LEGaTO

Low Energy Toolset for
Heterogeneous Computing

D4.4 "REPORT ON ENERGY-EFFICIENCY EVALUATIONS AND OPTIMIZATIONS FOR ENERGY-EFFICIENT, SECURE, RESILIENT TASK-BASED PROGRAMMING MODEL AND COMPILER EXTENSIONS"

Version 2

Document Information

Contract Number	780681
Project Website	https://legato-project.eu/
Contractual Deadline	30 November 2020
Dissemination Level	Public
Nature	Report
Author	Marcelo Pasin (UNINE)
Contributors	Isabelly Rocha (UNINE), Christian Göttel (UNINE), Valerio Schiavoni (UNINE), Pascal Felber (UNINE), Gabriel Fernandez (TUD), Xavier Martorell (BSC), Leonardo Bautista-Gomez (BSC), Mustafa Abduljabbar (CHALMERS), Oron Port (TECHNION), Tobias Becker (MAXELER), Alexander Cramb (MAXELER)
Reviewers	Madhavan Manivannan (CHALMERS), Daniel Ödman (MIS), Elaheh Malekzadeh (MIS), Chistian von Schultz (MIS), Hans Salomonsson (MIS)

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
1503	2020-10-01, File created, Initial draft structure
1513	2020-10-02, Added energy-efficient IoT draft
1547	2020-10-27, Added secure service networking in OP-TEE
1552	2020-10-27, Added Fault Tolerance introduction
1560	2020-10-28, Added refs to secure service networking
1564	2020-10-28, Added trusted consensus
1579	2020-10-29, Added smart contract execution in ARM TrustZone
1580	2020-10-29, Added enhanced XiTAO data parallel interface
1585	2020-10-29, Applied corrections to security section
1586	2020-10-29, Included first structure of DFiant evaluation
1587	2020-10-29, Added Initial IDE description
1588	2020-10-29, Added initial version of dfiant integration
1592	2020-11-01, Added Pipetune
1597	2020-11-02, Applied small corrections, reordered sections
1611	2020-11-03, Edited improvements in DFiant section
1700	2020-11-13, Added Mercurium
1711	2020-11-16, Added Alya
1712	2020-11-16, Added reference to D3.4 for OmpSs@cluster
1713	2020-11-16, Added Eclipse Che, updated references
1718	2020-11-16, Added task graph analysis
1726	2020-11-17, Added SGX/OmpSs integration
1759	2020-11-22, Document restructuration, introduction written
1766	2020-11-25, Completed IDE plugin and SGX results
1822	2020-11-26, Completed summary and conclusion
1836	2020-12-01, Applied reviews
2018	2021-02-02, Created second version, fixed references
2039	2021-02-04, Provided main findings in executive summary
2048	2021-02-05, Applied reviewer suggestions

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

Index

List of Figures

List of Tables

1. Executive Summary

This report is the final and last delivery of the work produced in LEGaTO's Work Package 4, regarding the toolchain frontend. As it was in all previous cases, it is issued at the same time as Deliverable D3.4 on the backend, and includes many components that complement the ones presented there. This deliverable historically complements previous deliverables SD1 (Chapter 4), D4.2 and D4.3. To avoid overlap between this and previous documents, we choose to highlight only those components that are novel or have been considerably updated since D4.3. Overall, this deliverable covers work that has been done during the past 6 months, from M30 until M36. The text of this deliverable reports the status of the LEGaTO tool chain frontend (work package 4) and is organized in four main technical chapters.

Chapter 3 Compiler Support and Development Environment

Chapter 4 Integration of the Dataflow Programming Substrates

Chapter 5 Energy-Efficiency

Chapter 6 Fault-Tolerance and Security

Chapter 3 starts with Section 3.1, which provides information about the work done on the compiler front-end as well as the programming environment. Its first section presents the task graph analysis developed when considering the integration of Maxeler's MaxCompiler into LEGaTO's stack. We describe how we brought together the Maxeler MaxCompiler and OmpSs, having OmpSs' task-based programming model to assist the dataflow development process, reducing efforts to develop and to optimise dataflow accelerators for the Maxeler platform. We have implemented a tool to automatically detect static sub-graphs from applications annotated in OmpSs, and show them to the programmer to easily identify parts of the algorithm that should be mapped to Maxeler's DFE accelerator. We demostrate that it is possible in many cases to analyse and reduce large task graphs, getting useful insights into program structure.

Section 3.2 presents the work done on enhancing XiTAO's data parallel interface, explaining how it distinguishes synchronous and asynchronous modes of execution. We enhance the XiTAO data-parallel interface to support both synchronous and asynchronous modes of operation, exposing XiTAO features of LEGaTO's tool-chain backend, via loop-parallelism. Due to its popularity in fork-join parallel programming models, the synchronous data-parallel interface provides energy-efficient execution enabled by dynamic moldability and task-based power profiling (demonstrated in D3.4) for a wide range of applications, especially those the adopt kernels from the Machine Learning and Numerical Science fields. We were able to demonstrate the energy efficiency of the enhanced version using a C++ port of the Alya code (from the Smart City use case). There, we compare to a highly optimized work-stealing scheduler and achieve more than 50% better energy-efficiency while still achieving faster time-to-solution on the heterogeneous Nvidia TX2 CPU architecture.

A performance evaluation of OmpSs when running tasks inside SGX enclaves appears in Section 3.3. We have achieved secure task execution within an OmpSs application, by means of using Intel SGX Enclaves as the secure framework. The Enclave interface is very similar to the OmpSs task interface, so mapping tasks onto it is straightforward. With this platform, we have been able to measure the performance penalty that the secure infrastructure introduces to secure the data, with and without encryption, and we have observed that a good way to overcome this penalty is by using task-level parallelism.

Section 3.4 concludes Chapter 3, presenting the last evolution of the Integrated Development Environment. As presented in previous deliverables, we developed a set of Eclipse IDE Plugins with the goal to provide programming hints to the user. The add-on provides context-based help regarding the OmpSs/OpenMP directives and clauses, assisting on the decision about what is the most appropriate expression for the heterogenous parallelism. We have also achieved the integration of the OmpSs development environment as a Docker container within Eclipse CHE.

Our work regarding dataflow engines appears in Chapter 4, which in this deliverable is entirely dedicated to present our work with DFiant, the dataflow hardware description language that decouples functionality from implementation developed in LEGaTO. Our main contribution focuses on the DFiant evaluation and the significant improvements in programmability of FPGA devices without sacrificing performance or fine-grain hardware generation control. Our empirical findings show that DFiant designs usually save between 50% to 70% in lines of code compared to the RTL baseline.

Chapter 5 highlights the progress done in energy-efficiency, in two sections. All research results described in this chapter were published and presented as peer-reviewed papers in conferences.

We present a study of IoT applications running in trusted environments, with special attention given to power consumption when using Arm's TrustZone in Section 5.1. We developed two tools, a network perfomance evaluation tool for secure services, and an extension of Hyperledger Fabric for running (blockchain) smart contracts, both using OP-TEE and exploiting ARM TrustZone. The performance tool is capable of identifying bottlenecks in the network performance of secure services, for which we were able to identify and quantify the energy and performance overheads. Using the smart contract tool, we have shown that it is possible on ARM hardware to reduce the energy cost by one order of magnitude compared to state-of-the-art hardware used with Hyperledger Fabric. The improved energy consumption comes at the cost of performance, which is also a result of the additional security when running the service in OP-TEE or ARM TrustZone in general.

Section 5.2 is dedicated to a study of adapting system- and hyperparameters in machine learning jobs in order to be be more energy-efficient. We designed and implemented a novel parameter tuning algorithm for deep learning clusters which takes system parameters into account. Our work resulted in an open-source tool that leverages the repetitive behaviour of tuning jobs for deep neural networks, to quickly find the best set of parameters. The tool is modular which makes it easy to swap between similarity functions and underlying search algorithms. We evaluated 7 different real-world datasets from different domains, including text classification and image recognition. When compared against state-of-the-art deep neural network tuning systems, our tool shows evidence that we can greatly reduce tuning and training time (27% and 41% respectively) without accuracy loss.

The final technical part is Chapter 6, with two sections. Section 6.1 contains a thorough study on how to implement trusted consensus using SGX enclaves, including an evaluation with a replicated database. Trusted consensus is a rollback-resistant, integrity preserving data service that aims to provide data replication in untrusted cloud environments. It combines hardware enforced anti-rollback mechanisms over a customized protocol that executes database requests and replicates the changes. To address attacks that involve abusing the protocol majority balance, we use a single-instance protocol that leverages hardware monotonic counters to ensure that attacks are much harder to achieve. Our results demonstrated that the associated overheads are limited and scale harmoniously.

Section 6.2 concludes our last technical chapter, with a brief reference of LEGaTO's Fault Tolerant Interface to use with FPGAs, as it has been previously described in Deliverable D4.3. Also, in Deliverable D5.3 we presented the work of Alya at large scale, a LEGaTO use case in a smart city simulation that sometimes runs for long hours on the Marenostrum supercomputer in Barcelona. To mitigate the high probability of failures during Alya jobs, we apply checkpoint/restart as described in Deliverable D3.3, substantially decreasing the amount of re-computation required upon a failure. We created a tool capable of checkpointing heterogeneous applications running on CPU and GPU clusters. Our tool uses multiple storage systems to implement multilevel checkpointing with parallel device-to-host data streams when writing into reliable storage. With FTI, we have enabled fault tolerance for heterogeneous applications while substantially decreasing the checkpoint and recovery time. Our work around FTI and Alya was also published and presented as a peer-reviewed paper in a conference.

2. Introduction

LEGaTO is a Horizon 2020 research and innovation action to develop advanced techniques to make it easier to build large performance-hungry applications. This deliverable marks the end of the work done in LEGaTO's Work Package WP4, named "Tool-chain front-end". WP4 was composed of seven tasks, each task producing research outputs on a different aspect being developed in WP4. The tasks are listed below.

- Task 4.1: Definition / Design (M1-9)
- Task 4.2: Programming Model features for energy efficiency (M1-36)
- Task 4.3: IDE plugin (M7-36)
- Task 4.4: Compiler support (M7-30)
- Task 4.5: High-level Synthesis for FPGA (M1-36)
- Task 4.6: Task-based kernel identification/DFE mapping (M7-36)
- Task 4.7: Fault Tolerance and Security (M1-36)

WP4 was intended to develop techniques for high-productivity, energy-efficient, high-performance heterogeneous programming with added security and fault-tolerance. Heterogeneity is obtained using computers that incorporate central processing units (CPUs), field-programmable gate arrays (FPGAs), and graphics processing units (GPUs). Security is obtained by using novel hardware extensions that allow for building trusted execution environments. Fault-tolerance is managed using standard techniques, tailored for the original environment developed in the project.

Requirements for the techniques developed in WP4 (energy efficiency, security, fault-tolerance, performace) may compete against each other, so we often strived for finding adequate trade-offs. To achieve application development under such constraints, a great number of components has been developed, allowing to map applications written in a high-level task-based dataflow language onto the heterogeneous platform. Each component presents different ways of tackling different requirements, working at different levels, and often obtaining trade-offs between two or more aspects. Finally, the components in WP4 are developed in tight integration with the work done in WP3. Many times, deliverables from WP3 refer to work done in WP4, and vice-versa. In the paragraphs that follow, we recap the deliverable history of WP4, then we present the structure for this document.

Deliverable D4.1 (Definition/design of front-end toolbox) was integrated in Chapter 4 of Superdeliverable SD1 [34]. It corresponded to the work done in Task 4.1 (Definition/Design), ended at month 9. SD1 contains a comprehensive set of functionalities designed to be offered as front-end tools for programming applications. A software architecture was introduced with all main aspects on which LEGaTO is focused, along with a number of extensions, then proposed to be implemented in the infrastructure management to support the execution of the proposed task model.

Deliverable D4.2 (First release of energy-efficient, secure, resilient task-based programming model and compiler extensions) was delivered at month 20. It presented OmpSs, the execution and programming model for the tool chain, with a description of the resource sharing between different runtimes. For programmability, we proposed the development of OpenMP and OmpSs support into Eclipse and we developed our first plugins. We extended the OmpSs compiler to support autoVivado, to compile OmpSs applications which target Xilinx FPGAs. We also developed and assessed a checkpoint-restart fault-toloerant library to be integrated inside the runtime system. We analysed energy-efficiency and security trade-offs, with different trusted hardware support, and produced the design of a task-based scheduler called HEATS. To understand the overhead generated by trusted execution environments, we also developed a monitoring framework. Finally, we designed and started implementing DFiant, a Scala-embedded hardware description language that leverages dataflow semantics to decouple functionality from implementation constraints.

Deliverable D4.3 (Final release of energy-efficient, secure, resilient task-based programming model and compiler extensions, including FPGA toolchain) was delivered at month 30. There, we presented our work to upgrade OmpSs to support FPGAs, including autoAIT (to support Xilinx ZCU102) and Vivado HLS 2019.3. We offered a DAG-friendly data-parallel interface for XiTAO runtime system, leveraging its energy-efficient scheduling features on heterogeneous platforms. We also added a Maxeler DFE architecture plugin to the OmpSs runtime, to support spawning of tasks to Maxeler hardware. We extended the compiler to obtain the communication weights and computation load, helping to decide the proper mapping of DFE kernels. Our first experiments using OmpSs with SGX are also presented, with new tools for monitoring and for adding fault-tolerance to legacy applications inside SGX enclaves. The IDE plug-in also evolved, to support new releases of Eclipse CHE. We presented a stable version for DFiant, with composable port addition, a simplified finite state machine syntax and constraint tags for injecting RTL-like semantics, along with is first OmpSs integration. We proposed a novel data deduplication technique that allows for saving energy of IoT devices. Finally, we have rewritten the core of the smart city use case to use our checkpointing library.

This is **Deliverable D4.4**, the last deliverable of Work Package WP4. Here, we detail the research progress we made in the last few months of LEGaTO concerning front-end tools for productivity, security, fault-tolerance and energy efficiency. Mimicking previous deliverables, D4.4 contains four technical chapters. Each of these chapters is presented below.

Chapter 3 (compiler support and development environment) is split into four sections. In Section 3.1 we present how we automatically identify static sub-graphs in OmpSs' task graph for mapping them to Maxeler's dataflow engines and we demonstrate it with three different algorithms: n-body simulation, a neural network, and Cholesky matrix decomposition. We describe the enhancements provided on top of the XiTAO data parallel interface in Section 3.2. With this interface, applications can leverage the XiTAO runtime for energy efficiency and interference awareness, and resource moldability. The section concludes with an evaluation of the Alya solver with different programming models including OpenMP, OmpSs, and XiTAO. Section 3.3 describes our efforts on integrating secure tasks into OmpSs using SGX and, finally, Section 3.4 presents the last improvements made with the IDE plugin.

Chapter 4 contains two sections concerning DFiant dataflow engines. Section 4.1 presents an evaluation of DFiant, focusing on programmability of FPGA devices. Section 4.2 shows how DFiant kernels are integrated with OmpSs in a FPGA project.

Chapter 5 presents two recent research results on energy efficiency. Section 5.1 presents an analysis of the energy-efficiency of two distributed, hardware-secured services specifically designed for IoT devices. Section 5.2 presents a novel study about how to tune machine learning models' hyperparameters in order to trade performance for energy efficiency.

Chapter 6 is the last chapter, containing our last months' contributions on security and fault-tolerance. Section 6.1 presents recent research results on hardware-secure computing with an implementation of a distributed consensus algorithm in an untrusted environment. Section 6.2 contains a summary of the progress of fault-tolerant checkpointing for FPGAs, and refers to the work reported in D4.3 (of Work Package WP3).

3. Compiler support and Development Environment

3.1. Task graph analysis

Here we consider integration of the Maxeler MaxCompiler into the LEGaTO tool stack. As described in deliverable D2.1, Maxeler uses a dataflow oriented model to describe accelerators for FPGA-based Dataflow Engines (DFE), and the compiler uses a Java-based meta-language MaxJ to describe the compute kernels. Since developing and optimising a dataflow accelerator in MaxJ can involve a considerable amount of effort, we want to leverage the task-based programming model in OmpSs to assist the dataflow development process. The task-graph generated by OmpSs is conceptually very suitable for being translated into a dataflow model. Therefore, we want to automatically identify static sub-graphs in the task graph generated from OmpSs and translate them into a form that helps the developer to identify parts of the algorithm that should be mapped to the DFE accelerator. This includes analysis of the compute to I/O ratio of the subgraphs to be accelerated.

The process for developing an accelerator for any given program begins with an understanding of the algorithm to be accelerated. The next step is the identification of compute intensive portions of code, then finally the transfer of those code sections to the accelerator. This can be a straightforward process when the implementation is well understood, however in real world scenarios it is often the task of a developer to optimize an existing implementation for which complete information is not available.

3.1.1. Project goals

The goal of the task graph analysis project is to investigate automated analysis of task-based program traces. The aim being to aid developers by providing the insight they need to make good acceleration decisions for existing applications. This step of gaining insight into the program is crucial, especially for FPGAs because of high implementation effort and large time requirements for generating FPGA bitstreams.

For trivial examples, tools like graphviz can easily render visual representations of task graphs as in Figure 3.1. For programs at realistic problem sizes though, these graphs quickly become unintelligible as in Figure 3.2.

3.1.2. Useful acceleration information

One important metric for accelerating code is the ratio between task cost and the size of the working set. Typically the best candidates for acceleration are pieces of code that involve large amounts of processing and small volumes of data. To determine these quantities, the task cost in terms of primitive operations as well as the size of task inputs and outputs need to be instrumented and made available for processing. In the case of FPGA accelerators it is also important to know the types of operations taking place in tasks, as the hardware cost of implementation can vary significantly between operation types.

In a task graph, nodes represent tasks and edges represent dependencies between tasks due to shared variables or other synchronization. It is important to capture this structural information in the analysis as it is possible for a collection of tasks to represent a better candidate for acceleration than any individual task. Figure 3.3 contains a graph with several tasks, none of which are ideal acceleration candidates by themselves. Collectively however, they make a better candidate for acceleration than any individual task.

Collected information needs to be displayed to the user in a meaningful way. One representation that suits this purpose is the loop-flow graph (see Figure 3.4). A Loop-flow graph is a form of task graph where iterative processes are folded away to hide redundant information. Nodes in the graph represent operations within the code and are optionally annotated with an iteration count to indicate that the process runs multiple times. Edges are annotated with the quantity of data moving between tasks. The graph may be partitioned to indicate which tasks take place on the accelerator and which take place on the CPU. In the case of the graph in Figure 3.4, tasks above the dotted line (convolution, rectifier linear operator and spatial pooling steps) are on

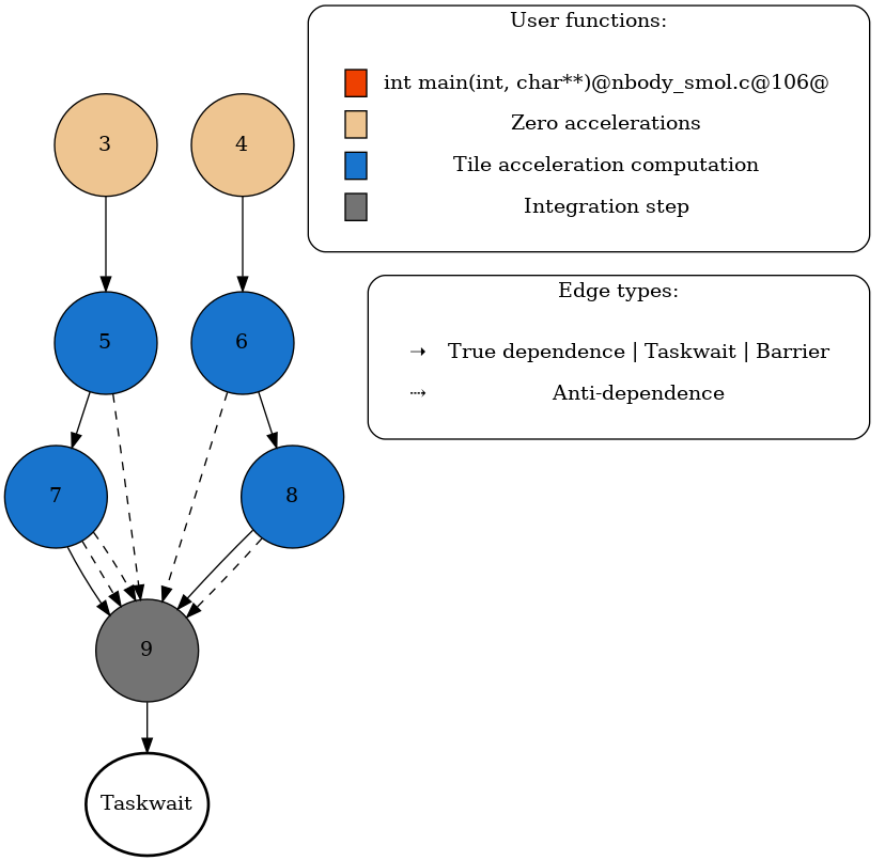


Figure 3.1. Tiled n-body program graph for a trivial problem size.

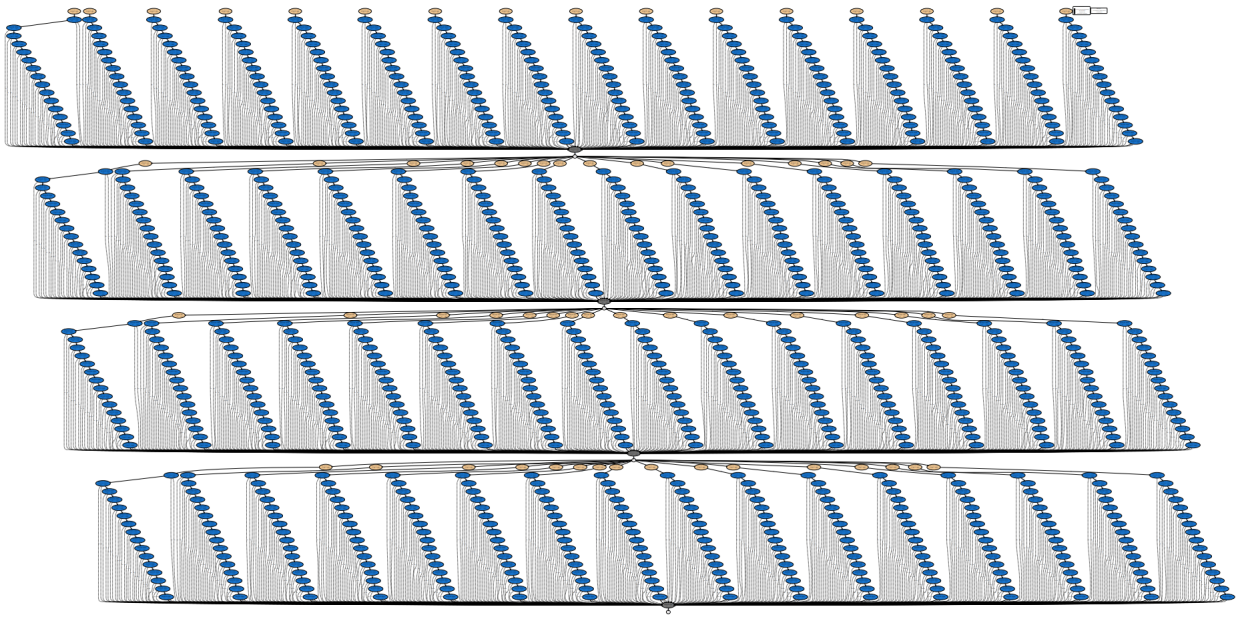


Figure 3.2. Tiled n-body graph at a larger (though still trivial) problem size.

the accelerator, whereas the fully connected layers are on the CPU. This assignment of tasks was determined by manual analysis.

