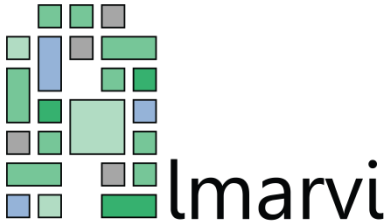


ALMARVI

"Algorithms, Design Methods, and Many-Core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing"

Project co-funded by the ARTEMIS Joint Undertaking under the

ASP 5: Computing Platforms for Embedded Systems

ARTEMIS JU Grant Agreement no. 621439

D4.6 – *Integrated System Software Stack*

Due date of deliverable: December 1, 2016

Start date of project: 1 April, 2014 **Duration:** 36 months

Organisation name of lead contractor for this deliverable: PHILIPS

Author(s): Pekka Jääskeläinen, Timo Viitanen, Michal Babej, Aleksi Tervo (TUT), Steven van der Vlugt (PHILIPS), J.R. van Kampenhout , Amir Baghbanbehrouzian, Dip Goswami (TUE)

Validated by: Jiri Kadlec (UTIA)

Version number: 1.0

Submission Date: 30.11.2016

Doc reference: ALMARVI_D4.6_final_V10.docx

Work Pack./ Task: WP4 task 4.3

Description: ***(max 5 lines)*** Presents details on the implementation and usage of the ALMARVI system software stack algorithms and tools.

Nature:	**R**		
Dissemination Level:	**PU**	Public	**X**
	PP	Restricted to other programme participants (including the JU)	
	RE	Restricted to a group specified by the consortium (including the JU)	
	CO	Confidential, only for members of the consortium (including the JU)	

DOCUMENT HISTORY

Release	Date	Reason of change	Status	Distribution
V0.1	2016-07-11	First draft / placeholder.	Draft	CO
V0.2	2016-10-08	TUT contribution (software stack description) added.	Draft	CO
V0.3	2016-11-07	Added Philips contribution	Draft	CO
V0.4	2016-11-15	Final editing Philips	Draft	CO
V0.5	2016-11-15	Final editing TUT.	Draft	CO
V0.6	2016-11-17	Final editing before review	Draft	CO
V1.0	2016-11-30	Final after internal review ready for release	Final	PU

Contents

Glossary

Abbreviation / acronym	Description
AlmaIF	**ALMA**RVI Common IP Integration **I**nter**F**ace. A common hardware wrapper interface for IP blocks with capabilities to integrate to an OpenCL-based heterogeneous software stack.
CPU	Central Processing Unit. Typically an ARM or Intel processor that is mainly used for running the host/control code of ALMARVI applications.
DSP	Digital Signal Processors. Programmable processors originally designed for DSP workloads.
Dyplo	DYnamic Process LOader: a middleware solution to seamlessly integrate software and hardware.
FPGA	Field Programmable Gate Array. Programmable hardware.
GPU	Graphics Processing Unit. Nowadays general purpose programmable high performance processors originally designed for graphics rendering.
OpenCL	Open Computing Language. An open heterogeneous programming standard that is used as a core of the system software stack of the ALMARVI project, thus also drove the design of the hardware integration interface.
ISA	Instruction Set Architecture. The "command language" the processor understands, the interface between the software and the hardware.
PS	Processing System in Xilinx Zynq device. Two ARM Cortex A9 processors, with DDR3 memory controller and memory interfaces and ARM peripherals.
PL	Programmable Logic area of Xilinx Zynq device.
TTA	Transport triggered architecture: a style of CPU micro-architecture and its instruction set, focusing on data transfers. Studied in Almarvi in WP3.

2 Introduction

In the ALMARVI project, a customizable image/video processing hardware platform with easy integration of components such as hardware accelerators and application-specific processors which work together in executing applications with multiple tasks in a coordinated manner was developed. The platform was designed to be programmable by utilizing OpenCL as a middleware API. On top of OpenCL, it is possible to implement higher-level programming models as long as the underlying devices support OpenCL programming.

In order to make hardware integration cleaner, a common ALMARVI IP block interface called AlmaIF was developed. This interface, more thoroughly explained in D3.8, was designed with OpenCL support in mind, specifically with the intention to make OpenCL driver development and their use from application programs more unified across the ALMARVI custom devices.

This document describes the software aspects of the integrated system software stack of ALMARVI in chapter 3, of which key components include *pocl*, an open source OpenCL implementation which is described in chapter 4 and the AlmaIF that provides a common control interface utilized by the AlmaIF pocl driver, which is described in chapter 5.

Chapter 6 presents the programming model and software integration of Dyplo, which has been used by Philips as a commercial alternative to AlmaIF.

Chapter 7 describes the programming model for image processing applications with high load variation targeting a TMDA-scheduled multicore platform – e.g., CompSOC. A scenario-aware dataflow (SADF) programming model is developed that captures the load dynamism. The experimental evaluation validates the models with high accuracy. Moreover, SADF based models for various stages were developed for performance analysis of an image-guided assembly line prototype – xCPS, which is presented in chapter 8. The image-guided assembly line closely resembles a typical healthcare settings considered under ALMARVI. The SADF programming model in CompSOC and the SADF model for the xCPS machine are complimentary towards modeling the entire system (i.e., image sensing pipes and assembly stages).

3 OpenCL-Based System Software Stack

First released in December 2008, Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs (Central Processing Units), GPUs (Graphics Processing Units), DSPs (Digital Signal Processors), FPGAs (Field Programmable Gate Arrays) and other processors. OpenCL supports data and task level parallelism through a simple to understand and use programming model. Furthermore, since it abstracts away the details of different compute architectures, OpenCL provides an API to enable software portability.

The interaction between the host and the OpenCL devices occurs through commands posted by the host to command-queues defined within the OpenCL contexts. These commands wait in the command-queue until they execute on an OpenCL device. A command-queue is created by the host and attached to a single OpenCL device after the context has been defined. The host places commands into the command-queue, and the commands are then scheduled for execution on the associated device. The commands can synchronize with each other via events or implicit data dependencies as defined by the buffer arguments defined for each command.

OpenCL supports three types of commands:

- Kernel execution commands execute a kernel on the processing elements of an OpenCL device.
- Memory commands transfer data between the host and different memory objects, move data between memory objects, or map and unmap memory objects from the host address space.
- Synchronization commands put constraints on the order in which commands execute.

In a typical host program, the programmer defines the context and the command-queues, defines memory and program objects, and builds any data structures needed on the host to support the application. Then the focus shifts to *devices* and command-queues used to control them. Memory objects are moved from the host onto the devices; kernel arguments are attached to memory objects and then submitted to the command-queue for execution. When the kernel has completed its work, memory objects produced in the computation may be copied back onto the host. Furthermore, to support platforms with devices that can interoperate, the devices can directly transfer buffers between them, in case the runtime system is advanced enough.

Due to the fact that OpenCL provides a hardware agnostic and portable programming framework that supports data and task level parallelism and is now a relatively wide spread standardized programming model for heterogeneous compute platforms, adopted by multiple vendors, OpenCL was leveraged within the context of ALMARVI as a backbone to implement its system software stack.

A major issue that hindered the applicability of OpenCL in ALMARVI is the platform dependence issue. At the moment, all chip vendors provide their own OpenCL implementations that target their own chips. When a programmer wants to leverage multiple OpenCL capable cards/chips from different vendors, the only possibility is to create a different OpenCL context for each OpenCL platform. This limits the programmer in the sense that multiple algorithms processing the same video frame in a pre-determined order cannot be synchronized and pipelined on chips coming from different vendors, since synchronization between different contexts is not natively supported by current OpenCL standard. Within the context of ALMARVI, this issue was targeted by pocl, an open source OpenCL implementation helped in cross-vendor integration issues, enabling single OpenCL platforms to contain ARM or Intel CPUs, AMD GPUs with HSA support, and the ALMARVI partner devices from TUT and TUD.

The benefits of a common programming standard are clear; multiple vendors can provide support for application descriptions written according to the standard, thus reducing the program porting effort. While the standard brings the obvious benefits of platform portability, the performance portability aspects are largely left to the programmer. The situation is made worse due to multiple proprietary vendor implementations with different characteristics, and, thus, required optimization strategies. In addition, the OpenCL API is rather detailed, hindering the programmer productivity, which called for having room for higher-level programming standards on top of the system software stack.

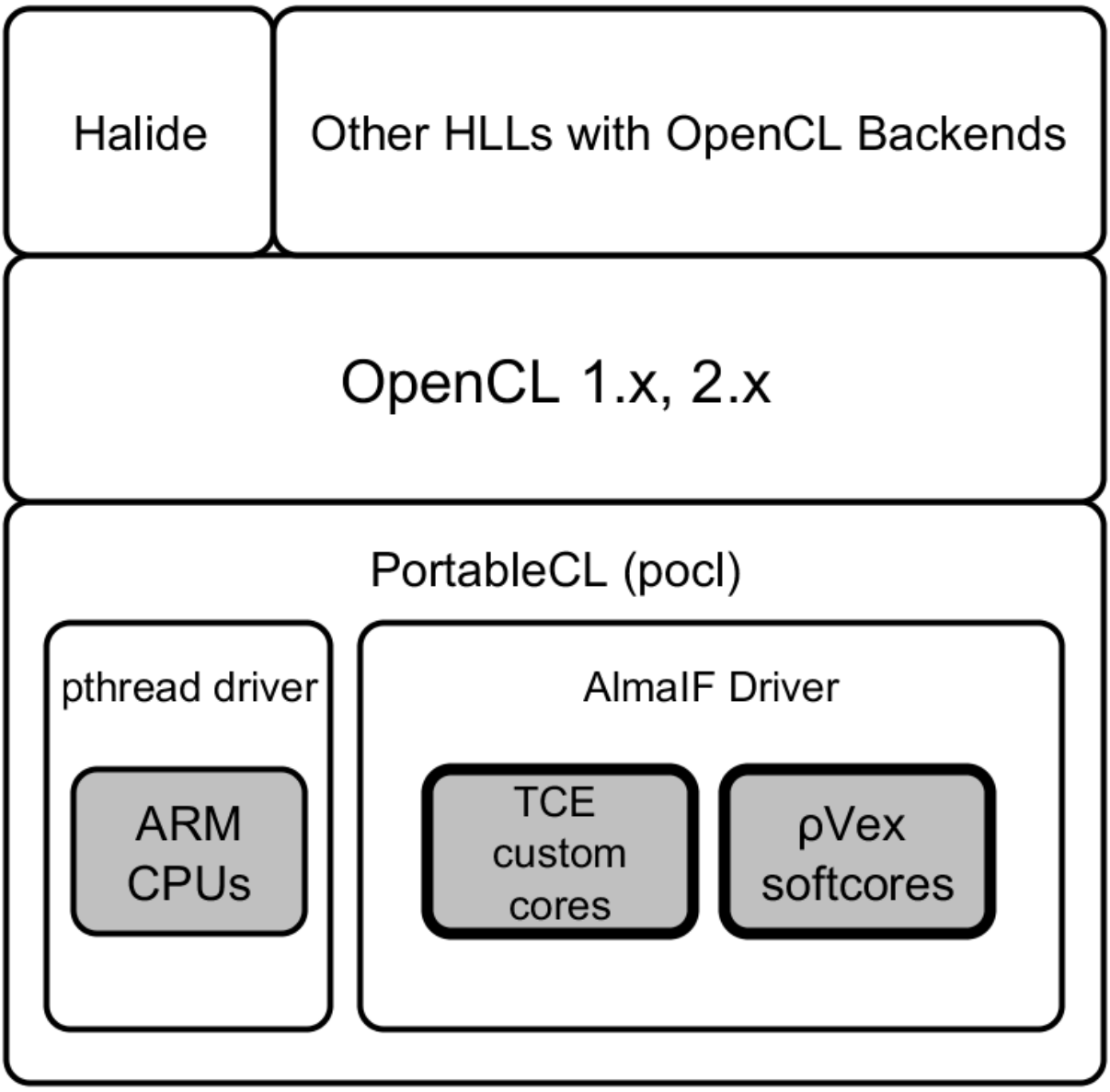


Figure 1: ALMARVI OpenCL-based System Software Stack.

