TTADF: Power Efficient Dataflow-Based Multicore Co-Design Flow

Ilkka Hautala, Jani Boutellier, *Senior Member, IEEE,* and Olli Silvén, *Member, IEEE*

Abstract—The era of mobile communications and the Internet of Things (IoT) has introduced numerous challenges for mobile processing platforms that are responsible for increasingly complex signal processing tasks from different application domains. In recent years, the power efficiency of computing has been improved by adding more parallelism and workload-specific computing resources to such platforms. However, programming of parallel systems can be time-consuming and challenging if only low-level programming methods are used. This work presents a dataflow-based co-design framework TTADF that reduces the design effort of both software and hardware design for mobile processing platforms. The paper presents three application examples from the fields of video coding, machine vision, and wireless communications. The application examples are mapped and profiled both on a pipelined and a shared-memory multicore platform that is generated by TTADF. The results of the TTADF co-design-based solutions are compared against previous manually created designs and a recent dataflow-based design flow, showing that TTADF provides very high energy efficiency together with a high level of automation in software and hardware design.

Index Terms—Dataflow, application specific processor, design flow, low power.

1 Introduction

Rapidly evolving technology has been shrinking the time-to-market window of mobile software and computing platforms. To this extent, the design time should be as short as possible and the lifetime of a design should be maximized [1]. To meet these requirements, automated design flows are needed, and the resulting system should be programmable by high-level programming languages, enabling fixes and inclusion of additional functionality after deployment. Consequently, software has a significant role in contemporary mobile computing platforms.

Advances in semiconductor manufacturing processes have continuously increased the efficiency of programmable processing platforms. Therefore, software programmable off-the-shelf digital signal processors (DSPs) and general purpose processors (GPP) have increasingly replaced application specific integrated circuits (ASIC) in mobile computing platforms. However, application requirements, including power consumption, throughput, and latency, have led to the trend where multiprocessor System-on-Chips (MPSoC) have become mainstream [2]. A heterogeneous MPSoC can integrate for example a multicore GPP, a graphics processing unit (GPU), DSPs, a field programmable gate array (FPGA) and multiple ASICs or application-specific instruction processors (ASIP) for wireless communications, computer vision, and security.

ASIPs have been used for applications where low power consumption and high performance are essential, but also programmability is needed [3]. The energy efficiency of ASIPs has usually been achieved by tailoring the instruction set for a single application domain, which enables simplifying the architecture compared to GPPs. Compared to ASICs, ASIPs have a time-to-market advantage thanks to programmability and highly automated ASIP design tools [4]. ASIP design tools commonly include a *retargetable compiler*, which automatically adapts to the ASIP architecture, and enables machine code generation from a high-level language without compiler redesign.

Programming of MPSoCs is challenging due to the lack of high-productivity programming environments that can compile application descriptions into useful implementations for parallel and heterogeneous target platforms [5]. Also, existing legacy code has hindered the adoption of more intuitive and suitable methods for software development for these platforms. Utilizing the potential of MPSoCs is a complex task requiring correct specification, implementation, decomposition, and mapping of applications [5].

Dataflow [6], a well-known and widely used programming paradigm for expressing the functionality of signal processing and streaming applications, has been proposed as a next-generation programming solution for heterogeneous MPSoCs [7] [8] [9] [10] [11]. Dataflow programming is an intuitive approach for describing the parallelism of application software, while at the same time increasing its modularity, flexibility and re-usability.

In this paper, the *dataflow-based design framework for transport triggered architectures* (TTADF[1]) is presented. The main contributions of this work are as follows:

- TTADF integrates ASIP development and C-language based dataflow programming for the design of power-efficient and high-performance multicore ASIPs and their software.

- *I. Hautala and Olli Silvén are with the Center for Machine Vision and Signal Analysis Research Group, University of Oulu, Oulu, Finland E-mail: {ilkka.hautala, olli.silven}@oulu.fi*
- *Jani Boutellier is with the School of Technology and Innovations, University of Vaasa, Finland, and with the Faculty of Information Technology and Communications, Tampere University, Finland. E-mail: jani.boutellier@univaasa.fi*

Manuscript received –, –, 2018; revised –, –, 2019.

1. The source code of TTADF is available for download at http://github.com/ithauta/ttadf

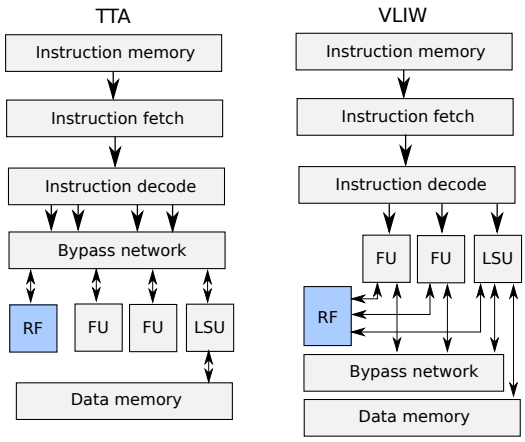

Fig. 1. Comparison between TTA and VLIW architectures

- TTADF has tools for both rapid and accurate simulation of multicore ASIPs and their software.
- The outcome of the TTADF design flow is a synthesizable register transfer level (RTL) description of the designed platform and executable program binaries. Hardware synthesis for an FPGA or using a standard cell library can be done using existing Electronic Design Automation (EDA) tools.
- Comprehensive experiments confirm high energy efficiency. Example applications from three different domains, including new dataflow representations of High Efficiency Video Coding (HEVC) in-loop filters and stereo depth estimation.

This paper is organized as follows: Section 2 presents the theory of the dataflow paradigm and the Transport Triggered Architecture, whereas Section 3 reviews related work. In Section 4 implementation details of the proposed TTADF framework are described. In Section 5 experimental results of the designs produced using the proposed framework are shown, and later compared to related work in Section 6. Finally, a brief discussion and the conclusion of the achieved results can be found in Sections 7 and 8.

2 Background

First we present a brief introduction to *transport triggered architectures* and after that basic concepts of *dataflow* programming.

2.1 Transport Triggered Architecture

Transport Triggered Architecture (TTA) processors resemble VLIW processors concerning the fetching, decoding and execution of multiple instructions each clock cycle, and by providing *instruction level parallelism* [12]. Fig. 1 shows the fundamental differences between datapaths of TTA and VLIW architectures. In the case of TTA, each FU and RF is connected to the bypass network, whereas in VLIW each FU is directly connected to the RF. The fully exposed datapath of TTAs allows direct control of data transfer for the programmer (or compiler), in contrast to VLIWs that are programmed by operations.

In TTA, operations take place as side effects of data transfers, controlled by *move* instructions that are the only

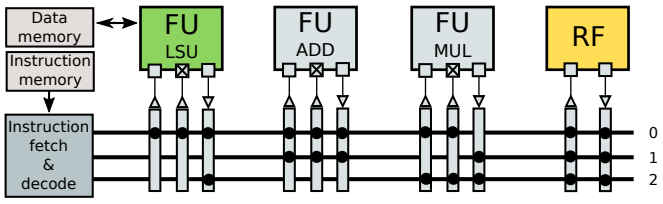

Fig. 2. A simple TTA processor

instruction of TTA processors. Using *move*, data is transferred between function units (FU) and register files (RF) via the *bypass network* that can consist of multiple *transport buses*. FUs are logic blocks that implement different operations, such as additions or multiplications. Depending on the set of operations included in FUs, the FU has one or more input ports and registered output ports. In every single FU, one input port is a *triggering port* and a data move to it triggers an operation of the FU in question. If an operation has multiple operands, it is assumed that all the other operands are transferred to FU input ports before or at the same time as the operand, which is going to be written to the trigger port. After triggering an operation, the operation result can be moved from an FU output port to the input ports of one or more FUs/RFs, which makes TTAs *exposed datapath* processors, enabling many compiler optimizations.

Fig. 2 presents a simple TTA processor, which has three transport buses (black horizontal lines), a load store unit (LSU), an adder, a multiplier, and one RF. Small squares are FU ports, and a cross inside the square indicates the triggering port. The instruction fetch and decode unit is responsible for loading one instruction per bus from the instruction memory and executing them. Three transport buses enable executing three instructions in parallel each clock cycle.

