LEGaTO

Low Energy Toolset for
Heterogeneous Computing

D4.3 "FINAL RELEASE OF ENERGY-EFFICIENT, SECURE, RESILIENT TASK-BASED PROGRAMMING MODEL AND COMPILER EXTENSIONS, INCLUDING FPGA TOOLCHAIN"

Version 1

Document Information

Contract Number	780681
Project Website	https://legato-project.eu/
Contractual Deadline	31 May 2020
Dissemination Level	Public
Nature	Report
Author	Osman Unsal (BSC)
Contributors	Marcelo Pasin (UNINE), Christian Göttel (UNINE), Isabelly Rocha (UNINE), Do Le Quoc (TUD), Oleksii Oleksenko (TUD), Xavier Martorell (BSC), Leonardo Bautista-Gomez (BSC), Mustafa Abduljabbar (CHALMERS), Oron Port (TECHNION), Tobias Becker (MAX), Nils Voss (MAX)
Reviewers	Pascal Felber (UNINE), Tobias Becker (MAX)

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
1226	2020-03-05, File created
1270	2020-04-23, Added section structure for partner input
1302	2020-05-06, Added SCONE Engergy section
1303	2020-05-07, Added SCONE Fault Tolerance section
1310	2020-05-09, Updated SCONE energy section
1321	2020-05-13, Added DFiant section
1326	2020-05-13, Adding Alya with FTI
1327	2020-05-14, Drafted XiTAO section
1329	2020-05-15, Added DFiant FSM example
1332	2020-05-16, Updated DFiant section
1334	2020-05-18, Added OmpSs Maxeler interoperability
1335	2020-05-18, Updated XiTAO section
1338	2020-05-19, Added Hermes
1341	2020-05-20, Added OmpSs-SGX
1353	2020-05-28, Completed OmpSs-SGX, fixed formatting
1361	2020-05-28, Added MAXJ specific sections
1364	2020-05-28, Updated MAXJ specific sections
1365	2020-05-29, Added compilation figures and full stack
1401	2020-06-04, 1st review
1404	2020-06-08, Revised Chapter 5
1406	2020-06-08, Added better reference to XiTAO
1408	2020-06-08, 2nd review
1430	2020-06-09, Revised introduction, chapters 3, 4, 6
1434	2020-06-09, Revised XiTAO energy
1435	2020-06-09, Updated introduction

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

Index

List of Figures

List of Tables

1. Executive Summary

This report describes the final release of the LEGaTO toolchain frontend. It is issued at the same time as Deliverable D3.3 on the backend, and includes many components that complement the ones presented there.

This deliverable D4.3 extends and supersedes deliverable D4.2 "First release of the task-based programming model and compiler extensions". To avoid overlap between the two documents, we choose to highlight only those components that are novel or have been considerably updated since D4.2. Overall, this deliverable covers work that has been done during the past 10 months, from M20 until M30.

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

Chapter 3 provides the pointer to OmpSs compiler related work that is discussed in D3.3. The data parallel interface which forms the core of the XiTAO programming model is introduced next. We then discuss the stable version of the Dfiant language and compiler. The final MaxJ compiler toolflow concludes this section.

The next section Chapter 4 details the first integration of the LEGaTO runtimes. In this first integration, the focus is on individual integration of the LEGaTO dataflow substrates of XiTAO, Maxeler and DFiant with the OmpSs programming model. Each integrated component is described in this section. A new integration page is available at https://legato-project.eu/software/integration and all integrated components are added to the LEGaTO github.

Chapter 5 highlights the energy-efficiency effort, first introducing the XiTAO energy-aware scheduling approach, followed by the work on energy-efficient IoT Data Deduplication. The advance on the HEATS energy scheduler in the period of M20 to M30 is described next. Finally two orthogonal work that focus on security are described, first the energy-efficient aspects of the secure SCONE framework is described. Finally the first integration of OmpSs tasking model with Intel SGX security framework is detailed.

Chapter 6 summarizes the Fault-tolerance work at the programming model and compiler layers. First the integration of the ALYA software framework which powers the smart city use case with the Fault Tolerance Interface (FTI) checkpointing library is described. Then, finally fault tolerance aspects of the secure SCONE framework is described.

2. Introduction

The objective of the LEGaTO project is to develop a software toolchain for heterogeneous hardware with the energy efficiency as the main focus. The project also considers security, fault-tolerance, and programmability together with energy-efficiency. To set the stage, we include a brief reminder of the LEGaTO stack in Figure 2.1. In the figure, we present a birds-eye view of the various components used in the project, arranged hierarchically in the compute stack starting from use cases, to programming model, compiler and high level synthesis (HLS) languages, runtime, middleware and hardware. In addition to the description of the particular stack level (hardware, middleware, runtime, applications), Work Package identifiers were also included.

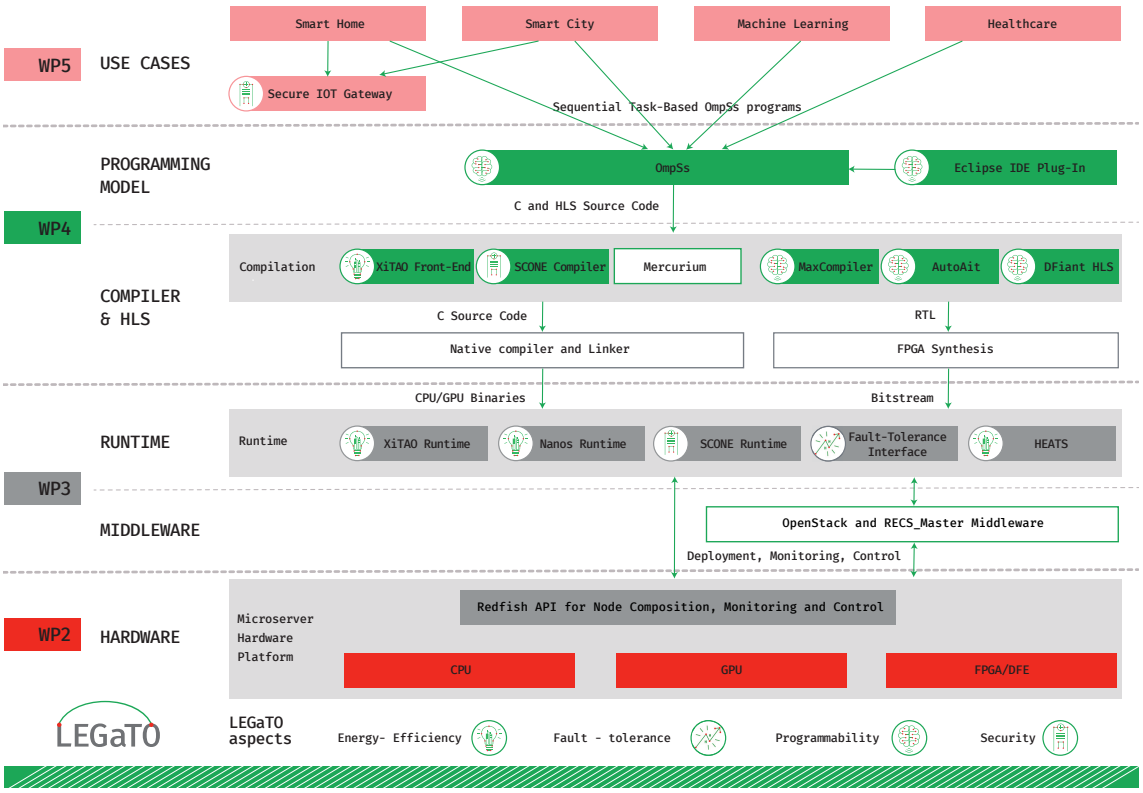


Figure 2.1. The LEGaTO stack and relation of components with LEGaTO goals

Five use cases are developed in the context of LEGaTO. The first use case is a smart home application, processing privacy-sensitive information in order to provide assisted living recommendations. The second use case is a smart city application to model air quality in near-real time using fluid dynamics. The third use case is an energy-efficient machine learning application with surrounding perception and trajectory calculation for self-driving cars. The fourth use case is a computationally intensive Monte Carlo method for obtaining reliable classifiers, to identify specific biomarkers that allow for diagnosing each disease more precisely. The fifth use case is related to the smart home/city use cases and is focused on IoT security and usability, developing a gateway for (secure) network connections of local and remote network devices.

In order to support applications (and the use cases), LEGaTO proposes a programming model and corresponding compilers, including high-level synthesis for FPGAs. Below the use cases in Figure 2.1, one finds the task-based OmpSs programming model that implements a write-once run-anywhere approach with mapping into CPUs, GPUs and FPGAs. An eclipse IDE plug-in facilitates programming leveraging OmpSs. OmpSs leverages multiple compilers to map tasks to CPUs (Mercurium, SCONE and XiTAO front-end), Dataflow Engines (MaxCompiler), and FPGAs (autoVivado), as well as the LEGaTO-developed HLS, called Dfiant, to map high-level kernels into FPGAs. XiTAO is an experimental task-based programming abstraction that naturally accompanies OmpSs and provides elasticity. SCONE tool ensures secure execution of applications.

The LEGaTO runtimes of XiTAO and Nanos work together to seamlessly run OmpSs tasks in a dataflow fashion on the various hardware platforms in an energy-efficient way. In particular, the XiTAO runtime is used as a research development vehicle within the Nanos scheduler providing higher level scheduler decisions. LEGaTO tasks are run securely by leveraging Intel SGX security hardware extensions through SCONE's runtime. For fault-tolerance the project has developed checkpointing solutions on LEGaTO GPUs and FPGAs as well as CPUs that are enacted using the Fault-tolerant Interface (FTI) annotations. Finally, the HEATS heterogeneous scheduler ensures energy-efficiency at the runtime level.

The runtime-optimized tasks are then mapped to energy efficient low form factor LEGaTO hardware through the use of the Redfish interface that offers configurability of the hardware resources. Monitoring and control of the hardware is exposed through a REST API to higher layers.

Task Progress Summary

We now provide a brief synopsis of the technical advances by task. WP4 has seven tasks, as presented below, and each task pushes forward a different aspect in the development of a number of components.

- Task 4.1: Definition / Design
- Task 4.2: Programming Model Features for Energy Efficiency
- Task 4.3: IDE Plugin
- Task 4.4: Compiler Support
- Task 4.5: High-Level Synthesis for FPGA
- Task 4.6: Task-based Kernel Identification/DFE Mapping
- Task 4.7: Fault Tolerance and Security

Task 4.1 ended at month 9 and most of its work has been reflected into Chapter 4 of Superdeliverable SD1. The remaining tasks (4.2–4.7) are intended to run throughout almost the entire project, and will develop different components of the front-end tool chain. Task 4.1 Definition/Design presented in SD1 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 (fault-tolerance, heterogeneity, multicomputer execution, energy efficiency and the extension of the programming model). It also provided a number of extensions to be implemented in the infrastructure management to support the execution of the proposed task model.

In Task 4.2 Programming Model Features for Energy Efficiency, we have upgraded OmpSs@FPGA through several releases. The current release is 2.2.0. It includes the new version of autoAIT (formerly autoVivado), that now supports the Xilinx Development kit ZCU102 (various revisions, up to 1.1). We also upgraded the support for up to Vivado HLS 2019.3. autoAIT incorporates better support for internal FPGA instrumentation, improved kernel module, with better management of the information exposed to the runtime about the environment characteristics, and configuration file. Moreover, in Task 4.2, the XiTAO runtime system now offers a DAG-friendly data-parallel interface, that can leverage XiTAO's energy-efficient scheduling features on heterogeneous platforms in a wider spectrum of applications. The parallel loops can now be nested in generic DAG workflows in place. This essentially eases the construction of mixed-mode parallel DAG in XiTAO. Additionally, we observe that energy efficient high-performance compression algorithms are compute intensive and therefore are not applicable to IoT devices. Therefore, we propose data deduplication (avoiding to send same or similar data) to save energy in IoT devices. Finally, we continued our work in the HEATS scheduler which was introduced in D4.2. For D4.3, we evaluated performance and energy efficiency of HEATS.

In Task 4.3 IDE Plugin, we have upgraded the IDE plugin support for new releases of Eclipse CHE, currently up to CHE version 7.

