FRED Framework Documentation

Release 0.1

ReTiS Lab, Scuola Sant'Anna, Pisa, Italy.

Jun 11, 2022

CONTENTS

FRED is a framework to support the design, development, and execution of *predictable* software on FPGA system-on-chips platforms.

It exploits **dynamic partial reconfiguration** and recurrent execution to virtualize the FPGA fabric, enabling the user to allocate a larger number of hardware accelerators than could otherwise be fit into the physical fabric. It integrates *automated floorplanning* and a set of *runtime mechanisms* to enhance predictability by scheduling hardware resources and regulating bus/memory contention.

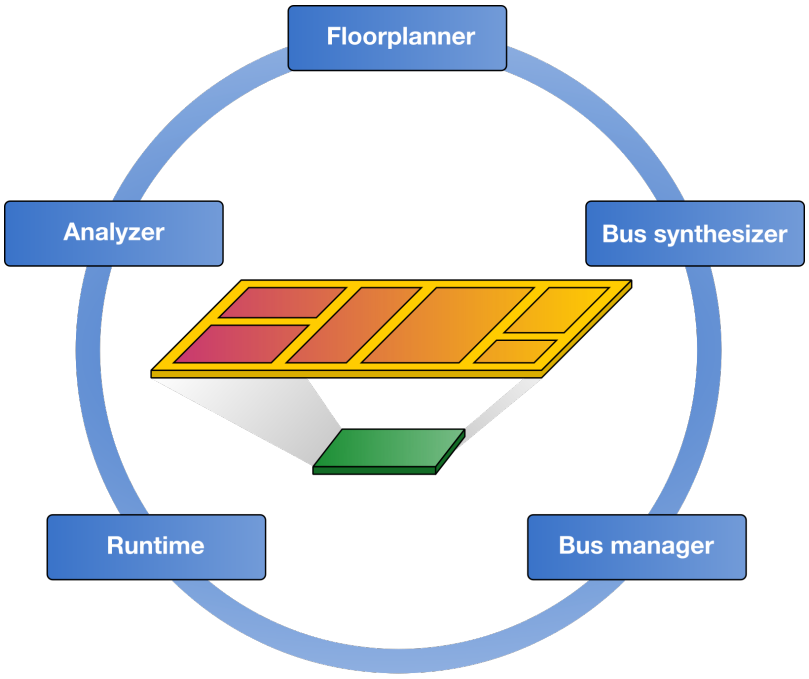


Fig. 1: FRED workflow

The *FRED workflow* figure shows the 5 pillars of FRED framework: floorplanner, bus synthesis, bus manager, runtime, and the analyzer. As presented next, the current focus is on the automatic floorplanner and FRED runtime. The other items are to be included into the framework in the future.

The project is developed at the Real-Time Systems Laboratory (RETIS Lab) of the Scuola Superiore Sant'Anna of Pisa.

This document is organized as follows. First, it presents an introduction to the framework. The five following sections describe the five parts presented in the *FRED workflow* figure. Next, the *getting started* section guides through the framework installation and setup. The following section presents some case studies, a research roadmap, a complete list of publications, and additional project-related information.

CHAPTER

ONE

INTRODUCTION TO THE FRED FRAMEWORK

FRED is a *framework* to support the design, development, and execution of *predictable software* on FPGA system-on-chips that include both general-purpose processors and an FPGA fabric, sharing a common memory. It exploits *dynamic partial reconfiguration* and recurrent execution to *virtualize* the FPGA fabric, enabling the user to allocate a larger number of hardware accelerators than could otherwise be fit into the physical fabric. It integrates automated *floor-planning* and a set of *runtime mechanisms* to enhance *predictability* by scheduling hardware resources and regulating bus/memory contention.

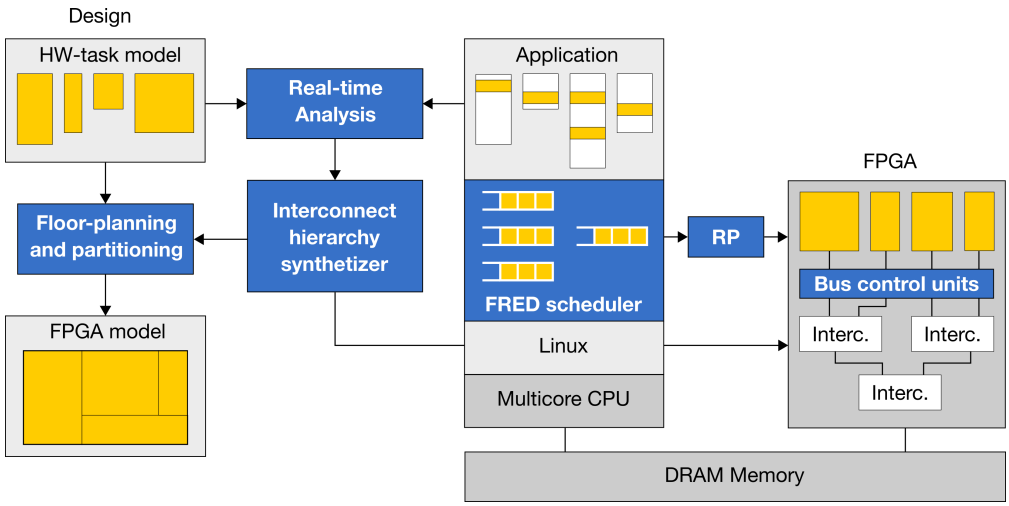


Fig. 1: FRED framework schema

The *FRED framework schema* illustrates the overall architectural schema of FRED framework. The following set of tools are currently available:

- *DART design tool*: a floorplanner that optimizes the allocation of hardware accelerators on the FPGA fabric and generates optimized partial bitstreams for dynamic partial reconfiguration;
- *FRED Runtime*: a Linux service for managing hardware acceleration requests executing on a virtualized FPGA fabric with predictable response times.

The *FRED framework schema* also describes other parts of the FRED framework still under development. Please refer to the Research Roadmap.:

- *FRED Analyzer*: a schedulability analysis tool that verifies whether a set of real-time tasks and hardware accelerators can be executed within their timing constraints, considering all the sources of delays introduced by the architecture.
- *Bus Manager*: a bus control unit that allows achieving predictable arbitration, protection from timing attacks, and bandwidth isolation to shield the system from misbehaving accelerators. This block is accompanied by an

automatic synthesis tool that optimizes the interconnect hierarchy to match timing constraints.

- *Bus Synthesis*: **TODO**

The FRED framework manages two kinds of computational tasks:

- *software tasks* (SW-tasks) are computational activities running on the processors; and
- *hardware tasks* (HW-tasks) are hardware accelerators programmed to execute on the FPGA fabric.