For example, in one clock cycle, we can simultaneously move a result from the LSU to the input port of the adder unit using bus 0, move the `mul` result to the adder's triggering port using bus 2 and move the result of the previous `add` operation to the RF using bus 3. The previous example can be presented by one TTA assembly instruction word that executes in a single clock cycle:

```
LSU.out1 -> ADD.in2,
MUL.out1 -> ADD.in1t.add,
ADD.out1 -> RF.1;
```

FUs and RFs are connected to transport buses via sockets (rectangular vertical blocks), and arrows above sockets indicate the input/output direction of a socket. Black dots on sockets indicate where FUs/RFs are connected to the transport buses.

Similar to DSPs, also TTAs use the Harvard architecture that has separate program and data memories. More than that, TTAs can have *multiple* data memories, each of them appearing as a different address space to the programmer.

Implementing full context switch support to TTAs is regarded as unfeasible due to the high number of registers within (pipelined) TTA FUs. Hence, interrupts and pre-emptive multitasking are commonly unsupported, and TTAs are used as slave co-processors of a master GPP

that runs control-intensive software, such as the operating system [13].

TTA processors can be designed using the open source TTA-based Co-design Environment (TCE) [14]. By using the TCE compiler *tcecc*, programs written in high-level programming languages (C/C++, OpenCL) can be compiled to TTA machine code. Tcecc is a retargetable compiler, adapting itself to the designed architecture and automatically taking advantage of added processing resources, which allows fast experimentation with different processor designs and design space exploration. TCE also offers a cycle-accurate simulator for program execution analysis on a given TTA processor. Moreover, a synthesizable RTL description of the given processor design can be created using the TCE processor generator tool. TCE currently supports the design and simulation of single-core and GPU-style data parallel multicores. *TTADF builds on TCE and adds a design and simulation framework for task-parallel streaming applications.*

2.2 Dataflow paradigm

The dataflow programming paradigm is used in this work for top-level application descriptions. The dataflow programming paradigm started forming in the end of the 1960s inspired by highly parallel computer architectures [15]. Dataflow applications are represented as directed graphs, which consist of *actors* as vertices and unidirectional first-in-first-out (FIFO) channels as edges. The data, transferred between actors via channels, is quantized to *tokens* of arbitrary size. Dataflow description of an application ensures application modularity, re-usability and exposes its concurrency, which simplifies distributing the application execution across multiple processing elements.

Several different formal dataflow Models of Computation (MoC) have been proposed, including synchronous dataflow [16] (SDF) and dataflow process networks [17] (DPN). Dataflow MoCs can be classified to static or dynamic depending on whether data can affect the behavior of actors. In a static MoC, the token production and the token consumption rate of actors are statically defined whereas in dynamic MoCs production and consumption rates can be data-dependent. Static MoCs enable more compile-time optimizations than dynamic MoCs and can therefore lead to more efficient machine code. However, the limited expressiveness of static MoCs reduces the set of applications that can be described compared to dynamic MoCs. Thus, different dataflow MoCs offer a tradeoff between efficiency and expressiveness.

The dataflow framework used in this work is designed to support the DPN MoC [17] that is very dynamic. However, it also allows using more restricted MoCs such as SDF.

3 Related Work

In [18], Park et al. review different MPSoC design methods and categorize them to four approaches: compiler-based, language-extension (OpenMP, OpenCL), model-based and platform-based approaches. In this paper we focus on the model-based approach, where the designer utilizes a MoC for implementing applications. There are several works, which are based on the dynamic dataflow programming paradigm, and some of them also target TTA architectures. The most relevant works considering this paper are briefly presented below.

Ptolemy [19], a simulation and prototyping framework for heterogeneous systems, was the pioneering software development framework for dataflow-based design. In addition to dataflow, Ptolemy also supports a variety of non-dataflow MoCs.

Orcc [20] is a recent open source dataflow development environment that is based on the DPN MoC. Orcc applications are written using the RVC-CAL language, to be translated to software and hardware code. By using Orcc, various video coding applications such as High Efficiency Video Coding have been implemented for various target platforms [21] [22] [23].

Distributed Application Layer (DAL) [24] is a scenario-based design flow, which can map Kahn Process Network (KPN) applications onto heterogeneous many-core systems. Differing from many other frameworks, DAL offers support for OpenCL capable platforms, which enables the use of GPUs [25].

PRUNE [11] is an open source framework for design and execution of dynamic and decidable dataflow applications on heterogeneous platforms. The PRUNE framework defines its own dynamic MoC, which is developed specifically for the requirements of heterogeneous platforms. PRUNE allows the execution of *dynamic* actors on OpenCL platforms, which is not possible in DAL.

Dardaillon [10] et al. proposed an LLVM based compilation flow that can compile parameterized dataflow graphs to a heterogeneous MPSoC platform. They use actor based C++ for expressing dataflow graphs. Their framework is dedicated to software-defined radio applications that require parallel processing and fast dynamic reconfiguration.

In [9] Bezati et al. introduce a tool that compiles RVC-CAL dataflow programs into RTL descriptions, targeting MPSoCs. They translate the RVC-CAL description to C and feed it to the Xilinx Vivado High-Level Synthesis compiler with automatically generated constraints and directives to get RTL descriptions.

PREESM [26] is a dataflow-based framework for multicore DSP programming. PREESM exploits the Parameterized and Interfaced Synchronous Dataflow (PiSDF) MoC. The behavior of PREESM actors is described using C language, while XML is used for describing the target architecture, the algorithm graph, and the scenario. In [26], PREESM is used to execute a stereo image matching application [27].

In [28] and [8], the authors present automatic synthesis of TTA processor networks from dynamic dataflow programs using the Orcc framework. They propose a design flow where the RVC-CAL dataflow language is used to describe an actor network. Orcc [20] compiles the actor network into LLVM (Low Level Virtual Machine) [29] assembly code in the case of [8], or into C code in the case of [28]. In both works, TCE is then used to generate RTL descriptions of processor cores and the machine code for each core. In both works, a dedicated TTA processor is generated for each actor, and inter-processor communication is implemented via hardware FIFOs between TTA cores. In [8], the authors define three TTA processor configurations: *standard*, *custom* and *huge* with different numbers of function units and

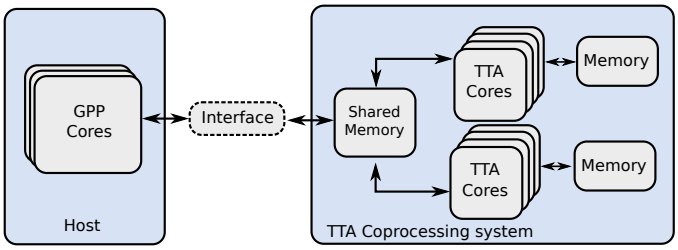

Fig. 3. An example of a TTA co-processing system

transport buses. Finally, the work of [28] is generalized in [30] by allowing an arbitrary number of actors to be mapped to a TTA core.

Similarly, Yviquel et al. [23] refine the basic ideas presented in [8] by introducing a hybrid memory architecture designed for dataflow programs. Instead of using hardware FIFOs for inter-processor communication the authors exploit shared memories. As in [28], RVC-CAL is used as the input language, which is first transformed into the LLVM intermediate representation and then into binary code, which is suitable for the target TTA processor. In [23], Yviquel et al. demonstrate their work by implementing HEVC and MPEG-4 video decoders on top of *custom*, *fast* and *huge* TTA processors. Currently, their work is a part of Orcc [20], and it is named the Orcc TTA Backend.

4 TTA Dataflow Framework

This section describes the proposed TTADF framework. First, a brief overview of the framework is given, followed by a high-level description of its usage. Finally, the central framework components are explained in detail.

In the proposed framework, TTAs are used as co-processors, which are communicating with the host processor using shared memory, as shown in Fig. 3. TTA cores and the memories connected to them form the *TTA co-processing system*. The *host architecture* encompasses all other processing cores, which have a memory address space in common with the TTA co-processing system.

By using TTADF, the designer can specify both the *host architecture* and the TTA co-processing system and synthesize a unified dataflow program of the whole host-co-processing ensemble. A dataflow program, which is described using the TTADF API can be executed on numerous different target platforms without modifications to the software code, but by merely switching the architecture description. TTADF also allows automatic RTL generation of the *TTA co-processing system*, which can be imported to EDA tools for hardware synthesis.