In Task 4.4 Compiler Support we have added a Maxeler DFE architecture plugin to the OmpSs runtime, to support spawning of tasks to Maxeler hardware. After having an initial implementation working with matrix multiplication, we have finalized the integration of the OmpSs and Maxeler programming models, so that the OmpSs runtime is now able to start DFE kernels developed with the Maxeler toolchain. We have checked the implementation on MAX3 (local at BSC) and MAX5 (JuMAX at Julich Supercomputing Center) platforms. Also in Task 4.4 by using resource sharing constructs developed in D4.2, we now showcase interoperability between XiTAO and OmpSs, where the two runtimes collaboratively execute batch-tiled matrix multiplication, and should eventually be compatible with the OmpSs programming model. The integration repository is available at : https://github.com/legato-project/nanos-xitao-integration.

For Task 4.5 High-Level Synthesis for FPGA, we have produced the stable version of the DFiant compiler and programming model. The key improvement is in the compiler: the DFiant Intermediate Representation (IR) was modified to allow functional programming of its compilation stages. Other improvements involved updating the interface to facilitate composable port addition, a new simplified Finite State Machine (FSM) syntax and the addition of constraint tags that allow the injection of RTL like semantics into Dfiant code. We have integrated the execution of DFiant kernels from OmpSs. The programmer can now write DFiant kernels to replace the computation functionality to be executed in the FPGA. These kernels are linked to the wrapper of the function as generated from OmpSs through autoAIT.

Task 4.6 Task-Based Kernel Identification/DFE Mapping: Besides the integration with OmpSs, we have developed a Mercurium compiler pass to obtain the communication weights between DFE kernels, and the computation load of DFE kernels, in order to compute the ratio between computation and IO of the kernels and decide the proper mapping of DFE kernels. Furhtermore we have developed new graph analysis tools.

In Task 4.7 Fault Tolerance and Security, we have done a first integration of OmpSs with Intel SGX security extensions. Moreover, we successfully integrated our toolchain SCONE with Hardware-Assisted fault tolerance (HAFT) to enable fault-tolerance for legacy applications running inside SGX enclaves. We ensure the confidentiality and integrity of the applications by running them inside SGX enclaves with the help of SCONE. We make use of HAFT to provide fault-tolerance for the applications, i.e., protect them against transient hardware faults. HAFT leverages instruction-level redundancy for fault detection and hardware transactional memory for fault recovery. In addition, we developed TEEMon—the first continuous performance monitoring and analysis tool for TEE-based applications. The tool provides not only fine-grained performance metrics during runtime, but also assists the analysis of identifying causes of performance bottlenecks, e.g., excessive system calls. We integrated TEEMon with Kubernetes to monitor the performance of an application running inside more than 6000 distributed SGX enclaves using SCONE. Finally, we have rewritten the ALYA library, which forms the core of the smart city use case to use Fault-Tolerance Interface, our checkpointing library.

Document Structure

In the following, this document contains four technical chapters, laid out as follows. Chapter 3 describes the advance with respect to D4.2 on main components that pertain to the LEGaTO frontend toolchain, namely OmpSs, XiTAO, DFiant and MaxCompiler. Chapter 4 details individually the integration of XiTAO, DFiant and MaxCompiler with OmpSs. This is followed by the project advance in energy-efficiency aspects in Chapter 5. Finally, Chapter6 discusses the annotation of the smart city use case application to support FTI, the LEGaTO checkpointing solution, and concludes by how the project is addressing fault tolerance and security together through the SCONE tool.

3. Compiler Support and Development Environment

This Chapter presents the current snapshot of LEGaTO´s compiler and programming model substrates as well as the advance since D4.2. These main four substrates are OmpSs, XiTAO, DFiant and MaxCompiler. In Section 3.1 we provide a short pointer to the OmpSs compiler work which is being reported in D3.3. Section 3.2 details the data parallel interface for XiTAO while Section 3.3 discusses the last stable version of the Dfiant language and compiler and finally Section 3.4 outlines the MaxCompiler toolflow.

The next Chapter will discuss the first integration of the dataflow substrates which consist of individual integration of OmpSs with XiTAO, Dfiant and MaxCompiler.

3.1. The OmpSs Compiler

Most of the OmpSs compiler work, both for FPGA and for cluster versions have involved heavier involvement of the backend, therefore the OmpSs compiler work has also been described in D3.3.

3.2. The XiTAO Data Parallel Interface

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. The interface is relatively simple to use. Listing 3.1 shows how a loop parallel region, for example, can be part of a full DAG structure using the XiTAO programming interface. Also, Table 3.1 highlights the interface parameters. In this deliverable, the capability of nesting loop parallel nodes in a DAG workflow has been supported. Also, a few explanatory benchmarks adopted from Rodinia Benchmark Suite and Barcelona OpenMP Task Suite are being developed and will soon be part of 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 3.1. The parameters input by user to the XiTAO data parallel interface

3.3. The DFiant Language & Compiler

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

```
TAO_A* head_tao = new TAO_A();
// build DAG
TAO_B* serial_region = new TAO_B();
DataParallelTAO* data_parallel_region
  // parallel for (int iter = 0; iter < end; ++iter) becomes
  = __xitao_vec_region(width, iter , end, sched, block_size,
    // data parallel code region
)
serial_region->make_edge(data_parallel_region);
// continue building DAG
// push the head node
xitao_push(head_tao);
// start the DAG execution
xitao_start();
// wait for the DAG execution
xitao_finish();
```

Figure 3.1. The basic structure of a DAG based program inserting SPMD code regions

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. DFiant is controlled by the software stack through OmpSs. See Section 4.2 for more information on the OmpSs–DFiant integration.

In this deliverable we introduce additional features of the DFiant language and compiler that further ease hardware programmability. Nonetheless, the most significant change occurred in the compiler internals and facilitated all the additions and improvements we bring forth in this deliverable. The DFiant Intermediate Representation (IR) was modified to allow functional programming of its compilation stages. This means that each compilation stage is immutable and we can safely define transformations between stages. The IR is a very simple database composed of a member list and a reference table (e.g., each member contains an owner reference). The transformations between stages are just patch queries that are applied on this database and can be pretty-printed as DFiant code at each stage. Only the first compilation stage is unique since it is mutable and constructed as the design is dynamically elaborated.

3.3.1. Interfaces

Interfaces in DFiant facilitate composable bulk port definition and connection. Figures 3.2 and 3.3 demonstrate such a use case where we need to test the memory interface of our device under-test (DUT). The memory interface signaling typically requires redefinition for different port directionality: the DUT and Driver port directions are reversed, the Monitor ports are input-only, and finally the signals connecting them require declaration too. The DFiant syntax enables declaring the interface once and reusing it in every required direction context. It is possible to define interface hierarchies, compose them freely, or take an existing interface class and expand it via extension. Additionally, bulk connection is applied to connect the whole interface in a single line of code.

```
@df class MemConn extends DFInterface {
  val addr    = DFBits(10)  <> OUT
  val rdData  = DFBits(8)   <> IN
  val wrData  = DFBits(8)   <> OUT
  val wrEn    = DFBit()     <> OUT
}
@df class Dut extends DFDesign {
  val memConn = new MemConn <> ASIS //ASIS annotation is optional
}
@df class Driver extends DFSimDesign {
  val memConn = new MemConn <> FLIP //Inverted directionality
}
@df class Monitor extends DFSimDesign {
  val memConn = new MemConn <> IN   //Input-only directionality
}
@df class Simulation extends DFSimDesign {
  val dut     = new Dut
  val driver  = new Driver
  val monitor = new Monitor
  val memConn = new MemConn <> VAR  //Node connection variable

  memConn     <> dut.memConn
  memConn     <> driver.memConn
  memConn     <> monitor.memConn
}
```

Figure 3.2. Interface definition example (MemConn) and multi-directional instantiation

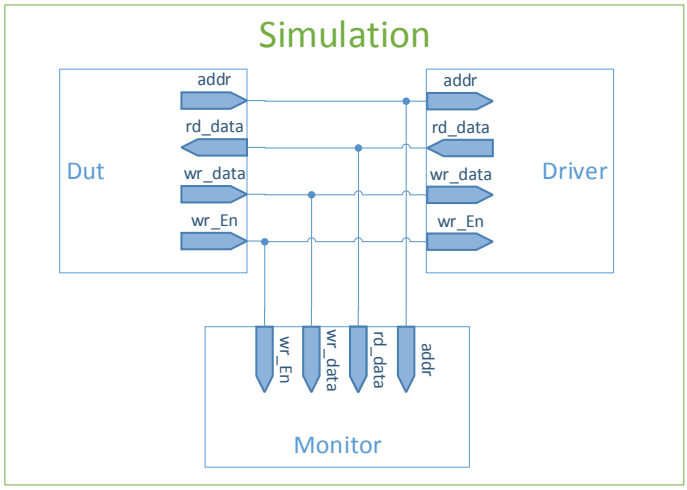


Figure 3.3. Block diagram of the connectivity in Figure 3.2

3.3.2. Simplified State Machines

Finite state machines (FSMs) are commonly used to construct a sequential process in hardware. So far DFiant enabled to define FSMs using a dataflow state variable and a match case statement. The new FSM syntax allows defining and composing state machines in a much simpler fashion. Figure 3.4 shows a simple FSM that generates a valid 0x55 data for a single "step" (a step can be one clock or more, depending on the backend and constraints).

3.3.3. Synchronous RTL Hardware Description

As discussed in the previous deliverable, DFiant is not an RTL but a dataflow HDL, since it carries different HDL semantics. Notwithstanding, we defined a small set of constraint tags that manifest RTL semantics into DFiant code. To describe single clock synchronous designs, all required is to add clock and reset constraints, ports, and additional conditional dependency logic. The clock and reset can be

```
val single55_fsm =
  step {
    valid := 1
    data  := h"55"
  } ==> waitWhile(!ready) =^> { //=^> means state-exit execution
    valid := 0
  } ==> waitForever()
```

Figure 3.4. FSM example that generates a valid 0x55 data for a single step

treated just like any dataflow values because all other data transactions are synchronized to them. Consequently, we added a new compiler stage that compiles regular (asynchronous dataflow) DFiant code into a synchronous equivalent DFiant code with the additional logic and constraints. This extra stage simplifies the RTL backend stage significantly and enables inspecting the expected RTL code still at the DFiant level (DFiant users that are inexperienced with RTL languages can still understand its equivalent DFiant code since it still uses dataflow semantics). Figures 3.5, 3.6, and 3.7 demonstrate asynchronous DFiant code for a cumulative sum translated into an RTL DFiant code and VHDL code, respectively.

```
import compiler.sync._
@df class csum extends DFDesign {
  val i  = DFUInt(8)  <> IN
  val o  = DFUInt(32) <> OUT init 0
  o      := o + i
  this !! ClockParams("clk", ClockParams.Edge.Rising)
  this !! ResetParams("rst", ResetParams.Mode.Async, ResetParams.Active.Low)
}
```

Figure 3.5. Cumulative sum DFiant code with synchronous constraint tags

```
trait csum extends DFDesign {
  final val clk      = DFBit()    <> IN  !! Sync.Tag.Clk
  final val rst      = DFBit()    <> IN  !! Sync.Tag.Rst
  final val i        = DFUInt(8)  <> IN
  final val o        = DFUInt(32) <> OUT init 0
  final val o_var    = DFUInt(32)
  final val o_sig    = DFUInt(32) init 0
  final val o_prev1  = DFUInt(32) init 0 !! Sync.Tag.Reg
  o_var              := o_prev1
  o_var              := o_var + i
  o_sig              := o_var
  o                  := o_sig
  ifdf(rst === 0) {
    o_prev1          := 0
  }
  .elseifdf(clk.rising()) {
    o_prev1          := o_sig
  }
}
```

Figure 3.6. Cumulative sum compiled DFiant RTL-equivalent code

3.3.4. Multistage Meta-Programming

For advanced users, DFiant has a multistage compiler that can be easily expanded. Internally, each stage is usually composed of patch queries to the IR database. To easily add members to the IR, DFiant allows a (meta) design to be planted at various position options within the database. This ability gives power users full control over design automation, analysis, and optimization.

```
entity csum is
port (
  clk              : in  std_logic;
  rst              : in  std_logic;
  i                : in  unsigned(7 downto 0);
  o                : out unsigned(31 downto 0) := 32d"0"
);
end csum;

architecture csum_arch of csum is
  signal o_sig     : unsigned(31 downto 0) := 32d"0";
  signal o_prev1   : unsigned(31 downto 0) := 32d"0";
begin
  async_proc : process (all)
    variable o_var : unsigned(31 downto 0);
  begin
    o_var          := o_prev1;
    o_var          := o_var + i;
    o_sig          <= o_var;
    o              <= o_sig;
  end process;
  sync_proc : process (rst, clk)
  begin
    if rst = '0' then
      o_prev1      <= 32d"0";
    elsif rising_edge(clk) then
      o_prev1      <= o_sig;
    end if;
  end process;
end csum_arch;
```

Figure 3.7. Cumulative sum compiled VHDL (2008) code

3.4. The MaxCompiler Toolflow

MaxCompiler is a tool, which uses a Java-based meta language, called MaxJ, to facilitate FPGA development. The tool focuses on the dataflow abstraction to provide a high productivity programming environment. Using this abstraction, users with a scientific or engineering background can harness the performance of FPGAs for their use cases, without the need to become hardware experts. More details on MaxCompiler were already described in Deliverable D2.1.

