D4.2 "FIRST RELEASE OF ENERGY-EFFICIENT, SECURE, RESILIENT TASK-BASED PROGRAMMING MODEL AND COMPILER EXTENSIONS"

Version 1

Document Information

Contract Number	780681
Project Website	https://legato-project.eu/
Contractual Deadline	31 July 2019
Dissemination Level	Public
Nature	Report
Author	Marcelo Pasin (UNINE)
Contributors	Isabelly Rocha (UNINE), Christian Göttel (UNINE), Valerio Schiavoni (UNINE), Pascal Felber (UNINE), Rafael Pires (UNINE), Sébastien Vaucher (UNINE), Do Le Quoc (TUD), André Martin (TUD), Konstantinos Parasyris (BSC), Xavier Martorell (BSC), Leonardo Bautista-Gomez (BSC), Miquel Pericàs (CHALMERS), Mustafa Abduljabbar (CHALMERS), Yoav Etsion (TECHNION)
Reviewers	Hans Salomonsson (MIS), Oron Port (TECHNION)

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
865	2019-06-24, Initial draft TOC
881	2019-06-28, Heterogeneity- and energy-aware task scheduling
888	2019-07-02, Added work on TEE
976	2019-07-14, Added performance monitoring section
986	2019-07-15, Added context regarding BSC fault tolerance work
992	2019-07-16, Reviewed fault tolerance section
997	2019-07-16, Added section on Eclipse IDE
1000	2019-07-16, Added Energy-efficient Programming Model
1004	2019-07-16, Filled missing bits in programming model
1006	2019-07-17, Reviewed fault tolerance
1016	2019-07-19, Added content for h-l synthesis for FPGA and DFiant
1026	2019-07-20, Global document review, missing bits
1028	2019-07-21, Reviewed programming model
1035	2019-07-23, Reviewed sections, introductions, balance and structure
1050	2019-07-24, Fixed many typos, checked coherences
1054	2019-07-24, Added executive summary and conclusion
1059	2019-07-26, Merged DFiant and MaxJ into one chapter for dataflow engines
1081	2019-07-31, Addressed all internal review remarks.

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

Index

List of Figures

List of Tables

1. Executive Summary

This deliverable contains the first detailed report on energy-efficient, secure, resilient task-based programming model and compiler extensions. It goes along with the first release of compiler extensions, checkpointing tools and other components. It is issued at the same time as Deliverable 3.2, and includes many components that complement the ones presented there.

The text of this deliverable reports the status of the LEGaTO tool chain frontend (work package 4) at the end of month 20 of the project, and is organized in four main technical chapters. It covers (1) compiler support and development environment, (2) dataflow engines integration, (3) energy efficiency, and (4) fault-tolerance mechanisms.

Chapter 3 details the extensions for the OmpSs compiler and Eclipse integrated development environment. It describes the compiler interface for a fault-tolerant library. It also includes a presentation of the LEGaTO model with implementation details and evaluation. It then presents Eclipse plugins developed in LEGaTO.

Chapter 4 presents the DFiant dataflow hardware description language. DFiant is embedded in Scala and abstracts away registers and clocks, bringing together constructs and semantics from dataflow hardware and modern programming languages. It also presents LEGaTO's work on task-based kernel identification (in OmpSs) and mapping on Maxeler dataflow engine graphs.

Chapter 5 describes LEGaTO's work on programming and execution models integrating CPU and FPGA, energy-efficiency and security trade-offs, and a task-based scheduler. LEGaTO models are presented, including transformations for running combined CPU and GPU applications. It also presents a task-based scheduler for containerised applications and a detailed analysis of several trusted execution environments. The same chapter contains a study on performance and energy when using trusted execution environments.

Chapter 6 presents the LEGaTO components for fault-tolerance and security. The first part describes a performance monitoring framework for trusted execution environments. The second part presents and evaluates a library for fault-tolerant implementation of task-based applications with checkpointing.

2. Introduction

LEGaTO is a Horizon 2020 research and innovation action to develop advanced techniques to make it easier to build large performance-hungry applications. The intention of these techniques is to be able to attain high performance and yet save energy while offering adequate security and fault-tolerance. These requirements compete against each other, so LEGaTO aims at finding adequate trade-offs by using multiple heterogeneous computers that incorporate central processing units (CPUs), field-programmable gate arrays (FPGAs), and graphics processing units (GPUs). To achieve application development under these constraints, we are building a tool chain that maps applications written in a high-

level task-based dataflow language onto these heterogeneous platforms.

The tools developed in WP4 are intended for high productivity, performance, security and fault tolerance. These aspects function as requirements for many components being designed. Each component presents different ways of tackling the these requirements, working at different levels, and often obtaining trade-offs between two or more aspects. Moreover, the components in WP4 are developed in tight integration with the work done in WP3.

This deliverable describes the first release of the front-end system that is being developed in LEGaTO's Work Package 4 (WP4) — Tool-Chain Front-End — to support applications at compilation and runtime. The lower-level runtime and library aspects of this tool chain are covered in the Deliverable D3.2 of Work Package 3 (WP3) — Tool-Chain Back-End. Figure 2.1 highlights the tool-chain front end components, developed in WP4, and gives an overview of the remaining components.

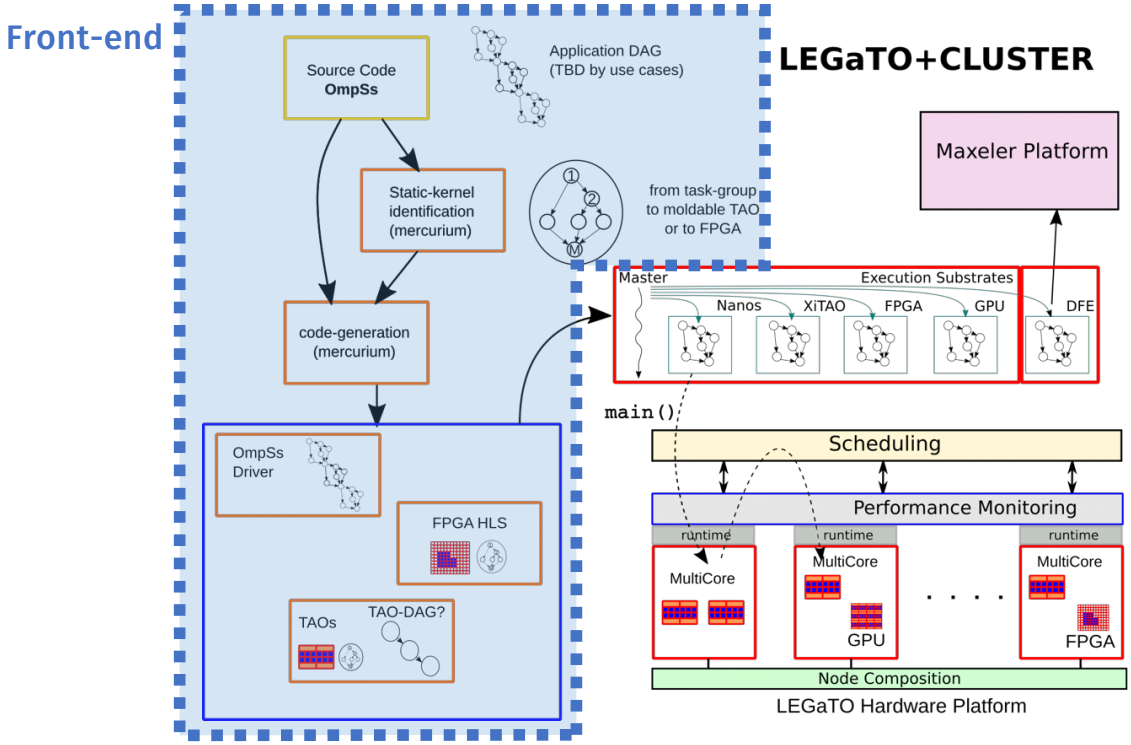


Figure 2.1. Focus on the front-end tool chain components within the entire LEGaTO framework.

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 (M1-9)
- Task 4.2: Programming Model features for energy efficiency (M1-36)
- Task 4.3: IDE plugin (M7-36)
- Task 4.4: Compiler support (M7-30)
- Task 4.5: High-level Synthesis for FPGA (M1-36)

- Task 4.6: Task-based kernel identification/DFE mapping (M7-36)
- Task 4.7: Fault Tolerance and Security (M1-36)

Task 4.1 ended at month 9 and most of its work has been reflected into Chapter 4 of Superdeliverable SD1 [91]. 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. Their first technical and scientific output is presented in this deliverable, as a partially-integrated catalogue of components, implemented in the first 20 months.

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.

Task 4.2, *Programming Model Extensions for Energy Efficiency*, works on defining of the execution model and the programming model for the tool chain, specifying what needs to be supported by the runtime. This task has so far defined the overall software development flow and an execution model. Programming is driven by OmpSs [40], executing tasks (natively or externally generated) in different hardware resources such as CPUs, GPUs and FPGAs. Running on FPGAs requires partitioning the CPU cores between two different runtimes, one for FPGA and another one for CPU. Secion 5.1 in Chapter 5 describes the resource sharing between these runtimes.

Also in Task 4.2, we worked on analysing energy-efficiency and security trade-offs, with multiple types of hardware such as Intel SGX and AMD SEV. This analysis was used in the design of a task-based scheduler called HEATS. Both the analysis and the task-based scheduler are described respectively in Section 5.3 and Section 5.2 in Chapter 5.

Task 4.3, *IDE plugin*, incorporates OpenMP and OmpSs support into Eclipse. We developed plugins to propose directives for parallel tasks and work sharing, as decribed in Section 3.2 of Chapter 3. Consequently, Eclipse is capable of suggesting directives and clauses while the programmer is typing. Our plugins can also automatically invoke the appropriate compiler (including Mercurium for FPGA with autoVivado).

Task 4.4, *Compiler support*, works on extending the OmpSs compiler to support all novel functionalities proposed in LEGaTO. In the first 20 months, we improved the support for autoVivado, to compile OmpSs applications which target Xilinx FPGAs, and extended support for discrete Alpha-Data FPGAs. Additionally, we extended the compiler to support the Fault-Tolerance Interface (FTI) developed in Task 4.7, as presented in Section 3.1 of Chapter 3.

Task 4.5, *High-level Synthesis for FPGA*, comprises the design, implementation and integration of DFiant, a Scala-embedded hardware description language that leverages dataflow semantics to decouple functionality from implemen-

tation constraints. DFiant enables describing hardware by using the dataflow firing rule as a logical construct, combining modern software language features with classic HDL traits. Section 4.1.1 in Chapter 4 is devoted to presenting DFiant and its current implementation and usable feature status.

Task 4.6, *Task-based kernel identification / DFE mapping* intends to identify static sub-graphs for FPGA and DFE mapping on task-based applications. Maxeler and BSC have collaborated to use OmpSs programming model as a front-end for Maxeler's dataflow compute kernel generation. This collaboration is still on-going and is described in Section 4.2 of chapter 4.

Task 4.7, *Fault Tolerance and Security*, extends the front-end tool-chain implementation concerning fault-tolerance and security features. To understand the overhead generated by trusted execution environments, we developed a monitoring framework. The framework, as described in Section 6.1 in Chapter 6, helps to produce meaningful performance metrics and allows to identify the main overhead contributors. We also developed and assessed a checkpoint (and restart) library to be integrated inside the runtime system. This resulted in a multilevel library with features for local checkpointing and data replication, as described in Section 6.2, also in Chapter 6.

3. Compiler Support and Development Environment

This chapter gives technical details of the work done in WP4 concerning extensions in the OmpSs compiler and Eclipse Che Integrated Development Environment (IDE). The results presented in this chapter are outcomes of Tasks 4.3 and 4.4. Section 3.1 details the compiler interface for a fault-tolerant library developed in LEGaTO. It includes a presentation of its model, a number of implementations details, and an evaluation of the interface. Section 3.2 presents the plugin extensions we developed for Eclipse Che in order to offer a well suited IDE for LEGaTO and its programming model. The section includes details of the Eclipse plug-ins implemented.

3.1. Compiler Support for Fault Tolerance

In this section we present source-to-source compiler support that translates *pragma* directives to FTI library API calls. By doing so we provide to the developer a uniform programming paradigm. OMPss task specification is based on *pragma* directives, therefore providing also the fault tolerance support with *pragma* directives reduces the programmers effort, and also decouples the underlying API-calls with simple *pragmas*. The *pragma* support is called *OpenCHK*.

3.1.1. OpenCHK Model

In Figure 3.1 we present the *pragma* directives of a sample code, on the left we present the source code containing FTI direct API calls whereas on the right we present an implementation with *pragma* directives.

The model supports four directives. Some of them also may be annotated with clauses that can modify their semantic in some way. Details on both directives and clauses are provided as follows.

```c
int main(int argc, char *argv[]){
  int rank, nbProcs;
  double *h,*g;
  int i;
  MPI_Init(&argc, &argv);
  FTI_Init(argv[1], MPI_COMM_WORLD);
    int CkptLVL = atoi(argv[2]);
  MPI_Comm_size(FTI_COMM_WORLD, &nbProcs);
  MPI_Comm_rank(FTI_COMM_WORLD, &rank);
  h = (double*) malloc (sizeof(double)*nElements);
  g = (double*) malloc (sizeof(double)*nElements);
  initData(&h,&g);
  FTI_Protect(0, &i, 1, FTI_INTG);
  FTI_Protect(1, h, nElements, FTI_DBLE);
  FTI_Protect(2, g, nElements, FTI_DBLE);
  for(i = 0; i < N; i++){
        if ( (i % 1000) == 999)
        FTI_Checkpoint(i,CkptLVL);
    performComputations(h,g,i);
  }
  FTI_Finalize();
  MPI_Finalize();
}
```

```c
int main(int argc, char *argv[]){
  int rank, nbProcs;
  double *h,*g;
  int i;
    int CkptLVL = atoi(argv[2]);
  MPI_Init(&argc, &argv);
#pragma chk init comm(mpi_communicator)
  MPI_Comm_size(mpi_communicator, &nbProcs);
  MPI_Comm_rank(mpi_communicator, &rank);
  cudaMallocManaged(&h,sizeof(double)*nElements,
      flags);
  cudaMalloc(&g,sizeof(double)*nElements);
  initData(&h,&g);
#pragma chk load (i, h[0:nElements], g[0:nElements])
  for(i = 0; i < N; i++){
#pragma chk store (i, h[0:nElements], g[0:nElements]) \
  if((i%1000) == 999) id(i) level(CkptLVL)
        performComputations(h,g,i);
  }
    #pragma chk shutdown
  MPI_Finalize();
}
```

Figure 3.1. Source code using FTI. FTI API calls and variables are marked as red. On the left side we demonstrate the original FTI and on right side we present the extended FTI with GPU support.

1. `init [clauses]`: The init directive defines the initialization of a checkpoint context. A checkpoint context is necessary to use the other directives. It accepts the clause:
   - `comm(comm-expr)`: comm-expr becomes the MPI communicator that should be used by the user in the checkpoint context that is being created. This clause is mandatory.
2. `load(data-expr-list) [clauses]`: This directive triggers a load of the data expressed inside the parentheses. The load directive accepts the clause:
   - `if(bool-expr)`: The if clause is used as a switch-off mechanism: the load will be ignored if the bool-expr evaluates to false.
3. `store(data-expr-list) [clauses]`: The store directive may request the library to save the specified data. It accepts the clauses:
   - `if(bool-expr)`: The if clause is used as a switch-off mechanism: the store will be ignored if the bool-expr evaluates to false.
   - `id(integer-expr)`: Assigns an identifier to the checkpoint. This clause is mandatory for the store directive.
   - `level(integer-expr)`: Selects the checkpoint level which is associated with where is the data stored (e.g., local node storage, parallel file system, etc.). This clause is mandatory for the store directive. More details about the levels are provided in section 6.2.2.
   - `kind(kind-expr)`: Selects the checkpoint kind. Currently, two kinds are supported. They are CHK_FULL, which performs a full checkpoint; and CHK_DIFF, which performs a differential checkpoint. Differential checkpoint is thoroughly described in Deliverable D3.2.
4. `shutdown`: Closes a checkpoint context.

3.1.2. Implementation Details

We provide our own implementation of the model on top of the Mercurium C/C++ and Fortran source-to-source compiler [30] and the Transparent Checkpoint Library (TCL) [29] intermediate library.

We have designed an implementation based on three components:

1. The compiler (Mercurium) that translates directives and clauses into calls to an intermediate library. The OpenCHK directives and clauses are translated to application-level checkpoint/restart functionalities. To decouple the compilation procedure with the actual implementation of the checkpoint library, Mercurium does not translates directly the *pragma* directives to FTI calls, but it translates the directives to an intermediate library called TCL.

2. An intermediate library (TCL) which is in charge of forwarding the user-requested actions to the adequate back-end library; The user is agnostic of the back-end library, but TCL must send the correct information in the proper way to each back-end library. As each back-end library implements a different interface, TCL must format the information in the way each back-end library expects it, as well as calling the appropriate methods to perform those actions required by the user.

3. Several back-end libraries. For the *Legato* project, we focus on the FTI back-end checkpoint library.

Figure 3.3 shows our three-layer architecture. We would like to highlight that this approach allows us extending the model to support new features if the back-end libraries evolve.

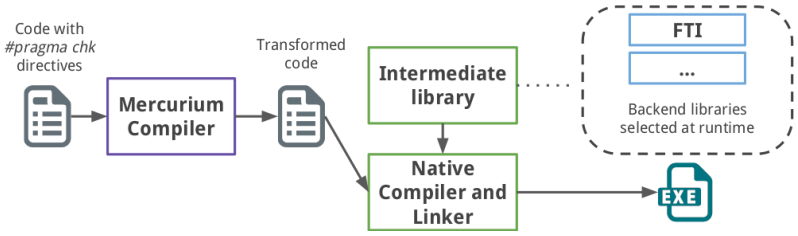


Figure 3.2. Diagram of the three-layer architecture.

3.1.3. Evaluation of the Fault-Tolerance Interface

This section details the benchmark of our approach when compared with a direct FTI invocation. Our evaluation focuses on two aspects: i) Programmability of the *pragma* directives. This is done by comparing the lines of code (LOCs) between the two approaches. ii) Overhead of our approach. The results are obtained by averaging the execution times of 5 different runs for each version. We demonstrate that our compiler assisted approach adds no additional overhead when compared to the direct use of the native back-end library.

The evaluation executions are conducted with 50 MPI processes, when possible. Some applications present some constraints on the number of MPI processes

that can be used. For those applications, we will specify the number of processes used. We do not perform bigger experiments because nothing suggests that more processes will introduce additional overhead than those introduced by the back-end libraries. Depending on whether or not the original version of the benchmark/application has intra-node parallelism, the number of threads per process varies. Those having intra-node parallelism were executed with 48 threads per process, while the others were executed with 1 thread per process.

With respect to the execution time, all the runs took about 10 minutes. The checkpoint frequency is 1 checkpoint per minute, so that we perform 10 checkpoints per run. This frequency is expressed in terms of iterations. Hence, we carry out a checkpoint every 10% of the iterations. We have selected a high checkpoint frequency to stress the checkpointing mechanisms and facilitate the performance comparison between the different approaches evaluated. Coarser checkpoint frequencies should result in even lower overheads.

With regard to the faults, for the evaluation part, all the faults are deterministically injected when the application has already done 90% of the work. The faults introduced are exceptions that cause the abortion of the process.

We provide a brief explanation of seven applications and benchmarks used in this evaluation. The size of the applications range from $\approx$ 500 to $\approx$ 15000 lines of code. **Duct** [36]: A CFD application performing a Large Eddy Simulation (LES) of turbulent flow in square. This application is pure MPI. **Heat**: Performs a heat 2D transfer simulation. It is pure MPI. **LULESH2.0** [58]: C++ OpenMP+MPI sample application from Lawrence Livermore National Laboratory that models the propagation of a Sedov blast wave. The problem is formulated using a 3-dimensional unstructured mesh. **N-Body**: The classic N-body simulation of a dynamical system of particles. This benchmark uses MPI+OpenMP. Due to restrictions of the implementation, it must be run only with 32 nodes. **SPECFEM3D**: A simulation of a seismic wave propagation using a Galerkin spectral element method. Implemented with MPI+OpenMP. Due to restrictions of the implementation, it must be run only with 32 nodes. **TurboRVB** [37]: Developed at SISSA, this application is used to understand high-temperature superconductivity by means of Quantum MonteCarlo simulations. **xPic**: C++ OpenMP+MPI HPC application deduced from iPic3D [71]. It is designed for large scale production runs. xPic simulates space plasma in a 3-dimensional parallel code.