Figure 1 illustrates the system software stack that utilizes OpenCL as the portability layer on top of which more productive (in terms of engineering time) higher level programming models can be implemented. Within the ALMARVI there was an experiment with Halide, a domain specific language for image processing pipelines which has an OpenCL backend. With work done in ALMARVI, Halide programs can now be used to program TCE custom cores and also invoke custom instructions in them, a

programming path that is currently being optimized. The Halide part is explained in more detail in a conference paper [1].

Portable Computing Language (PortableCL/pocl) was used as an implementation framework for different supported device types in the project, and also as a foundation for the OpenCL implementation targeting the AlmaIF devices. Its previously implemented 'pthread' driver can be used to program the ARM CPUs in the demonstrator platform, like they were yet another OpenCL devices. Most importantly, under the AlmaIF driver layer, we implemented support for two targets, the customizable TCE cores from Tampere University of Technology, and the rVex runtime reconfigurable VLIWs from Delft University of Technology. Thanks to the common control interface, the different device types varied mostly on the details on how to invoke their code generation tools.

In OpenCL, kernels (the pieces of the application mapped to devices) are typically built "*online*", at the runtime of the host application. This means that the OpenCL kernel code is compiled at run time by the host, invoking the compiler toolchain of the target device for the kernel sources. This allows portability of the kernel code across different target architectures. For embedded applications, such run-time support is sometimes (or often) considered overly expensive as it requires an operating system and compiler toolchains for each supported device to be installed in the CPU that runs the host parts of the OpenCL application. This was the case also with the ALMARVI demonstration platform based on Zynq that has an ARM CPU with a limited amount of memory. Compiling and code generating kernels running in the ALMARVI devices at runtime is clearly useless and wasteful in the case where the program has been finalized and its kernels are already known. For supporting such cases, OpenCL also allows *'offline*' compilation, where the kernel code is compiled not in the running host, but up front in the system software development desktop environment. The host then contains device-specific binaries for each kernel-device pair and can launch them directly without the compilation step nor having the compiler installed in the deployment system. Although the offline compilation mode is less popular with the later OpenCL versions, it was found useful for the ALMARVI targets and for resource constrained embedded deployment scenarios in general.

4 pocl Driver for AlmaIF

pocl is a project that aims towards an OpenCL implementation that is both portable and performance portable. At its core is a kernel compiler that can be used to exploit the data parallelism of OpenCL programs on multiple platforms with different parallel hardware styles. The kernel compiler is modularized to perform target independent parallel region formation separately from the target-specific parallel mapping of the regions to enable support for various styles of fine-grained parallel resources such as subword SIMD extensions, SIMD datapaths and static multi-issue.

Unlike previous similar techniques that work on the source level, the parallel region formation retains the information of the data parallelism using the LLVM IR and its metadata infrastructure. This data can be exploited by the later generic compiler passes for efficient parallelization.

pocl is platform portable with a modular internal software structure, enabling implementing OpenCL on a wide range of architectures, both already commercialized and on those that are still under research. Preliminary results show that most of the benchmarked applications when compiled using pocl were faster or close to as fast as the best proprietary OpenCL implementation for the platform at hand.

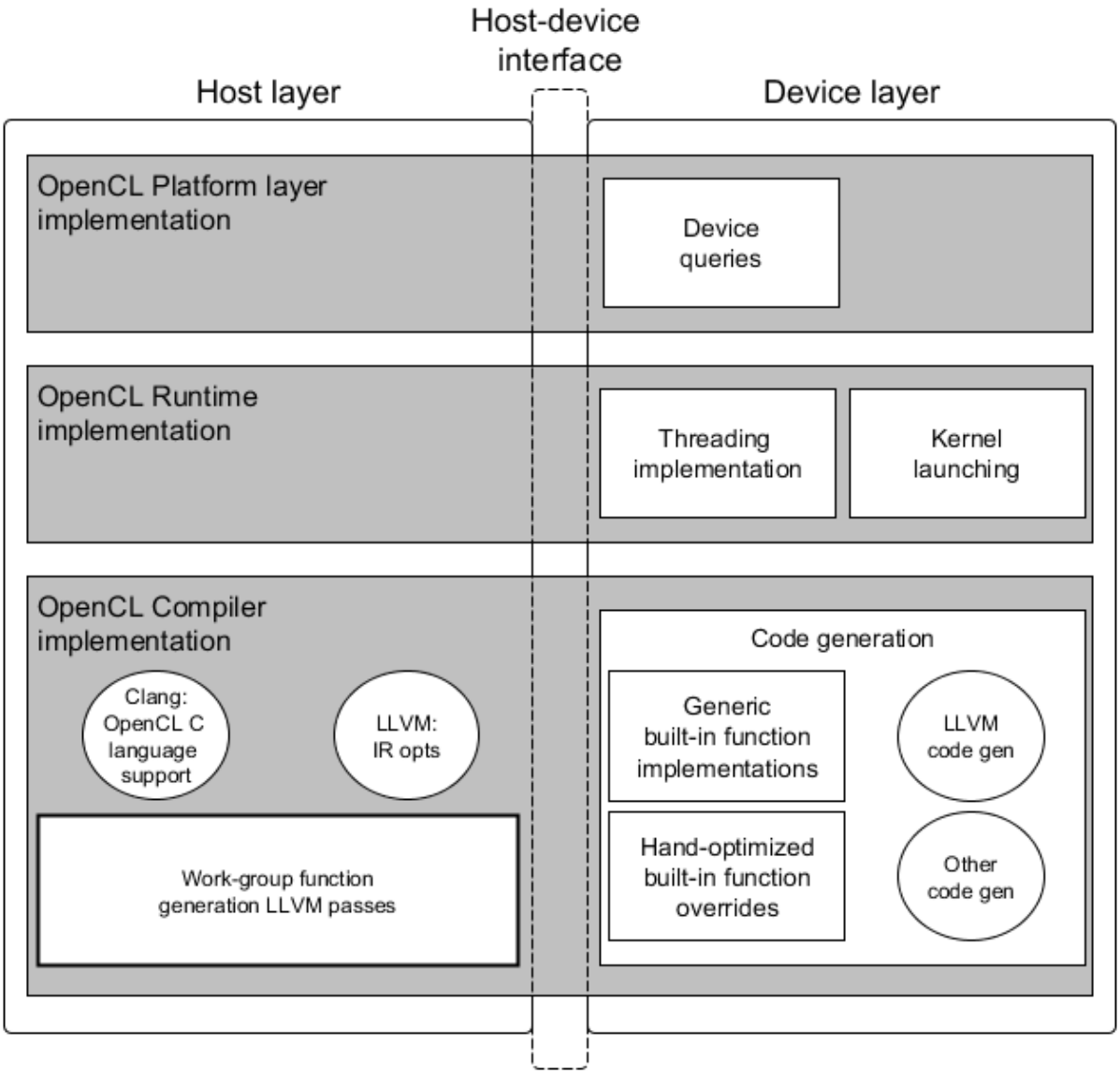


Figure 2: Software Components in the pocl OpenCL Implementation Framework.

Figure 2 shows the internal structure of the *pocl*. On the right side is the device-layer, which consist the parts of pocl that must be adapted when support for a new device type is added. The *host layer* implementation is portable to targets with operating system C compiler support. The *device layer* encapsulates the operating system and instruction-set architecture (ISA) specific parts such as code generation for the target device, and orchestration of the execution of the kernels in the device.

Most of the API implementations of the OpenCL framework in pocl are generic implementations written in C which call the *device layer* through a generic host-device interface for device-specific parts. For example, when the OpenCL program queries for the number of devices, pocl returns a list of supported devices without needing to do anything device-specific yet. However, when the application asks for the size of the global memory in a device, the query is delegated down to the device layer implementation of the device at hand.

The device layer consists of target-specific implementations for functionality such as target-specific parts of the kernel compilation process, the final execution of the command queue including uploading the kernel to the device and launching it, querying device characteristics, etc.

The division of responsibilities between the device-specific and generic parts is more easily explained by describing the currently implemented device interface implementations. As of this writing, the responsibilities between the device-specific and generic parts in the currently supported device interfaces are as follows:

Basic - A minimal example CPU device implementation. The execution of kernels happens one work-group at a time without multithreading. This driver can be used for implementing a device on a POSIX-compliant operating system for the case where the host and the device are the same.

Pthread - Similar to 'basic' except that it uses the POSIX threads library to execute multiple work-groups in parallel. This is an example of a device layer implementation that is capable of exploiting the thread level parallelism in multi-work-group execution. In the ALMARVI Zynq demonstration platform, this driver can be used to execute kernels on the ARM CPU cores.

Ttasim - Implementation of a simulated heterogeneous accelerator setup. The driver simulates customizable TCE-designed TTA cores executing the kernels. The processors are simulated by calling the instruction set simulator of the TCE. The driver performs the memory management of the device memories at the host side, and controls the kernel execution at the device. The functionality has been verified against an FPGA based implementation on a development board attached to the PCIe bus.

HSA – Driver for devices that implement the HSA standard. Currently it's useful for integrating with AMD's Kaveri and Carrizo APUs.

AlmaIF – The device driver implemented in the ALMARVI project for controlling devices that implement the AlmaIF hardware interface.

One important responsibility of a device layer implementation is resource management, which is, ensuring the resources of the device needed for kernel execution resources are properly shared and synchronized between multiple kernel executions. The allocation of the OpenCL buffers from the device memory requested via the clCreateBuffer and similar APIs is also part of the resource management responsibility of the device layer.

For assisting in memory management, pocl provides a memory allocator implementation called *Bufalloc* which aims to optimize the allocation of large continuous buffers typical in OpenCL applications. There are two main motivations for the customized kernel buffer allocator: 1) exploit the knowledge of the ``throughput computing'' workloads of OpenCL where the buffers are usually relatively big to reduce fragmentation, and 2) offer a generic memory allocator for devices without such support on device.

The working principle of the allocator is similar to memory pools in that a larger region of memory can be allocated at once with a single malloc call (or at compile time by allocating a static array). Chunks of this region are then returned to the application using a fast allocation strategy tailored for the OpenCL buffer allocation requests. As the allocation of the initial region can be done in multiple ways, the same memory allocator can be also used to manage memory for devices without operating systems. In that case, the host only keeps book of all the buffer allocations using Bufalloc for a known available region in the device memory and the device assumes all the kernel buffer pointers are initialized by the host to valid memory locations. The memory allocation strategy is designed according to the assumption that the buffers are long lived (often for the whole lifetime of the OpenCL application) and are allocated and de-allocated in groups (space for the entire kernel buffer arguments reserved and freed with successive calls to the allocator). These assumptions imply that memory fragmentation can be reduced by allocating neighboring areas of the memory for the successive allocation requests. A simple first fit algorithm is used in finding free space for the buffer allocation requests.

The internal book keeping structure of Bufalloc is split to chunks with a free/allocated flag and a size. The chunks are ordered by their starting address in a linked list. The last chunk in the list is a sentinel that holds all the unallocated memory. When a buffer allocation request is received, the linked list is traversed from the beginning to the end until an unallocated chunk with enough space is found. This chunk is then split to two chunks; one having the exact size of the buffer request that is returned to the caller, and another carrying the rest of the unallocated space in the original chunk. The allocation strategy has a customizable greedy mode which always serves new requests from the last chunk (end of the region) if possible. This mode results more often in the successive kernel buffer allocation calls being allocated from continuous memory space given the original allocated region is large enough.

Bufalloc was used to keep book of buffers to the distributed pieces of on-chip memories in each of the AlmaIF device, with the control interface and the buffers accessible via memory mapping /dev/mem. The next version is planned to include better support for inter-device streaming to reduce the host bottleneck.

Other details of interest in the AlmaIF driver implementation regarding the different address spaces are the following:

- For instruction memory, instructions are byteswapped if ins_size_padded > 0 and host & device endianness don't match. This is TCE specific quirk it and should be removed in the later versions.
- Parameter memory is currently unused. This will be later utilized for storing the command queues in case of autonomous device execution.
- There is currently a fixed size (2KB) unallocated space in the beginning of the data memory that the Bufalloc won't map buffers to. This is reserved to global data in the program loader.