MaxCompiler is a commercial tool which has developed by Maxeler for more than 15 years. As such, it already has a rich feature set, including a runtime library, a functional simulator and drivers in addition to a rich library of hardware functions.

MaxCompiler typically supports targeting Maxeler's in-house developed FPGA-based Dataflow Engines (DFEs). Support for Amazon EC2 F1 instances we also previously available. During the LEGaTO project, Maxeler extended to compiler support to Xilinx Alveo data center accelerator cards. Xilinx Alveo U200 and U250 are now system-level compatible with Maxeler MAX5 DFEs and Amazon EC2 F1, therefore increasing the number of commercial platforms that can be targeted with the LEGaTO toolstack.

Within the LEGaTO project, we further improved the toolchain to enable integration with OmpSs and DFiant. Further details are described in section 4.3. Most notably, we improved the support and documentation of functionality to embed VHDL code into a MaxCompiler design. This enabled the integration of DFiant into MaxCompiler, which could then also use the Maxeler OmpSs integration. Additionally, we developed examples which were used by BSC to work on the integration and provided technical assistance.

4. Integration of the Dataflow Programming Substrates

This Chapter introduces the individual integration of the LEGaTO dataflow substrates of XiTAO, Maxeler and DFiant with the OmpSs programming model. The integration of OmpSs tasking model with Intel SGX security framework is discussed in the next Chapter in Section 5.5.

In this Chapter, Section 4.1 details the integration of XiTAO and Nanos, two task-based runtime engines with different features and capabilities. This integration will ensure interoperability, constructive and collaborative sharing of resources between the two runtimes. Section 4.2 outlines OmpSs and Dfiant integration, where dedicated DFiant plugins are developed to morph DFiant kernels into existing interfaces supported by OmpSs. Finally, Section 4.3 explains the integration of OmpSs with MaxJ Compiler where the OmpSs task-based programming model is used as a frontend for Maxeler's dataflow computing model where performance of critical computations are offloaded onto dedicated dataflow engines.

4.1. OmpSs–XiTAO Integration

Chalmers and BSC have integrated the XiTAO and Nanos runtimes. This integration ensures interoperability, constructive and collaborative sharing of resources between the two runtimes. Figure 4.1 shows the structure of the joint compilation flow. Eventually, OmpSs task-graphs that require minimization of inter/intra resource interference or include energy-critical computations can be seamlessly offloaded to XiTAO, while ensuring that OmpSs and XiTAO tasks independently run on the allocated subset of the system resources. The integration repository, available on: https://github.com/legato-project/nanos-xitao-integration, showcases interoperability between the XiTAO and Nanos runtime libraries (OmpSs backend) using a block-tiled matrix multiplication. The integration has the following dependencies:

- The XiTAO runtime library, which can be found at https://github.com/mpericas/xitao.git
- Nanos6 runtime for OmpSs-2 (tested with version 2.3.2) https://github.com/bsc-pm/nanos6.git
- gcc/g++ version >= v7.5.0
- To enable testing, cblas compliant API such as cblas-lapack (e.g., http://www.netlib.org/lapack/#_lapack_version_3_9_0_2)

4.2. OmpSs–DFiant Integration

The OmpSs–DFiant integration hooks into the existing OmpSs@FPGA–Vivado HLS interface and compilation flow.

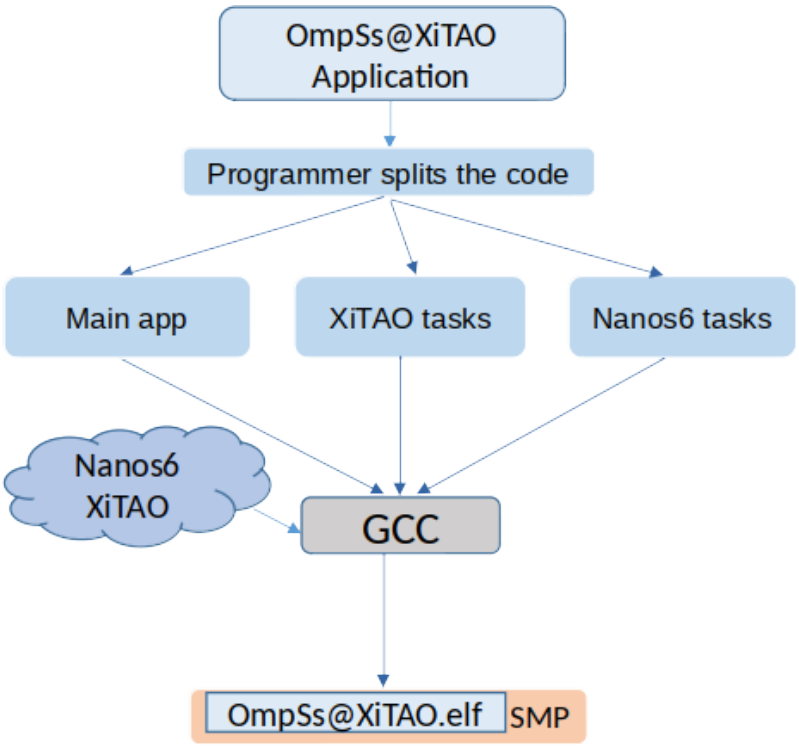


Figure 4.1. Integration of OmpSs and XiTAO environments

4.2.1. Interface Hooks

Figure 4.2 shows the compilation flow. OmpSs generates HLS C-code that includes the kernel task function and wrapper functionality. The task code manifests in RTL as the Vivado `ap_ctrl` interface and one of the following interfaces for its input/output variables:

- Array memory-mapped interface. This interface is applied when the OmpSs task kernel uses a constant `SIZE` for its vector arguments. The generated OmpSs wrapper constructs one or more arrays that are accessed by the kernel through a memory mapped interface. The number of arrays are determined by the `array_partition` HLS pragma. Better optimization may be possible if the AXI4 interface is applied instead (see next).
- AXI4 master interface + offset. This interface is applied when the OmpSs task kernel uses a dynamic `size` for its vector arguments. In this case, the dynamic size forces OmpSs to rely entirely on the kernel for optimization and thus provides full AXI Master interface (plus offset) for each variable. It is recommended to use this design option when we wish to control the entire optimization in HDL.

To hook on the kernel interface and provide an alternative design in DFiant, we created a dedicated `lib.ompss` API. Figures 4.3, 4.4, 4.5, and 4.6 provide examples of these interfaces and their equivalent DFiant hooks.

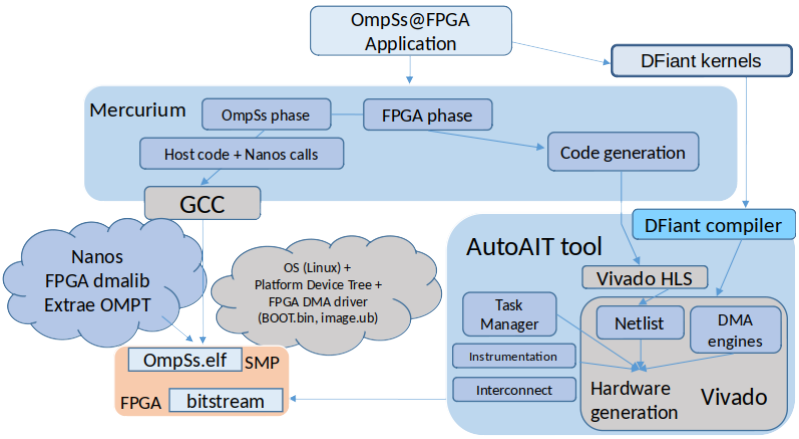


Figure 4.2. Integration of OmpSs with the DFiant environment

4.2.2. Compilation Flow Hook

The OmpSs compilation flow is executed via the arm gcc accelerator integration tool (AIT). The AIT executes several steps: generates the HLS IP from the kernel task, generates the entire FPGA design, synthesizes, implements, and finally generates a bitstream. To use DFiant with OmpSs, the Makefile just requires another dependency that compiles the DFiant design after the HLS generation and before the rest of the steps are executed (by utilizing the `from_step` and `to_step` AIT arguments).

4.3. OmpSs–Maxeler Integration

We have integrated the runtime environments of OmpSs and Maxeler, in such a way that we allow OmpSs applications to spawn tasks on Maxeler DFEs.

4.3.1. Overview of the Execution Environment

In this environment, programmers write the applications in C/C++ as it is regularly done with OmpSs, and they write the accelerated kernels using the Maxeler MaxJ language. The decision of which kernels to port to MaxJ from OmpSs is covered by Task T4.6 which is still ongoing.

Figure 4.7 shows the structure of the development environment. Compilation for the Maxeler device follows the usual toolflow provided by Maxeler. Starting from the MaxJ kernels, MaxCompiler generates the VHDL code, that is later synthesized by the Xilinx Vivado tool to generate the bitstream. Additionally, MaxCompiler generates functions which are callable from C/C++ to initiate data transfers and start the kernels in the Maxeler DFE, using the Maxeler SLiC interface. This is the interface that the OmpSs runtime will invoke to start the execution on the DFE.

The OmpSs part of the application is compiled with the CPU compiler (GCC, ICC). During initialization, the SLiC interface is used to register the kernel functions compiled through MaxJ, onto the Maxeler runtime.

The Nanos runtime creates a helper thread for each Maxeler DFE present in the system. Once tasks are created using the newly implemented *max* target device (see below), they are managed by such helper threads. Additionally, when a helper thread finds a task, it connects with the kernel function through the SLiC

```
#pragma omp target device(fpga) copy_deps
#pragma omp task in([size]i) out([size]o)
void loopback(int * i, int * o, int size) {
  //...
}
```

Figure 4.3. A loopback task OmpSs kernel signature

```
import lib.ompss._
@df class loopback_ifc extends OmpssIfc {
  val i    = OmpssAXI   <> IN
  val o    = OmpssAXI   <> OUT
  val size = DFBits(32) <> IN
}
@df class loopback extends DFDesign {
  val io = new loopback_ifc
  import io._
  //...
}
```

Figure 4.4. The DFiant design interface for the loopback in Figure 4.3

```
const unsigned int BSIZE = 16;
const unsigned int SIZE  = BSIZE*BSIZE;
#pragma omp target device(fpga) num_instances(1)
#pragma omp task in([SIZE]a, [SIZE]b) inout([SIZE]c)
void matmul(const elem_t *a, const elem_t *b, elem_t *c) {
  #pragma HLS array_partition variable=a cyclic factor=4
  #pragma HLS array_partition variable=b cyclic factor=BSIZE/4
  #pragma HLS array_partition variable=c cyclic factor=BSIZE/2
  //...
}
```

Figure 4.5. A matrix multiplication task OmpSs kernel signature

```
import lib.ompss._
@df class matmul_ifc extends OmpssIfc {
  val BSIZE = 16
  val SIZE  = BSIZE * BSIZE
  val a     = OmpssARR(SIZE, 4)          <> IN
  val b     = OmpssARR(SIZE, BSIZE / 4)  <> IN
  val c     = OmpssARR(SIZE, BSIZE / 2)  <> INOUT
}
@df class matmul extends DFDesign {
  val io = new matmul_ifc
  import io._
  //...
}
```

Figure 4.6. The DFiant design interface for the matrix multiplication in Figure 4.5

interface to transfer the data and execute the kernel in the DFE.