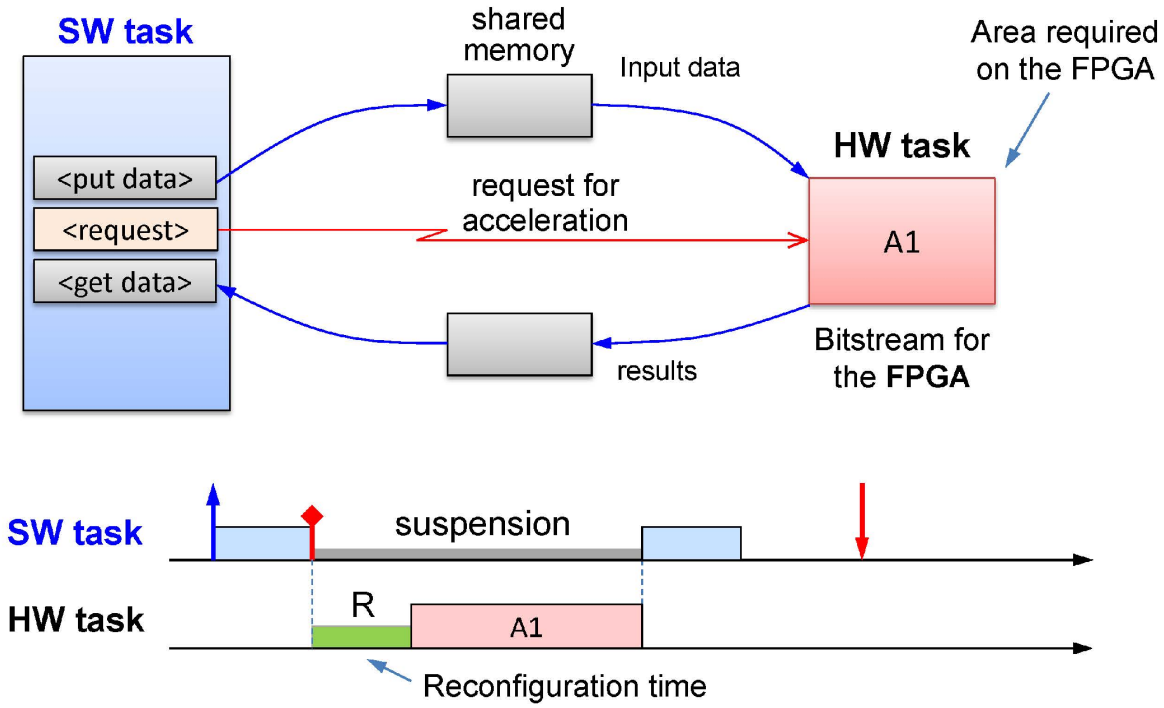


Fig. 2: FRED programming model

The interaction between SW-tasks and HW-tasks is illustrated in the *FRED programming model* figure. SW-tasks can request the execution of HW-tasks to accelerate specific computations. In design time, the set of HW-tasks used by the application(s) is synthesized, mapped, and floorplanned by *DART design tool*, such that their corresponding bitstreams are managed in runtime by *FRED Runtime*. In runtime, the SW-tasks issue acceleration requests managed by the *FRED Runtime* to ensure that they are served with a predictable and bounded delay. A shared-memory communication paradigm with blocking synchronization is employed between SW-tasks and HW-tasks. Before requesting an acceleration, a SW-task must fill a buffer with the input data to be processed by the HW-task. The execution of the SW-task is then suspended when the acceleration request is issued. Next, the corresponding HW-task is programmed on the FPGA. The HW-task autonomously accesses the shared memory to retrieve the input data and to store the output data is produced. Finally, once the acceleration request is completed, the SW-task is resumed and can access the output data produced by the HW-task in the shared memory.

SW-tasks are scheduled by the operating system that controls the processors. To ensure predictability in scheduling SW-tasks, FRED mandates the use of partitioned fixed-priority scheduling (each SW-task is statically allocated to a processor and assigned of a static priority).

CHAPTER

TWO

DART DESIGN TOOL

2.1 Introduction

The FRED framework goal is to accelerate in FPGA a set of functions of a given application. In the proposed *DART design flow*, each function must match with a dart-compatible hardware IP. DART main input is a set of IP cores synthesized for partial bitstreams. The repository dart_ips presents a set of IPs ready to be used by DART. The second set of inputs is the Hw task partitions into reconfigurable regions or the Hw task timing constraints, depending on the mode DART is compiled.

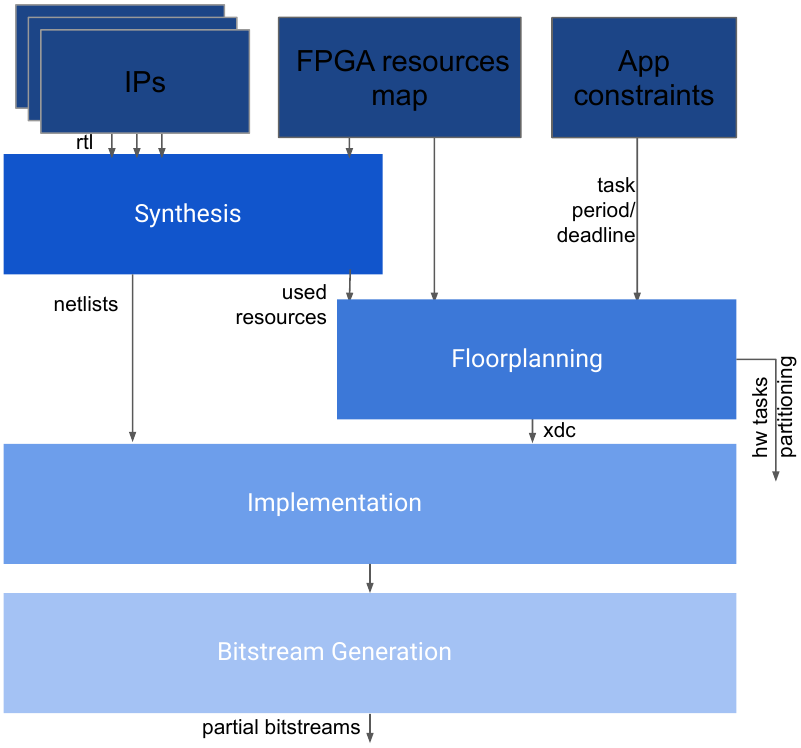


Fig. 1: DART design flow

The *FPGA resource map* input represents the FPGA internal organization that DART must know to perform its automated floorplanning optimization process. DART currently supports the following FPGA boards, where ZCU102 is the latest tested board:

- Pynq board: with device XC7Z020-1CLG400C;

- Zybo board: with the device XC7Z010-1CLG400C;
- ZCU102 board: with Zynq UltraScale+ XCZU9EG-2FFVB1156 MPSoC;
- Ultra96v2 board: with Zynq UltraScale+ XCZU3EG-SBVA484-1-i MPSoC;

DART is currently integrated with Xilinx Vivado 2020.2 for FPGA synthesis. The DART repository provides its source code, documentation, and additional examples.

Finally, DART generates a bitstream for the static part of the FPGA design and a set of partial bitstreams for each Hw task. In addition, DART also generates CSV configuration files describing the Hw task mapping into the FPGA floorplan. These configuration files are later read by FRED when the design is loaded in runtime.

2.2 The FLORA floorplanner

FLORA is the floorplanner used in DART to optimize the allocation of hardware accelerators on the FPGA fabric. The problem of floorplanning for dynamic partial reconfiguration consists in geometrically placing reconfigurable regions (RRs) within the available area of the FPGA. Producing a feasible floorplan means satisfying the resource requirements of the RRs while respecting the specific vendor-related technological constraints. In addition to feasibility, an optimal floorplan minimizes some performance metrics such as FPGA resource consumption or the maximum wire-length between RRs.

The resource requirement of RRs consists in ensuring that each RR must contain at least the maximum amount of resources required by all the reconfigurable modules that it hosts. Some of the vendor-related constraints on the Xilinx FPGAs include

- RRs must be rectangular;
- Vertical boundaries of RRs must not split back-to-back interconnect tiles;
- Horizontal boundaries of RRs must be aligned with clock regions if the reconfigurable module is to be reset after reconfiguration;
- RRs must not include some components of the FPGA (for 7 series devices this includes clock modifying logic, I/O related components, ICAP, XADC, etc;).

One of the major challenges in floorplanning automation is to adequately model the *non-uniform* distribution of *heterogeneous* resources on the fabric, since the model is tightly coupled with the definition of the aforementioned constraints.

FLORA solves the floorplanning automation problem and produces optimal floorplans for RRs by leveraging Mixed-Integer Linear Programming (MILP) optimization based on a fine-grained layout model of the computing resources (i.e., CLBs, BRAMs and DSPs) and system resources (i.e., interconnects, different functional blocks) on the FPGA.

The resource distribution *fingerprint* is key to the fine-grained resource layout model in FLORA. The resource fingerprint is generated by overlaying a 2D discrete Cartesian coordinate system on the FPGA fabric whose origin is at the bottom-left corner. Each unit on the x-axis denotes a column of resources (CLB, BRAM, DSP, interconnects, central clock column), while each unit on the y-axis represents a single clock region that is fused with the horizontally adjacent clock regions. The resource fingerprint in FLORA, as illustrated in *FLORA floorplanner*, is the representation of the resources in the first clock region with a piece-wise constant function. It also contains the locations of all the forbidden components on the fabric.

The inputs to FLORA are the FPGA resource fingerprint, the resource requirements of the RRs, and the parameters the designer wants to optimize. Inside FLORA, the resource fingerprint and the resource requirements are translated into a set of MILP constraints and solved using a solver.

As illustrated in *FLORA flow* figure, the FLORA output is a constraint file (e.g., an `.xdc` file for Xilinx Vivado) that describes the layout of each RR according to the syntax specified by the design tool provided by the vendor. Before generating the final constraint file, FLORA provides an additional visualization tool that allows the designer to inspect the generated floorplan.