In Table 3.1 we present the results of our evaluation, in terms of both LOCs and overhead in comparison with a direct FTI implementation of the different applications. Evidently, the *pragma*-based implementation does not show any significant execution time overhead in comparison with the original FTI implementation. But more importantly, the LOCs required to checkpoint restart the different applications is heavily reduced when using the *pragma* directives. Typically, the implementation of a checkpoint/restart scheme for an application requires 71% less lines of code when using the *pragmas* in comparison with directly calling the FTI API functions.

	FTI LOCs	OpenCHK LOCs	OpenCHK/FTI	Exec. Time Overhead
DUCT	31	5	0.1613	0.9977
HEAT	15	5	0.3333	0.9924
LULESH	12	5	0.4167	1.0082
NBODY	25	5	0.2	0.9987
SPECFEM3D	28	6	0.2143	0.9968
TURBORVB	80	6	0.075	1.0163
XPIC	8	5	0.625	0.9937
AVERAGE			0.2894	1.0005

Table 3.1. LOCs and overhead required to perform application-level checkpoint/restart using FTI and OpenCHK. Performance does not change much but the LOCs are reduced significantly.

3.2. Integrated Development Environment

We have developed an Integrated Development Environment for OmpSs, based on Eclipse (https://www.eclipse.org/). The approach includes two main components: an Eclipse plug-in for OmpSs and OpenMP directives auto-completion and user assistance; and a Docker container for Eclipse Che, providing the OmpSs tool chain for easy installation and distribution.

3.2.1. Docker container for OmpSs tool chain

A ready-to-use Docker image with all the configured tool chain has been prepared to allow the Eclipse Che stack creation. This image can be used standalone, allowing a quick hands-on evaluation of the tools, rather than installing them. The generated image contains the following third-party software:

1. Ubuntu;
2. papi;
3. boost;
4. arm32/64 gcc crosscompiler.

The image also includes the following BSC Tools:

1. autoVivado;
2. extrae;
3. paraver;
4. mcxx;
5. nanox;

6. OmpSs@fpga kernel module;
7. xdma;
8. xtasks.

3.2.2. Eclipse Che

The web-native IDEs are quickly gaining importance and may change the way we work, providing some benefits over traditional IDEs. A web IDE can be serviced from a remote server or locally, and can be accessed with any available modern web browser. As it is serviced using a website, one can seamlessly change the device used as server. The code doesn't have to leave the IDE workspace, making backups to all projects easy. The compilation and execution, if serviced by a server, is done remotely.

Eclipse Che offers iself a number of new useful features. In one unified environment, programmers can use different machine configurations for different projects. One can use preconfigured stacks for hassle-free setups and personal containers with root access. Workspaces can work in single- or multi-user mode, and it is easy to create collaborative workspaces and share configurations with co-workers. While these features are desirable, an important part of an IDE is all the customization and tools it gives the programmer to write code or analyze it. For this, Eclipse adapted the Theia IDE, a complete IDE based on Monaco Editor (the one that powers Visual Studio Code) to run as the default editor on Eclipse Che. With this, Eclipse Che has a mature, widely-used full-featured IDE with compatibility to hundreds of extensions thanks to being able to use VSCode plugins.

Alongside the benefits above, web-native IDEs usually have some disadvantages, like not being able to use tools the developers are used to, or having a significantly less powerful IDE compared to traditional ones. Eclipe Che makes an important step forward bringing all features necessary to prove itself useful. Che defines the concept of "stack" as a configuration that contains compilers, runtimes, tools and any configuration or application that may be useful for a development goal. This concept fits well in the OmpSs programming model, making possible the creation of a stack that has Mercurium and Nanos installed, alongside all the support tools like extrae, and third party vendor tools like Vivado.

After importing the necessary configurations to Che, the next logical step is to create a Workspace. In Che, a workspace is an Environment where the projects we are going to develop will live in. All it takes to create a workspace is a stack configuration, a name and some extra configurations, such as for instance how much RAM we want it to allocate. During the Workspace creation, a new container powered by Docker will be created, following the rules of the stack. Meaning that while we work in that workspace, all the changes we make or any application we run will be stored and executed from that container.

3.2.3. Single-User Eclipse Che deployment for OmpSs@fpga

To enable creating a user-friendly local installation of Eclipse Che with the Docker image and stack configuration ready to use OmpSs tools, a python script to automate the process has been created. Ideally, the user should have the docker

image imported into its docker daemon, but the script manages the creation of the image in case it is not available.

Since the tool flow relies on VIVADO to generate the bitstream, the script checks for all Vivado installations in the user-defined path. The plugin will execute Eclipse Che using the single-user-mode configuration, exposing all the vivado installations to the IDE. After the Che initialization, the script will create a stack per Vivado Installation and per architecture, which will be dynamically loaded using the Eclipse Che's API.

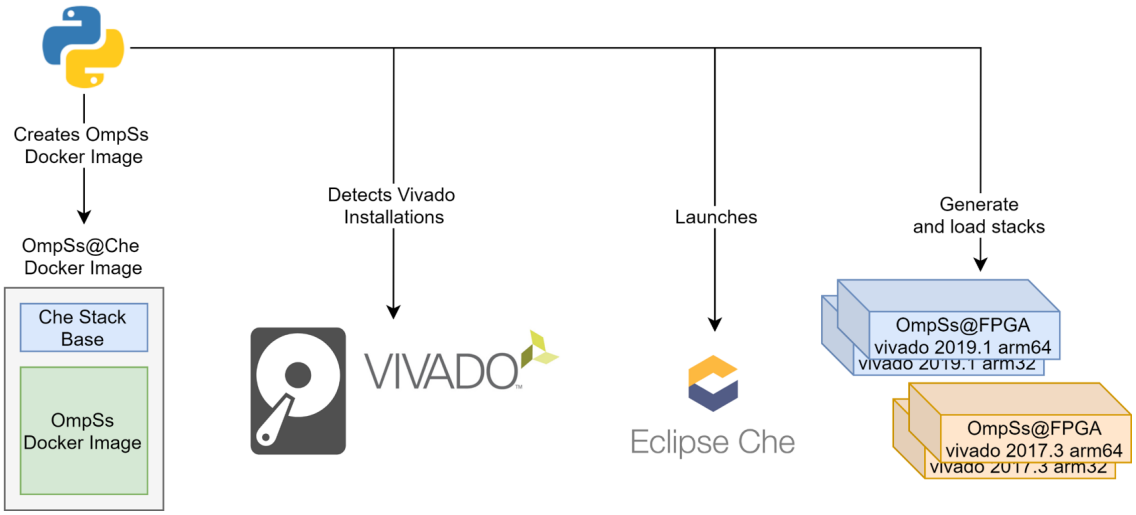


Figure 3.3. Diagram of Eclipse Che Deployment.

3.2.4. Workspace creation for OmpSs@fpga

Once the stacks are loaded into the IDE, we can create workspaces that use these stacks. As shown in Figure 3.4, all workspaces run on their own container, but all share the same Vivado installation folder. Once the workspace is created, the user can verify the installation by running the included Example project embedded in the stack, or begin developing an application.

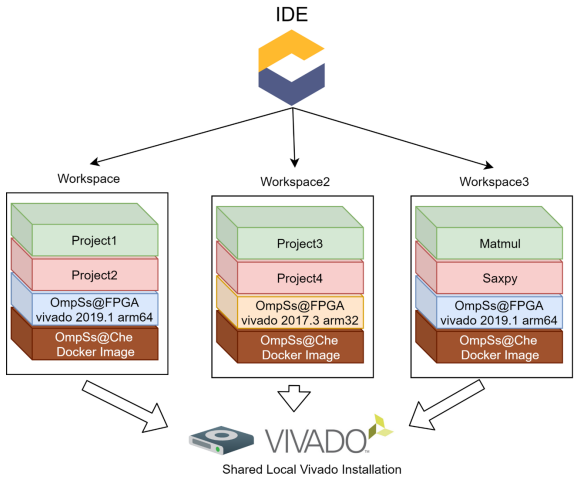


Figure 3.4. Workspaces with its projects on Eclipse Che.

3.2.5. Eclipse Plug-in for OmpSs and OpenMP

The OmpSs and OpenMP Eclipse plug-ins improve programmers productivity by providing hints on annotating C/C++ code, during the development process. For each OmpSs and OpenMP directive and clause, the plug-in implements auto-completion and a small explanations to guide the user.

3.2.5.1. Eclipse legacy

An auto-completion plugin for Eclipse IDE has been developed. It has support for the majority of OpenMP and OmpSs clauses. It displays a little help message to each clause and lets the programmer browse the available annotations for the current context (see Figure 3.5).

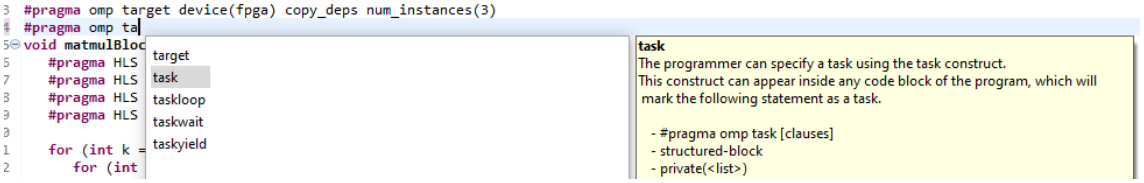


Figure 3.5. Eclipse IDE auto-completion plugin.

3.2.5.2. Theia Plugin for Che and VSCode

An auto-completion plugin for Tehia for Che, that is compatible with VSCode, has been developed, it has support for the majority of OpenMP and OmpSs Clauses, and it adds a little help message to each clause and lets the programmer browse the available annotations for the current context (see Figure 3.6).

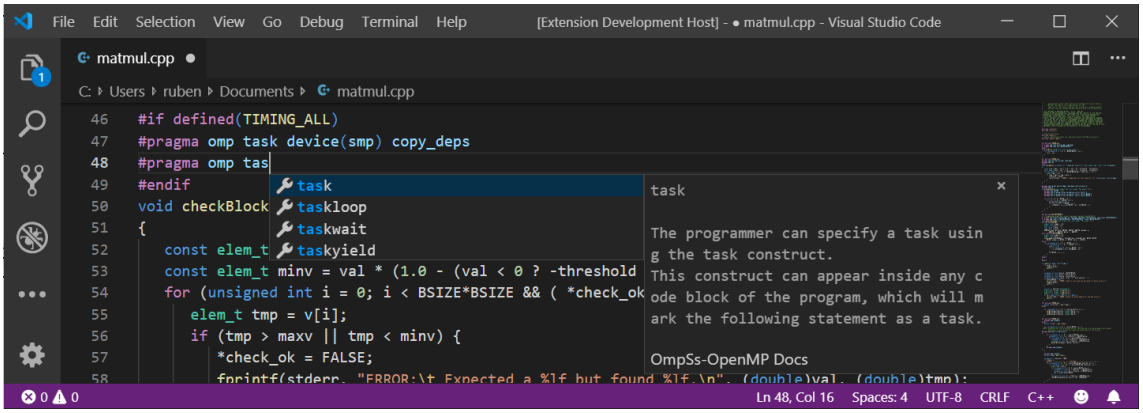


Figure 3.6. Eclipse che + Theia IDE and VSCode auto-completion plugin..

4. Dataflow Engines Integration

This chapter presents the results of the work done in two tasks: Task 4.5 (High-level Synthesis for FPGA and Task) and Task 4.6 (Task-based kernel identification / DFE mapping). The chapter is split into two parts, respectively covering the work done in both tasks.

In the first part in this chapter, the reader will find the design, implementation and integration of DFiant, a hardware description language that leverages

dataflow semantics to decouple functionality from implementation constraints. As presented below, DFiant enables describing hardware by using the dataflow firing rule as a logical construct, combining modern software language features with classic HDL traits. DFiant is embedded in Scala and abstracts away registers and clocks, bringing together constructs and semantics from dataflow hardware and modern programming languages.

We present in the second part of the chapter our work on task-based kernel identification and mapping on Maxeler dataflow engine graphs. To that goal, we use the OmpSs programming model as a front end to the generation of Maxeler dataflow graphs. At the end of the chapter, we present how to identify static sub-graphs in OmpSs task graphs and then map them onto dataflow kernels in the MaxJ language.

4.1. High-level Synthesis for FPGA

The register-transfer level (RTL) programming model paved the road for Verilog and VHDL to flourish as the leading hardware description languages (HDLs). That road, however, is steadily nearing its end as both hardware designs and devices become increasingly more complex. While the software world is striving for a "write once, run anywhere" programmability, the complexity of an RTL design implementing a given functionality may vary greatly across different FPGA and ASIC devices that incorporate various technologies and core components. Moreover, minor requirement changes may lead to significant redesigns, since RTL abstraction tightly couples functionality with timing constraints. For example, registers serve various roles such as preserving a state, pipelining and balancing a data path, deriving timed signals from an input clock, and synchronizing an input signal. This coupling between functionality, timing constraints, and device constraints leads to verbose and unportable RTL designs.

In this chapter we present our work on the DFiant dataflow-based HDL that abstract away registers and clocks. DFiant is an open-source, Scala-embedded HDL that utilizes these dataflow constructs to decouple functionality from implementation constraints. DFiant brings together constructs and semantics from dataflow [27, 52, 60, 89], hardware, and software programming languages to enable truly portable and composable hardware designs. The dataflow model offers implicit concurrency between independent paths while freeing the designer from explicit register placement that binds the design to fixed pipelined paths and timing constraints.

4.1.1. The DFiant Language Overview

DFiant is a Scala library and thus possesses various rich type safe language constructs. DFiant also incorporates unique language semantics that enable dataflow-based hardware description. Throughout this section we elaborate on these constructs and semantics via our running example, a four-by-four moving average (MA4) unit. The MA4 has four 16-bit integer input channels and is required to output the average of all channels, while each channel is averaged by a four-sample moving window continuously. The complete MA4 DFiant implementation and its equivalent dataflow graph are available in Fig. 4.2 and 4.4, respectively. Fig. 4.3 presents a subset of the DFiant-generated VHDL (2008) code

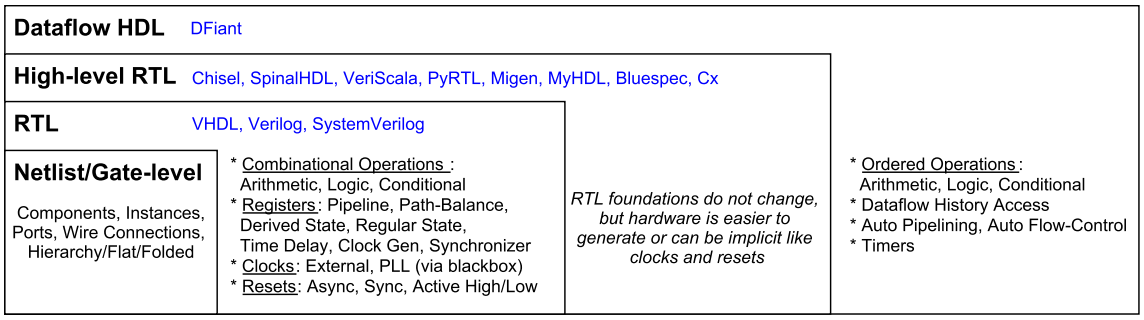


Figure 4.1. HDL abstraction layer summary (lowest=netlist, highest=dataflow). Each layer subsumes the capabilities of the layer below it. Dataflow constructs replace RTL registers with their true functionality (e.g., state) or inserts them implicitly (e.g., pipelining).

```
1  import DFiant._
2
3  trait MovingAvg4x4 extends DFDesign {
4    val a    = DFSInt[16] <> IN init 0
5    val b    = DFSInt[16] <> IN init 0
6    val c    = DFSInt[16] <> IN init 0
7    val d    = DFSInt[16] <> IN init 0
8    val o    = DFSInt[16] <> OUT
9
10   def ma(src : DFSInt[16]) = {
11     val acc = DFSInt[18] init 0
12     acc := acc - src.prev(4) + src
13     (acc / 4).toWidth(16)
14   }
15   def avg2(src1 : DFSInt[16], src2 : DFSInt[16]) =
16     ((src1 + src2).wc / 2).toWidth(16)
17     //  (_ + _).wc is a with-carry addition
18
19   o := avg2(
20     avg2(ma(a), ma(b)), avg2(ma(c), ma(d))
21   )
22 }
23
24 object MA4App extends App {
25   val ma4 = new MovingAvg4x4 {}
26   ma4.compileToVHDL.toFile("ma4.vhd")
27 }
```

Figure 4.2. The MA4 DFiant code.

derived from lines 11-12 in Fig. 4.2.

4.1.1.1. Hello DFiant world!

The DFiant code in Fig. 4.2 demonstrates the structure of any DFiant program: it imports the `DFiant` library (line 1), creates the top-level design by extending the `DFDesign` trait (lines 3–21), creates a runnable application that instantiates the top design trait, and compiles it into a VHDL file (lines 24–27).

The MA4 design is fairly straightforward. Lines 4-8 generate the signed dataflow ports and include a `0` value initialization (see Section 4.1.3). Lines 10-14 define the function `ma` that generates a single four-sample moving average, while lines 15-16 define the function `avg2` that generates a two-input average unit. Finally, lines 18-20 compose `avg2` and `ma` to generate the entire MA4 functionality and assign it to the output port `o`. We elaborate on the unique DFiant constructs and semantics in the next sections.

```
...
signal acc        : signed(17 downto 0);
signal acc_prev1  : signed(17 downto 0);
signal src_prev1  : signed(15 downto 0);
signal src_prev2  : signed(15 downto 0);
signal src_prev3  : signed(15 downto 0);
signal src_prev4  : signed(15 downto 0);
...
sync_proc : process (CLK, RSTn)
begin
  if RSTn = '0' then
    acc_prev1     <= 18d"0";
    src_prev1     <= 16d"0";
    src_prev2     <= 16d"0";
    src_prev3     <= 16d"0";
    src_prev4     <= 16d"0";
  elsif rising_edge(CLK) then
    acc_prev1     <= acc;
    src_prev1     <= src;
    src_prev2     <= src_prev1;
    src_prev3     <= src_prev2;
    src_prev4     <= src_prev3;
  end if;
end process sync_proc;
...
async_proc : process (all)
  variable v_acc  : signed(17 downto 0);
begin
  v_acc           := acc_prev1;
  v_acc           := v_acc - src_prev4 + src;
  acc             <= v_acc;
end process async_proc;
```

Figure 4.3. The MA4 DFiant lines 11-12 compiled to VHDL. Note that the DFiant code is extremely compact in comparison.

4.1.2. Dataflow Semantics

DFiant code is expressed in a sequential manner yet employs an asynchronous dataflow programming model to enable an intuitive concurrent hardware description. For this purpose, DFiant applies the following rules:

- *Concurrency and Execution Order* Concurrency is implicit and the data scheduling order, or *token-flow*, is set by the *data dependency*. DFiant schedules all independent dataflow expressions concurrently, while dependent operations are synthesized into a guarded FIFO-styled pipeline. The MA4 dataflow graph in Fig. 4.4 demonstrates the concurrent paths constructed from the dataflow dependency.

- *Basic Operations* Each application of an arithmetic/logic operator is translated into the appropriate hardware construction and applies a dataflow *join* on their arguments. The arguments require a valid token for consumption to produce a new token generated from the operations. For example, `+` in `avg2` joins `src1` and `src2` and requires a token from both to produce the token `src1 + src2`.

- *Path Divergence* Diverging paths are implicitly *forked*, so token production is possible if all target nodes are ready to consume the token. For example, `acc` result in `ma` is forked into a division operation and the state feedback. It is impossible to consume an invalid token and once a token is consumed it is invalidated.