4.3.2. Runtime Implementation

We have extended the Nanos++ runtime system to support the new target device architecture, named *max*. This information is contained on a device descriptor, which invokes the Maxeler runtime when needed for the following functionalities:

- Memory allocation: to obtain and manage memory that is accessible from the Maxeler DFE devices.
- Data transfers: to move data in and out of the DFE device, with various flavors, single dimension vectors, 2-dimensional matrices, etc.

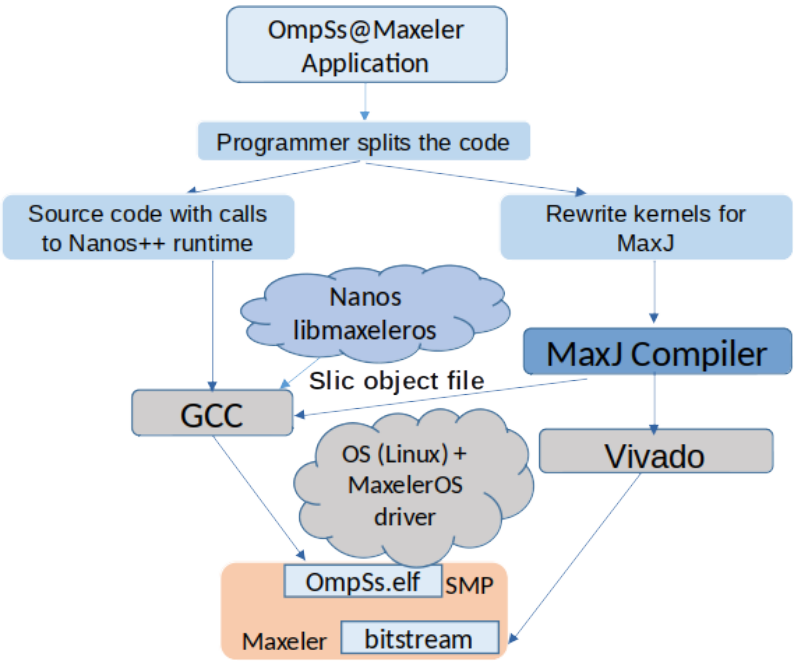


Figure 4.7. Integration of OmpSs with the Maxeler environment

- Kernel management: invoking kernels and triggering the data transfers are operations that the Maxeler DFEs accept through attached queues. When the task including the kernel is ready from dependencies, Nanos++ ensures to enqueue first the input data transfers, then the kernel invocation, and finally the output data transfers.

4.3.3. Current Tests

We have tested the environment in two different Maxeler platforms:

- MAX3 environment: Intel Xeon X5690 @3.4Ghz, with MAX3 boards.
- MAX5 environment: Intel Xeon, with MAX5 boards.

We have run the matrix multiplication kernel on both platforms.

5. Energy-Efficiency

This Chapter is composed of five sections presenting an overall status update on LEGaTO's tools for energy efficiency. Section 5.1 makes a small reference to XiTAO's energy efficiency, which is entirely reported in Deliverable D3.3. Our work on HERMES, an application-level protocol for the data-plane that can operate over generalized deduplication, as well as over classic deduplication is introduced in Section 5.2. HERMES significantly reduces the data transmission traffic while effectively decreasing the energy footprint, a relevant matter to consider in the context of IoT deployments. Section 5.3 presents an evaluation of HEATS, the energy-aware orchestrator for containerized applications already introduced in Deliverable D4.2. In Section 5.4 we present an evaluation of SCONE, also already introduced in previous deliverables. Our evaluation compares native and SCONE applications regarding energy consumption, which is due to additional encryption, decryption and paging. Finally, our initial efforts on integrating SGX enclaves with OmpSs are introduced in Section 5.5, with a first impression on energy consumption with a multithreaded matrix multiplication application.

5.1. The XiTAO Energy Aware Scheduling

XiTAO is a lightweight layer that provides a task-parallel and data-parallel interface using modern C++ features. The design goals of XiTAO are to be low-overhead and to serve as a development platform for testing scheduling and resource management algorithms. The new energy related features pertaining to XiTAO are at the runtime level, hence, they are described in D3.3. For convenience, we summarize here the most important features that were developed as part of the energy-efficient XiTAO runtime system.

XiTAO Virtual Places Mapping: The granularity of the software topology mapping is managed by XiTAO at runtime. Rather than mapping to a single core, the runtime can decide to mold a task on more than one core without disrupting the specified locality. To aid the mapping, a hardware layout description may be optionally passed to the runtime. This particularly convenient when targeting asymmetric platforms in which different clusters feature heterogeneous numbers of cores. The runtime scheduler is designed to search for the optimal execution place for a task depending on an online performance model.

The Energy-Aware Scheduler (EAS): We have developed an energy efficient task scheduler that reduces energy consumption by determining energy-aware mappings for all tasks, which addresses the problem of scheduling task-DAGs on asymmetric systems (such as e.g. big.LITTLE) where frequencies are either fixed or externally managed by an OS power governor. The task scheduler features several components: power characterization profiles, a performance tracer, a global parallelism tracer and a local task mapping algorithm.

Support for Tensor-Expression Language on top of a Pipeline Parallel Scheduler: Commonly, applications expressed as streaming chain graphs (DAGs) are implemented as parallel pipelines for efficient scheduling on multi-core computing platforms. In XiTAO, an application is expressed as a DAG where

each node is a standalone Task Assembly Object (TAO). A TAO can be serial or parallel. Parallel TAOs contain their own local scheduler. The mode of parallelism inside a TAO is mostly data parallelism, but amorphous forms of parallelism are also possible. Thus, XiTAO is usually used as a mixed-mode parallel programming model, which is convenient to express streaming parallel pipelines. We implement pipeline stages as self-calling TAOs (i.e. making an edge to themselves). For the sake of usability and expressiveness, we designed a simple template based language that generates XiTAO code for to process deep neural network applications. Convolutional Neural Networks (CNN) consist of several convolutional layers in combination with computationally light layers such as maxpooling and softmax layers. Each layer processes the output of the previous layer and produces output for successive layers. The structure of CNNs resembles chain graphs. Moreover the inference phase of CNNs is applied on a stream of input units for example object identification in video stream. Thus, CNNs are a good case for experimenting with pipeline parallelism under the XiTAO environment.

5.2. Energy-efficient IoT Data Deduplication

The ever growing number of connected devices is also a result of the adoption and expansion of *Internet of Things* (IoT) technologies. IoT devices are most commonly low-energy, yet efficient and powerful devices which disseminate data on the Internet. This data is predicted [13] to amount to 175 ZB by 2025. Furthermore, the number of deployed IoT devices will grow to 1.25 billion units by 2030 [11]. The pressure on the Internet will continue to increase and with it challenge researchers to come up with new innovations (*e.g.*, 5G and beyond) to avoid its collapse.

Data compression [20], deduplication [25] or *network coding* (NC) techniques [40] have been proposed to overcome these challenges. On the one hand, the redundancy introduced by NC is interesting given the unreliable nature of the data streams commonly found in real-world IoT deployments [6]. On the other hand, compression and deduplication are interesting given the compression potential of IoT-generated data (*e.g.*, smart power meters [27], weather stations [16], bio-medical body sensors [10]). To demonstrate the high compression potential, we applied different compression algorithms to a real-world IoT data set in Figure 5.1 from the domain of ambient water and energy [5, 8].

One problem of standard compression algorithms [14, 30] in the IoT domain are their computational requirements, while lightweight, memory-efficient approaches tend to have lower compression potential [38, 45, 46]. To make matters worse, IoT applications often rely on *small data packets* and *compress data on a per packet basis* due to memory limitations, which curbs the compression potential of standard compression algorithms [41]. Another limitation of state-of-the-art approaches is that compression is provided by finding only equal data chunks. Thus, two chunks will be considered different, if they differ in a single bit. Although techniques such as Rabin fingerprinting [32] can be used to split data in non-uniform chunks (*i.e.*, different sizes) in order to detect similarities, they also require significant computation and memory resources. All these limitations are obstructive for delivering energy- and memory-efficient protocols, in particular

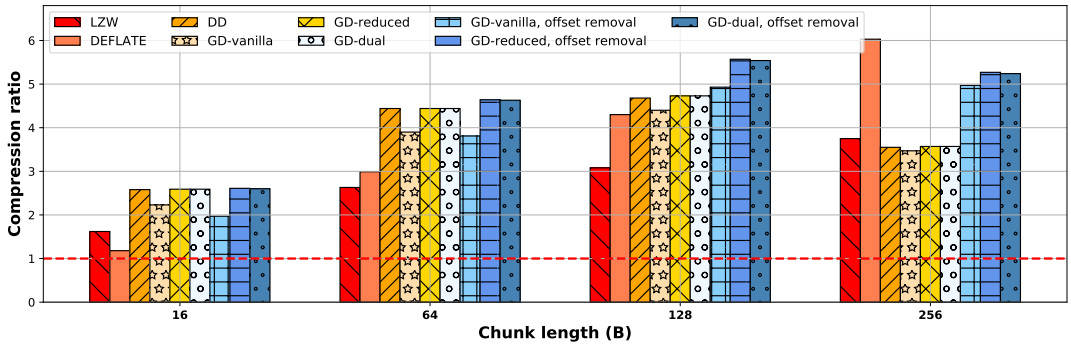


Figure 5.1. Compression ratio real-world water meter measurements IoT data set under different algorithms. The original data size is divided by the compressed data size - higher is better.

if the nature of the data is not known beforehand. In fact, the compression ratio for two standard compression algorithms LZW [38] and DEFLATE [14] decreases dramatically for smaller chunk lengths as shown in Figure 5.1.

A well-known technique to reduce network traffic is network deduplication [36]. With network deduplication repeating byte sequences are replaced with a shorter hash value, which is later used to identify the intended content by the receiving side. However, this technique is typically designed for point-to-point transmission scenarios with large data chunks, large hashes, local caching operating on files where data is known *a priori*.

Generalized deduplication (GD) [37] is a recently introduced scheme which reduces the cost of storage not only by finding equal data chunks, but also by finding similar data chunks. Similarities between chunks are identified *without* carrying out delta compression to a pool of previous chunks and *without* relying on similarity hashes for dynamic chunking as in other lossless compression schemes. The latter would be impractical for small amount of generated data in IoT devices. A lossless, multi-source data transmission compression approach inspired by the concept of GD was proposed in [41] to reduce the amount of data transmission. Figure 5.1 is showing the potential of GD to outperform *data deduplication* (DD) in IoT scenarios, and also to outperform LZW and DEFLATE for small packet sizes.

In this section we introduce HERMES, a protocol and its corresponding implementation for data transmission reduction in IoT networks, especially suited for resource-limited data nodes. Its design principles are inspired and expand the schemes proposed in [41]. HERMES allows multiple source nodes to share a common data pool at the sink node, typically a cloud- or edge-based device. All source node transmissions contribute to growing the data pool. Spatial data correlations (*e.g.*, similar temperature data at the same time across the same city, similar smart meter consumption of several households) as well as temporal correlations (*e.g.*, same electricity readings of house A today as house B a year ago) across multiple source nodes can be exploited to reduce data transmission and allow for better compression at the sink node. Thus, each device is benefiting from contributions of other devices in order to reduce their traffic, without direct interactions between devices.

5.2.1. Background on Data Deduplication

Let us define $f(.)$ as a function that associates a (nearly) unique fingerprint to its input, either using standard hash functions (*e.g.*, SHA-1, SHA-256) or checksums (*e.g.*, CRC32, MD5). Based on the fingerprint collision probability requirement, a deployment of HERMES should settle on a specific $f(.)$, but also the size of the fingerprint in relation to the amount of data transmitted per packet. The resulting compression gains are partially related (and limited) by the latter, *e.g.*, a payload of 40 B using a SHA-1 fingerprint of 20 B will not compress beyond a factor of 2.