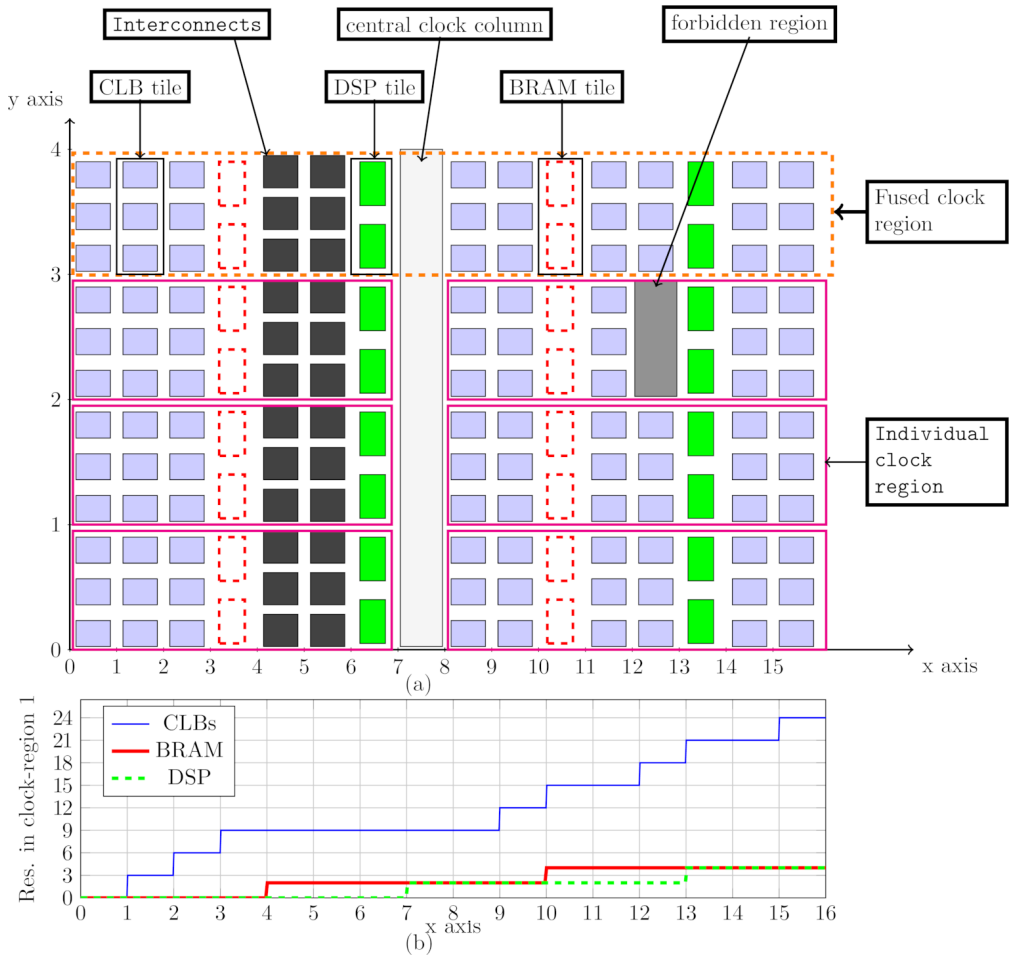


Fig. 2: FLORA floorplanner

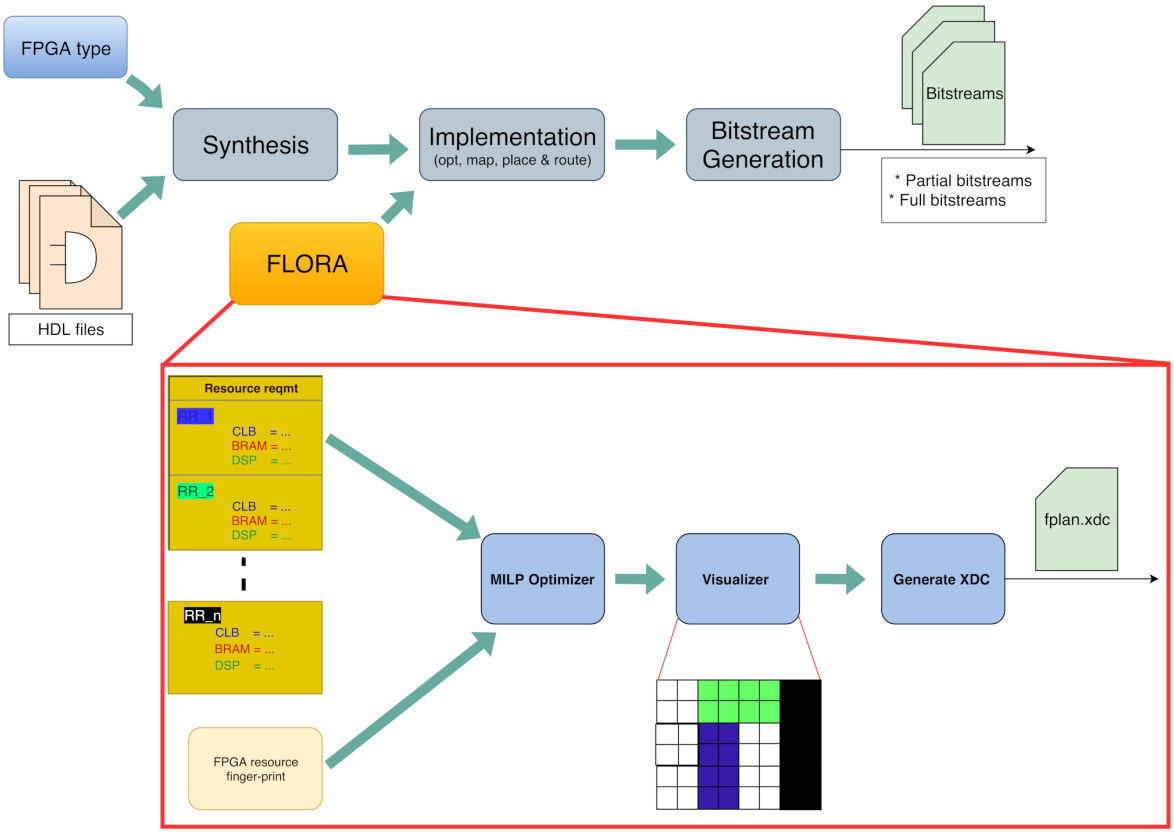


Fig. 3: FLORA flow

2.3 Testing DART

DART is available for download. Further instructions can be found in DART repository and in the Getting Started section.

2.4 Reference

- Biruk Seyoum, Alessandro Biondi, and Giorgio Buttazzo, FLORA: FLoorplan Optimizer for Reconfigurable Areas in FPGAs, ACM Transactions on Embedded Computing Systems. Presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2019), New York, USA, October 13 - 18, 2019.

CHAPTER

THREE

FRED RUNTIME

The *FRED runtime* is the reference implementation of the FRED framework for the GNU/Linux operating system. It has been designed to run on the *Xilinx Zynq-7000 and Zynq UltraScale+ SoC FPGAs platforms*. The FRED runtime consists of a *system support design*, automated by DART, and a set of *software support* components.

3.1 System support design

The *FRED support design* is a reference design for the FPGA side of the SoC that has been designed to support the deployment of dynamically-reconfigured hardware accelerators. The support design divides the FPGA into two regions: a *static region* and a *reconfigurable region*. The static region contains the logic needed to realize the communication infrastructure, namely a set of AXI Interconnects, which the user can extend by adding other support modules depending on the specific needs. The reconfigurable region is organized into a set of statically defined slots that are logically grouped into partitions.

3.2 Software support

The *FRED software support* comprises software components in charge of managing the FPGA and implementing the FRED scheduling policy on top of the system support design. The software support has been designed in a modular fashion, relying as much as possible on user space implementation to improve maintainability, safety, and expandability. The central component of the software support is a user-space server process, named the *FRED server*, which manages acceleration requests from Linux processes (and threads) according to the FRED scheduling policy. Linux processes and hardware accelerators share data through a zero-copy mechanism implemented using physically contiguous (uncached) memory buffers. The FRED server relies on two custom kernel modules and the UIO framework for controlling the hardware accelerators.

The *FRED software stack* figure introduces FRED runtime components. At the **application level** we see that it possible to write applications with C/C++/Python programming languages. It is also possible to write applications using ROS2 and Xilinx Vitis AI frameworks (*both currently under development*). Still in **user space**, we have the fred_lib, which is linked with the application to have access to the fred_server.

The FRED server initiates the FPGA support during the initialization phase and then manages Linux processes and threads requests. Internally, the FRED server uses I/O multiplexing to monitor all hardware and software component events from a single event loop. The FRED server communicates with the software processes using a simple client-server messaging protocol based on Unix domain socket. From a user perspective, the interactions between the software process and the FRED server are abstracted by fred_lib, which is available in C and Python.

In kernel space, two Linux kernel modules, called fred_buffctl and fpga_mgr, were developed/modified to abstract the access to the FPGA fabric. The *fred_buffctl* module is used to allocate the contiguous memory buffers used to share data between software processes and dynamically-reconfigured hardware accelerators. The *fpga_mgr* module manages the device reconfiguration in an optimized way with respect to the Xilinx's stock driver.