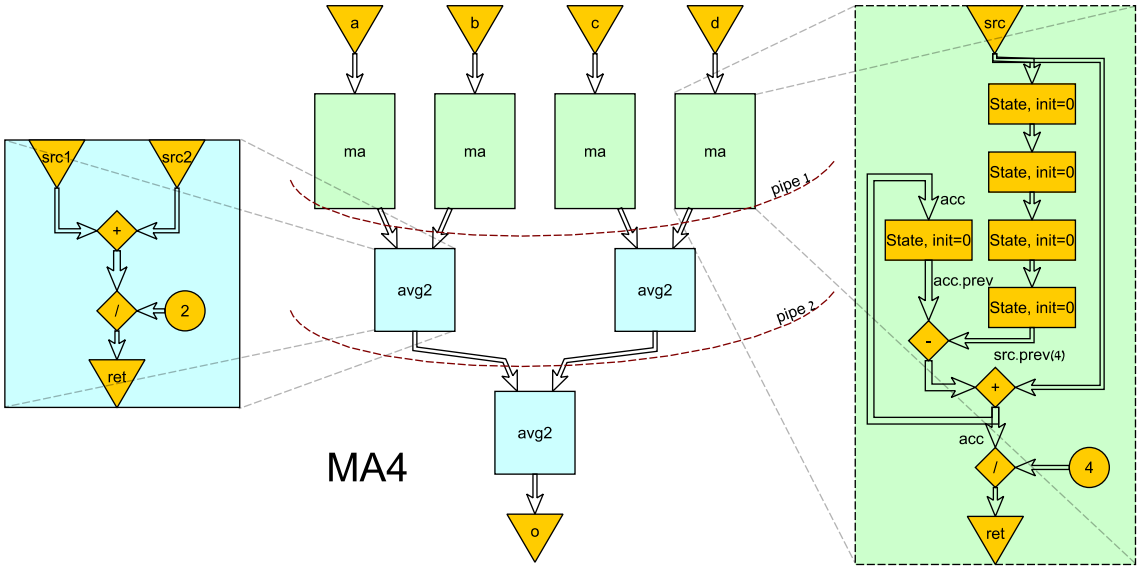


Figure 4.4. The MA4 dataflow graph (the inputs are `a`, `b`, `c`, `d` *and the output is* `o`*). The entire design is a composition of the* `ma` *and* `avg` *functions (detailed blowouts are depicted as well). The compiler places pipeline tags to achieve the required performance and the backend inserts registers accordingly. The concurrent construction is implied from a sequential composition thanks to the dataflow abstraction.*

- *Constants* Any Scala primitive value is considered as a constant when applied as an argument to a dataflow operation. For example, the value `2` in `avg2` is a primitive `Int` and is considered a constant in the division operation. Semantically, a constant is an infinite token generator that produces a new token with the same initial value each time the token is consumed.
- *Pruning* Unused nodes always consume tokens and are discarded during compilation.

4.1.3. State Constructs and Semantics

In contrary to RTL languages, DFiant does not directly expose register and wire constructs. Instead, DFiant assumes every dataflow variable is a stream and provides constructs to initialize the token history via the `.init` construct, reuse tokens via the `.prev` construct, and update the state via the `:=` construct. Lines 11-12 in Fig. 4.2 along with their compiled VHDL representation in Fig. 4.3 demonstrate the state semantics as follows:

- *Initialization* The `.init` construct is accompanied by one or more token values and only sets the initial state history. For example, line 11 constructs a dataflow variable and initializes all of its history as zero value tokens.
- *History Access* The `.prev` construct reuses the previous state token. The very first reused token is the one set via `.init`. It is also possible to call `.prev(step)` with a step number argument to reuse older stream values. For example, in line 12 we reuse a `src` token from four steps ago. If the `src` token stream is "$1, 2, 3, 4, ...$" with a 0 initialization, then the `src.prev(4)` token stream is "$0, 0, 0, 0, 1, 2, 3, ...$".

- *Stall Bubbles* Invoking `.prev` on an uninitialized dataflow variable generates a stall bubble. Stall bubbles are consumed and produced like any other token, yet a basic operation with a stall bubble token must produce a stall bubble token. Additionally, stall bubbles do not affect regular state. The backend compiler is responsible to generate the additional logic required for existing design stalls.

- *State Update Scheduling* The new updated token is pushed into a dataflow stream by using the `:=` assignment construct. There can be more then one assignment to same variable, however only the last assignment updates the state and occurs when all dependent dataflow firing rules are satisfied. This rule is similar to signal update semantics in VHDL processes.

- *Default Self-Generation* Any dataflow `a` variable has an implicit self-assignment `a := a.prev` that comes immediately after the variable construction. This creates an equivalent reference between `a` and `a.prev` which leads to a more intuitive programming. For example, in line 12 we used `acc - ...` and not `acc.prev - ...`, since both expressions are equivalent.

Fig. 4.3 emphasizes the advantages of DFiant state constructs over RTL registers and wires. One advantage is that the DFiant code resembles its RTL counterparts, but is also very concise since state elements are automatically constructed when a stream history is accessed. Another advantage is portability, because state elements are not registers and therefore any type of state component is applicable. Our first synchronous backend indeed maps state elements to registers, but even an asynchronous backend can compile the same code and apply the Muller C-element [78] as a state element.

4.1.4. Automatic Pipelining, Path-Balancing and Flow-Control

The dataflow abstraction enables designers to describe hardware without explicitly pipelining the design. The DFiant backend compiler automatically pipelines the design and places registers to split long combinational paths. The compiler has a propagation delay (PD) estimation database that can be tailored for any target device and technology. With this information and a target clock constraint the compiler tags the dataflow graph with the additional pipe stages required before producing the RTL code. One possible tagging is depicted in Fig. 4.4, in which two pipe stages were added between the large operations. Depending on the availability of DSP blocks in the target device, it is also possible to break the basic operations to multiple cycles by instantiating the proper vendor IP (e.g., a long multiplication operation should require several cycles). All of these target-specific adaptations are done without designer intervention and thus make any DFiant design highly portable.

To maintain design correctness the compiler adds path-balancing registers when pipeline registers are added and different-latency paths converge. Since these two features are separate, we can allow designers to explicitly place pipe stages in critical junctions should our PD estimation fail. The `.pipe` construct adds a pipe stage at a specific node and the compiler will balance the rest of the converging paths. While both `.pipe` and `.prev` constructs appear similar,

the `.prev` construct does affect the path-balancing mechanism. For example, `x - x.prev` creates a derivation circuit while `x - x.pipe` manifests as a constant zero since path-balancing applied at the subtraction input arguments results in a `x.pipe - x.pipe` operation.

The `ma` function creates the regular state referenced via the `acc` variable. The `ma` blowout in Fig. 4.4 exposes this problem by having a circular feedback that updates the `acc` state. This feedback cannot be pipelined as-is because path-balancing will never be able to satisfy the balancing rule due to circular path dependency. It is only possible to increase the clock rate in feedback circuitry by applying multi-cycle or speculative logic (e.g., a RISC-V processor core contains several feedback junctions like the PC update and therefore has single-clock, multi-cycle, and speculation-based pipelined implementations).

4.2. Task-based kernel identification and DFE mapping

MaxJ is Java-based meta-language that describes dataflow based on Maxeler dataflow extensions. The language uses Java syntax but does not generate Java byte code, instead it is used to describe static dataflow compute structures that are mapped to an FPGA through MaxCompiler. MaxCompiler controls the dataflow graph generation, shedules and optimises the graph, and maps it to an FPGA netlist which is then passed on to vendor-specific FPGA backend tools that handle placement, routing and bitstream generation. The fundamental principles of the MaxJ language and the compiler infrastructure were introduced in Deliverable D2.1.

The design process of accelerating an application through offloading the compute intensive parts to Maxeler's FPGA-based Dataflow Engines (DFEs) typically starts with analysing an existing CPU application in C, C++ or Fortran. This application is profiled for its computational hotspots. Next, the developer creates a performance model for the dataflow implementation, taking into account computation and communication, and optimised the projected performance based on the high-level model. The goal is to achieve a high degree of pipelining and parallelism within the dataflow model. Once the dataflow architecture is projected to be of good performance, it is then implemented in the MaxJ language. Here, computation is described in MaxJ kernels and a manager, also developed in MaxJ, facilitates the integration with the original host code. The MaxJ manager describes the interface to DFE kernels and during the compile process it automatically generates the API function calls for the host code. This API called Simple Live CPU Interface, or SLiC in short, and the developer will add SliC calls into the host code to offload computations to the DFE. Overall, this development process requires a fair amount of expertise and developer effort.

In the context of Task T4.6, we use the OmpSs programming model as a front end to the generation of Maxeler dataflow graphs. The rational is that OmpSs tasks seem naturally suitable for FPGA dataflow processing. OmpSs tasks have clearly defined inputs and outputs, and have self-contained state which is an important requirement for the dataflow model. Furthermore, it is necessary to generate static dataflow graphs, since branching and context switching causes significant overhead on FPGAs. We therefore focus on identifying static sub-

graphs in OmpSs task graphs and mapping them to dataflow kernels in the MaxJ language which then generate the FPGA configurations. This process is illustrated in Figure 4.5.

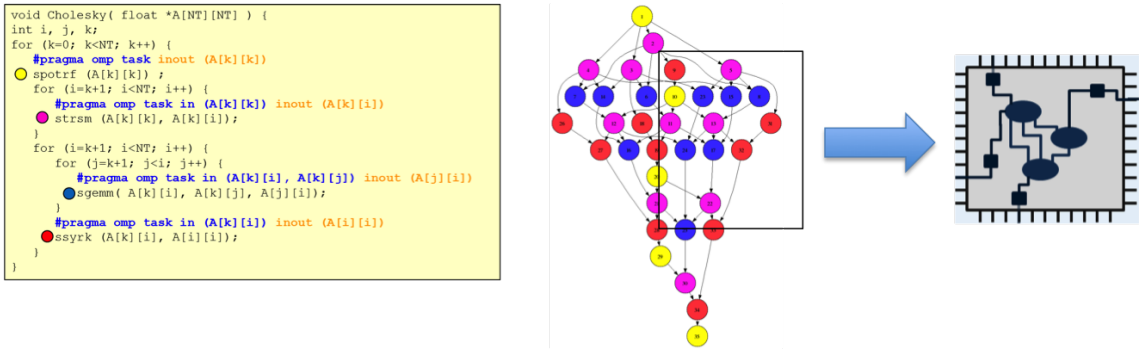


Figure 4.5. Mapping static sub-graphs from OmpSs to a dataflow implementation.

Current work focuses on the practical aspects of integrating OmpSs and MaxJ. To explore various approaches for OmpSs and MaxJ integration, two practical code examples are used: a tiled matrix multiply example and a convolutional neural network (VGG16) that already exist in either OmpSs or MaxJ form. When starting from OmpSs, an important consideration is to express and collect all the information that is necessary for a MaxJ dataflow kernel in OmpSs. As mentioned above, the integration of MaxJ kernels is facilitated via the SLiC API. This API provides a so calles basic static interface to execute computations on the DFE in one function call. This interface is a simple C function with parameters for all incoming and outgoing streams, scalar parameters and the number of cycles each compute kernel has to execute. An example is given below:

```
void movingAverage(int *x_in, int *y_out, int length);
```

The simple function movingAverage uses an input and output array as arguments as well as a value to indicate the length of the arrays. This information is necessary to control the number of execution cycles of the dataflow kernel that performs the movingAverage computation. Since the OmpSs model currently does not directly expose the array length of task inputs and output, our work focuses on adding this information to the OmpSs annotations, e.g. an additional pragma for exection cycles which can then be propagated to MaxJ. Here it is important to note that this information cannot be obtained dynamically as it is needed for the compile-time interface generation. Further options that are being explored involve SLiC interfaces without explicit execution cycles; however as this may limit the use in more complex scenarions, investigations of this are currently ongoing.

5. Energy Efficiency

The components developed in Task 4.2 are described in this chapter. It includes a description of LEGaTO programming and execution models, the integration of its CPU and FPGA runtimes, an analysis of energy-efficiency and security trade-offs, and the design of a task-based scheduler. Section 5.1 presents LEGaTO models

and their transformations for running combined CPU and GPU applications by appropriately sharing resources between Nanos and XiTAO (CPU and GPU runtimes, respectively). Section 5.2 presents a task-based scheduler that uses machine learning to predict performance and energy consumption of containerised applications. Section 5.3 contains a detailed analysis of several trusted execution environments. In this chapter's final section, we study performance and energy implications of using Intel SGX, ARM TrustZone, and AMD SEV.

5.1. LEGaTO Energy-efficient Programming Model

This section describes the status of the LEGaTO programming model and the execution model. The main concern is the description of the applications in a high level language, and the flow of transformations that need to be implemented in order to go from the high level LEGaTO program written in the OmpSs language down to the LEGaTO binary consisting of program code and kernels targeting various execution substrates, including Nanos and XiTAO (for CPUs), FPGAs and GPUs.

5.1.1. The LEGaTO Execution Model

The overall transformation flow of the LEGaTO tool chain is shown in Figure 5.1. This figure shows the transformations from the high level LEGaTO application written in the OmpSs language down to the execution binary.

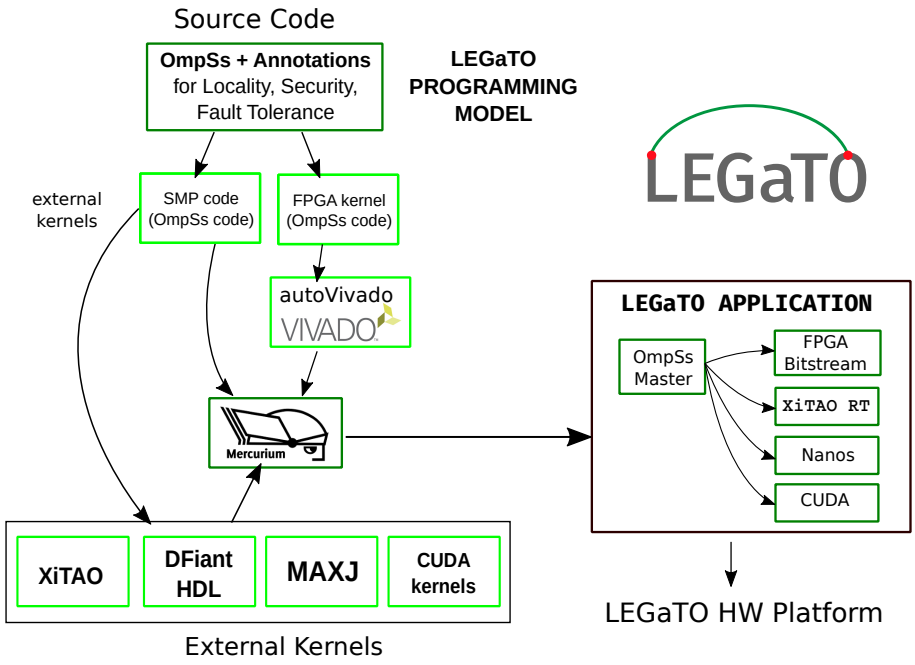


Figure 5.1. The LEGaTO tool chain and execution model

A LEGaTO application is written in C/C++ using the OmpSs programming model. OmpSs applications execute in an implicit parallel region. Initially a single thread, called the master thread, is started. The master thread can generate new threads or tasks, and it can specify the execution of kernels on top of accelerators or CPU threads. New tasks are generated using the `oss task` construct, as in the snippet below.

```
#pragma oss task in(...) out(...)
```

The clauses `in(..)` and `out(..)` specify the input and output dependencies (memory objects) that need to be available in order to mark a task as ready. A task may also not contain any input nor output dependencies, in which case it becomes ready as soon as it is instantiated by the parent thread (either the master thread, or a task thread).

The programmer can also specify that a task is to be offloaded to an accelerator. This is done by specifying the `target` directive.

```
#pragma oss task target(device)
```

This pragma indicates that the following task is to be executed on a device such as a GPU or an FPGA. It is often the case that multiple implementations exist for a particular task, targeting different devices such as CPUs, FPGAs or GPUs. In this case, the programmer can specify that different codes represent different implementations of the same function by using the `implements` clause. For example, the following code indicates that function `cuda_foo()` is an implementation of function `foo()` targeting the `cuda` device type.

```
#pragma oss task target(cuda) implements(foo)
cuda_foo();
```

The LEGaTO tool chain supports code generation directly from OmpSs source code down to CPU cores (smp target) and FPGA by using the autoVivado tool developed within the context of the LEGaTO project (see Deliverable D3.2). The remaining targets such as CUDA and the XiTAO RT execute kernels that are provided externally. CUDA kernels rely on the Nvidia compilation infrastructure, while XiTAO kernels are written using the XiTAO C++ interface and compiled using a C++ compiler such as GCC or LLVM. LEGaTO also provides two alternatives to specify externally provided FPGA kernels. These are the DFiant HDL and the MaxJ language which targets the Maxeler hardware platform.

The overall idea of the LEGaTO programming model is that OmpSs-level tasks are identified by a master thread and then offloaded onto the computational substrate which is managed by a variety of runtimes that operate in a coordinated manner. Within the context of LEGaTO we are developing five such runtimes: Nanos and XiTAO (targeting CPUs), and OmpSs@FPGA, DFiant and MaxJ (targeting FPGAs). Support for GPUs (via CUDA and OpenCL) is already existing in the OmpSs distribution which forms the basis of the LEGaTO tool chain.

5.1.2. Generation of Supertasks

LEGaTO supports a novel concept called supertasks. The idea behind supertasks is to decouple the parallelism of the computational directed acyclic graph (DAG) from the provided resources. Each supertask is then provided with its own scheduler which takes care of executing the individual tasks in the computational DAG. This concept is useful for the compilation of DAGs to FPGA hardware (where resources need to be statically provided, and the FPGA needs to take care of the execution of the kernel). It is also equivalent to the concept of Task Assembly Object (TAO) which is currently being explored in the XiTAO runtime.

Figure 5.2 shows two approaches in which supertasks can be generated from a computational DAG resulting from an OmpSs computation. The first approach,

called *explicit resource containers*, simply assigns a static set of resources to a parallel computation identified by the OmpSs code. This concept is currently implemented in the XiTAO runtime. The second approach is to take an OmpSs DAG and find a connected partition of nodes that tries to minimize the amount of communication between different partitions of the DAG. This approach is being explored as a way to generate code for both FPGAs and XiTAO.

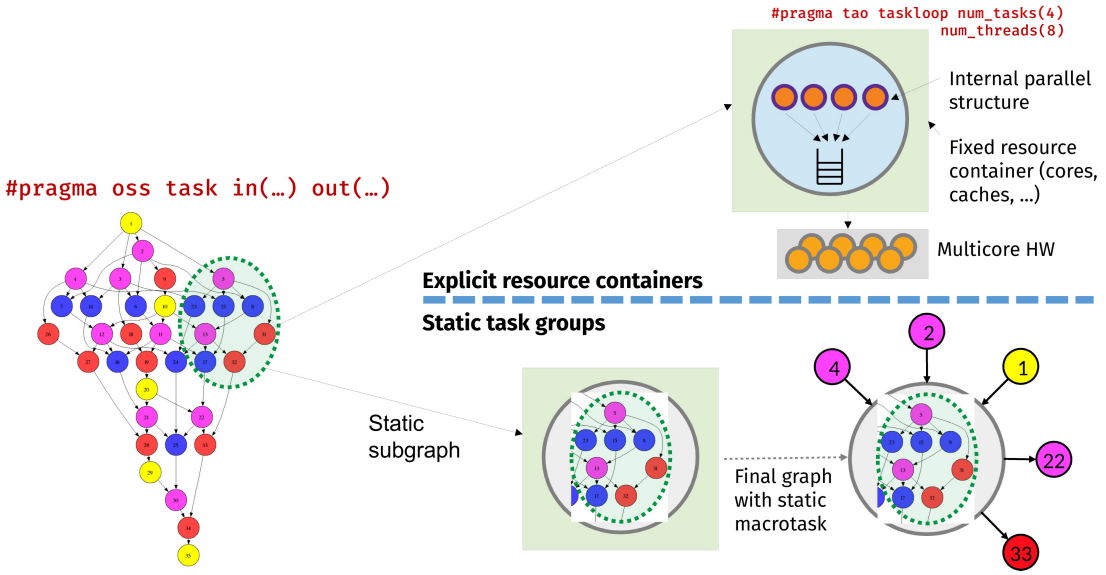


Figure 5.2. Generation of supertasks in LEGaTO

5.1.3. Resource Sharing Between Nanos and the XiTAO RT

In general, each device is managed by a single runtime. However, in the case of CPU devices, a challanging case arises when a user code calls multiple library functions that are parallelized using different runtime systems. Imagine that the programmer wants to execute a matrix multiplication written in Nanos and, concurrently, she wants to run an FFT written in XiTAO. In a traditional system, she would have to execute first the matrix multiplication written in Nanos and wait until the parallel resources have been reclaimed before executing the XiTAO FFT. This is shown on the left side of Figure 5.3. Evidently, this approach leads to idleness and overall low utilization of the hardware.