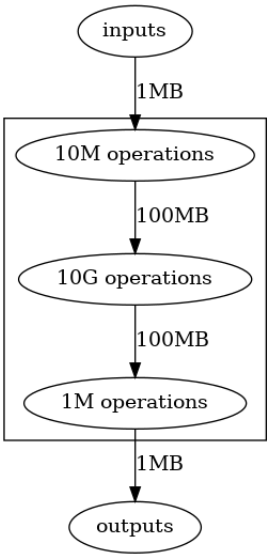


Figure 3.3. A set of tasks suitable for collective acceleration.

3.1.3. Acquiring program traces

Data exfiltration was achieved using the instrumentation capabilities of the nanox runtime. The nanox instrumentation API was expanded to include events for task I/O ranges and operation counts and a new instrumentation plugin was implemented to export this data in a machine readable format for further processing.

3.1.3.1. Mercurium analysis infrastructure

To facilitate the evaluation of task cost and enumeration of operations, a pass was implemented using the Mercurium analysis infrastructure to compute expressions for the number of operations in each task. Mercurium is a source-to-source compiler for C/C++ and FORTRAN with support for parallel programming models like OpenMP and OmpSs. It is formed by a series of phases that analyze and transform the code, which is later passed to a native compiler to generate the final binary. The picture in Figure 3.5 shows a high level overview of the compiler pipeline.

Mercurium implements a series of phases devoted to source code analysis, particularly of OpenMP/OmpSs codes. The infrastructure is built on top of a Parallel Control Flow Graph (PCFG) that represents the flow of the source code as well as the parallelism and other semantics expressed by OpenMP/OmpSs directives. Based on this representation, Mercurium offers different classical data-flow analysis, including use-definition chains and reaching definitions, and optimizations, like constant propagation and strength reduction, all adapted to consider the parallel semantics of OpenMP/OmpSs. Finally, more complex analysis like induction variables analysis and scalar evolution are available.

3.1.3.2. Estimating task cost

With the objective of estimating the cost of relevant regions of code, a set of features have been included in the Mercurium compiler. These are the following:

- A new analysis phase that computes the cost of OmpSs tasks and loops in terms of number of operations. The operations are organized based on the type of the operands (i.e., integer, float or double) and the type of the operation (i.e., addition/subtraction, multiplication or division), resulting in a total of 9 groups.
- A new transformation phase that instruments the user code with the information of the counters computed by the analysis. This code is executed at runtime generating per region events containing the operation counts.

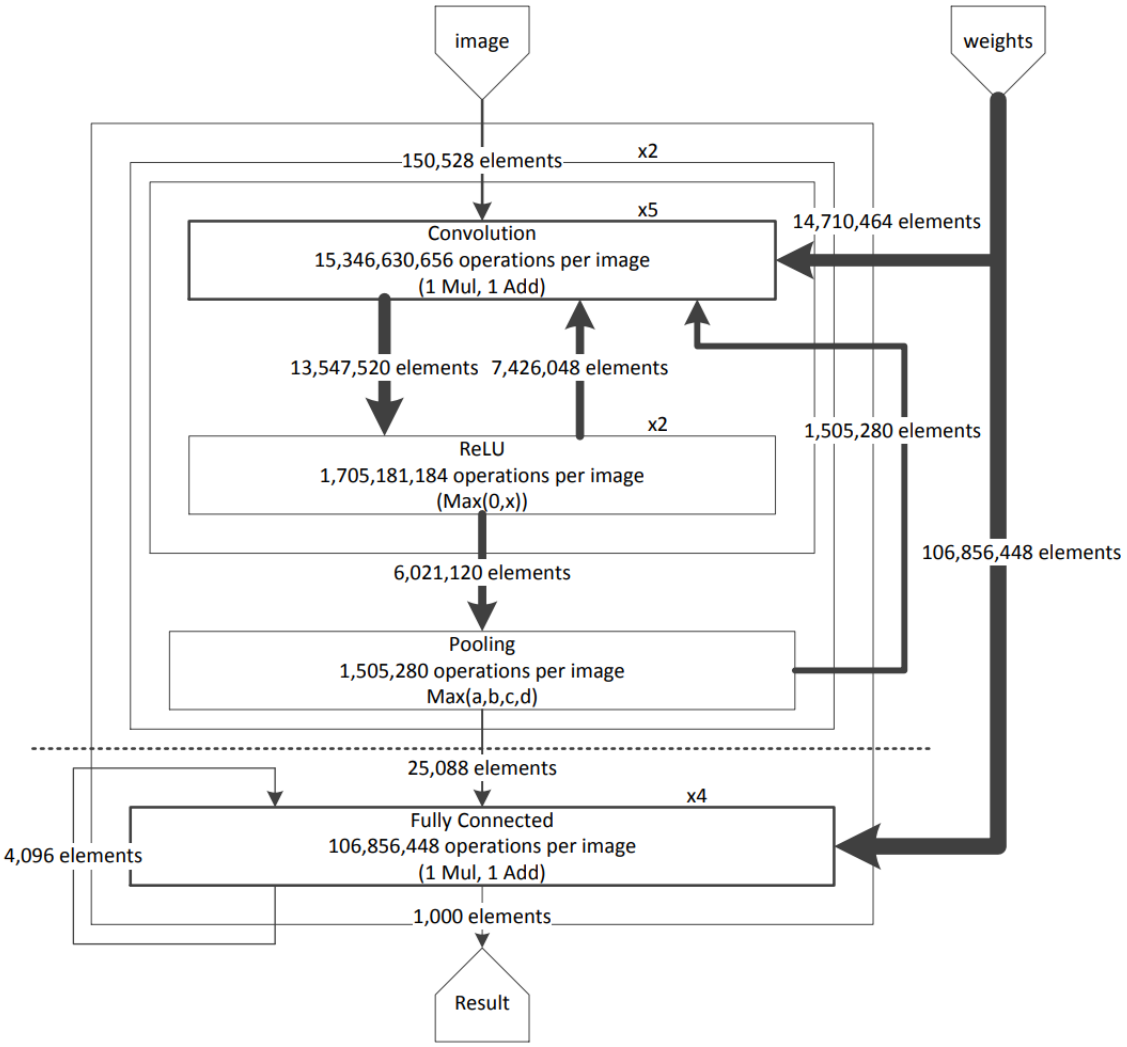


Figure 3.4. A loopflow graph for the VGG16 CNN.

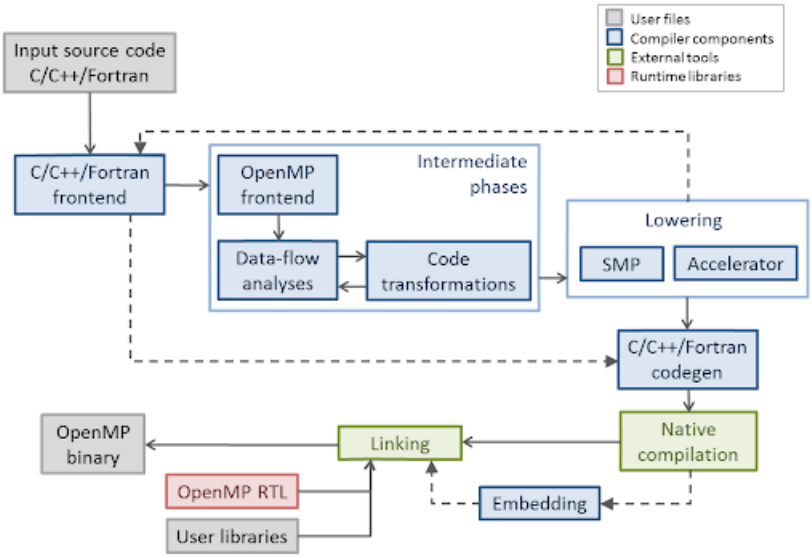


Figure 3.5. Diagram of Mercurium analysis infrastructure.

The new analysis phase performs a top-down inside-out traversal of the user code using the PCFG. This means that, although the code is traversed sequentially, inner levels are computed before outer levels, e.g., in a two-level nested loop, the counters of the inner loop are computed first, and then aggregated accordingly to the values of the variables of the outer loop. The main computations occur in the following points:

- Loops: the number of operations of the region inside the loop is multiplied by the number of iterations of the loop. In case the number of iterations depends on another induction variable, then this computation is delayed until the different values of the outer induction variable are known.
- Conditional statements: unless the condition of the statement can be decided to be true or false, the number of operations of each branch is computed, and then just the maximum of the two is used as an approximation.
- Tasks: the number of operations of the task region is computed.
- Function calls: when the code of the function code is accessible, the compiler performs inter-procedural analysis; otherwise, the function is also annotated as a different type of operation, so the number of times this function is called can also be computed.

The analysis does not modify the user code in any way. However, the information gathered about the counters is stored in the PCFG structure, particularly in the nodes corresponding to loops and tasks, for later use. Furthermore, the results are also detailed in a report available to the user.

The new transformation phase provided in Mercurium uses the data included in the PCFG during the analysis phase to instrument the user code using the API provided by Nanos to that end. The instrumentation occurs in two different places:

- The code associated with the task constructs include a new event at the end with the expression that computes the number of operations of each type.
- Loops include instrumentation before and after the loop, so the region associated with the counter can be recognized at runtime. In this case, if the compiler has not been able to compute the number of iterations, instrumentation is added to compute it at run-time (unlike for tasks, this cannot be automatically done at run-time because loops are not recognized by the Nanos runtime systems as tasks are.

In addition to the Mercurium analysis pass, two other methods for assessing task complexity were also implemented. One uses execution duration of the task as an approximation for task cost, and another uses processor hardware counters via the PAPI library to estimate operation counts. Support and accuracy for these methods varies and is summarised in Table 3.1.

3.1.3.3. Acquiring data ranges

The nanox runtime supports several different models for task dependencies as separate plug-ins that can be activated at runtime. To capture memory ranges associated with dependencies, the cregions (contiguous regions) plugin was used. To determine the data size of a given dependency, start addresses and end addresses were compared to determine the total size of the data.

3.1.3.4. Task graph analysis

Gathered data is processed using a set of analysis tools. These tools are responsible for processing the graph and delivering data to the user in a readable format. Before discussing the analysis process in detail it is important to define some terminology:

Method	Support	Accuracy
Execution time	High - Always available.	Low - Many variables not accounted for (CPU cache, processor performance, compiler optimisations). No way to distinguish between operation types.
Hardware counters	Medium - Counter support varies between processors.	Medium - Variances due to compiler optimisation and non-task code being counted. Some operation types can be distinguished.
Mercurium analysis pass	Low - Difficult to achieve robust operation for arbitrary programs	High - Results are invariant between runs and accurately represent operations as expressed in the source code.

Table 3.1. Summary of task cost evaluation methods.

- Task instance: Refers to an instance of a task. There may be many instances of any given task on a graph.
- Task: Refers to the source code associated with a task instance.
- Vertex: Vertices on a task graph represent task instances. For the purposes of this section, the term can be considered synonymous with the term “task instance”.
- Dependence: An edge on the graph, indicative of data transfer or other synchronization event between task instances.

The first step of the analysis involves detecting program “main loop” and averaging loop iterations. This process works by finding vertices that represent “taskwait” synchronization events and dividing the graph into segments delimited by these vertices. A vector of “structural fingerprints” is then computed for these segments by recursively hashing edges and vertices within each segment. The main loop is then identified by searching for repeating patterns in the structural fingerprint vector. Finally main loop iterations are averaged together by averaging the performance metrics associated with each vertex. In the event that main loop identification is not successful, the analysis treats the entire graph as a single iteration (see Figure 3.6).

Further folding of the graph is achieved by merging vertices and edges under certain conditions. The first condition is for “adjacent” vertices. A set of vertices are considered adjacent if they share a dependency with a previous task, but not with each other. The second condition is for “consecutive” vertices. Two vertices are considered consecutive if they are identical and one is dependent upon the other. Vertices can only be merged if they represent instances of the same task and have the same number of inputs and outputs of the same size. During merging, task performance information including execution time and operation counts are averaged and the vertex is assigned an iteration count equal to the sum of the iteration counts of the original vertices. Edges are also inspected and merged if the memory ranges attached to them are contiguous. This merging process is applied iteratively to the graph until no further merging opportunities are present.

The final stage of the analysis is the assignment of task code either to the accelerator or to the CPU. This is achieved by a brute-force algorithm which iteratively evaluates the effectiveness of all possible assignments. The evaluation process for any given assignment uses operation counts and graph structure to determine the amount of compute on both the CPU and the accelerator, as well as the amount of CPU-accelerator communication. These quantities are evaluated using a fitness function (see Figure 3.8), and the assignment with the highest fitness is selected. If no assignment scores better than zero, the assignment algorithm assigns the entire task code to the CPU.

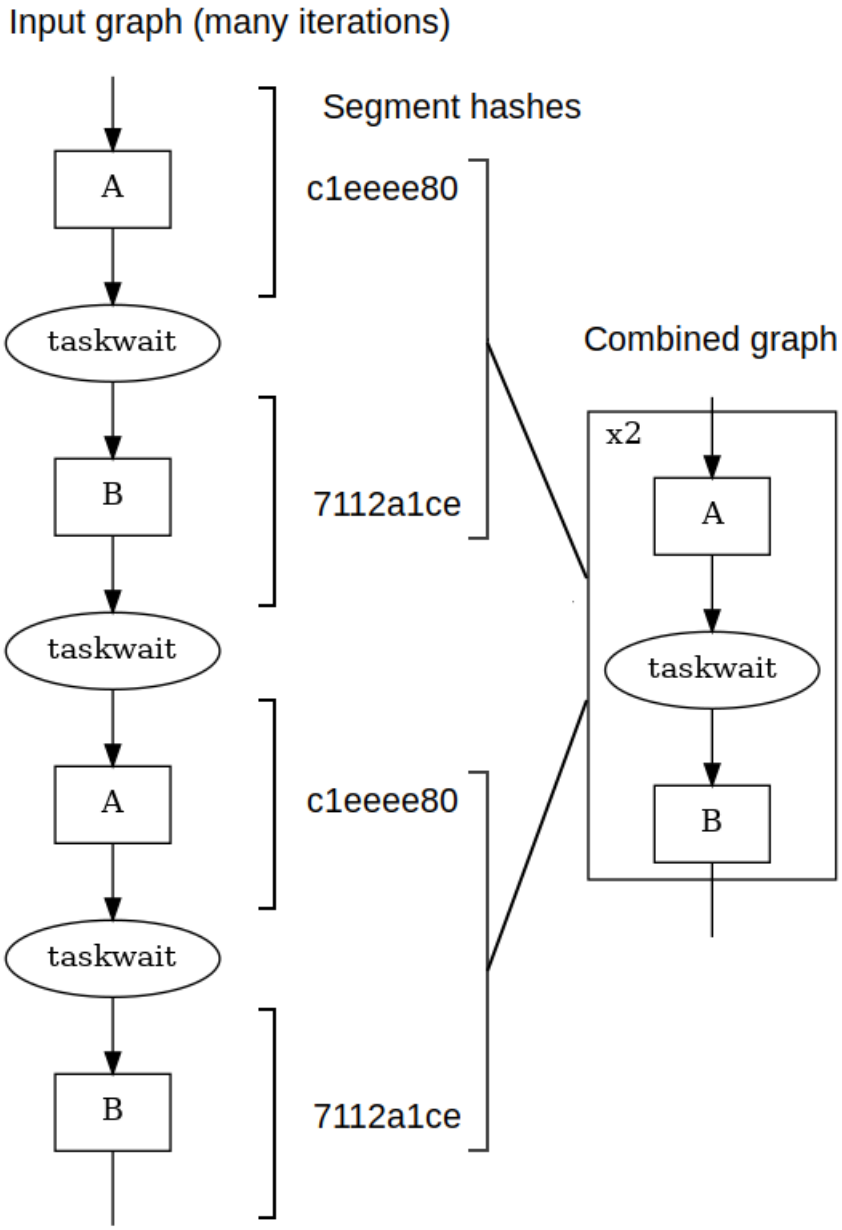


Figure 3.6. Diagram of main loop detection process.

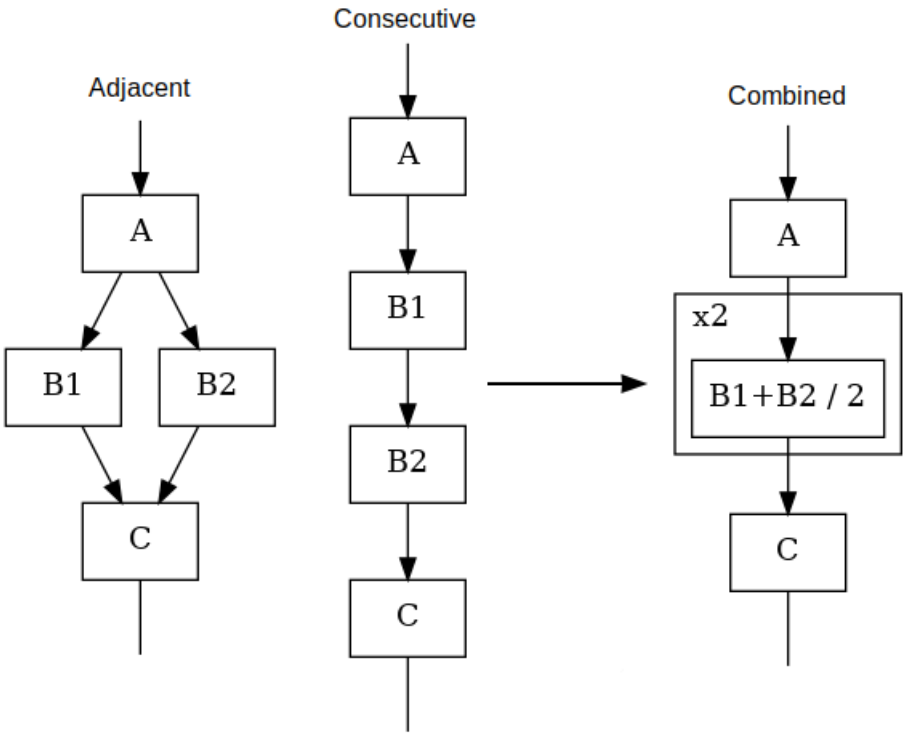


Figure 3.7. Illustration of vertex merging.

$$f(o_a, o_c, b_{io}) = \begin{cases} \frac{o_a}{b_{io}}, & \text{if } \frac{o_a}{o_c} > 10 \\ 0, & \text{otherwise} \end{cases}$$

where:

o_a = Operations on the accelerator.
o_c = Operations on the CPU.
b_{io} = CPU-accelerator I/O in bytes.

Figure 3.8. Task assignment fitness function.

Representation	Inputs	Outputs
Task graph	Task graph, ompss task graph	Task graph, total operation count, task summary (csv)
Loop flow graph	Loop flow graph, task graph	Loop flow graph, total operation count, task summary (csv), dot file

Table 3.2. Summary of tool inputs/outputs.

3.1.3.5. Output and tool structure

The tools define two representations, a task graph and a loop-flow graph. Each representation can be serialized or deserialized as json. Each representation has an associated executable into which files may be passed to perform conversions between representations and/or output analysis information. Table 3.2 contains a summary of possible analyses and input/output formats for each representation. Some output formats also accept optional parameters. For example, the dot file output representation may optionally include task assignments (shown as blue task outlines) or compute to I/O ratio (shown as red-green gradient coloring on task nodes). During development emphasis was placed on modularity and extensibility to enable the tools to be expanded and repurposed for future projects.

3.1.4. Testing

To test the tools, task-based implementations of three different algorithms were chosen as case studies. These were; n-body simulation, the vgg16 neural network and cholesky matrix decomposition.

3.1.4.1. N-body gravitation simulation

The term N-body refers to a class of algorithms for modeling the motion of sets of particles, typically under the influence of mutual forces. For this test case an n-body gravitational simulation was implemented whereby particles move relative to one another under the influence of mutual gravitation. In each timestep particle positions and velocities are computed based on their relative positions and velocities from the previous timestep. Each timestep is divided into two tasks; the force/acceleration computation stage, which involves computing the acceleration applied to each body as a result of gravitational interaction with each other body, and the integration step which involves updating body positions and velocities with respect to the computed forces/accelerations.

The acceleration computation is by far the most computationally expensive step, with computational complexity $O(N^2)$ where N is the number of particles. The integration step is inexpensive by comparison with computational complexity $O(N)$. A common strategy when accelerating workloads like this is tiling the acceleration computation loop so that body positions for a tile fit within the processor cache. In this task-based implementation, each tile is given its own task. The graph in Figure 3.2 is the result of analyzing the task graph generated by a 32768 particle simulation with a tile size of 512x512 bodies running for four timesteps.

For this test algorithm, task cost was estimated using the mercurium analysis framework to acquire operation counts on a per-operation-type basis. It can be seen that the majority of operations take place in the acceleration computation step (see Figure 3.9). The assignment generated by the tools suggests computing particle accelerations on the accelerator and integration on the CPU. This is reasonable within a single iteration but in practice the researcher is unlikely to be interested in seeing the results at the end of every timestep. In this case it would become favourable to execute all stages of the computation on the accelerator and only fetch results once the simulation is complete. Unfortunately, nanox is not currently capable of relating data volumes and address ranges to variable names. This means that it isn't possible to identify which memory regions are outputs when the analysis tools are run. Because of this, the tools assume that all output memory ranges at the end of the loop are required outputs. Additionally,

each body-to-body interaction includes a call to the C standard library square root operation that is not included in the operation count total due to it being implemented in a library.

