average of $560mV$ across different platforms and benchmarks, the accuracy of the benchmarks drops greatly, and the classifier behaves randomly. Our experiments show that benchmarks with more parameters, *e.g.,* ResNet and Inception are relatively more vulnerable to undervolting faults below V_{min}. Also, as seen, there is a variation of $\Delta V_{min} = 31mV$ and $\Delta V_{crash} = 18mV$ across different FPGAs. This variation can be due to the process variation across different FPGAs.

5.3.4. Frequency Underscaling

As shown earlier, in the critical voltage region below the guardband, CNN classification accuracy dramatically decreases. In this section, we aim to overcome this accuracy loss by exploiting frequency underscaling. To be more precise, we aim to find a more energy-efficient voltage setting than the undervolted V_{min}, which also provides accurate results. To this end, for each voltage setting below V_{min}, we aim to identify the maximum frequency value F_{max} with which the system does not experience any accuracy loss. When we find this frequency, we evaluate the energy efficiency of the system. As we underscale the frequency of the system, the performance of the application reduces. Therefore, we use the $GOPs/J$ metric as it accommodates for both performance and energy consumption. Ta-

ble 5.1 summarizes the experimental results of the frequency underscaling in the critical voltage region (averaged across three hardware platforms). These experiments are based on frequency and voltage steps of $25Mhz$ and $5mV$, respectively. The column V_{CCINT} corresponds to the supply voltage of a given setting. The column F_{max} corresponds to the maximum frequency at which leads to no accuracy loss. The remaining columns: $GOPs$, $Power$, $GOPS/W$, $GOPS/J$ are normalized to the respective values of executing the system in the setting $V_{CCINT} = V_{min} = 570mV, F_{Max} = 333Mhz$ which are the baseline settings used to obtain results presented in Section 5.3.2. Table 5.1 indicates that multiple voltage settings V_{CCINT} map to the same operating Frequency F_{max}: supply voltages between $560mV$ to $545mV$ require same frequency of $F_{max} = 250Mhz$. This is because the frequency step we use is $25Mhz$. Using smaller steps of frequency can distribute the F_{max} values more evenly.

Table 5.1. Evaluation of frequency underscaling to prevent CNN accuracy loss in the critical voltage region, at ambient temperature (averaged across three hardware platforms).

V_{CCINT} **(mV)**	F_{max} **(Mhz)**	$GOPs$ **(Norm)**	$Power$ **(Norm)**	$GOPs/W$ **(Norm)**	$GOPs/J$ **(Norm)**
570	333	1.00	1.00	1.00	1.00
565	300	0.94	0.97	0.97	0.87
560	250	0.83	0.84	0.99	0.75
555	250	0.83	0.78	1.06	0.80
550	250	0.83	0.75	1.10	0.83
545	250	0.83	0.74	1.12	0.84
540	200	0.70	0.56	1.25	0.75

For all the combinations of (V_i, F_i) that provide error-free results presented in the Table 5.1 in the critical voltage region, power consumption decreases with decreasing $V_i < V_{min}$ and $F_i < F_{max}$. This behavior is because we decrease both the supply voltage and the operating frequency. However, at the same time, we effectively decrease the performance of the system. Consequently, the best combination in terms of energy efficiency is the one with the highest frequency of $F_{max} = 333Mhz$, which also is our baseline. In other words, it is not worth to underscale the frequency and voltage to find a more energy-efficient execution point. However, as a trade-off, at lower voltage-frequency levels, the design is more power-efficient.

5.4. Trusted Key Management

In LEGaTO, we make use of TEE technologies (e.g., Intel SGX) to protect legacy applications in an untrusted environment. We enable these applications to run in Intel SGX enclaves without any source code modification by using our toolchain SCONE [3]. However, note that running applications inside SGX enclaves is not enough to ensure the confidentiality and integrity of them. First, we need to ensure the code and data of applications running inside enclaves are correct

and not be modified by anyone e.g., an attacker. Second, we need to guarantee that they are properly configured and securely provisioned with the "secrets" (e.g, encryption/decryption keys, TLS certificates) to execute inside enclaves. In other words, we need to provide a mechanism to securely transfer the configuration and secrets (encryption/decryption keys, TLS certificates, etc) to start them inside enclaves. We also need to ensure not only confidentiality, integrity but also freshness of their data and code to protect them against rollback attacks. In addition, these secrets of applications should not be maintained under control of single entity or stakeholder— which maybe compromised and would represent a single point of failure.

Typically, traditional approaches are often based on *ad hoc* techniques and rely on a hardware security module (HSM) as root of trust to handle these challenges. In the context of this project, we propose a more powerful and generic approach [20, 21] to trust management that relies on TEEs and a set of stakeholders as root of trust. We designed and implemented a framework, called PALAEMON, which can operate as a managed application deployed in an untrusted environment, e.g., a public cloud. We can delegate operations of PALAEMON to an untrusted cloud provider but still guarantee the integrity and confidentiality of data without trusting any individual human (even with privileged/root access) and system software.

5.4.1. Approach: A Trusted Management Service

In this section, we describe how our proposed framework addresses the above introduced challenges.

One of the major functionalities of the proposed framework (PALAEMON) is to transfer secrets in a trusted manner to applications running inside enclaves after attesting them. Each application is executed in Intel SGX enclaves and associated with a *security policy* that defines *which applications can access which secrets on which hosts*. Applications are identified by a MRENCLAVE [14] and the content of the files they can access. Secrets are typed and can either be explicitly defined, or randomly generated by PALAEMON.

Access to a security policy is guarded by a two-stage access control mechanism using a *certificate* and a *policy board* (see Figure 5.4). One can define the access control and security policy in such a way that only applications under the control of the security policy can gain access to the secrets. In this way, one can prevent any stakeholder from accessing the secrets.

Secrets can be passed to applications as command line arguments, environment variables, or can be injected into files. The files can contain PALAEMON variables referring to the names of secrets defined in the security policy. The variables are transparently replaced by the value of the secret when an application that is permitted to access the secrets reads the file. By *transparently*, we mean that the application is not aware of the replacement and its code does not need to be modified.

Secret management is supported through *security policies*, whose general structure is shown in Figure 5.4. Each policy has a unique name and can define: *(a)* the permitted MRENCLAVE of an application (several MRENCLAVEs can be specified

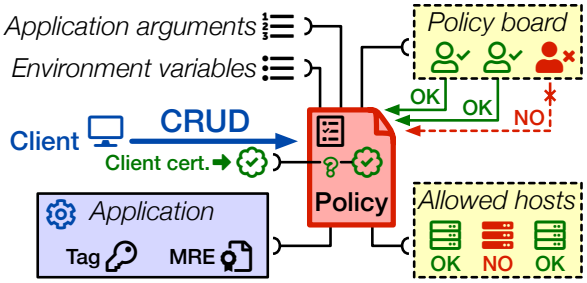


Figure 5.4. Overview of security policies.

to facilitate software updates); *(b)* the set of permitted platforms on which the application is permitted to run, or none if permitted to run on any platform; *(c)* the *key* and *tag* of the file system (the *tag* is a secure hash across all files, which are transparently en/decrypted with the *key* inside the Intel SGX enclaves); *(d)* the command line arguments; *(e)* the environment variables; *(f)* a set of files to inject secrets into; and *(g)* imports/exports of secrets from/to other policies.