4.1 Design flow

The proposed design flow is presented in Fig. 4, and starts by the application design step (*Actor Descriptions* and *Actor Network* items). As explained in Section 2.2, the dataflow application consists of *actors* and *FIFO* communication channels between them. In the proposed framework the *actor network* is the top-level description of the dataflow application, and it defines the actors and the FIFO connections, creating

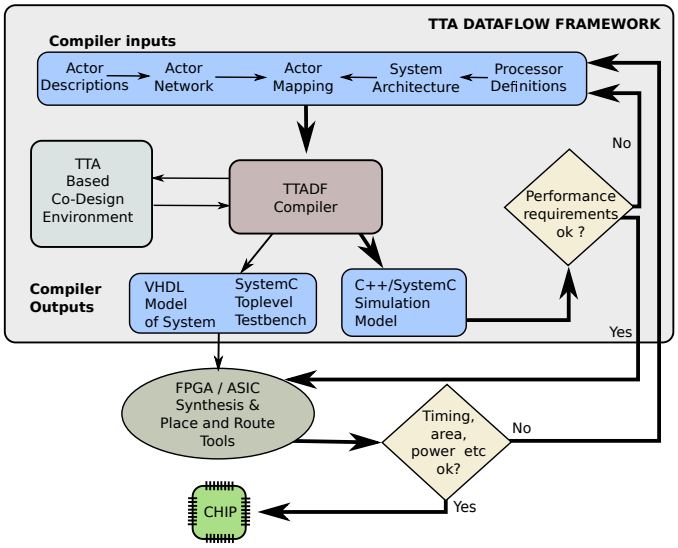

Fig. 4. The proposed TTADF design flow

the application datapath. The detailed behavior of each actor is defined in the *actor description*.

In the proposed framework, *the system architecture* file defines all computational resources available for the execution of software. Similarly as the *actor network* defines connections between different actors, the *system architecture* file defines processing resources and their interconnections. The *system architecture* file can refer to the *processor definition* that is the detailed specification of a TTA processor architecture.

These five description files are inputs for the *TTADF Compiler*, which analyses the inputs and generates a consistent description of the software and the hardware. By using that description, the compiler creates necessary inputs for the TCE tools to produce TTA machine code and processor RTL descriptions. The TTADF compiler produces C++ and SystemC simulation code, which can be compiled using GCC. The SystemC simulation model is cycle-accurate, whereas the C++ simulation model uses a simplified memory model for faster simulation. TTADF generates all needed VHDL models for the TTA co-processing system synthesis and a SystemC testbench to ease simulation and verification.

4.2 Actor network

The dataflow application is specified using the *actor network* file, which lists all actors and FIFOs used in the dataflow software. Fig. 5 presents the actor network of a simple dataflow program that generates numbers (`Source`), multiplies them by a constant factor (`CMultiply`) and then prints the values (`Sink`). The corresponding actor network presentation of the constant multiply application is presented in Listing 1 as pseudocode.

Each actor is defined by its name and has a behavior description written in C. Actor descriptions, including the number of input and output ports can be parameterized. For example, in Listing 1 the constant FACTOR is defined for the actor `CMultiply`.

The FIFO descriptions of the actor network define the token size (in bytes) and the token capacity of each FIFO.

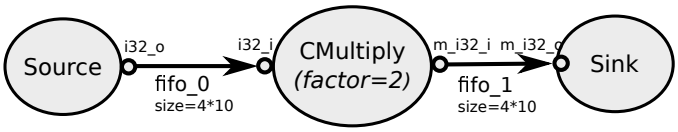

Fig. 5. The constant multiply dataflow application example

```
NETWORK multiply

 DEFINE parallelism=1

 ACTOR source_0
     MODELNAME source
     MODELSOURCE actors/source.c
     PARAMETER PARALLELISM=parallelism
     GENERATE i TO parallelism
         OUTPUTPORT port_$i
     END GENERATE
     STOPNETWORK 1
 END ACTOR

 GENERATE i TO parallelism
      ACTOR cmultiply_$i
          MODELNAME cmultiply
          MODELSOURCE actors/cmultiply.c
          PARAMETER FACTOR=2
          INPUTPORT in_port
          OUTPUTPORT out_port
      END ACTOR
 END GENERATE

 ACTOR sink_0
     ...
 END ACTOR

 GENERATE i TO parallelism
      FIFO fifo1_$i
          TOKENSIZE 4
          CAPACITY 10
          SOURCE source_0 port_$i
          TARGET cmultiply_$i in_port
      END FIFO

      FIFO fifo2
          ...
      END FIFO
 END GENERATE

END NETWORK
```

Listing 1. An example of a parameterized actor network description of the constant multiply dataflow program

For each FIFO, a source and a target port of an actor is determined by using an actor id and a port name.

Parameterization can be used for fast adaptation of the actor network to the system architecture. Considering Fig. 5 as an example, the degree of parallelism can be adjusted by adding more input and output ports to the `Source` and `Sink` actors, and by replicating the `CMultiply` actor. To implement that, the TTADF compiler supports a compiler pragma for static code generation, presented in Listing 1.

4.3 Actor description

TTADF provides a high-level template that has to be used for describing actors, as well as an API (TTADF API) for implementing framework specific behavior such as FIFO I/O. The detailed behavior of actors is described using the C language, which also allows the use of legacy code and C compiler tools. In the case of TTA processors, using C provides the designer the possibility for efficient use of *special function units* (SFUs) of TTA processors, by using TCE C macro calls for custom operations.

As can be seen from Listing 2 of the `Source` actor, the actor description file contains four different structural elements that are defined by the actor description name and source code:

- `ACTORSTATE` is a structural element that contains information about the actor state. The designer defines all actor state variables, which need to be maintained between actor firings into this structure.
- `INIT` is a function, which is executed once before the actor is fired for the first time by the TTADF runtime. The function is useful for opening input and output file streams.
- `FIRE` is an element that defines a function, which is executed whenever the actor is fired. A pointer to the `ACTORSTATE` structure is passed to the `FIRE` function so that the actor can preserve its state between firings.
- `FINISH` is an element that defines a function, which is called once when the execution of the actor network is terminated.

A similar `INIT` / `FIRE` / `FINISH` approach has also been used in other dataflow flavored frameworks (e.g. [24]).

Inside these three functions it is possible to use TTADF API function calls or macros. The TTADF API offers basic FIFO I/O operations including reading, writing, peeking and querying the number of tokens in a FIFO. Pragma commands for static code generation are supported in the actor description source files, for enabling parameterized actor specification.

4.4 System architecture model

This work proposes a system architecture model (SAM) to specify the ensemble of the host system and the TTA cores. The SAM can be divided into the *host architecture* and the *TTA Co-processing architecture* parts, as shown in Fig. 3 (below, *host* in italics refers the *host architecture*). In TTADF, the host architecture is required to have a) memory-mapped access to the co-processing architecture, and b) a global shared memory architecture, where all cores can access the same address space.

In practice, the *host* can be connected to the shared memory through a memory interface such as AMBA (Advanced Microcontroller Bus Architecture), PCI-E (Peripheral Component Interconnect Express), Ethernet, etc.

The TTA Co-processing system comprises all TTA cores, memory components and other components that are directly connected to them. The memory architecture of the TTA Co-processing system resembles the *hybrid memory architecture* presented in [23]. In the hybrid memory architecture, each TTA core has its own private data memory and instruction memory. Inter-core communication is performed through shared memories that form a communication network between cores. This kind of a memory architecture is natural for dataflow programs since the local data of actors can be

```
// Library includes here
#include <stdio.h>

ACTORSTATE source{
  //Actor state variable declarations
  int number;
  #PRAGMA GENERATE i to PARALLELISM)
  TTADF_PORT_VAR("port_$i",data_$i,"int");
  #PRAGMA END_GENERATE

}

INIT source(source_STATE *state){
  //All initialization setup here
  state->number = 0;
}

FIRE source(source_STATE *state){
  #PRAGMA GENERATE(i=0,i++,i< PARALLELISM)
    state->number ++;
    TTADF_PORT_WRITE_START("port_$i",state->data_$i)
    ;
      *state->data_$i$ = state->number;
    TTADF_PORT_WRITE_END("port_$i");
  #PRAGMA END_GENERATE

  // After 12 generated numbers
  // stop execution of actor network
  if(state->number > 12){
    TTADF_STOP();
  }
}

FINISH source(source_STATE *state){
  //Cleanup things here
  return 0;
}
```

Listing 2. The behaviour model of the Source actor

stored in the private memory and incoming and outgoing tokens can reside in shared memory. The memory organization also divides memory components into subcomponents, which reduces memory pressure, provides simultaneous R/W access and reduces power consumption when compared to a global shared memory architecture used in many GPPs [23].