Instead, in LEGaTO we implemented a scheme in which both Nanos and XiTAO can operate in parallel by sharing resources. This is achieved by a two-step approach in which the user first partitions the resources between Nanos and XiTAO by using the `xitao_set_mask()` and `nanos_set_mask()` functions. Once resources have been partitioned, the user can then concurrently offload kernels to the two runtims by using the `xitao_async_kernel_launch()` and `nanos_async_kernel_launch()` calls. This is shown on the right side of Figure5.3.

5.2. Heterogeneity and Energy-Aware Task-based Scheduling

Current cloud providers offer many different types of hardware choices supporting their virtualised services. Examples are machines following architectures as Intel x86, ARM, or IBM Power, featuring GPUs, FPGAs, or specific architectural ex-

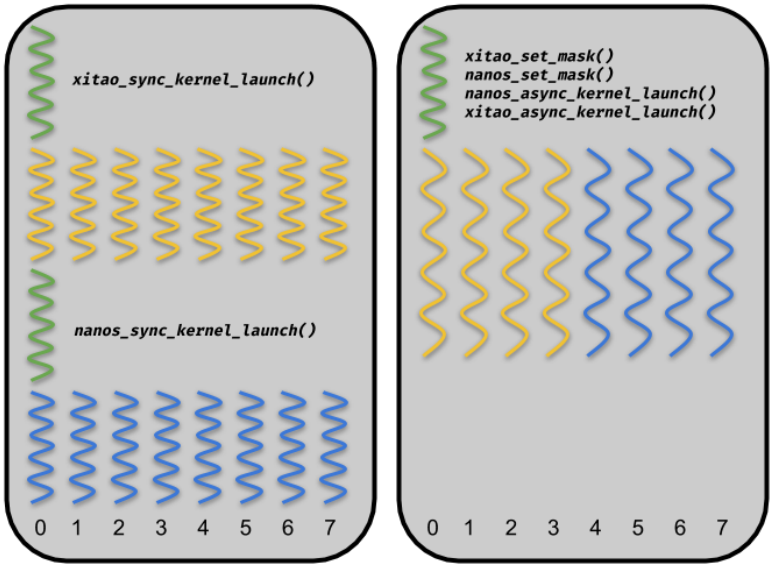


Figure 5.3. Sharing CPU resources between Nanos and the XiTAO RT

tensions as SSE or SGX. This allows for customers to tailor their applications according to specific hardware. We performed the study described below considering that containers are the instances used by customers to deploy applications, as a de facto standard for micro-services, or to execute computing tasks. With that consideration in mind, we argue that the underlying orchestrator should be designed so as to take into account and exploit hardware diversity. In addition, besides the feature range provided by different machines, there is an often overlooked diversity in the energy requirements introduced by hardware heterogeneity, which is simply ignored by the default container orchestrator placement strategies. In the sections below we introduce HEATS, a new task-oriented and energy-aware orchestrator for containerized applications targeting heterogeneous clusters. HEATS allows customers to trade performance for energy requirements. We will explain how HEATS learns the performance and energy features of the physical hosts and then monitors the execution of tasks on the hosts and opportunistically migrates them onto different cluster nodes to match the customer-required deployment trade-offs.

5.2.1. Motivation for Energy and Heterogeneity Awareness

Cloud providers nowadays provide access to a wide range of heterogeneous resources to their customers. Hence, the diversity of resources encourages application developers and deployers to program for, and offload even more workloads to, the cloud. There, specialized hardware (*e.g.*, GPU, FPGA) can be rented for limited time, reducing upfront costs and allowing for better scalability.

To illustrate this diversity, Table 5.1 shows an overview of the commercial offering of heterogeneous resources at six major public cloud providers. For each, we list the CPU architecture (x86, IBM Power, ARM), and the availability of GPU, FPGA or ASIC units. We further indicate if such resources can be accessed using bare metal (BM) or virtual machine (VM) instances. Additionally, we show whether the operating frequency of the processor can be dynamically scaled up or down, a feature that could be leveraged to reduce the generated energy costs of a node. This quick survey reveals that it is possible to combine a very heterogeneous en-

semble of machines, each offering specific hardware feature sets. This capability represents the ideal case for applications that have different resource demands, as it is sometimes better to migrate the execution from a machine of one kind to a different one, in order to better match the expected trade-off requested by the customer. Resource diversity can also be exploited to deploy applications and workloads of different nature.

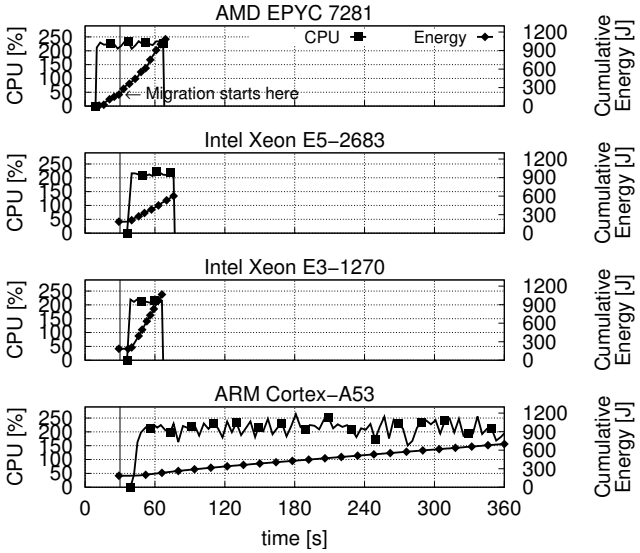


Figure 5.4. Migrating a task to a different host allows for energy savings but increased run time.

Containers (*e.g.*, Docker [75]) have recently become the de facto standard to deploy applications on the cloud, executed by specialized container orchestrators, such as Google's Kubernetes [74]. Current policies of container orchestrators often ignore the diversity found in hardware, leading to subtle trade-off between energy and performance. To better understand this aspect and motivate our work, we conducted a simple experimental study (Figure 5.4). We set up an on-premise cluster composed of 4 different types of nodes: three server-grade machines (two Intel and one AMD) and one ARM-based low-energy device (a Raspberry Pi). Each machine has different hardware characteristics (*e.g.*, number and type of CPU cores, memory and operating frequency) and energy requirements.

While these properties are known by the cluster owner at deployment time, the energy requirements as well as the raw computing power of the machines for a specific workload are not. Typically, customers are only able to evaluate those at runtime, while executing their applications. Because of that, they can face unexpected costs or missed deadlines upon completion of tasks.

In our scenario, we developed and deployed a simple task implementing the popular *k-means* clustering algorithm. At first, the task is deployed on the AMD node (Figure 5.4, top-most plot). Given our cluster settings, with the default Kubernetes scheduler, we observe the deployment on the machine with more cores and memory. When remaining in the same host, the task completes after 69 seconds, consuming 1,047 Joules.

Next, we consider customers wishing to compromise the running time for energy costs. This requires a dynamic container rescheduling policy that can migrate a task into the ARM node after it has made some progress but before completion

*Table 5.1. Heterogenous resources available at public cloud providers. Some types available only via bare metal (BM) or virtual machines (VM). ***** = frequency scaling enabled. ✓= feature available from VM or BM. ✗= not available.*

Provider	x86-64		POWER		ARM		GPU		FPGA		ASIC	
	BM	VM	BM	VM	BM	VM	BM	VM	BM	VM	BM	VM
Amazon [17]	✓*	✓	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗
Microsoft [76]	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗
Google [49]	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
IBM [55]	✓*	✗	✓*	✗	✗	✗	✓	✗	✗	✗	✗	✗
Oracle [83]	✓*	✓*	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
Scaleway [87]	✓*	✗	✗	✗	✓*	✗	✗	✗	✗	✗	✗	✗

(e.g., 30 seconds after startup, as highlighted by the vertical line in each plot). In doing so, the net energy savings are important (up to 34%) but at the cost of a $5.4\times$ increase of the task's running time.

Such trade-offs are often desirable (especially for deadline-free, low-priority workloads), but difficult to achieve in practice. A task (or container) orchestrator would need to be aware of several factors and able to: (1) know or learn the characteristics of the underlying cluster and its hardware resources; (2) understand the trade-off that a customer is willing to accept; (3) observe if a better placement opportunity exists for the currently executing tasks; and (4) migrate the task accordingly. In the text that follows we introduce HEATS, a scheduling system geared toward heterogeneous clusters that achieves these goals.

The key mechanism used by HEATS consists in offering to clients the ability to indicate, at deployment time, their intended energy-performance ratio (the acceptable *trade-off*), in the form of an *H* value. Thereafter, HEATS continuously matches the demanded *H* value to the available resources, considering the resources themselves, pre-built performance and energy models, and the possibly conflicting requirements from other concurrent tasks.

5.2.2. HEATS Scheduling Policy

In this section we describe the scheduling algorithm implemented by HEATS. Algorithm 1 describes the main functions, which we detail next.

The resource requirements of a task, as for instance memory or number of cores, are specified before submission. Resource availability in the hardware nodes is monitored (in our practical experiment we used Heapster [8]) and reported to HEATS monitoring module. Then, HEATS computes suitable nodes for execution considering the resource requirements for all previously running tasks as well as the availability reported by the underlying system. Next, the algorithm executes a profiling phase and estimates the performance and energy requirements of the given task in each of the previously computed available nodes. Finally, the scheduling module relies on these estimations to compute scores for each node, to be weighted by the energy/performance ratio defined by the client (e_w and p_w in Algorithm 1). The best fitting node is chosen to deploy the given task.

Algorithm 1 Task scheduling in HEATS.

```
1:  function SCHEDULE
2:      while pendingTasks ≠ ∅ do
3:          t = pendingTasks.poll()
4:          bestFit ← BESTFIT(t, e_w, p_w)
5:          ASSIGN(t, bestFit)
6:  function RESCHEDULE
7:      for t ∈ runningTasks do
8:          bestFit ← BESTFIT(t, e_w, p_w)
9:          if bestFit ≠ currentHost then
10:             MIGRATE(t, currentHost, bestFit)
11: function BESTFIT(t, e_w, p_w)
12:     r ← REQUIREDRESOURCES(t)
13:     n ← AVAILABLENODES(r)
14:     scores ← SCORES(n, r, e_w, p_w)
15:     return n | {n, s} ∈ scores ∧ s = max_s
16: function SCORES(nodes, r, e_w, p_w)
17:     scores ← ∅
18:     n_e, n_p ← PREDICT(nodes, r)
19:     for n ∈ nodes do
20:         n_s ← e_w(1 − n_e/max_e) + p_w(n_p/max_p)
21:         scores.add({n, n_s})
22:     return scores
```

In summary, the HEATS strategy will attempt to place tasks on the most efficient host that still has enough resources to run the given task. We define *most efficient* as the closest match to the demanded energy/performance trade-off. However, the ideal node for a task will not always be available at scheduling time. Therefore, we recompute our scheduling decision every now and then. When a better fit than the current host of a task is found, the scheduler performs a migration.

The scheduling phase is triggered for the queue of all pending tasks. The algorithm starts by finding the best fit for the next task (lines 4 and 11–15). It identifies its resource requirements, *e.g.*, CPU and memory, as well as the available nodes for these resources (lines 12–13). Then, it computes the score for each of the nodes (lines 16–22). The model is used for the profiling of nodes (line 18). The scores are computed by normalizing the predictions and adding the demanded weights (line 20). Every x seconds the rescheduling phase is triggered for the set of all running tasks. If the re-execution of the best fit decides on a different target node, the task is migrated to the new host and removed from the current one (lines 9–10). We show in our evaluation that x, for our specific workload and cluster settings, has minimal impacts on the runtime or the energy efficiency of HEATS.

5.2.3. Architecture for Scheduling Heterogeneity and Energy

The architecture of HEATS is composed of several interacting components. Figure 5.5 depicts these interactions. We describe each of them in details in the remainder of this section.

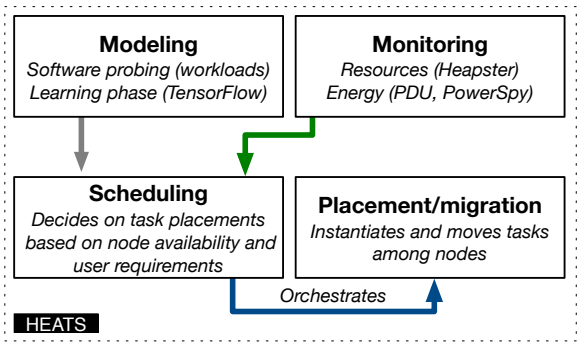


Figure 5.5. HEATS' abstract components and interaction.

Modeling. The modeling component executes two main operations, namely probing and learning, descibed below.

The probing phase discovers the properties and capabilities of the cluster, *i.e.*, the machines composing it. This probing phase is executed upon the initial setup of HEATS, as well as for every major hardware reconfiguration (such as the integration of new machine types in the cluster pool). We implemented this probing so that it also takes care of exploring the performance of the nodes by scaling up and down the frequency of the CPUs [61]. We report that, in a typical setup, to produce an accurate model of a new machine usually requires a few hours. Figure 5.6 shows the results of possible characterizations that this phase can produce, when applied to the machines of our cluster. In particular, it outputs the runtime and energy requirements of two different families of *probing tasks*. The energy requirements reported here do not consider the idle state of the machines but of the task itself only. In this way we can better understand the tasks energy requirements for the differnt types of hardware given. We show the results with two of such CPU-bound tasks: the aforementioned *k-means* clustering algorithm, as well as a typical matrix multiplication operation. For both types of probing tasks, we observe that the energy requirements can be reduced on a given performance cost for almost every machine type. The framework further executes these probing tasks by frequency scaling of the underlying CPUs. We achieve this by leveraging two different Linux CPU governors [13], *powersave* and *performance*, respectively running the CPU at the minimum and maximum frequency. We can observe that, within the same machine type, the energy and performance are largely affected by scaling the CPU frequency. The output of this phase is used next.

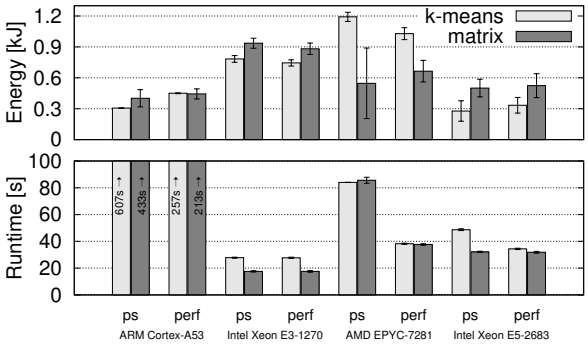


Figure 5.6. Runtime and energy spent by tasks executing `k-means` *and a matrix multiplication with two different CPU governors: powersave (ps) and performance (perf).*

The data collected by the probing phase is used to train a multiple linear re-

gression model [79]. Given a task and its CPU and memory requirements, a fitted regression model is used to predict its energy and performance for each machine type available in the cluster. We did a preliminary analysis of different machine learning techniques and, for the workload used, TensorFlow [14] presented better results. While the probing component constantly records new data, HEATS uses it to refine the predictions at a given frequency. In our evaluation, we execute the learning phase every 24 hours.

Monitoring. Kubernetes is equipped with several tools to monitor resources: cAdvisor [7] has been partially integrated into Kubernetes' node agent kubelet [12], and it is capable of measuring resources used by containers. Heapster [10] exploits the measurements from cAdvisor, aggregates them and provides means to analyze and monitor the state of the Kubernetes cluster using Grafana [9]. Furthermore, Heapster allows us to store the aggregated data in InfluxDB [11], a time-series database that supports SQL-like queries to retrieve historical resource measurements of the Kubernetes cluster.

In order to decide whether a task has to be migrated from one node to a different heterogeneous node, the HEATS scheduler has to be able to rely on a fine grained resource monitoring system. Despite the potential capability to gather resource measurements every 5 s, we found out that Heapster cannot reliably deliver these resource measurements at a fixed rate. A custom resource measurement system was therefore implemented and installed on the Kubernetes nodes, which queries every second the local Docker instances for up-to-date resources used by the containers. These resource measurements can then be aggregated and used by the HEATS scheduler to provide the needed support for migrating tasks.

The monitoring component is responsible for actively gathering information regarding the resources currently being consumed at each node by the tasks in execution. This information is required by the scheduling component (described below) to know which node has sufficient resources for the pending tasks. HEATS leverages some default software probes from Heapster to continuously fetch the hardware resources available on any given node.

Additionally, to access in real-time the current power and energy levels of a node, we assume the availability of hardware monitors that are remotely accessible. In order to assess power consumption in heterogeneous environments, we experimented with two different types of energy monitors, one for server-grade machines and one for low-energy profiles.

Scheduling. Finally, the scheduling component is in charge of orchestrating the inputs received by the modeling and monitoring components. To that end, it first ensures that a prediction for the resources used by the task on the different set of machines is completed. Then, it combines this prediction with the energy and performance trade-offs, as defined by the end-user, to decide on the best fitting node. Periodically, the scheduling component reconsiders its past decisions: when a better fitting node is found, a migration decision is taken and the corresponding task is moved to the target node.

5.3. Trusted Execution Environments

Nowadays, public cloud systems are the *de facto* platform of choice to deploy online services. All major *information technology* (IT) players provide some form of *infrastructure as a service* (IaaS) commercial offerings, including Microsoft [86], Google [41] and Amazon [15]. IaaS infrastructures allow customers to reserve and use (virtual) resources to deploy their own services and data. These resources are eventually allocated in the form of virtual machines (VMs) [16], containers [70] or bare-metal [31] instances over the cloud provider's hardware infrastructure, in order to execute the applications or services of the customers. Privacy concerns have greatly limited the deployment of such systems over public clouds [84]. Moreover, despite the existence of pure software-based solutions leveraging homomorphic encryption [80], their performance is several orders of magnitude behind the requirements of modern workloads.

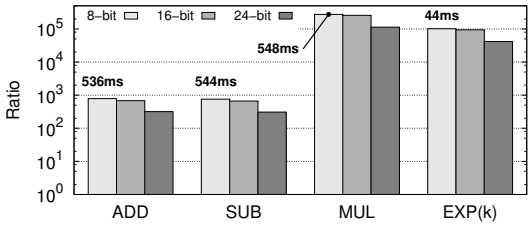


Figure 5.7. Performance of simple arithmetic operations using state-of-the-art homomorphic encryption with HElib [53]. Numbers indicate execution time in milliseconds for a batch of operations.

The recent introduction of new hardware technologies to enable *trusted execution environments* (TEEs) inside x86 processors by Intel and AMD paves the way to overcome the limitations of the aforementioned software-only solutions. Intel introduced *software guard extensions* (SGX) [39] with its Skylake generation of processors in August 2015. These instructions allow applications to create TEEs to protect code and data against several types of attacks, including a malicious underlying *operating system* (OS), software bugs or threats from co-hosted applications. The security boundary of the application becomes the CPU die itself. The code is executed at near-native execution speeds inside *enclaves* of limited memory capacity.

AMD recently introduced *secure encrypted virtualization* (SEV) [57,88] with its Zen processor micro-architecture. Specifically, the EPYC family of server processors introduced the feature on the market in mid-2017 [18, 62]. The SEV encrypted state (SEV-ES) [56] technology, an extension to SEV, protects the execution and register state of an entire VM from a compromised hypervisor, host OS or co-hosted VMs. Unmodified applications are protected against attackers with full control over the hosting machine, which in turn can only access encrypted memory pages.