5.4.1.1. Managed PALAEMON

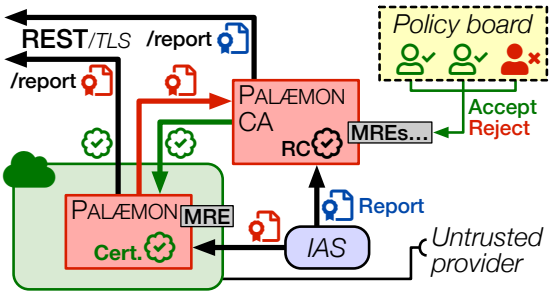


Figure 5.5. Principle of managed PALAEMON deployment and operation.

Our objective is to support a feature that we can delegate the management of a PALAEMON instance to an untrusted party, e.g., a cloud provider, while the clients of PALAEMON can still trust that their secrets are safe and well protected. Note that the cloud provider has full control over what code it executes and might try to run variants of PALAEMON that are wrongly configured or have modified code. We ensure that clients connecting to a PALAEMON instance can attest it, i.e., they can verify that this instance runs the expected unmodified PALAEMON code. Moreover, this code does not support any configuration options that negatively influence the confidentiality, integrity and freshness of client data stored in the instance.

We support two ways to attest a PALAEMON instance (see Figure 5.5): *(i)* using TLS [19, 51]; and *(ii)* with explicit attestation. The TLS-based attestation requires a trusted CA with a known Root Certificate (RC). The CA first attests the PALAEMON instance using approach *(ii)* to ensure that this instance runs inside a SGX enclave and has a correct MRENCLAVE. Only then will the CA provide the instance with a certificate signed with the RC. The CA itself runs inside of a SGX enclave and can be attested using explicit attestation. Entities that trust the CA can attest the instance by checking that its TLS certificate is signed by the RC.

To support software updates of PALAEMON itself, the CA includes a set of correct MRENCLAVE. The CA only signs certificates for these MRENCLAVE and also

limits the duration of the certificates to ensure timely upgrades to new versions of PALAEMON. The set of MRENCLAVE is stored inside of the CA's binary, i.e., an adversary cannot modify the set without invalidating the MRENCLAVE of the CA. Hence, deploying a new version of PALAEMON requires first to deploy a new version of the CA.

Clients might not trust the CA if they do not use the current set of valid MRENCLAVE, e.g., they only trust code instances that have been deployed some time ago, or are not represented in the PALAEMON policy board. These clients need to attest the PALAEMON instance in the same way that the CA attests instances, as described in § 5.4.1.4. In practice, any updates of PALAEMON must be approved by all stakeholders.

5.4.1.2. Robust Root of Trust

Our threat model permits Byzantine behaviour of stakeholders like software developers and system administrators. Any change to an application or its configuration can impact the confidentiality, integrity and freshness of both data and code. PALAEMON therefore includes a mechanism to ensure that any security policy modification must be approved by at least $f{+}1$ stakeholders. Here, we assume that we have a set of n stakeholders and a threshold f (with $f{<}n$), such that $n{-}f$ stakeholders can be trusted at any point in time, i.e., at most f of them exhibit Byzantine behaviour because of neglect or malicious intent. Therefore, if at least $f{+}1$ stakeholders approve a change at least one of them judged it to be trustworthy. To that end, a securities policy can define a *policy board* and a threshold—typically set to $f{+}1$—of policy board members that must give approval for PALAEMON to permit any create, read, update and delete (CRUD) access to the policy. Upon creation, the board of the new policy must also approve the operation. In that way, any client can create policies as long as they have unique names, and the policy board agrees to take control over them.

Each policy board member is represented in the security policy by a *certificate* and a URL of an *approval service*, responsible to *approve* or *reject* accesses to the policy. Upon a client access, PALAEMON contacts the board members, verifies their certificates and asks them for approval of the request via a TLS-secured REST call to their approval service.

Approval services typically run inside Intel SGX enclaves. In case the associated board member is a person, they should perform a two-factor authentication with one being based on biometric identifiers. Approval services may also consist of services that check certain aspects of a policy, e.g., through source code analysis and verification of the MRENCLAVE. In particular, a policy board member could be an organisation that validates software, i.e., perform checks on behalf of their clients to ensure that the software associated with a certain MRENCLAVE can be trusted to protect the confidentiality, integrity and freshness of data.

Some policy board members can be given *veto* rights, i.e., they can unilaterally reject a policy change. For example, a data provider might only provide data to applications for which it is a policy member with veto rights. In that way, the data provider can ensure that policy changes will not result in data leakage.

5.4.1.3. Application Attestation and Configuration

Upon startup, an application is transparently linked with the SCONE runtime and loaded inside a SGX enclave. The SCONE runtime first attests the application with the help of PALAEMON before passing control to the application. To do so, it creates a random key pair and gets a *report* from a local *quoting enclave* [27] that associates the public key with its MRENCLAVE. The runtime sends the report via a newly-established TLS connection to PALAEMON and passes along the name of its security policy which is stored in an unprotected environment variable. The PALAEMON instance verifies that: *(i)* the public key of the TLS client certificate matches the public key of the report; *(ii)* the security policy name exists and the MRENCLAVE is valid for the application; and *(iii)* the application runs on a permitted platform—which we can verify with the report. If this attestation succeeds, PALAEMON sends the following data to the application: the command line arguments; the environment variables; the keys and tags for the file system; and the set of files in which secrets should be injected together with the secrets as key/value pairs.

The PALAEMON runtime supports transparent injection of secrets into existing configuration files via a simple variable replacement mechanism. This allows us to inject different secrets into different instances of the same application image, without the need to change the source code. Like all files, they can be confidentiality, integrity and freshness-protected via transparent encryption by the PALAEMON runtime. The runtime injects the secrets it received from the PALAEMON instance in each file as follows. The file is first read in enclave memory, then parsed, and all variables found are replaced by their values. Whenever the file is accessed, it is served from memory. While sizeable files can also be stored encrypted in main memory or on the file system, configuration files are typically small, so we keep them in enclave memory as long as they fit.

5.4.1.4. PALAEMON Attestation

A client of a managed PALAEMON instance must be able to ensure that the code of PALAEMON was not modified and indeed runs inside of an Intel SGX enclave. As a matter of fact, we must guarantee that an infrastructure provider cannot configure PALAEMON in any way that breaks the trust given by the client in PALAEMON. We enforce this by designing PALAEMON for its behaviour to depend solely on MRENCLAVE, i.e., PALAEMON has zero configuration parameters that affect its behaviour with regard to ensuring the confidentiality, integrity and freshness of the data stored in the instance by the clients.

A client connecting to a PALAEMON instance has to attest the instance before performing any action, such as creating a new security policy. During the initial startup, a PALAEMON instance creates a unique public/private key pair, as well as a random key to encrypt its file system, and stores these keys in sealed storage [22]. During a restart (after an exit or a failure), the instance reads the keys from sealed storage to be able to authenticate itself. We actually use SGX to enforce that only PALAEMON instances on the same platform can read the sealed file.

When the PALAEMON instance starts up, it attests itself via IAS [2, 26]. On a successful attestation, it gets a report from IAS that associates its MRENCLAVE with

its public key. The instance can send this report to the PALAEMON CA to obtain a certificate for the public key mentioned in the report. Clients that connect to the instance via TLS are served this certificate after successful verification of the certificate by the instance. The clients can then verify the instance via TLS by ensuring that the certificate is signed by the PALAEMON CA. Alternatively, clients can request the IAS report via a REST API provided by PALAEMON, and then verify that the report: *(i)* was indeed signed by IAS, and *(ii)* associates the PALAEMON MRENCLAVE with the public key of the certificate. At that point, the clients know that the instance runs inside of a SGX enclave and has the correct MRENCLAVE, i.e., they can now safely send requests.