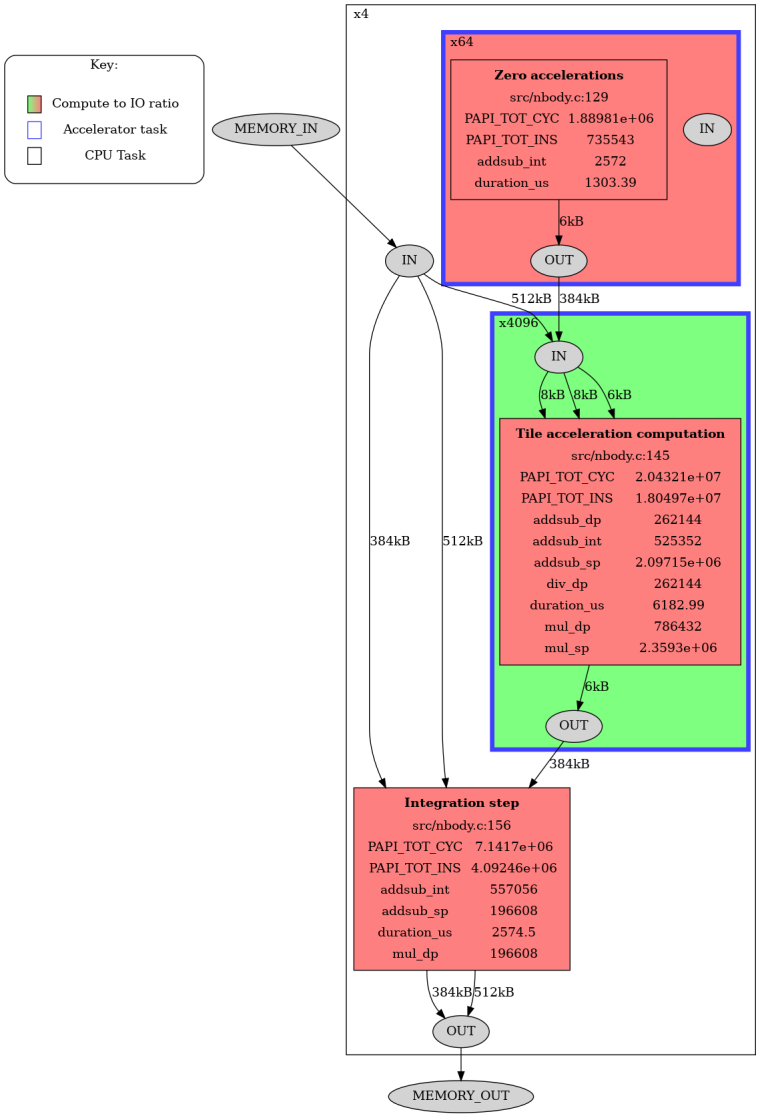


Figure 3.9. Analyzed task graph; 32768 bodies, 512x512 tiles, 4 iterations.

3.1.4.2. Image recognition using VGG16

The VGG16 neural network is a convolutional neural network for image classification [37]. The computation involves passing data through multiple layers of computation each with differing characteristics (see Figure 3.10). There are three major data items required to process an image. Image and layer output state, weights and biases. Convolutional layers have a large amount of network state, but a small number of weights and biases as the same values are used for each kernel. By contrast, fully connected layers have a large number of weights but a comparatively small amount of network state. For this implementation of the network, layer computations were chunked into blocks and separate buffers were allocated for the results of each layer.

The assignment algorithm suggests computing the results of convolutional layers on the accelerator and fully connected layers on the CPU. This assignment makes sense as the large quantity of weight data associated with fully connected layers makes them unfavourable. This aligns well with the manual algorithm partitioning discussed earlier (see Figure 3.4).

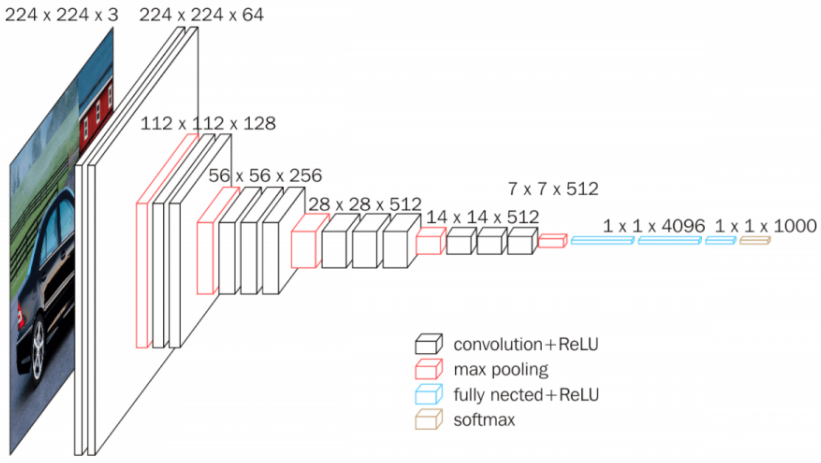


Figure 3.10. VGG16 neural network structure.

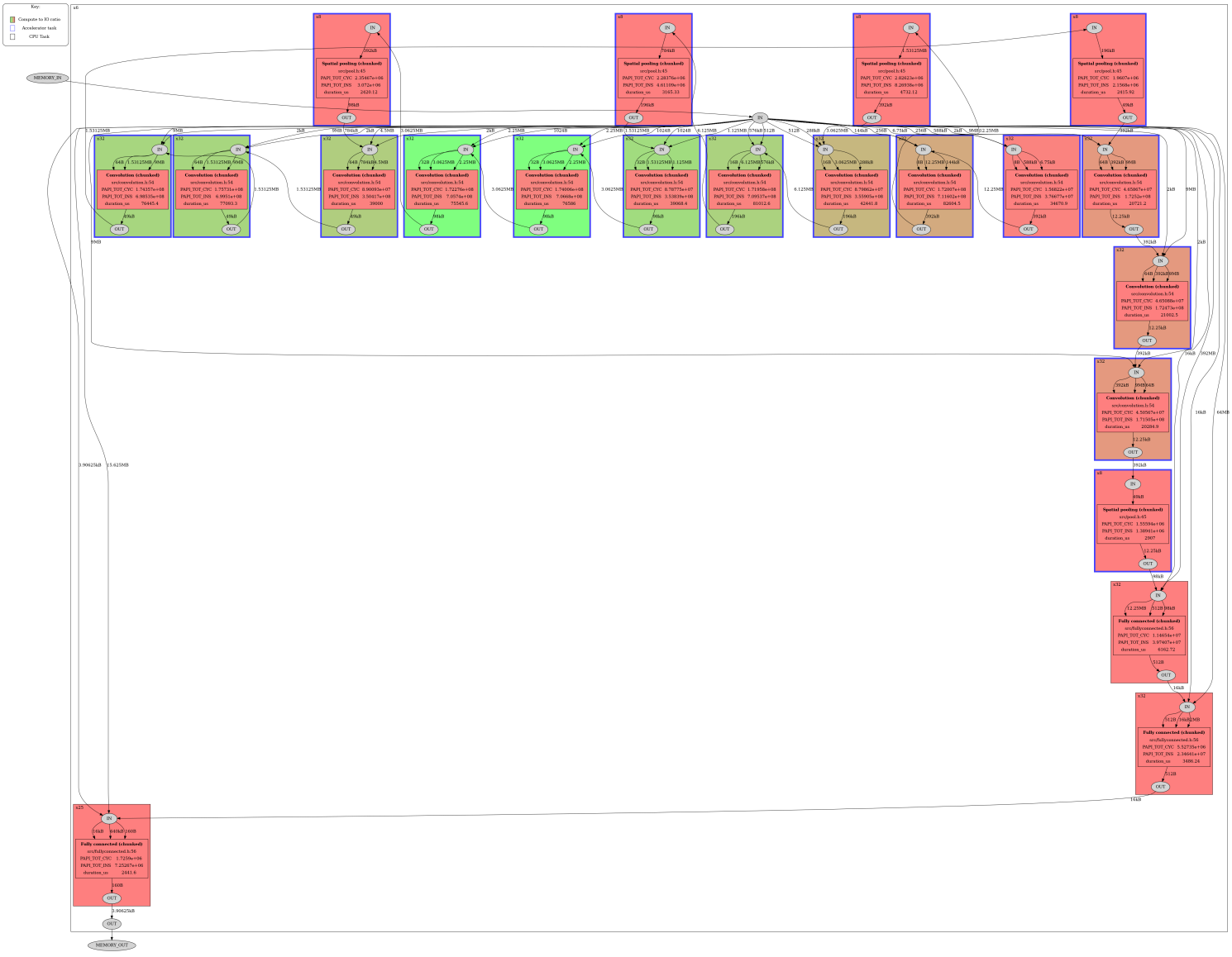


Figure 3.11. VGG16 loop flow graph after analysis.

3.1.4.3. Cholesky matrix decomposition

Cholesky decomposition is a decomposition of a hermitian positive definite matrix into the product of a lower triangular matrix and its conjugate transpose. This process has applications in numerical methods and solving systems of linear equations.

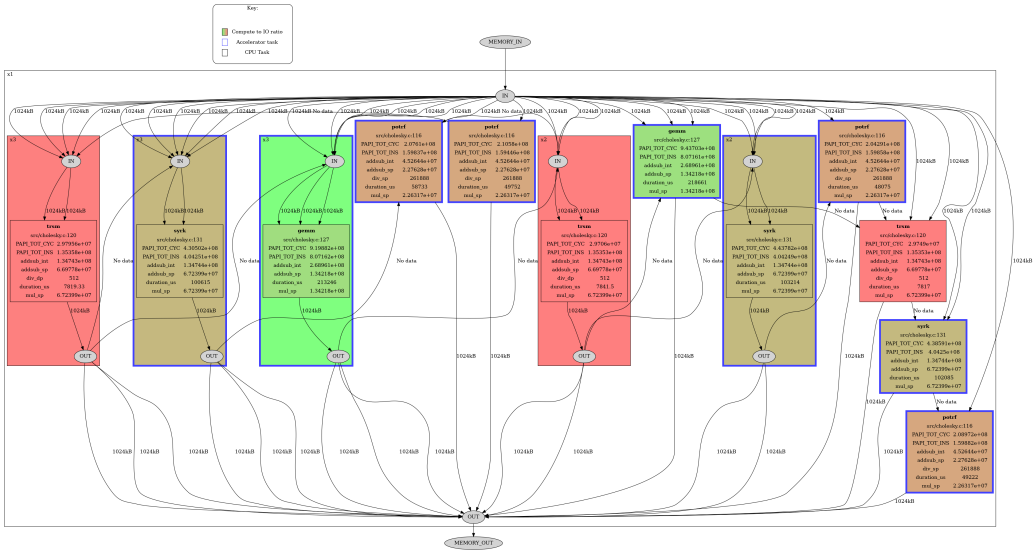


Figure 3.12. Analyzed task graph, choleksy decomposition, 2048x2048 matrix with tile size 256x256.

Unlike VGG16 or n-body examples, the task-based Cholesky matrix decomposition algorithm exhibits a less regular graph structure. Because of this the analysis phase fails to reduce the complexity of the graph appreciably and graphs generated at larger problem sizes still quickly become unintelligible.

3.1.5. Limitations and future work

3.1.5.1. Task cost estimation

The static analysis of the source code can provide useful information regarding the operation count, while avoiding any overhead as required by dynamic instrumentation techniques. However it has the following limitations:

- The Mercurium compiler only allows the analysis of one translation unit at a time. This means that, for a given file, the compiler can only access the code in that file and any included header. Inter-procedural analysis is then limited to one translation unit.
- The capability to compute the exact number of operations in a given region depends on the knowledge the compiler can gather about the variables involved in the region. Particularly, the compiler requires knowledge about:
  - Variables determining the iteration space of the loops: when these are not known, the accurate expression can still be generated but its exact value will only be known at run-time.
  - Variables involved in conditional statements: when these are not known, then only an approximation of the total number of operations can be provided.
- No loop unrolling is performed to obtain the operation count. As a consequence, there are limitations on the type of code that can be analyzed successfully by this phase. Particularly, only loops in the canonical form for (init-expr; test-expr; incr-expr) structured-block are accepted.
- Currently, conditional statements (e.g., if-else statement) are just approximated, i.e., the number of operations of the statement is always computed to the maximum number of operations of any of the possible branches, unless the condition always evaluates to the same value and it is known at compile-time.

- Several features of advanced versions of C++ are not supported in the analysis, e.g., lambda expressions, range-based loops and concurrency.

As part of future work, the plan is to:

- Provide a more accurate solution for conditional statements.
- Support more complex loops like while-loops.
- Accept information from the user regarding functions that are called in the source code which code is not available at compile time. This could be in the form of annotations associated with the source code, or a separate template-file with the function headers and the expected number of operations of each type.

3.1.5.2. Task graph analysis

The analysis tool has shown the capacity to reduce the redundancy of plain task graphs considerably, with excellent results when applied to data parallel algorithms with regular task graphs like the n-body and vgg16. However testing exposes a number of limitations:

- Irregular graphs like those produced by cholesky decomposition are handled poorly by the analysis. Investigation of improved methods for detecting repetitive structures in the graph both for main loop detection and folding of less regular graph structures remains as future work.
- Variable names are not attached to data ranges. This prevents the possibility of suggesting/analyzing more complex strategies like loop unrolling, a common strategy on FPGAs to optimise I/O and memory bandwidth. The lack of this feature is primarily due to limitations in the nanox instrumentation API and could conceivably be solved through use of an alternative instrumentation technique, or by further extending the API.
- Main loop detection is error prone and expects main loops to be delimited by synchronisation events. This may not be the case for all algorithms and generally cannot be assumed. More general ways to detect repetitive graph structures remain as future work.
- Output graphs sometimes have confusing layouts due to graphviz's dot layout engine. Currently the dot file output for the loop flow graph exercises no control over the layout of vertices during rendering. By manually adding rank allocations for vertices, it may be possible to further improve the readability of output graphs. This also remains as future work.
- Instrumented programs have to be executed with a single worker thread because of the manner in which the nanox runtime handles dependencies between tasks. This limits the usage of the tools in their current state to the instrumentation of programs at fairly small problem sizes. Solving this issue remains as future work.
- Concurrent and commutative sections are currently not supported by the nanox instrumentation plugin developed for the project. Supporting these features of the programming model remains as future work.

In conclusion, the proof of concept analysis techniques implemented for this project show that it is possible in many cases to analyze/reduce large task graphs and achieve useful insights into program structure, but that substantial work remains to be done to develop the technique to a point where it is usable for a wide variety of input programs.

3.2. Enhanced XiTAO Data Parallel Interface

In this deliverable, we explain the enhancements provided on top of the XiTAO data parallel interface. As highlighted previously in D4.3, XiTAO incorporates modern C++ compiler technology to deliver a DAG-friendly data parallel interface. With this interface, many applications that consist of parallel Single Program Multiple Data (SPMD) regions can leverage the backend features offered by the XiTAO RT including energy efficiency and interference awareness depicted by WP3.

```
// tao_width: XiTAO specific resource hint
// i: the loop counter
// loop_start: loop iterator start
// loop_end: loop iterator end
// scheduling_type: XiTAO scheduler type (e.g. dynamic)
// block_length: the chunk size for each task

auto dataparallel_nodes = __xitao_async_data_parallel_region
  (tao_width, i, loop_start, loop_end,
   scheduling_type,  block_length,
     for (int j = 0; j < N; j++) {
       C[i][i] = 0;
       for (int k = 0; k < N; k++)
         C[i][j] += A[i][k] * B[k][j];
     }
);

for(int i = 0; i < dataparallel_nodes.size(); ++i) {
  previous_node[i]->make_edge(dataparallel_nodes[i]);
}

for(int i = 0; i < dataparallel_nodes.size(); ++i) {
  next_node[i]->make_edge(dataparallel_nodes[i]);
}
```

Figure 3.13. The basic structure of a DAG based program inserting SPMD code regions

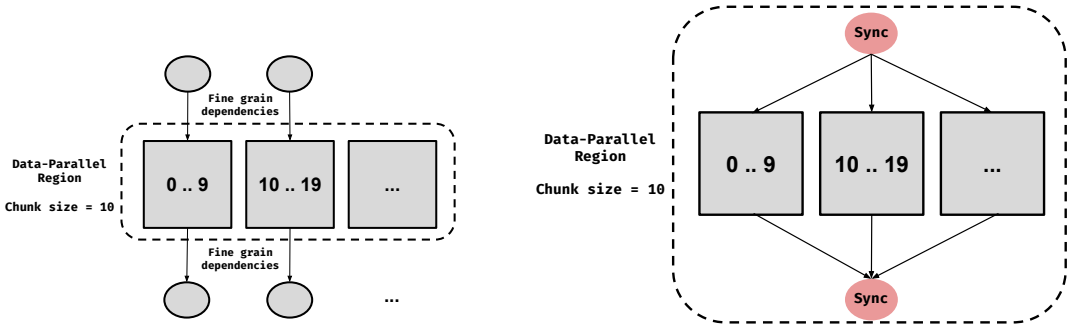
The interface makes it possible to indicate a resource hint that the runtime can use to aggregate resource to a specific task within the loop or a set of tasks (resource moldability). Here, we make the distinction between the asynchronous and synchronous modes of executions.

The data-parallel frontend depicted here resides on top of the XiTAO Energy Efficient Scheduler (EAS), which is evaluated in Deliverable D3.4. EAS is an energy efficient work stealing runtimes targeting modern platforms with asymmetric cores and cluster-based DVFS (e.g. NVIDIA Jetson TX2), as well as symmetric homogeneous platforms. The scheduler estimates the energy consumption on a per-task level and performs task placement decisions for each task to minimize the energy consumption. Deliverable D3.4 evaluates several alternatives in the design of the XiTAO's energy efficient scheduler and has the following characteristics:

1. Task type-awareness that is important to select the most efficient resources for each task instance;
2. Utilizing an exponential backoff sleep strategy, which helps reduce energy waste from work stealing loops with minimal impact on performance;
3. Adaptive task moldability that can further improve energy efficiency by reducing resource over-subscription and inter-task interference.

3.2.1. The Asynchronous Data Parallel Mode

This mode arises from the assumption that programs can be expressed as DAGs with different granularities. One of the main motivations behind the inclusion of async data parallel nodes is the fact that task loops can then be seamlessly inserted into task graphs (see Figure 3.14a), and will benefit from reducing the overhead of fork-join programming approaches and achieve energy-efficiency from the runtime backend. The snippet on Figure 3.13 shows how a loop parallel region, for example, can be part of a full DAG structure using the XiTAO programming interface. Also, Table 3.3 highlights the interface parameters. The capability of nesting loop parallel nodes in a DAG workflow has been supported. Also, a few explanatory benchmarks adopted from Ro-



(a) Asynchronous mode with fine-grain dependencies

(b) Synchronous mode that is analogous to fork-join models

Figure 3.14. XiTAO data parallel modes

```
__xitao_sync_data_parallel_region (tao_width, i, loop_start,
                                   loop_end, scheduling_type,
                                   block_length,
  for (int j = 0; j < N; j++) {
    C[i][i] = 0;
    for (int k = 0; k < N; k++) C[i][j] += A[i][k] * B[k][j];
  }
);
```

Figure 3.15. The basic structure of a DAG based program inserting SPMD code regions

dinia Benchmark Suite and Barcelona OpenMP Task Suite have been developed and will soon appear in the XiTAO online repository.

Parameter	Usage
`width`	The XiTAO resource hint to be given to the loop tasks.
`iter`	The loop index/iterator.
`end`	The loop end.
`sched`	The scheduling options (e.g., static, dynamic, energy-aware, etc.)
`block_size`	Governs the granularity of task creation.

Table 3.3. The parameters input by user to the XiTAO's asynchronous data parallel interface

3.2.2. The Synchronous Data Parallel Mode

The sync data parallel mode is semantically equivalent to OpenMP/OmpSs taskloops, and mainly supported for backward compatibility of legacy codes. In this mode, operation happens in 3 steps, shown in Figure 3.14b). First, the DAG execution of previous nodes is synced. Second, the loop is divided into chunks of tasks according to the `block_length` parameter. Third, an implicit wait is inserted to pause the execution until all loop tasks have finished. Listing 3.15 shows an example of such usage.

3.2.3. Evaluation of the Alya solver

The most time consuming part of Alya code is the solution of the Poisson equation. The procedure consists in solving a linear system of equations in each time integration step. The matrix of the system is a laplacian, and has as many rows as the number of cells used in the discretization of the computational domain. For the smart city use case, the matrix remains constant during the full simulation, therefore the pre-processing operations are negligible. The iterative solvers are the best option to solve the system due to its low memory footprint when compared with direct solvers. The Preconditioned Conjugate Gradient (PCG) is the iterative solver most commonly used in Alya. The solver is mainly composed of three algebraic operations: vector operations (axpy and dot) and the sparse matrix vector multiplication (spmv). The operations are memory

bounded, and its implementation consist of a loop of the size of rows of the matrix. This configuration enables using directive-based models or run-times that automatically parallelize that loop. Four implementations of the solver have been evaluated using different components of the LEGaTO tool-chain.

- **OpenMP**: GCC version 7.4, loop-based, using parallel for directives
- **OpenMP taskloop**: GCC version 7.4, using parallel taskloop directives
- **OmpSs-2**: OmpSs-2 version 2020.09, using tasking
- **XiTAO**: xitao v0.8-alpha
- **CUDA**: version 10.0 is used, axpy and dot operations are linked with the cublas library.

The experiments have been performed on the NVIDIA Xavier nodes provided by Bielefeld. The solver was executed on matrix systems obtained from Alya execution using unstructured meshes with sizes ranging from 100,000 to 1,600,000 rows. In these matrices, the number of non-zero elements per row is less than five and oddly distributed due to the unstructured meshes. The sequential execution has been included as a baseline. Parallel executions have been obtained using 8 cores for the SMP models, and CUDA 10.0 for the execution on the GPU. The execution time of the solver for the different models is shown in Table 3.4.

Matrix Size	Sequential	OpenMP	OmpSs	Xitao	OpenMP taskloop	CUDA
50,000	0.47	**0.13**.	0.27	0.35	0.64	0.17
100,000	1.11	**0.24**	0.48	0.69	1.40	0.29
200,000	2.38	0.48	0.85	1.41	2.77	**0.42**
400,000	7.63	1.32	2.05	4.32	10.13	**1.07**
800,000	17.64	3.14	4.22	9.98	22.85	**2.18**
1,600,000	48.35	8.56	11.04	27.30	61.03	**5.64**

Table 3.4. Time in seconds for the different executions of Alya's solver.

The best cases for each matrix size are highlighted in bold. Note that for matrix sizes with less than 200,000 rows, the OpenMP version is the most efficient implementation with a speedup of 3.6 times with respect to the sequential version. On the other hand, from 200,000 rows and above, the CUDA implementation is the most efficient one with a speedup of up to 8.5 times with respect to the sequential case. CUDA implementation does not perform well with small matrix sizes due to two factors: i) the GPU requires of certain occupancy in order to exploit the maximum memory bandwidth that is hard to achieve with the sparsity pattern of our matrices, and ii) the GPU time includes memory transfers of the right hand side (r.h.s) and resulting vectors from host to device, the relative weight of those transfers is higher in the smaller cases. Anyhow, the workload in the final smart cities use case ranges from 1,000,000 to 4,000,000 per node, being in the range in which the GPU implementation is the most efficient one.