An example of a generated system can be seen as a block diagram in Fig. 6. The designer can set a core of the *host* architecture to be of type X86-64, ARM or ARM64, whereas for the TTA co-processing system all cores will be of the type TTA. Additionally, the clock frequency and all memory connections are defined for each core. A core can be connected to a memory component directly or via a memory arbiter. The capacity of each memory component is defined in bytes. Memory arbiters can be used to connect multiple cores to one single port memory.

For each TTA core, a TTA Architecture Definition File (ADF) needs to be provided. The ADF describes all resources of the TTA core instance, including FUs, RFs, etc. *Prode*, a TCE tool, can be used to design TTA cores and produce their ADFs for the coprocessing system. For enabling RTL generation, also an Implementation Definition File (IDF) needs to be provided for each TTA. The IDF file defines which RTL description is used for each processor component.

Memory and arbiter instances are connected to dedicated LSUs of TTA cores. Therefore, the ADF of each TTA

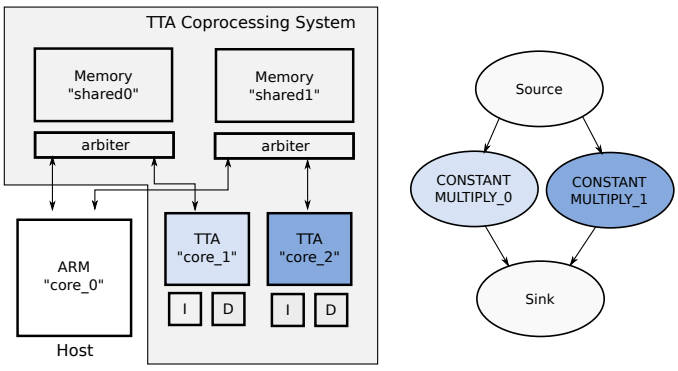

Fig. 6. System architecture model and actor mapping

instance has to include an LSU for each memory or arbiter instance that is connected to the TTA core. The TTADF framework automatically sets the address spaces for these LSUs.

The designer can also define a SAM that does not include any TTA cores at all. An example of this is presented later on in Section 5, where the ARM-based embedded ODROID XU3-platform is used to execute TTADF applications.

4.5 FIFO communication channels

In this work, FIFO channels are implemented using a lock-free circular buffer structure that is placed into addressable memory. Each FIFO must be connected to exactly one input port and exactly one output port of an actor. Each FIFO has user-defined *token size* and *capacity*, which define the maximum number of tokens the FIFO can hold. Token size can be freely chosen with the accuracy of one byte.

For synchronizing FIFO-read and FIFO-write transactions, the writing actor has write access to the FIFO-write pointer, and the reading actor has write access to the FIFO-read pointer. Both, the reading actor and the writing actor have read access to the pointers, which enables querying (peeking) of the FIFO state.

Each FIFO operation is started by requesting a pointer to the token buffer for reading or writing. If the requested FIFO operation is feasible (FIFO has room / FIFO has tokens), a pointer to the token is returned. The actor can read data from the port, process it and simultaneously write the processed data directly to another port without additional data copying. After all needed data has been read or written, the FIFO access is ended.

Choosing a suitable FIFO buffer size is essential for reaching high performance. The buffer capacity should be two or more tokens since single buffering prevents simultaneous execution of the producer and consumer actor. With the set of applications used to test TTADF, no significant performance increase was observed when increasing FIFO capacity beyond three.

4.5.1 Blocking and non-blocking communication

The proposed framework supports blocking FIFO communication channels where actor execution is halted until the required number of tokens for reading, or suitable space for writing to the FIFO is available. The implementation of blocking communication is not straightforward because

when multiple actors have to be executed on the same core, there has to be a mechanism for transferring execution from a blocked actor to another actor to prevent system deadlock.

Since TCE does not offer support for *preemptive multitasking*, the proposed framework addresses the problem by using *protothreads* (PT) [31], which is a non-preemptive multitasking concept that does not rely on context switches. The idea of using protothreads in graph-based processing has previously been presented in [24].

The proposed framework implements actor firing functions so that blocking FIFO operations are labeled, and can be seen as *entry points* for the function. The actor state holds the current entry point, and when the actor is fired, execution jumps to the current entry point. If a FIFO operation blocks, the FIFO operation in question is stored as the entry point.

In some cases blocking communication can be used to reduce overhead caused by checking actor firing rules. If firing rules are independent, it is not necessary to check all rules, but start directly with the blocking one [32]. As TTA processors generally do not have branch predictors, this is a considerable advantage.

The framework also supports non-blocking communication where actor firing also continues in the case when a FIFO cannot provide enough tokens or free space. Non-blocking communication is especially needed for supporting the DPN MoC.

4.5.2 Hardware accelerated FIFO operations

Low-level FIFO access operations are suitable to be accelerated by custom instructions. The TTADF default FIFO access SFU includes two custom operations:

- *get_population* for calculating the token population of a FIFO.
- *update_fifo_pointer* – an operation for finishing FIFO access.

TTADF detects if a TTA core is equipped with the FIFO access SFU that includes hardware accelerated FIFO operations, and automatically makes use of the custom operations. The speedup advantage of the FIFO operation SFU is case dependent, however use of the FIFO SFU can significantly improve code density in cases where the compiler would inline a high number of software-based FIFO access functions.

4.6 Mapping and scheduling

In the TTADF framework, the actor mapping file defines for each actor, which core takes responsibility for the execution of that actor. The actor mapping influences the actions of the TTADF compiler, and therefore actor mapping is static, unmodifiable at runtime. Based on the actor to core mapping, the TTADF compiler automatically maps FIFOs from the actor network to the memory components. If connected, actors are assigned to different cores, and a FIFO is mapped to the shared memory component. If two (or more) actors are mapped to the same TTA core, the TTADF compiler maps the FIFO to the private data memory of that core.

For each core, a dedicated actor firing scheduler is created. The actor firing scheduler handles all actors that are assigned to the core in question. Currently, actor firings are scheduled in a round-robin fashion. In round-robin scheduling, each actor attempts to fire, and regardless of success (token availability / free FIFO space), the scheduler proceeds to fire the next actor until an actor triggers a stop condition. Michalska et al. [33] show that despite its simplicity, round-robin scheduling is a very efficient scheduling methodology for TTA processors, whereas more complex scheduling methodologies can provide only minimal improvements or even degrade performance.

4.7 The TTADF Compiler

The TTADF Compiler is the main component of the framework, and it is written in the Python programming language. The TTADF Compiler deserializes the *actor network*, *actor mapping* and *system model* files, and constructs a unified object representation which is used to generate C software code for the cores and hardware synthesizable VHDL code of the TTA co-processing system including the SystemC testbench.

One of the main tasks of the compiler is to translate the behavior model of the actors into plain C code so that it can be compiled to binary code for the target processor: the compiler processes `ACTORSTATE`, `INIT`, `FIRE`, and `FINISH` elements and TTADF API calls, translating them to the unified object representation. For each core in the system, the TTADF compiler generates runtime code, which has the responsibility for initialization, scheduling and cleaning up of actors that are mapped to the core in question.

4.7.1 Simulation and testing

The TTADF Compiler can generate three different simulation models with different accuracy and speed tradeoffs:

- C/C++ simulation - TTA cores run on top of a cycle-accurate simulator by sharing a memory with GPP core threads, which are running on the host computer. This model assumes that memory instances are ideal in the sense that the core can always access them in constant time, and it is the fastest simulation model of the framework. This model is targeted towards performance evaluation in cases where only a few TTA cores are connected to the same shared memory.
- SystemC simulation - System-level simulation of the design. TTA cores are instantiated as a SystemC model and *host* GPP cores are running as threads on the host machine of the framework. Memory and arbiter instances have generic cycle-accurate SystemC simulation models. This simulation model is primarily intended for cases where the designer needs to accurately know how simultaneous memory accesses influence the TTA co-processor performance.
- HDL simulation - a SystemC testbench where the TTA co-processing system is instantiated in RTL level. The simulation needs mixed language support from the HDL tools. The testbench can be used for RTL, gate-level and post-layout verification, and power estimation of the TTA co-processing system.