Note that clients might themselves run inside a SGX enclave and obtain the permitted MRENCLAVE from their security policy. Moreover, they might be limited to connecting only to certain PALAEMON instances identified by their public keys.

We implemented PALAEMON based on our toolchain SCONE [3] running on top of Intel SGX [14]. However, note that we designed PALAEMON in a generic way that can be used not only for SCONE but also for other SGX platforms such as Graphene [46]. In addition, we used Rust [32] to implement PALAEMON since it ensures strong type safety. We use an encrypted embedded SQLite [1] database running inside the same enclave as PALAEMON.

5.4.2. Evaluation: Micro-benchmarks

We evaluate PALAEMON using both micro- and macro-benchmarks (see §5.4.3). All our experiments are executed on a rack-based cluster of Dell PowerEdge R330 servers. Each machine is equipped with an Intel Xeon E3-1270 v6 CPU and 64*GB* of RAM. The machines are connected to a 20*Gb/s* switched network. SGX is statically configured to reserve 128*MB* of RAM for the EPC [14]. We use Ubuntu 16.04 LTS with Linux kernel v4.13.0-38. The CPUs use the latest microcode patch level.

The underlying SCONE runtime, beside hardware mode (HW) running with Intel SGX, it also supports emulation mode (EMU) to run legacy applications without any Intel SGX support. We conducted experiments also with EMU mode to highlight the performance overhead of the Intel SGX.

Attestation and Configuration. First, we evaluate how long it takes to attest and configure an application. The advantage of PALAEMON over the traditional way using IAS to perform attestation is that PALAEMON runs on the local cluster. We measured the time it takes to perform the individual steps of remote attestation (see § 5.4.1.4). The IAS experiment ran on servers in Europe and in Portland, OR, USA (close to IAS servers).

Figure 5.6 shows the time it takes to: *(i)* initialize the necessary resources; *(ii)* send the quote to PALAEMON; *(iii)* wait for PALAEMON to confirm the successful attestation; and *(iv)* receive the configuration.

The initialization phase includes key pair generation, DNS resolution, connection establishment, and TLS handshake with PALAEMON. Overall, the initialization time is similar for each attestation service and is dominated by the TLS handshake.

Obtaining and sending the quote takes longer for IAS variants for two reasons.

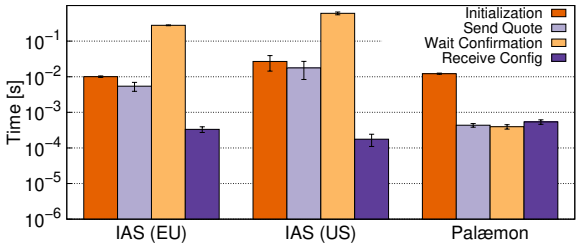


Figure 5.6. Attestation and configuration latencies: even when located close to Intel's IAS server, attestation with IAS takes about an order of magnitude longer than with PALAEMON.

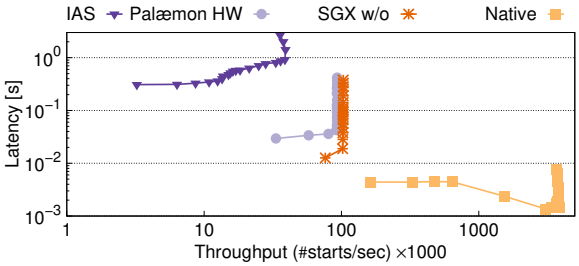


Figure 5.7. Startup latency and throughput using attestation variants.

First, performing IAS attestation requires providing information that is embedded into the generated quote, which adds one round trip. Second, PALAEMON attestation cryptography (Ed25519 [6]) is less expensive than the one used by IAS (EPID [9]). However, the dominating factor for IAS is the time spent waiting for the attestation. PALAEMON has to verify the quote either by querying the IAS or by verifying the signature and looking up the public key of the QE. Overall, PALAEMON attestation takes around 15*ms* to complete, which is an order of magnitude faster than IAS attestation which takes 280*ms* when performed from the USA, or 295*ms* from Europe.

Our benchmark starts multiple minimal programs in parallel to measure the startup throughput and latency. Figure 5.7 depicts the latency and throughput for different attestation variants. In the *Native* case (SGX and attestation are not involved), the throughput scales well until all eight hyper-threads are fully utilized. At this point, the system runs around 3,700 programs every second. If the program is compiled with SGX but without attestation (*SGX w/o*), the throughput drops to about 100 executions per second. This variant does not scale well with increasing parallelism. We tracked down the bottleneck to the Intel SGX driver synchronising EPC page (de)allocations with a single lock. Since every enclave has to obtain EPC pages at roughly the same time, this lock basically enforces page requests to be served sequentially.

With *IAS* and *PALAEMON*, the startup routine performs remote attestation before executing the actual program. With PALAEMON attestation, we quickly reach the maximal achievable start rate of about 90 runs per second. IAS attestation needs a considerable amount of parallelism to partially hide the higher latency, reaching about 40 runs per second (60 parallel instances) at 1.4*s* latency.

Secret Injection Latency. We measure the impact of injecting secrets in a file by an application running inside an enclave. To that end, we read a $4kB$ file in

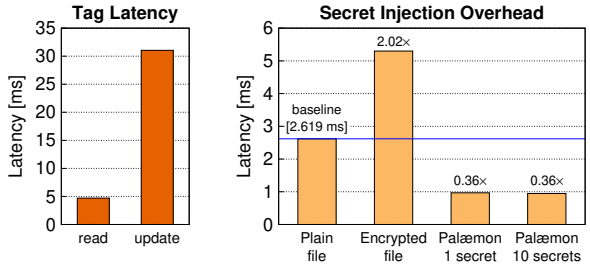


Figure 5.8. Left: latency of PALAEMON tag reads and updates. Right: reading overhead for a file with 1 or 10 secrets normalized by the time to read a plain file.

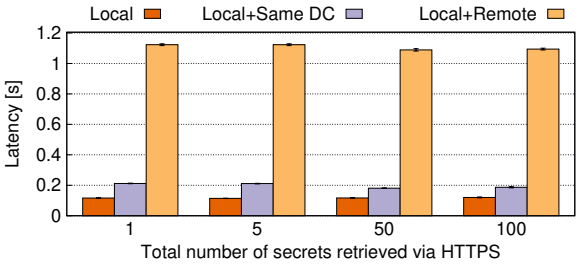


Figure 5.9. Latency to retrieve multiple secrets (up to 100) from a PALAEMON service deployed locally, from the same data centre (DC) or from an instance running on a different continent.

which we inject 1 and 10 secrets (Figure 5.8 right). We show the latency as well as the overhead compared to the baseline on top of each bar. PALAEMON achieves better latencies for files with injected secrets—even compared to the *plain file* baseline—because the secrets are injected during startup and stay in enclave memory.

Secret Access Latency. PALAEMON supports the retrieval of keys from remote PALAEMON services. We measure the overhead of retrieving local and remote secrets, i.e., when using PALAEMON in a decentralized fashion (Figure 5.9). There is no visible increase in latency when retrieving 1, 5, 50 and 100 keys of 32*bytes*. As a matter of fact, retrieving 50 or 100 keys consistently outperforms 1 or 5 keys. However, there is an impact if a peer service is located on a different continent instead of the same data centre. This is mainly caused by the time it takes to establish of a TLS connection.