Data Deduplication (DD) eliminates redundant data by removing copies of repeating data chunks. Classic DD divides each piece of data into multiple data chunks, C_i, and stores each unique data chunk exactly once by distinguishing repeated data chunks. A fingerprint is then linked to each unique data chunk. Thus, a piece of data can be represented as a sequence of fingerprints and their corresponding data chunk. The original piece of data can then be recovered by searching for the data chunk associated to each fingerprint and concatenate the data chunks in the right order.

Generalized Deduplication (GD) [37] is a lossless data compression approach, that eliminates identical as well as similar data chunks. A piece of data is first split into multiple data chunks, C_i, that are then mapped onto a pair of basis, b_i, and associated deviation, d_i, by applying a transformation function in the form of an *error-correcting code* (ECC). Identical and similar data chunks are identified by comparing their basis. Each basis b_i is stored exactly once gets assigned a fingerprint $f(b_i)$. For simplicity, from here on the notation f_{b_i} will be used instead of $f(b_i)$. GD stores a pair f_{b_i} and d_i rather than C_i. It should be noted that DD can be considered as a special case of GD with $d_i = 0$, and $b_i = C_i$.

Data transmission can be reduced with GD in a lossless manner [41]. Source nodes apply GD onto each data chunk C_i to obtain a pair (b_i, d_i). In order to reduce network overhead, the source node first transmits the pair (f_{b_i}, d_i) to the sink node. The basis b_i is then transmitted if and only if it is not available at the sink node. After receiving the acknowledgement from the sink node, a source node can erase all data (*i.e.*, chunk, basis and deviation) to free up space in memory. Without loss of generality this process can be generalized by transmitting a batch of pairs (b_i, d_i) in a single packet. Since all source nodes leverage the same hash function $f(.)$, each of them is exploiting the set of all bases fingerprints available at the sink node, independent of its origin source node.

A variety of functions can be used to create a mapping from C_i to (b_i, d_i). We rely on ECC, where C_i is the *codeword* and, by applying the decoding function of the ECC, the received *message* is the basis b_i. The deviation d_i is obtained by taking the difference between the codeword and the *error-free codeword*. The latter is obtained by encoding the basis b_i using the ECC encoding function.

Hamming codes. Hamming codes [26] are a family of linear ECCs, that can be used to establish the mapping [29]. Let m be the number of parity bits, the codeword and the message are of length $n = 2^m - 1$ bits and $k = 2^m - m - 1$ bits, respectively. The corresponding byte-length of the codeword and message is then obtained by $n_B = \lceil \frac{n}{8} \rceil$ and $k_B = \lceil \frac{k}{8} \rceil$, respectively. Since Hamming codes

can correct single-bit errors, this means that codewords are at most one bit apart from their error-free codeword. This means that we systematically match data chunks to one another that differ in a single bit. The location of this bit is specified in a *syndrome vector* of length m bits. With Hamming codes, the last bit in the last byte is never used due to the length of the codeword which is $n = 2^m - 1$ bits. Therefore, we consider data chunks of length $n + 1$ bits to represent data in bytes. As a result the Hamming transformation is applied on the first n bits and the remaining bit is left untouched as part of the deviation (concatenated with the m deviation bits). Thus, the resulting deviation for data chunks of size $n + 1$ bits is $m + 1$ bits long.

Reed-Solomon codes. Another family of ECCs that can be used to establish a mapping [37] are Reed-Solomon codes [39]. Reed-Solomon codes operate on a block of data treated as a sequence of symbols from a finite field of size q, $\mathbb{F}_q$. For $q = 256$, symbols are 8 bits (1 byte) long. The codeword and message length are $n_B = q - 1$ and $k_B < n_B$, respectively. Unlike Hamming code, Reed-Solomon codes can correct more errors. The error correction capability of the code is given by $t = \lfloor \frac{n_B - k_B}{2} \rfloor$. Our implementation is using a short version of the code [26], where $k_B < n_B < q$.

The deviation can be computed by a bitwise XOR of the original data chunk and the error-free data chunk. However, with Reed-Solomon codes it is not possible to predict the length of a deviation, but it is possible to derive the maximum length of the deviation. To this end we consider the *covering radius* of the code $R(\mathcal{C})$, which is the maximum Hamming distance any data chunk is apart from its associated error-free codeword. Thus, we will need a maximum number of bits for to represent the location and content of non-zero symbols given by $R(\mathcal{C}) \cdot \lceil \log_2 n_B \rceil$ bits and $R(\mathcal{C}) \cdot \lceil \log_2 q \rceil$ bits, respectively. The deviation is the result of the XOR operation applied to the location and the content of the non-zero symbols.

Deduplication is used widely to reduce both storage and transmission cost [31]. *Asymmetric caching* was proposed by Sanadhya *et al.* [34], where a source node performs deduplication on the outgoing chunks of data based on its cache content and the sink node's feedback, similar to what was originally done with Web caches in [36]. Timely feedback is sent by the sink node to the source node containing a selected portion of its cache that is most likely going to increase the probability of matching. [44] describes a deduplication-based file communication system that leverages manifest (hash values, addresses and sizes of data chunks) feedback. Each file is split into data chunks and associated hash values on the source node. In case of a deduplication cache misses, the hash values of missing data chunks are sent to the sink node for further duplication detection. Response messages from the sink node include the query information and the manifests of the chunks that have been matched at the sink node. In [19] a traffic deduplication approach is proposed to merge independent streams of the same video content on the Internet using a novel overlay network. An enabled router can then merge a matching video stream, if their identifiers are the same. We leverage previous works on the domain to propose HERMES, which allows to reduce data transmission for IoT applications while still offering trade-offs for saving energy.

5.2.2. HERMES architecture

In order to design and deploy a distributed IoT network using HERMES, we assume the following: *(1)* all nodes use the same fingerprint length; *(2)* all GD nodes use the same transformation configuration; and *(3)* all nodes use the same chunk length.

A distributed deployment of HERMES allows for different types of nodes: *source*, *sink* and *intermediate* nodes. Source nodes inject data into the network, while sink nodes ingest data without further retransmission. An intermediate node is any node between a source and a sink node. Each node handles messages according to its own class: *basic*, *deduplication* and *generalized deduplication*. Basic nodes serve as a pass-through and do not process data chunks. Deduplication nodes perform DD on incoming data chunks. Finally, a generalized deduplication node performs GD on incoming data chunks. All nodes respond with success, acknowledgement or failure messages to communicate their state in the system. We devise three different communication mechanisms on top of GD depicted in Figure 5.2.

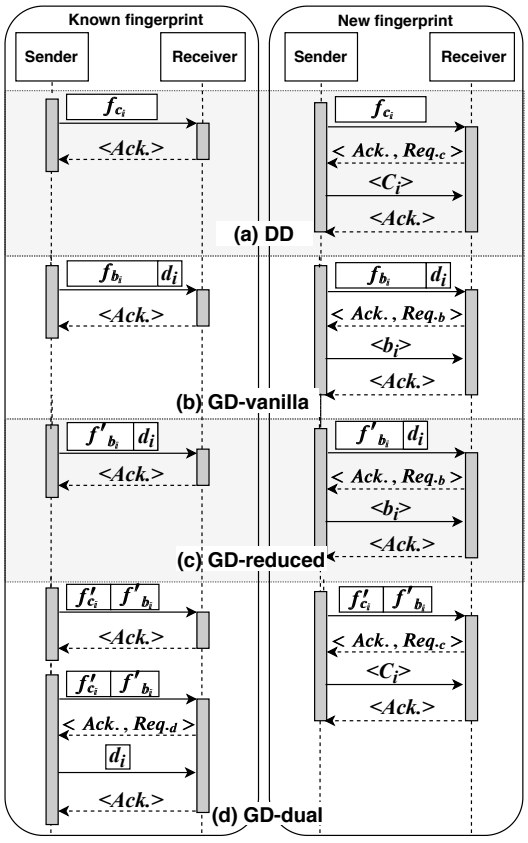


Figure 5.2. Communication mechanisms.

Baseline deduplication (DD): Figure 5.2 (a) presents the baseline mechanism on top of DD. Each fingerprint corresponds to a unique chunk value (with high probability given the fingerprint function).

Baseline gen. deduplication (GD-vanilla): A similar mechanism (b) Figure 5.2 which uses the same fingerprint length as for DD can also be applied on top of GD. With this setup GD-vanilla may incur a slightly larger overhead than DD if an exact duplicate chunk is already in the sink node, since the transmission of the

deviation would be redundant.

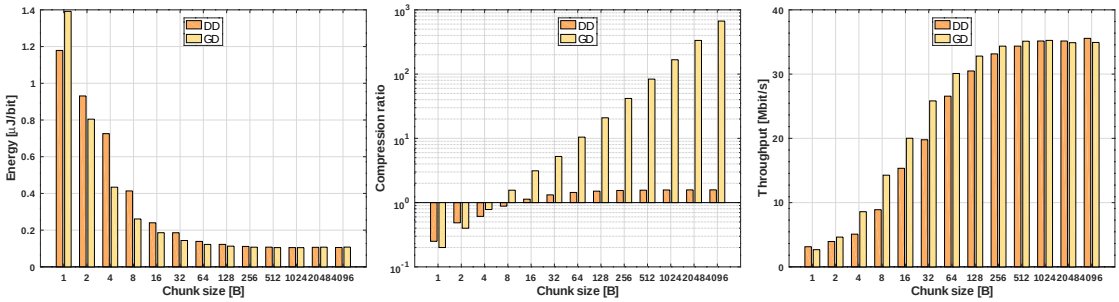
Reduced fingerprint gen. deduplication (GD-reduced): Figure 5.2 (c) shows a variant that compensates the overhead of GD-vanilla. Since the bases for GD require fewer bits than the data chunks, we can consider a system such that the length of the fingerprint plus the deviation is equal to the length of the fingerprint of mechanisms (a) or (b). While this variant removes the transmission overhead, it increases the probability of collision of the fingerprint.

Dual fingerprint generalised deduplication (GD-dual): Finally, we describe a hybrid approach Figure 5.2 (d), which simultaneously transmits the fingerprint of the data chunk, f'_{c_i}, and of the basis, f'_{b_i}. The fingerprint length is chosen to be half the length of f_{c_i} in (a). If a given data chunk is already available at the sink node, both f'_{b_i} and f'_{c_i} will match. Otherwise, if a similar data chunk matches to the same basis, f'_{b_i} will match, but not f'_{c_i}. Depending on the availability of the data chunk and basis, the sink node will either acknowledge the message or request the missing part.

5.2.3. Evaluation of HERMES

Our experiments are deployed over a switched cluster of 16 Raspberry Pi 4B[1] featuring a Raspberry Pi PoE-HAT[2] to enable 802.3af Power-over-Ethernet [24]. We deploy a simple network topology where Raspberry Pis are used as source nodes and are connected to a Dell PowerEdge R330 server acting as sink node. The Raspberry Pis are powered using PoE by an Ubiquiti Networks UniFi USW-48P-750 switch and are connected over a Gigabit Ethernet link.

In the micro-benchmark we present the results of running the approaches locally on a Raspberry Pi 4B and in the macro-benchmark we present the combined results of running the approaches on the Raspberry Pi 4B cluster.



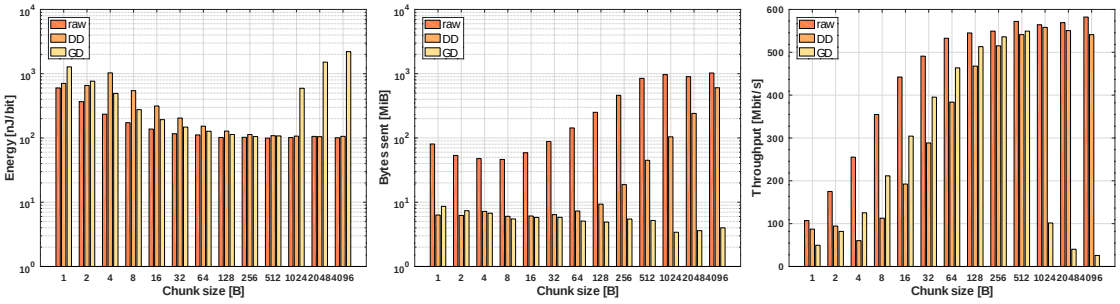
(a) Energy per bit (b) Compression ratio (c) Throughput

Figure 5.3. Micro-benchmark on Raspberry Pi 4B.