When adding new device types, the main thing to add is a pointer to the kernel compilation hook (*compile_kernel* pointer in "struct almaif_device_info_t"). Each ALMAIF device must set this to its own compilation callback. The callback may do whatever it wants, as long as the driver finds these in the pocl_binary (or the kernel cache directory) after the compiler has been invoked:

- parallel.img – instruction memory image (binary) that includes the kernels and the kernel launcher
- parallel_data.img – global data image for the data memory (max 2048 bytes). TCE puts here the kernel name and more importantly, the address of the kernel's workgroup function.
- parallel_param.img – parameter memory image. Initial data that is put to the parameter memory, if any.

As the ALMARVI demo platform relies on offline kernel compilation, each device type should implement offline compilation support by adhering to the following in their kernel compiler hook:

- Check the POCL_OFFLINE_COMPILE env var value at start
- If enabled, the driver
    - should create fake device instances with device infos
    - must avoid calling external runtimes (like HSA), external interfaces (like ALMAIF control interface) or anything similar that may fail/abort
    - must check if the device exists (device->available) before doing any "physical" operations from device->ops

The following sequence of events happens in the driver at runtime when executing an OpenCL program with the kernels compiled offline to binaries:

The compiler callback merely unpacks the binary to the kernel compiler cache directory hierarchy and stores the unpacked images (parallel{,_data,_param}.img) in memor. In the *run* callback, the device is halted, images are loaded to device memory (via memcpy, with byteswap if required), ``struct __kernel_exec_cmd`` is set up (see *include/pocl_device.h*), the command's status is set to POCL_KST_READY, work dim / local / global sizes & offsets are initialized, pointers to arguments (in ``__kernel_exec_cmd``) are set to point to the kernel arguments, and the device is unhalted. Then the driver waits for ``struct __kernel_exec_cmd->status`` to change to POCL_KST_FINISHED after which the device is halted again.

When adding a new AlmaIF device type, the following needs to be adapted in the AlmaIF driver:

- Add a new struct to "almaif_device_infos[]" array. It contains the basic information about the ALMAIF devices recognized by the driver.
- The important fields are **dev_class** and **dev_id** (used to match device to the info structs), the **build_hash** (used to match pocl binaries to devices), **endian_little** (device endianness), **ins_size_padded,** padding of instructions (32/64 supported, should be 0 for all but TCE), **has_scratchpad_mem** and **scratchpad_size** - local memory support, **fill_kernel_metadata** and **kernel_imem_offset** - rVEX specific, but might be required for other devices in future

Your device (FPGA) must implement the same version of the ALMAIF interface version the driver is using (ALMAIF_IF_VERSION in the beginning of **lib/CL/devices/almaif/almaif.c** for macros mapping register numbers).

Your device needs to expect data in ``struct __kernel_exec_cmd`` at offset 2048 of dmem, loop on status to wait for POCL_KST_READY, and write POCL_KST_FINISHED there when done. This is done by **lib/CL/devices/tce/tta_device_main.c** for TCE.

For TCE, the compiler takes care of including **tta_device_main.c** in compilation process; other devices must take care of it themselves. RVEX has a precompiled **device_main.c** that jumps to a hardcoded location in imem where the kernel should be placed. This is not optional - the outer loop (over groups) is done by device_main.c, only the inner loop (over local_size_{x,y,z}) is done by the work group function produced by the pocl kernel compiler. Compilation interface (``compile_kernel()`` callback). If this is not implemented, you must have another way to produce pocl binaries offline.

5 Programming AlmaIF Devices on the ALMARVI Demo Platform

The previous chapter explained the internals of how the AlmaIF pocl driver functions. This chapter focuses on the user aspects: How to actually run OpenCL applications on the ALMARVI platform (illustrated in Figure 3).

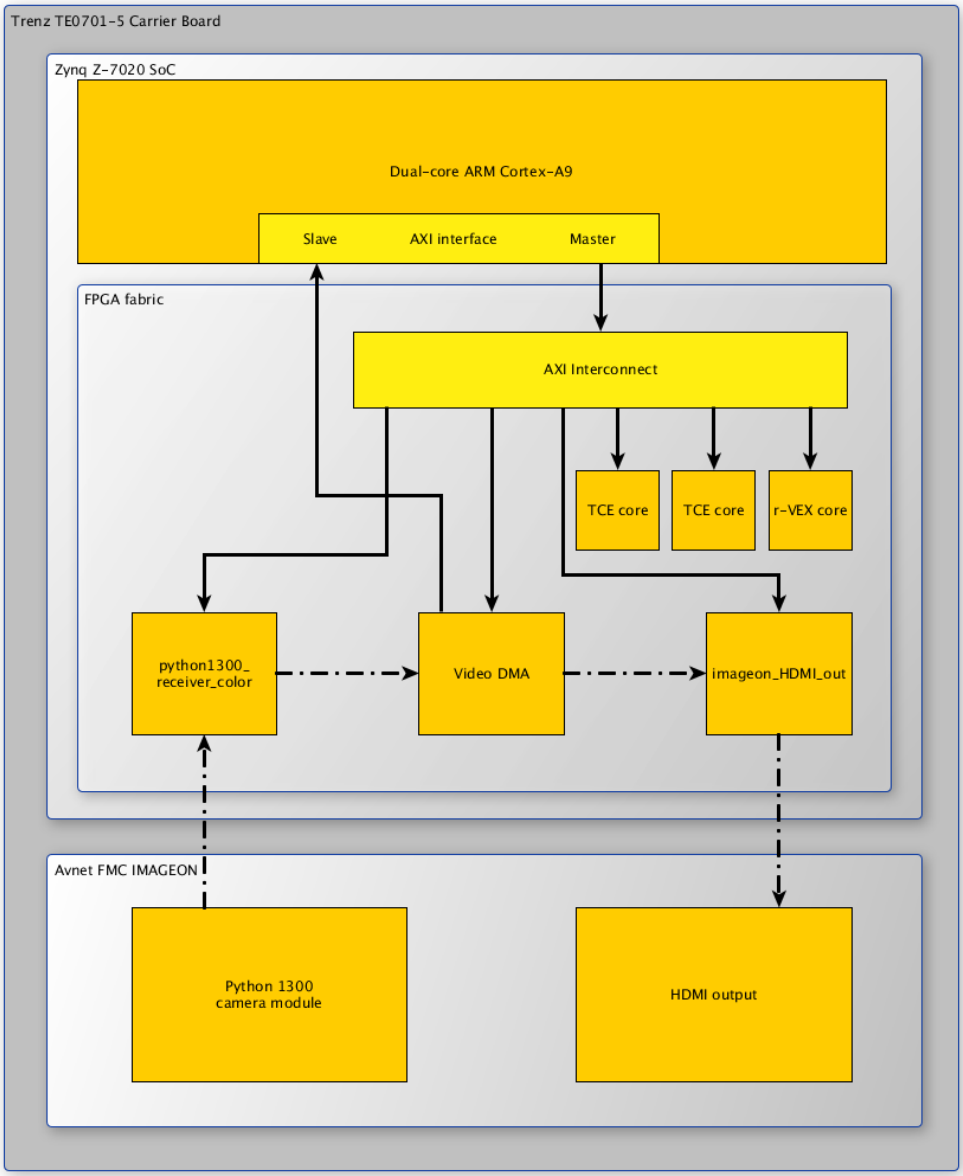


Figure 3: An ALMARVI Platform Instance.

5.1 Petalinux Setup

With the board support package for the target board, you can create a template Petalinux project and configure it:

source ./bin/petalinux/settings.sh

petalinux-create -t project -s <Path to BSP>

cd te0720-plnx

petalinux-config -c rootfs

The menu-based configuration utility allows you to choose which packages to install to the target root filesystem. Petalinux 2016.1 offers one pocl dependency which is not installed by default -- libltdl -- but this is not present in Petalinux 2015.4. In addition, standard C++ libraries are needed for the demo application.

Build the root filesystem with the command

petalinux-build

and generate the bootloader image with the command

petalinux-package --boot --format BIN --fsbl <FSBL image> --fpga <FPGA bitstream> --u-boot

These will create **BOOT.bin** and **image.ub** files in **images/linux** which will later be used to boot Petalinux on the target board.

5.2 Offline-Compiling the OpenCL Binaries

Building prebuilt libraries for AlmaIF requires a pocl with appropriate device support. For example, pocl with TCE compilation support can be built with

cmake -DENABLE_TCE=1 -DWITH_LLMV=<Path to llvm-config>

To compile OpenCL binaries for TCE cores, you will need to set environment variables to specify a TCE device:

export POCL_DEVICES=ttasim

export POCL_TTASIM0_PARAMTERS=<Path to .adf>

Once the device has been specified, binaries can be built with

./bin/poclcc -o <output-filename> <source>.cl

5.3 Cross-compiling pocl for the ARM Host

The cross-compilation requires further dependencies -- hwloc and libltdl -- not present on the Petalinux root filesystem. This can be done by following the documentation of the respective projects, and requires only minimal configuration and tweaking. You will need to point then to the root filesystem under the petalinux project in **build/linux/rootfs/targetroot**.

The runtime libraries use a device build hash to match binaries to devices. Currently, the build hash is hardcoded in the **almaif_device_infos** –struct in **lib/Cl/devices/almaif/almaif.c** and needs to be edited for TCE cores to match the architecture. The build hash can be seen with

poclcc -l

on the development machine.

To cross-compile pocl, you will need to configure a toolchain file for the target platform. A template can be found in **pocl/ToolchainExample.cmake.** Cmake will use the file to locate the appropriate cross-

compiler and target root filesystem. After editing the template, llvm-less pocl can be configured with a command like

**cmake -DHOST_DEVICE_BUILD_HASH=tce-ECDNLBKOKLAFKOHBCAOHKLJGNKOFBBDEMBEJGELC_ -DOCS_AVAILABLE=0 **

**-DENABLE_ALMAIF=1 -DCMAKE_TOOLCHAIN_FILE=Toolchain.cmake **

-DLLC_TRIPLE=arm-linux-gnueabi -DLLC_HOST_CPU=armv7a ..

and built with **make**.

5.4 Writing and Running OpenCL Programs for the AlmaIF Platform

You can now link your OpenCL program against the cross-compiled libraries. The kernel binaries should be read to a buffer and passed to **clCreateProgramwithBinary**. Furthermore, OpenCL buffers and scalar kernel arguments need to be byteswapped, if the platform and AlmaIF core differ in endianness.

At this point, you should have

- A bootloader binary (BOOT.bin),
- Petalinux root filesystem image (image.ub),
- pocl runtime libraries and their dependencies compiled for the ARM host,
- an OpenCL program, and
- precompiled OpenCL binaries.

Copy these files over to the target system and let it boot. Once booted, copy the libraries to /usr/lib. Some environment variables need to be set on the target device for the pocl runtime to find the AlmaIF devices:

export POCL_DEVICES=almaif

export POCL_DEVICES_ALMAIF=Ox43c00000,0x43x20000

You will need to replace the addresses with the relevant offset addresses, which are listed in Vivado's Address Editor for the hardware design project. The demo program takes as arguments the compiled OpenCL binaries. To run the program with a r-VEX core as the first device (sobel) and a TCE core as the second device (blur), launch the program with

sobel.elf tce.bin tce.bin

This should display the image filtering pipeline with input read from the camera and results displayed in the HDMI display with the two example soft cores performing a single step in the kernel. This is all that is required to program AlmaIF devices with OpenCL standard programs.

6 Dynamic Process Loader

6.1 Introduction

Dyplo or DYnamic Process LOader from Topic Embedded Products is a commercially available development environment API and IP, or middleware solution. Within Philips we have been using this alternative in the Almarvi project for several reasons (as previously described in Deliverable 3.8).

In this chapter we will describe the Dyplo software integration with our impelementation and the run-time re-configuration. The Dyplo hardware architecture and Development Environment have previously been described in Deliverable 3.8.

6.2 Dyplo Eco-System

A detailed description of the Dyplo eco-system was shown in Deliverable 1.5; Figure 4 shows a hardware model abstraction for a Zync device where we identify 1) the ARM processor (PS) with software nodes / processes; 2) a CPU node connecting the ARM processor (PS) to the Dyplo IP in the programmable logic (PL) of the FPGA; 3) hardware nodes in the Dyplo network (fixed or reconfigurable); 4) input/output nodes providing an interface to programmable logic in the FPGA outside the Dyplo network.