Among the CPU-only executions, OpenMP loop-based implementation is the most efficient code for all the matrix sizes. In the range of the smart city use case, OpenMP outperforms OmpSs by a 28% and XiTAO by a 318%. The drop of performance with OmpSs tasking is caused by the lower data locality achieved, compared to OpenMP for loops. Additionally, it is important to note that the XiTAO data-parallel interface is semantically similar to the OpenMP taskloop, and it loses some performance to achieve energy efficiency as was shown in D3.4. Hence, a more fair comparison with XiTAO loops would be considering the taskloop implementation of OpenMP, since XiTAO uses a similar strategy to associate threads with task DAGs. For the largest case, XiTAO is 2.2 times faster than OpenMP taskloop based implementation, and is consistently faster in the other cases.

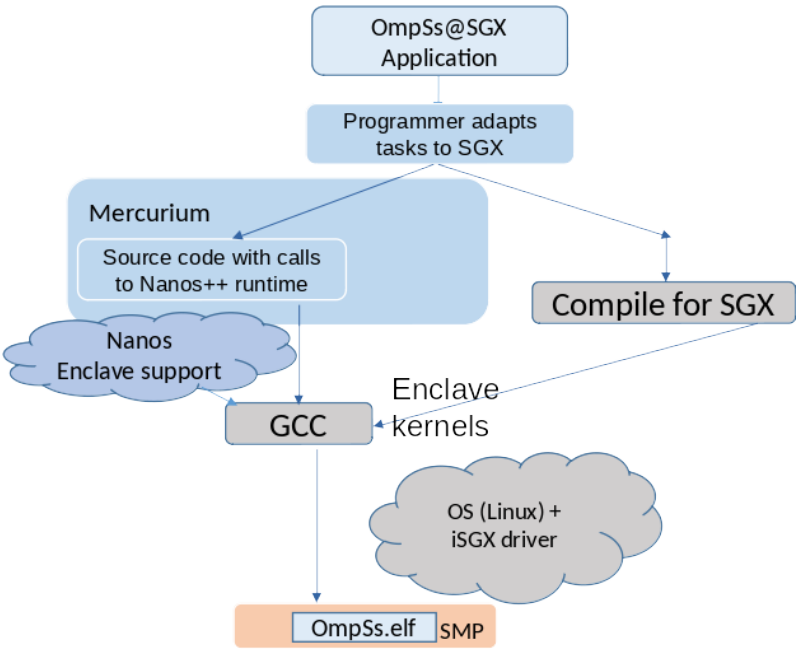


Figure 3.16. Intel SGX applications with OmpSs support.

3.3. OmpSs Integration with SGX

Figure 3.16 depicts the proposed approach by showing how the combination of OmpSs programming model with Intel SGX can be performed. First, the programmer has to adapt the tasks to SGX which means defining the tasks as C/C++ functions which will be declared and implemented using the Enclave's interface. Then, the parallelization pragmas can be added to the secure application in the same way as it is done for non-secure application except that all the pragmas have to be placed outside the Enclave. For instance, if the user wants to annotate a function declared inside the Enclave then the pragmas have to be placed on the function call instead of declaration. Finally, the annotated code is compiled using the Mercurium compiler and the Nanos++ runtime schedules the application tasks on the platform.

We evaluated the proposed approach by means of four benchmarks consisting of cholesky factorization, dot product, matrix multiplication and two versions of the STREAM benchmark. We analyze each of these applications in the following 3 versions: a pure OmpSs implementation, an implementation combining OmpSs and SGX, and a third one adding encryption on top of the OmpSs and SGX combination. All the applications used here take as input a matrix and the parallelization is done based on the blocks of this input matrix. Considering that, the size of a given input matrix indirectly defines the graph size. i.e., the number of total tasks spawned. Moreover, the block size in which we split the matrix defines the input size of each task as the larger a given block is the more data has to be processes by a given task. We then evaluate how each version performs when we vary the matrix and block size as well the scheduling algorithm. Finally, We also vary the number of threads to see how the applications scale for multiple threads.

We deploy our experiments on a cluster of 4 quad-socket Intel E3-1275 CPU processors with 8 cores per CPU, 64 GiB of RAM and 480 GB SSD drives. The machine runs Ubuntu Linux 20.04.1 LTS on a switched 1 Gbps network. Power consumptions are reported by a network connected LINDY iPower Control 2x6M Power Distribution Unit (PDU), which we query up to every second over an HTTP interface to fetch up-to-date measurements for the active power at a resolution of 1W and 1.5% precision.

Cholesky Kernel.

The Cholesky Kernel is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. The kernel uses four different linear algorithms: potrf, trsm, gemm and syrk. The way we parallelize the code is by annotating these kernel functions so that each call in the previous loop becomes the instantiation of a task. The

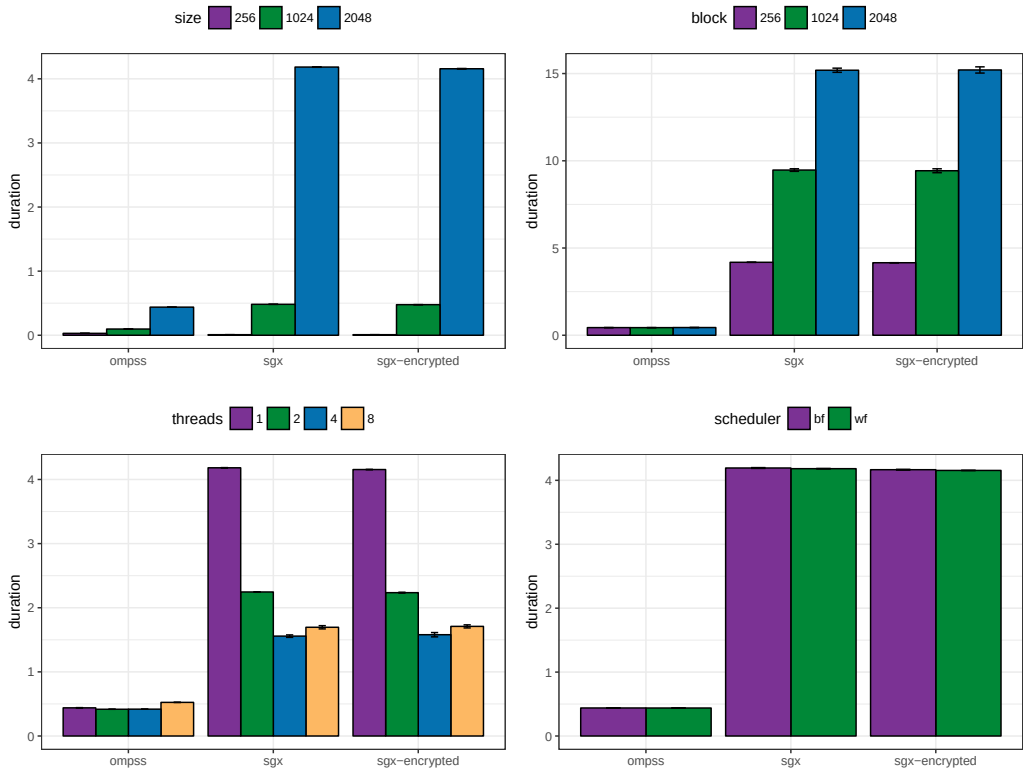


Figure 3.17. Evaluation results for Cholesky Kernel application.

original application used these algorithms from the MKL or OpenBLAS linear algebra libraries, and, in order to have the algorithms available inside the enclave without dependences on Linux services we have provided the four algorithms in source code inside the application itself.

Figure 3.17 shows the comparison of the results of the application. Comparing results when varying the matrix size (plot on the top-left), we can observe that the overhead when using SGX and SGX with encryption is noticeable (7x slowdown). Nevertheless encryption does not introduce overhead compared to the non-encrypted version. Regarding the influence of the block size, results (plot in the top-right) show that when the block size if larger, the overhead is also larger. Regarding the number of threads (plot in the top-right), it shows that the execution time scales better when using SGX and SGX with encryption, due to the fact that each task has more work to do, and thus, the overhead of the runtime is reduced in comparison. Finally, the OmpSs scheduling policy (plot on the bottom-right) has no impact on the performance. Both breadth-first and work-first (Cilk-like policy) give the same performance.

Dot Product

The dot product is an algebraic operation that takes two equal-length sequences of numbers and returns a single number obtained by multiplying corresponding entries and then summing those products. A common implementation of this operation accumulates the result of each iteration on a single variable. This kind of operation is called reduction, and it is a common pattern in scientific and mathematical applications.

There are several ways to parallelize operations that compute a reduction but in our implementation we use a vector to store intermediate accumulations. This means that tasks operate on a given position of the vector and the parallelism is determined by the vector length. Finally, when all the tasks are completed then the contents of the vector are summed up.

Figure 3.18 shows the experimental results for the 3 versions of this application. In this case we observe an overhead between 2x and 3x on the SGX and SGX Encryption versions when compared with pure OmpSs implementations. Regarding the scalability of applications, for the parameters values chosen the input size does not seem to impact performance. When it comes to number of threads, for this given application the optimal solution is 2 threads which is at least 15% faster

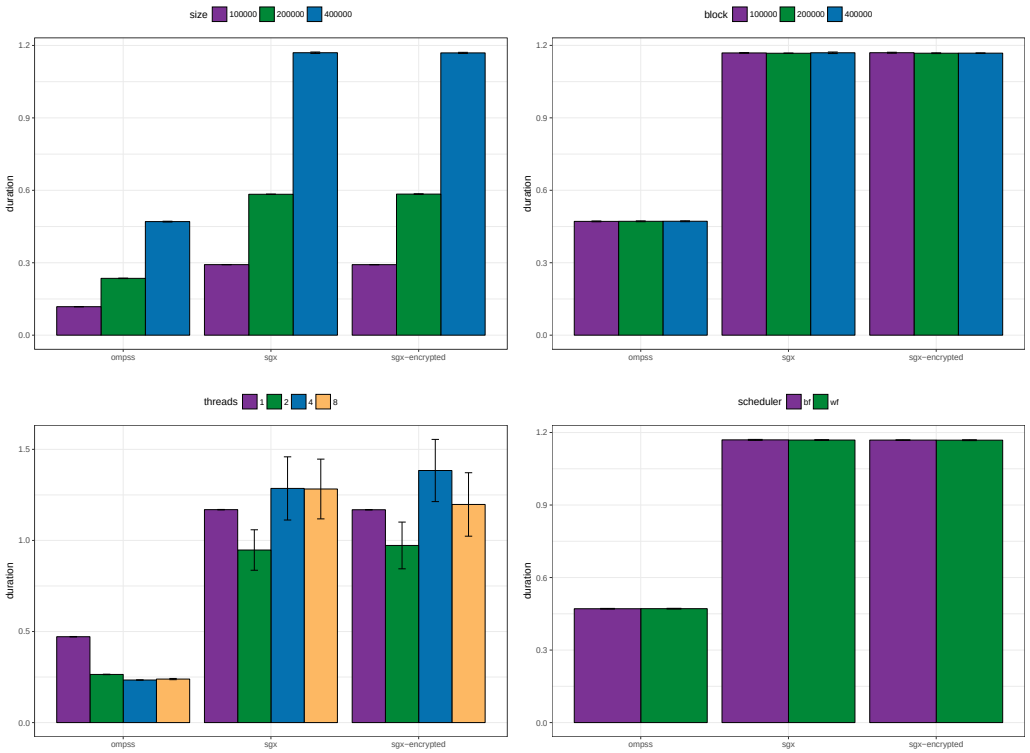


Figure 3.18. Evaluation results for Dot Product application.

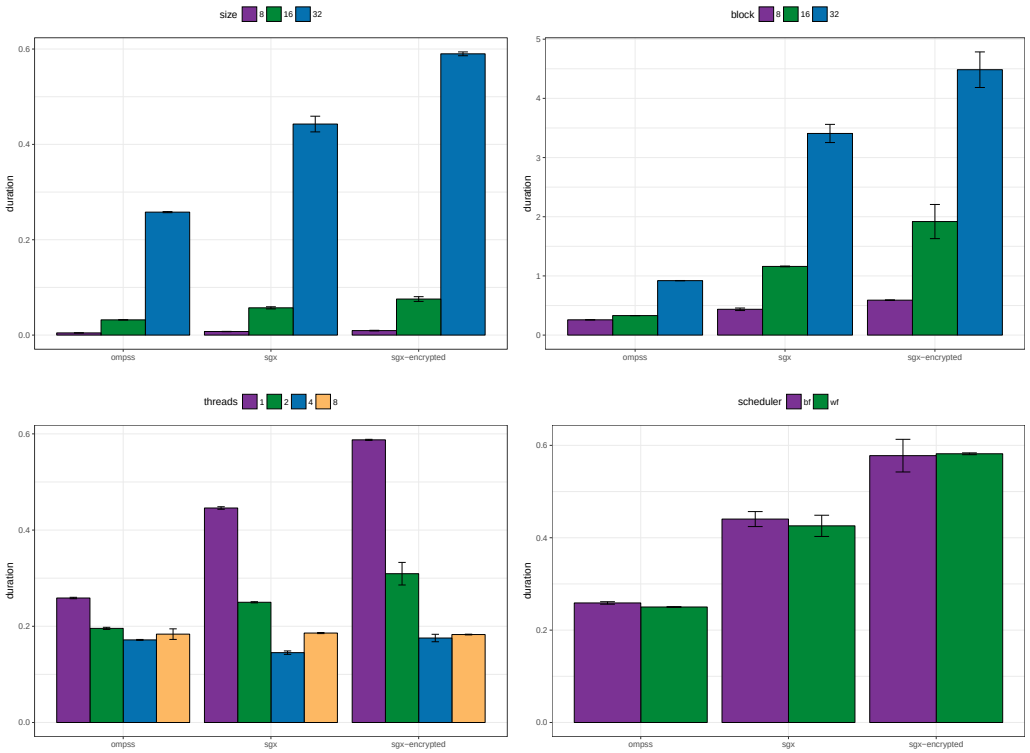


Figure 3.19. Evaluation results for matrix multiplication application.

than the sequential version. However, with 4 and 8 threads the performance starts to decrease again and this can be due to the overhead added when compared with the size of tasks.

Matrix Multiplication

The matrix multiplication application receives matrix A and matrix B as input, performs the mul-

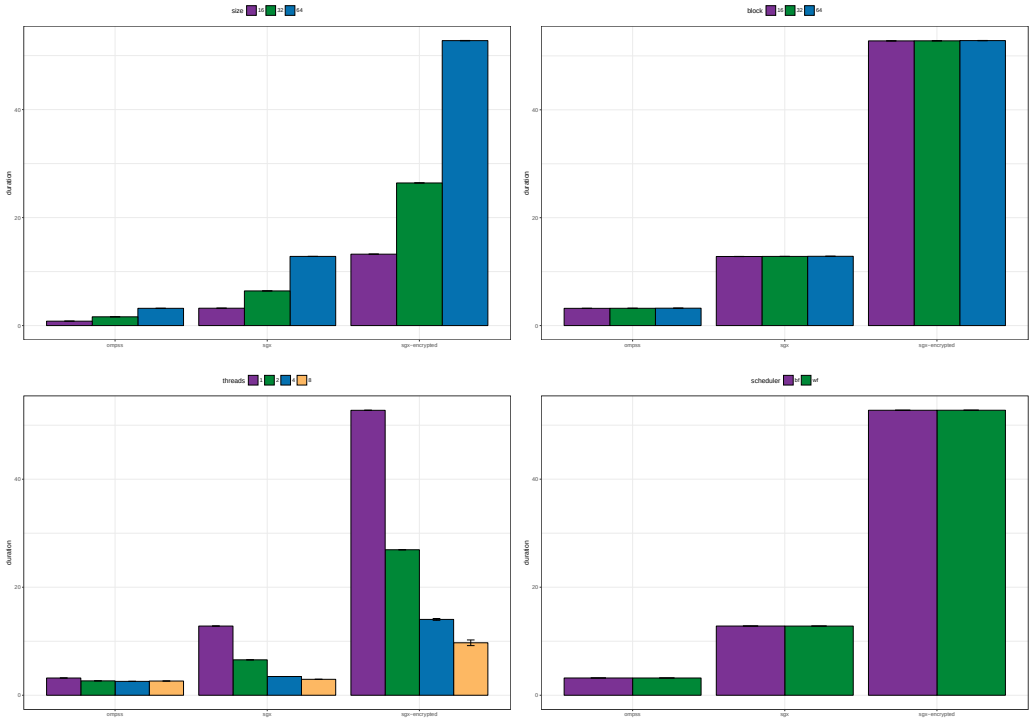


Figure 3.20. Evaluation results for STREAM application (barrier based version).

tiplication calculations between A and B, and outputs the result in a third matrix C. Figure 3.19 shows the experimental results for the 3 versions of this application. In this context, we observe at least 1.5x to 2x overhead for the SGX and SGX plus encryption versions when compared to the pure OmpSs version, respectively. However, all three applications scale well when a smaller input sizes is used and more threads are added. In fact, the both SGX versions present at least 2x improvement when using multiple threads when compared with their respective sequential versions while the pure OmpSs version only present up to 25% improvement. With this we can conclude that although there is a clear trade-off between performance and security, we can easily minimize the extra cost of adding a security layer by combining SGX with OmpSs.

STREAM benchmark

The STREAM benchmark [6] is a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector kernel. We present the following two versions of this application: one that inserts barriers and another without barriers. The behavior of the version with barriers resembles the OpenMP version, where the different functions (e.g., Copy, Scale) are executed one after another for the whole array. In the version without barriers, functions that operate on one part of the array are interleaved and the OmpSs runtime keeps the correctness by means of the detection of data-dependences.

Figure 3.20 and Figure 3.21 show the experimental results of the STREAM application with and without barriers, respectively. Independent of the versions, we can observe that STREAM based on the dependency graph performs better than STREAM using barriers. For the SGX versions this observation becomes even more evident as the overhead of running all the tasks inside the enclave, encrypting/decrypting the input/output data becomes adds an extra cost. Another difference between the versions is that the block size does not seem to play a role in the barrier based version while the dependency based version performs better with larger block sizes. This behavior is consistently observed in all 3 variations of this version. Finally, another observation is that the encryption overhead is much larger in the barrier version than in the dependency version. In fact, in the dependency one we do not observe a significant variation between the SGX versions with and without encryption.

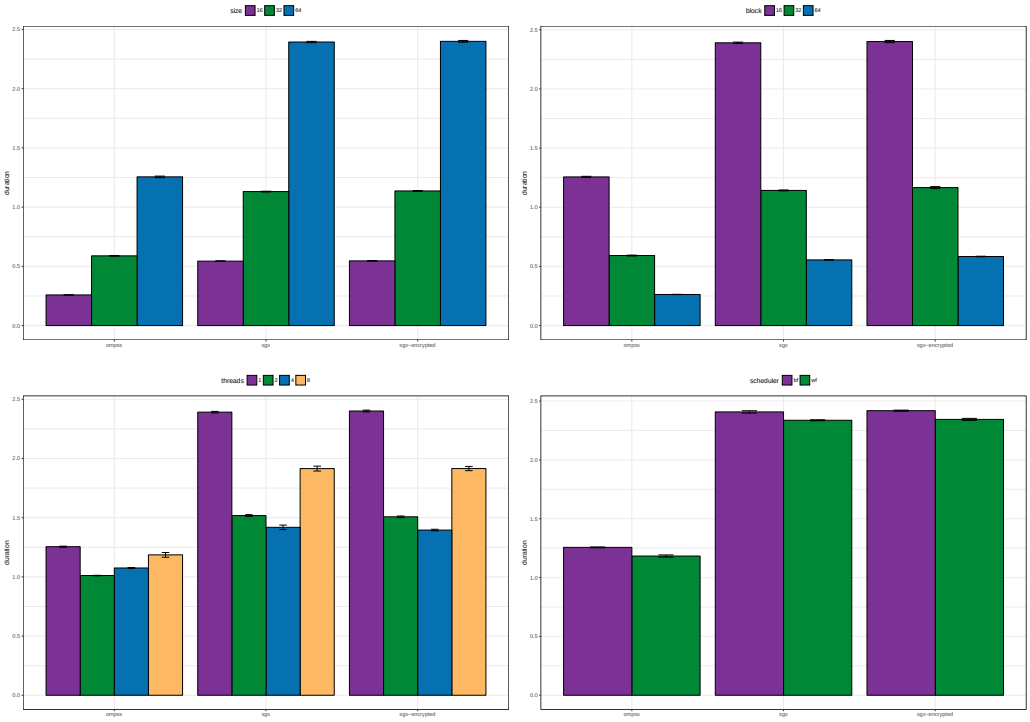


Figure 3.21. Evaluation results for STREAM application (dependency based version).

3.4. IDE Plugin

The integration of OmpSs within Eclipse consists of the development of the Eclipse plugins to display coding hints for the programmer, the OmpSs@FPGA installation within Eclipse Che, and the installation with the Conan Package Manager.

3.4.1. Eclipse plugins

Figure 3.22 shows a sample hint provided by the OmpSs eclipse plugin. Depending on the initial directive inserted, the plugin informs the programmer about how to complete it, through valid clauses and their short manual explanation. The plugin for OmpSs includes all OmpSs directives and clauses, and the Xilinx HLS directives and clauses, which offers an integrated view.

We have also developed a plugin for OpenMP. Figure 3.23 shows the OpenMP example, on which the programmer is presented with the information about the *final* clause on the *task* directive, just before the programmer selects it for insertion in the code.

The Eclipse plugins for OmpSs are provided in the repository: https://github.com/bsc-pm-ompss-at-fpga/eclipse-ompss.

3.4.2. Eclipse Che integration

We have built a tool based on Eclipse Che [5], and Docker to offer the service to compile OmpSs applications for a specific target environment.