5.4.3. Evaluation: Macro-benchmarks

Our macro-benchmarks run using real-world systems the NGINX web-server, the ZooKeeper distributed coordination service, and the MariaDB database server (a fork of MySQL) All these systems benefit from PALAEMON for additional security guarantees; we evaluate its impact on performance.

NGINX. Along the same lines, we use an encrypted NGINX [38] container image and rely on PALAEMON to: *(i)* encrypt all the files; *(ii)* inject the certificates; and *(iii)* inject private keys used by NGINX for TLS termination. The benchmark issues GET requests on 67*kB* files (nowadays' average size of an HTML web page [29]) with the `wrk2` tool (see Figure 5.10 (a)). We see that the overhead of SGX alone is less pronounced than that of encrypting all files. Tuning the caching done by NGINX could improve the performance when encrypting files. There is little

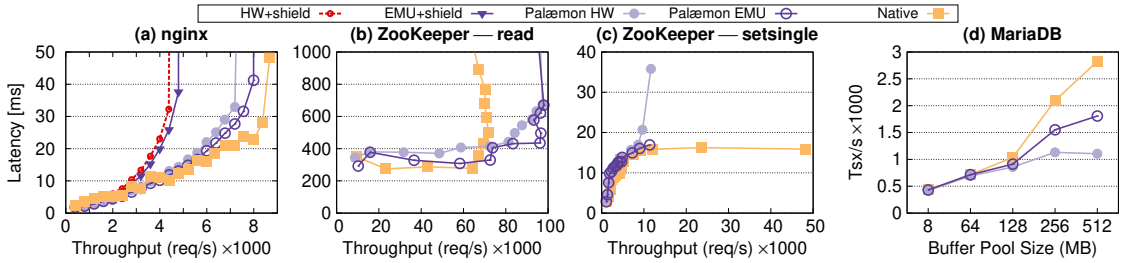


*Figure 5.10. (a) Throughput/latency for GET requests on 67*kB *files, in five variants of* `nginx`*. ZooKeeper: read (b) and write (c) operations. (d) MariaDB with TPC-C benchmark (d): increasing buffer pool helps native more than EMU or hardware.*

difference between running in emulation mode and inside of an SGX enclave, since not much paging is taking place.

ZooKeeper. Next, we evaluate the overhead of PALAEMON with the ZooKeeper coordination service. We deploy a cluster of three nodes and evaluate three ZooKeeper variants: *(i)* native using `stunnel` [49] for TLS termination between servers; *(ii)* shielded ZooKeeper running together with the JVM in hardware mode; and *(iii)* EMU mode.

We use the ZooKeeper Benchmark [30] to measure read and write throughput. The read throughput of the shielded versions is consistently better than the native one (Figure 5.10 (b)). The write throughput (Figure 5.10 (c)) exhibits better performances in native mode, as it involves the execution of consensus [23] via TLS, resulting in more code and system calls being executed. Our results are on par with *SecureKeeper* [8], despite its use of an encryption proxy to protect the content only.

MariaDB. We conclude our macro-benchmarks by measuring the throughput of MariaDB configured to perform encryption at rest [31]. We use PALAEMON to inject a generated X.509 certificate, the private key and the encryption key. We execute the TPC-C benchmark [45] and vary the available buffer cache. Figure 5.10 (d) presents experimental results. For small buffer pool sizes [34], i.e., $<128MB$, all configurations behave similarly since the main overhead is hardware I/O. For larger buffer caches, EPC paging increases in hardware mode. Hence, adding more buffer cache reduces the throughput while it increases the throughput in emulation and native mode. A fair comparison with the recently proposed EnclaveDB [37] system is currently not possible since it lacks paging support and its performance figures are only based on simulations.

6. Runtime Support for Application Development

6.1. OmpSs@Linter as a debug tool

We have completed the implementation of the Linter tool for OmpSs-2 applications. Changes applied to OmpSs-2 include:

- New instrumentation variant. We have included a new instrumentation option to Nanos6, to subscribe to some task-level events of interest related to the Linter tool, mainly related to task management.

- New Mercurium directive. We introduced an OSS *lint* directive and a new *verified* clause to the OSS task directive.
- New Mercurium analysis pass. We also enriched the Mercurium built-in infrastructure for static analysis with a new algorithm that analyzes the source code of an OmpSs-2 program and annotates it with the verification annotations.

6.1.1. OmpSs Linter

The Linter is a run-time tracing tool based on a dynamic binary instrumentation tool built on top of Intel's PIN [25]. It takes as input an application parallelized using OmpSs-2, and it provides a report of all parallelization issues that are encountered by tracing the application. When an OmpSs-2 application is run through this tool, memory accesses issued by tasks are recorded and temporarily saved to a storage area. For each task, the recorded memory accesses are later processed at task completion time and compared with task information (e.g., dependences) to check for potential parallelization errors. For each of such errors, a warning is generated to report to the user about the problem. The report comes with additional contextual information:

- address and size of the mismatching memory access (if any) along with its access mode (i.e., read or write);
- name of the involved dependency (if any) with the expected directionality (in, out, inout);
- variable name (if found);
- task invocation point in the source code (i.e., line in the respective file).

The tool operates at two different levels of abstraction: (1) the abstraction level provided by the OmpSs-2 programming model to deal with task and dependences, and (2) the abstraction level provided by the target Instruction Set Architecture (ISA) to recognize accesses to memory, which in our case is AMD64. As illustrated in Figure 6.1, the target program is composed of the actual OmpSs-2 application and, if available, debugging information (i.e., symbol table and DWARF sections). The target program interacts with Nanos6 to execute in parallel on the available cores. When OSS directives are translated by Mercurium into calls to Nanos6, these calls invoke task-based primitives which update the internal execution state of each task. Internally, Nanos6 maintains a state machine for each task to keep track of its execution state over time. Nanos6 provides an Instrument API to subscribe to state transitions in the task state machine and perform custom actions. Our instrumentation tool is composed of three main components: the PIN virtual machine (VM) that performs dynamic binary instrumentation and two modules that perform memory access tracing of the binary executable. The frontend module is dedicated to intercepting the accesses performed by the application at run-time and generating the actual traces, while the backend module is responsible for the processing of traces and generates the final user report. Our tool interacts with the rest of the software as follows. It executes the original application via the Pin VM. Events of interests at the ISA