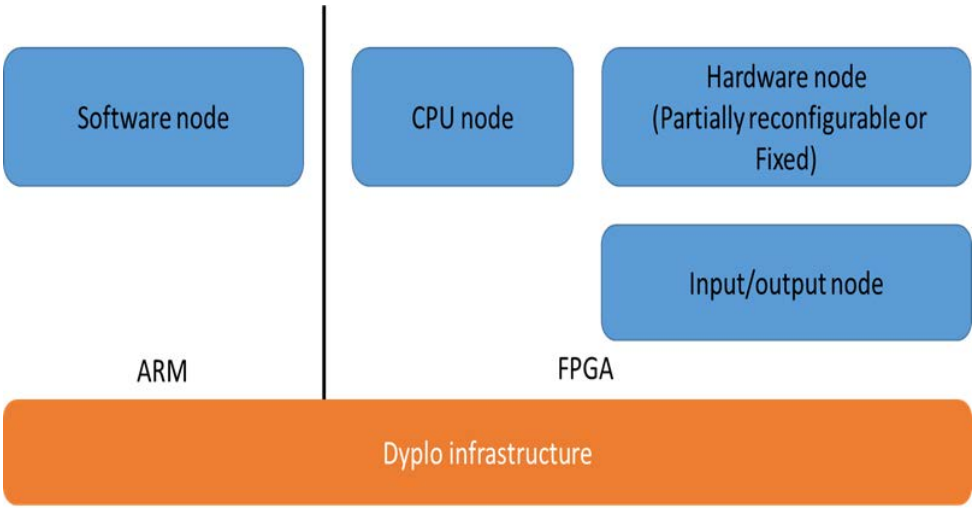


Figure 4: Dyplo hardware model abstraction

6.3 Programming Model

An abstraction of the Dyplo programming model is shown in Figure 5. The ARM processor is running an OS, in this case Open Embedded Linux. Topic provides a device tree for Open Embedded and Yocto Linux and a Dyplo driver, library and utilities on github [2] . The driver creates the interface to the Dyplo hardware, the library provides an abstraction of the interface allowing the programmer to use data queues on both the software and the hardware nodes. The programmer creates an application in the OS where the Dyplo libraries are used. The static bitstream is loaded at start-up of the Zync device, the partial bitstreams are programmed by the user through the application.

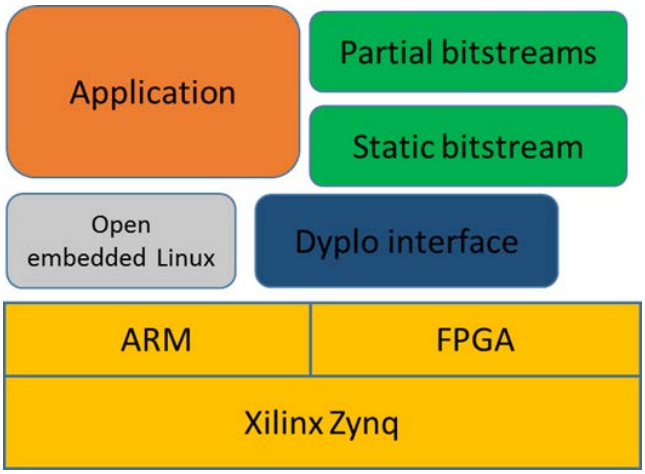


Figure 5: Dyplo programming model abstraction

6.3.1 (re-) Routing and Partial FPGA (re-)Programming

We consider the situation as shown in Figure 6 below:

1) I/O nodes to connect external Inputs c.q. Outputs (e.g. video frames) to the Dyplo network
2) Fixed nodes for Pre-/Post-Processing and a Procedure A (e.g. a certain filter on a video frame)
3) A Partial Reconfigurable (PR) area with a PR node Procedure B or C (e.g. a different kind of filter)
4) An application running on the processor which, through a CPU node (not in the figure), controls the connection from Pre-Processing to Procedure A or Procedure B/C and from there to Post-Processing and programs the PR node with Procedure B or C. We would like to be able to switch the Procedure, which is being performed in the hardware, at runtime.

A programming example can be found at:
https://github.com/topic-embedded-products/dyplo-example-app/blob/master/dyplodemoapp.cpp [2]

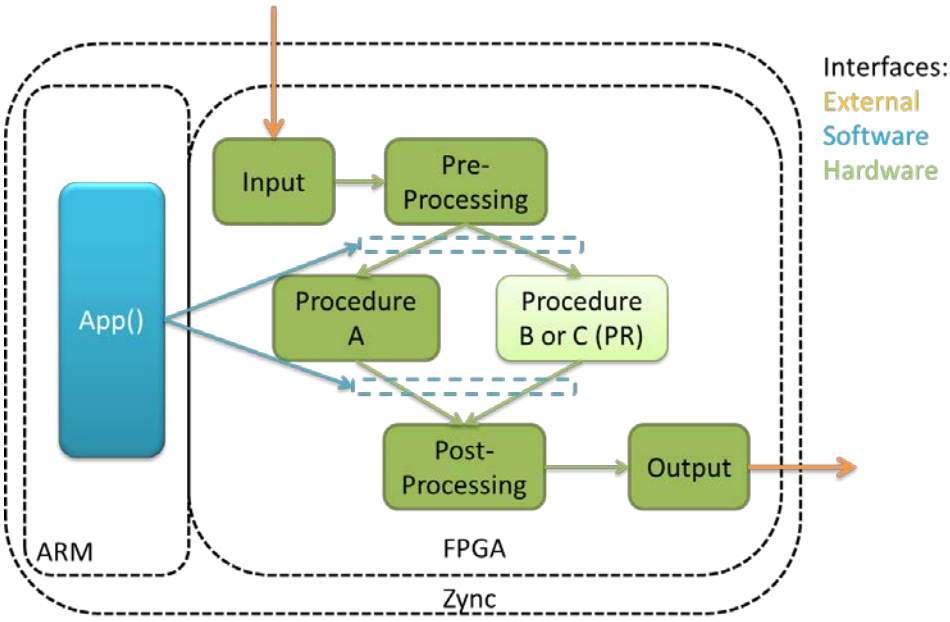


Figure 6: Dyplo re-routing and re-programming use case

6.3.2 Interfaces

The default interface which is being used to communicate between software and hardware in the Dyplo programming model is a CPU node interface. From software point of view the interface is similar to a data queue or FIFOs in between software processes / nodes and therefore allows for easy interchanging of software and hardware functionality. This configuration is shown in Figure 7, the *CPU Node* and *Connect* together model the Dyplo network.

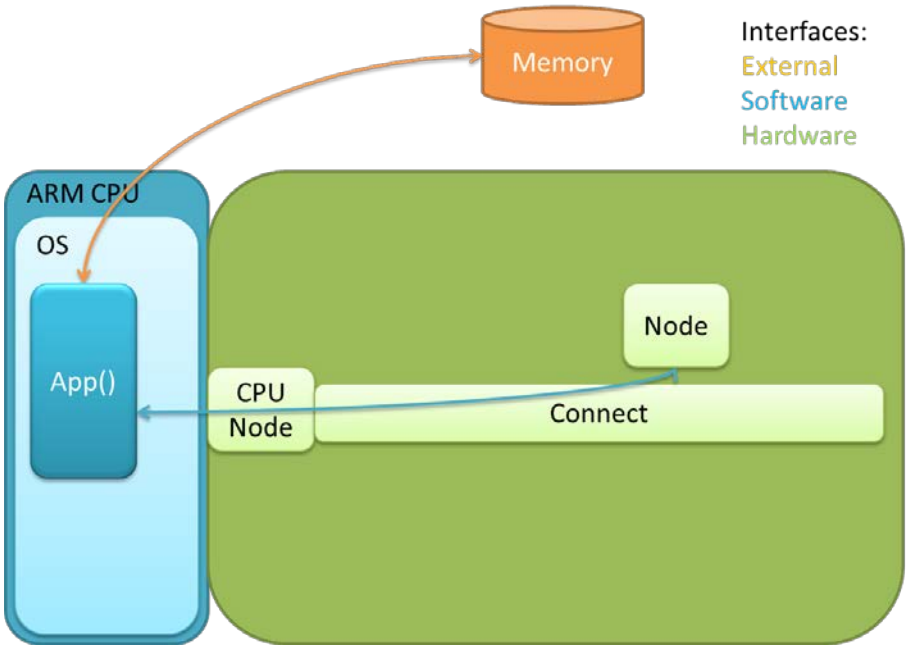


Figure 7: Inter software to hardware communication

Higher data rates can be achieved by using zero copy Direct Memory Access (zDMA). The external memory is shared between the Zync Processor (PS) and the FPGA fabric (PL) and can therefore easily be used to transfer memory with a DMA controller as IP connected to the Dyplo network. Using the DMA controller in zero copy mode allows to only pass on a pointer (memory address) from the software to hardware side instead off copying the total memory being shared. This configuration is shown in Figure 8.

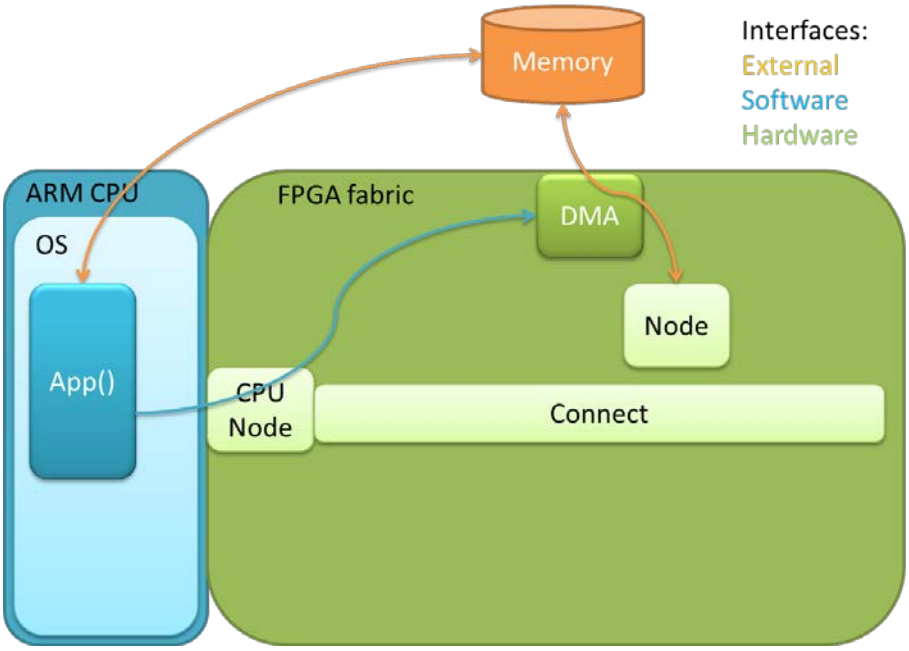
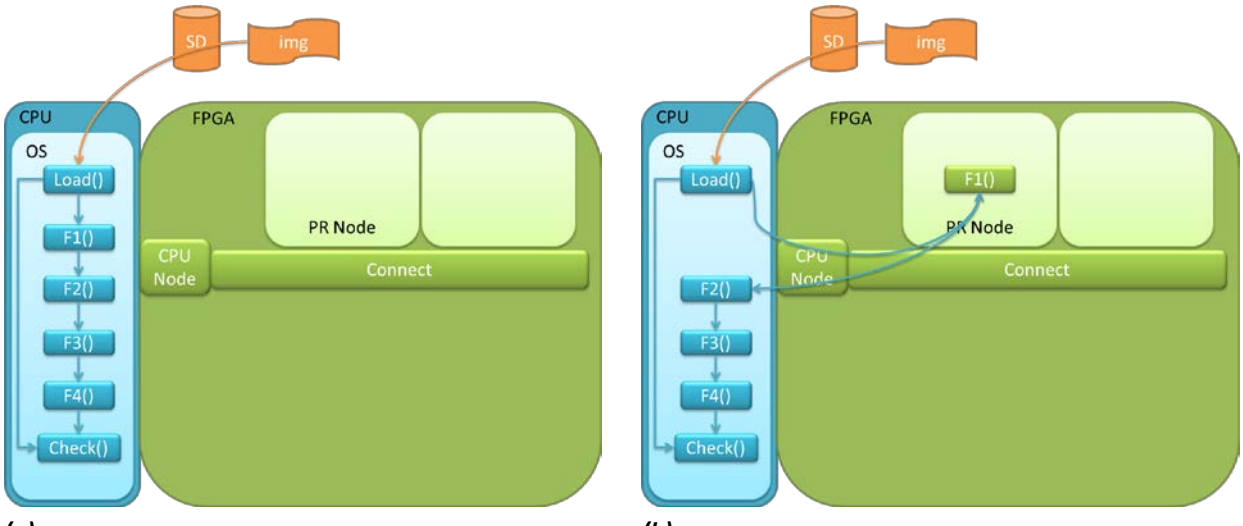


Figure 8: Inter software to hardware communication through the DMA controller

6.4 Software Integration

This section discusses how to move from an all software implementation to an all hardware implementation, this is especially useful as a development setup. Figure 9 (a) shows the begin situation where 4 functions *F1()* through *F4()* have been connected sequentially as Dyplo software nodes, the *Load()* and *Check()* function together form a test bench. Figure 9 (b) shows the first step where *F1()* is moved to a PR area in the hardware using High Level Synthesis Tooling and the Dyplo Development Environment as described in Deliverable 3.8. This requires little effort on the application side, since the interfaces of *Load()* and *F2()* are already compatible with the hardware interface. The result can be tested due to the test bench on the application side. The functions *F2()* through *F4()* are moved to the hardware in a similar way. Figure 9 (d) through (f) show that multiple functions are combined into one PR area to save resources; we could have created 4 PR areas on the FPGA instead. Finally the test bench might be replaced by an external interface as shown for example in Figure 6.