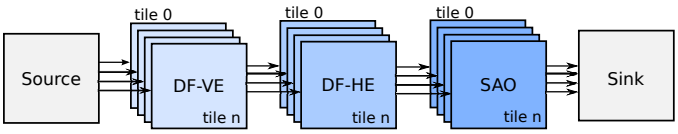

Fig. 7. HEVC Inloop filtering dataflow application

4.7.2 Actor mapping in testing and analysis

The TTADF feature that allows assigning actors to the *host* cores, and simulating these actors rapidly on the host system can be exploited in many ways in testing. At the beginning of application development, it is useful to map all actors to the host machine, since it allows application debugging using conventional C/C++ debuggers. When the application prototype works on the host machine, the designer can proceed by mapping actors to TTA cores. Finally, tailoring the TTA processor for a specific actor is an iterative process, where the designer modifies the processor and profiles actor performance repeatedly. The possibility of mapping actors to the *host* accelerates testing, as these actors do not need to be run on slow cycle-accurate simulators. Finally, the ease of actor-to-host mapping also speeds up design space exploration: discovering the best execution core type for each actor can be performed merely by changing the mapping, without code modifications.

5 Experiments

In this section, three test case workloads from different application domains are presented. After that, the TTA co-processing architectures which are used to execute the workloads are introduced. Finally, the experimental test results are presented.

5.1 HEVC inloop filtering

The High Efficiency Video coding standard [34] introduces two inloop filters for reducing coding artifacts caused by image transforms and quantizations. These filters are the deblocking filter (DF) [35] and the sample adaptive offset filter (SAO) [36]. The dataflow description of the inloop filters is based on the authors' prior work [37], and it consists of five actors in the simplest case, where only one *tile* is used. As shown in Fig. 7. the DF filter is divided into two actors: vertical edges filtering (`DF-VE`) and horizontal edges filtering (`DF-HE`). SAO is performed in the `SAO` actor after the `DF-VE` and `DF-HE` filters. In the case where the video is coded using multiple tiles and filtering over tiles is not allowed, the filtering pipeline can be parallelized up to the number of tiles. In the experiments, one, two and four tiles are used for the video size of 1920 × 1080 pixels. The actor network processes the video in a coding tree block (CTB) basis, and the token sizes of FIFOs are selected based on the size of the CTB. In the case of a 64 × 64 CTB, the token size is about 5 kB depending on the needed coding parameters, which are explained in detail in [37]. The HEVC Inloop Filtering application uses TTA special function units, which is not the case for the other test applications.

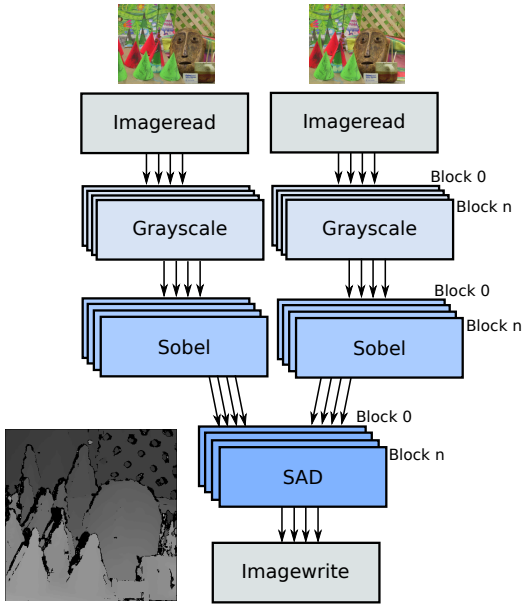

Fig. 8. SAD based depth estimation dataflow application network and a resulting depth map image

5.2 Stereo depth estimation

The dataflow implementation of Stereo Depth Estimation (SDE) is based on the open-source computer vision (OpenCV) implementation of block matching for camera calibration and 3D reconstruction. The dataflow implementation of SDE includes five different actors for the reading of image (`imageread`), image grayscaling (`grayscale`), Sobel filtering (`sobel`), Sum of Absolute Differences (`SAD`) [38] calculating, and image writing (`imagewrite`). As shown in Fig. 8, image reading, grayscaling and Sobel filtering is performed for both the left and right images before the SAD is calculated between the images, to determine the depth map. The SDE dataflow application processes the images line-by-line or multiple lines at a time, depending on available memory. Because of memory constraints, all experiments presented in the paper use line-by-line processing. In the experiments, stereo images with a size of 450 × 375 pixels are used. The search window size is set to 9, and the maximum disparity is limited to 64 pixels. The token size of the FIFO is set to be the same as the input image width, and the FIFO capacity is defined as one.

5.3 Dynamic predistortion filter

Dynamic Predistortion Filter (DPD) is a wireless communications application tailored to suppress the most harmful spurious emissions at the mobile transmitter power amplifier output [39]. The DPD dataflow application mainly consists of parallel 10-tap complex valued FIR filters, which are implemented using fixed-point arithmetic. The `configure` actor controls at runtime, which set of FIR filters is used for processing the input signal by notifying the `poly` and the `add` actors. There can be two to ten active filters at each time instant, depending on the adaptive runtime configuration. Since an external input controls the configuration, the network behavior is truly dynamic. The dataflow network for the DPD is presented in Fig. 9. The fixed point DPD dataflow

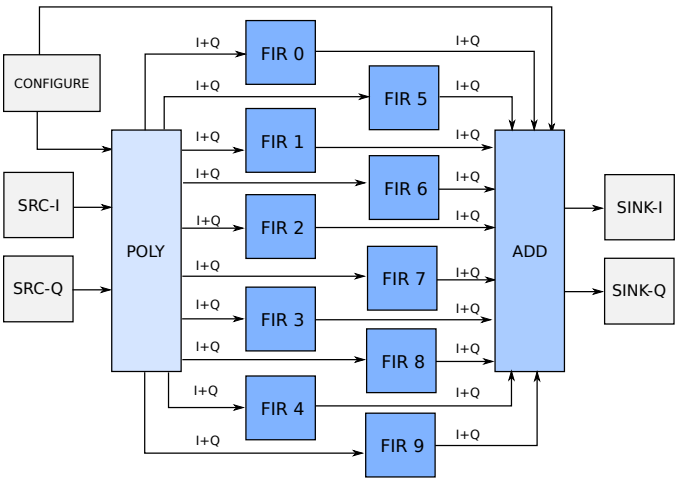

Fig. 9. Dataflow network of the dynamic predistortion filter application

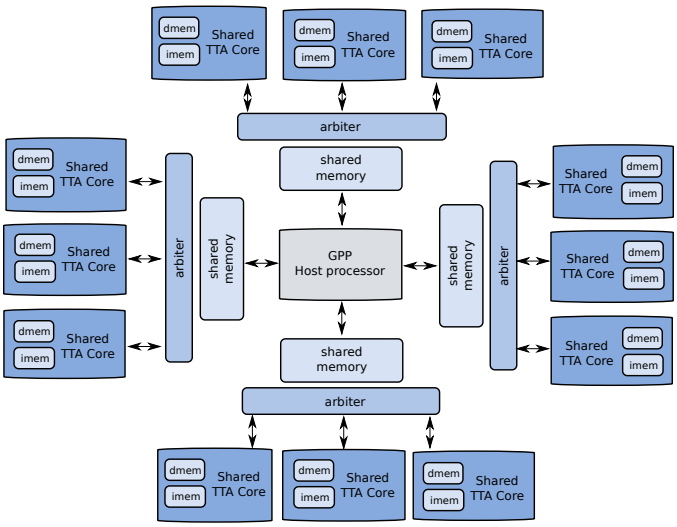

Fig. 10. The four-cluster triple-TTA core architecture (*shared12*)

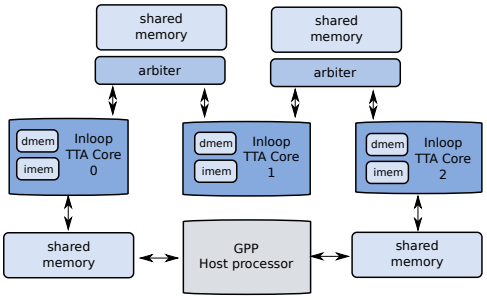

Fig. 11. The pipelined triple-core Inloop TTA architecture (*Inloop3*)

network has been designed for TTADF from a floating point version of DPD network presented in PRUNE [11]. In the experiments, the token size is set to 256 bytes, which equals to 64 32-bit integer numbers. The FIFO capacity is also set to two to enable double buffering. The DPD is a *dynamic* dataflow application, containing actors that have dynamic token rates, whereas in other test applications only fixed token rate actors are used.