On the edge-side, devices are typically designed for collecting and disseminating data to cloud infrastructure. The devices have a small form factor on which they arrange a specific set of low power hardware needed to control or monitor a physical system. A majority of these embedded and mobile devices is equipped with ARM processors. Since more than 15 years [54], most ARM application level processors feature a set of security extensions known as ARM TrustZone. ARM has been continuously improving TrustZone specifications with new processor revisions. For instance, ARM recently [24] updated its ARMv8.4 architecture of

application processors enabling virtualization in the secure world. The introduction of virtualization in the secure world better improves the isolation of components and resources, and it is expected to boost the trusted applications (TA) ecosystem in developing and using common standards and APIs.

Despite the availability of security-oriented instruction sets in consumer-grade processors, high-level frameworks that can help developers use such extensions are still at an early stage. Moreover, little has been said regarding the performance and usability of these frameworks. It is only very recently that the first open-source tools aiming to exploit these capabilities have emerged. Notable examples for ARM TrustZone include Linaro ARM Trusted Firmware [63], ARM GNU Toolchain [19], Android's Trusty [50], Trustonic's Kinibi [90], NVIDIA's TLK [81], and finally Linaro's OP-TEE [66].

A major challenge for developers of trusted applications resides in the complexity of the secure platforms themselves. Despite the existence of standards and APIs, trusted applications remain OS-specific because of the custom libraries provided by the different vendors. Theses libraries are specialized for the various processors and are required to access secure storage and processing elements. They rely on drivers shipped with the hardware by the silicon manufacturer. Furthermore, dispatching trusted OSs requires trusted OS-specific code in the firmware, which adds up to the issue. This greatly hinders the portability of trusted applications across different trusted OSs and, as consequence, forces TA developers toward implementing and supporting several versions of trusted OS-specific TAs.

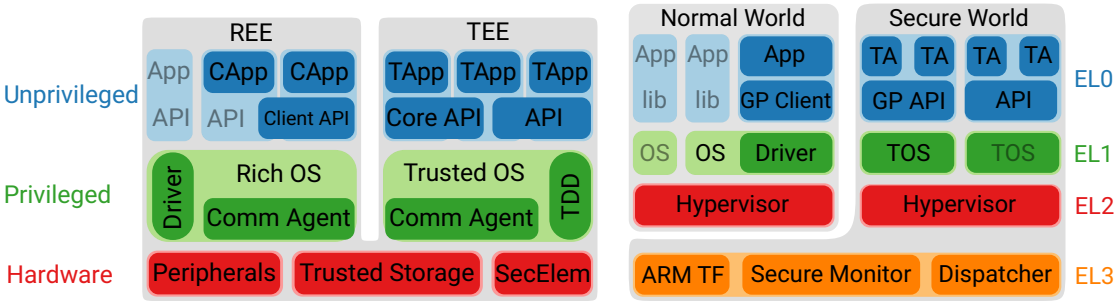


Figure 5.8. Block diagram of GlobalPlatform TEE System Architecture [47] (left) and block diagram of ARM TrustZone according to ARMv8.4 [23] (right). Blocks with rich colors represent components which interact with the TEE, while blocks with pale colors represent components which are not involved with the TEE or involved with a different TEE.

5.3.1. Background of Trusted Execution Environments

5.3.1.1. GlobalPlatform Specifications

GlobalPlatform [48] is an industry association which publishes specifications for secure digital services and devices. These specifications serve as basis to many implementations [67,72]. In the scope of this background, we introduce some of the terms defined by GlobalPlatform [47], also depicted in Figure 5.8 (left).

An *execution environment* (EE) has to provide all necessary components to execute applications. This includes hardware and software components, such as a processing unit, physical memory in volatile and non-volatile form, peripherals, buses connecting system resources, *application programming interfaces* (API),

and instruction set architecture (ISA) [82]. A *rich execution environment* (REE) is based on a device with all its components (excluding any secure components), which are managed by at least one *rich operating system* (rich OS). A rich OS is an *operating system* (OS) that is generally designed for performance and offers an extensive amount of functionality. In contrast, a *trusted execution environment* (TEE) is designed for security and offers only a reduced set of functionalities to programmers. Generally speaking, a TEE complements a REE, while the two are executed alongside each other. The TEE is managed by a *trusted operating system* (trusted OS), typically with very small memory and storage footprint. According to [47], a TEE can host no more than one trusted OS, a limitation that is relaxed by ARM's latest design [23] for ARMv8.4. In addition, a trusted OS implements the necessary system calls to provide *trusted applications* (TA) with the *TEE Internal API* [45] and to facilitate communication with *client applications* (CApp) in the REE over the *TEE Client API*.

The TEE Internal APIs provide a common basis to TAs for accessing system resources or to issue requests from the trusted OS. Figure 5.8 (left) highlights the *TEE Internal Core APIs* and the *TEE Socket API*, which are the two primary libraries used to implement our network performance tool. The CApps can request services from TAs by making use of the TEE Client API. The TEE Client API forwards these service requests to a REE *communication agent* (Comm Agent), typically implemented as a driver in the rich OS. The REE communication agent then uses some form of data exchange with the TEE *communication agent* residing in the trusted OS. Finally, the TA receives the request and sends an answer over the same path back to the CApp.

5.3.1.2. ARM TrustZone

ARM TrustZone is a hardware security architecture, which partitions all hardware and software resources of the *system on a chip* (SoC) into two worlds: a *secure world* and a *normal world*. The partitioning of the SoC is materialized by introducing an additional bit used for read and write channels on the main system bus and in cache tags [20]. This additional NS bit (*i.e.*, *non-secure*) indicates a secure transaction when set low, or a non-secure transaction when set high. The bus master sets the NS bit upon each transaction issued on the main system bus. The NS bit then has to be decoded by the bus or the slave to verify that security requirements are not violated. In order for a transaction to succeed, the decoded address has to match the system resource being accessed. Thus, it is not possible for a non-secure bus master to access a secure bus slave.

This TrustZone architecture design allows to span beyond the main system bus to include IO peripherals (*e.g.*, GPUs [93]) as well without adding a dedicated processor. For backward compatibility reasons, the peripheral bus is connected over a bridge to the main system bus. Consequently, the peripheral bus is not carrying any NS bits and it is up to the bridge to secure the signals to the peripherals. The bridge must only forward valid transactions and prevent invalid transactions from reaching the peripheral bus.

Every physical core of a TrustZone-enabled SoC consists of two virtual cores, respectively a secure and a non-secure one. The two virtual cores are executed in a time-shared manner. Hence, the currently executing virtual core determines

what are the accessible system resources. For instance, the non-secure virtual core can only access non-secure system resources. Vice versa, the secure virtual core can access all system resources.

ARM-based SoCs are equipped with a *memory management unit* (MMU). This provides one virtual MMU for each virtual core. The MMU maps the virtual address space onto the physical address space. The *translation lookaside buffer* (TLB) within the MMU is used by privileged software to store recent translations of virtual to physical memory addresses. Privileged software typically flushes the set of address translation entries stored in the TLB and replaces it by a new set of address translations when switching contexts. This can speed up the address translation process and avoids privileged software to walk through address translation tables. By considering the NS bit in cache tags, the entire address space is effectively divided into a secure and a non-secure address space. Furthermore, it allows secure and non-secure address translation entries to co-exist in the TLBs. Similarly, secure and non-secure lines can co-exist in caches. Independent of the value of the NS bit, secure lines can evict non-secure lines and non-secure lines can evict secure lines.

TrustZone is found in ARM application processors since 2003 [23]. Since then, the implementation of TrustZone is organized into four *exception levels* (EL) with increasing privileges [22], also shown in Figure 5.8 (right). EL0 is the lowest exception level and used to execute unprivileged software. EL1 is used to execute operating systems, and EL2 provides support for virtualization and is used to execute hypervisors. Finally, EL3 controls the secure state. Exception levels can be changed by executing instructions that either *take an exception* (increase the exception level) or *return from an exception* (decrease the exception level).

Context switches between the two worlds are supervised by a firmware [21], *i.e.*, the *secure monitor*. The secure monitor is executed at the highest exception level EL3 and can be invoked in two ways: by executing a *secure monitor call* (SMC) instruction, or by a subset of *hardware exception mechanisms* [20]. When invoked, the secure monitor saves the state of the currently executed world before restoring the state of the world being switched to. After having dealt with the states, the secure monitor returns from exception to the restored world.

Open Portable Trusted Execution Environment. The *Open Portable Trusted Execution Environment* (OP-TEE) is an open-source framework and implementation of several GlobalPlatform's specifications [43, 44, 46, 47]. It is actively developed and maintained by the Linaro *Security Working Group* [65] (SWG). OP-TEE provides support for TrustZone-enabled SoCs. The OP-TEE OS forms the primary component of the project and the TEE it manages. Any Linux-based distribution can be used as rich OS to run alongside OP-TEE OS. Two types of TAs are supported by OP-TEE: *(1)* regular TAs [47], and *(2) pseudo TAs* which are statically linked against the OP-TEE OS kernel. Regular TAs are being executed at EL0, while pseudo TAs run at EL1 as secure privileged-level services inside the kernel of OP-TEE OS. OP-TEE provides a set of client libraries to interact with TAs and to access secure system resources from within the TEE.

5.3.1.3. Intel SGX

Intel SGX provides a TEE in modern processors that are part of the Skylake and more recent generations. It is similar in spirit to ARM TrustZone [6]. Applications create secure *enclaves* to protect the integrity and confidentiality of the code being executed and its associated data.

The SGX mechanism, as depicted in Figure 5.9 (middle), allows applications to access confidential data from inside the enclave. An attacker with physical access to a machine cannot tamper with the application data without being noticed. The CPU package represents the security boundary. Moreover, data belonging to an enclave is automatically encrypted and authenticated when stored in main memory. A memory dump on a victim's machine will produce encrypted data. A *remote attestation protocol* (not shown in the figure) is provided to verify that an enclave runs on a genuine Intel processor with SGX enabled. An application using enclaves must ship a signed, yet unencrypted shared library (a shared object file in Linux) that can be inspected, possibly by malicious attackers.

The *enclave page cache* (EPC) is a 128 MiB area of memory predefined at boot to store enclave code and data. Any access to an enclave page outside the EPC triggers a page fault. The SGX driver interacts with the CPU and decides which pages to evict. Traffic between the CPU and the system memory is kept confidential by the *memory encryption engine* (MEE) [51], also in charge of tamper resistance and replay protection. If a cache miss hits a protected region, the MEE encrypts or decrypts data before sending to, respectively fetching from, the system memory and performs integrity checks. Although future releases of SGX intend to relax the EPC limitation [59, 73], we cannot yet guess the performance of the new mechanism, and it sounds reasonable to expect some loss due to encryption and decryption. Finally, data can also be persisted on stable storage, protected by a seal key. This allows storing certificates and waives the need of a new remote attestation every time an enclave application restarts.

5.3.1.4. AMD SEV

AMD SEV provides transparent encryption of the memory used by virtual machines. To exploit this technology, the AMD *secure memory encryption* (SME) extension must be available and supported by the underlying hardware. The architecture relies on an embedded hardware *advanced encryption standard* (AES) engine, itself located on the core's memory controller. SME creates one single key, used to encrypt the entire memory. As explained next, this is not the case for SEV, where multiple keys are being generated. The overhead of the AES engine is minimal.

SEV delegates the creation of *ephemeral* encryption keys to the AMD *secure processor* (SP), an ARM TrustZone-enabled *system on a chip* (SoC) embedded on-die [57]. These keys are used to encrypt the memory pages belonging to distinct virtual machines, by creating one key per VM. Similarly, there is one different key per hypervisor. These keys are never exposed to software executed by the CPU itself.

It is possible to attest encrypted states by using an internal challenge mechanism, so that a program can receive proof that a page is being correctly en-

crypted.

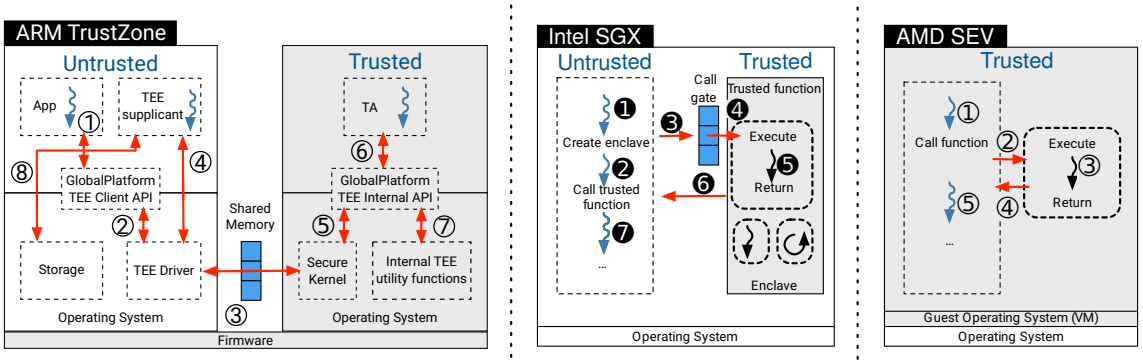


Figure 5.9. ARM TrustZone, Intel SGX and AMD SEV operating principles

5.3.1.5. Comparison of TEEs

Memory limits *ARM TrustZone.* The main memory of an ARM TrustZone-enabled SoC can be partitioned at boot time into a non-secure and a secure memory area. When the SoC boots, it is in secure mode from where it has access to the entire memory. In a first boot stage the firmware is instructed to reserve a defined memory area for the secure world. It is then the task of the trusted OS to manage the secure memory. The trusted OS can impose further restrictions on the shared memory used to exchange data between the worlds or the size of TAs. Typically, only a few megabytes are allocated for the secure memory, to keep the *trusted computing base* (TCB) as small as possible.

Intel SGX. The EPC area used by SGX is limited to 128 MiB, of which 93.5 MiB are usable in practice by applications; the remaining area is used to maintain SGX metadata. The size of the EPC can be controlled (*i.e.*, reduced) by changing settings in the *unified extensible firmware interface* (UEFI) setup utility from the *basic input/output system* (BIOS) of the machine.

AMD SEV. This limit does not exist for SEV: applications running inside an encrypted VM can use all its allocated memory.

Usability *ARM TrustZone.* The communication between an application in the normal world and a TA evolves around functions handling the context, session, command, and shared memory as shown in Figure 5.9 (left). This facilitates interoperability between different GlobalPlatform API compatible TEE implementations and allows REE applications to set up multiple contexts. A context is initialized by referencing the device file Figure 5.9-① connecting to the TEE driver Figure 5.9-②. TAs are identified by a *universally unique identifier* (UUID), which is referred to when setting up a session to a TA Figure 5.9-③. To set up a session, OP-TEE will load the TA from the normal world to the secure world with the help of `tee-supplicant` Figure 5.9-④. The `tee-supplicant` is a daemon running in the normal world used by OP-TEE to request services from the REE. These steps are skipped when a session to a pseudo TA is established. A TA can initialize and set up its environment upon TA creation and session establishment (Figure 5.9-⑤ & Figure 5.9-⑥). From this point on, the REE application can request services from the TA by invoking commands. These commands can pass up to four parameters, which are either values or references to shared memory regions. Values are pairs of unsigned 32 bit integers. Shared memory regions are allocated, reg-

istered and released through GlobalPlatform API calls in `libteec`. Without the availability of `libteec`, developers would have to communicate directly with the kernel driver through `ioctl` system calls.

In OP-TEE, TAs can use services accessible through GlobalPlatform Internal Core API Figure 5.9-⑥ implemented in `libutee`. TAs are statically linked against `libutee`, which wrapps the API functions around assembler macros to OP-TEE OS system calls. The library provides interfaces to secure storage Figure 5.9-⑧, time, arithmetic and cryptographic operations Figure 5.9-⑦. The secure storage API encrypts data objects by the use of a secure storage service. The encryption process involves three keys: *secure storage key* (SSK), *trusted application storage key* (TSK) and *file encryption key* (FEK). The SSK is generated from the *hardware unique key* and is used to derive TSKs. Each TA has a TSK that is generated from the SSK and the TA's UUID. Both SSK and TSK are generated using HMAC SHA256 algorithm [68]. Finally, for every created file, a FEK is generated from the pseudo random number generator. The encrypted data objects are then transferred to the `tee-supplicant` by a series of *remote procedure calls* (RPC) and stored in a special file. OP-TEE further provides TAs with libraries for TLS and SSL protocols (`libmbedtls` [25]), arithmetic (`libmpa`) and a subset of ISO C functions (`libutils`). These libraries are used in part by OP-TEE to implement GlobalPlatform's Internal Core APIs, in particular the *Arithmetical API* and the *Cryptographic Operations API*. Without these libraries, TA developers would have to provide this code, and they would not be able to just simply link their TA's code against this set of initial libraries. Once the REE application has no further service requests, the session is terminated and the context is destroyed.

Intel SGX. To use SGX enclaves, a program needs to be modified—requiring a re-compilation or a relink—*e.g.*, using the official Intel SGX SDK [38]. It is the responsibility of developers to decide which sections of the programs will run inside and outside the enclave. Recently, semi-automatic tools [69] have been introduced to facilitate this process.

The execution flow of a program using SGX enclaves is as follows. First, an enclave is created (see Figure 5.9-❶, left). As soon as a program needs to execute a trusted function (Figure 5.9-❷), it invokes the SGX `ecall` primitive (Figure 5.9-❸). The program goes through the SGX call gate to bring the execution flow inside the enclave (Figure 5.9-❹). Once the trusted function is executed by one of the enclave's threads (Figure 5.9-❺), its result is encrypted and sent back (Figure 5.9-❻) before giving back the control to the main processing thread (Figure 5.9-❼).

AMD SEV. As mentioned in the previous section, no changes need to be made to programs when using SEV. From the programmer perspective, SEV is completely transparent. Hence, the execution flow of a program using it is the same as a regular program, as shown in Figure 5.9 (right). Notably, all the code runs inside a trusted environment. First, a program needs to call a function (Figure 5.9-①). The kernel schedules a thread to execute that function (Figure 5.9-②) before actually executing it (Figure 5.9-③). The execution returns to the main execution thread (Figure 5.9-④) until the next execution is scheduled (Figure 5.9-⑤).

Integrity protection *ARM TrustZone.* TrustZone does not provide hardware integrity protection mechanisms. However, software leveraging TrustZone can pro-

Device	Intel	AMD
Machine	Supermicro 5019S-M2	Supermicro 1023US-TR4
CPU	Intel Xeon E3-1275 v6	$2\times$ AMD EPYC 7281
CPU Frequency	3.8 GHz	2.1 GHz
Memory	16 GiB DDR4	64 GiB DDR4
Memory data rate	2400 MT/s	2666 MT/s

Table 5.2. Comparison of cloud platforms

vide services that introduce integrity protection, such as for example secure storage or integrity measures during secure boot.

Intel SGX. Intel SGX has data-integrity protection mechanisms built-in. Memory pages that are read from EPC memory by an enclave are decrypted by the CPU, and then cached within the processor. In the opposite direction, data that is being written to the EPC by an enclave is encrypted inside the CPU before leaving its boundaries. The integrity of the data is safeguarded by associating metadata that is themselves integrity protected. The metadata is stored in a Merkle tree structure [42], the root of which is stored in SRAM, inside the processor. These integrity mechanisms incur an overhead that has been previously evaluated and shown to be acceptable for sequential read/write operations, but up to $10\times$ for random read/write operations [26].

AMD SEV. Conversely, to the best of our knowledge, the current version of AMD SEV (or SME) does not provide any integrity protection mechanism [77]. We expect this limitation to be addressed in future revisions.

5.3.2. Evaluation of Trusted Execution Environments

5.3.2.1. Setup for Cloud TEEs

Our evaluation uses two types of machines. The Intel platform consists of a Supermicro 5019S-M2 machine equipped with an Intel Xeon E3-1275 v6 processor and 16 GiB of DDR4-2400 RAM. The AMD machine is a dual-socket Supermicro 1023US-TR4 machine, with two AMD EPYC 7281 processors and 8×8 GiB of DDR4-2666 RAM. Both client and server machines are connected on a switched Gigabit network.