Micro-benchmark: Figure 5.3 shows the results of running the synthetic data set on the micro-benchmark. Results in Figure 5.3a show that energy per bit performance of GD is comparable to DD even considering the added overhead due to the transformation computation. For chunk sizes of 4 B to 8 B GD is clearly more energy-efficient than DD. GD compresses the data chunks significantly better than DD for chunk sizes of 8 B and above. For example, with chunks of 64 B GD requires an order of magnitude fewer bits than for DD and for chunks of

[1] https://www.raspberrypi.org/products/raspberry-pi-4-model-b/
[2] https://www.raspberrypi.org/products/poe-hat/

4096 B it increases to three orders of magnitude. This is due to the large number of similar chunks matched to the same basis. Finally, our system's achieves a maximum throughput at about 35 Mbit s^{-1} for a single thread. To measure the throughput we have considered all operations: reading the data from memory, applying the compression algorithm and looking for fingerprints in memory to check the availability of the data chunk for DD and GD. In a real world scenario, a source node only needs to apply the compression algorithm and the sink node will handle fingerprint lookups. Thus, in real deployments we expect a higher throughput. Furthermore, we could consider using NEON instruction sets in ARM to speed up processing.



(a) Energy per bit (b) Network traffic generated (c) Throughput

Figure 5.4. Macro-benchmark on Raspberry Pi 4B cluster.

Macro-benchmark: For the macro-benchmark we evaluate three approaches methods: raw, DD and GD. By raw, we designate sending raw data chunks and consider it as our baseline. In our setup we use one thread to handle the networking and a second thread to do the necessary transformations and look-ups. Figure 5.4a shows that for smaller chunks the energy is higher. This is due to the inaccuracy of the measurement which is twofold: (1) the switch has a low time resolution to update the PoE status statistics and (2) the short runtime of the benchmark. As the chunk size (larger than typical IoT ones) increases, so does the energy for GD due to the computational overhead. Most importantly the network traffic generated for the different approaches in Figure 5.4b is reduced significantly with GD. As seen in the micro-benchmark, the macro-benchmark also shows a optimal throughput for chunk sizes 4 B to 64 B. Approaching 1024 kB we recognize a significant drop in throughput for GD. We assume this is primarily due to hardware limitations, like exceeding cache limits.

5.3. HEATS Scheduling Policy

HEATS is a task-oriented and energy-aware orchestrator for containerized applications targeting heterogeneous clusters, first introduced and described in Deliverable D4.2. HEATS is a prototype scheduler, implemented as a plug-in within Google's Kubernetes, allowing users to trade performance for energy requirements. It learns the performance and energy features of physical hosts, monitors the execution of tasks on the hosts and opportunistically migrates them onto different cluster nodes, to match the customer-required deployment trade-offs.

In this section we present an evaluation of HEATS' Scheduling Policy, also described in Deliverable D4.2. We first describe our experimental settings and the synthetic trace used to compare HEATS against the default settings using un-

modified Kubernetes. We then compare both schedulers in terms of energy and resource utilization. We analyse how the user demands (energy/performance ratios) affect the observed performance. Finally, we look at the impact of the rescheduling frequency on the overall job runtime. The evaluation indicates that our approach can yield considerable energy savings (up to 8.5%) and only marginally affect the overall runtime of deployed tasks (by at most 7%).

	Arch.	Cores	Frequency	TDP	Mem.
ARM Cortex-A53	BIG.LITTLE	4	1.4 GHz	5 W	1 GiB
AMD Epyc 7281	amd64	32	2.1 GHz	155 W	64 GiB
Intel Xeon E3-1270 v6	x86	4	3.8 GHz	72 W	64 GiB
Intel Xeon E5-2683 v4	x86	32	2.1 GHz	120 W	128 GiB

Table 5.1. Hardware characteristics of our cluster.

Evaluation settings. We deploy and conduct our experiments over a cluster composed of 4 different types of machines (see Table 5.1). Our cluster is composed of 9 machines, where one is the Kubernetes master, orchestrating the deployments and the remaining nodes are workers executing the tasks. The 8 worker nodes consist of one AMD, 3 Intel and 4 ARM machines. The energy consumption is measured using a LINDY iPower Control 2x6M power distribution unit (PDU) for the server type machines and PowerSpy devices for the three Raspberry Pi. The PDU records up-to-date measurements for the active power at a resolution of 1 W and with a precision of 1.5%. We query it up to every second via HTTP.

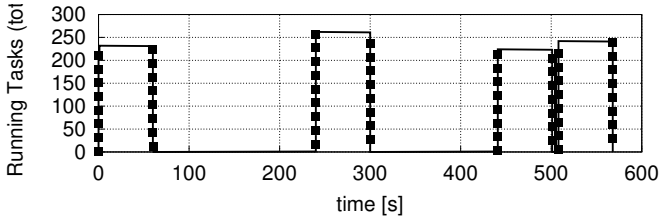


Figure 5.5. Workload injected by the synthetic trace: tasks arrive in 4 bursts of up to 262 concurrently running containers.

Synthetic trace. We use a synthetic trace to evaluate the gains and trade-offs of our system. Figure 5.5 shows the workload injected by this trace. We use it to deploy multithreaded tasks executing an iterative implementation of the `k-means` algorithm in the C programming language. The program, shipped as statically linked binary for Alpine Linux, executes over a predefined dataset of 65 536 data points along 32 dimensions. Once deployed, the tasks will compute clusters by splitting the dataset into blocks processed by two worker threads for a specified maximum number of iterations, chosen randomly in the range of 500 to 1000. The result is stored as file inside the container's image. In total, 480 `k-means` jobs are deployed following four bursts over 10 minutes, executed randomly within a time frame of 150 seconds. The same sequence of pseudo-random numbers is ensured upon every run of a trace by using a fixed random seed.

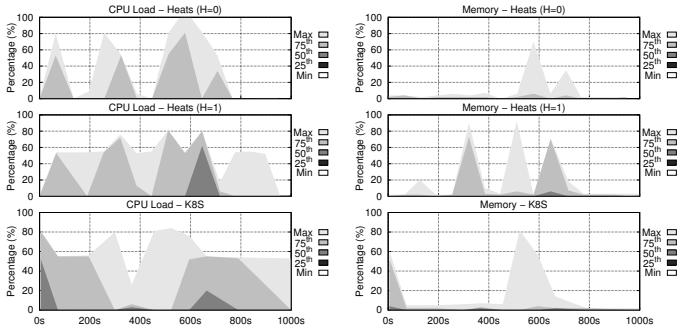


Figure 5.6. CPU and memory usage distribution (percentiles) across all machines in the cluster. We show these metrics with three different schedulers: HEATS in two configurations (H=0, H=1) on the first and second row, Kubernetes in the third row.

Kubernetes vs. HEATS. First, we compare the CPU load induced on the cluster by HEATS against the default scheduling policy of Kubernetes. Figure 5.6 shows these results. We observe how the load patterns are very similar and closely follow the arrival pattern of the tasks in the trace Figure 5.5). We conclude that the HEATS scheduler does not deteriorate the lifetime of the processors by artificially stressing them. Next, we look at the memory usage across the cluster. Figure 5.6 shows this for two different HEATS configurations, one for performance (e = 0, p = 1), the other for energy-efficiency (e = 1,p = 0). The memory load of two schedulers induce similar patterns.

We compare the energy efficiency of the default Kubernetes scheduler against HEATS. Figure 5.7 presents the total runtime and the cumulative energy consumption of the cluster throughout the execution of the trace, including their idle state requirements. We compare five different approaches: (1) the default scheduler (k8s), (2) HEATS configured to deploy tasks on the fastest possible machines, ignoring any energy concerns (e0p1), (3) HEATS trying to be as energy-efficient as possible. Moreover, for the sake of comparison, we include the results achieved by (4) a fixed *H* value chosen out of our practical experience (rand, for e = 0.618, p = 0.382) and (5) other variations of the H-value. When compared to the default scheduler (k8s), (2) performs better on an energy cost of 1.5% while (3) performs worse but presents 7.1% of energy savings. Besides, when compared to each other, (2) performs better while (3) is more energy efficient. Finally, for approach (4) we can observe that the runtime as well as the energy consumption are in between the observations for approach (2) and (3). Therefore, we can conclude that our observations follow the expected behaviour.

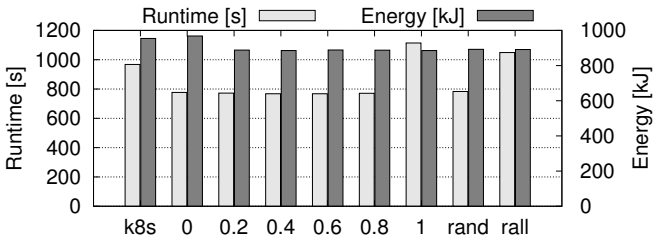


Figure 5.7. Energy efficiency and impact on the overall runtime of the trace for several scheduling policies.

Energy vs. performance weights. The value chosen for the *H* parameter is of paramount importance, especially when considering the resulting energy costs

and impact on the overall runtime of the jobs. To better understand this aspect, we choose 6 different configurations (from 0 to 1, by increments of 0.2), for different energy/performance ratios, 0 being the least and 1 the most energy-efficient versions. We compare the achieved results with a HEATS configuration that randomly select the value of *H*, mimicking a customer with no particular requirements. Figure 5.7 shows our results. For each configuration, we show the cumulative energy costs (in kJ) and the achieved runtime, respectively on the left and right vertical bars. We observe how the configurations achieve similar results, with a sensible deviation only with the less energy efficient variant. While these results require further investigations, we believe them to be of practical interest for end-users. We intend to confirm these by evaluating the same configurations on real-world traces, where the variations of the *H* parameter might have more impact.

5.4. Secure Energy-efficient Computation using SCONE

In LEGaTO, we ensure the confidentiality and integrity of the applications against potential attacks from the privileged system software (or attackers with privileged accesses) by developing security mechanisms based Trusted Execution Environments such as Intel SGX [12]. We extended our toolchain SCONE [7] to not only enable legacy applications to run inside Intel SGX enclaves but also provide a key management system to securely transfer the configuration and secrets (e.g., encryption/decryption keys, certificates) for running these applications after attesting them [17, 18]. However, there is no free lunch as usual, running applications with Intel SGX incurs a performance overhead since the fact that it requires extra computation power to carry out encryption/decryption operations. In addition, current Intel SGX hardware supports only a limited memory space ($\sim 94MB$) for applications running inside enclaves. If applications allocate more memory than the EPC size limit, it is required to perform encrypting data and paging mechanisms which are very costly. Furthermore, this computation overhead introduces an extra energy consumption when running these applications.

In LEGaTO, our goals are not only providing security but also achieving energy-efficiency for applications running on top of our proposed framework. To balance and optimize these goals, we designed and implemented our toolchain SCONE in the way that reduces the overhead. For example, we modified the C library musl [2] instead of glibc [4] to reduce the trusted computing base size. We implemented an asynchronous system call interface so that the application inside the enclave does not need to exit the enclave to execute system calls. In addition, in SCONE, we handled many system calls inside enclaves without interacting with the kernel.

To demonstrate the efficiency of our design, in the context of LEGaTO, we integrated SCONE with Tensorflow — a widely used machine learning framework in industry to build a secure machine learning framework. We also evaluate the energy consumption of Tensorflow with SCONE in performing machine learning computations. We conducted experiments to measure energy consumption of the machine learning framework in a training model for image classification. We selected this workload since it can be extended further to train models for medical/ healthcare images in a secure and energy-efficient manner as a project use

case.