5.4 TTA co-processing system architectures

For the experiments, eight different TTA co-processing system configurations are defined. The configurations can be divided into two categories based on the TTA processor core architecture which they use. The first category of co-processing systems is referred to as *Inloop*, and consists of TTA cores tailored for HEVC Inloop filtering. This core type is presented in detail in [37] [2]. In the second category, referred to as *Shared*, the co-processing system is based on one of the predefined TTA core architectures from [23], where it is called *custom* [2]. Both categories include different configurations concerning the core count with 1, 3, 6 and 12 TTA cores. From here on, when some specific configuration is referred to, the notation is the name of the category, followed by the number of cores, such as *Inloop3*, for example.

Inloop1 and *Shared1* are simple system configurations, where only a single TTA core is connected to the *host* processor using shared SRAM memory. *Shared3*, *Shared6*, *Shared12*, *Inloop6* and *Inloop12* consist of clusters of three TTA cores. Each cluster is connected to the *host* via a shared SRAM, which is also used for inter-cluster communication. Each TTA core is connected to the shared memory through a memory arbiter. In Fig. 10, a four-cluster *Shared12* TTA coprocessing system is presented.

Inloop3, presented in Fig. 11, is a special case, and it is initially tailored for HEVC Inloop filtering [40]. The TTA cores are connected using shared SRAM memories so that a three-stage pipeline is formed. Since the *host* is connected to the SRAMs at both ends of the pipeline, the architecture follows a ring topology, where data can be moved into both directions. In [40], the authors show that the Inloop TTA co-processing system can achieve a 1.2 GHz clock frequency (1.0 V operating voltage) when placed and routed using 28 nm standard cell technology. Based on that result, in the experiments, a clock frequency of 1.2 GHz is assumed in all *Inloop* system configurations.

In all system configurations, each core has a 32 kB private data memory and a 64 kB instruction memory. Shared memory blocks are of the size 64 kB except for *Inloop3* which has four 16 kB shared memory blocks. All test case dataflow applications are configured to fit these tight memory constraints. The size of memory blocks has been minimized for keeping on-chip SRAM size realistic and for improving power efficiency. The authors of this paper have already shown in [40] that the power consumption of a TTA co-processing system such as *Inloop3* is between 66 mW and 207 mW when the clock frequency is ranges between 530 MHz and 1.2 GHz.

To get power estimates for the *Shared* system configurations, *Shared3* was placed and routed using a 28 nm standard cell library, which yielded a power consumption of 154 mW for the 1.0 V operating voltage and 1.0 GHz clock frequency. In the experiments, a 1.0 GHz clock is assumed in all *Shared* system configurations.

2. For this work the processor endianness has been changed from big-endian to little-endian and the SFU for FIFO operations (Section 4.5.2) has been added.

TABLE 1
Comparison of Inloop and Shared TTA cores

Processor	Inloop [37]	Shared [23]
Bitwidth	32	32
ALUs	0	2
Adders	3	0
Relational ops fu	1	0
Logic ops fu	2	0
Bitwise ops fu	2	0
Multipliers	1	1
LSUs	3	2
Int RFs	5×16	3×12
Bool RFs (1 bit)	1×2	1×2
Special Ops	9	0
Buses	5	6
Connectivity	Partial	Full
Instruction Width	131	258
Gate count (NAND2)	106K [40]	66K

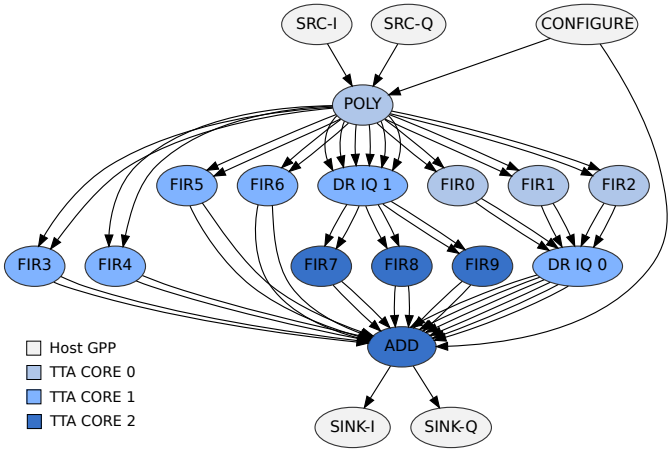

Fig. 12. The DPD application mapped to the triple-core TTA using data repeater (DR) actors

5.5 Mappings

In the case of the HEVC Inloop application, `Source` and `Sink` actors are always mapped to the host processor. For architectures *Inloop1* and *Shared1*, actors `DF-VE`, `DF-HE` and `SAO` are mapped to the same TTA core. In the triple-TTA core cases (*Inloop3* and *Shared3*), actors are mapped in a pipelined manner: `DF-VE` to core 0, `DF-HE` to core 1 and `SAO` to core 2. In the case of two and four triple-TTA clusters, the actor pipeline is replicated so that the input video is divided to 2 or 4 tiles, respectively (see Fig. 7).

The SDE application has two `Imageread` actors and an `Imagewrite` actor, which are mapped to the host processor in all cases. In the single TTA core cases, the input images are processed by two `Grayscale` actors, two `Sobel` actors and one `SAD` actor, which are all mapped to the same core. In the multicore cases the input images are divided to blocks so that the block number matches the number of TTA cores (see Fig. 8), and each core gets the same (but replicated) group of actors, as in the single core case.

When considering Fig. 10 or Fig. 11, it can be seen that the TTA co-processing system configurations are not fully connected, meaning that there is no direct connection between all cores. Therefore, there is a set of actor-to-core mappings, which are not feasible. (E.g. in the case of actors that have connections that are mapped to different clusters.) However, it is possible to reduce the set of unfeasible mappings by using *data repeater actors*.

Data repeater actors can be used when there is an indirect connection between processors through other processors. Data repeater actors are added to the actor network and mapped to the processors just for enabling communication between specific actors. For example, if actor `A` communicates with actor `B`, and actor `A` is assigned to the TTA core 0, and `B` to the TTA core 2 in *Inloop3*, there is no way to transfer data directly from TTA core 0 to TTA core 2. Due to that, the actor network is changed so that a new data repeater actor `DR` is placed between `A` and `B` and it is mapped to TTA core 1. `DR` actors are needed in the DPD application when triple-core TTA clusters are used. The actor-to-core mappings of *Inloop3* and *Shared3* cases are shown in Fig. 12.

TABLE 2
Performance and power measurements

Platform	Application					
	HEVC INLOOP		SDE		DPD	
	Perf. (fps)	Power (W)	Perf. (Mde/s)	Power (W)	Perf. (MS/s)	Power (W)
Inloop1	60.102	0.068	14.074	0.068	1.816	0.068
Inloop3	148.355	0.211	42.273	0.211	4.280	0.211
Inloop6	271.291	0.422	86.977	0.422	7.458	0.422
Inloop12	536.237	0.843	143.819	0.843	9.031	0.843
Shared1	11.657	0.050	7.314	0.050	1.409	0.050
Shared3	24.013	0.154	21.344	0.154	3.197	0.154
Shared6	49.156	0.309	42.963	0.309	5.252	0.309
Shared12	93.103	0.617	85.783	0.617	7.362	0.617
Odroid1	19.187	2.618	26.836	2.752	3.650	2.702
Odroid2	32.783	4.447	51.933	4.253	6.886	4.042
Odroid3	33.378	5.590	71.349	5.651	9.777	7.119
Odroid4	49.152	5.101	82.692	5.862	6.986	5.866
Odroid6	33.609	5.843	47.786	6.156	5.276	6.112
Odroid8	25.650	5.693	63.920	5.643	6.039	6.041

5.6 Results

The performance and power consumption results of different TTA co-processing architectures are presented in Table 2. The place and route results of *Inloop3* and *Shared3* are used to estimate power figures for other Inloop and Shared configurations. Fig. 13, Fig. 14 and Fig. 15 show the energy efficiency of the architectures for all test applications. As explained in Section 4, TTADF enables executing applications on the *host* cores only. To demonstrate this possibility, the performance and power consumption results of an Odroid XU3 platform have been included to the results.