(a) ***(b)***

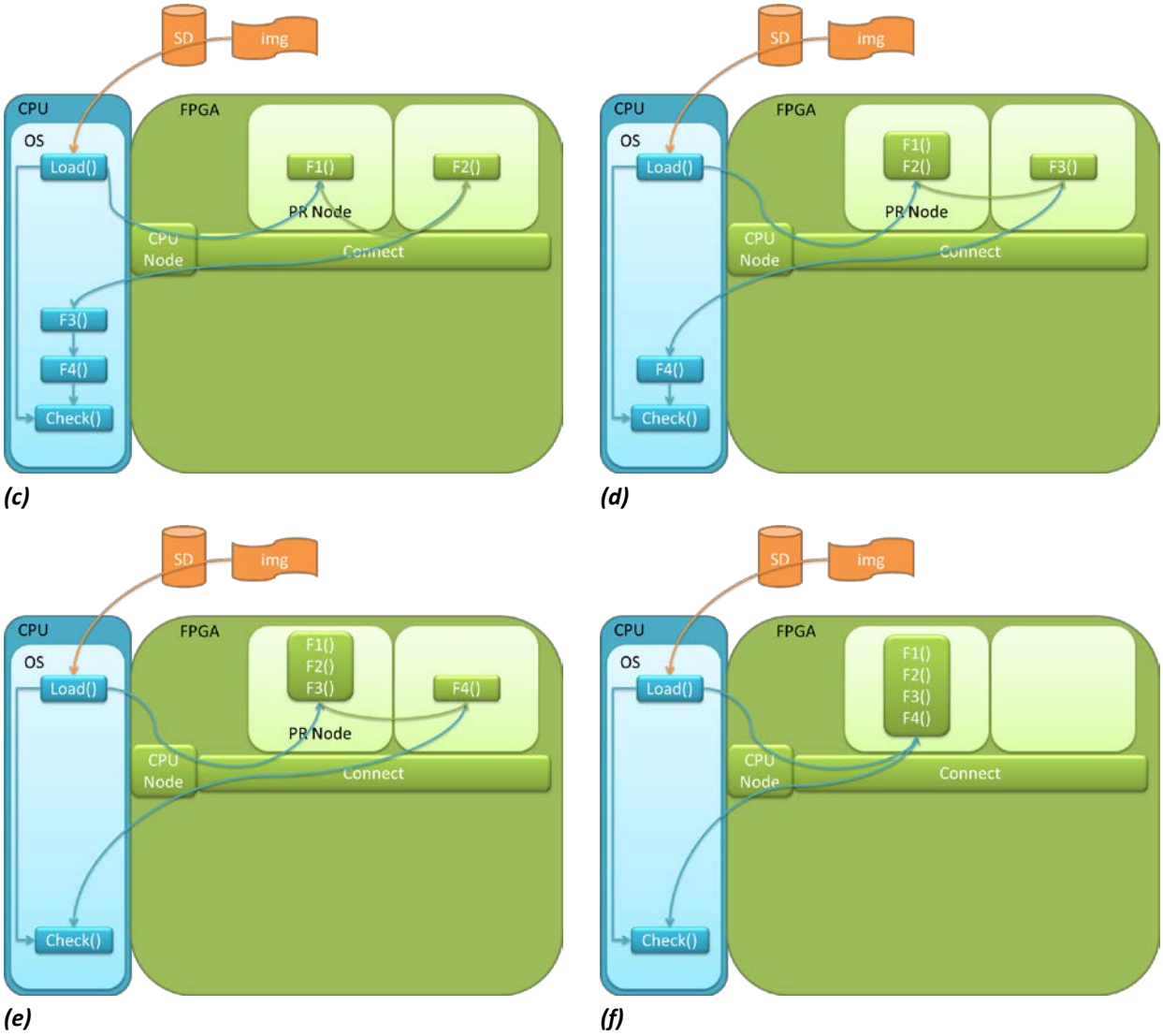


Figure 9: Software accelerator integration phases

7 Scenario-Aware Dataflow Programming Models for Streaming Applications

7.1 Introduction

7.1.1 Problem Statement

Many contemporary real-time (RT) applications are streaming, meaning that data is processed when it is received rather than read on demand from local storage. The control flow of such applications often depends on the received data and can vary per iteration. Upon receiving a video frame for example, a decoder will detect if it is a full frame or delta frame and call the appropriate decoding function. A programmer may solve such a dependency with a simple *if-else* construct in a sequential language as shown in Figure 10.

```
1: frame = buffer_frame()
2: if detect_frame_type(frame) = full then
3:     x = decode_full(frame)
4:     sub = subtitle_overlay(x)
5: else
6:     x = decode_delta(frame)
7: end if
8: output = construct_frame(x)
9: display(output, sub)
```

Figure 10: Pseudo-code of an abstract video decoder.

The dataflow model of computation (MoC) is a natural way to describe data-dependent behaviour [3]. In particular, finite state machine scenario-aware dataflow (FSM-SADF) can capture different input-dependent control flows in *scenarios* [4]. A *scenario graph* must be created manually for each scenario by the programmer, possibly based on existing sequential code. The scenario graph for decoding a full video frame (S_{full}) is depicted in Figure 11. When a delta frame is detected, the application behaviour and thus the scenario graph (S_{delta}) is different, see Figure 12. Actor names are abbreviations of the functions in Figure 10. All allowed scenario sequences are specified by the FSM in Figure 13. The graphs will be further explained in Subsection 7.3.2.

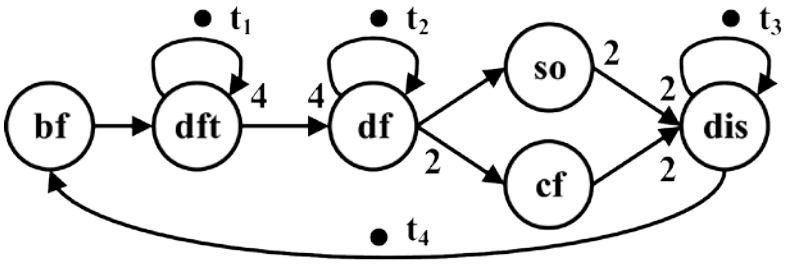


Figure 11: Scenario graph for decoding a full video frame, S_{full}.

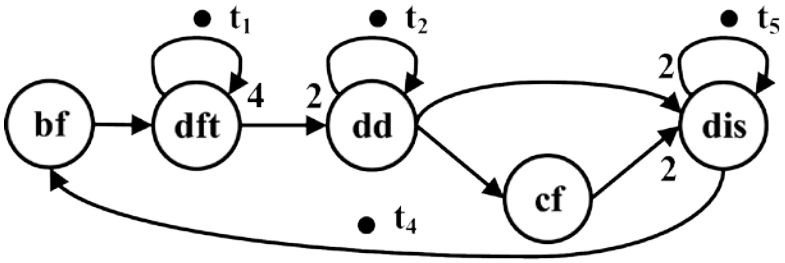


Figure 12: Scenario graph for decoding a delta video frame, S_{delta}.

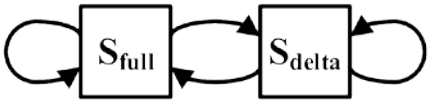


Figure 13: The FSM of the video decoder with scenarios S_{full} and S_{delta}.

While FSM-SADF allows tight analysis of applications whose control flow is input-data dependent, it is not an execution model. Consider the video decoder, where `bf` and `dft` are always executed first. Only after executing `detect_frame_type()` the next scenario is known. At runtime the system cannot decide which of the scenario graphs to start with because that information is not yet available. For analysis this is not a problem, but during execution it presents a causality dilemma because the next scenario is detected only *after* the next scenario graph is partially executed. Another difference between analysis and execution concerns the transfer of state (persistent tokens t_i) between scenario graphs. Consider t_1, which stores the bitstream position required by the frame detection. Token t_2 on the other hand stores motion vectors required by the different decoding functions. During analysis the value of t_2 is irrelevant, but at runtime it is state that must be transferred to the other graph once a change of scenarios is detected.

7.1.2 Contribution

In this work we propose a concept for sequencing scenarios that can be both analysed and executed. We require the programmer to identify an identical subgraph in each scenario, after whose execution the next scenario is known. This subgraph, {`bf`, `dft`} and {t_1, t_4} in our example, is then automatically split off in a detector scenario S_{det}, see Figure 14. After execution of S_{det} the next scenario is known, and t_2 must be transferred before executing the next scenario graph, e.g. S_{full} depicted in Figure 15. This solves the causality dilemma and is explained in detail in Subsection 7.3.3. We model the exact timing impact of this solution on the platform in our analysis. This allows us to provide timing guarantees on the application when executing a sequence of scenarios on the CompSOC hardware platform developed at TUE.

We present the following contributions:

- a scenario sequencing concept for FSM-SADF that is both executable and analysable in Section 7.3;
- a corresponding method for scenario execution with novel scheduling concept in Section 7.5;
- a platform-aware analysis model in Section 7.5.

The background and FSM-SADF in general are discussed in Section 7.2. The contributions are validated experimentally in Section 7.6, conclusions are presented in chapter 9.

7.2 Background

Several models of computation are suitable for programming real-time applications, but many of those enforce conservative assumptions when encoding input-dependent behaviour. The time-triggered (TT) MoC for example applies time-triggered static-order scheduling, which is non-work conserving [5]. As task scheduling is triggered by time rather than data, input-dependent behaviour cannot be encoded in the high-level control flow. There are multiple flavours of the dataflow MoC, which are work-conserving provided that a suitable scheduling method is applied [3]. For analysis of dataflow graphs the SDF^3 tool is available [6]. However, many dataflow types such as synchronous dataflow (SDF) are unable to respond to input-dependent behaviour. Consider the code in Figure 10, where `decode_full()` might have a longer WCET than `decode_delta()`. Yet due to data dependencies invisible at this level of control flow, the WCET of `construct()` might be longer for a delta frame. The aforementioned MoCs will consider the WCET of `decode_full()` and longest WCET of `construct()` at the same time although this situation will never occur. This results in a throughput bound that is overly negative.

Dataflow graphs capture the control flow of an application, so different behaviours can be encoded if graph parameters are allowed to vary each iteration (full execution) of the graph. Two dataflow models support this: FSM-SADF and mode-controlled dataflow (MCDF) [7], [4]. MCDF leverages a mode controller actor that triggers execution of a subgraph. It is a programming model suitable for timing analysis, but no analysis tools are publicly available.

The expressiveness of FSM-SADF that was developed at TUE is similar to MCDF, but it does not specify a programming model. Analysis considers a unique WCET per scenario per actor. This allows to exclude combinations that can never occur, such as `decode_full()` and the `construct()` WCET for a delta frame. Thus the throughput bounds are tighter than with SDF [8]. A bound on the worst-case throughput of the whole application is determined by considering the worst possible scenario sequence. We have selected FSM-SADF for this work because an analysis tool also developed by TUE is publicly available (SDF^3), which we extended in the context of this work.