3.4.2.1. Initialization of the environment

When initializing the environment, a docker container is created, with an Ubuntu installation, the cross compilers for the Arm and Aarch64 architectures, the Eclipse Che service, and with the OmpSs tools:

- Mercurium compiler
- Ait tool
- Nanos++ runtime library
- Extrae instrumentation library

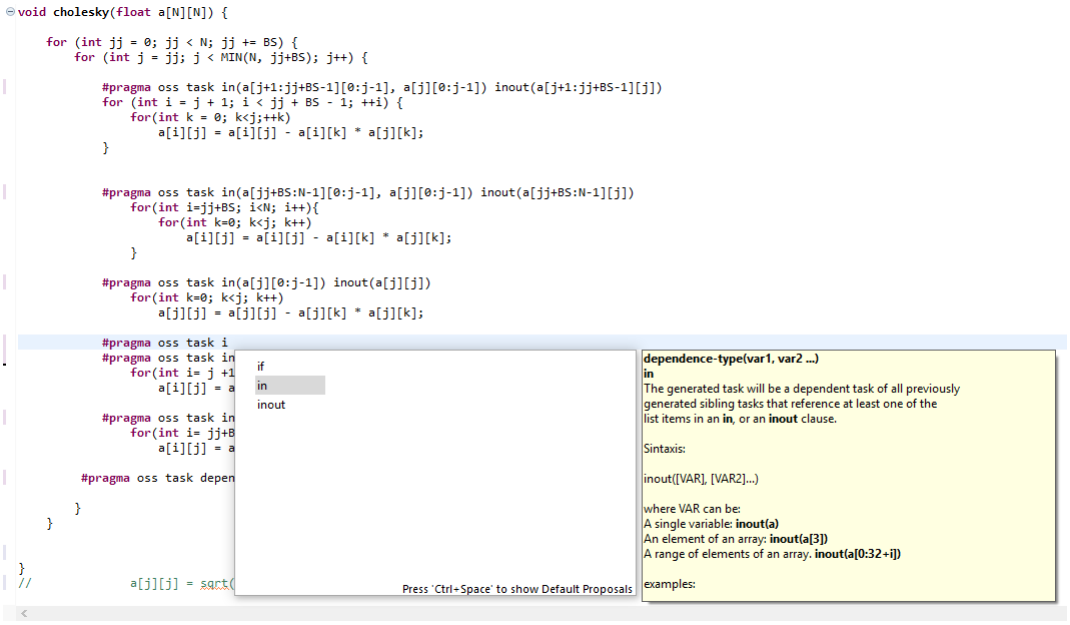


Figure 3.22. IDE plugin showing the OmpSs dependences hints offered to the programmer

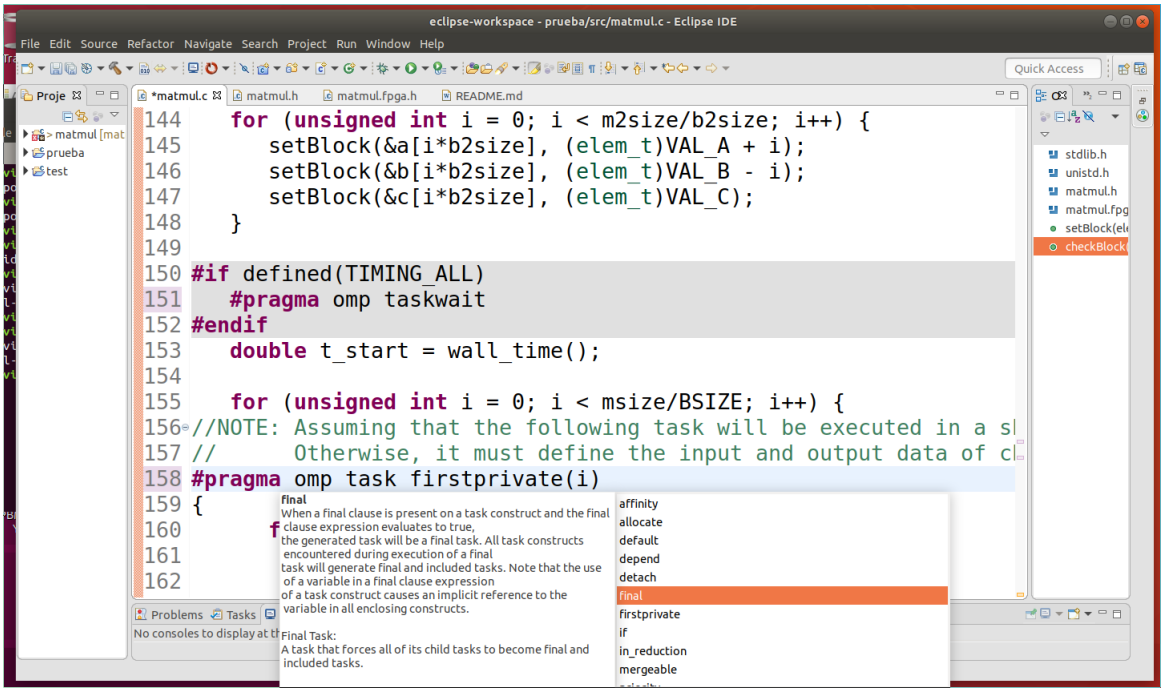


Figure 3.23. IDE plugin showing the OpenMP task hints offered to the programmer

- DMA and Xtasks low-level libraries
- The Linux driver for the FPGA

3.4.3. Working with OmpSs in Eclipse Che

When Eclipse Che has started after initialization of the environment, a browser is started automatically, with the view of the workspaces available. The user can create new workspaces, by selecting the profiles fpga32 or fpga64, in order to have OmpSs@FPGA installed for 32-bit or 64-bit Arm respectivelly. Figure 3.24 shows the initial workspace selection view.

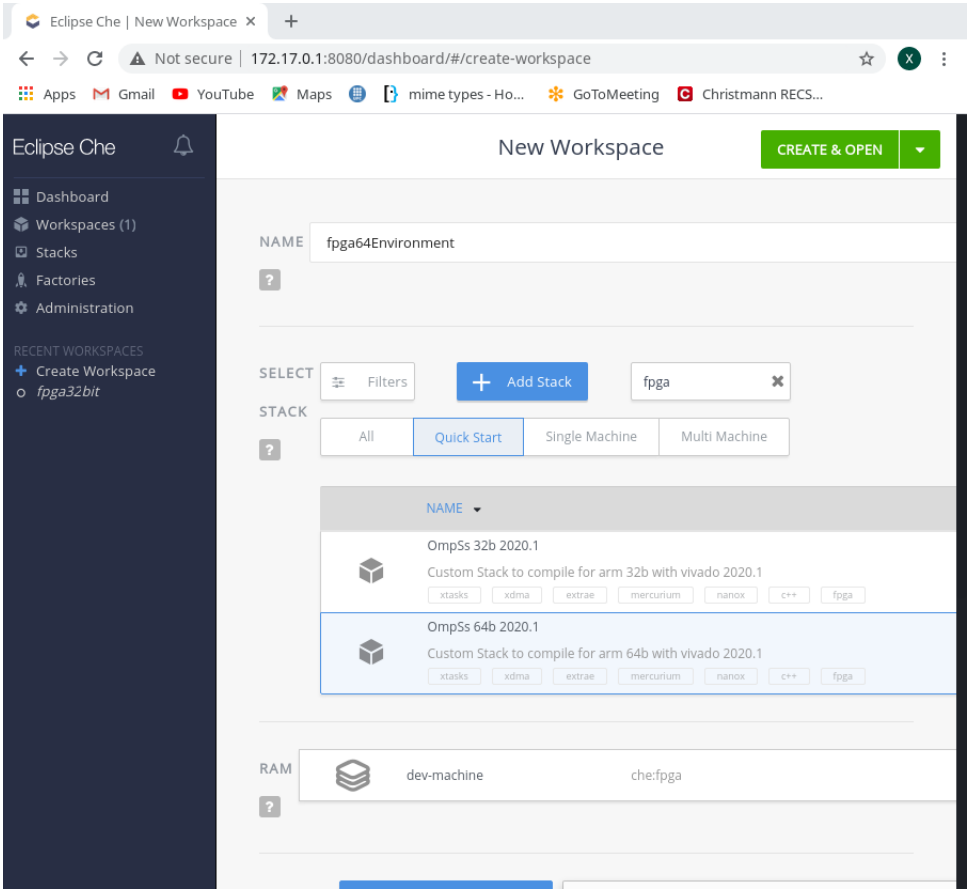


Figure 3.24. Workspace selection in Eclipse Che with OmpSs@FPGA available

When then workspace is created, projects can be imported or created. Figure 3.25 shows the *dotproduct* project open. The session shows the file browser on the left-hand side, a main section on the center and center-right with the code view, and the target machines defined and a terminal session for command line utilities at the bottom. The top level menu is equivalent to the traditional Eclipse menu, allowing the programmer to manage projects, files, and control the build process.

When building a project, Eclipse Che compiles the source files included in the project with the cross-compile tools. Mercurium splits the code following the target directives, and invokes the cross-compiler gcc to generate the binary for the target, and invokes the Xilinx Vivado compiler with the tasks targeting the FPGA, in order to build their IPs and merge them with the FPGA interconnection and the task manager, and generate the bitstream.

The Eclipse Che repository is provided in: https://github.com/bsc-pm-ompss-at-fpga/eclipse-ompss.

3.4.4. Conan-based installations

Conan [4] is a package manager that installs packages on a target directory. Conan checks the compiler/architecture configuration on which it is running and the target environment, and com-

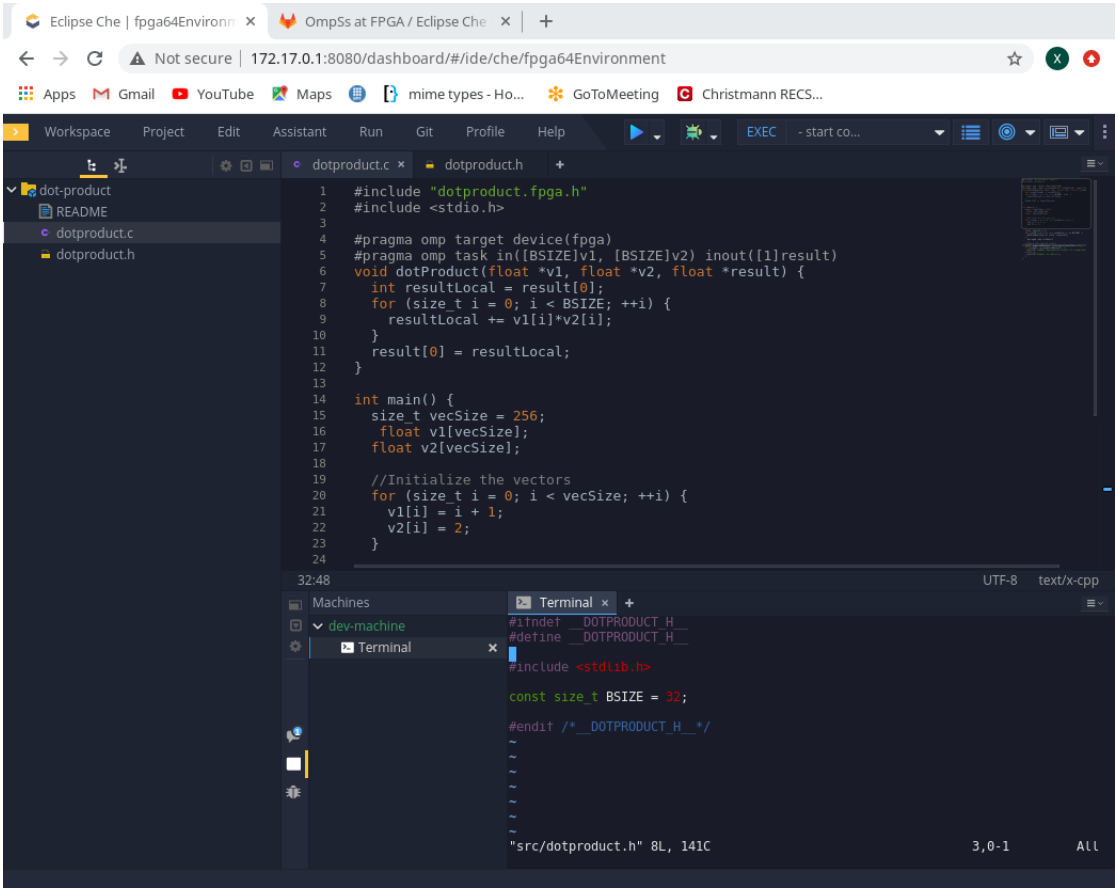


Figure 3.25. Dotproduct project in OmpSs@FPGA open with Eclipse Che

piles the packages from source.

We have taken advantage of Conan to provide a set of Conan scripts that guide the installation of OmpSs@FPGA for the target x86_64, Aarch64, and Arm32 architectures. The development includes scripts for the following components that are generic, but it is preferable to have them configured with a particular setting that we enforce by incorporating them in the compilation phase:

- libz, the library supporting file compression and decompression.
- libxml2, used for Extrae configuration files in XML format
- papi, used for Extrae to collect hardware performance counters
- binutils, used for Extrae to obtain debugging information

When those packages are already available to be used in the OmpSs compilation, the process continues by compiling and installing the OmpSs packages:

- Ait, the Accelerator integration tool for Vivado HLS
- Extrae, the instrumentation library
- libxdma/libxtasks, the low level FPGA support for OmpSs
- Nanos++, the OmpSs-1 runtime library
- Mercurium, the OmpSs compiler

After the compilation is successful a script *activate.sh* is provided to the user, in order to set the proper environment variables (like PATH and LD_LIBRARY_PATH) in order to have access to the new OmpSs installation.

The Conan-based installation scripts are provided in the repository: https://github.com/bsc-pm-ompss-at-fpga/conan-recipes.

4. Dataflow engines

DFiant is a dataflow hardware description language (HDL) that decouples functionality from implementation constraints. DFiant brings together constructs and semantics from dataflow, hardware, and software programming languages to enable truly portable and composable hardware designs. The dataflow model offers implicit concurrency between independent paths while freeing the designer from explicit register placement that binds the design to fixed pipelined paths and timing constraints. DFiant is implemented as a Scala library and offers a rich type-safe ecosystem alongside its own hardware-focused type system (e.g., bit-accurate dataflow types, input/output port types).

For the LEGaTO stack DFiant can be used on platforms with FPGA hardware, where more fine-grain hardware control is required. DFiant can ease both hardware engineers into the LEGaTO stack and software engineers into hardware programming.

4.1. DFiant Evaluation

In this deliverable we introduce the DFiant evaluation while focusing on significant improvements in programmability of FPGA devices without sacrificing performance or fine-grain hardware generation control.

4.1.1. Programmability

We claim that DFiant is highly advantageous when it comes to hardware programmability, yet it is difficult to prove so empirically. The closest empirical study can be found in Section 4.1.1.5, where we compare the Lines of Code (LoCs) between DFiant designs and equivalent non-DFiant implementations. Other sections demonstrate the ease of programmability by showcasing various DFiant features compared to the hardware design status-quo.

4.1.1.1. Mitigating Design Flow Proliferation

Hardware design flow currently suffers from over-proliferation of languages and tooling. Various languages are used for design description (VHDL, Verilog, System Verilog), applying constraints (SDC, XDC, UCF, LPF, LDC), and build scripts (Makefile, Bash, Python, TCL). Tools are varied between vendors and product families, and other third-party tools. This proliferation leads to confusion among new users and inhibit FPGA adoption and build portability. Symbiflow [28] can help reduce tooling proliferation, in an effort to create a “GCC for FPGAs,” but users still require a separate language for constraints and building. Since DFiant is a Scala library, it can also offer a way to deal not only with the design, but with the rest of the build flow. We can bridge this gap by providing a common API to both constrain a design and build it. This advantage is twofold. First, we reduce the entire flow language proliferation. Second, all the properties of our design are now tied together, and can enjoy the same editor exprience, especially when refactoring. For example, say we change a design of a top-level port name. In traditional design flow we may have a mismatch between the design and constraints, but with both set in the same language the compiler and editor can clearly mark the error early.

Figure 4.1 contains an example DFiant script that instantiates a design, compiles it to VHDL 2008, commits it to a folder, prepare the GHDL simulator, and finally runs the simulation. This automation is also available for VHDL 93, Verilog 95/2005 and the modelsim and verilator tools. All that is required to change the language/tool is to change the import statements in the first two lines. This is an example how DFiant-based scripts can unify various tooling and flow to ease hardware design. Additionally, since DFiant can enjoy the entire Scala ecosystem tooling, it can provide nice features such as automatic ip downloading and version management (as a Scala dependency). Also in the future, after migration to the imminent Scala 3 release, DFiant will also benefit from the new python-like indentation-based (braceless) syntax.

4.1.1.2. Bit-accurate Compile-time Type Safety

Type safety improves programmability since it confines the user to provide arguments that only fit the expected type and reports such errors early in the build process. The earlier we detect failures, the better, and for hardware we need to detect bit-accurate type errors. This is why

```scala
import compiler.backend.vhdl.v2008
import sim.tools.ghdl
val topSim =
  new MyDesignSim() //instantiate the design
  .compile          //compile to the imported backend (VHDL2008)
  .toFolder("MyDesignSim") //commit to folder
  .simulation //prepare the simulation for the imported tool (GHDL)
  .run() //run the simulation tool
```

Figure 4.1. Example of a script in DFiant that runs simulation in GHDL

DFiant is equipped with bit-accurate compile-time type safety. In DFiant we actually have two different compile-times. First, since DFiant is a Scala library, a design must initially pass a Scala compiler build process. Second, during Scala runtime, the DFiant compiler runs and completes a series of checks and stages until it produces the required RTL solution. What is currently unique in DFiant is the ability to detect various bit-accurate errors during the early Scala compilation process. This ability, incorporated with an editor that supports the Scala presentation compiler, enables viewing the errors inside the IDE. Figure 4.2 depicts how such errors are marked in an editor.

```
18
19  trait Bad extends DFDesign {
20    val x = DFUInt(8)
21    val badSelection = x.bits(9,3)
22    x := 371
```

Bit index 9 is out of range of width 8

Peek Problem No quick fixes available

Lab0.scala 2 of 2 problems

An assignment operation does not permit a wider RHS expression. Found: LHS-width = 8 and RHS-width = 9

```
23  }
```

Figure 4.2. Example of DFiant bit-accurate compile-time error indications in the editor

4.1.1.3. Three Programming Paradigms in One

We demonstrated in previous deliverables how DFiant provides automatic pipelining for concurrent code. We also presented how finite state-machine (FSM) meta-programming enables to easily construct and concatenate FSMs from sequential-like statements (e.g., `doFor`, `doWhile`, etc.). We also showed that the DFiant compiler has an intermediate presentation (IR) with RTL semantics. This RTL-IR is among the last compilation stages before the backend is applied.

4.1.1.4. Inspected Compilation

DFiant enables inspecting the code at any compilation stage. A simple call to '.printCodeString' applied to the design instance or compilation stage prints out a DFiant-equivalent code representation of that stage. This feature is extremely helpful in understanding the effect of complex stages such as automatic pipelining. Figure 4.3 contains a simple arithmetic-logic unit (ALU) implementation with a manual three-cycle pipelining set for the multiplication operation. Contrarily, consider the design flow in regular HLS tooling. The developer writes code in a software language such as C, and then needs to infer from other tool outputs (timing analysis, RTL output, etc.) what happened after HLS compilation. This is a huge barrier for fine grain performance tuning and debugging.

4.1.1.5. Reducing Lines of Code

DFiant aims to greatly improve designer productivity. To evaluate possible productivity gains we chose various open-source use-cases and implemented them in DFiant: an AES cypher [19], an IEEE-754 double precision floating point multiplier [24], a two-stage RISC-V core [32], a bitonic-sort network [1], a cyclic redundancy check (CRC) generator/checker [14], and other simple examples [2, 31]. We compared the use-cases and our implementations in lines of code (LoC). We documented all results in Table 4.1.

We observe that DFiant code is often significantly more compact than its equivalent RTL implementation (from 50% to 70% less code), because DFiant semantically implies much of the additional information required by the RTL description. The only difference occurs at the RISC-V

```
object ALUOp extends EnumType.Auto {
  final val Or,Add,Mul = Entry()
}
@df class ALU extends DFDesign {
  final val op     = DFEnum(ALUOp) <> IN
  final val arg1   = DFBits(32)    <> IN
  final val arg2   = DFBits(32)    <> IN
  final val result = DFBits(32)    <> OUT
  final val arg1u  = arg1.uint
  final val arg2u  = arg2.uint
  matchdf(op)
  .casedf(ALUOp.Or) {result := arg1 | arg2}
  .casedf(ALUOp.Add) {result := (arg1u + arg2u).bits}
  .casedf(ALUOp.Mul) {result := (arg1u * arg2u).pipe(3).bits} //manual pipeline set
}
import compiler.{Pipelining, PipelineMethod}
new ALU() //instantiate the ALU
  .pipeline(PipelineMethod.Manual) //auto-balance a manual pipeline
  .printCodeString //print the code of the balanced ALU
```

Figure 4.3. ALU manual pipelining example and running compiled code inspection

```
object ALUOp extends EnumType.Auto {
  final val Or,Add,Mul = Entry()
}
@df class ALU extends DFDesign {
  final val op     = DFEnum(ALUOp) <> IN
  final val arg1   = DFBits(32)    <> IN
  final val arg2   = DFBits(32)    <> IN
  final val result = DFBits(32)    <> OUT
  final val arg1u  = arg1.uint
  final val arg2u  = arg2.uint
  matchdf(op.pipe(3)) //balanced
  .casedf(ALUOp.Or) {result := (arg1 | arg2).pipe(3)} //balanced
  .casedf(ALUOp.Add) {result := (arg1u + arg2u).pipe(3).bits} //balanced
  .casedf(ALUOp.Mul) {result := (arg1u * arg2u).pipe(3).bits}
}
```

Figure 4.4. ALU inspected code printout. Notice the addition of balancing pipe stage.

implementation. The reason is that the RTL code design is flat, while we invested substantial code in hierarchical abstractions.

4.1.2. Performance and Other Metrics

Typically, FPGAs are really good platforms for energy efficient, secure, and massively parallel applications. However, they are difficult to adapt generically across various vendor devices. Therefore, once we mitigate the adoption problem and make it easier for developers from both software and hardware worlds to program FPGAs productively, then we also get the other benefits the LEGaTO stacks aims to achieve.

4.2. Integration with OmpSs

Figure 4.5 shows how the programmer provides the DFiant kernels through the DFiant compiler to the Vivado tool, and they are integrated in the FPGA project.

Use Case	Design Source	LoC [#]	LoC Reduction [%]
AES Cypher	**DFiant**	334	64
	Hsing Core	922	
FP Multiplier	**DFiant**	180	47
	Lundgren Core	340	
Two-Stage RISC-V	**DFiant**	557	-79
	Samsoniuk Core	311	
CRC	**DFiant**	41	60
	Drange Core	103	
Fibonacci Gen	**DFiant**	8	74
	ExampleProblems	31	
Sequence Detector	**DFiant**	32	56
	FPGA4Student	73	
Moving Average 4x4	**DFiant**	19	74
	Our RTL	72	
Bitonic Sort Network	**DFiant**	36	60
	VLSICoding	90	
Priority Encoder	**DFiant**	10	52
	Kaufmann Core	21	

Table 4.1. Comparing various RTL codes with equivalent DFiant codes in LoCs
The DFiant implementation is usually much more concise

5. Energy-Efficiency

In this Chapter we present two new research results produced in this Work Package. Section 5.1 presents an analysis of the energy-efficiency of two distributed, hardware-secured services specifically designed for IoT devices. Section 5.2 presents a novel study about how to tune machine learning models' hyperparameters in order to trade performance for energy efficiency. These studies being very recent, their results are not taken into account in the LEGaTO usecases. Nevertheless, we believe that both studies present pertinent results, who will help building LEGaTO-like applications in the future.

5.1. Energy-efficient IoT Applications

Distributed secure services benefit from the increasing availability of hardware-based trusted execution environments (TEE) in mobile, edge and IoT-grade processors. Arm TrustZone [8] and Intel SGX [12] are the most prominent TEEs among those processors. Such TEE-enabled devices can shield applications from powerful attacks, compromised platforms or malicious users. The cumulative availability is paving the way for large-scale deployments of secure services which allows trusted dissemination and processing of confidential and sensitive data.