The Odroid XU3 is powered by the mobile Samsung Exynos 5422 SoC, which includes four ARM Cortex-A15 @ 2.0 GHz and four Cortex-A7 cores @ 1.4 GHz, and utilizes

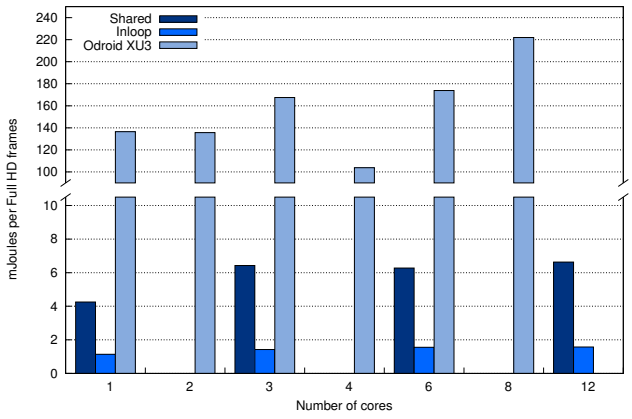

Fig. 13. Energy efficiency of the HEVC Inloop Filter application

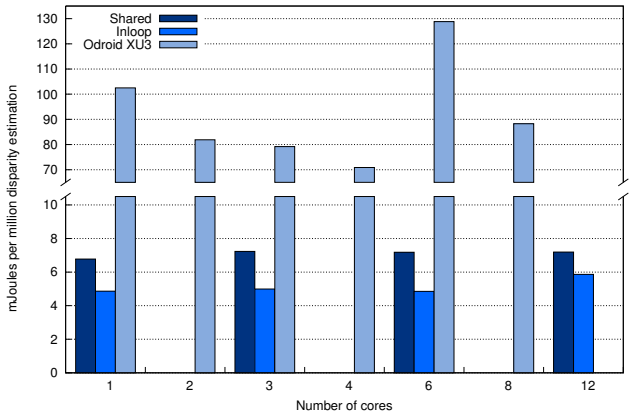

Fig. 14. Energy efficiency of the Stereo Depth Estimation application

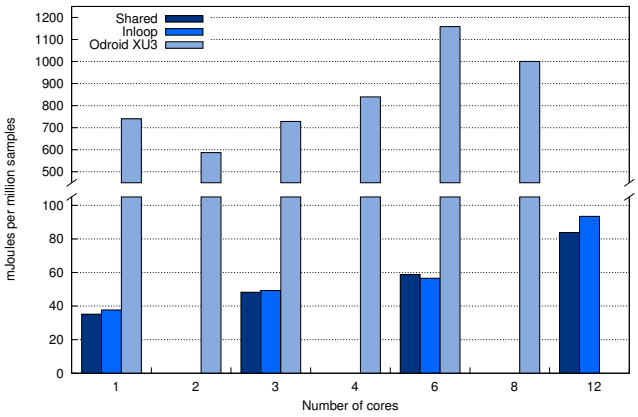

Fig. 15. Energy efficiency of the Dynamic Predistortion Filter application

the ARM big.LITTLE heterogeneous multi-processing solution. The dataflow applications are compiled for the Odroid using GCC 5.4 with an optimization level of 3. The operating system of the platform was Ubuntu Mate 16.04. The power figures of the Odroid XU3 platform contain only the power consumption of the processor cores, measured by current/-voltage sensors which are integrated to the platform.

Table 2 and Fig. 13 show the performance and energy efficiency results with the HEVC Inloop filtering workload. Thanks to the special function units, the *Inloop* configurations show superior performance and energy efficiency when compared to the Odroid-XU3 platform or the *Shared* configurations. For example, in the three-core case, *Inloop* can filter 148 frames per second, whereas the Odroid-XU3 can filter only 49 frames per second at best. Since HEVC Inloop filtering can easily be parallelized by using tiles, speedups are about $2\times$, $4\times$ or $8\times$ when the number of cores is increased from 1 to 3, 6 or 12 in the cases of *Inloop* and *Shared* configurations.

In stereo depth estimation the performance advantage of the *Inloop* based TTA co-processing system is narrow when compared to the *Shared* configurations, but it is still notable. This was expected since the special instructions of the *Inloop* cores cannot be utilized in SDE. On the other hand, when the same instruction memory size (64 KB) is used in all TTA cores, more aggressive loop unrolling can be used in the case of the *Inloop* core, due to its instruction width being significantly narrower (131 b) than that of the *Shared* core (258 b). When the FIFO SFU is exploited, instruction memory requirements decrease up to 55% per core. The dataflow implementation of SDE can easily be parallelized by processing multiple rows at the same time. The parallelization enables almost linear speedups when increasing the core count from single core to 3, 6, or 12 in the case of *Inloop* ($\times$3.0, $\times$6.1 and $\times$10.2) and *Shared* ($\times$2.9, $\times$5.8 and $\times$11.7) configurations. In single core cases, where all actors are mapped to the same core, actor scheduling overhead and conservative loop unrolling (due to program memory limits) decreases throughput with the consequence that speedups can become superlinear. The Odroid platform can utilize four cores efficiently, but going beyond that in the number of processing cores has only a negative impact. In the best case, Odroid can compute 82 Mde/s.

Dynamic Predistortion filter performance results are presented in Table 2 and energy efficiency figures in Fig. 15. *Inloop* based TTA co-processing architectures outpace *Shared* architectures by a small margin regardless of the number of cores. *Inloop* configurations can filter from 1.8 up to 9.0 mega samples per second (MS/s), depending on the number of cores. Both *Inloop* and *Shared* show considerable speedup when the core count is 3 or 6, but in the case of 12 cores, the DPD application structure limits the achievable speedup. That is caused by the fact that the workloads of the cores are varying due to dynamic application behavior: FIR filters can be switched on and off dynamically. The workload of the DPD application is more suitable to the Odroid XU3 than the other test applications. The best result for the Odroid is 9.8 Ms/s when using three cores. This is more than double when compared to the *Inloop3* or *Shared3* configurations. However, increasing the number of cores over three does not give any performance advantage. As

with other workloads, TTA-based platforms show over 10× higher energy efficiency than the Odroid.

6 Comparison of results

Key figures from related existing TTA based works have been collected to Table 3 for comparison. Yviquel et al. [23] use RVC-CAL descriptions of applications, and have implemented a complete HEVC decoder using 12 *Fast* TTA processors. The *fast TTA* is similar to the *Shared* TTA, but it has one additional ALU (arithmetic-logical unit), more registers and 18 transport buses in total. The implementation can decode five 720p video frames per second.

As the authors of [23] have observed that inloop filtering takes about 22% of the total decoding time for Full High Definition (FHD) frames, it was possible to scale the results of [23] to match our test case of HEVC inloop filtering: the scaled performance for inloop filtering in [23] is 14.7 FHD frames per second, which is about 7.6 times to 36.5 times slower than with our *Shared12* and *Inloop12* configurations, respectively. Although the *Fast* TTA has more computational resources than the *Shared* TTA, the Orcc TTA Backend implementation on 12 *Fast* TTAs has equal performance as the proposed TTADF implementation of a single *Shared* TTA. That can be a consequence of one or both of the following issues: 1) the RVC-CAL description of HEVC inloop filtering is not efficient, or 2) the Orcc TTA backend cannot produce high-quality LLVM code for TTA processors.

Since the proposed *Inloop* configuration uses tailored SFU units for HEVC inloop filtering, it is not surprising that this configuration is about five times faster than the *Shared* configuration. For exposing possible overhead caused by the TTADF framework, the manually tuned implementation of [40] was included in the comparison (Table 3, Inloop TTA × 3). In the case of *Inloop3*, TTADF has only 3% overhead which means five frames per second.

Nyländen et al. [41] implemented a highly optimized OpenCL based SAD depth estimation algorithm for a tailored data-parallel SIMD TTA accelerator. In their work, 16-bit floating point arithmetic is used instead of 32-bit integer arithmetic to decrease memory requirements and for achieving better energy efficiency by slightly sacrificing image quality. Compared to the SDE application presented in this work, their algorithm does not include Sobel filtering or uniqueness thresholding. Their implementation runs on a single tailored SIMD TTA core clocked at 800 MHz, which can compute 117 Mde/s, whereas *Inloop1* can compute only 14.9 Mde/s, about eight times less. *Inloop12* can compute 30% more Mde/s than a single SIMD TTA core, which shows that the TTADF SDE implementation is scalable. Because the *Inloop* architecture is not optimized for the SDE workload, moderate performance results are not a surprise. On the other hand, *Inloop3* can compete with the general purpose Intel Core i5-440M mobile processor, while *Inloop6* achieves the performance of the Qualcomm Adreno 330 mobile GPU. Surprisingly, the Odroid XU3, using the TTADF implementation of SDE, has better performance than the OpenCL SDE implementation running on an Intel Core i5-480M.