The two machines run Ubuntu Linux 16.04.4 LTS. On the AMD platform, we use a specific version of the Linux kernel based on v4.15-rc1[1] that includes the required support for SME and SEV. Due to known side-channel attacks exploiting Intel's hyper-threading [35], this feature was disabled on the Intel machine, and so was AMD's simultaneous multithreading (SMT) on the AMD machine. We use the latest version of Graphene-SGX [34],[2] while we rely on the Intel SGX driver and SDK [38], v1.9. In order to match the hardware specification of the Intel machine, we deployed para-virtualized VMs on the AMD machine, limited to 4 VCPUs, 16 GiB of VRAM and have access to the host's real-time hardware clock.

[1] https://bit.ly/2y6TVcI
[2] https://github.com/oscarlab/graphene/tree/2b487b09

The power consumptions are 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 1 W and with a precision of $1.5\,\%$.

5.3.2.2. Memory-Bound Operations

We begin with a set of micro-benchmarks to show the performance overhead in terms of memory's access speed imposed by Intel SGX and AMD SEV. We rely on the virtual memory stressors of STRESS-NG as a baseline. On the Intel architecture, we use STRESS-SGX [92], a fork of STRESS-NG for SGX enclaves. We ensure that both SGX-protected and unprotected versions of the stressors execute the exact same binary code, to provide results that can be directly compared against one another.

In the case of the AMD machine, the benchmark is first run in a traditional virtual machine, and subsequently the same benchmark is run again with AMD SEV protection enabled. We replace the `mmap` memory allocation functions of the virtual memory stressors with `malloc` functions to have a fair comparison between STRESS-NG and STRESS-SGX (where `mmap` is not allowed).

Figure 5.10 summarises the results of this micro-benchmark.

Values are taken from the average of 10 executions, where each method is spawning 4 stressors with an execution limit of 30 seconds. The figure can be read in the following way: the percentage of the surface of each disk that is filled represents the relative execution speed in protected mode, compared to the native speed on the same machine for the same configuration. For example, a disk that is $75\,\%$ full (◔) indicates that a stressor ran with protection mechanisms enabled at $0.75\times$ the speed observed in native mode. A full disk (●) indicates that the performance of the associated stressor is not affected by the activation of SGX/SEV.

On both platforms, performance is not affected when the program operates on a small amount of memory (*i.e.*, 4 MiB). The reason is that the protection mechanisms are only used to encrypt data leaving the CPU package. As 4 MiB is smaller than the amount of cache embedded on the CPU on both platforms (as detailed in subsubsection 5.3.2.1), the data never leaves the die and is therefore processed and stored in cleartext.

Both technologies perform better when memory accesses follow a sequential pattern, as observed in the tests *read64*, *gray*, *incdec*, *inc-nybble*, *walk-d1* and *rand-sum*. Conversely, Intel SGX is negatively affected by random memory accesses, as seen for tests *swap*, *modulo-x*, *prime-gray-1*, *walk-0a* and *walk-1a*. AMD SEV is also partially affected under these conditions (tests *swap*, *modulo-x*, *walk-0a* and *walk-1a*). Memory accesses beyond the size of SGX's protected memory (*i.e.*, EPC) are the slowest in our experiment, up to $0.05\times$ less than native memory accesses. Under these conditions methods such as *modulo-x* were not able to produce any results. However, supplemental tests, during which hyperthreading was enabled and all 8 CPUs used, did return results.

Finally, SEV appears to be much faster than SGX (an overall *greener* look for the disks), due to its lack of checks to ensure data integrity protection (as explained

in subsubsection 5.3.1.5). Similarly, larger memory accesses also do not suffer from drastic performance penalties like in the case of Intel SGX.

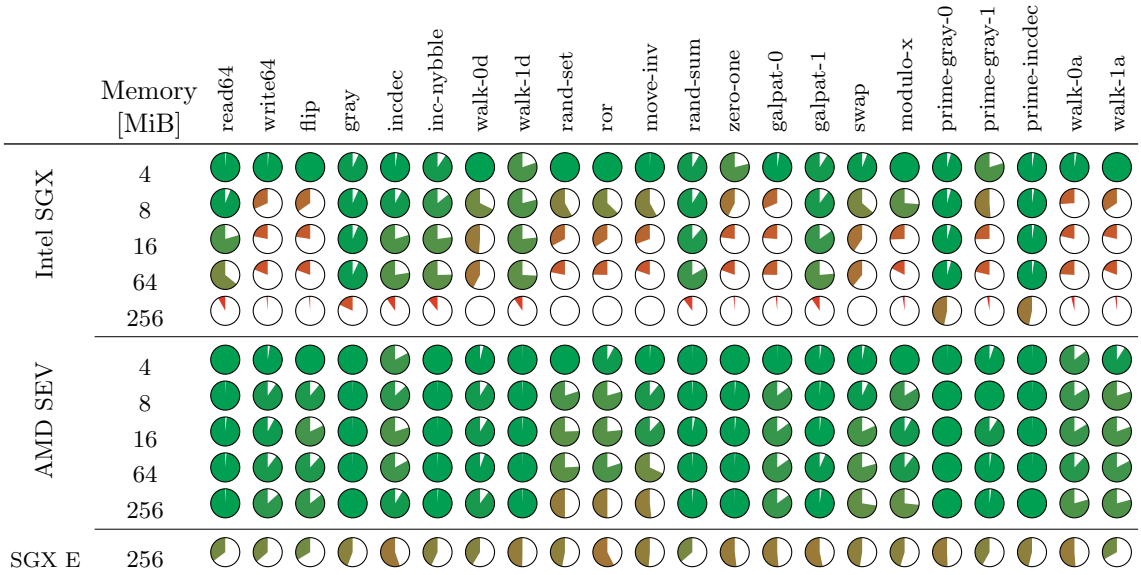


Figure 5.10. Micro-benchmark: relative speed of memory-bound operations using Intel SGX or AMD SEV as protective mechanisms against native performance on each platform. The bottom row shows the relative energy consumption for Intel SGX protective mechanism against native performance. Methods are ordered from sequential (left) to random (right) accesses by increasing memory operation size.

5.3.2.3. Energy Cost of Memory-Bound Operations

To evaluate the energy cost of memory-bound operations we recorded the power consumption while running the micro-benchmark of Figure 5.10.

The results are shown in the bottom row of the table, row *SGX E*. The pie-chart is read as follows: a disk that is 67 % full (◔) indicates that the STRESS-SGX method consumed $1.67\times$ more energy during execution with SGX enabled compared to native performance.

As expected, the energy consumption using SGX increases when the memory size considered is bigger than the EPC memory, and a similar behaviour is observed for each of the stressor method. However, the case of the *move-inv* stressor is different. In this case (Figure 5.11 (right)), SGX mode consumes more energy than native, independently from the memory size. The *move-inv* stressors sequentially fill memory with random data, in blocks of 64 bits. Then they check that all values were set correctly. Finally, each 64 bit block is sequentially inverted, before executing again a memory check.

Conversely, in the case of AMD SEV we did not observe higher energy consumptions compared to native energy consumption, hence these results do not appear in Figure 5.10. Specifically, 108 out of 110 memory stressors confirm that the energy consumption lies within the 3.7 % margin of error, *i.e.*, the precision of the measurement. Only two measurements (*read64* with memory size 16 MiB, and *modulo-x* with memory size 256 MiB) lie slightly outside the range of error and do not confirm the observation.

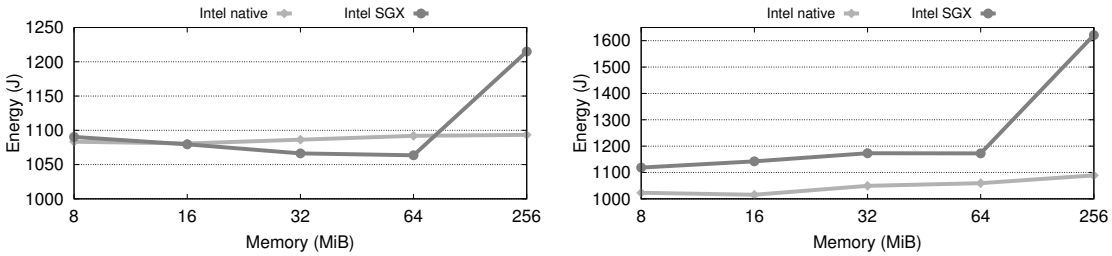


Figure 5.11. Energy measurements for micro-benchmark: overall results (left) and method move-inv *(right).*

Device	QEMU	Raspberry
CPU	Intel Xeon E3-1270 v6	Broadcom BCM2837
CPU Frequency	3.8 GHz	1.2 GHz
Memory	63 GiB DDR4	944 MiB LPDDR2
Memory data rate	2400 MT/s	800 MT/s
Disk	Samsung MZ7KM480HMHQ0D3	Transcend micro SDHC UHI-I Premium
Disk Size	480 GB	16 GB
Disk Read Speed	528.33 MB/s	90 MB/s

Table 5.3. Comparison of edge platforms

5.3.2.4. Setup for Edge TEE

The OP-TEE framework has built-in support for QEMU [32] deployments, providing an easy to use and inexpensive way for developers to explore ARM TrustZone, with little to no downsides compared to hardware deployments. For this reason, we decided to deploy OP-TEE's Sanity Testsuite v3.2.0 [64] on the following two platforms: Dell PowerEdge R330 Server and Raspberry Pi 3B v1.2. The Dell PowerEdge R330 is running Ubuntu 18.04.1 LTS with the 4.15.0-43-generic Linux kernel and is used to emulate the Raspberry Pi 3B platform with QEMU v2.12.0. A comparison of the two platforms can be found in Table 5.3. OP-TEE provides a build environment which, by default, deploys and emulates its OS on an ARM Virtual Machine `virt` using a Cortex-A57 with no more than two cores. The deployment was changed to match the specification of the Raspberry Pi 3B platform as close as possible.

5.3.2.5. Secure Storage

The secure storage benchmark is part of the OP-TEE sanity test suite adhering to the *Trusted Storage API for Data and Keys* described in [46]. Neither of the platforms is equipped with an eMMC, for which reason the secure storage has to be offloaded to the REE file system. The benchmark executes three commands `WRITE`, `READ`, and `REWRITE`, for data sizes in the range of 256 B to 1 MiB, that are accessed in chunks of at most 1 KiB. The `REWRITE` command first reads data from an object, resets the cursor and writes the data back to the same object. The data to be stored in the secure storage is allocated and filled with scrambled

data within the TEE.

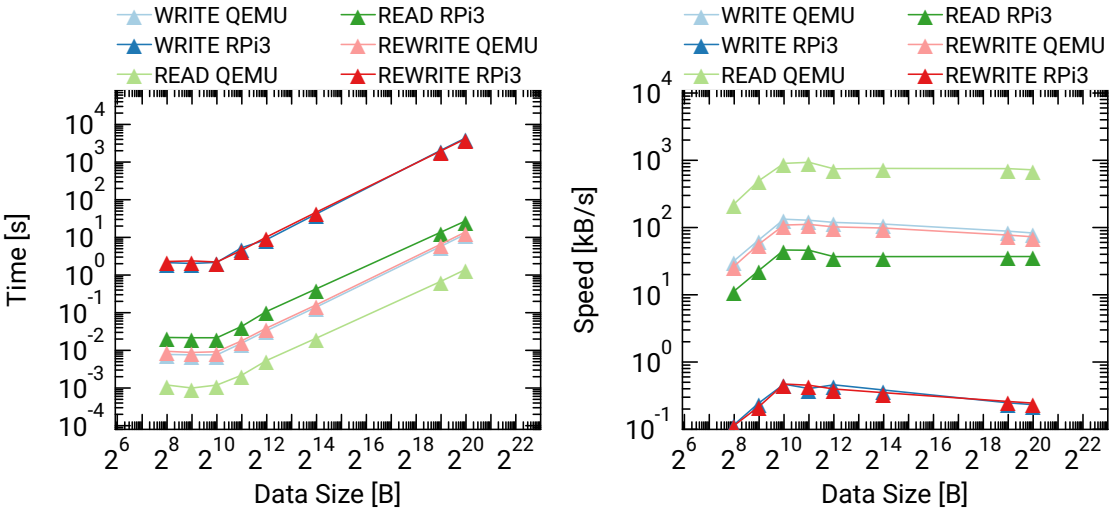


Figure 5.12. Secure storage benchmark execution time and throughput

Figure 5.12 shows the overhead of accessing data in chunks of 1 KiB in the secure storage. In general, the overhead becomes more significant with increasing data sizes, more precisely once the data size exceeds the chunk size. Maximum speed is achieved when the data size equals the chunk size. Overall, the `REWRITE` command has the highest overhead, because it basically executes the `READ` and `WRITE` commands in one batch. It should also be noticed, that the `WRITE` and `REWRITE` operations experience an overhead of about one order of magnitude and more compared to the `READ` operation. The reason for this high overhead is due to the delay in switching between worlds. In the ARM TrustZone usability paragraph of subsubsection 5.3.1.5 are the keys described that, are involved with the secure storage. For encrypting data blocks in the secure storage, the FEK has to be first fetched from the secure storage's meta data. Hence, the `WRITE` and `REWRITE` operations have two additional world switches to perform. Thus, we observe a $7\times$ overhead on QEMU platform and a $100\times$ overhead on Raspberry Pi platform for the `READ` and `WRITE` operations depicted in Figure 5.12.

5.3.2.6. Assessment of Trusted Execution Environments

Privacy-preserving systems would dramatically benefit from the new wave of trusted hardware techniques that are now available in most recent processors sold by Intel and AMD. Their design could be greatly simplified, for instance, by avoiding to rely on complex cryptographic primitives. Development of secure services benefits from well established APIs and standards. OP-TEE implements several GlobalPlatforms specifications and APIs and provides common interfaces for secure services.

We presented an energy, performance, and usability evaluation on the impact of three hardware protection techniques: ARM TrustZone, Intel SGX, and AMD SEV. Our results suggest that AMD SEV is a promising technology: many of our memory-intensive benchmarks run at near native speed. Furthermore, our benchmarks have shown that requesting services from TAs in TrustZone on ARMv8-A using OP-TEE incurs a significant overhead compared to service execution in the normal world.

Limiting the space available to a TA is sensible, in order to minimize the TCB. However, the default memory limit of 1 MiB for TAs in OP-TEE becomes a major inconvenience with respect to secure storage and shared memory. Generating the SSK in OP-TEE requires the HUK. However, most platforms lack of documentation to access or obtain the HUK. OP-TEE avoids this issue by considering a static string value instead of the HUK. This alternative can potentially weaken the cryptographic protection of the objects stored in the REE file system of the secure storage. TEEs would greatly benefit from unrestricted access to HUKs and could so improve the protection of secure storage. Additional energy costs can be avoided as long as the system complies with the imposed restrictions of the hardware-assisted protection mechanisms, in particular for Intel SGX.

6. Fault-Tolerance Mechanisms

This chapter gives technical details of the components produced or extended in WP4 concerning fault-tolerance and security. The results presented here are output from the work done in task 4.7. Section 6.1 presents a performance monitoring framework, developed to understand the overhead generated by trusted execution environments. This framework helps in producing meaningful performance metrics and allows to identify the sections in the trusted environments that produce the larger overheads. Section 6.2 presents and assesses our Fault-Tolerant Interface, a library for checkpointing programs that follow a task model. We recall that in Chapter 3 we already presented how this library was integrated in the runtime system.

6.1. Performance Monitoring

To ensure the confidentiality and integrity of applications, we design and implement security mechanisms based on TEE technologies such as Intel SGX, AMD SEV, or ARM TrustZone. However, the security guarantees come with performance overhead [94] [26]. To understand the overhead, we need to collect a set of metrics and extensive statistical data which are able to provide the necessary insights into the inner workings of the system and applications. Therefore, there is an increasing need for extensive research into monitoring, profiling, and optimizing applications running inside TEEs.

Most of the state-of-the-art in this research field has focused on profiling and measuring application performance during the development phase [94] [28] [2]. These tools provide significantly detailed data on the application behavior and performance-critical events during runtime. This can, in turn, provide hints on where major issues and bottlenecks reside in the application code and subsequently aid code improvements and performance optimizations.

Although there is already an extensive amount of visibility into applications, this is currently limited to the development environment, leaving developers blind when the time comes to run their application in production. This makes it harder and more complicated when specific issues and bugs are encountered, as the limited visibility does not allow for proper debugging. In conclusion, while profilers are extremely valuable for gaining a better understanding of an application, most of the time they require source code modifications and re-compilation

which inherently bring a higher performance impact and is not acceptable in production environments.

To overcome these issues, we provide a monitoring and observability framework based on currently available open-source solutions. Our goal is to help users of LEGaTo frameworks to address the following questions:

- What are the main contributors for enclave overheads for a given application?
- What performance metrics help to explain the overheads of applications running inside enclaves?
- Could these metrics give an indication of how to reduce the performance overheads of the applications?
- How can these metrics be visualized to convey the overheads?

6.1.1. Monitoring Framework

We propose a low-overhead monitoring framework which not only provides some of the performance profiling functionality but also allows users to continuously keep track of their applications running with TEEs, e.g., inside SGX enclaves. To our knowledge, no comparable solution exist so far. Our framework offers easy integration into any system given that it is capable of running inside Docker containers. Moreover, it offers extensive visibility into the underlying system metrics by using eBPF programs attached to kernel probes and tracepoints [1], while also providing developers with SGX specific data thanks to our modified SGX instrumented driver. In addition, it allows for further configurability, by easily modifying a configuration file written in YAML format, and offers developers the option to extend the list of metrics provided by instrumenting their own application. Finally, by integrating our monitoring system with Grafana [9], an open-source platform for time-series analytics, we provide rich and meaningful visualization options to get a better understanding on the available metrics data.

Figure 6.1 shows the high-level architecture of our monitoring framework. It consists of a set of standalone components, each playing an explicit role: *(i)* Metrics Aggregation Service (MAS), *(ii)* Metrics Visualization Service (MVS), *(iii)* TEE Metrics Exporter (TME), and *(iv)* System Metrics Exporter (SME). This design allows other components to be added easily if needed, depending on the level of insight required and the requirements of each application.

Metrics Aggregation Service. The main component in our architecture is the Metrics Aggregation Service (MAS). The MAS is a standalone component consisting of a time-series database, a metrics retrieval component and an HTTP server. It is capable of collecting, processing and aggregating a large number of metrics, from a dynamically changing list of services, with low overhead on the applications.

It stores all metrics data samples locally and groups them into chunks for faster retrieval. Additionally, it allows for multi-dimensional data with the help of metric labels specified as a set of key-value pairs. To help developers aggregate the

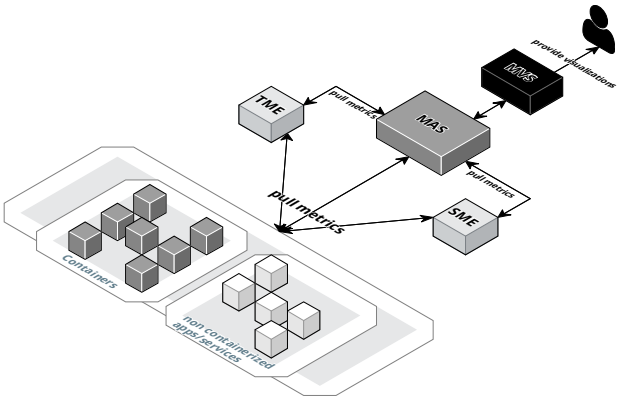


Figure 6.1. The architectural design of the Monitoring framework.

data, it provides support for data queries over specified time ranges and labeled dimensions. It provides detailed quantitative analysis on the data, by selecting and applying aggregation functions to query results. Typically, the MAS is capable of connecting to every kind of service that provides a metrics endpoint so that developers are able to easily add their application metrics to it. Differently from other monitoring systems, it uses a pull approach to collect the data from its targets.

Apart from application metrics, two additional metric components were added to the architecture: (i) **TEE Metrics Exporter:** pulls metrics and statistics from the TEE platform the application is running on. (ii) **System Metrics Exporter:** pulls metrics and statistics from the supporting infrastructure such as CPU, memory, cache, etc.