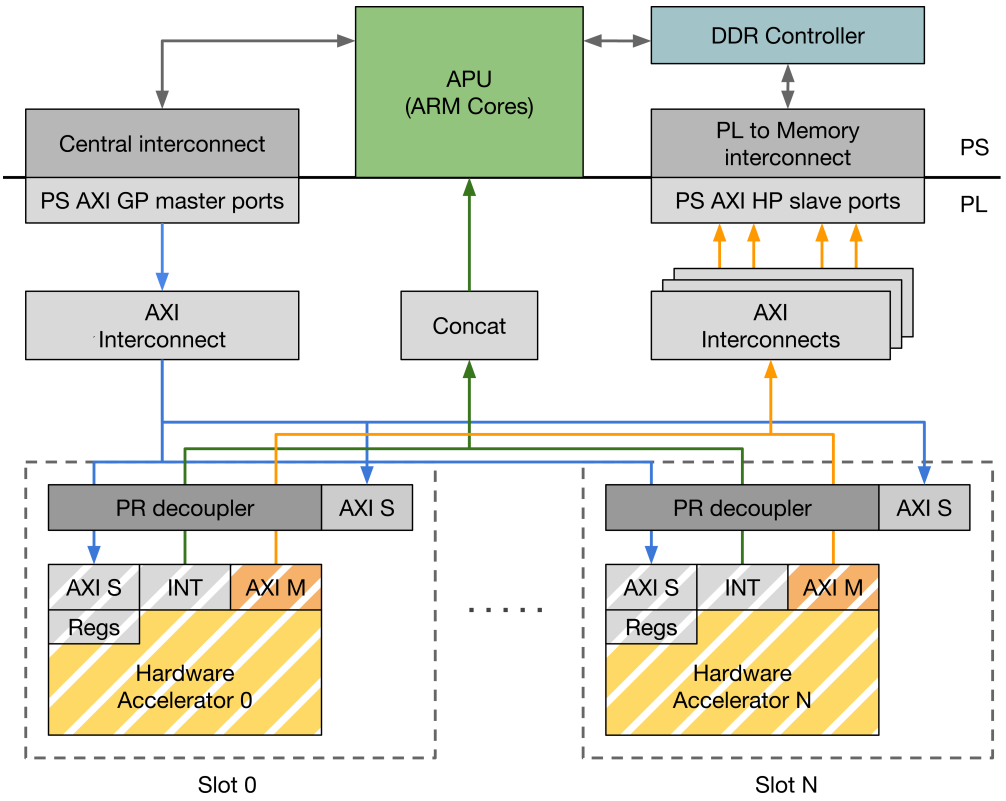


Fig. 1: FRED support design

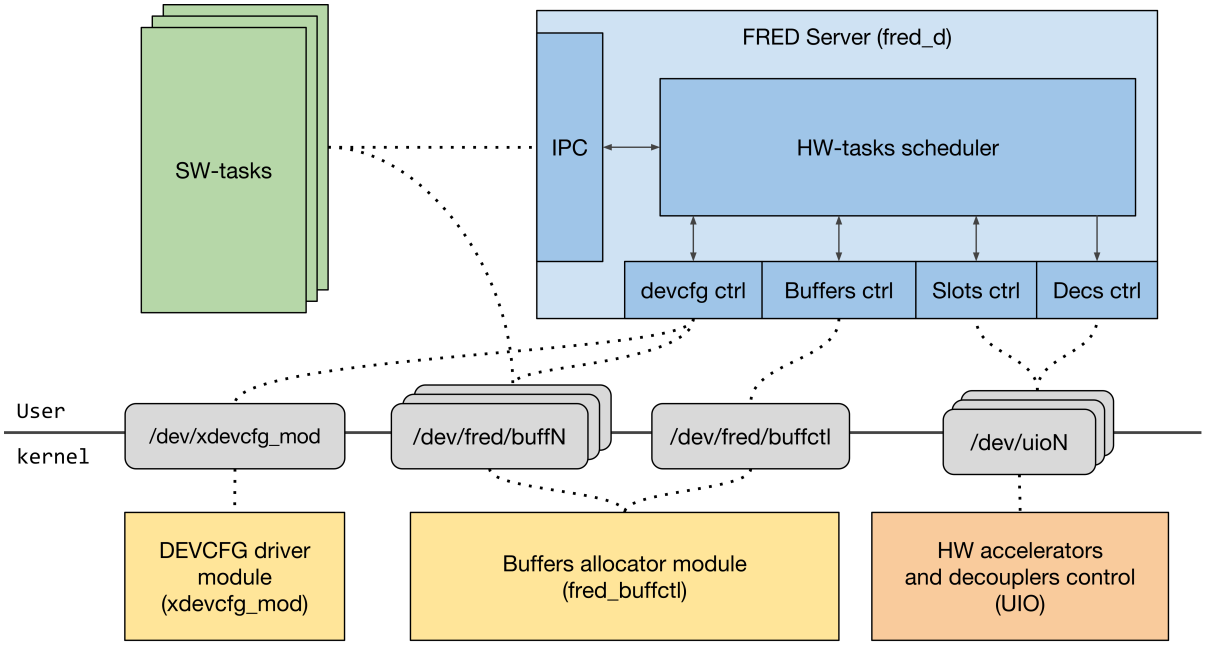


Fig. 2: FRED software support

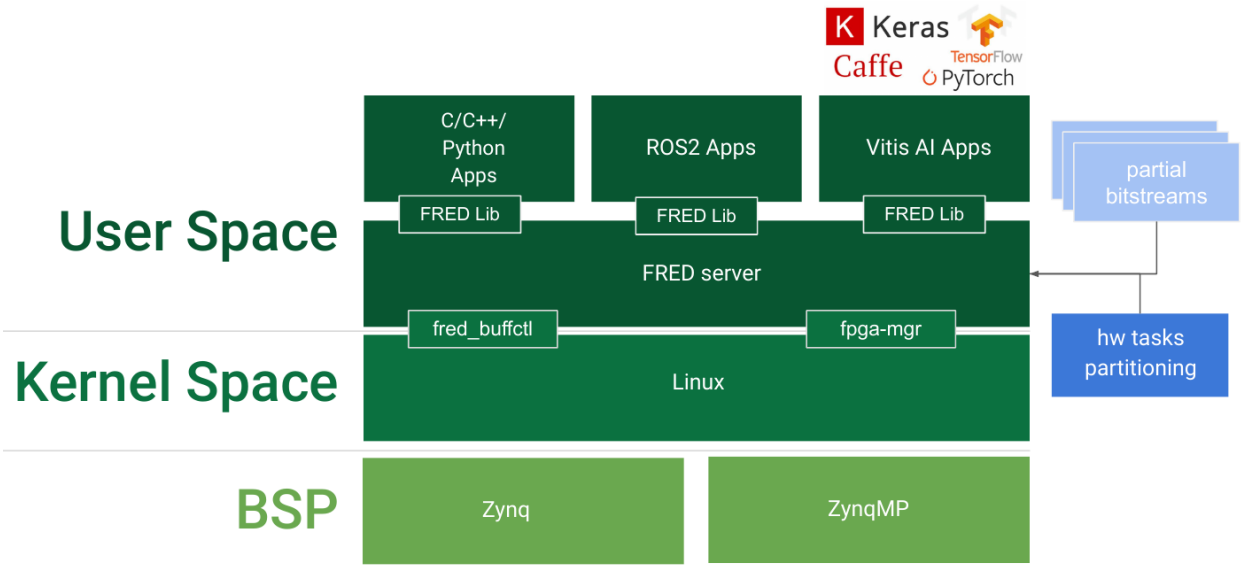


Fig. 3: FRED software stack

3.3 OS Support

It is necessary to have a Linux image with the basic setup and requirements to run the FRED Framework. For this, we created two Yocto layers called meta-fred and meta-retis that can be used with **Petalinux v2020.2** to create the tested Linux image. *meta-fred* cross-compile all software part of FRED runtime. Using this is Yocto layer is the recommended way to compile FRED runtime. Alternatively, *meta-retis* provides all the Kernel setup and software compilation/analysis/debugging tooling to compile and test FRED software directly in the board. *meta-retis* also provides setup for running/debugging real-time Linux systems using, for instance, PREEMPT-RT, perf, ftrace, stress-ng, etc. So, the user can decide which approach to use, but the recommendation is to use both Yocto layers for an increased and more flexible design/debug capability. Finally, fred-framework is a meta repository that combines all FRED Framework, facilitating compilation in the board;

3.4 Testing FRED runtime

FRED runtime is available for download. Further instructions can be found in FRED repository and in the Getting Started section.