We used of a Laptop ThinkPad T480 (Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, 16GB of memory, running Ubuntu 18.04.1 LTS) and $powerstat$ tool [3] to measure the energy usage by the SCONE-Tensorflow framework and native Tensorflow in training Cifar-10 image dataset [21]. The size of this dataset is $\sim$ 178MB, it contains a total of $60,000$ pictures with size 32×32 pixel. Each picture belongs to one of ten classes, which are evenly distributed, making a total of $6,000$ images per class. We split the dataset into $50,000$ training images and $10,000$ testing images. We use a simple Convolutional Neural Network (CNN) [1] for the training.

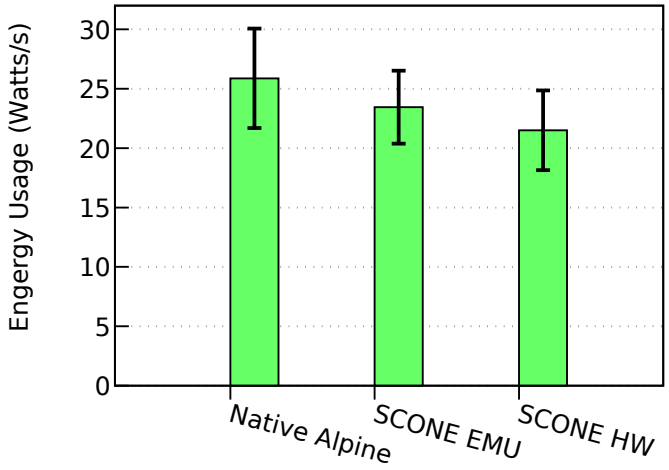


Figure 5.8. The average of energy consumption of native Tensorflow and Tensorflow with SCONE in training Cifar10 dataset.

Figure 5.8 shows the average energy consumption of the training using native Tensorflow running with a Alpine Docker container, with SCONE containers in both hardware (HW) mode and simulation (EMU) mode. Every second, the native Tensorflow consumes $\sim$ 25.88Watts, while Tensorflow with SCONE consumes $\sim$ 23.45Watts and $\sim$ 21.50Watts in EMU mode and HW mode, respectively. However, note that the native version took only 49 seconds for the training, meanwhile the SCONE version took 82 seconds in EMU mode and 152 seconds in HW mode. To further reduce the energy consumption, we need to reduce the processing time the systems running with SCONE. To achieve that we can tune the configuration of SCONE which defines how many enclave threads and system threads can be use for the Tensorflow application.

5.5. OmpSs–SGX Integration

We have integrated the use of Enclaves within OmpSs tasks. Figure 5.9 shows the compilation flow followed by our approach. The programmer inserts the Enclave e-calls inside OmpSs tasks, and compiles the application with Intel-provided Enclave support. Execution of the OmpSs application proceeds normally, starting outside of the enclave, and some tasks when run, use the resources of the Enclave, to perform secure execution.

All the experiments described in this section were performed in a CPU server 4-core (2 threads per core) Intel E3-1275 processors and 64 GiB of RAM, 480 GB storage. The machine runs Ubuntu Linux 16.04.1 LTS. We rely on SGX driver and

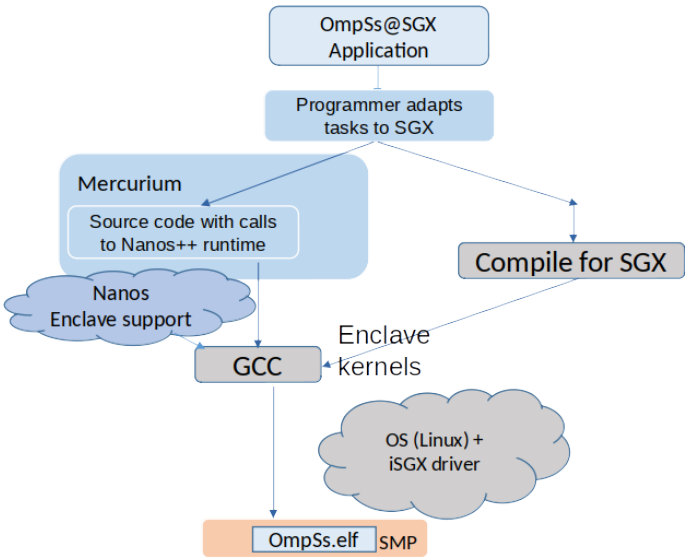


Figure 5.9. Integration of OmpSs with SGX Enclaves

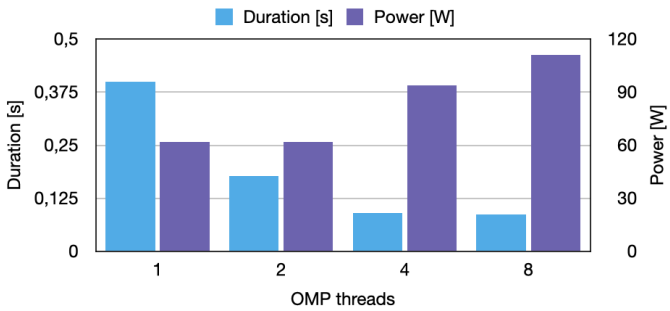


Figure 5.10. Matrix multiplication application implemented with OmpSs.

SDK, v2.0. We install OmpSs-1 (Mercurium compiler and Nanos++ runtime). The power consumption is reported by a network connected LINDY iPower Control 2x6M Power Distribution Unit (PDU). The PDU can be queried up to every second over an HTTP interface and returns up-to-date measurements for the active power at a resolution of 1W and with a precision of 1.5

We start our analysis by running a multithreaded matrix multiplication application relying on OmpSs for the parallelization. Figure 5.10 bellow shows the characterisation of this application regarding performance. Here we can observe that more threads yields in better performance regarding duration but worse performance when it comes to energy consumption. This happens because the power consumption increases as the utilized number of threads also increase and the gain in runtime is only fractions of a second.

In Figure 5.11 we observe the same matrix multiplication application but now integrated with Intel SGX. In the current version of this scenario 100% of the tasks are executed inside the enclave. As in this example the matrix is very small, the application does not surpass the EPC limit and therefore we observe an insignificant overhead due to the added security mechanisms.

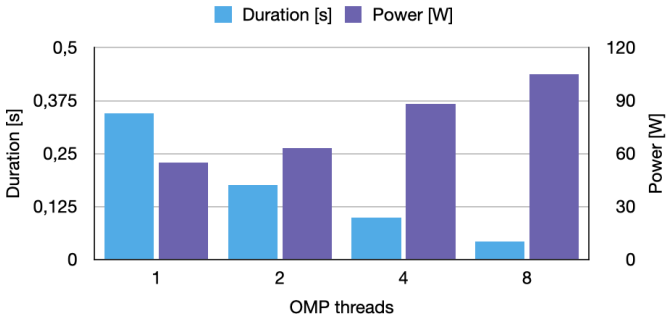


Figure 5.11. OmpSs matrix multiplication application integrated with SGX.

6. Fault-Tolerance

This Chapter on Fault Tolerance commences with the adaptation of ALYA, the smart city use case software library, to utilize the Fault Tolerance Interface, the LEGaTO checkpointing solution in Section 6.1. Finally in Section 6.2, we discuss ensuring Fault Tolerance within Scone, the LEGaTO security solution whoose security aspects were discussed in the previous Chapter.

6.1. Alya Integration with FTI

In the previous deliverables D3.2 and D4.2 we discussed several features that we develop to the FTI multilevel checkpointing library. In order to test the efficacy and efficiency of those features we have integrated the Alya application (relevant to the Smart City use-case) with FTI to do fast multilevel checkpointing. Alya is an application capable of simulating the pollution level of a city by simulating the wind flow in the streets following the 3D structure of the city and the traffic data in the streets. Such simulations can take hours and they are prone to failures. Therefore, it is important to protect them with checkpointing while keeping its overhead very low.

Alya is a large scientific application with tens of thousands of lines of code in Fortran90. Therefore, making changes in such a code is not always an easy task and it tests the simplicity of the API of runtime libraries. We are glad to report that integrating FTI into Alya was a relatively straight forward task. Only 4 files needed to be edited and it took only about a hundred lines of code to integrate both checkpoint and restart with FTI.

```
1  #ifdef __FTI__
2      !
3      ! Write restart file using FTI
4      !
5      if ( FTI_write_ckpt == 1_ip) then
6          FTI_Tmp_ptrTime=> ittim
7          FTI_Tmp_ptrnPart => npart
8          FTI_Tmp_ptrcuTime => cutim
9          FTI_Tmp_ptrdpthe => dpthe
10         FTI_Tmp_ptrprthe => prthe
11         call FTI_setIDFromString("ittim", FTI_id)
12         call FTI_Protect(FTI_id,FTI_Tmp_ptrTime,FTI_error)
13         call FTI_AddVarICP(FTI_id,FTI_error)
14
15         call FTI_setIDFromString("cuTime", FTI_id)
16         call FTI_Protect(FTI_id,FTI_Tmp_ptrcuTime,FTI_error)
```

```
17          call FTI_AddVarICP(FTI_id,FTI_error)
18
19          call FTI_setIDFromString("ptrnPart",FTI_id)
20          call FTI_Protect(FTI_id,FTI_Tmp_ptrnPart,FTI_error)
21          call FTI_AddVarICP(FTI_id,FTI_error)
22
23          call FTI_setIDFromString("ptrprthe", FTI_id)
24          call FTI_Protect(FTI_id,FTI_Tmp_ptrprthe,FTI_error)
25          call FTI_AddVarICP(FTI_id,FTI_error)
26
27          call FTI_setIDFromString("ptrdpthe", FTI_id)
28          call FTI_Protect(FTI_id,FTI_Tmp_ptrdpthe,FTI_error)
29          call FTI_AddVarICP(FTI_id,FTI_error)
30  #endif
```

Listing 6.1. Alya checkpointing with FTI

Listing 6.1 shows the part of the code in charge of checkpointing Alya with FTI. As we can see it takes less than 30 lines of code. Moreover, we make use of the Incremental Checkpoint (ICP) feature available to GPUs presented in D3.2. This is shown in the calls to `FTI_AddVarICP` which add only the target dataset into the checkpoint. In this code we can see that we protect a total of 5 datasets corresponding to a coherent state of the application after restart. We performed checkpointing Alya with an example test case, we obtained the final result of the computation, then we re-executed the code with checkpointing, injected a failure and restarted from the last recorded checkpoint. We let the second execution finish and compare the results with the failure-free execution. The results were identical, meaning that the checkpoint is correct and the execution accurate.

6.2. Fault Tolerance with Scone

Hardware and software fault are a major cause for errors in applications. They can corrupt the internal application state and, ultimately, they may lead to crashes, data loss, or to incorrect results of computations. In the LEAGaTo project, we introduce a compiler-based protection mechanism against software faults.

Our main focus is transient faults—the faults that change the internal application state only once or re-appear intermittently. In contrast to permanent faults that cause predictable and relatively stable state corruptions, transient faults are hard to detect because after the fault is over, the application appears to behave normally. Therefore, defining the source of the fault and mitigating it becomes extremely difficult. Such faults might have different reasons, such as overheating, weardown, manufacturing issues, or even natural radiation.

Transient faults could be also intentionally triggered by malicious actors in order to compromise the application security. For example, Plundervolt [28] is a recent attack that performs undervolting of Intel processor that can harm the integrity and confidentiality of SGX architecture. The root of this attack are bitflips (aka calculation errors) in the CPU instructions. Murdock et al. [28] show that a bitflip in the CPU can result in information leakage, unauthorized access, and stolen secret. Therefore, to ensure both correctness and security of the applications running on top of the LEGaTO platform, it must be adapted to tolerate these transient faults.