In this section we report on the energy-efficiency of two distributed secure services which are specifically designed for IoT devices: in subsection 5.1.1 we evaluate the networking performance of secure services and in subsection 5.1.2 we deploy a prototype blockchain network for secure

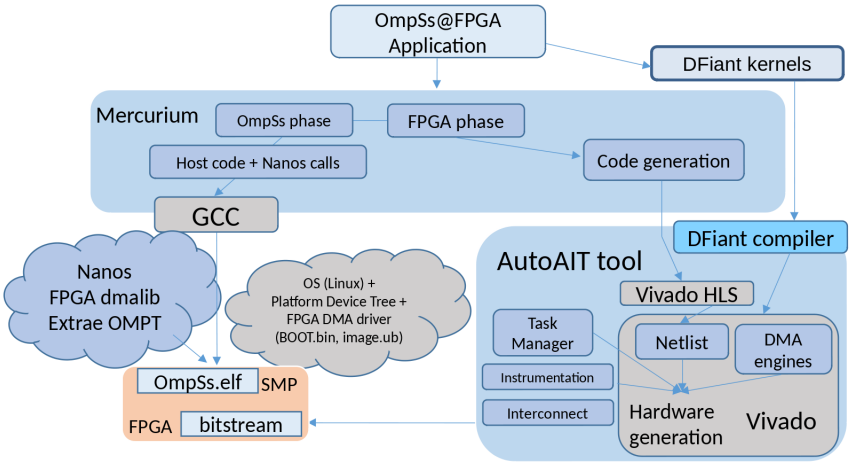


Figure 4.5. Compilation environment showing the integration of the DFiant kernels onto the OmpSs toolchain

smart contract execution.

5.1.1. Networking in ARM TrustZone

IPERFTZ [17] is an open-source tool for secure services running on ARM TrustZone. It provides information on network performance that is necessary in order to uncover bottlenecks in the design and implementation of secure services. Our micro-benchmark expands on the results presented in [18] and reported in *D4.2 "First release of energy-efficient, secure, resilient task-based programming model and compiler extensions"*.

5.1.1.1. Networking for Secure Services in OP-TEE

At this point we refer the reader to D4.2 section 5.3.1.2 where we have already outlined the ARM TrustZone architecture as well as the GlobalPlatform APIs. In this section we describe the networking capabilities of secure services, also known as trusted applications (TA) in OP-TEE [1].

TAs can receive their network configuration or network parameters from their normal world counterpart using shared memory. The GlobalPlatform proposes two different approaches for TEEs to implement a network stack: (1) the TEE borrows the network stack of the rich execution environment (REE) or (2) the TEE relies on trusted device drivers. The former approach requires the presence of a supplicant in the REE (an agent that responds to requests from the TEE) in order to invoke remote procedure calls (RPC) on the borrowed network stack. The later requires a full network stack implementation in the secure world. Clear tradeoffs of these two approaches are security in the form of trusted computing base and confidentiality, performance in terms of latency and portability. Both approaches have their advantages and disadvantages and should be used based on the requirements of a secure service. For the remaining sections we will only refer to the first approach borrowing the network stack.

Figure 5.1 visualizes the interaction between secure and normal world in OP-TEE starting from a secure service network request. The secure world hosts the TA, which interacts directly with `libutee` (Figure 5.1-❶). When using GlobalPlatform's Socket API [16], `libutee` does a system call (Figure 5.1-❷) to OP-TEE. OP-TEE then delegates the request to the socket pseudo TA (PTA, Figure 5.1-❸). The secure monitor is invoked through a SMC (Figure 5.1-❹), which maps the data from the TEE to the REE's address space. From there execution switches into the normal world and the OP-TEE driver (Figure 5.1-❺) resumes operation. Requests are then handled by the supplicant (Figure 5.1-❻). The agent executes system calls using `libc` (Figure 5.1-❼) to directly relate the underlying network driver (Figure 5.1-❽) over the POSIX interface. Once data reaches the network driver, it can be sent over the wire (Figure 5.1-❾). Whenever data is sent or received using this approach, it traverses all exception levels, both secure (from EL0 up to EL3) and non-secure (from EL2 to EL0 and back up).

[1] `https://www.op-tee.org/` accessed on 28.10.2020

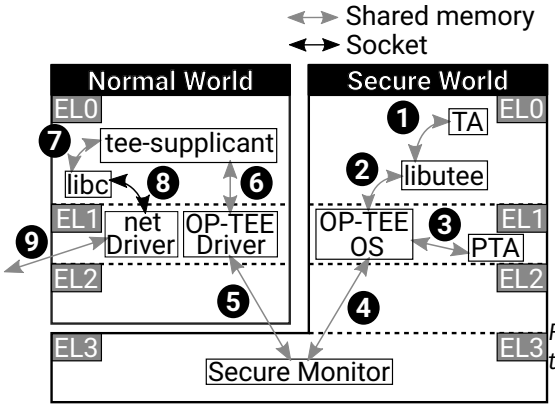


Figure 5.1. Execution flow inside OP-TEE

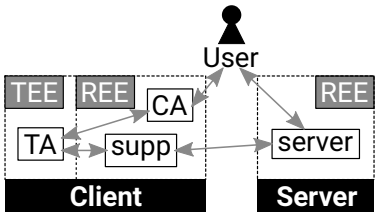


Figure 5.2. Interaction of iperfTZ's components in the client-server model.

5.1.1.2. Architecture of IPERFTZ

iperfTZ consists of three components: (1) the client application (CA), (2) the TA, and (3) the server. Together, these three components form a client-server model as shown in Figure 5.2 which allows exchanging network packets between the server and the TA. In the following paragraphs we explain each component individually.

Client application. The CA is the user interface to the TA running in the normal world. It uses two shared memory areas to (1) exchange arguments passed over the command line interface with the TA and (2) to retrieve metrics gathered by the TA during the network measurement. Most notable arguments exchanged with the TA are the IP address of the server component, the dummy data size, and the socket buffer size. Depending on the arguments, iperfTZ will run a network throughput measurement while either maintaining a constant bit rate, transmitting a specific number of bytes or running for 10 seconds.

Trusted application. According to the GlobalPlatform standard, TAs can only take the role of a client in a client-server model. Therefore, the TA has to open a client connection to a running server. As soon as the connection is established, the measurement starts. During the measurement, the TA will gather metrics on the number of transmit calls, bytes sent, time spent in the transmit calls and the total runtime.

Server. The server is deployed in a REE waiting for incoming network connections to start a network measurement. Similar to the TA, the server will gather metrics and additional data not accessible to a TA during the measurement.

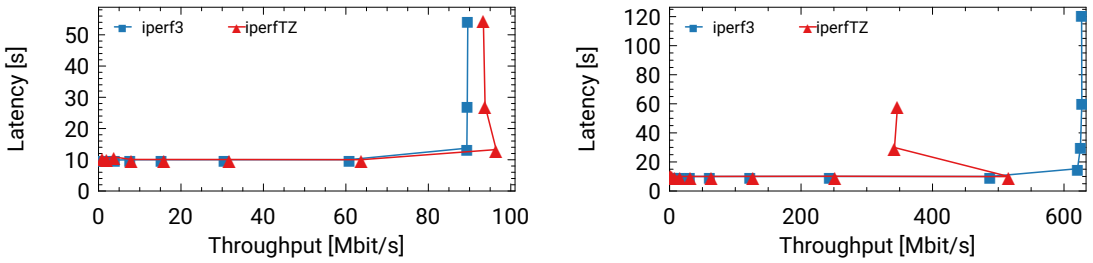
5.1.1.3. Evaluation

We have evaluated iperfTZ's performance and energy consumption to iperf3's [2] on real and emulated hardware. The CA and the TA were deployed on a Raspberry Pi [3] as well as QEMU [4] emulating a Raspberry Pi. Due to network bandwidth limitations of the Raspberry Pi we included emulation using QEMU into our evaluation.

Figure 5.3 shows that iperfTZ in general exceeds on both setups the network throughput of iperf3 due to its simplistic nature. As expected, we were not able to observe any degradation of the network throughput on the Raspberry Pi due to an overhead from frequently switching between secure and normal worlds, as in [18]. However, by lifting the network bandwidth limitation using QEMU we observe a serious degradation of the network throughput when trying to achieve Gbit s^{-1} bit rate. Network throughput beyond 500 Mbit s^{-1} is strongly affected by a world switching overhead, even degrading beyond unaffected throughput rates.

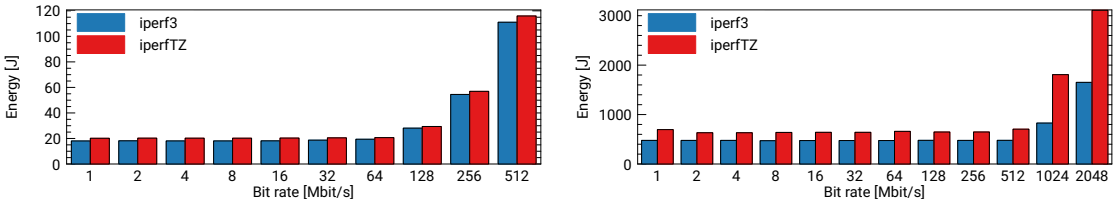
In Figure 5.4 we show the energy consumption of both setups. Before reaching the saturation point, iperfTZ consumes about 2 J (11 %) respectively 173 J (36 %) more energy than iperf3. The additional energy consumption if iperfTZ can be attributed to the execution in the TEE and cer-

[2] `https://software.es.net/iperf/` accessed on 28.10.2020
[3] `https://www.raspberrypi.org/products/raspberry-pi-3-model-b` accessed on 28.10.2020
[4] `https://www.qemu.org` accessed on 28.10.2020



(a) Raspberry Pi (b) QEMU

Figure 5.3. TCP network throughput measurements for 128 KiB *buffer sizes.*



(a) Raspberry Pi (b) QEMU

Figure 5.4. Energy consumption during TCP network throughput measurements. Bit rates on the x-axis are given in logarithm to base 2.

tainly also to the world switching overhead under a saturated system. With our evaluation we were able to quantify the world switching overhead by a factor of 1.7 ×.

5.1.2. Executing Smart Contracts in ARM TrustZone

TZ4Fabric [27] is an open-source extension to Hyperledger Fabric [5] blockchain framework that enables execution of smart contracts with ARM TrustZone. By exploiting the TEE, TZ4Fabric can introduce confidentiality guarantees to protect the smart contract logic as well as the processed data from malicious attackers.

5.1.2.1. Architecture of TZ4Fabric

TZ4Fabric leverages the OP-TEE framework to exploit ARM TrustZone. The design of TZ4Fabric is strongly influenced by Fabric Private Chaincode [10] which exploits Intel SGX to shield smart contract execution. TZ4Fabric is divided into three components as shown in Figure 5.5: (1) a wrapper that facilitates communication with (2) the proxy using gRPC [6] and the ledger. The proxy is deployed in the normal world and provides an interface for the wrapper to the (3) chaincode (Hyperledger Fabric term for smart contract) running as TA. In order to simplify the design we separate the proxy from the wrapper. However, these two components could be integrated into the same TrustZone-enabled device. The chaincode, the client, the orderer and the ledger are integral components of Hyperledger Fabric and constitute a blockchain network.

[5] `https://www.hyperledger.org/use/fabric` accessed on 28.10.2020
[6] `https://grpc.io/` accessed on 28.10.2020

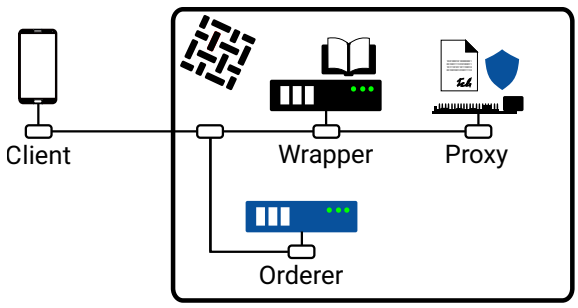


Figure 5.5. Architecture of TZ4Fabric

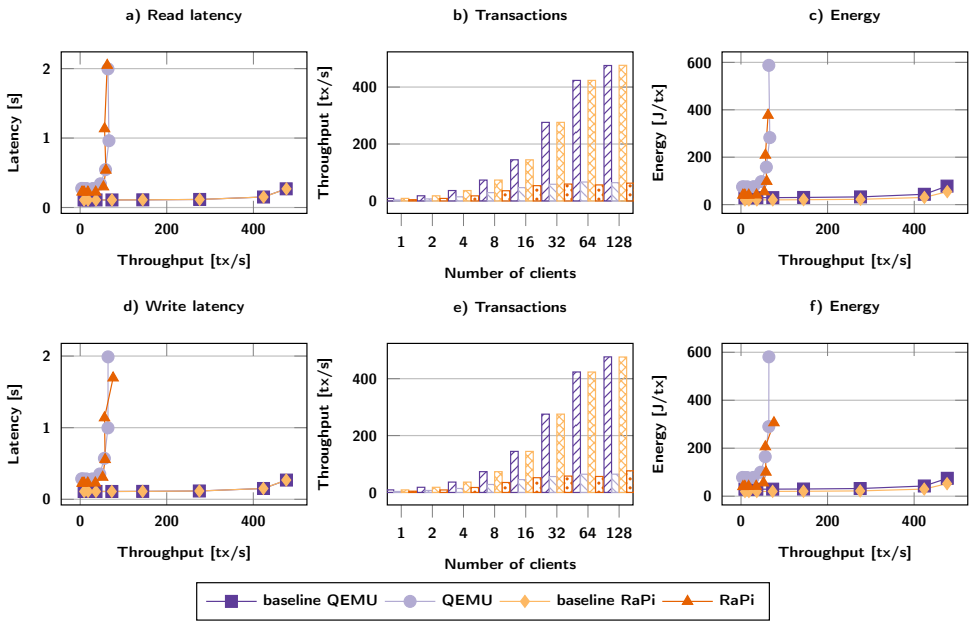


Figure 5.6. Throughput-latency, transactions and transaction energy for read/write invocations. Top row are read transactions, bottom row are write transactions.

5.1.2.2. Evaluation

We evaluate our prototype in a small-scale deployment of 27 machines in a Hyperledger Fabric blockchain network. The orderer, 8 wrapper and 8 proxy (QEMU instance emulating a Raspberry Pi) nodes each run on their own physical server. As IoT devices we deploy 8 Raspberry Pi 3B+ as proxy nodes. In our benchmark we use a simple chaincode which tracks the coffee consumption in an office. Clients submit transactions to track their coffee consumption and query the current coffee statistics.

In a benchmark, with results shown in Figure 5.6, we evaluate the throughput and latency of our prototype. Up to 128 clients repeatedly invoke read and write transactions over a duration of 5 minutes. During the baseline measurement the chaincode is run in the normal world instead of the secure world.

For the read and write baselines we observe that both are very similar to each other. With the baseline measurements we are not able to saturate the system. However, with the ARM TrustZone-enabled environments we consistently observe saturation of the system beginning at 8 clients around 65 tx/s. This means that client transactions on the chaincode are essentially invoked sequentially. The reason for this low throughput is twofold: (1) the substantial overhead due to shielding TAs with ARM TrustZone and (2) potential performance and reusability improvements for TAs running as chaincodes. In general we observe an increase in latency by a factor of 2.3 before reaching the saturation point, which is in line with the world switching overhead described in subsection 5.1.1.

During the benchmark we recorded the energy consumption shown in Figure 5.7 of the nodes in the Hyperledger Fabric blockchain network. The energy consumption of the nodes is rather stable and only slightly increasing with the number of clients. We highlight two deviations from our expectations: (1) the orderer and wrapper energy for the baselines rise (Figure 5.7-[a,b,d,e]) as well as (2) the energy of the proxy (Figure 5.7-[c,f]).

The former indicates the saturation of the Hyperledger Fabric network. This assumption is confirmed by the decline of average transactions per client in the network and the approach to the saturation point seen in Figure 5.6. The later is a result of the world switching overhead when chaincodes are invoked. As a result the performance governor in the operating system has to operate the CPU for longer intervals at higher operating points (*i.e.*, voltage and frequency). This is better seen in Figure 5.6-[c,f], where the average energy per transaction increases proportionally to the number of clients. On the Raspberry Pi the energy consumption increases by a factor

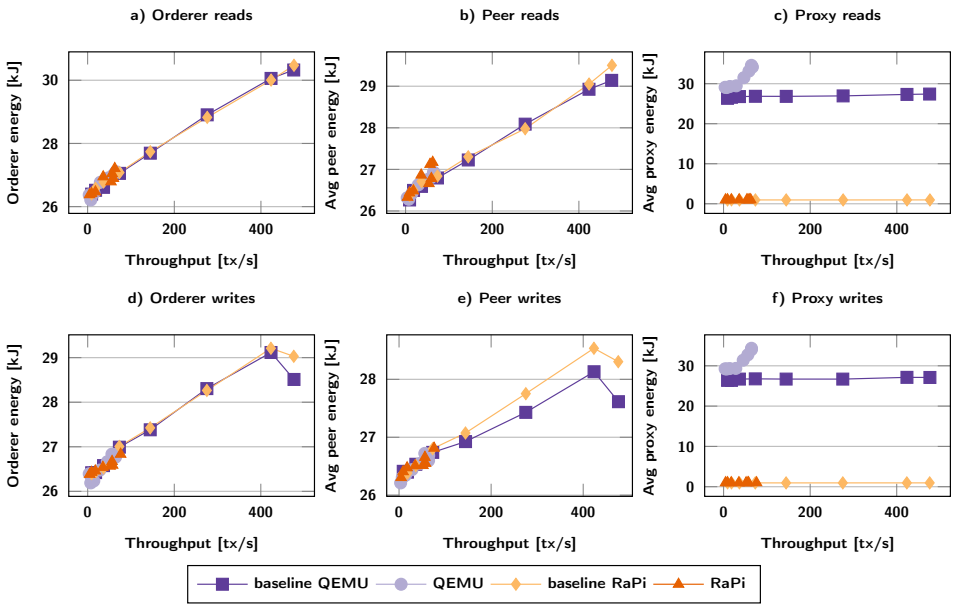


Figure 5.7. Energy consumption of nodes in the Hyperledger Fabric network. Top row are read transactions, bottom row are write transactions.

of 1.05 and by a factor of 1.25 with QEMU beyond the saturation point.

5.1.3. Achievements

To the best of our knowledge we have developed the first network performance tool called iperfTZ for secure services in OP-TEE exploiting ARM TrustZone. iperfTZ is an open-source prototype [7] that is capable of identifying bottlenecks in the network performance of secure services. Furthermore, with iperfTZ we were able to identify and quantify the energy and performance world switching overhead of networked secure services. Our research results were published and presented at SSS'19 in Pisa, Italy.

TZ4Fabric is a novel approach to bring support for executing smart contracts in Hyperledger Fabric on OP-TEE using ARM TrustZone. With the TZ4Fabric prototype we have shown that it is possible on ARM hardware to reduce the energy cost by one order of magnitude compared to state-of-the-art hardware used with Hyperledger Fabric. However, the improved energy consumption comes at the cost of performance, which is also a result of the additional security when running the service in OP-TEE or ARM TrustZone in general. Our research results were published in presented at SRDS'20 in Shanghai, China.

5.2. Energy Efficiency Through Deep Learning Parameter Tuning

Deep neural network (DNN) learning jobs are common in today's clusters due to the advances in AI driven services such as machine translation and image recognition. The most critical phase of these jobs for model performance and learning cost is the tuning of hyperparameters. Existing approaches use techniques such as early stopping criteria to reduce the tuning impact on learning cost. However, these strategies do not consider the impact that certain hyperparameters and systems parameters have on training time. To tackle this problem, we propose PipeTune, a framework for DNN learning jobs that addresses the trade-offs between these two types of parameters. PipeTune takes advantage of the high parallelism and recurring characteristics of such jobs to minimize the learning cost via a pipelined simultaneous tuning of both hyper and system parameters. Our experimental evaluation using three different types of workloads indicates that PipeTune achieves up to 22.6% reduction and $1.7\times$ speed up on tuning and training time, respectively. PipeTune allows for improving performance and, in our experiments, lowered energy consumption up to 29%.

[7] https://github.com/legato-project/iperfTZ

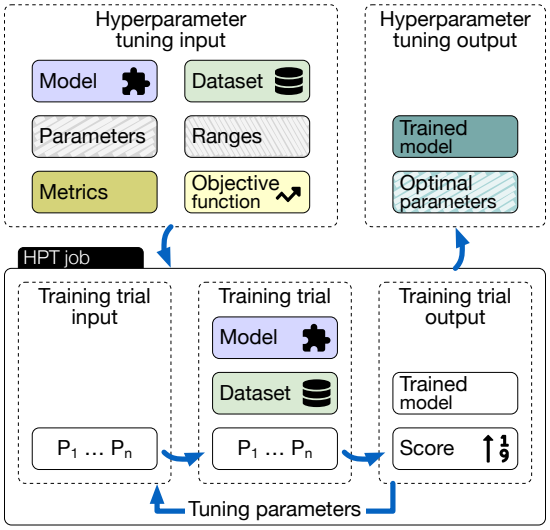


Figure 5.8. Hyperparameter tuning flow.

5.2.1. Problem statement

One of the first challenges of applying deep learning algorithms in practice is to find the appropriate hyperparameter values for a given workload. We assume that most DNN tuning jobs make use of some existing hyperparameter optimization solution. In the following we refer to these types of jobs as HPT Jobs (i.e., Hyperparameters Tuning Jobs).

A given HPT Job takes as input a given workload, a set of parameters, its respective set of range values, an objective function and the metric of interest (e.g., accuracy, performance, energy). This job spawns a collection of *training trials* based on the possible values of the parameters, following a given search algorithm (e.g., GridSearch, HyperBand). Each *training trial* takes as input the workload and a set of fixed values for the parameters of interest, where these values belong to their respective given ranges. These trials can run either sequentially or in parallel depending on the setup. They produce a trained model and a score for the given parameters values. Scores correspond to the metric of interest defined by the user. The optimal set of parameters values is chosen by applying the objective function to the scores. Figure 5.8 illustrates this process.

We consider a deep learning cluster consisting of N nodes, each containing C cores and M GB of memory. Note that despite a common trend to include GPUs in DNN clusters, we explicitly put aside this option. We do this given the (rather small) nature of jobs on which we focus, for which commodity machines are sufficient for training. HPT Jobs are scheduled in a FIFO manner. We categorize these jobs in the following two main types: Type-I: tuning the same model for different datasets (e.g., recommendation engines), and Type-II: tuning different models for the same dataset (e.g., computer vision).

Both types of tuning jobs can still be divided into two sub-types: *(a)* same set of hyperparameters and ranges, and *(b)* same set of hyperparameters but different ranges. Each job, independent of its category, performs the earlier described tuning process from scratch. **A key observation is that these jobs could benefit from previously computed results for other jobs in the same category to converge faster**. Moreover, training trials spawned by the same HPT Job run all with the same system parameters even though they might require different resources configuration.