3.5 Reference

- M. Pagani, A. Balsini, A. Biondi, M. Marinoni, and G. Buttazzo, A Linux-based Support for Developing Real-Time Applications on Heterogeneous Platforms with Dynamic FPGA Reconfiguration, Proc. of the 30th IEEE Int. System-on-Chip Conference (SOCC 2017), Munich, Germany, September 5-8, 2017.

CHAPTER

FOUR

FRED ANALYZER

FRED-analyzer is a tool to analyze the timing properties of applications running upon the FRED framework.

The tool offers the following features:

- it provides analytical bounds on the maximum delay that can be experienced by a SW-task when requesting the execution of a hardware accelerator;
- it allows computing an upper-bound on the worst-case response time of a SW-task that requests the execution of hardware accelerators, and runs together with other SW-tasks under preemptive fixed-priority scheduling; and
- it allows bounding the worst-case delay experienced by hardware accelerators when accessing a shared memory through a series of hierarhically-connected bus interconnects (required to bound their worst-case execution times). The analysis can also take into account the presence of the predictable bus manager.

The tool is capable of testing different configurations of the system to perform a design space exploration. Specifically, it allows controlling:

- the partitioning of the FPGA fabric and the affinities of the hardware accelerators;
- the priorities and the deadlines of SW-tasks;
- the structure of the bus hierarchy and its connections to hardware accelerators; and
- the configuration of the predictable bus manager (transaction budget, nominal burst size, and maximum number of outstanding transactions).

4.1 Reference

- Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs, Proc. of the IEEE Real-Time Systems Symposium (RTSS 2016), Porto, Portugal, Nov. 29 - Dec. 2, 2016.

CHAPTER

FIVE

BUS MANAGER

This section presents the current development related to the *bus manager*. Currently, the *bus manager* is still not integrated into DART. Section *System support design* explains the current DART bus architecture. The rest of this chapter explains the *future* (i.e. to be integrated) FRED framework bus infrastructure, consisting of the following blocks.

5.1 AXI HyperConnect

Many Cyber-Physical systems are safety-critical and must undertake a certification process, at least for their critical subsystems. A key requirement for certifying safety-critical systems is timing predictability. This is particularly challenging for FPGA SoC, especially when considering that the hardware accelerators can experience bus/memory contention that can severely affect their timing performance. Furthermore, isolation capabilities are required to avoid propagating faults across subsystems and to bound contention delays generated by low-criticality untrusted subsystems.

Virtualization via hypervisor technologies is an established industrial practice for the co-existence of subsystems with mixed-criticality on the same platform while enforcing isolation. However, new challenges arise when virtualization is applied to FPGA SoC. Indeed, isolation must not only be ensured for software components running on the processors (as done by most hypervisors) but also for hardware accelerators (HAs) belonging to different subsystems that are jointly deployed on the same FPGA fabric. Furthermore, bus arbitration and the corresponding latency must be known and predictable.

The *AXI HyperConnect* is a hypervisor-level AXI interconnect thought for COTS FPGA SoC, which allows interconnecting multiple hardware accelerators to the same bus while ensuring isolation and predictability. The AXI HyperConnect has been integrated within a type-1 hypervisor under development in our laboratory.

The main features introduced by the AXI HyperConnect are:

- **Openness.** The proposed hardware architecture for AXI HyperConnect is slim and open, making it prone to low-level inspection to extract worst-case timing bounds, as well as to the corresponding validation. Furthermore, the AXI HyperConnect comes with an open-source driver to control it.
- **Low latency and resource consumption.** AXI HyperConnect improves the propagation latency with respect to the state-of-the-art AXI interconnects while maintaining a comparable throughput. Its development in HDL language also allows obtaining low resource consumption on the FPGA fabric with respect to the state-of-the-art AXI interconnects.
- **Bandwidth reservation.** AXI HyperConnect implements a bandwidth reservation mechanism, which works by limiting the number of transactions to a given budget within periodic time windows. This mechanism can be configured by the hypervisor or the Operative System, allowing to reserve a given bus bandwidth to each hardware accelerators and also controlling the overall memory traffic coming from the FPGA fabric directed to the shared memory subsystem (which can delay the execution of software running on the processors of the PS).

- **Fair bandwidth distribution.** AXI HyperConnect implements a mechanism to equalize bus transactions to a nominal burst size and liming the number of outstanding transactions. Combined with bandwidth reservation, this mechanism guarantees a very predictable bus access as both the number of transactions.
- **Runtime reconfiguration.** AXI HyperConnect exports a control AXI slave interface that allows changing its configuration from the PS as a standard memory-mapped device. In the considered framework, this control interface is managed by the hypervisor. This feature allows dynamic reconfiguration of AXI HyperConnect under Dynamic Partial Reconfiguration.
- **Decoupling from the memory subsystem.** AXI HyperConnect allows to individually enable/disable at runtime the access to the memory subsystem for each hardware accelerator connected to its slave ports. This feature allows isolating misbehaving/malicious hardware accelerators (even due to faulty silicon) detected in the system. AXI HyperConnect decouples all the signals of a disabled slave port, which is a useful feature under dynamic partial reconfiguration.
- **Compatibility.** All the features offered by the AXI HyperConnect have been developed to be compliant with the AXI standard. This means that the AXI HyperConnect is completely transparent to both the hardware accelerators and the memory subsystem and can hence be installed in place of state-of-the-art interconnects without any extra effort. Furthermore, the AXI HyperConnect is compatible with both AXI3 and AXI4 devices.

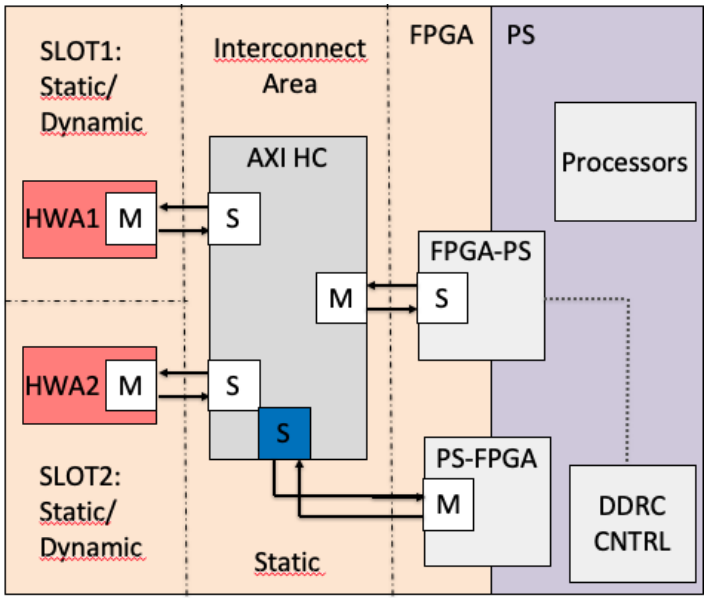


Fig. 1: AXI HyperConnect

Reference:

- Francesco Restuccia, Alessandro Biondi, Mauro Marinoni, Giorgiomaria Cicero, and Giorgio Buttazzo, AXI HyperConnect: A Predictable, Hypervisor-level AXI Interconnect for Hardware Accelerators in FPGA SoC, In Proceedings of the 57th ACM/ESDA/IEEE Design Automation Conference (DAC 2020), San Francisco, CA, USA, July 19-23, 2020.

5.2 AXI Budgeting Unit

The FRED framework includes the *AXI Budgeting Unit (ABU)* a hardware-based reservation mechanism for the AMBA BUS aimed at controlling the contention incurred by hardware accelerators in the BUS bandwidth domain. The ABU provides bus bandwidth reservation for hardware accelerators deployed on the FPGA. The ABU infrastructure comprises a set of ABU modules controlled by a central unit named ABU controller. Each ABU module is meant to be placed between a hardware accelerator and the remainder of the bus infrastructure. The ABU module supervises the bus traffic generated by the corresponding hardware accelerator providing both temporal and spatial isolation effectively shielding the accelerator from possible misbehaviors of other accelerators. The bandwidth reservation is implemented with a budgeting mechanism, i.e., by enforcing a limit on the number of transaction that the hardware accelerator can perform in a predetermined period.