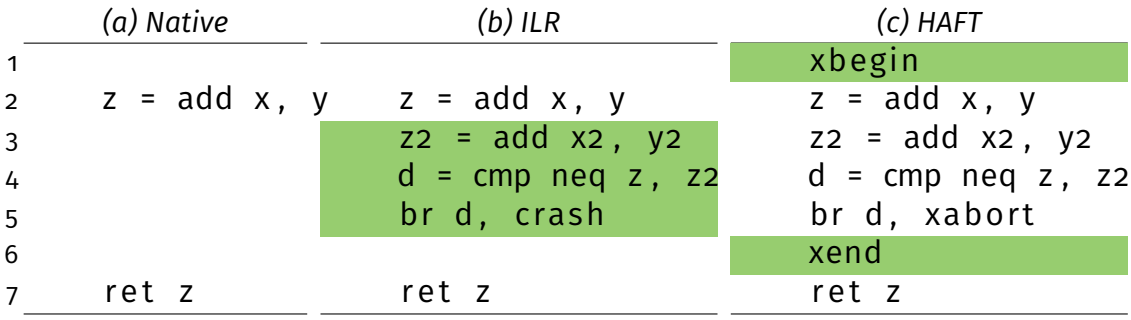


Figure 6.1. Our compiler-based protection mechanism transforms original code: (a) replicating original instructions with ILR for fault detection (b) and covering the code in transactions with Tx for fault recovery (c). Green lines highlight instructions inserted by our compiler using ILR and Tx.

Recently, a few hardening approaches against transient faults have been proposed [9, 15, 23, 33, 35, 42, 43]. However, these approaches still contain at least one of the following limitations: *(1)* do not support mutithreaded and/or nondeterministic applications, *(2)* requiring to modify the source code of applications, *(3)* requiring operating system support, deterministic multi-threading and/or spare core for redundant executions, *(4)* providing only fail-stop semantics without providing recovery from faults.

We overcome these issues by applying Hardware-Assisted Fault Tolerance (HAFT) [22], a compiler-based technique to prevent transient faults at the instruction level. It mitigates the faults by duplicating the instructions in a binary and periodically comparing their execution results. The compiler-based nature of HAFT allows us to harden applications transparently, without modifying the code of the application. HAFT neither enforces deterministic execution nor requires spare cores, and thereby, it does not limit the available application parallelism. Finally, HAFT achieves high availability by providing fault detection as well as recovery from faults.

To achieve fault tolerance, our protection mechanism utilizes HAFT to transform an application as follows (see Figure 6.1). First, we apply Instruction Level Replication (ILR) to the application and we add periodic integrity checks into it. We apply ILR first by replicating all instructions except control flow ones (Figure 6.1b). To detect faults, we use ILR to insert a check before returning the result. A fault is detected if two copies of data differ then an error is reported by enforcing application termination. Next, to provide fault recovery, we cover the whole execution of the application using HTM-based transactions (Figure 6.1c). Whenever a fault is detected by ILR, the transaction is automatically rolled back and re-executed. Our mechanism attempts to rerun aborted transactions for a certain number of times (three times in our current implementation), after that the code is executed non-transitionally until a new transaction is encountered. If a fault is detected during the non-transactional part of the code, the ILR-based mechanism has no choice but terminate the application. Note that the HTM implementation we employed is best-effort which also renders HAFT's recovery guarantees best-effort and it falls back to the fail-stop model in rare cases the bounded-number of re-executions (three times) is reached. However, our exper-

iments show that a clever placement of transactions allows our fault tolerance mechanism to achieve high availability even in the presence of frequent faults (see [22]).

Currently, we integrate HAFT with the security toolchain [7]. Our main challenge is to achieve both security and high availability, even under conditions where transient faults may occur frequently.

HAFT and Intel SGX are two technologies that complete each other: Intel SGX ensures integrity on the memory side while HAFT complements it with preventing bitflips in the CPU. In the memory side, SGX protects the integrity in the processor-reserved memory (PRM) region by conserving an integrity tree for the whole PRM. This tree is updated every time when the processor loads the necessary data for an application inside the enclave. If there is a bitflip in the value retrieved from memory, Memory Encryption Engine (MEE) will detect it and lock the CPU to prevent further corruption. The motivation for this is that all single bitflips in memory are corrected by ECC memory and hence, lock downs should be rare and only happen if the system is under an attack. However, this approach hinders availability in presence of transient faults outside the main memory, such as faults in the CPU.

To prevent the CPU lockdown and to transparently tolerate transient faults, we make HAFT a part of the security toolchain. We take the SCONE libc architecture and capabilities into consideration. First, we have to ensure our version of HAFT compatible towards musl libc which SCONE uses at its core. SCONE uses custom libc, which program will be compiled and linked against. It pools all the system calls from the application and will execute those with its own threading system. The configuration of this threading system, in some cases, may improve the performance. The previous version on HAFT is still relying on some configurations that are only available on glibc. Also, we noticed that although most of LLVM IR (intermediate representation) is backward compatible, the internals (intrinsics, pass functions, etc.) are not. We have rewritten most of the HAFT internals and adjust HAFT passes to satisfy those requirements and limitations. Also, we maintain our version of LLVM to be able to maintain the compatibility later on.

7. Conclusion

This document describes the work undertaken towards D4.3. One main focus during the period was on an initial integration of the dataflow substrates. Accordingly, we have integrated OmpSs with XiTAO, OmpSs with MaxJ and OmpSs with Dfiant. Additionally, we have also integrated OmpSs with Intel SGX security extensions. Our final release will integrate all the dataflow substrates of the project.

On the compiler side, we developed the data parallel interface for XiTAO, released the stable version of the Dfiant language and compiler and finalized the MaxJ compiler toolflow integration. The compiler work on the OmpSs side has been reported in D3.3 since it is tightly coupled to the runtime.

On energy-efficiency effort, we first introduced the XiTAO energy-aware scheduling approach, followed by the work on energy-efficient IoT Data Deduplication. The advance on the HEATS energy scheduler in the period of M20 to M30 was described next. Finally two orthogonal work that focus on security were described, first the energy-efficient aspects of the secure SCONE framework was described. Finally the first integration of OmpSs tasking model with Intel SGX security framework was detailed.

On the fault tolerance side, the integration of the ALYA software framework which powers the smart city use case with the Fault Tolerance Interface (FTI) checkpointing library was described. Then, finally fault tolerance aspects of the secure SCONE framework was described.

8. References

[1] Convolutional Neural Network (CNN) Tensorflow Tutorial. https://www.tensorflow.org/tutorials/images/cnn. Accessed: May, 2020.

[2] Musl Libc. https://www.musl-libc.org/. Accessed: May, 2020.

[3] Powerstat - A Tool To Measure Power Consumption. http://manpages.ubuntu.com/manpages/xenial/man8/powerstat.8.html. Accessed: May, 2020.

[4] The GNU C Library. https://www.gnu.org/software/libc/. Accessed: May, 2020.

[5] Indian dataset for ambient water and energy. http://iawe.github.io/, 2013. Accessedon : 03.02.2020.

[6] Farzad Amirjavid, Petros Spachos, Liang Song, and Konstantinos N Plataniotis. Network coding in internet of things. In *2016 IEEE 21st International Workshop on Computer Aided Modelling and Design of Communication Links and Networks (CAMAD)*, pages 140–145. IEEE, 2016.

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

[8] Nipun Batra, Manoj Gulati, Amarjeet Singh, and Mani B Srivastava. It's Different: Insights into home energy consumption in India. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, pages 1–8, 2013.

[9] Diogo Behrens, Marco Serafini, Sergei Arnautov, Flavio P. Junqueira, and Christof Fetzer. Scalable error isolation for distributed systems. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2015.

[10] Philip A Catherwood, David Steele, Mike Little, Stephen Mccomb, and James McLaughlin. A community-based iot personalized wireless healthcare solution trial. *IEEE journal of translational engineering in health and medicine*, 6:1–13, 2018.

[11] Louis Columbus. Roundup of internet of things: Forecasts and market estimates, 2018.

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

[13] John Rydning David Reinsel, John Gantz. The digitization of the world from edge to core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf, 2018.

[14] L. Peter Deutsch. Deflate compressed data format specification version 1.3. RFC 1951, RFC Editor, May 1996. http://www.rfc-editor.org/rfc/rfc1951.txt.

[15] Björn Döbel and Hermann Härtig. Can we put concurrency back into redundant multithreading? In *Proceedings of the 14th International Conference on Embedded Software (EMSOFT)*, 2014.

[16] Olakunle Elijah, Tharek Abdul Rahman, Igbafe Orikumhi, Chee Yen Leow, and MHD Nour Hindia. An overview of internet of things (IoT) and data analytics in agriculture: Benefits and challenges. *IEEE Internet of Things Journal*, 5(5):3758–3773, 2018.

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

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

[19] Kien A Hua, Ning Jiang, Jason Kuhns, Vaithiyanathan Sundaram, and Cliff Zou. Redundancy control through traffic deduplication. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 10–18. IEEE, 2015.

[20] Abdallah Jarwan, Ayman Sabbah, and Mohamed Ibnkahla. Data transmission reduction schemes in WSNs for efficient IoT systems. *IEEE Journal on Selected Areas in Communications*, 37(6):1307–1324, 2019.

[21] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014.

[22] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. HAFT: hardware-assisted fault tolerance. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[23] Dmitrii Kuvaiskii, Oleksii Oleksenko, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Elzar: Triple Modular Redundancy using Intel AVX. In *proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2016.

[24] Galit Mendelson. All you need to know about Power over Ethernet (PoE) and the IEEE 802.3 af Standard. *Internet Citation,[Online] Jun*, 2004.

[25] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7(4):14, 2012.

[26] Jorge Castiñeira Moreira and Patrick Guy Farrell. *Essentials of error-control coding*. John Wiley & Sons, 2006.

[27] Rosario Morello, Claudio De Capua, Gaetano Fulco, and Subhas Chandra Mukhopadhyay. A smart power meter to monitor energy flow in smart grids: The role of advanced sensing and IoT in the electric grid of the future. *IEEE Sensors Journal*, 17(23):7828–7837, 2017.

[28] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[29] Lars Nielsen, Rasmus Vestergaard, Niloofar Yazdani, Prasad Talasila, Daniel E Lucani, and Márton Sipos. Alexandria: A proof-of-concept implementation and evaluation of generalised data deduplication. In *Globecom. IEEE Conference and Exhibition*, 2019.

[30] Igor Pavlov. 7-zip. https://www.7-zip.org/. Accessed on: 06.09.2019.

[31] Zahra Pooranian, Kang-Cheng Chen, Chia-Mu Yu, and Mauro Conti. RARE: Defeating side channels based on data-deduplication in cloud storage. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 444–449. IEEE, 2018.

[32] Michael Oser Rabin. Fingerprinting by random polynomials. *Technical report*, 1981.

[33] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *proceedings of the 3th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2005.

[34] Shruti Sanadhya, Raghupathy Sivakumar, Kyu-Han Kim, Paul Congdon, Sriram Lakshmanan, and Jatinder Pal Singh. Asymmetric caching: improved network deduplication for mobile devices. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 161–172. ACM, 2012.

[35] A. Shye, T. Moseley, V.J. Reddi, J. Blomstedt, and D.A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007.

[36] Neil T Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. *ACM SIGCOMM Computer Communication Review*, 30(4):87–95, 2000.

[37] Rasmus Vestergaard, Daniel E Lucani, and Qi Zhang. Generalized deduplication: Lossless compression for large amounts of small iot data. In *European Wireless Conference*. IEEE, 2019.

[38] Terry A. Welch. A technique for high-performance data compression. *Computer*, (6):8–19, 1984.

[39] Stephen B Wicker and Vijay K Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[40] Niloofar Yazdani and Daniel E Lucani. Revolving codes: High performance and low overhead network coding. In *2019 IEEE 2nd Wireless Africa Conference (WAC)*, pages 1–5. IEEE, 2019.

[41] Niloofar Yazdani and Daniel Enrique Lucani Rötter. Protocols to reduce cps sensor traffic using smart indexing and edge computing support. In *Ieee Globecom 2019 Workshop on Edge Computing for Cyber Physical Systems*. IEEE, 2019.

[42] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. Runtime asynchronous fault tolerance via speculation. In *proceedings of the 10th International Symposium on Code Generation and Optimization (CGO)*, 2012.

[43] Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. DAFT: Decoupled acyclic fault tolerance. In *proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[44] Bing Zhou and Jiangtao Wen. Efficient file communication via deduplication over networks with manifest feedback. *IEEE Communications Letters*, 18(1):94–97, 2014.

[45] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[46] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.