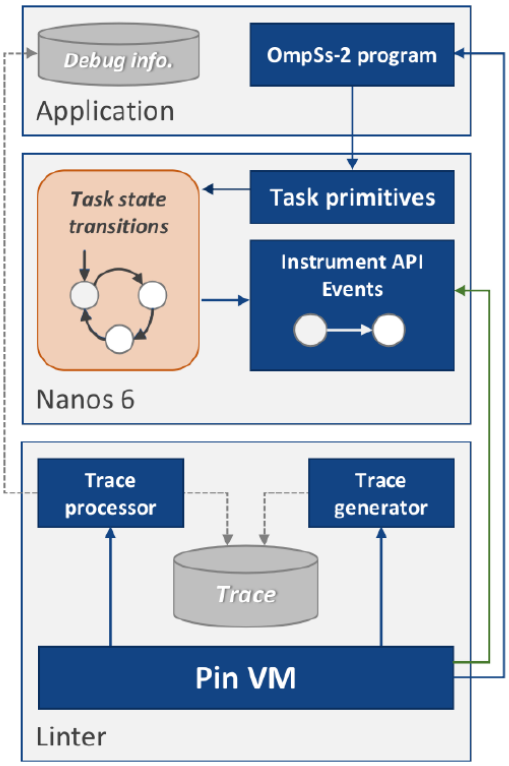


Figure 6.1. OmpSs@Linter environment

abstraction level (i.e., memory reads and writes) are intercepted via the PIN VM itself, which gives control to the trace generator module. Out tool also intercepts events of interests at the OmpSs-2 programming model level via Nanos6's Instrument API; these events are: when a task is created or destroyed, begins or ends, is put to wait via a full or partial synchronization, or when dependences are available or released. When one of these events occurs, the PIN VM once again gives control to the frontend module. The backend module is invoked when a task completes execution. It loads traces from the storage area and combines them with information coming from Nanos6 via the Instrument API to detect parallelization errors. To provide contextual information, each entry of this report is further combined with information coming from the available debug metadata, if any, in the executable file. To date, the tool supports all core functionalities of OmpSs-2 shared with TMP, plus the manual release of dependences, the if and final task attributes, commutative and concurrent dataset entries. Other advanced parallelization constructs are not supported, meaning that all errors resulting from the use of these directives cannot be detected. Despite that, the resulting restricted OmpSs-2 model is nevertheless general enough to support a wide range of real-world applications and use cases, as shown in the next section.

Internally, local task analysis is implemented using binary search tree operating on intervals of memory addresses. These intervals can be easily derived from dataset entries, due to the way OmpSs-2 enables the user to specify as dataset entries entire C, C++, Fortran objects, or portions of it (included arrays). As for task accesses, our tool implements an aggregation operation that merges all the

accesses coming from the same instruction and involving consecutive memory accesses. While this aggregation has a detrimental effect on performance when it comes to building the access-set of a task, it speeds up the subsequent local task analysis. Indeed, checking for the different errors becomes a matter of intersecting different binary search trees, looking for overlapping intervals, and checking for the conditions 1 to 3. All intervals without found matches are reported as an error. The case of E3 errors is slightly more complex as it involves barriers. Our approach for condition 3 was to maintain an expiry clock for all dataset entries of all child tasks of a task. The expiry clock is set as soon as we exit from a barrier. If it is a barrier with no dataset, all the child dataset entries are set as expired. Otherwise, we only set as expired the dataset entries that match with the entries in the barrier. Accesses are then compared with the child dataset entries that are not expired at the time each access is performed.

6.1.1.1. The lint directive

The lint directive is used inside task code to avoid analyzing the code wrapped by the directive. This is useful to mark code that has no effect on the dataset of a task. For example, we can mask library functions performing I/O operations, or exclude initialization and finalization code in a task that is known to only access private memory. The directive also allows specifying which memory operations are performed within a marked region of code, and to which memory addresses. The directive accepts three main clauses: in, out, inout. They are equivalent to those specified for a task dataset and allow the user to state which are the shared data objects read and/or written within the wrapped code. The local task analysis can use these data-references as a hint of the accesses that were performed within the code so that it does not have to derive them by itself. This brings many benefits in terms of accuracy and performance of the analysis.

Consider the code cases depicted in Figure 6.2. In the first case (i), we are using the oss lint directive to instruct the analysis that an allocation has been performed. At the same time, we are also disabling the analysis within the malloc function, which is probably a good idea given that its implementation can be arbitrarily complex and can involve accesses to shared objects (e.g., locks) defined within the library that implements it. The second case (ii) is analogous to the first, but memory is released. The analysis is also disabled, so that whatever happens within the call to free() will not be considered during the analysis. The third example (iii) describes a very common case in which we are using some API functions to read and write buffers in memory via I/O operations. It is what happens, for example, with the MPI_Send and MPI_Recv functions. According to the textual description of these functions, and regardless of their implementation, they are respectively reading/writing N bytes from/to memory. Not only the implementation of these functions can be arbitrarily complex so as to slow down the analysis by orders of magnitude, but it can also affect the accuracy of the generated report negatively. Indeed, the synchronization mechanisms used within these functions are independent of the OmpSs execution model and, as such, may require to access objects that are not (and should not be) declared as dependences in the task where these functions are invoked. The fourth case (iv) is that of nested loops where the final effect is that of traversing a well-defined portion of an array for reading and/or writing. By marking this code

```
(i)
#pragma oss lint inout ( x [0: n ])
   x = malloc ( n * sizeof ( int ));

(ii)
#pragma oss lint out ( x [0: n ])
   free ( x );

(iii)
#pragma oss lint in ( sendbuf [0: size ]) out ( recvbuf [0: size ])
{
   MPI_Send ( sendbuf , size , MPI_BYTE , dst ,
                                block_id +10 , MPI_COMM_WORLD );
   MPI_Recv ( recvbuf , size , MPI_BYTE , src ,
              block_id +10 , MPI_COMM_WORLD , MPI_STATUS_IGNORE );
}

(iv)
double A [ N / TS ][ M / TS ][ TS ][ TS ];
#pragma tmp lint out ( A [ i ][ j ])
   for ( long ii = 0; ii < TS ; ii ++)
      for ( long jj = 0; jj < TS ; jj ++)
         A[i][j][ii][jj] = value;
```

Figure 6.2. Examples of use of the lint directive

with the pragma and summarizing its behavior, we are saving the cost of analyzing a number of accesses that are proportional to the number of iterations. The analysis would still correctly detect all the accesses performed within the loop without any accuracy concern, but the overall execution time would be much greater due to the cost of analysis.

6.1.1.2. The verified clause

Users are also offered another way to disable analysis at the level of whole tasks, via the verified keyword that can be passed to the oss task pragma. Conceptually, the effect of the verified flag is equivalent to wrapping the entire task code within a lint pragma, specifying as dataset the same data-references used for the task pragma. Performance-wise, the verified attribute is equivalent to wrapping the entirety of task code with a lint pragma, because the analysis is disabled for the whole task. The verified task also comes with an optional expression that evaluated at run-time to decide whether memory tracing for the task will be disabled or not. This expression can be used to conditionally evaluate tasks that are more likely to be subject to programming errors, for example, tasks that are related to boundary loop iterations. Additionally, it can be used as a way to implement task-level sampling and reduce the overall memory tracing overhead of the application (e.g., instrument one out of N task instances).

Figure 6.3 shows an example of the use of the *verified* clause. In the first task, *verified* states that the task data accesses do not need to be checked, as the complete code in the task is assured to be correct. The *verified* clause in the second task takes an expression. If the the expression is true, the task code

```
int x;
...
for (int i = 0; i < N; i++) {
   #pragma oss task inout (x) verified
   {
      ...
   }

   #pragma oss task inout (x) verified(x < 5)
   {
      ...
   }
}
```

Figure 6.3. Example of use of the verified clause

does not need to be checked. This characteristic can be used to check only corner cases.

6.1.2. Evaluation: Benchmarks

The evaluation of the Linter tool has been carried out on six typical kernels (matmul, dot-product, multisaxpy, mergesort, cholesky, and nqueens), and three applications (nbody, heat, and HPCCG). We describe the codes below:

6.1.2.1. matmul

This benchmark runs a matrix multiplication operation C = A · B, where A has size N × M, B has size M × P, and the resulting matrix C has size N × P. Parallelization is achieved using block partitioning with block size TS. The problem size for this benchmark is given by M × N × P . The degree of parallelization for this benchmark is given by TS. In our experiments we use M = N = P = 128, and TS = 8.