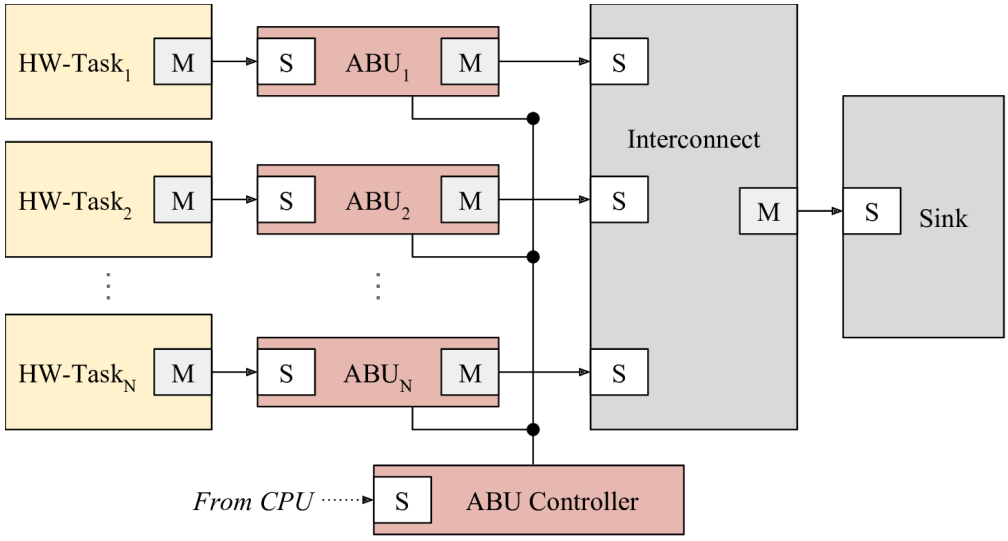


Fig. 2: AXI Budgeting Unit (ABU)

Reference:

- M. Pagani, E. Rossi, A. Biondi, M. Marinoni, and G. Buttazzo, A Bandwidth Reservation Mechanism for AXI-based Hardware Accelerators on FPGAs, Proc. of the Euromicro Conference on Real-Time Systems (ECRTS 2019), Stuttgart, Germany, July 9-12, 2019.

5.3 AXI Stall Monitor (ASM)

A big problem in using FPGA SoC platforms in safety-critical applications is that the interference occurring in accessing shared resources (such as the memory subsystem) may introduce unbounded and unpredictable delays in the computational activities, preventing any form of a-priori timing guarantee, required in such systems for certification purposes.

In modern FPGA SoC platforms data exchange mostly occurs through the AMBA AXI open standard. The AXI standard provides advanced features that make it highly flexible for different applications, but it does not define any mechanism to supervise the behaviour of bus masters. The lack of supervision allows hardware accelerators to behave (or misbehave) in the system without any control.

This is especially critical when hardware accelerators are provided as specialized IP blocks developed from external sources so that it is not possible to accurately validate them to verify the absence of misbehavior. To further complicate this issue, in systems using dynamic partial reconfiguration (DPR), misbehaving/malicious hardware accelerators can more likely be programmed on the FPGA. Such misbehaving conditions can compromise the functionality of the entire

system, up to requiring a system reset to restore a safe condition. This leads to large recovery delays that may not be acceptable in safety-critical applications and can harm the quality of service in non-critical systems.

The *AXI Stall Monitor* is a component conceived to address this issue. The ASM is a minimal hardware module IP which shields the system from misbehaving HW-tasks that may stall the bus. A sample architecture comprising the ASM is reported in Figure 1. The configuration of the ASM is supported by a worst-case analysis to bound the worst-case response time of periodic hardware tasks sharing a common memory.

Leveraging the worst-case analysis, ASM leaves some flexibility in the behaviour of the hardware accelerators, while keeping the HW-task set schedulable even in the presence of one or multiple misbehaving hardware accelerators. ASM does not introduce any additional latency on the performance and has a minimal impact on resource consumption.

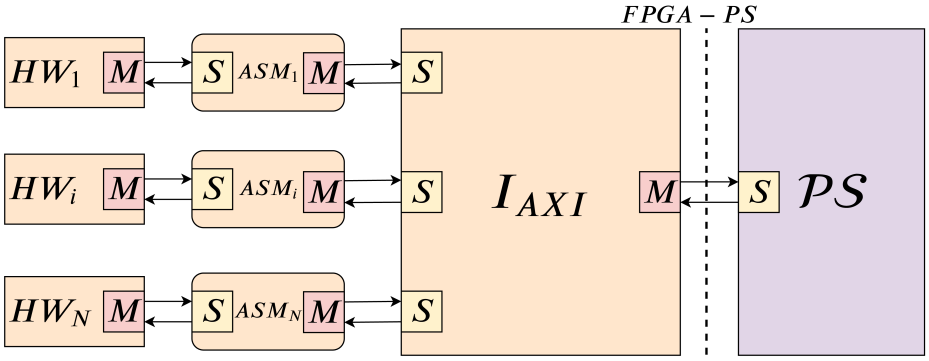


Fig. 3: AXI Stall Monitor

Reference:

- Francesco Restuccia, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo, Safely Preventing Unbounded Delays During Bus Transactions in FPGA-based SoC, the 28th IEEE International Symposium On Field-Programmable Custom Computing Machines (FCCM 2020), Fayetteville, Arkansas, USA, May 3-6, 2020.

5.4 AXI Bandwidth Equalizer

A typical FPGA SoC architecture combines a processing system (PS) (generally based on one or more processors) with a Field Programmable Gate Array (FPGA) subsystem in a single device. Both subsystems access a DRAM controller in the PS for accessing a shared DRAM memory.

The next figure illustrates *A typical SoC FPGA architecture* in which two interfaces allow the communication between the FPGA subsystem and the PS through a limited set of ports. The de-facto standard interface for interconnections is the ARM Advanced Microcontroller Bus Architecture Advanced eXtensible Interface (AMBA AXI).

Whenever multiple AXI masters in the FPGA want to access the same output port, an AXI Interconnect is in charge of arbitrating conflicting requests. The AXI protocol does not specify how conflicting transactions are arbitrated and hence the design of bus arbiters is left to the vendors that adopt AXI. For instance, the AXI arbiters for FPGA SoCs by Xilinx implement round-robin. Round-robin arbitration should guarantee fairness in contending the bus; specifically, it should guarantee a fair distribution of the bus bandwidth among the masters that contend a port.

However, a completely unfair bandwidth distribution can be achieved under some configurations, like in the presence of transactions with heterogeneous burst sizes issued by the masters. This issue makes it possible to arbitrarily decrease the bus bandwidth of a target master node.

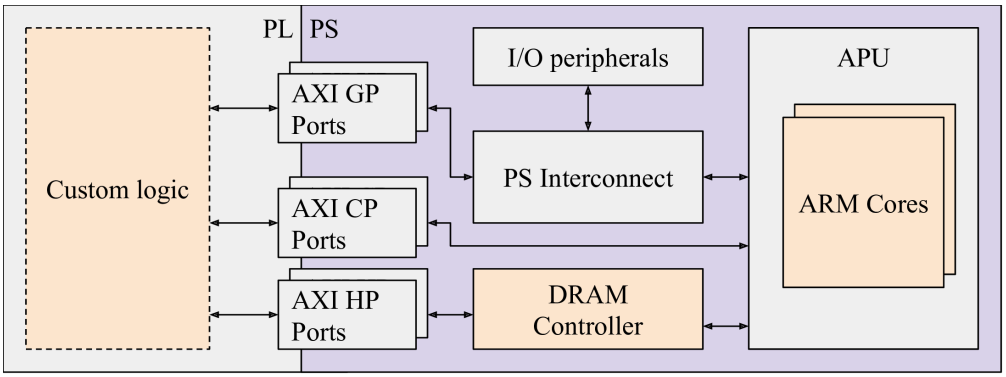


Fig. 4: A typical SoC FPGA architecture

The *AXI Bus Equalizer (ABE)* developed in this framework restores fairness in the bus arbitration. The ABE is conceived to be placed between each hardware accelerator and an input port of an AXI Interconnect to equalize the address burst requests issued by the AXI master hardware accelerators.

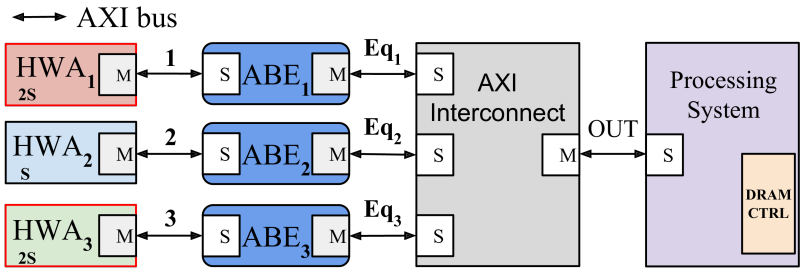


Fig. 5: AXI Bus Equalizer (ABE)

The main objective of the ABE is to achieve a fair bus bandwidth allocation in the presence of round-robin arbitration. ABE is implemented in HDL language, hence it is highly optimized in terms of performance, parallelism, and area consumption. Latency introduced by ABE on a single transaction is just one clock cycle, independently of the burst size of the transactions. The ABE is provided as a Xilinx IP block to simplify its integration in realistic designs. The impact of ABEs on resource consumption is very marginal (less than the 0.5% on a Zynq Ultrascale and about 4% in a ZYNQ Z-7020).