Another major limitation of the currently available approaches to hyperparameter auto-tuning is that only a single objective metric can be specified. This means that for a given HPT Job, one could choose to optimize either accuracy or performance, but not both simultaneously.

In summary, our problem's input consists of an HPT Job with the objective of achieving either maximum accuracy, or maximum accuracy with minimum training time. The former must output the best possible hyperparameters leading to the highest accuracy, independent of training time. For the latter, a combination of optimal hyper and system parameters is expected which leads to the highest accuracy and lowest training time. Note that for both scenarios, a shorter tuning

Algorithm 1 PipeTune algorithm.

Function *train(model, data, hyperparameters)***:**
    job = ***async*** *model.train(data, hyperparameters)* ***async*** *tuneSystem(job)* job.wait() **return** model
PipeTuneTunetuneSystem(model, data) profile = *getProfile(job)* (score, config) = *getSimilarity(profile)* **if** *score > threshold* **then**
    setSystemParameters(config)
else
    foreach $sp_v \in$ *systemParameters* **do**
        setSystemParameters(sp_v) wait until epoch finishes add collected metrics to m
    bestConfig = find best config in m *setSystemParameters(bestConfig)*

Table 5.1. Workloads used for experiments.

	Model	Dataset	Datasize	Train Files	Test Files
Type-I	LeNet5	MNIST	12 MB	60 000	10 000
	LeNet5	Fashion-MNIST	31 MB	60 000	10 000
Type-II	CNN	News20	15 MB	11 307	7538
	LSTM	News20	15 MB	11 307	7538
Type-III	Jacobi	Rodinia	26 MB	1650	7538
	SPK-means	Rodinia	26 MB	1650	7538
	BFS	Rodinia	26 MB	1650	7538

time is beneficial, as allowed by our approach.

5.2.2. Implementation

PipeTune implements Algorithm 1 in Python (v3.5.2) and it consists of 947 LOC. We leverage two open-source projects, namely Tune and BigDL. Tune [23] is a Python library for hyperparameter search, optimized for deep learning and deep reinforcement learning [22]. Tune provides several trial schedulers based on different optimization algorithms. While we select HyperBand for the reminder of this work, Tune allows to switch among the available ones, as well as to implement new ones. As a consequence, PipeTune indirectly supports all its hyperparameter optimization algorithms.

The training applications are executed by BigDL [13], a distributed deep learning framework on top of Apache Spark. BigDL supports TensorFlow and Keras, hence PipeTune supports models defined using such frameworks. The Ground Truth module is based on a battle-tested k-means implementation openly available in the *scikit-learn* machine learning library for Python [30].

Finally, as storage backend, we leverage `InfluxDB` (v1.7.4), an open-source time series database. It offers a convenient `InfluxDB-Python` client for interacting with InfluxDB which we use to query information regarding the collected system metrics. PipeTune is released as open-source [8].

5.2.3. Evaluation

This section presents our in-depth evaluation of PipeTune using real-world datasets. Our main findings are:

1. PipeTune achieves significant tuning speedups without affecting model performance (i.e.,

[8] https://github.com/isabellyrocha/pipetune

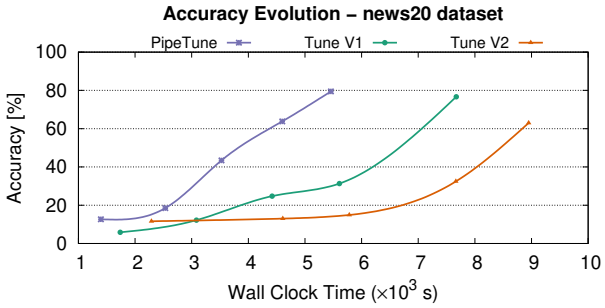


Figure 5.9. Accuracy convergence.

accuracy);

2. By speeding up the tuning process, we also have a more energy efficient approach, not only due to the runtime reduction but also because of the more efficient utilization of system resources;
3. The proposed approach is sensitive to varying system loads as this is also reflected on the events used to profile and our system adapts on a fine granularity (i.e., epochs level).

5.2.3.1. Experimental Setup

We deployed our experiments using Type-I and Type-II workloads on a cluster of 4 quad-socket Intel E3-1275 CPU processors with 8 cores per CPU, 64 GiB of RAM and 480 GB SSD drives. Experiments involving Type-III workloads are deploy on a single node containing an Intel E5-2620 with 8 cores, 24 GB of RAM and a 1 TB HDD. All machines run Ubuntu Linux 16.04.1 LTS on a switched 1 Gbps network. Power consumption is reported by a network connected LINDY iPower Control 2x6M Power Distribution Unit (PDU), which we query up to every second over an HTTP interface to fetch up-to-date measurements for the active power at a resolution of 1W and 1.5% precision.

We consider 7 state-of-the-art deep learning workloads for image classification, LLC-Cache computational sprinting and natural language processing. Table 5.1 summarizes their details.

LeNet5 [21] is a convolutional network for handwritten and machine-printed character recognition. Convolutional Neural Networks (CNNs) [25] are a special kind of multi-layer neural networks, trained via back-propagation. CNNs can recognize visual patterns directly from pixel images with minimal preprocessing. Long Short-Term Memory (LSTMs) [15] are artificial Recurrent Neural Networks (RNNs) architectures used to process single data points (such as images, connected handwriting recognition and speech recognition), as well as sequences of data (i.e., speech, videos). Finally, Jacobi is a differential numerical solver, BFS is breath-first-search and spk-means is k-means implemented on top of Spark framework.

The MNIST dataset [20] of handwritten digits has a training set of 60 000 examples, and a test set of 10 000 examples. The digits have been size-normalized and centered in a fixed-size image. Fashion-MNIST dataset [35] is a dataset of article images consisting of a training set of 60 000 examples and a test set of 10 000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Fashion-MNIST shares the same image size and structure of training and testing splits as the original MNIST dataset. The News20 dataset [3] is a collection of 20 000 messages collected from 20 different netnews newsgroups. We sample uniformly at random 1000 messages from each newsgroup, and we partition them by name. The Rodinia Benchmark Suite [11] is a collection of profiling short-term resource allocation (i.e., computational sprinting) policies which targets heterogeneous computing platforms with both multicore CPUs and GPUs. These workloads have the objective to classify or predict the original data reserved for testing purposes.

There are several potential hyperparameters to tune. For practical reasons, in our evaluation we select the 5 described below. Note that their recommended range is typically application-driven, and we settle on specific values without however generalizing for any workload.

1. **Batch size.** Number of samples to work through before updating the internal model param-

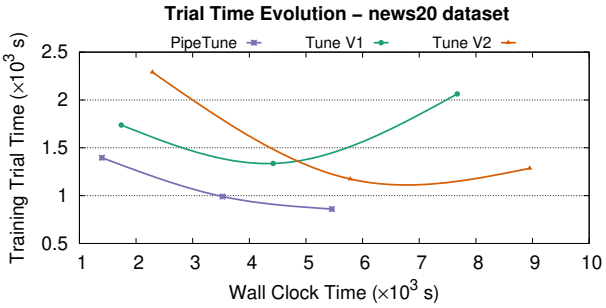


Figure 5.10. Training trial time convergence.

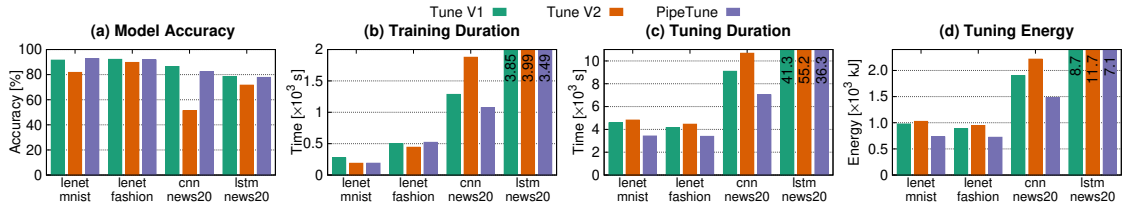


Figure 5.11. Evaluation of PipeTune's accuracy, performance and energy consumption for Type-I and Type-II Jobs.

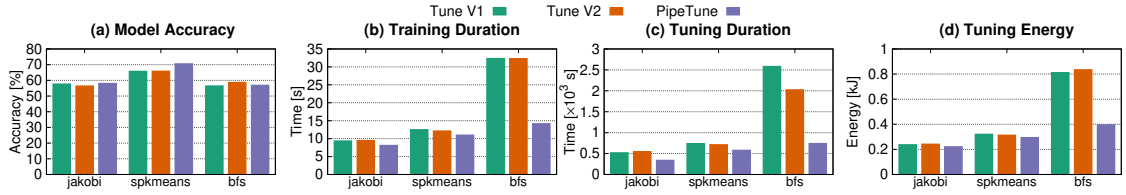


Figure 5.12. Evaluation of PipeTune's accuracy, performance and energy consumption for Type-III Jobs.

eters. Large values for batch size have a negative effect on the accuracy of network during training, since it reduces the stochasticity of the gradient descent. Range: [32 - 1024].

2. **Dropout rate.** Dropout randomly selects neurons to be ignored during training. Dropout layers are used in the model for regularization (i.e., modifications intended to reduce the model's generalization error without affecting the training error). The dropout rate value defines the fraction of input to drop to prevent overfitting [26]. Range: [0.0 – 0.5].

3. **Embedding dimensions.** Word embeddings provide a mean of transfer learning. This mechanism can be controlled by having word vectors fine-tuned throughout the training process. Depending on the dataset size on which word embeddings are being refined, updating them might improve accuracy [7]. Range: [50 – 300].

4. **Learning rate.** Rate at which the neural network weights change between iterations. A large learning rate may cause large swings in the weights, making it impossible to find their optimal values. Low learning rates requires more iterations to converge. Range: [0.001 - 0.1].

5. **Number of epochs** Number of times that the learning algorithm will work through the entire training dataset. Typically, larger number of epochs yields longer runtimes but also higher training accuracy. However, the number of epochs required to achieve a given minimum desired accuracy depends on the workload. Range: [10 - 100].

For the purpose of this evaluation, we restrict the list of parameters to number of cores and memory. However, the same mechanisms can be applied to any other parameter of interest (e.g., CPU frequency, CPU voltage). In our cluster, the ranges of valid values for system parameter tuning are [4 - 16] and [4 - 32] (GB) for for number of cores and memory, respectively.

We define as our baseline (Tune V1) a system tuning hyperparameters and ignoring any system parameter. We rely on HyperBand for the parameter optimization with the objective function set to maximize accuracy. We then include the list of system parameters to be considered in the list of parameters to be tuned by the HyperBand algorithm (Tune V2). We also include the training duration as part of the optimization function which in this baseline is set to maximize the ratio accuracy to duration.

5.2.3.2. Convergence Evolution

In order to build our initial similarity model we rely on profiling data of the workloads described in Table 5.1. For each workload, we vary the system configurations as follows. Memory allocation can be 4GB, 8GB, 16GB, and 32GB. The total number of cores that could be allocated were 4, 8, or 16. Finally, batch size could take the values 32, 64, 512, or 1024. In total, this sums up to 48 different configurations for each workload. There is no reason to expect variations in the data collected from different training instances using the exact same parameters. However, we repeat this process twice for each configuration to make sure that the achieved model is not affected by potential unseen variations.

We begin our evaluation by analyzing the convergence trajectory of PipeTune compared to Tune V1 and Tune V2. 5.9 illustrates the accuracy evolution of the training trials over the tuning time of a CNN model on the News20 dataset. We observe that PipeTune converges to an accuracy value comparable to Tune V1 but at a much faster rate. For instance, PipeTune reaches a 60% accuracy after approximately 4500 seconds. On average our approach is $1.5\times$ and $2\times$ faster than Tune V1 and Tune V2, respectively.

The training time achieved shows similar behavior (see 5.10). Interestingly, Tune V1 performs worse than Tune V2. Since Tune V1 optimizes only for accuracy, the most accurate model not necessarily achieves the shortest training time. On the other hand, as Tune V2 optimizes for the ratio accuracy to performance, the accuracy achieved might not be the highest possible. However, the training time in the given configurations might be lower (which is exactly what happens in this instance of the problem). Finally, we observe that PipeTune consistently presents shorter trial times than the other two approaches during the entire tuning process.

5.2.3.3. Single-Tenancy

We now consider a single-tenancy scenario, and assume each HPT Job runs in a dedicated cluster, where the required resources demanded by the system parameters are available and exclusive for a given tenant. This prevents interference caused by other jobs co-located on the same cluster. However, as a given HPT Job spawns several *training trials* asynchronously, the cluster still remains shared among these sub jobs. We evaluate how PipeTune performs in such stable setting, comparing it against Tune V1 and Tune V2, for all the workloads.

Comparison with baseline. 5.11 presents the results of model accuracy, training and tuning runtime, and overall cluster energy consumption of offline HPT Jobs for the different workloads described in Table 5.1.

5.11 (a) presents the accuracy results. We can observe that the accuracy of PipeTune is not affected by the performance optimization. In fact, results are on par with Tune V1, where hyperparameters tuning is done with the only objective of maximizing accuracy. As expected, Tune V2 decreases accuracy up to 43%, since the objective function no longer tries to optimize accuracy but also takes the runtime into account.

5.11 (b) shows the training time of the achieved model. In this case, PipeTune presents comparable results to the baseline. In fact, we observe up to $1.7\times$ speed-up in comparison with Tune V2 which focuses exactly in reducing training runtime. We observe that Tune V2 increases tuning duration by up to 18% when compared to Tune V1. This happens for the following two reasons. First, the search space of Tune V2 is larger than of Tune V1, as it includes the system-parameters. Second, the optimization function consists of accuracy and runtime together. These two reasons make it harder for the search algorithm to find the optimal set of configurations, hence longer tuning times are observed.

On the other hand, PipeTune reduces tuning runtime by at least 18% when compared against Tune V1, as shown in 5.11 (c). This performance gain is obtained because the search space and

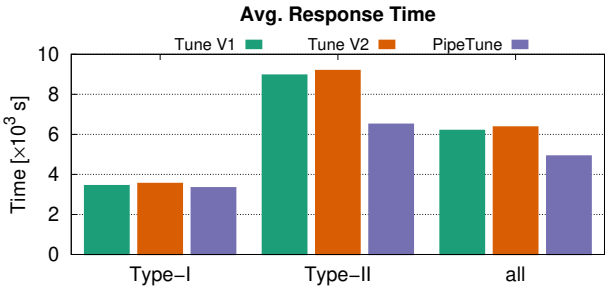


Figure 5.13. Average response time for Type-I and Type-II Jobs considered independently and all together.

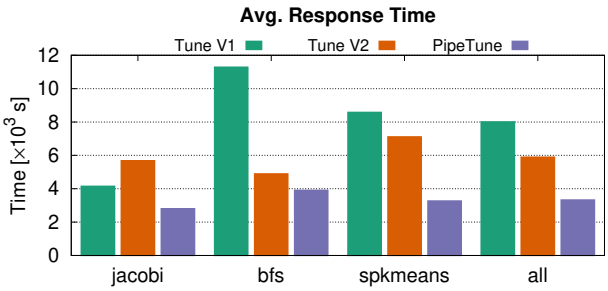


Figure 5.14. Average response time for Type-III Jobs.

optimization function remains the same, and at the same time PipeTune finds and applies during runtime the optimal system configurations for each trial. Moreover, all the additional steps introduced by PipeTune are done in parallel, without impacting the hyperparameters tuning process.

5.11 (d) reports the energy results. The overall energy consumption of the cluster is directly affected both by the performance decays and gains. Compared against Tune V1, we observe up to 22% energy increase for Tune V2 and up to 29% energy decrease for PipeTune.

5.12 compares Tune V1, Tune V2 and PipeTune on a single node. The Type-III workloads used in these experiments have shorter epochs and each a different CNN model. Previous experiments deploy PipeTune on workloads with epochs lasting minutes. Long epochs work in favor of PipeTune since low-overhead profiling is performed across the first couple of epochs to classify new workloads. Therefore, next we perform an extra analysis on Type-III Jobs which present this more challenging setup for PipeTune to observe how it behaves.

5.12 (a-d) plots the same metrics as seen in 5.11. The goal is to test how well PipeTune can improve tuning for workloads with short but many epochs per trial. Here we can observe that PipeTune also achieves the expected results in this more challenging scenario and reduces both training and tuning time when compared to the baseline systems. Regarding model accuracy, we can also see that our approach achieves comparable or better results than the baseline. Finally, the energy results reflects the performance gains resulting in a more energy efficient approach as well.

To summarize, for these single-tenancy scenarios, PipeTune presents better performance with up to 23% reduction on tuning time, is more energy efficient reducing up to 29% the overall energy consumption of the utilized cluster, and does not affect model accuracy as the observed differences in this aspect are negligible.

Profiling overhead. Profiling is a fundamental part of our system design and essential for the decision making process. During the profiling of a given epoch, the extra computation introduce additional load, depending on the system configuration. However, as this profiling overhead only occurs in the epoch granularity and does not apply for all the epochs, the performance benefits resulting from tuning the system-parameters overtake the measured overhead. The experimental results presented above also support these assumptions as, otherwise, we would

not observe performance gains when compared with the approaches Tune V1 and Tune V2 which do not perform any profiling.

5.2.3.4. Multi-Tenancy

Next, we evaluate PipeTune in a multi-tenancy scenario (i.e., a shared cluster handling multiple HPT Jobs). In this case, we show the average response time of jobs as an indicator of performance. We consider that jobs arrive randomly with the interarrival times being exponentially distributed. For the case where two workload types are considered together, each of them corresponds to 50% of the overall jobs (i.e., equally balanced). In all cases, within a given workload type, the workloads are chosen following a round-robin strategy. The portion of overall unseen jobs corresponds to 20%.

5.13 shows the results for the multi-tenancy scenario considering workloads of Type-I and Type-II grouped by type as well as the overall results. As in Section 5.2.3.3, this evaluation has been performed in a distributed environment. In this experiment we observe improvements similar to the ones in the single-tenancy scenario. Regarding response time, PipeTune results in up to 30% reduction when compared with Tune V1 and Tune V2.

5.14 shows the same results described above but considering workloads of Type-III. This trace was executed in a single node in contrast with the distributed environment of the previously described results. In this specific scenario we observe that the performance gain trends earlier observed becomes even more evident in such environment and workload type. In this case, PipeTune results in up to 65% reduction on the average response time in comparison with Tune V1 and Tune V2. This indicates that the overhead of computation added for the unseen jobs is compensated by the gain of future similar incoming ones.

6. Security and Fault-Tolerance

6.1. Trusted Consensus in Untrusted Environments

The shift of computing and storage from one's own datacenters to the cloud has created concerns in regards to the confidentiality, integrity and consistency of the processed and stored data. Consistency implies that storage replicas can neither diverge nor can they be rolled back to an older version. At the same time, clouds offer data durability and service availability at new levels by implementing convenient and affordable replication mechanisms across multiple data centers. Ensuring confidentiality, integrity, consistency as well as durability and availability poses new challenges.

Our focus is on ensuring the consistency of replicas even if an adversary would have root access on all machines. To address this problem, we introduce *trusted consensus*. To implement trusted consensus, trusted execution environments are insufficient since the do not prevent, e.g., Sybil attacks. We show how to ensure trusted consensus with the help of monotonic counters and local election.

As contributions, we introduce efficient monotonic counters based on Replay Protected Memory Block (RPMB) to ensure sufficient performance, make a succinct performance comparison of fresh and worn out Trusted Platform Module (TPM) against RPMB monotonic counter update and read rate, showing a very significant latency improvement. We show how to implement a local leader election on top of monotonic counters to protect against Sybil attacks. We ensure the consistency of the consensus log with the help of the monotonic counters. Further, we quantify the overheads introduced by our solution. We use a native replication setup to serve as baseline.

6.1.1. Architecture

6.1.2. Overview

We use RAFT as our consensus protocol [29]. To provide trusted replicated storage, we employ an architecture comprised of Docker containers powered by SCONE images and therefore with binaries that run inside SGX enclaves. Each RAFT instance is coupled with an individual embedded multimedia card device (eMMC). We implement policies to assure that a single eMMC client will be able to communicate with the eMMC device. Additional instances will no longer continue once they detect another working instance coupled with the device. Chosen instances then participate in a consensus algorithm cluster, namely a RAFT cluster. As participants of that cluster, the container instances copy and execute commands given by RAFT's cluster leader. Because of that, they work as replicas of that global leader.

The threat model under which RAFT has been developed does not consider the possibility that non-volatile memory records are not trusted. Because of that we change RAFT's protocol to also rely on RPMB's monotonic counters and avoid being rolled back. Moreover, with the help of a SCONE policy, we ensure that the components transparently attest each other: only entities belonging to the same application, executing the code specified in the policy inside of a TEE, can communicate with each other.

To demonstrate our solution, we implemented a typical *database as a service* application. For that, we run a DQLite instance, a database management system based on popular SQLite but that packs a RAFT implementation (C-RAFT) to create an automatically replicated database service.

6.1.2.1. Trusted Storage and Clock

Storage: With a once-in-a-lifespan programmed key, a client application requests an update to the starting monotonic counter value provided by the RPMB, hereon referred to as κ, whenever a new write is done. Naturally, the client application, when in unbuffered mode, will not commit the transaction before getting an acknowledgment message from the RPMB server. The application keeps a copy of κ, $c(\kappa)$. That $c(\kappa)$ is later used to check for data integrity.

Notice that the last step is necessary because the eMMC server is a binary that is installed in an untrusted environment and could be tampered with. One attack we predict is to simply drop

write requests. In the eMMC protocol, acknowledgement messages are not authenticated, meaning that they could be crafted and spoofed by the adversary. Reads, in the other hand, are HMAC and nonce protected. HMACs are hash-based message authentication code, so it cannot be falsified, and nonces are single-use numbers used in the request, so it cannot be replayed.

Clock: In the local election, timeout plays a very important role. The consequence of that in the context of trusted computing is that the safety property we propose, that is, there is only a single instance that could operate a single eMMC device, can be invalidated by a powerful user.

To tackle that problem, we utilize an enclave-interval timer that securely measures non-interrupted time intervals inside a TEE enclave, such as in SGX, based on T-lease, a trusted lease primitive [33]. This provides a bounded, short-term trusted clock. Whenever the enclave code loses context, the time spent outside of the enclave will be nullified and the source of time will be verified when entering the enclave again.

The enclave-interval timer uses the TSC (x86 Timestamp Counter) register to count cycles while inside the enclave. Verifying the cycle rate is possible by observing the *RDRAND* instruction rate, which is independent of CPU frequency.

Our approach inherited a few factors from T-lease that affect accuracy. Although admittedly, this approach has a certain variance compared to untrusted clocks, it keeps its safety quality - a clock may have an abnormal rate within bounds that never damage our program.

6.1.2.2. Isolation

As mentioned in subsection 6.1.2, only one client could operate on an eMMC at a time to guarantee that messages cannot be dropped by a maliciously modified eMMC server. We also state that this is necessary to prevent an adversary from achieving a majority by maliciously instantiating multiple enclaves. However, in the context of an untrusted cloud environment, the control over enclave instantiation is beyond our power.