An FSM-SADF programming model has been presented that also splits off detector scenarios, similar to our concept [9]. However, these consist of one actor rather than a subgraph. Moreover, that work proposes to merge all scenarios and execute the entire resulting graph while sending an additional scenario identifier (ID) token to all the actors. The actor bodies (functions) are executed conditionally based on that ID. While this preserves dataflow execution semantics, it causes a considerable overhead in large graphs, which we avoid here. Also, it does not present an analysis model such as presented in Section 7.5.

Applications are mapped to a platform to validate the dataflow concept proposed in this work. Platforms for which dataflow execution library is available are SoD and CompSOC [10]. Because a homogeneous synchronous dataflow (HSDF) timing analysis model is available for the latter [11] and because it was developed in-house at the TUE, we decided to use the CompSOC platform. The libDataflow library implements dataflow scheduling and offers an actor wrapper, while libFIFO provides first-in first-out (FIFO) channels.

7.3 Scenario Sequencing Concept

7.3.1 Motivation

Section 7.2 explained how FSM-SADF analysis acquires tight bounds on the throughput of applications with input-dependent behaviour. It also enables programming and parallelizing such dynamic applications in a natural way. Because each scenario graph can have a different topology and different rates, WCETs and persistent tokens, multiple application behaviours can be encoded. For example, one can capture different control flows by varying the graph topology, thereby also changing the mapping of an application on the platform. This can be used to vary the degree of parallelism in an application to deal e.g. with varying resource availability or quality-of-service requirements. Different rates can be applied to maintain throughput when dynamic voltage and frequency scaling (DVFS) is used. Modelling different WCETs is useful for functions such as variable length decoding for different frame types. A scenario switch can be triggered by the outside world (incoming data, user input, sensors) or internal platform properties (temperature, faults, frequency). Executing multiple scenarios after each other, including S_{det}, is the topic of this section.

7.3.2 Semantics

The full frame scenario depicted in Figure 11 is described by a graph with predefined topology and rates. The nodes represent the actors named `{bf, dft, ...}`, each with a known WCET. The edges represent channels, the rates at their start and end indicate how many tokens are produced and consumed each time the actors fire (execute). Persistent tokens indicated with t_i can appear on any edge and are present at the start and end of each iteration of the graph.

The delta frame scenario is depicted in Figure 12, and the scenario sequences that are allowed in Figure 13. Some actors occur in both scenarios (`bf, dft, cf` and `dis`), possibly with different rates. These rates may result in a different repetition vector, which is the number of times an actor fires each iteration. Other actors occur only in one scenario (`df, dd`, and `so`), but might consume a persistent token that also occurs in another scenario.

In short, tokens and actors can be connected at will in different scenarios, allowing programmers to express coarse-grained changes in the control flow. We pose a constraint in the context of this work, which is that there must be a subgraph with identical topology that is the longest prefix graph of each scenario. This subgraph must consist of at least one actor, the repetition vectors of the actors must be identical in each scenario, and after execution of the subgraph the next scenario must be known.

7.3.3 Switch and Select

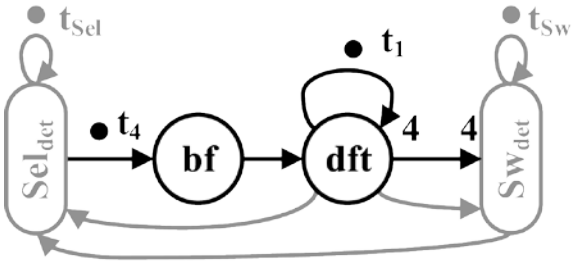


Figure 14: Analysis graph of the detector scenario S_{det}.

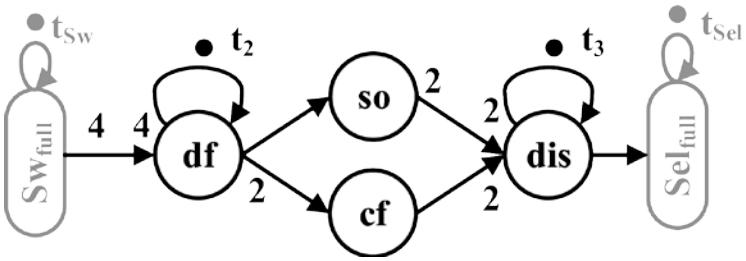


Figure 15: Analysis graph of the full frame scenario S_{full}.

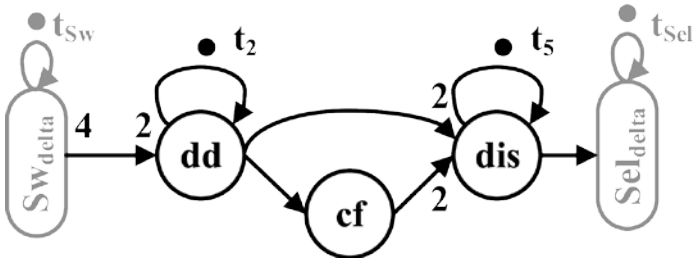


Figure 16: Analysis graph of the delta frame scenario S_{delta}.

Figure 17: The extended FSM with detector scenario S_{det}. The starting state is indicated with a dot.

The last section introduced a solution to the causality dilemma by splitting off the detector scenario S_{det} depicted in Figure 14 from the original scenarios. Identification of the detector subgraph is a task of the programmer for now, but its validity is automatically verified. The FSM is automatically transformed accordingly; S_{det} is now executed before each original scenario. See Figure 17, the starting state is indicated with a dot. To transport tokens from and to the detector scenario, we propose to leverage *switch* and *select* actors. We instantiate these on the outgoing edges (switch) and incoming edges (select) at which the scenario graphs are split, and assign them a WCET of zero. This results in the analysis models shown in Figure 14, Figure 15 and Figure 16. The Sw and `Sel` actors are indicated in grey, including the channels and tokens necessary for analysis as explained next. The analysis of decoding a full video frame with the extended SDF^3 tool proceeds as follows:

1. execution starts in S_{det} , `bf` fires followed by `dft` which detects scenario S_{full};
2. `dft` produces its data tokens on the black edge to Sw_{det} and a control token with the scenario ID onto each grey edge;
3. the firing of Sw sequences the correct scenario:
    a. Sw_{det} consumes all of its tokens;
    b. the timestamp at which t_{Sw} is consumed is registered by SDF^3 , execution of S_{det} halts [4];
    c. the timestamp of t_{Sw} is transferred to t_{Sw} in S_{full};
    d. execution of S_{full} starts with firing Sw_{full} , because both Sw actors have zero execution time no time has passed between the start of Sw_{det} and the end of Sw_{full};
    e. Sw_{full} finishes by producing the data tokens into the FIFO towards `df`.
4. S_{full} executes as normal;

5. when t_{Sel} is consumed by Sel_{full}, execution of Sw_{full} halts and the timestamp of t_{Sel} is transferred to S_{det};
6. Sel_{det} fires and produce the new t_4 for the next iteration.

The analysis graph of a detector frame is similar.

7.4 Scenario Execution

The scenario sequencing concept for analysis presented in Section 7.3 applies to the existing FSM-SADF MoC. Analysis of the split scenario graphs yields identical results to analysis of the original graphs. From the implementation perspective, the split solves the causality dilemma because the next scenario graph is only started after it is detected. In this section we present solutions to practical aspects encountered when the MoC concept is implemented as model of execution (MoE):

A. executing a sequence of scenarios;
B. extending static-order actor schedules on-the-fly;
C. sharing persistent tokens (state) between scenarios.

7.4.1 Executing a Sequence of Scenarios

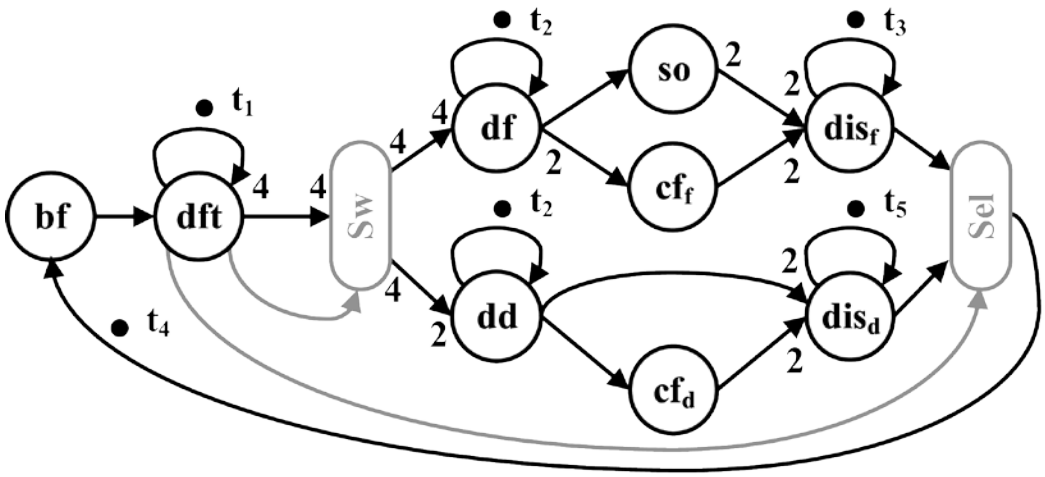


Figure 18: The merged scenario execution graph, the switch and select actors are indicated in grey.

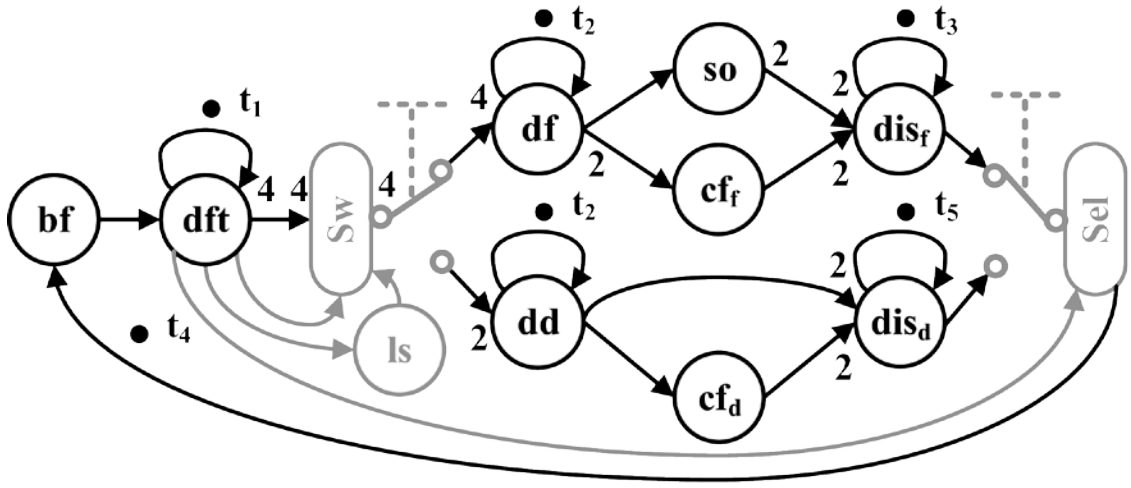


Figure 19: The merged graph with implementation details, the `switch/select` and load schedule (`ls`) actors for a one-processor mapping is indicated in grey. The swapping of FIFO channels in indicated with the electrical symbol for a switch, also in grey. Note that this is not a valid dataflow graph.

For execution, we propose to implement the switch and select actors as multiplexer and de-multiplexer respectively. This glues the boundaries of the scenarios together again, with the difference that now there is only one detector subgraph serving both S_{full} and S_{delta} with tokens, as shown in Figure 18. Actors `Sw` and `Sel` are indicated in grey. On the MoC level, such (de-)multiplexing solves the sequencing of scenarios during analysis. Synchronisation tokens t_{Sw} and t_{Sel} can be omitted for execution.

Such (de-)multiplexers are not available in current dataflow libraries. Consider a switch, whose data tokens on the input port are forwarded to one of the output ports. The rate on the other output is effectively zero. Both changing rates within a scenario and rates with value zero are not part of the FSM-SADF formalism. This is a practical issue in the MoE only, as the models in Subsection 7.3.3 fully adhere to the FSM-SADF MoC.