6.1.2.2. dot-product

The dot-product takes two equal-length N vectors and returns a single scalar. Parallelization is achieved using block partitioning with block size TS. The problem size for this benchmark is given by N. The degree of parallelization for this benchmark is given by TS. In our experiments we use N = 8192 and TS = 64.

6.1.2.3. multisaxpy

This benchmark runs several SAXPY operations. SAXPY stands for “Single-Precision A·X Plus Y”. It is a Level 1 operation in the Basic Linear Algebra Subprograms (BLAS) package and is a typical operation in computations with vector processors. Each SAXPY operation solves the equation Y = A · X + Y , where X and Y are two vectors of size N , and A is a scalar value. Parallelization is achieved by block partitioning with block size TS. The problem size for this benchmark is given by N. The degree of parallelization for this benchmark is given by TS. A further parameter I specifies the number of times the SAXPY problem will be solved. In our experiments we use N = 4194304, TS = 1024, and I = 1.

6.1.2.4. mergesort

This benchmark runs a Merge Sort operation. It recursively halves an unsorted vector X and sorts these chunks, until it gets to a sorted vector Y. Parallelization is achieved via a divide-and-conquer approach, which relies on a maximum chunk size TS. This means that all chunks will a size lower than TS will not create new tasks. The problem size for this benchmark is given by N. The degree of parallelization for this benchmark is given by TS. In our experiments we use N = 65536 and TS = 512.

6.1.2.5. cholesky

This benchmark runs a Cholesky decomposition over a square matrix A of side N. The code uses the CBLAS and LAPACK Linear Algebra libraries. Parallelization is achieved using tiling, which relies on block partitioning with block size TS. The problem size for this benchmark is given by N. The degree of parallelization for this benchmark is given by TS. In our experiments we use N = 16384 and TS = 128.

6.1.2.6. nqueens

This benchmark computes, for a N × N chessboard, the number of configurations of placing N chess queens in the chessboard, such that none of them is able to attack any other. It is implemented using a Branch-and-Bound algorithm. A sub-problem consists of checking if the current queen, placed at coordinates (i, j) on the chessboard, can be attacked by any of the existing M < N of queens already placed. Further search on each sub-problem is stopped as soon as an attack from one of the existing M queens is found. Otherwise, the current queen is added to the solution. If it was the N-th queen, the current solution is a valid solution to the N-queens problem. Placements are evaluated from left to right, top to bottom. Parallelization is achieved by spawning an independent task for each sub-problem until we reach the j-th column. The rest N - j columns will not generate any tasks and will be executed serially. The input to the problem is the chessboard size N and the column threshold TS. Therefore, the problem size for this benchmark is given by N. The degree of parallelization for this benchmark is given by TS. In our experiments we use N = 10 and TS = 4.

6.1.2.7. nbody

An N-body simulation numerically approximates the evolution of a system of bodies that interact with each other. It has applications in many scientific fields: astrophysical simulation, protein folding, and turbulent fluid flow simulation, to name a few. This application makes use of MPI for coarse-grained parallelization. Both computation and communication phases are taskified. However, communication tasks are serialized to prevent deadlocks between processes since communication tasks perform blocking MPI calls. The input to the problem is the number of interacting particles N and the number of iterations I to compute the interactions between them. The problem size for this benchmark is given by N. The degree of parallelization for this benchmark is given by TS, which is the number of particles that are handled in parallel by each thread of each process, and R, which is the number of processes. In our experiments we use N = 4096, TS = 64, and I = 1.

6.1.2.8. heat

The Heat simulation uses an iterative Gauss-Seidel method to solve the heat equation, which is a parabolic Partial Differential Equation (PDE) that describes the distribution of heat (or variation in temperature) in a given region over time. Just like N-body, the application uses MPI for coarse-grained parallelization. The input to the problem is the size along one dimension of the grid used to compute heat N and the number of iterations I. The problem size for this benchmark is given by N. The degree of parallelization for this benchmark is given by TS, which is the block size along one dimension, and R, which is the number of processes. In our experiments we use N = 4096, T S = 64, and I = 1.

6.1.2.9. HPCCG

Solves the equation A · X = b, where A is a large sparse matrix, and B and X are vectors such that X is the unknown. The problem is discretized with a finite difference scheme on a 3D rectangular grid domain and solved via a preconditioned conjugate gradient method. The input to the problem is Nx, Ny, and Nz , which are the sub-grid dimensions in the 3D space assigned to the different processes. Parallelization is achieved by assigning each sub-grid to a different process and by using block partitioning based on the number of threads for each process. Therefore, the problem size for this benchmark is given by Nx × Ny × Nz × R, where R is the number of processes. The degree of parallelization for this benchmark is given by R and T, which is the number of threads per process. In our experiments we use Nx = 50, Ny = 150, Nz = 50.

6.1.3. Evaluation: Results

All the experiments have been executed on the Marenostrum4 supercomputer. Each compute node is equipped with two Intel Xeon Platinum 8160 CPUs with 24 cores each, thus totaling 48 cores per node, and 96 GB of main memory. The interconnection network is based on 100 Gbit/s Intel Omni-Path HFI technology. The MPI benchmarks are run on 4 nodes, while the other benchmarks are run on a single node.

Figure 6.4 shows the slowdown and the absolute execution time (in seconds) for the benchmarks described above. Each bar represents a different benchmark and a different evaluation experiment. Benchmark names with no lint or autolint suffix are run without the aid of the static analysis tool and without using the OSS lint directives.

Benchmark names with the lint suffix are run with the OSS lint directives to annotate third party libraries. Only *cholesky* and the MPI applications use this technique, in order to annotate the calls to the Intel MKL and Intel MPI third-party libraries.

The autolint suffix represents the case of running the benchmark with the aid of the static analysis tool implemented in Mercurium, which in turn exploits the existing OSS lint directive, thus showing the effectiveness of the analysis pass implemented in Mercurium.

For each bar, we also report the slowdown in the execution time cost, split into: (green) the minimum instrumentation cost to run the application using PIN (the

base case in the legend); (orange) the minimum instrumentation cost to instrument memory instructions, even without processing them (the instr case); (blue) the full analysis cost (the full case in the legend).

Overall, the mechanism works in three phases. Initially, users can annotate the portions of code that they know for sure are correct with the proposed directives and clauses (see subsection 6.1.1.1 and 6.1.1.2). Then, the compile-time compiler pass analyzes the code with the objective of verifying parts of code. At this stage, the compiler can take different actions: (a) any issue detected is reported to the user prior to execution, (b) the parts of the code that cannot be analyzed statically are left for instrumentation, and (c) the parts of code that are analyzed and decided to be correct, are verified using the same directives and clauses offered to the user with such a purpose. Finally, a run-time tool executes the code and instruments only those parts that have been verified neither by the user nor by the compiler. As we can see from the figure, the slowdown for the pure runtime instrumentation case (no suffix) can be quite high for some benchmarks (e.g., dot-product or mergesort). In the case of cholesky, the overhead is quite high due to the heavy use it makes of Intel's MKL library. However, we note that even so, the absolute execution time is in the order of minutes, thus not undermining the usability of the tool-chain. We ran a breakdown analysis of these cases and detected the major source of overhead to be the insertion of accesses in our binary search tree, which is used to aggregate contiguous accesses coming from the same instruction over time, and to compare them with task dependencies. In the case of nqueens, another major source of overhead is that of recording the issues encountered in the application, which still uses a binary search tree.