Reference:

- Francesco Restuccia, Marco Pagani, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo, Is Your Bus Arbiter Really Fair? Restoring Fairness in AXI Interconnects for FPGA SoCs, ACM Transactions on Embedded Computing Systems, to appear. Presented at the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES 2019), New York, USA, October 13 - 18, 2019.

CHAPTER

SIX

BUS SYNTHESIS

Note: To be done!

CHAPTER

SEVEN

GETTING STARTED

This section summarizes the FRED Framework workflow and the steps required to run a FRED-based application on the FPGA. The proposed workflow can be divided into three phases: hw/sw partitioning, hardware design, software deployment.

The *hw/sw partitioning phase* is when the designer has to decide which parts of the target application will be offloaded to FPGA. Typically this is decided based on profiling data from the application, where the bottlenecks are usually good candidates for FPGA offloading. For each function to be offloaded there must have an equivalent hw IP, like those already provided in dart_ip repository. Finally, the applications timing requirements should also be extracted in this phase.

Next, is the *hardware design phase*, where DART produces the bitstreams and hardware partitions. Note that it is also possible to work with FRED Runtime with hardware designed with Xilinx DPR flow called Dynamic Function eXchange instead of DART. The difference is that it will require more time and experience with Xilinx tools to have an equivalent hardware design. Moreover, the timing analysis done by DART would not exist, and the designer would need to do it himself.

The *software deployment phase* needs a FRED-ready Linux distribution, like the one created with the Yocto layers meta-fred and meta-retis and Petalinux. Once the Linux image is ready, then starts the application software design. One can use a Yocto-based embedded software development flow, the recommended flow, or design the software directly on the board. The Linux image designed for FRED works with both approaches.

Here is the steps to build a Linux image for FRED:

```
$ wget https://raw.githubusercontent.com/fred-framework/fred-docs/main/docs/07_getting-
↪started/build_img.sh .
$ chmod +x build_img.sh
$ wget https://raw.githubusercontent.com/fred-framework/meta-fred/main/scripts/pt-config
# wget bsp
$ ./build_img.sh -h
```

Or, it is possible to download from here a pre-built image for the ZCU-102 board with a basic example built-in.

Once the image is running on the board, To run the basic example, execute:

```
$ load_hw
$ fred-server &
$ sum-vec
```

The tutorials in DART repository are a good starting point for a development flow based on DART and FRED. The alternative is this other tutorial within Vivado Xilinx for the hardware design and FRED for the software deployment phase.

CHAPTER

EIGHT

CASE STUDIES

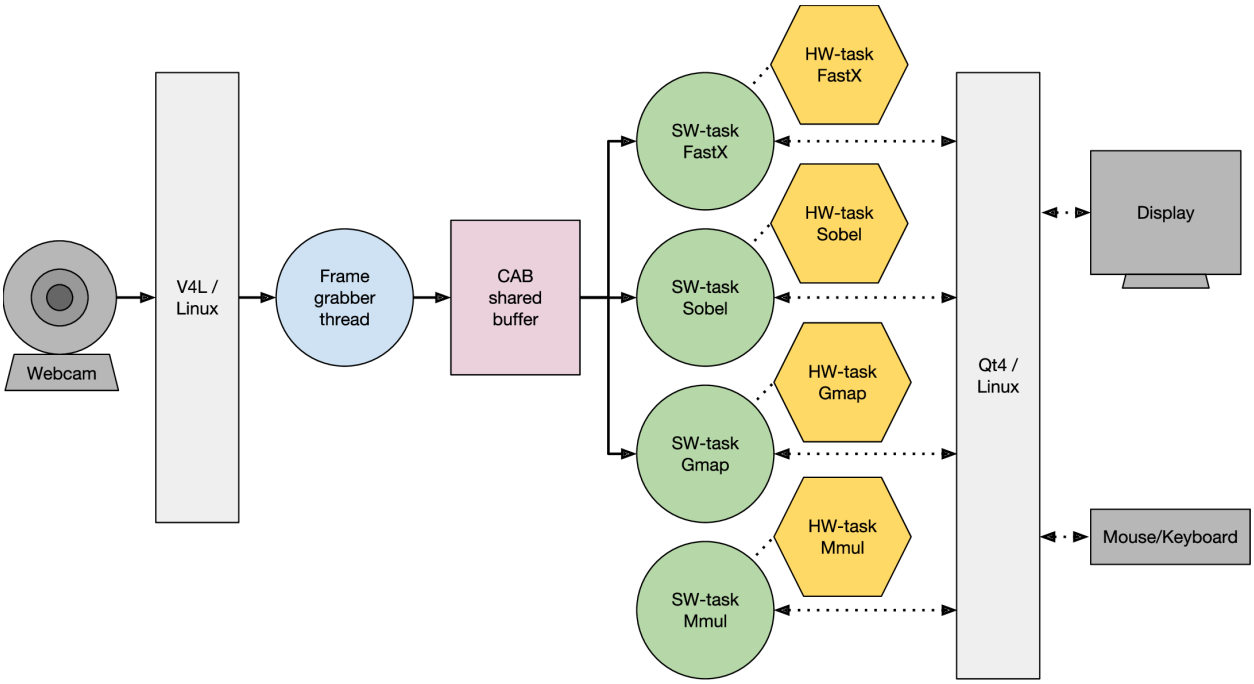
The following case studies demonstrates the capabilities of the FRED framework:

8.1 FRED/DART Tutorial

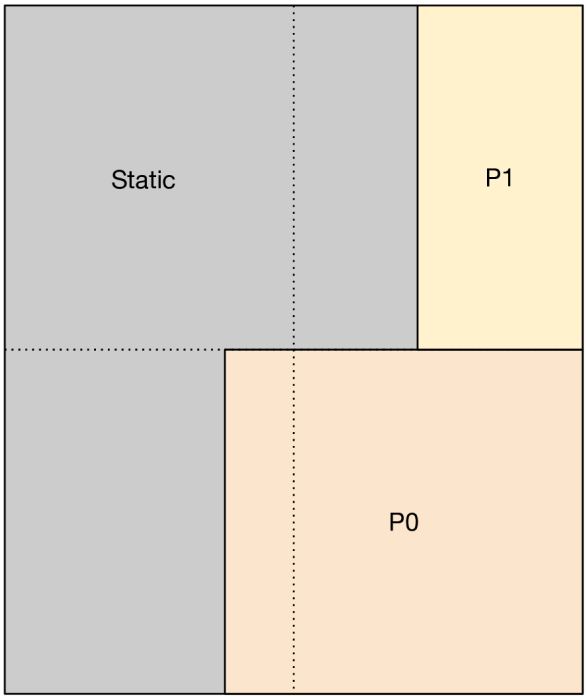
Attention: Add a modified version of this document.

8.2 Video Processing with ZYBO Board

This application has been designed to test the FRED runtime in a realistic scenario. The application makes use of the virtualized FPGA support to speed up the processing of live images acquired by a USB webcam and multiplications of integer matrices. The set of hardware image filters includes a Sobel filter, a FAST edge detection filter, and a color map filter. These filters have been implemented both as HLS hardware accelerators and equivalent software procedures using the popular OpenCV library with the purpose of testing the speedup factors.



In this application, the reconfigurable region is divided into two partitions containing a single slot each. These two slots are shared at runtime by four hardware accelerators (Sobel, FAST, Gmap, and Mult) using the FPGA virtualization mechanism offered by the FRED server.



The following video shows the case-study application operating in hardware mode. Please, note that the processing of each sub-image triggers a partial reconfiguration of the FPGA fabric, resulting in a rate higher than 50 reconfigurations per second. It is also worth noting that the FPGA contains resources to statically host only two of the four hardware accelerators and that a pure software implementation is considerably slower. Only by leveraging resource virtualization through partial reconfiguration all the four tasks can achieve a reasonable performance.