Both services act as metrics providers and translators by scraping metrics from their specific targets and providing them to the MAS in a standard format. There can be different implementations of these services depending on the specific infrastructure, operating system, platform vendor, etc. We implemented MAS using the open-source tool Prometheus [4]. In addition, we instrumented the Linux SGX driver and implemented the SGX-Exporter as well as wrote several eBPF programs to extract low level performance metrics and exposed them to our metrics engine.

Metrics Visualization Service. Visualizations hold special importance in the monitoring space as it is not an easy task to make sense of time-series metrics just by looking at the raw data. Visualizations also make it easier to spot interesting trends or underlying issues. Additionally, in cases of failures or incidents it helps to limit the data view subset to an interesting time frame. This, in turn, allows for better post-mortem and faster and more effective root cause analysis. But choosing the right type of visual representation is not trivial, as it depends as much on individual preferences as on the metrics one is trying to visualize.

Although the MAS offers data queries and aggregation functions, it does not provide any support for metrics visualization and analytics. Therefore, an external

component is needed, which allows for querying, visualizing and understanding the data. The MVS offers a diverse set of visualization options such as graphs, histograms, gauges, gradient fills, tables, etc. It is also possible to group different metrics together, so that metrics from the same service or serving the same purpose are shown in the same *dashboard*. This makes it easier to correlate between them and recognize familiar patterns.

System Metrics Exporter. The main functionality of the System Metrics Exporter (SME) is to collect and export performance metrics of the underlying system infrastructure.

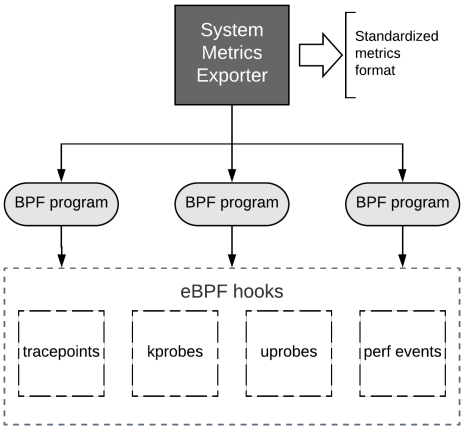


Figure 6.2. System Metrics Exporter architecture.

eBPF programs connect to specified hooks in the underlying software such as the kernel or operating system.

In order to obtain low level system metrics, we would need to access CPU performance counters, kprobes, and tracepoints similarly to the Linux `perf` tool [3]. This requires writing code which gets executed in kernel space and making sure that it has no security or performance impact on the normal kernel execution. This is why we decided to make use of eBPF, the in-kernel virtual machine allowing kernel instrumentation programs to run in a secure and restricted environment.

By attaching small eBPF programs to each kernel hook, we are able to read and extract low-level system statistics. Afterwards, we export them to userspace by using BPF_MAPS. An architecture diagram of this component is shown in Figure 6.2.

The metrics are translated to a standard format understood by the MAS, and published to its metric endpoint to be scraped. To measure the overhead of application running inside enclaves, we instrument system calls, context switches, page faults, and last-level cache metrics.

Next, we provide some experimental measurements taken during the testing of the framework. This helps to evaluate its performance overhead on different applications. We make use of **cAdvisor** [7] to collect performance metrics of containers.

6.1.2. Monitoring Resource Usage

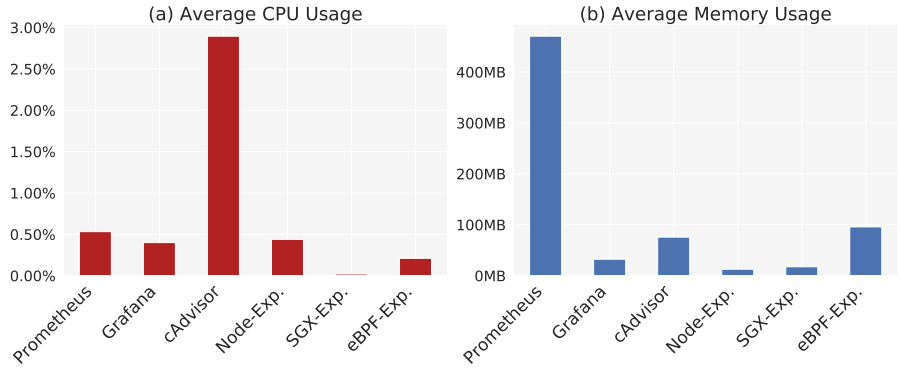


Figure 6.3. 24h-average CPU and Memory consumption of the framework components.

We first present the CPU and memory consumption values of the monitoring framework components averaged across a 24-hour window. To measure the values we used the built-in cAdvisor component, which monitors all Docker container metrics. During the observed 24-hour period all framework components were continuously running and providing system and application metrics during our benchmarking runs.

To understand the overhead of the proposed framework implementation, we measure the CPU and memory utilization of each component separately. The values are split per component in order to better understand their respective overheads and resource usage. As shown in Figure 6.3 (a), all monitoring components exhibit a CPU consumption of less than 3% on average. Interestingly, cAdvisor CPU usage is higher than the other components. We believe this is mainly because of the large number of metrics it provides and can be reduced by setting the `–ignore-metrics` flag. Particularly expensive metrics are the ones relating to disk I/O, UDP, and TCP networking.

Figure 6.3 (b) shows the average memory consumption of each monitoring component. Conversely to the CPU graph, the main resource consumer, in this case, is Prometheus. While all other components stay under 100MB average memory usage, Prometheus uses more than 400MB of system memory. This brings the overall framework memory consumption to around 700MB. However, the higher memory usage of Prometheus is not surprising as by design it keeps all currently used data chunks in memory, as well as most recently used chunks for faster data retrieval. In order to reduce Prometheus' memory usage, one needs to limit its caching allowance by setting the `storage.local.target-heap-size` flag. The default value is 2GB according to the documentation [5]. Depending on host specifications and developer preferences the value can be changed accordingly. It must be said though that higher limits allow for improved performance, especially if we are dealing with high volumes of metric data as it allows to keep more metric chunks in memory.

6.1.3. Application Overheads

Next, we measure the overhead of applications (Redis, Nginx, and MongoDB) running inside enclave when we activate our monitoring framework.

Figure 6.4 shows a summary of the results for all three applications. The appli-

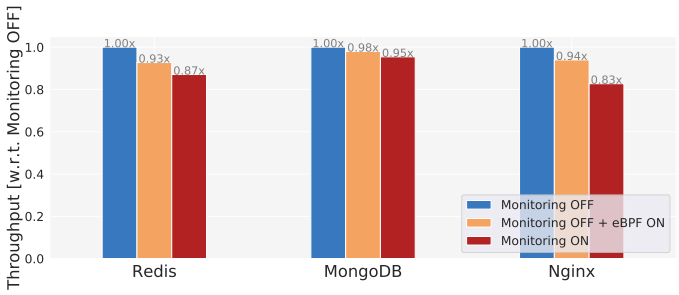


Figure 6.4. Overhead of the monitoring system on the application's throughput.

cation's throughput varies from 83% and 87% of native executions for Redis and Nginx respectively, to 95% for MongoDB. The monitoring framework appears to account for half of that performance drop and the other half comes from the eBPF programs running in the kernel.

We have implemented several eBPF programs attached to frequently used performance counters like cache misses and references, system calls, page faults, and context switches. In some cases like context switches we have instrumented both hardware and software counters with a high sampling frequency and this accounts for some of the added overhead. This extra overhead can be reduced by disabling unnecessary performance counters, reducing sampling frequency for perf software counters or filtering metrics like system calls and context switches to only a specified PID. To facilitate filtering, we provide a macro for some of the programs which can be set in the eBPF configuration file.

Finally, we present several screenshots showing different visualization panels from the Grafana dashboards included in our monitoring framework. Figure 6.5 shows the per-minute rate of page faults happening in kernel and user-space. The top panel takes the data from the eBPF program, while the bottom panel shows the page fault metrics provided by the `VMStat` tool in Linux.

In Figure 6.6, we show different visualization panels relating the EPC and eBPF metrics. The top row consists of several panels showing: (i) total number of enclaves, (ii) active enclaves, (iii) total number of eBPF programs, (iv) status of each eBPF program. Second row includes several counters relating to the EPC pages such as: (i) total number of pages, (ii) free pages, (iii) allocated pages, (iv) pages added to enclaves using EADD instruction, (v) evicted pages (`EWB` instruction), (vi) pages loaded back to the enclaves from main memory (`ELDU` instruction). In the third and last row, we provide a panel showing the per-minute changes of the EPC counters. The visible bump shows how the value changes when we started an instance of Redis running inside an enclave using our SCONE tool chain.

In Figure 6.7, we visualize the per-minute rate for each system call at entry. On the right-hand side of the panels is a table showing the *current* and *total* values for each system call in descending order. The visualization panel for system calls at exit is identical.

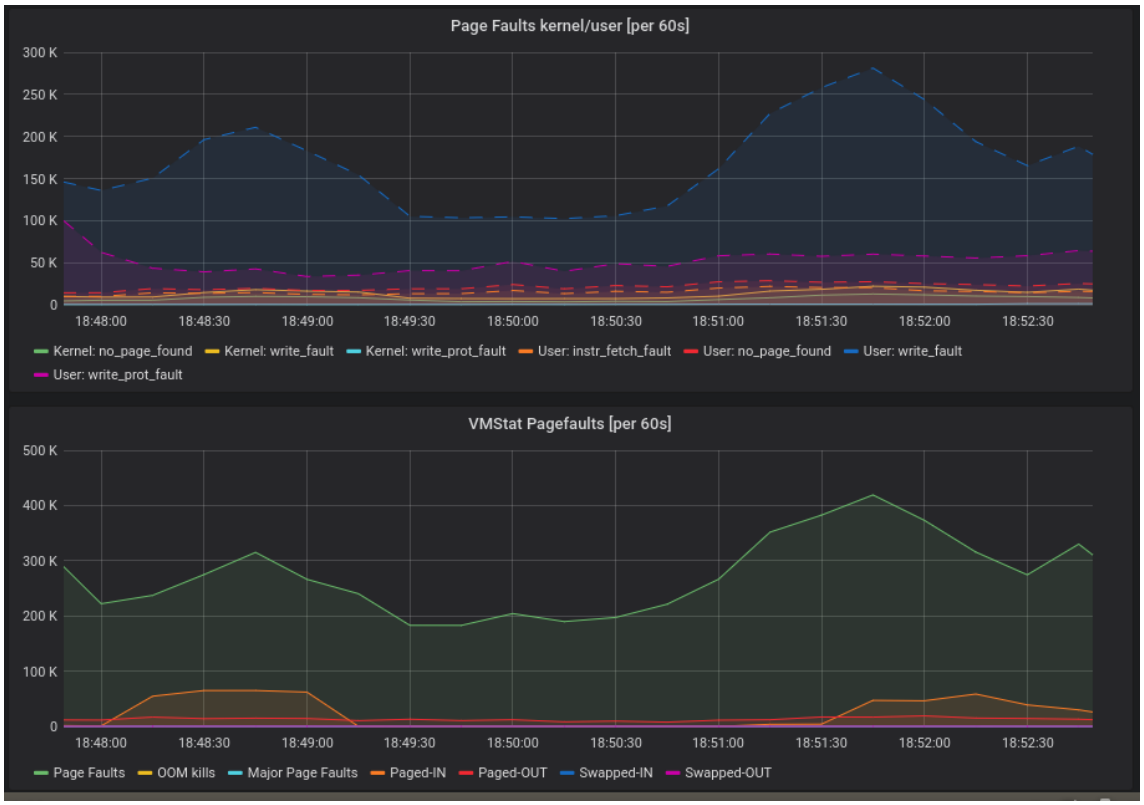


Figure 6.5. Page faults metric visualizations: eBPF metrics (top), VMStat metrics (bottom).

6.2. Design space Exploration of a Fault Tolerant Runtime System

This section presents a case study of integrating a checkpoint library inside a runtime system. We use *LEGaTO*'s checkpoint library called FTI.

Overall the rest of this section provides a brief background on the runtime system we used, on the internals of the checkpoint library and finally an initial assessment of how checkpoint/restart could be integrated inside a runtime system.

Figure 6.6. Enclave Page Cache metric visualization.

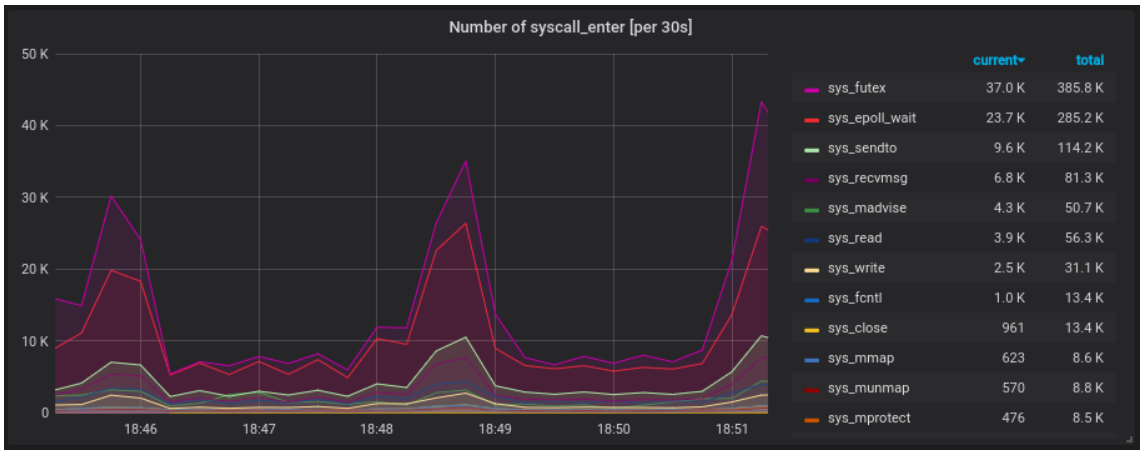


Figure 6.7. System Calls metric visualization.

6.2.1. Runtime (MPC) System Background

MPC [85] is a framework dedicated to the smooth integration of shared-memory parallel programming models in MPI applications. To this end, MPC provides different implementations such as MPI, OpenMP, and Pthread, all unified on top of the same user-level thread scheduler. By having its own MPI implementation and its own thread scheduler, MPC is then able to execute MPI processes in different configurations, as discussed below. One can consider that all MPI implementations fit into one of two categories: process-based and thread-based.

Process-based implementations are based on MPI Processes being regular UNIX processes, with separate address spaces. Most MPI implementations fit in this category, such as MPICH and OpenMPI. An indirect consequence is that applications may feature global variables duplicated for each MPI process running as a UNIX process.

To address this second configuration, MPC relies on a privatizing compiler to transparently separate global variable by creating multiple copies of it for each MPI process, thanks to a hierarchical TLS storage approach [33].

6.2.2. Fault Tolerance Interface Background

FTI is a multilevel checkpointing library with a wide set of features. Writing checkpoints in local storage is sufficient to put up with soft errors but cannot withstand node failures, data stored in the local storage being inaccessible until the node is repaired. Therefore, a local checkpoint has to be combined with some sort of data redundancy to tolerate one or multiple node crashes. For this purpose FTI implements several approaches, which in the context of FTI are called levels, such as data replication on a partner node, data redundancy through Reed-Solomon encoding, or data persistence into the parallel file system. The concept[1] is depicted in Figure 6.8. Level 1 checkpoint is the least reliable level but also the fastest, while Level 4 is the most reliable but also the slowest of all levels. Given that most failures in supercomputers do not affect all nodes simultaneously, there are possibilities to combine the levels to yield improved

[1]The relation between resiliency and C/R overhead is not linear and depends on multiple factors, e.g number of nodes of the system, we only demonstrate the concept of multi-level checkpoints

performance, with in the FTI context.

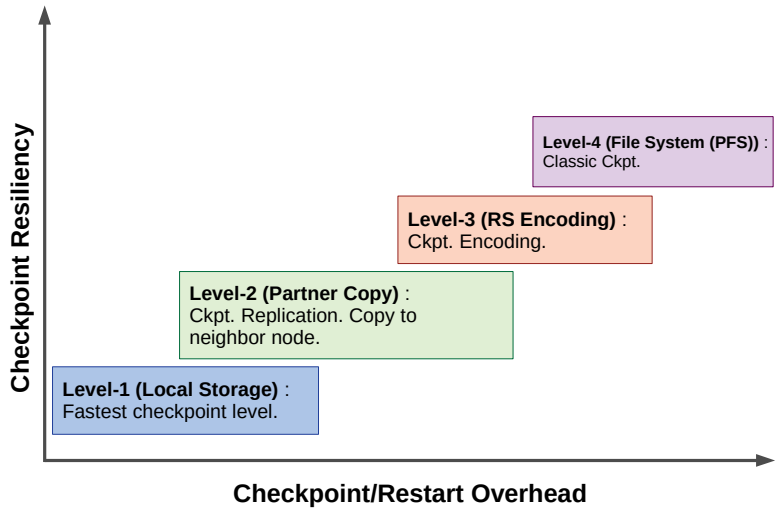
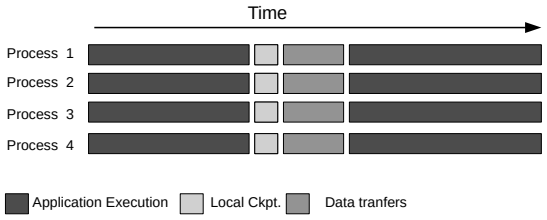


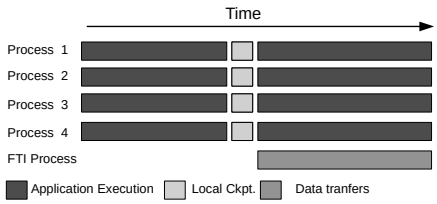
Figure 6.8. Different FTI checkpoint level and the conceptual trade-offs between Resiliency and C/R overhead.

FTI implements the checkpoint procedure into two phases as follows: i) write the checkpoint in local storage ii) post-process the local checkpoint. Post processing typically includes a data redundancy technique, which requires some kind of processing, either by transferring data through the network or by performing extra computations. This additional work can be done locally, impeding extra overhead to the application. As far as this post processing is concerned, FTI offers the option to dedicate one process per node from the application in order to perform the post-processing. In this case, the dedicated processes are isolated from the application processes.

Using this technique, application processes can pursue their execution as soon as the local checkpoint has been made. Data replication is offloaded to these helper processes in parallel to the regular execution. In Figure 6.9 we depict the main concept of the approach. This has been proven to be quite efficient.



(a) MPI Process performs inline checkpoint, all data are moved by the application itself



(b) MPI Process only stores local checkpoint, the data are moved by extra FTI processes

Figure 6.9. Application processes perform the data movements to the correct checkpoint level versus FTI performing the data movements on the background

6.2.3. Integration of FTI in a Modern Runtime

Although, the benefits of asynchronous checkpoints are obvious, as in terms of overhead it virtually transforms all checkpoints into local checkpoints, it is not

always feasible or suggested to assign an entire CPU core to a checkpoint dedicated process. On the one hand, there are limitations depending the characteristics of the application. For example many scientific application which operate on 3D grids require the number of processes to be a cube of naturals number. This restriction, combined with the number of available cores in a specific system might not leave room for extra processes. On the other hand, strong scaling applications can get better performance as the number of available cores increases. It is not advisable to sacrifice in such a case an entire core to perform checkpoints.

For all these reasons, we initially sought to benefit from the runtime specificities. In particular, we moved the dedicated MPI process inside a user-level thread to benefit from oversubscription. However, this is mainly true when the MPI process runs on its own resources. If no core is available, this additional MPI process will be oversubscribed. It means that this additional MPI process shares resources with the original application. As two processes, or even threads, cannot run at the same time on the same core, their respective code will be executed, turn by turn, after context switches.

In order to mitigate oversubscription overhead, we targeted MPC's thread-based MPI capabilities. The interest is twofold with i) lighter context switches and ii) the ability to use MPI waiting time (in the application) to progress checkpointing.

Hence, in a full thread-based mode, each MPI process on a node is a user-level thread managed by a unified scheduler. Switching from one MPI process to another is a user-level thread context switch, which is lighter than between two UNIX processes. This then makes the approach involving an oversubscribed MPI process more attractive than in a regular process-based MPI setup.