The other two sources of overhead are the instrumentation cost depicted in the *base* and *instr* cases. This is not only due to the way PIN internally works, but also to the nature of each application. In fact, we already disable instrumentation whenever the application performs calls to the OmpSs-2 run-time system or the standard C/C++ libraries. Additionally, we disable the instrumentation of private-memory instructions (such as stack instructions) in serial regions of the application, where there cannot be issues.

When using external libraries protected by the directive OSS lint directive, the improvements in terms of slowdown can be critical. By appropriately marking those calls with the new directives, the run-time instrumentation tool only needs to instrument the OSS lint directive itself, and store a number of accesses that is proportional to the intervals specified in the in, out, and/or inout parameters of the pragma itself. This is critical for the case of cholesky, as each task only performs a single call to a different function in the MKL library, but these calls internally hide a huge number of accesses to memory that are the primary source of overhead for the instrumentation. Performance improvements can also be observed for the case of MPI benchmarks, which use the Intel MPI library, although the impact is not so important. For example, while heat is communication-intensive and so protecting calls to MPI is highly effective, nbody and HPCCG are computation-intensive, so the use of the directive does not improve the execution overhead by much.

Using OSS lint directives and leveraging the compile-time analysis tool brings the most evident benefits, as it can be seen for matmul, dot-product, and mul-

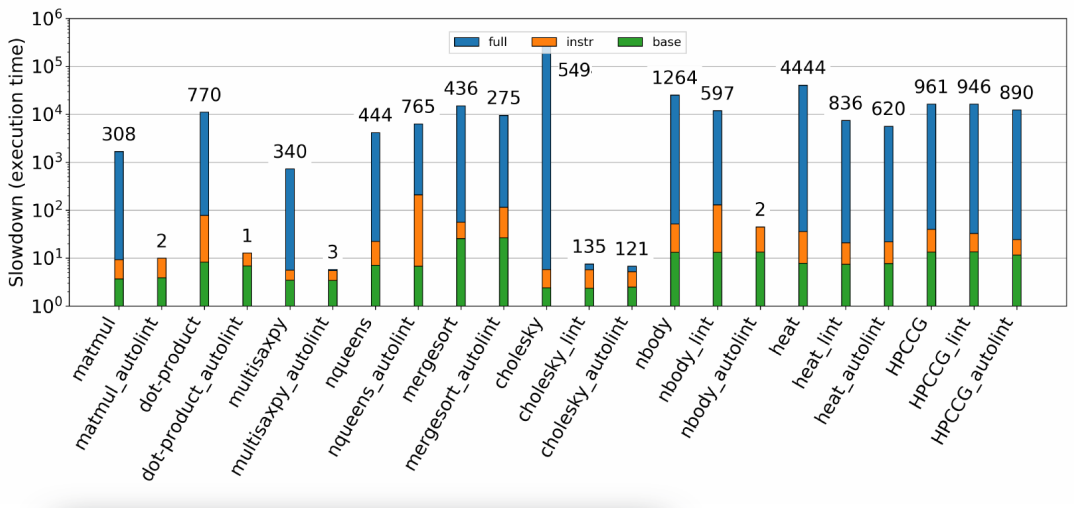


Figure 6.4. Evaluation of the overhead (slowdown in the execution time) for the benchmarks executed under OmpSs@Linter

tisaxpy. In this case, the compile-time tool can automatically wrap whole for-loops into directives, or even insert the verified flag to tasks within loops. In all these cases, the performance improvements are drastic because the instrumentation tool can disable tracing for most of the execution time of the application. We note that these improvements are not uncommon for real-world scenarios, as many kernels have a regular loop structure.

As for nqueens, we observe that the static tool is unable to infer useful information due to the nature of the code and can only wrap in directives simple memory-accessing statements. This has an effect in our run-time tool that is actually worse than not placing them, thus suggesting that additional work can be put in the static tool to avoid wrapping simple statements, and in the run-time tool to reduce the overhead of processing OSS lint directives. In mergesort, the tool can only partially simplify the handling of accesses in the merge phase of the algorithm because the number of iterations that are performed in each part depends on the contents of the two sub-arrays to merge. For this reason, the overhead improvements are minimal. Similar considerations can be made for the MPI benchmarks and especially for HPCCG, where the main kernel performing an MKL-like dgemm operation could not be annotated at all due to the fact that a sparse matrix representation is used. As for cholesky, we observe that each task only performs a single call to an MKL library function, so the compile-time tool can successfully promote the existing oss lint pragmas to the verified clause at the level of tasks, but this brings little additional benefits compared to the lint case.

Overall, the experimental evaluation suggests that the absolute execution cost of running the selected applications against the run-time tool is affordable and that the synergistic exploitation of static analysis and pragmas can drastically reduce this cost.

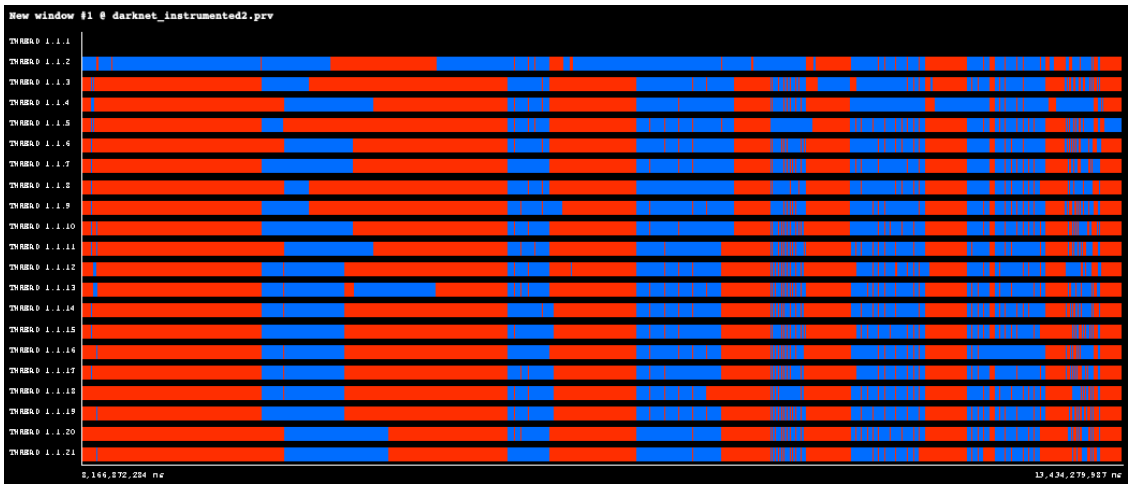


Figure 6.5. VGG traces for two events.

6.2. Extrae support in XiTAO

Extrae is a dynamic instrumentation tool to trace programs compiled with parallel programming models. The traces are visualised using paraver. Extrae has a built-in support for state-of-the art parallel programming models such as OpenMP, pthreads, MPI and CUDA. Extrae library can also be used to trace events of interest in our applications. We added Extrae support in XiTAO to trace events of interest. Figure 6.5 shows the execution of VGG code with 20 threads. We collected traces for two events; Task stealing shown in red and TAOs executed from native queues, shown in blue.