That attack consists of manipulating a newly instantiated container configured to access the same eMMC device as the targeted container. When the targeted container writes to the RPMB partition in the eMMC device, calls to the subsequent read, that confirms the write, get dropped by the attacker. Then, the new container is induced to make a write to the same address (the adversary could spam to every block to make sure the write is successful). Then the read is unsuccessful.

In this context we had to enforce the single enclave instance per eMMC device policy. The algorithm starts by requesting a read of the RPMB monotonic counter κ. The container waits for 1 *term*, an adjustable length of time tuned for performance. Then, it reads κ again. If κ has been incremented, then another enclave is running the algorithm and is currently at a later step. Because this algorithm tries to emulate a *First Come, First Served* policy, this makes this enclave give up and terminate itself. Otherwise, if the value of κ is still the same, it writes its own randomly generated, collision resistant identifier in the *Election Block* of the RPMB storage partition. This is a block we reserve for this election algorithm. The enclave then waits for another term, and reads the contents of the *Election Block*. If it still contains the same id, it declares itself the local leader and proceeds with operations. If another id is found, it gives up and terminates, again. At least every one term, the leader checks whether its id is still written in the *Election Block*.

To assess the correctness of this algorithm, let's assume there are two leaders α_1 and α_2. In that case they must both have read the *Election Block* and read their own ids, since the ids are assumed to be collision resistant, and if some $id_1 \neq id_2 \implies id_1 > id_2 \vee id_2 > id_1$. They are also both expected to have written their ids to the *Election Block*, when they arrived at state 3. They then both wait for 1 term (δ_t), with trusted clocks. If α_2 wrote after α_1, then α_1 should read first and, given an unfavorable id comparison, terminate. In order for α_2 to read its own id, it must read after α_1. Given both wait for δ_t, it would imply that $\delta_t > \delta_t$, which is absurd.

The main design goals of this election algorithm are to preserve safety and incur in minimal overhead. To that end, we have also implemented a measure for less adversarial scenarios. We call it *election by connection*, and it consists of enclaves simply attempting to get hold of the single connection socket available in the eMMC Server. The untrusted server should reject further attempts to connect and their originating enclaves will terminate.

6.1.2.3. Trusted Replication

We use trusted consensus to ensure the consistency of data. Each participant of the consensus protocol has access to a monotonic counter. Consensus - using either a crash or Byzantine failure model - can be attacked in the same way as the eMMC devices: an adversary can spawn multiple instances of each participant. In this way, an adversary can force a divergence of consensus protocol by virtually partitioning a system as each partition would look like a majority partition. Thus, data replicated with the help of consensus can become inconsistent.

We address the Sybil attack cited above by ensuring that each participant must be a local leader - as explained in section 6.1.2.2. We use RAFT as our consensus protocol which is designed for crash failures. RAFT itself also performs elections to elect one of its participants (aka instances) as the *global leader*, and we call the RAFT elections also *global elections*.

The RAFT protocol tolerates messages being deliberately and indefinitely dropped by an adversary that controls the world external to the participants themselves. However, RAFT being based on a crash failure model, trusts its log contents, i.e., the storage is considered trusted. Thus, we protect the log against rollbacks.

We use the RPMB partition as an anti-rollback mechanism. Writing to RPMB storage triggers an increase in its monotonic counter κ. With κ increased, the RAFT participant container can write its new log entry m, along with a copy of κ, say, $c(\kappa_n)$, being n the current size of the log, and some signature of these contents using a key only available to the enclave, e.g. a keyed Hash Message Authentication Code (HMAC), hereby denoted $\sigma(m, c(\kappa_n))$. This 3-tuple is, at principle, enough to prevent rollback attacks. The adversary cannot prevent the enclave from verifying whether the $c(\kappa_m)$ and m in the last entry match both $\sigma(m, c(\kappa_n))$ and κ.

That straight forwards solution would suffice, if it were not for RAFT's policy of removing uncommitted minority entries. At first glance, changing RAFT's policy to *write-then-delete* in this case would always prevent $\kappa > c(\kappa_n)$ from happening. However, there are two problems with this approach. First, there will be a small window of time of an incorrect log index in both RAFT message exchange and receiver's state, between writing and deleting. Second, the intuitive rule that no gaps should appear in the logs is no longer enforced.

We avoided changes in RAFT's behaviour by writing to RPMB instead of to the log itself. Specifically, being q its commited index, the participant having its entries evicted would write its last committed entry $c(\kappa_q)$ and its *second next monotonic counter copy* i.e., $c(\kappa_{n+2})$ to a dedicated RPMB block, in what we call a *pre-deletion saved state*. The value in $c(\kappa_q)$ prevents any further entries from being evicted, than the ones that are not present on the new leader, since it is the same value as in the latest committed entry. The *next monotonic counter copy* keeps consistency with κ, preventing new entries from being rolled back. Notice that 2 is added to $c(\kappa_m)$ because the write of the *pre-deletion saved state*, itself, increments κ by 1. The writing of the next new entry will further increment it by 1.

This presents the opportunity to an attacker to remove older entries on the log, since there is no mechanism to track continuity of the $c(\kappa_1), c(\kappa_2), c(\kappa_3), ..., c(\kappa_n)$ sequence on the log. Such discontinuity could be caused by regular RAFT operations or by attackers, indistinguishably. Existing recorded entries with the right $c(\kappa)$ can also be introduced to the log.

To remedy that problem, we have added a new field to the log entries. It stores the *previous* $c(\kappa)$. In a typical case, this field will simply hold the previous entry's $c(\kappa_{n-1})$. When a gap forms due to entries being evicted, the new field receives the *pre-deletion saved state* values for *previous* $c(\kappa)$ and *current* $c(\kappa)$. The *previous* $c(\kappa)$ must match the last entry's $c(\kappa_m)$, forming a "bridge" in the gap. With this, gaps still exist on the *current* $c(\kappa)$ sequence, but when they happen, the *previous* $c(\kappa)$ value in the newer entry must match the *current* $c(\kappa)$ in the older. In summation, for some entry j, being $c'(x)$ *previous* $c(\kappa)$, such that $c(\kappa_j) + 1 \neq c(\kappa_{j+1})$, it must be that $c'(\kappa_{j+1}) = c(\kappa_j)$. Any entries not attending to these requirements constitute evidence that entries in the log have been removed.

6.1.3. Results

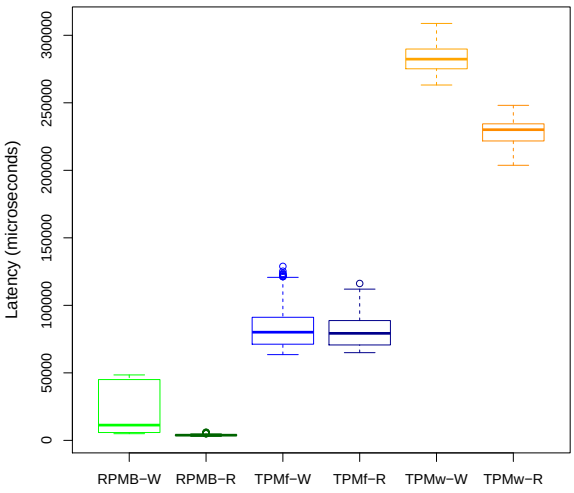


Figure 6.1. RPMB x fresh and worn out TPM NVRAM monotonic counter - Latency distribution for reads and writes

6.1.3.1. RPMB as a Source of Trust

In this section we demonstrate that the RPMB partition, present on eMMC storage units, yields better practical results in relation to standard TPM NVRAM in regards to the utilization of monotonic counters.

6.1.3.2. Writing Latency Evaluation

Figure 6.1 visually demonstrates a distinct difference in performance between the three experimental units. We have used an early to mid-life TPM model with some 400.000 updates to represent the worn-out performance.

Notice that the mean latency for RPMB writes is down to 4x lower than that of a fresh TPM, despite the sparser distribution. Compared to an unit at approximately half life that ratio goes to over nearly 15x.

In addition, TPM's monotonic counters become unusable at about 10^6 updates, while RPMB memory becomes read-only when its monotonic counter reaches $2^{32} - 1$. At a rate of approximately 1 update per 20 milliseconds, one RPMB module would yield about 2.72 years of lifespan, greatly surpassing TPMs in speed and specially longevity.

6.1.3.3. Reading Latency Evaluation

Figure 6.1 shows that the latency performance difference is even more accentuated on reads of the monotonic counter than on writes. Much of the difference can be explained on the basis that updating its monotonic counter implicates on writing to a block on RPMB, while reading that monotonic counter consists in retrieving a Flash memory stored 32 bit integer. On Table 6.1, it can be seen that reads of the monotonic counter are over 20x faster on RPMB. Taken the worst case to account, the ratio reaches 22:1 in favor of RPMB.

6.1.3.4. Trusted Replication

The main goal in our Trusted Replication evaluation is to quantify the costs of adoption of our architecture. To that end, we execute custom benchmarks in order to discuss which factors impact its performance. We evaluate how latency scales as the number of replicas increases, so we can derive a ratio that relates the costs asymptotically.

Our experimental units are a native, a SCONE-only, a SCONE-only without parameter adjustments, and the Trusted Replication variants. The Trusted Replication variant is the complete

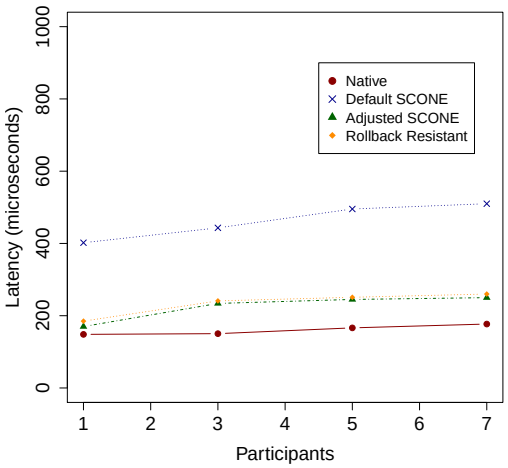


Figure 6.2. Trusted Replication Latency - retrieve performance for large queries

solution, including RPMB enforced rollback protection and adjusted SCONE parameters.

6.1.3.5. Retrieving

We have chosen a memory access and buffering intensive benchmark in the form of a series of large retrieve queries. The objective was to pressure memory access inside enclaves and verify how memory reads would affect latency.

Figure 6.2 shows the average latency for *Select* statement queries that match about 14000 rows. Loading the query results to the client application was included in the benchmark. It demonstrates that as the number of participants increases, little difference can be seen on latency. That is to be expected, since *retrieve* operations do not trigger RAFT operations, and are rather executed on the leader node. Conversely, it is somewhat surprising to notice the upwards slope in response time between 1 and 3 participants. We believe that is caused by the activation of C-RAFT's routine operations when multiple nodes are detected in the cluster.

Since rollback-resistant mechanisms are not triggered on *Select* statements, which make up this benchmark, difference between the Rollback Resistant variant and its Adjusted SCONE counter part is marginal. The results in comparison with the native variant have stayed within 20%, which is considered to be satisfactory for our objectives.

6.1.3.6. Updating

Updating should pressure memory, network I/O and storage I/O. Our benchmark consists of a 10000 row update operation. Figure 6.3 shows each variant's mean latency.

The Rollback Resistant variant stayed within 10% of the Adjusted SCONE variant, with not much more than the expected 20μs RPMB update penalty in difference, showing that for this kind of long update query our solution presents acceptable overheads.

In relation to Native, the Adjusted SCONE variants had nearly 2x latency for this benchmark. We attribute that difference to the increase in memory writes, known to cause high overhead in SCONE.

Above 3 participants, the Default SCONE parameters variant would run out of memory, or exhibit unexpected behaviour and fail to finish the operation. We believe that the deficiency in balance between I/O threads and enclave threads may be causing buffers to grow for too long, exhausting enclave resources.

6.1.3.7. Insertion

Our insertion benchmark differs from the previous ones, in that it is composed by some 20000 `insert` queries, instead of one large query. It intends to overwhelm I/O operations, so the deficiencies of SCONE and RPMB can be better isolated. Figure 6.4 show the mean latency for executions of this 20000 queries benchmark.

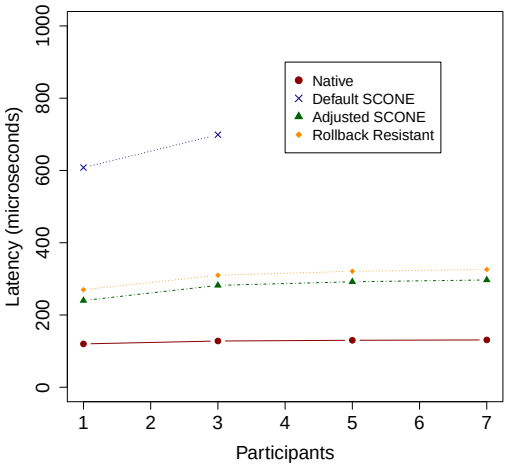


Figure 6.3. Trusted Replication Latency - update performance for large queries

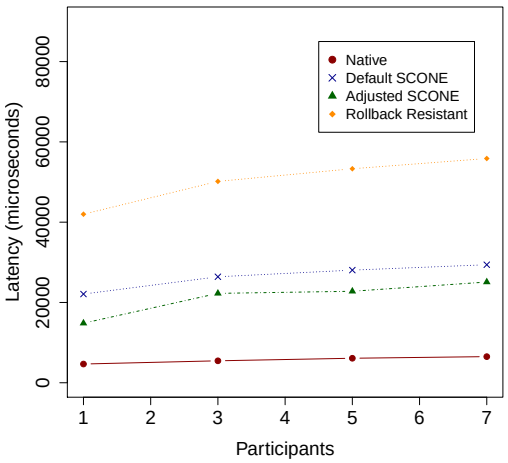


Figure 6.4. Trusted Replication Latency - Insertion performance

The penalty RPMB updates cause visibly accumulates in this benchmark. Resistant Rollback performance ranges at about twice that of the Adjusted SCONE variant and stays below unadjusted SCONE. The I/O intensive nature of this benchmark seems to better match SCONE's default configuration than the previous ones.

In general, the difference between native and the Adjusted SCONE is about 5x, showing that TEE's need further, more tailored parameterization to be effective in high I/O scenarios.

6.2. FPGA Fault Tolerance

Field-Programmable Gate Arrays (FPGAs) are computational devices known for their energy efficiency in comparison to instruction set architecture (ISA)-based devices like CPUs and GPUs. Multiple works in the literature have shown the energy efficiency of FPGAs. For instance, a recent studies have demonstrated that the FPGA implementation of a Gaxpy kernel is an order of magnitude lower in terms of power consumption, while also being very competitive in terms of performance. However, FPGAs are also more challenging for application development. The workload to be executed has to be converted into a set of bits containing the information about the configuration of its logic cells, Block RAM, etc. Originally, this had to be done by developing the application in hardware description languages like VHDL.

Fortunately, there are now tools like Vivado High Level Synthesis (HLS) [36] which allow developing for such devices in high level languages. Also, frameworks such as OmpSs@FPGA [9] make it even more straightforward and do most of the extra work needed to develop and build

FPGA applications. Because of this, heterogeneous systems with FPGAs are becoming increasingly popular and are being considered as accelerators for supercomputers in high-performance computing (HPC). One of the best examples of this is the porting of HPC applications such as our Smart City use case (Alya) into FPGAs (See Deliverable 5.4).

However, HPC applications run for long periods of time and are subject to failures and errors. Thus, it is important to protect those applications (e.g., Alya) with fault tolerance techniques to be able to tolerate those errors and avoid wasted time. In our previous report (See deliverable 4.3 Section 6.1) we reported the porting of Alya with FTI to profide fast and efficient multilevel checkpointing. The porting was implemented, integrated and tested in the Marenostrum Supercomputer at the BSC. D4.3

In addition to that, we have implemented FPGA checkpointing with the same interface as we previously implemented GPU checkpointing, We use the multilevel checkpointing library FTI and its API to be able to transparently checkpoint FPGA applications. This work is reported in our Deliverable 3.4. link between WP4 and WP3

In this way, we connect the achievements previously reported in this WP4, with the efforts done in WP5 and WP3 to connect all the dots, and achieve a universal fault tolerance technique (FTI) capable of supporting multiple heterogeneous devices (i.e., FPGAs and GPUs) and demonstrate its effectiveness with a real use case (Alya) that has been ported to new acccelerators.

7. Conclusion

The compiler tools developed in LEGaTO were integrated with the front-end components developed in Work Package 4. We studied how to exploit the task-based programming model of OmpSs to assist the dataflow development process when using Maxeler's MaxCompiler. We presented a number of enhancements done on XiTAO's data parallel interface. We also analysed the performance impact of running a few typical LEGaTO functions with OmpSs inside SGX enclaves. We also provided more details on the plug-in we developed for integrating OmpSs development interface into Eclipse Che.

In the Dataflow Engines field, we concluded the work on DFiant and we reported an evaluation of its programmability. As stated above, the work on integrating Maxeler's interface with OpsSs was reported in the compiler section.

Concerning energy efficiency, we proposed two new studies we deemed important in the context of LEGaTO and its aplications. The first study evaluates the energy efficiency and the performance impact of IoT applications when ruuning inside Arm's TrustedZone. The second study proposes a method for taking energy consumption into account when adjusting machine learning hyperparamenters. The first study allows for understanding the energy cost of running trusted applications, while the second study allows for finding tradeoffs of energy and performance.

We completed this deliverable with a chapter on security and fault-tolerance aspects. For security, we present our novel research results in obtaining consensus with the help of trusted execution environments. We also describe how the fault-tolerant interface deceloped in the project can be used with FPGAs, with a reference to the more details given in Deliverable D3.4.

All in all, this document describes the work undertaken in Work Package 4 towards the end of the whole project. Our main interest was to complete, evaluate and report the final efforts on integrating all components. We were nevertheless able to produce new research outputs during the period, on both energy efficiency and security aspects.

8. References

[1] VHDL Code for Bitonic Sorter. https://vlsicoding.blogspot.com/2016/01/vhdl-code-for-bitonic-sorter.html, January 2016.

[2] Full VHDL code for Moore FSM Sequence Detector. https://www.fpga4student.com/2017/09/vhdl-code-for-moore-fsm-sequence-detector.html, September 2017.

[3] 20 Newsgroups. http://qwone.com/~jason/20Newsgroups, 2020. Accessed: 2020-14-09.

[4] Conan, the C/C++ Package Manager. https://conan.io, 2020. Accessed on 29.10.2020.

[5] Eclipse Che. https://www.eclipse.org/che/, 2020.

[6] Hpc challenge benchmark. http://icl.cs.utk.edu/hpcc/, 2020 (accessed October 29, 2020).

[7] Ahmad Aghaebrahimian and Mark Cieliebak. Hyperparameter tuning for deep learning in natural language processing. In Mark Cieliebak, Don Tuggener, and Fernando Benites, editors, *Proceedings of the 4th edition of the Swiss Text Analytics Conference, SwissText 2019, Winterthur, Switzerland, June 18-19, 2019*, volume 2458 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[8] Arm Limited. ARM Security Technology: Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, April 2009. Accessed on 30.07.2019.

[9] Barcelona Supercomputing Center. OmpSs@FPGA. https://pm.bsc.es/ompss-at-fpga, December 2020.

[10] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti. Trusted computing meets blockchain: Rollback attacks and a solution for hyperledger fabric. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 324–333, 2019.

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*, pages 44–54. IEEE Computer Society, 2009.

[12] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016(86):1–118, February 2017.

[13] Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, Jiao Wang, Shengsheng Huang, Zhongyuan Wu, Yang Wang, Yuhao Yang, Bowen She, Dongjie Shi, Qi Lu, Kai Huang, and Guoqiong Song. Bigdl: A distributed deep learning framework for big data. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, pages 50–60. ACM, 2019.

[14] Geir Drange. Ultimate CRC. https://opencores.org/projects/ultimate_crc, January 2016.

[15] Felix A. Gers, Jürgen Schmidhuber, and Fred A. Cummins. Learning to forget: Continual prediction with LSTM. *Neural Comput.*, 12(10):2451–2471, 2000.

[16] GlobalPlatform, Inc. *TEE Sockets API Specification Version 1.0.1*, January 2017.

[17] Christian Göttel, Pascal Felber, and Valerio Schiavoni. iperfTZ: Understanding Network Bottlenecks for TrustZone-Based Trusted Applications. In Mohsen Ghaffari, Mikhail Nesterenko, Sébastien Tixeuil, Sara Tucci, and Yukiko Yamauchi, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 178–193, Cham, 2019. Springer International Publishing.

[18] Christian Göttel, Pascal Felber, and Valerio Schiavoni. Developing Secure Services for IoT with OP-TEE: A First Look at Performance and Usability. In José Pereira and Laura Ricci, editors, *Distributed Applications and Interoperable Systems*, pages 170–178, 2019.

[19] Homer Hsing. AES Core Specification. `http://opencores.org/usercontent,doc,1354351714`, 2013.

[20] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[21] Yann LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 20:5, 2015.

[22] Yuxi Li. Deep reinforcement learning: An overview. *CoRR*, abs/1701.07274, 2017.

[23] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *CoRR*, abs/1807.05118, 2018.

[24] David Lundgren. Double Precision Floating Point Core VHDL, 2014.

[25] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In Takao Kobayashi, Keikichi Hirose, and Satoshi Nakamura, editors, *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pages 1045–1048. ISCA, 2010.

[26] Dmitry Molchanov, Arsenii Ashukha, and Dmitry P. Vetrov. Variational dropout sparsifies deep neural networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2498–2507. PMLR, 2017.

[27] Christina Müller, Marcus Brandenburger, Christian Cachin, Pascal Felber, Christian Göttel, and Valerio Schiavoni. TZ4Fabric: Executing Smart Contracts with ARM TrustZone. In *2020 IEEE 39th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2020.

[28] Kevin E Murray, Mohamed A Elgammal, Vaughn Betz, Tim Ansell, Keith Rothman, and Alessandro Comodi. Symbiflow and vpr: An open-source design flow for commercial and novel fpgas. *IEEE Micro*, 40(4):49–57, 2020.

[29] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[30] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011.

[31] Volnei A Pedroni. *Generic Priority Encoder from Digital electronics and design with VHDL*. Morgan Kaufmann, 2008.

[32] Marcelo Samsoniuk. DarkRISCV: Opensource RISC-V implemented from scratch in one night! `https://github.com/darklife/darkriscv`, 2019.

[33] Bohdan Trach, Rasha Faqeh, Oleksii Oleksenko, Wojciech Ozga, Pramod Bhatotia, and Christof Fetzer. T-lease: a trusted lease primitive for distributed systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 387–400, 2020.

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

[35] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017.

[36] Xilinx, Inc. *Vivado Design Suite User Guide: High-Level Synthesis*.

[37] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE transactions on pattern analysis and machine intelligence*, 38(10):1943–1955, 2015.