We propose a solution that exploits the fact that libFIFO buffers can be disconnected and reconnected without invalidating data. A switch is given just one output port, to which the proper channel is connected depending on the detected scenario, see Figure 19. This swapping of FIFO channels is indicated with the symbol for an electrical switch, which connects the single output port either to the channel to `df` or that to `dd`. The other channel is left unconnected on one end, effectively giving it rate zero. Select actors are similar but demultiplex two channels to one. The behaviour during runtime can be matched to the analysis steps described Subsection 7.3.3 as follows:

1. `Sw` fires and consumes all of its tokens (3a, 3b);
2. the output port of `Sw` is connected to the FIFO towards `df` (3c, 3d);
3. `Sw` ends firing by producing tokens into that FIFO (3e);
4. execution of S_{full} continues as usual (4).

Before an actor is fired, the libDataflow library checks if all the required input tokens are available and if there is sufficient space in the outgoing FIFOs for the actor to produce its tokens in. Therefore the FIFO swap must take place before the firing rules of `Sw` are checked.

7.4.2 Extending Static-Order Schedules

The libDataflow library executes dataflow graphs by iterating over the static-order (SO) schedule that is given by SDF^3, and blocks if an actor is not ready to fire. Unlike existing dataflow implementations, the SO schedule of our proposed solution changes depending on the detected scenario. Therefore we introduce a new scheduling concept that we dub the *rolling static-order* (RSO) scheduler.

Execution starts with S_{det}, so if we were to map the decoder to a single processor the SO schedule starts with [`bf, dft, Sw`]. After firing `dft` the next scenario is known and the SO schedule can be extended. Should S_{full} be detected, the sequence [`df, cf`$_f$`, cf`$_f$`, so, dis`$_f$`, Sel, bf, dft, Sw`] must be concatenated to the "rolling" schedule. Note that `Sel` comes at the end of S_{full} , and we immediately concatenate the next detector scenario. In this way it is ensured that the scheduler will never run out of actors to schedule. A multi-processor mapping works similarly, the only constraint we impose is that `dft` and `Sw` must be mapped onto the same processor.

The RSO scheduler was implemented in libDataflow. We initialize it with a unique SO schedule for each scenario. The start is set to S_{det}, which ensures that `Sw` will be executed. An additional load schedule (`ls`) actor is inserted right after `dft` on every processor, see Figure 19. These `ls` actors receive a scenario ID token from `dft` and extend the schedule accordingly. The example schedule of S_{det} changes to [`bf, dft, ls, Sw`]. We furthermore exploit the `ls` actor to connect all FIFOs correctly before the firing rules of `Sw` or `Sel` are checked. This dependency is visualized with a grey edge in Figure 19.

7.4.3 Sharing Persistent Tokens

Lastly, persistent tokens that appear in different scenarios must be transferred upon a scenario switch. We solve this by mapping ports from multiple actors to the same FIFO. This is another useful property of libFIFO and avoids actual data transfers and synchronisation issues. Each FIFO holding a shared persistent token is instantiated only once, and is connected to the appropriate actor in every scenario the token appears in. This leads to the requirement that the actors at both ends of the FIFO must be mapped onto the same processor in each scenario. The RSO scheduling ensures that actors connected to this FIFO can never execute simultaneously or "overtake" each other.

7.5 Platform-Aware Analysis Model

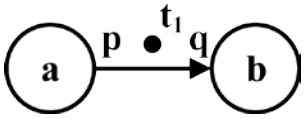


***Figure 20: Graph with two actors* a *and* b*, connected by a FIFO with production rate* p*, consumption rate* q *and persistent token* t_1.**

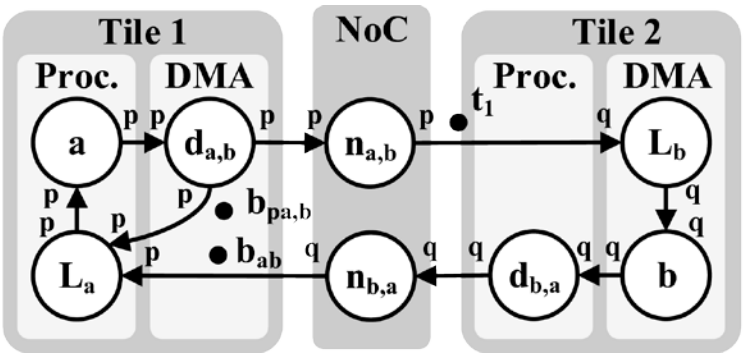


Figure 21: CompSOC binding-aware model of the graph in Figure 20 with both actors mapped to different processors, based on [11]. `Proc` ***denotes processor,*** $b_{p,ab}$ ***and*** b_{ab} ***model the source buffer and the FIFO buffer.***

Section 7.3 introduced the scenario sequencing concept in the MoC, Section 7.4 described a matching Model-of-Execution. To ensure correct timing analysis we re-visit the MoC, and propose a platform-aware analysis model with which the exact timing impact of our solution can be analysed. The SDF^3 tool generates different mappings from a set of storage distributions. The throughput of each mapping is analysed after converting the mapped scenario graph to a binding-aware graph (BAG) [12]. This means the graph is annotated with timed models of the hardware and software platform components such as the Network-on-Chip (NoC), direct-memory access units (DMAs) and the SO actor schedule. We will now discuss how each of the three changes in Section 7.4 is modelled in the BAG.

The principle of sequencing scenarios was presented in Subsection 7.3.3. The timing impact of `Sw` and `Sel` actors is modelled as follows. Firstly, the time required by the `Sw` and `Sel` actors for simply forwarding the data tokens is added to their execution time in S_{det}. Secondly, the time it takes to connect the correct FIFOs to these actors is added to the `ls` actor on that processor. The time for extending the rolling SO schedule is also added to `ls`. Connecting multiple actors to one FIFO that holds a persistent token reduces the memory footprint.

SDF^3 can currently map and analyse SDF and cyclo-static dataflow (CSDF) applications onto a CompSOC platform using an HSDF platform model [10], [11]. In the context of this work we extended the HSDF platform model for FSM-SADF analysis. The key difference is that HSDF is mono-rate, meaning that all rates on all edges have a value of one, but scenario graphs are multi-rate. Let us consider the example shown in Figure 20. If the actors are mapped to different processors, the edge is replaced by a combined model of the DMA, NoC and CoMik microkernel as explained in [11]. We annotate the original HSDF model with rates as shown in Figure 21, which also shows the location of persistent token t_1 in the model.

A consequence of multi-rate graphs is that actors can have a repetition vector larger than one. This complicates the schedule encoding in the BAG because multiple actors might be enabled (ready to fire) at the same time and the encoding must ensure that only the scheduled actor fires. The original HSDF schedule encoding cannot do this, so we integrated an existing technique that can in the FSM-SADF toolflow [12]. The actors modelling the DMA units (see Figure 21) follow the schedule of the actors whose FIFO they model, therefore these must be encoded using the same method. This introduces additional complexity because there can be multiple DMA actors per processor. We add simple dependency edges between these that follow the order in which the tokens are consumed or produced. This limits the number of enabled actors for the schedule encoding, simplifying the BAG. However, as DMA actors belonging to different actors have no such implicit dependencies, the schedule encoding considers all of them enabled at the same time. To enforce an order, tokens are inserted in the BAG which can slow down analysis for graphs with high rates.

Once a mapping is selected, the scenario graphs are merged together automatically which results in the graph shown in Figure 18.

7.6 Experimental Evaluation

7.6.1 Setup

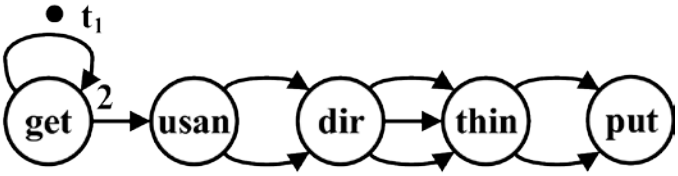


***Figure 22: Scenario graph of the sequential* SUSAN *scenario,* S_{seq}.**

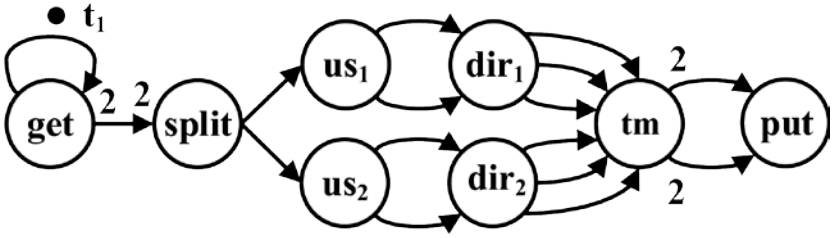


***Figure 23: Scenario graph of the parallelized* SUSAN *scenario,* S_{par}.**

To validate the concept, implementation and model we mapped a dataflow version of the SUSAN edge detection algorithm to a two-processor platform [13]. Scenario S_{seq} is a sequential graph similar to the original algorithm, see Figure 22. Scenario S_{par} is a parallelized version, see Figure 23. The algorithm reads an image block by block, one of the graphs must be executed for each block. Scenarios are switched based on the image resolution, the FSM is similar to the one shown in Figure 17. If mapped to two or more processors, scenario S_{par} can potentially achieve a higher throughput because the computationally intensive actors `usan` and `dir` (direction) can be executed in parallel (us_1, dir_1 and us_2, dir_2). To provide both these chains with a block, split should have a consumption rate of two so the `get` (get image) actor would have to execute twice. This violates the constraint that the repetition vectors of actors in the detector scenario must be identical, which we imposed in Section 7.3. Therefore the production rate on the channel from `get` to both `usan` and `split` is set to two, which means each image must contain an even number of blocks. This is an example effect of the constraints that we impose. We instantiate a CompSOC platform with two processors, both clocked with a frequency of 100MHz. Each processor features instruction and data memories of 256kB each as well as one DMA with two communication memories of 16kB each. The processors are connected to each other and to an external DDR memory via a NoC [10].

7.6.2 Results

Figure 24: Original test image (left) and the outcome of* SUSAN*. Note that the outer band of the image is skipped.

The SDF^3 tool takes the scenario graphs annotated with execution times and memory sizes as an input, plus an architecture file. Furthermore a throughput bound must be provided, which SDF^3 will try to meet while mapping the application onto the architecture. The throughput is the inverse of the number of cycles required to complete an iteration of the application. For SUSAN it was found that a maximum throughput of $2.7 \cdot 10^{-7}$ can be met. When the FSM-SADF application was executed on the platform using the implementation described in Section 7.4, a throughput of $3.1 \cdot 10^{-7}$ was measured. The fact that the actual throughput is a little higher than the throughput given by the analysis means that the model is conservative. The graphic of the SUSAN edge detection algorithm is shown in Figure 24.

A load schedule actor takes 365 cycles to execute including the wrapper, the switch actor takes 1635 cycles. This brings the timing cost of our libDataflow modifications to 2000 cycles, which are accounted for in the analysis model. We argue that the impact of scenario sequencing on the applications timing is minimal as the total WCET of SUSAN is 3.3 million cycles. The size of the library is increased by 3kB.

8 Performance Analysis of an Image-Guided Assembly line Prototype

Manufacturing systems have key-performance indicators based on throughput and latency, e.g. the number of products produced per time unit while satisfying certain deadlines in the system. Latency metrics are defined as the time distance between two specified events in the system, for instance between the start of processing and the output of a product. These metrics must be considered in the design-space exploration, scheduling decisions, or for validation purposes. Various system level characteristics and requirement a manufacturing system have similarity with healthcare systems. In this chapter we introduce a number of temporal analysis techniques for an assembly line prototype called eXplore Cyber Physical System (xCPS), a platform (see Figure 25) of industrial complexity for research on CPSs [14]. It embeds a wide range of typical CPS components including many of the common challenges seen in healthcare systems and is used as a vehicle for CPS research. It allows for experimentation, research and development of components relating to different disciplines, as well as multi-disciplinary aspects of complex systems.