7. Conclusion

This document describes the final version of the LEGaTO toolchain backend (LEGaTO runtime system). Compared to the first release (M20) this M30 final release contains many new and updated components. At the middleware layer, an improved version of the Redfish API and an improved WebGUI have been introduced. In addition, novel backend drivers to execute on the platform's FPGAs have been developed. Next, targeting energy efficient scheduling, the XiTAO and OmpSs runtimes have been extensively improved. XiTAO now includes a more advanced heterogeneous scheduler that can target both performance and minimization of energy consumption. XiTAO has also been extended with new support for executing applications using pipeline parallelism. Finally, the support for virtualized topologies has been improved. In terms of OmpSs, an improved version of OmpSs@FPGA has been developed, as has the OmpSs@Cluster runtime. These exentensions are described in detail in this deliverable. Deliverable D3.3 also describes many improvements and extensions to our fault tolerance and security stack. We describe advanced support for GPU Checkpointing and novel support for FPGA checkpointing. The support for FPGA undervolting has been improved and is described in the context of CNN Accelerators. On the security side, we have developed support for Trusted Key Management so that secrets can be transferred in a trusted manner to applications running inside of enclaves, such as Intel SGX. The deliverable is completed with tools for development support in the context of both OmpSs and XiTAO. For OmpSs, the final

release of the OmpSs@Linter tool is described. On the XiTAO side, we describe how XiTAO has been integrated with the Extrae/Paraver toolchain from BSC. During the period M20-M30, we have also worked on the integration of the various LEGaTO components. The integration of OmpSs's Nanos6 runtime with XiTAO is described in deliverable D4.3. D4.3 also includes a description of integrations between OmpSs with DFiant and MaxJ, and also an integration of Maxeler with DFiant.

8. References

[1] Grant Allen and Mike Owens. *The Definitive Guide to SQLite*. Apress, 2010.

[2] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13 of *HASP '13*. ACM, 2013.

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *2016 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI '16, pages 689–703. USENIX Association, 2016.

[4] Rosa M Badia. Superscalar programming models: making applications platform agnostic. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[5] Barcelona Supercomputing Center. Ompss-2 specification. https://pm.bsc.es/ftp/ompss-2/doc/spec. (Online; Last access: 11.05,2020).

[6] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

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

[8] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. Securekeeper: Confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 14:1–14:13. ACM, 2016.

[9] E. Brickell and J. Li. Enhanced privacy id from bilinear pairing for hardware authentication and attestation. In *IEEE Second International Conference on Social Computing*, SocialCom '10, pages 768–775, 2010.

[10] ML CIFAR CAFFE. https://github.com/Xilinx/Edge-AI-Platform-Tutorials/tree/master/docs/ML-CIFAR10-Caffe. 2019.

[11] E. Castillo, M. Moreto, M. Casas, L. Alvarez, E. Vallejo, K. Chronaki, R. Badia, J. L. Bosque, R. Beivide, E. Ayguade, J. Labarta, and M. Valero. Cata: Criticality

aware task acceleration for multicore processors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 413–422, May 2016.

[12] christmann informationstechnik + medien GmbH & Co.KG. Extended Redfish API documentation. https://christmann.github.io/recs-redfish-api/index.html, 2020. (Online; Last access: 12.05.2020).

[13] ML CIFAR10. https://github.com/Xilinx/Edge-AI-Platform-Tutorials/tree/master/docs/ML-CIFAR10-Caffe. 2019.

[14] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[15] 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.

[16] Christian Szegedy et al. Going deeper with convolutions. In *CVPR*, 2015.

[17] Kaiming He et al. Deep residual learning for image recognition. In *CVPR*, 2016.

[18] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2018.

[19] Kenneth Goldman, Ronald Perez, and Reiner Sailer. Linking remote attestation to secure tunnel endpoints. In *Proceedings of the First ACM Workshop on Scalable Trusted Computing*, STC '06, pages 21–24. ACM, 2006.

[20] Franz Gregor, Wojciech Ozga, Sébastien Vaucher, Rafael Pires, Do Le Quoc, Sergei Arnautov, André Martin, Valerio Schiavoni, Pascal Felber, and Christof Fetzer. Trust management as a service: Enabling trusted execution in the face of byzantine stakeholders. In *2020 50th IEEE/IFIP Int. Conference on Dependable Systems and Networks (DSN)*, 2020.

[21] Franz Gregor, Wojciech Ozga, Sébastien Vaucher, Rafael Pires, Do Le Quoc, Sergei Arnautov, André Martin, Valerio Schiavoni, Pascal Felber, and Christof Fetzer. Trust management as a service: Enabling trusted execution in the face of byzantine stakeholders. Technical report, 2020.

[22] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 11:1–11:1. ACM, 2013.

[23] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *2010*

USENIX Annual Technical Conference, USENIX ATC '10. USENIX Association, 2010.

[24] Intel Corp. Quartus Prime, 2017.

[25] Intel Corporation. Pin. `http://www.pintool.org`.

[26] Intel Corporation. Attestation Service for Intel Software Guard Extensions (Intel SGX): API Documentation. `https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf`, 2018.

[27] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. In *Intel Whitepaper*, 2016. `https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services`.

[28] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[29] KeyCDN. The growth of web page size. `https://www.keycdn.com/support/the-growth-of-web-page-size/`, 2017.

[30] Chen Liang, Andrew Ferguson, and Rodrigo Fonseca. Zookeeper Benchmark. `https://github.com/brownsys/zookeeper-benchmark`, 2014.

[31] MariaDB. Data-at-rest encryption. `https://mariadb.com/kb/en/library/data-at-rest-encryption-overview/`, 2019.

[32] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104. ACM, 2014.

[33] 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.

[34] Oracle Corporation. InnoDB Startup Options and System Variables, MySQL 8.0 Reference Manual. `https://dev.mysql.com/doc/refman/8.0/en/innodb-parameters.html`, 2019.

[35] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé. Improving the integration of task nesting and dependencies in openmp. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 809–818, 2017.

[36] Miquel Pericàs. First release of the task-based runtime. Technical Report D3.2, July 2019.

[37] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A Secure Database Using SGX. In *2018 IEEE Symposium on Security and Privacy*, S&P '18, pages 264–278, 2018.

[38] Will Reese. Nginx: The high-performance web server and reverse proxy. *Linux Journal*, 2008(173), 2008.

[39] A. Rigo, C. Pinto, K. Pouget, D. Raho, D. Dutoit, P. Martinez, C. Doran, L. Benini, I. Mavroidis, M. Marazakis, V. Bartsch, G. Lonsdale, A. Pop, J. Goodacre, A. Colliot, P. Carpenter, P. Radojković, D. Pleiter, D. Drouin, and B. Dupont de Dinechin. Paving the way towards a highly energy-efficient and highly integrated compute node for the exascale revolution: The exanode approach. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 486–493, 2017.

[40] 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.

[41] Sahand Salamat, Behnam Khaleghi, Mohsen Imani, and Tajana Rosing. Workload-aware opportunistic energy efficiency in multi-fpga platforms. *arXiv preprint arXiv:1908.06519*, 2019.

[42] 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.

[43] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[44] E Ayguadé X Teruel, P Unnikrishnan, and G Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[45] Transaction Processing Performance Council. TPC Benchmark C. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf`, 2010.

[46] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of USENIX ATC*, 2017.

[47] Osman Unsal. Architecture definition and evaluation plan for LEGaTO's hardware, toolbox and applications. Technical Report D2.1, July 2019.

[48] Cats vs Dogs. https://github.com/Xilinx/Edge-AI-Platform-Tutorials/tree/master/docs/CATSvsDOGs. 2019.

[49] Wesley Wong. Stunnel: SSLing Internet Services Easily. White paper, SANS Institute, 2001.

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

[51] Y. Yu, H. Wang, B. Liu, and G. Yin. A trusted remote attestation model based on trusted computing. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, TrustCom '13, pages 1504–1509, 2013.

[52] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.