If you have a **ZYBO board**, you can try the case study application on your own. In addition to the board, you need a USB webcam capable of acquiring `640x480` frames, and a `5V` min `2A` capable power supply. To prepare the application, download the image file available here. Then unzip the archive and copy the image to a micro SD of size `2 GB` or more using `dd` or an equivalent tool.

```
$ unzip fred_cs.zip
$ dd if=fred_sd.img of=/dev/mmcblk0 bs=8M conv=fsync
```

Once the micro SD is ready, insert it into the board, connect the external power supply, and remember to set the ZYBO for SD boot and external power supply using the board’s jumpers. When ready, connect the USB webcam, an HDMI monitor, and then start the ZYBO. Once the boot process has completed, login usign `root` as username and password. Then, launch the FRED runtime and the Qt client application.

```
$ ./start_fred.sh
$ ./fredVideoApp -qws
```

8.3 AMPERE Project Case Study

Attention: Case study developed in the AMPERE project, using ROS and OpenMP

CHAPTER

NINE

RESEACH ROADMAP

These are the main areas of research/development for the near future:

- Currently DART optimizes FPGA resource usage and timing. In the future it will also optimize power;
- The regulating bus/memory contention (aka the predictable bus manager) mechanisms are still not integrated into DART.;
- Integrate the FRED framework with APP4MC/Amalthea to enable Model Based Engineering with FPGA offloading;
- Integrate the FRED framework with Xilinx Vitis AI flow for accelerating deep learning applications;
- Integrate the FRED framework with ROS2 for robotics and automonous vehicle applications;

CHAPTER

TEN

PUBLICATIONS

13. Biruk Seyoum, Marco Pagani, Alessandro Biondi, Sara Balleri, and Giorgio Buttazzo, *"Spatio-Temporal Optimization of Deep Neural Networks for Reconfigurable FPGA SoCs"*, IEEE Transactions on Computers, to appear;

12. Biruk Seyoum, Alessandro Biondi, Marco Pagani, and Giorgio Buttazzo, *"Automating the Design Flow under Dynamic Partial Reconfiguration for Hardware-Software Co-design in FPGA SoC"*, In Proceedings of the 36th ACM/SIGAPP Symposium on Applied Computing (SAC 2021), March 22-26, 2021;

11. Francesco Restuccia, Marco Pagani, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo, *"Modeling and Analysis of Bus Contention for Hardware Accelerators in FPGA SoCs"*, In Proceedings of the 32nd Euromicro Conference on Real-Time Systems (ECRTS 2020), July 7-10, 2020;

10. Francesco Restuccia, Alessandro Biondi, Mauro Marinoni, Giorgiomaria Cicero, and Giorgio Buttazzo, *"AXI HyperConnect: A Predictable, Hypervisor-level AXI Interconnect for Hardware Accelerators in FPGA SoC"*, In Proceedings of the 57th ACM/ESDA/IEEE Design Automation Conference (DAC 2020), San Francisco, CA, USA, July 19-23, 2020;

9. Francesco Restuccia, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo, *"Safely Preventing Unbounded Delays During Bus Transactions in FPGA-based SoC"*, To be presented at the 28th IEEE International Symposium On Field-Programmable Custom Computing Machines (FCCM 2020), Fayetteville, Arkansas, USA, May 3-6, 2020;

8. Francesco Restuccia, Marco Pagani, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo, *"Is Your Bus Arbiter Really Fair? Restoring Fairness in AXI Interconnects for FPGA SoCs"*, ACM Transactions on Embedded Computing Systems, to appear. To be presented at the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES 2019), New York, USA, October 13 - 18, 2019;

7. Biruk Seyoum, Alessandro Biondi, and Giorgio Buttazzo, *"FLORA: FLoorplan Optimizer for Reconfigurable Areas in FPGAs"*, ACM Transactions on Embedded Computing Systems, to appear. To be presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2019), New York, USA, October 13 - 18, 2019;

6. Marco Pagani, E. Rossi, A. Biondi, M. Marinoni, and G. Buttazzo, *"A Bandwidth Reservation Mechanism for AXI-based Hardware Accelerators on FPGAs"*, Proc. of the Euromicro Conference on Real-Time Systems (ECRTS 2019), Stuttgart, Germany, July 9-12, 2019;

5. Enrico Rossi, M. Damschen, L. Bauer, G. Buttazzo, J. Henkel, *"Preemption of the Partial Reconfiguration Process to Enable Real-Time Computing with FPGAs"*, ACM Transactions on Reconfigurable Technology and Systems, Vol. 11, Issue 2, pp. 10:1–10:24, November 2018;

4. Alessandro Biondi and G. Buttazzo, *"Timing-aware FPGA Partitioning for Real-Time Applications Under Dynamic Partial Reconfiguration"*, Proc. of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2017), Pasadena, CA, USA, July 24-27, 2017;

3. Marco Pagani, A. Balsini, A. Biondi, M. Marinoni, and G. Buttazzo, *"A Linux-based Support for Developing Real-Time Applications on Heterogeneous Platforms with Dynamic FPGA Reconfiguration"*, Proc. of the 30th IEEE Int. System-on-Chip Conference (SOCC 2017), Munich, Germany, September 5-8, 2017;

2. Marco Pagani, M. Marinoni, A. Biondi, A. Balsini, and G. Buttazzo, *"Towards Real-Time Operating Systems for Heterogeneous Reconfigurable Platforms"*, Proc. of the 12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT16), in conjunction with ECRTS16, Toulouse, France, July 5, 2016;

1. Alessandro Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, *"A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs"*, Proc. of the IEEE Real-Time Systems Symposium (RTSS 2016), Porto, Portugal, Nov. 29 - Dec. 2, 2016;

CHAPTER

ELEVEN

ABOUT

The FRED framework has been developed at the RETIS Lab of the Scuola Superiore Sant'Anna of Pisa.

11.1 Project coordinators

- Alessandro Biondi
- Giorgio Buttazzo
- Mauro Marinoni
- Tommaso Cucinotta

11.2 Contributors

- Alessandro Biondi: FRED Analyzer, AXI Budgeting Unit, AXI Bandwidth Equalizer, Floorplannig;
- Alessio Balsini: FRED scheduling simulator, FRED runtime;
- Alexandre Amory: FRED/DART integration, DART testing, DART IPs, AMALTHEA codegen, FRED build system, documentation, testing on ZCU102 board;
- Biruk Seyoum: DART and Floorplannig;
- Enrico Rossi: Preemptable reconfiguration, AXI Budgeting Unit;
- Francesco Restuccia: AXI Bandwidth Equalizer, AXI Stall Monitor, AXI HyperConnect, Xilinx DNN;
- Giuseppe Lipari: AXI Budgeting Unit;

- Lorenzo Molinari: Support for PYNQ;
- Marco Pagani: FRED runtime, AXI Budgeting Unit, AXI Bandwidth Equalizer, Image processing demo;
- Sara Balleri: Deep learning case study.

11.3 Funding

The FRED framework has been partially developed in the context of the AMPERE project. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871669.

CHAPTER

TWELVE

PAPERS

If you are using FRED and/or its software on your research projects, please cite our papers:

```
@inproceedings{biondi2016framework,
title={A framework for supporting real-time applications on dynamic reconfigurable FPGAs}
↪,
author={Biondi, Alessandro and Balsini, Alessio and Pagani, Marco and Rossi, Enrico and␣
↪Marinoni, Mauro and Buttazzo, Giorgio},
booktitle={IEEE Real-Time Systems Symposium (RTSS)},
pages={1--12},
year={2016},
organization={IEEE}
}
```

```
@article{fred-linux,
title = {A Linux-based support for developing real-time applications on heterogeneous␣
↪platforms with dynamic FPGA reconfiguration},
journal = {Future Generation Computer Systems},
volume = {129},
pages = {125-140},
year = {2022},
issn = {0167-739X},
doi = {https://doi.org/10.1016/j.future.2021.11.007},
url = {https://www.sciencedirect.com/science/article/pii/S0167739X21004362},
author = {Marco Pagani and Alessandro Biondi and Mauro Marinoni and Lorenzo Molinari and␣
↪Giuseppe Lipari and Giorgio Buttazzo},
keywords = {Heterogeneous computing, FPGA, DPR, Real-time, Linux}
}
```

If you are using DART, please cite:

```
@inproceedings{seyoum2021automating,
title={Automating the design flow under dynamic partial reconfiguration for hardware-
↪software co-design in FPGA SoC},
author={Seyoum, Biruk and Pagani, Marco and Biondi, Alessandro and Buttazzo, Giorgio},
booktitle={ACM Symposium on Applied Computing (SAC)},
pages={481--490},
year={2021}
}
```

CHAPTER

THIRTEEN

LICENSE

FRED/DART and its software are protected under the GPLv3 license.

CHAPTER

FOURTEEN

FEEDBACK

Don't hesitate to ask about additional info or the next guides, and also if you find some mistakes, please let us know. Issues and push requests can be done on github.