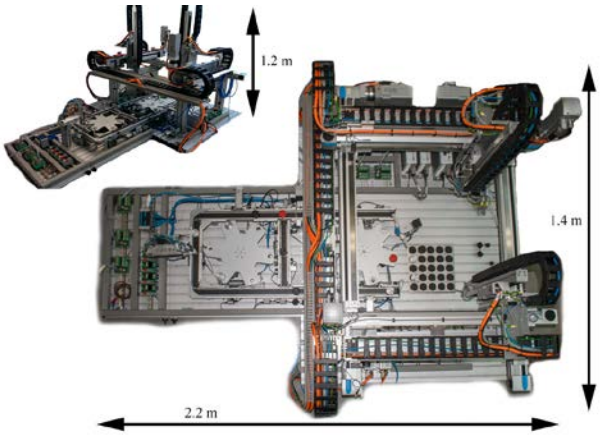


Figure 25 Top and side view of the xCPS platform.

For xCPS, we look at performance analysis at the system level and at the resource level. The particular challenge is to capture the timing behavior of tasks in the product flow taking into account the tasks themselves, the resources they use, as well as their mutual dependencies. For example, the product flow relies on synchronizing events between different actions of actuators or sensors. This can be modeled as tasks with dependencies. Such systems also have resource dependencies when resources are shared between multiple tasks. Besides task and resources dependencies, these systems depend on the pipelining of multiple products to increase performance. This causes additional dependencies between different products, for example while one product is being assembled, other pieces can be already introduced in the system.

We explore the use of data flow models-of-computation, such as Synchronous Data Flow graphs [3] or extensions of this model such as Scenario Aware Data Flow graphs (SADF) [15], since they can naturally capture (cyclic) task dependencies, resource dependencies and pipelined behavior. Task durations (execution times) and synchronizations are also natural ingredients of data flow models. Moreover, data flow models have good analyzability properties due to their deterministic and time-monotone behavior. For the analysis we employ our research data flow analysis tool SDF^3 [6] to determine throughput and latency.

Consider a part of xCPS that takes bottom and top objects, assembles them and outputs the assembled object (see Figure 26). As shown in the Figure 26-(a), top and bottom objects take different paths in the

machine to reach the assembling station, which are shown with red and blue lines. Each path consists of a different sequence of actions, with some resources, namely Source and Movement 1, being shared between paths (see Figure 26-(b)). The durations of the individual actions are known. They can for instance be measured. We can model this system using SADF, in which each path (bottom or top) can be modeled by a specific SDF graph and the possible orderings of bottoms and tops can be expressed as a formal language accepted by a finite state automaton on infinite words as shown in Figure 27. Once the order of the input pieces is decided, data flow analysis results can be used to find and resolve bottlenecks within xCPS.

(a) (b)

Figure 26: Assembly line of xCPS. (a)Bottom and top pieces are entered to the system on the conveyor belt at bottom left side of the picture. The assembled pieces output at top left side of the picture. The paths for bottom, top and assembled pieces are shown with blue, red and yellow arrows.

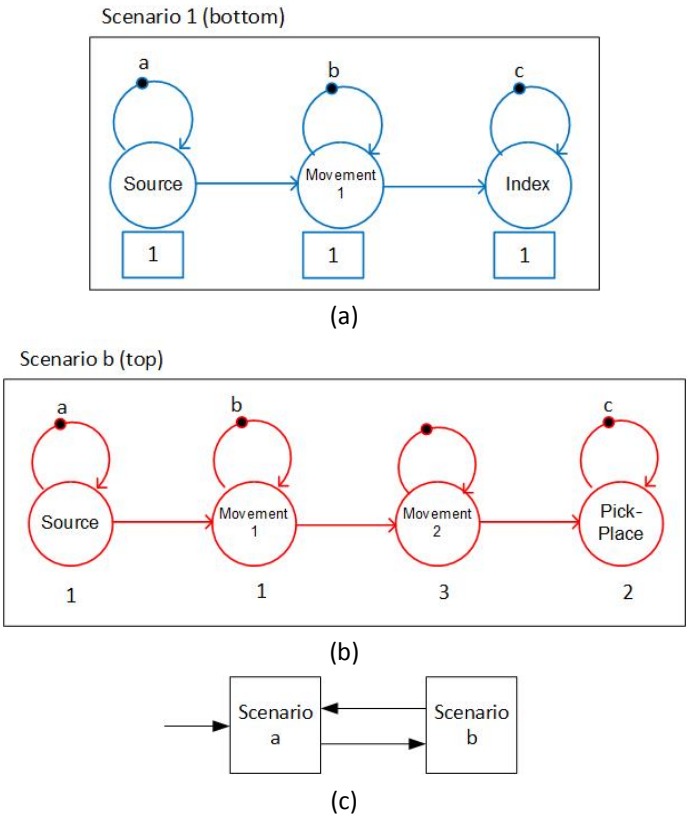


Figure 27: SADF graph of the assembly line. This SADF graph consist of (a) SDF graph of the bottom scenario, (b) SDF graph of the top scenario and (c) finite state machine indicating the possible transitions between the two scenarios.

With the SADF model we can analyze the maximal achievable throughput for the fixed input bottom-top, using the SDF^3 data flow analysis tool. For this particular example model, we get a value of 1/60 pieces per time unit. Moreover, the tool can perform a simulation of a selected sequence of scenarios to obtain a Gantt chart that visualizes the behavior and the resulting execution trace can be studied to obtain parameters such as latency, throughput and to discover potential bottlenecks. Figure 28 depicts the Gantt chart of the system for a scenario sequence of *8* pieces *{a,b,a,b,a,b,a,b}*.The execution of the actors in the same scenario and also same iteration are shown with the same color. As shown in the figure, for example, the latency of assembling the second object equals 116 time units. Furthermore, it is also possible to obtain the bottleneck of the system.

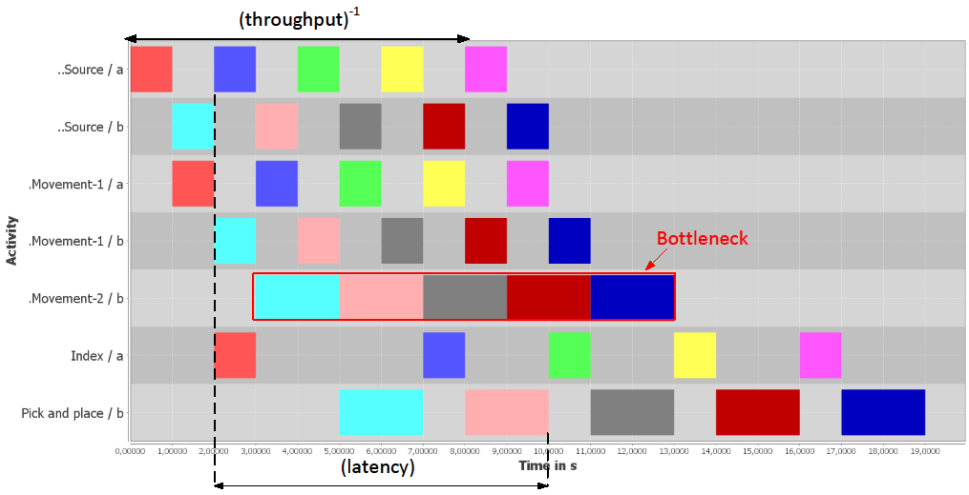


Figure 28: Gantt chart of the system for a scenario sequence of 8 pieces {a,b,a,b,a,b,a,b}

While the above models the various stages of the assembly line and how they impact the system-level performance, the overall analysis can be done in a closed loop using image-based information extraction as proposed in the recent work [16]. Using the image processing, it is possible to extract the modelling parameters and adapt the actuation/sensing (e.g., motor speed) towards system level optimization for throughput and/or latency.

9 Conclusions

In this work we described the integrated system software stack of ALMARVI: The concept of the OpenCL-Based System Software Stack; details of the *pocl* driver for AlmaIF devices which were provided by ALMARVI "hardware" partners and finally presented step by step instructions on how to program in OpenCL the ALMARVI Demo platform which can include one or more AlmaIF devices in the FPGA configuration.

Next we presented the software integration of the Dyplo eco-system and programming model from Topic Embedded Products, used by Philips. Work is on-going together with TUE to model, simulate and analyse custom build accelerators and the Dyplo network, the first results are presented in Deliverable 4.2.

Furthermore, in this work we proposed a concept for sequencing FSM-SADF scenarios that can be analysed and executed of TDMA-scheduled platforms. The concept for programming and analysing FSM-SADF on multi-processor platforms that we presented allows to capture input-dependent application behaviour. Each scenario describes one such behaviour and can have a different topology and different rates, WCETs and persistent tokens to do so.

We also presented an implementation with three novel aspects. Firstly we glue all scenarios together by implementing the *switch* and *select* actors as (de-)multiplexers by disconnecting and reconnecting FIFO channels. The timing behaviour of these actors is modelled in the scenario graphs. Secondly we propose a rolling static-order scheduler that automatically extends the schedule each time a scenario is detected. Thirdly we map persistent tokens that are shared in multiple scenarios to the same physical FIFO. The platform-aware analysis model annotates the scenarios with the exact timing behaviour of the implementation. We extended an existing HSDF model of the platform to suit FSM-SADF analysis. A more sophisticated method for encoding SO schedules in dataflow graphs was integrated in the toolflow to account for multi-rate actors and their DMAs. We mapped the SUSAN edge detection algorithm to a platform with two processors and found that the real throughput is slightly higher than the constraint given to the analysis tool. This proves that the model is both conservative and precise. The implementation cost in terms of timing and memory footprint is marginal.

Finally, we presented a use case for Scenario Aware Data Flow graphs on an image-guided assembly line prototype representing similar challenges as encountered in many streaming data applications for example in healthcare video processing systems.

10 Works Cited

[1] V. Korhonen, P. Jääskeläinen, M. Koskela, T. Viitanen and J. Takala, “Rapid customization of image processors using Halide,” 2015.

[2] T. E. Products, “Topic Embedded Products GitHub,” Topic Embedded Products, [Online]. Available: https://github.com/topic-embedded-products. [Accessed 10 2016].

[3] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” in *Proc. of the IEEE*, 1987.

[4] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M Voeten, S. V. Gheorghita, S. Stuijk, “A scenario-aware data flow model for combined long-run average and worst-case performance analysis,” in *MEMOCODE*, 2006.

[5] H. Kopetz, Real-time Systems, Springer, 2011.

[6] S. Stuijk, M. Geilen and T. Basten, “SDF For Free,” in *Proc. of 6th International Conference on Application of Concurrency to System Design*, 2006.

[7] O. Moreira, H. Corporaal, Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor, Springer, 2014.

[8] S. Stuijk, M. Geilen, B. Theelen, T. Basten, “Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications,” in *SAMOS*, 2011.

[9] R. van Kampenhout, S. Stuijk, K. Goossens, “A scenario-aware dataflow programming model,” in *2015*, DSD.

[10] K. Goossens, A. Azevedo, K. Chandrasekar, M. Dev Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, S. Sinha, “Virtual Execution Platforms for Mixed-time-criticality Systems: The CompSOC Architecture and Design Flow,” *SIGBED Rev.,* 203.

[11] A. Nelson, K. Goossens, B. Akesson, “Dataflow formalisation of real-time streaming applications on a composable and predictable multi-processor soc,” *Journal of Systems Architecture,* 2015.

[12] S. Stuijk, T. Basten, M. C. W. Geilen, H. Corporaal, “Multiprocessor Resource Allocation for Throughput-constrained Synchronous Dataflow Graphs,” in *DAC*, 2007.

[13] S. M. Brady, J. M. Smith, “SUSAN — A new approach to low level image processing,” *International Journal of Computer Vision,* 2007.

[14] Adyanthaya, Shreya, et al., “xCPS: a tool to eXplore cyber physical systems,” in *Proceedings of the WESE'15: Workshop on Embedded and Cyber-Physical Systems Education, ACM*, 2015.

[15] B. D. Theelen, M. Geilen, T. Basten, J. Voeten, S. V. Gheorghita, and S. Stuijk, “A scenario-aware data flow model for combined lond-run average and worst-case performance analysis,” in *Formal Methods and Models for Codesign (MEMOCODE)*, 2006.

[16] S. S. D. G. T. B. Róbinson Medina Sánchez, “Reconfigurable pipelined sensing for image-based control,” in *SIES*, 2016.