6.2.4. FTI Evaluation

This Section now studies the impact of our runtime integration on FTI for application-level checkpointing. In particular, we compare performance between additional MPI processes and our oversubscribed model, taking advantage of runtime threads. For first approach, we ported Lulesh to use FTI.

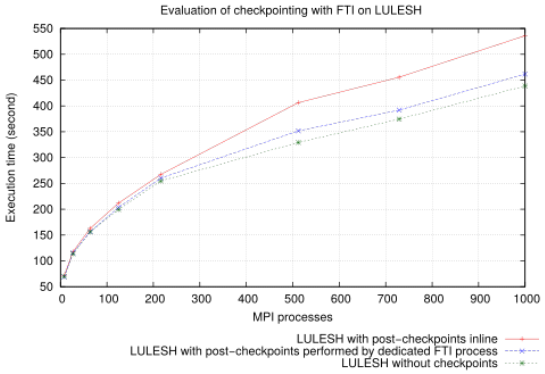
As presented in Figure 6.10a which does not rely on oversubscription, using a dedicated checkpointing process is advantaging when compared to the synchronous approach which does not provide any overlap. This shows that there is an interest in integrating such support through a user-level scheduler. Since *Lulesh* works with numbers of MPI processes which are power of 3, it was not possible to produce a sufficient number of configurations where all cores are loaded with computation.

To effectively test oversubscribed application-level checkpoint restart approach we employed FTI on a heat distribution benchmark (heatdis). Heatdis is a 2D stencil code that distributes a 2D grid among MPI processes. Processes only communicate with neighbor processes for exchanging ghost cells. As this benchmark does not impose restriction on the number of MPI processes (unlike Lulesh), we were able to validate multiple configurations. Performance measurements were taken on the *MareNostrum 3* supercomputer at the Barcelona Supercomputing Center (BSC). *MareNostrum 3* is a 1.1 petaflop peak performance supercomputer

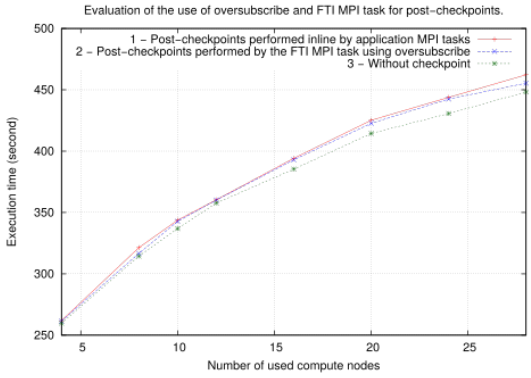
with Intel SandyBridge processors. The machine features 3056 nodes connected through an Infiniband FDR network. As presented in Figure 6.10b, we ran this benchmark in different configurations:

i) Without FTI to provide the base time; ii) with FTI and without dedicated checkpoint threads (called inline post processing); iii) with FTI and a dedicated oversubscribed MPI process (running as a runtime thread).

It can be seen that when relying on a thread to perform post checkpointing operations the overhead is slightly lower than if it was done inline, directly impacting the code. This shows that such a model can lead to some benefits when being used in threads.



(a) Performances without and with FTI checkpointing methods, no oversubscribe



(b) Oversubscribing with an MPC MPI thread

Figure 6.10. Evaluation of the checkpoint overhead when performed inside of the runtime system.

6.2.5. Assessment and Future Work

In this section we presented an initial implementation of a fault tolerant runtime system. The FTI fault tolerance library was smoothly integrated into a runtime system without substantial effort. When the runtime systems use background helper threads (which oversubscribe the system) to perform the data transfers in the background the checkpoint overhead is slightly reduced.

As a next step we plan to integrate FTI inside the Nanos runtime system. In contrast to MPC, the Nanos runtime system through the task annotations can determine dynamically which exact memory locations have changed their values. This information can be exploited at runtime and push data to the checkpoint files incrementally and in parallel with the execution of other tasks. The same reasoning can be applied during recovery, as the data can be read from the checkpoint files as they are requested by the tasks. Hence, FTI does not need to read all the checkpoint data immediately but can incrementally read the requested data. Finally, Nanos has better overview of the utilization of the cores, therefore, the runtime can detect application phases in which there is limited parallel execution. During these phases the runtime system can perform the asynchronous checkpoints, without requiring extra cores or penalise the application execution time.

Given the positive results of this work and the LEGaTO objectives, the FTI library

has been extended to support incremental checkpoint/recovery API functions (described in Deliverable D3.2). These functions will be used in the upcoming integration of FTI with the Nanos runtime system.

7. Conclusion and Next Steps

During the first 20 months of LEGaTO, the tasks of Work Package 4 have been able to produce and release a number of software components. Many of these components were either developed as direct extensions of existing software (as OmpSs or Eclipse).

Chapter 2 has put the work presented in this document in context of the tasks of the work package. Chapter 3 presented extensions for OmpSs compiler and Eclipse integrated development environment. Chapter 4 presented DFiant as a dataflow hardware description language, and OmpSs kernel identification for mapping on Maxeler DFE. Chapter 5 described LEGaTO's work on programming and execution models, integrating CPU and FPGA, energy-efficiency and security trade-offs, and a task-based scheduler. Chapter 6 presented components for fault-tolerance and security.

We will continue to work in tandem with Work Package 3 for the months to come, developing components that are natively integrated. Moving closer to the final months of the project, more attention will be given to integration and support of the use cases. Several specific actions are forecasted for the upcoming months of LEGaTO in each of the running tasks of Work Package 4. We present below a brief summary of these actions.

In Task 4.2, we will work to support scheduling with security combined with heterogeneity- and energy-awareness, add asynchronous calls, enhance interference awareness in XiTAO. In Task 4.3 we will complete integration in Eclipse Che and its compilation environment. In Task 4.4 we will finalize support for FPGAs, advance LLVM prototype and determine its limitations. In Task 4.5 we will improve dynamic DFiant dataflow constructs, integrate with OmpSs, multi-level simulator, and implement a use-case. In Task 4.6 we will continue the integration of MaxJ with OmpSs. Finally, in Task 4.7 we will integrate checkpointing with scheduling and runtime and integrate hardware monotonic counters against rollback attacks.

8. References

[1] eBPF. extended Berkeley Packet Filter. `https://www.iovisor.org/technology/ebpf`.

[2] Intel VTune amplifier. `https://software.intel.com/en-us/vtune`.

[3] Perf. Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`.

[4] Prometheus - monitoring system & time series database. `https://prometheus.io`.

[5] Storage | Prometheus. https://prometheus.io/docs/prometheus/1.8/storage/.

[6] ARM Security Technology: Building a Secure System using TrustZone Technology, April 2009. Accessed on: 02.07.2019.

[7] cAdvisor. https://github.com/google/cadvisor, October 2018. Accessed on: 15/10/2018.

[8] Compute Resource Usage Analysis. https://github.com/kubernetes/heapster, October 2018. Accessed on: 15/10/2018.

[9] Grafana. https://grafana.com/, October 2018. Accessed on: 15/10/2018.

[10] Heapster. https://github.com/kubernetes/heapster, September 2018. Accessed on: 15/10/2018.

[11] InfluxDB. https://www.influxdata.com/time-series-platform/influxdb/, October 2018. Accessed on: 15/10/2018.

[12] kubelet. https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/, October 2018. Accessed on: 15/10/2018.

[13] Linux cpu governors, October 2018. Accessed on: 15.10.2018.

[14] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[15] Inc. Amazon Web Services. Coming Soon: Amazon EC2 C5 Instances, the next generation of Compute Optimized instances. About AWS, What's New (http://amzn.to/2nmIiH9), November 30 2016. Accessed on: 24.06.2019.

[16] Inc. Amazon Web Services. Amazon EC2. https://aws.amazon.com/ec2/, 2018. Accessed on: 24.06.2019.

[17] Amazon Web Services, Inc. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, September 2018. Accessed on: 12.09.2018.

[18] Inc. AMD. AMD EPYC™ Datacenter Processor Launches with Record-Setting Performance, Optimized Platforms, and Global Server Ecosystem Support. https://www.amd.com/en/press-releases/amd-epyc-datacenter-2017jun20, June 2017. Accessed on: 24.06.2019.

[19] Arm Limited. ARM GNU Toolchain. https://developer.arm.com/open-source/gnu-toolchain/gnu-a. Accessed on: 25.06.2019.

[20] Arm Limited. ARM Security Technology: Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_

trustzone_security_whitepaper.pdf, April 2009. Accessed on: 01.07.2019.

[21] Arm Limited. Fundamentals of ARMv8-A. http://tiny.cc/oage7y, March 2017. Accessed on: 02.07.2019.

[22] Arm Limited. *Arm Cortex-A53 MPCore Processor: Technical Reference Manual*, June 2018. DDI 0500J (ID012219).

[23] Arm Limited. Isolation using virtualization in the Secure world. https://developer.arm.com/-/media/Files/pdf/Isolation_using_virtualization_in_the_Secure_World_Whitepaper.pdf?revision=c6050170-04b7-4727-8eb3-ee65dc52ded2, 2018. Accessed on: 01.07.2019.

[24] Arm Limited. Isolation using virtualization in the Secure world. https://developer.arm.com/-/media/Files/pdf/Isolation_using_virtualization_in_the_Secure_World_Whitepaper.pdf?rev 04b7-4727-8eb3-ee65dc52ded2, 2018. Accessed on: 25.06.2019.

[25] Arm Limited. mbed TLS. https://tls.mbed.org, July 2019. Accessed on: 01.07.2019.

[26] 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 *OSDI'16*, pages 689–703, November.

[27] Rishiyur S Nikhil Arvind. Id: A language with implicit parallelism. In *A Comparative Study of Parallel Programming Languages*, pages 169–215. Elsevier, 1992.

[28] Maurice Bailleu, Donald Dragoti, Pramod Bhatotia, and Christof Fetzer. TEE-Perf: A profiler for trusted execution environments. *49th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2019)*, 2019.

[29] Barcelona Supercomputing Center . Transparent Checkpoint Library (TCL). Accessed: 2019-03-24.

[30] Barcelona Supercomputing Center. Mercurium Compiler. Accessed: 2019-03-24.

[31] Jeff Barr. Amazon EC2 Bare Metal Instances with Direct Access to Hardware. http://amzn.to/2jobQuo, November 28 2017. Accessed on: 24.06.2019.

[32] Fabrice Bellard. QEMU. https://www.qemu.org, May 2019. Accessed on: 02.07.2019.

[33] Jean-Baptiste Besnard, Julien Adam, Sameer Shende, Marc Pérache, Patrick Carribault, Julien Jaeger, and Allen D Malony. Introducing task-containers as an alternative to runtime-stacking. In *Proceedings of the 23rd European MPI Users' Group Meeting*, pages 51–63. ACM, 2016.

[34] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX ATC'17*, pages 645–658.

[35] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *Computing Research Repository (CoRR)*, abs/1802.09085, February 2018.

[36] Chernousov, Andrei. Free CFD Source Codes. Accessed: 2019-03-24.

[37] CINECA. High Temperature Superconductivity. Accessed: 2019-03-24.

[38] Intel Corporation. Intel SGX SDK for Linux. https://01.org/intel-softwareguard-extensions, 2018. Accessed on: 01.07.2019.

[39] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[40] Adrian Cristal, Osman S. Unsal, Xavier Martorell, Paul Carpenter, Raul De La Cruz, Leonardo Bautista, Daniel Jimenez, Carlos Alvarez, Behzad Salami, Sergi Madonar, Miquel Pericàs, Pedro Trancoso, Micha vor dem Berge, Gunnar Billung-Meyer, Stefan Krupop, Wolfgang Christmann, Frank Klawonn, Amani Mihklafi, Tobias Becker, Georgi Gaydadjiev, Hans Salomonsson, Devdatt Dubhashi, Oron Port, Yoav Etsion, Vesna Nowack, Christof Fetzer, Jens Hagemeyer, Thorsten Jungeblut, Nils Kucza, Martin Kaiser, Mario Porrmann, Marcelo Pasin, Valerio Schiavoni, Isabelly Rocha, Christian Göttel, and Pascal Felber. Legato: Towards energy-efficient, secure, fault-tolerant toolset for heterogeneous computing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, CF '18, pages 276–278, New York, NY, USA, 2018. ACM.

[41] Barb Darrow. Google Is First in Line to Get Intel's Next-Gen Server Chip. Fortune (http://fortune.com/2017/02/24/google-intel-cloud-chip/), February 24 2017. Accessed on: 24.06.2019.

[42] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 295–306, February 2003.

[43] GlobalPlatform, Inc. *TEE Client API Specification Version 1.0*, July 2010. GPD_SPE_007.

[44] GlobalPlatform, Inc. *TEE Sockets API Specification Version 1.0.1*, January 2017. GPD_SPE_100.

[45] GlobalPlatform, Inc. *TEE Internal Core API Specification 1.1.2.50*, June 2018. GPD_SPE_100.

[46] GlobalPlatform, Inc. *TEE Internal Core API Specification Version 1.2*, October 2018. GPD_SPE_010.

[47] GlobalPlatform, Inc. *TEE System Architecture Version 1.2*, November 2018. GPD_SPE_009.

[48] GlobalPlatform, Inc. GlobalPlatform Homepage. `https://globalplatform.org`, July 2019. Accessed on: 01.07.2019.

[49] Google LLC. Google Compute Engine Pricing. `https://cloud.google.com/compute/pricing`, September 2018. Accessed on: 12.09.2018.

[50] Google LLC. Android Trusty. https://source.android.com/security/trusty, June 2019. Accessed on: 25.06.2019.

[51] Shay Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive*, 2016(204):1–14, 2016.

[52] John R. Gurd, Chris C. Kirkham, and Ian Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

[53] Shai Halevi and Victor Shoup. Design and Implementation of a Homomorphic-Encryption Library. Technical report, IBM Research, Yorktown Heights, NY, USA, November 30 2012. Accessed on: 01.07.2019.

[54] HEXUS.net. ARM Everywhere. https://hexus.net/static/arm-everywhere/. Accessed on: 25.06.2019.

[55] IBM. Bare metal servers. `https://www.ibm.com/cloud/bare-metal-servers`, September 2018. Accessed on: 12.09.2018.

[56] David Kaplan. Protecting VM Register State with SEV-ES. https://support.amd.com/TechDocs/ProtectingFebruary 17 2017. Accessed on: 24.06.2019.

[57] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption. AMD Developer Central, April 21 2016.

[58] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[59] Dmitrii Kuvaiskii, Somnath Chakrabarti, and Mona Vij. Snort Intrusion Detection System with Intel Software Guard Extension (Intel SGX). *arXiv e-prints*, 1802.00508:1–21, February 2018.

[60] Paul Le Guernic, Albert Benveniste, Patricia Bournai, and Thierry Gautier. Signal–a data flow-oriented language for signal processing. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 34(2):362–374, 1986.

[61] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8, 2010.

[62] Kevin Lepak, Gerry Talbot, Sean White, Noah Beck, and Sam Naffziger. The Next Generation AMD Enterprise Server Product Architecture. https://www.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.22-Tuesday-Pub/HC29.22.90-Server-Pub/HC29.22.921-EPYC-Lepak-AMD-v2.pdf, August 2017. Accessed on: 24.06.2019.

[63] Linaro Limited. Trusted Firmware. https://www.trustedfirmware.org/. Accessed on: 25.06.2019.

[64] Linaro Limited. OP-TEE Sanity Testsuite. https://github.com/OP-TEE/optee_test/tree/3.2.0, June 2018. Accessed on: 02.07.2019.

[65] Linaro Limited. Linaro Homepage. https://www.linaro.org/, June 2019. Accessed on: 01.07.2019.

[66] Linaro Limited. Open Portable Trusted Execution Environment. https://www.op-tee.org, May 2019. Accessed on: 25.06.2019.

[67] Linaro Limited. Open Portable Trusted Execution Environment. https://www.op-tee.org, May 2019. Accessed on: 01.07.2019.

[68] Linaro Limited. Secure Storage in OP-TEE. https://optee.readthedocs.io/architecture/secure_storage.html, May 2019. Accessed on: 01.07.2019.

[69] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic Application Partitioning for Intel SGX. In *USENIX ATC'17*, pages 285–298, July 2017.

[70] Google LLC. Kubernetes Engine. https://cloud.google.com/kubernetes-engine/, 2018. Accessed on: 24.06.2019.

[71] Stefano Markidis, Giovanni Lapenta, and Rizwan-uddin. Multi-scale simulations of plasma with ipic3d. *Mathematics and Computers in Simulation*, 80(7):1509 – 1519, 2010. Multiscale modeling of moving interfaces in materials.

[72] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan. Open-tee – an open virtual trusted execution environment. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 400–407, Aug 2015.

[73] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proc. Hardware Architectural Support Security Privacy 2016*, HASP '16, pages 10:1–10:9, New York, NY, USA, June 2016. ACM.

[74] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. Modelling performance & resource management in kubernetes. In *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, pages 257–262. IEEE, 2016.

[75] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[76] Microsoft Corporation. Pricing - Linux Virtual Machines. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/, September 2018. Accessed on: 12.09.2018.

[77] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proc. 11th Eur. Workshop Syst. Security*, EuroSec '18, pages 1:1–1:6, New York, NY, USA, May 2018. ACM.

[78] DE Muller and WS Bartky. A theory of asynchronous circuits ii. *Digital Computer Laboratory*, 78, 1957.

[79] Raymond H Myers and Raymond H Myers. *Classical and modern regression with applications*, volume 2. Duxbury press Belmont, CA, 1990.

[80] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can Homomorphic Encryption Be Practical? In *Proc. 3rd ACM Workshop Cloud Comput. Security Workshop*, CCSW '11, pages 113–124. ACM, 2011. Chicago, IL, USA.

[81] NVIDIA Corporation. TLK Repository. http://nvtegra.nvidia.com/gitweb/?p=3rdparty/ote_partner/tlk.git, October 2015. Accessed on: 25.06.2019.

[82] Open Mobile Terminal Platform. Advanced Trusted Environment: OMPT TR1. `http://www.omtp.org/OMTP_Advanced_Trusted_Environment_OMTP_TR1_v1_1.pdf`, May 2009. Accessed on: 02.07.2019.

[83] Oracle Corporation. Bare Metal Cloud Computing. `https://cloud.oracle.com/compute/bare-metal/features`, September 2018. Accessed on: 12.09.2018.

[84] Siani Pearson and Azzedine Benameur. Privacy, Security and Trust Issues Arising from Cloud Computing. In *2010 IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, CloudCom '10, pages 693–702. IEEE Computer Society, November 2010. Indianapolis, IN, USA.

[85] Marc Pérache, Hervé Jourdren, and Raymond Namyst. Mpc: A unified parallel runtime for clusters of numa machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, pages 78–88, Berlin, Heidelberg, 2008. Springer-Verlag.

[86] Mark Russinovich. Introducing Azure Confidential Computing. Microsoft Azure Blog (https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/), September 14 2017. Accessed on: 24.06.2019.

[87] Scaleway. BareMetal SSD Cloud Servers. `https://www.scaleway.com/baremetal-cloud-servers/`, September 2018. Accessed on: 12.09.2018.

[88] Brijesh Singh. x86: Secure Encrypted Virtualization (AMD). https://lwn.net/Articles/716165/, March 2 2017. Accessed on: 24.06.2019.

[89] Ghislaine Thuau and Daniel Pilaud. Using the Declarative Language LUSTRE for Circuit Verification. pages 313–331. Springer London, 1991.

[90] Trustonic. Trustonic Kinibi. https://www.trustonic.com/markets/iot, June 2019. Accessed on: 25.06.2019.

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

[92] Sébastien Vaucher, Valerio Schiavoni, and Pascal Felber. Stress-SGX: Load and Stress your Enclaves for Fun and Profit. In *Networked Systems*, NETYS '18, Cham, Switzerland, May 2018. Springer International Publishing.

[93] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: trusted execution environments on GPUs. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 681–696, 2018.

[94] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. Sgx-perf: A performance analysis tool for intel sgx enclaves. In *Proceedings of the 19th International Middleware Conference*, pages 201–213, 2018.