Finally, the DPD application offers a fair comparison between the proposed TTADF framework and its closest competitor, the Orcc TTA Backend [23]. The DPD application was written in the RVC-CAL language, and a multicore TTA implementation was generated for the application using the Orcc TTA Backend. Very similar TTA processor cores were used, as our *Shared* core is essentially the same as the *Custom* core available in the Orcc TTA Backend. Performance results show that the Orcc TTA Backend based implementation produces 2.1, 2.9 and 4.0 MS/s per second for 3, 6 and 12 *Custom* TTA cores, respectively. In comparison, the DPD implementations produced by TTADF using the *Shared* TTA configuration were on average 2 × faster. Also, performance scaling as a function of core count was slightly better for TTADF, since *Shared6* and *Shared12* were ×1.9 and ×2.3 faster than *Shared3*, while the corresponding speedups were 1.4× and 1.6× for the Orcc TTA Backend.

7 Discussion

The TTADF and the Orcc TTA Backend frameworks can essentially be used for the same purpose, but their design flows have a substantial difference. In TTADF, the designer separately specifies the system architecture (TTA core definition, core connections, and host connections), after which the dataflow application is mapped to the architecture. In contrast, in the Orcc TTA backend, the system architecture (TTA core interconnections) is derived from the dataflow application. From this viewpoint, TTADF can be considered to be more generic. On the other hand, the Orcc TTA Backend provides more automation due to automatic interconnect generation and actor mapping features. Besides, Orcc offers high-level dataflow analysis features which are currently not available in TTADF.

TTADF enables the possibility to map actors to the *host* processor, and in simulations, these actors are executed on the host system of the framework. Testing of an individual actor on a particular TTA core is easy and fast since test data for the actor is created by other actors of the application at runtime.

A previous paper [40] from the authors of the proposed work presented a 3-core TTA accelerator for HEVC inloop filtering with manually optimized C code, which achieved 152 HD frames/s processing performance at 207 mW. Now, as TTADF has been measured to achieve a throughput of 148 HD frames/s with practically the same processor core, but generated from a generic dataflow-based design framework, it is justified to state that *TTADF offers a way to raise the abstraction level of multicore co-design with negligible impact on performance*. Additionally, the experimental results suggest that TTADF can outperform the current state of the art, the Orcc TTA backend, by a clear margin regarding performance. Especially the possibility of exploiting special function units gives a substantial competitive advantage for TTADF over the Orcc TTA backend.

In the future, various new features can be adapted to TTADF: automatic mapping of actors to cores and automatic creation of data repeater actors. When considering hardware, the possibility of power saving could be achieved by observing FIFO fill counts. There are also plans to directly support SoC FPGAs so that a TTA co-processing system is placed on the programmable logic, and the *host*-mapped actors are executed on hard processor cores.

TABLE 3
Comparison of different programmable implementations of the test case applications

Application	Architecture	Framework	Tech. (nm)	Clkf (MHz)	Perf.	Power (mW)	Energy Eff. (mJ/ Perf. unit)
HEVC Inloop	[23] Fast TTA × 12	Orcc	40	1000	14.67 fps [1]	-	-
	Shared12	TTADF	28	1200	93.1 fps	617	6.63
	Inloop12	TTADF	28	1200	536 fps	843 [2]	1.57
	[40] Inloop TTA × 3	None	28	1200	153 fps	207	1.35
	Inloop3	TTADF	28	1200	148 fps	211 [2]	1.43
	Shared3	TTADF	28	1000	24.0 fps	154	6.42
Stereo Depth Estimation	[41] OpenCL SIMD TTA	OpenCL	28	800	117 Mde/s [3]	33	0.28
	[41] Intel Core i5-480M	OpenCL	32	2600	30.3 Mde/s [3]	35000	1155
	[41] Qualcomm Adreno 330	OpenCL	28	578	99.1 Mde/s [3]	1800	18.16
	Inloop1	TTADF	28	1200	14.9 Mde/s	69 [2]	4.63
	Inloop3	TTADF	28	1200	44.8 Mde/s	211 [2]	4.71
	Inloop12	TTADF	28	1200	152 Mde/s	843 [2]	5.55
	Odroid-XU3	TTADF	28	2000	82.1 Mde/s	5861	71.4
Dynamic Predistortion Filter	Shared3 (Orcc TTA Backend)	Orcc	28	1000	1.75 Msample/s	154	88.0
	Shared6 (Orcc TTA Backend)	Orcc	28	1000	2.38 Msample/s	309	135.5
	Shared12 (Orcc TTA Backend)	Orcc	28	1000	3.31 Msample/s	617	248.79
	Shared6	TTADF	28	1000	5.2 Msample/s	309	59.42
	Inloop6	TTADF	28	1200	7.46 Msample/s	422 [2]	56.57

[1]Estimated assuming that the share of inloop filtering is 22% [23] of total HEVC decoding workload
[2]Estimate based on work [40], [3]16-bit floating point, no sobel filtering and uniqueness thresholding

8 Conclusion

In this paper, TTADF, a dataflow framework dedicated to transport triggered architectures is presented. TTADF enables software synthesis of dynamic dataflow applications to a TTA based co-processing system, which can be co-designed with the dataflow software. The dataflow descriptions of applications are written using C and XML. The design flow achieves energy efficient implementation by offering many design options:

- Special (custom) operations by calls from C code,
- Hybrid memory architecture for reduced congestion and improved access times,
- Hardware accelerated FIFO access operations that save instruction memory.

TTADF enables three different simulation approaches with different accuracy and speed tradeoffs, including C++, SystemC, and mixed HDL simulations. TTADF was evaluated using three applications from different fields, consisting of video coding, machine vision, and wireless communications. The experimental results show that the energy efficiency of the TTADF-generated system falls within 3% of a manually designed baseline and that the generated multiprocessing platform overcomes a commercial multicore by 10× in energy efficiency while providing similar or better performance.

Acknowledgments

The work was partially funded by the Academy of Finland project 309693 UNICODE and Tauno Tönning Foundation.

References

[1] S. Bhattacharyya, R. Leupers, and P. Marwedel, "Software synthesis and code generation for signal processing systems," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, no. 9, pp. 849–875, 2000.

[2] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor system-on-chip (MPSoC) technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, 2008.

[3] P. G. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goossens, "Embedded software in real-time signal processing systems: application and architecture trends," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 419–435, 1997.

[4] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 2003, pp. 137–147.

[5] G. Martin, "Overview of the MPSoC design challenge," in *Design Automation Conference, 2006 43rd ACM/IEEE*. IEEE, 2006, pp. 274–279.

[6] W. M. Johnston, J. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, 2004.

[7] J. Castrillon, R. Leupers, and G. Ascheid, "Maps: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 527–545, 2013.

[8] H. Yviquel, J. Boutellier, M. Raulet, and E. Casseau, "Automated design of networks of transport-triggered architecture processors using dynamic dataflow programs," *Signal Processing: Image Communication*, vol. 28, no. 10, pp. 1295–1302, 2013.

[9] E. Bezati, S. Casale-Brunet, M. Mattavelli, and J. W. Janneck, "High-level synthesis of dynamic dataflow programs on heterogeneous MPSoC platforms," in *Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016 International Conference on*. IEEE, 2016, pp. 227–234.

[10] M. Dardaillon, K. Marquet, T. Risset, J. Martin, and H.-P. Charles, "A new compilation flow for software-defined radio applications on heterogeneous MPSoCs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 2, p. 19, 2016.

[11] J. Boutellier, J. Wu, H. Huttunen, and S. S. Bhattacharyya, "PRUNE: Dynamic and Decidable Dataflow for Signal Processing on Heterogeneous Platforms," *IEEE Transactions on Signal Processing*, vol. 66, no. 3, pp. 654–665, 2017.

[12] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*. New York, NY, USA: John Wiley & Sons, Inc., 1997.

[13] P. Jääskeläinen, V. Guzma, A. Cilio, T. Pitkänen, and J. Takala, "Codesign toolset for application-specific instruction set processors," in *Multimedia on Mobile Devices 2007*, vol. 6507. International Society for Optics and Photonics, 2007, p. 65070X.

[14] O. Esko, P. Jääskeläinen, P. Huerta, C. S. de La Lama, J. Takala, and J. I. Martinez, "Customized Exposed Datapath Soft-Core Design Flow with Compiler Support," in *Proceedings of the 2010*

International Conference on Field Programmable Logic and Applications. Washington, DC, USA: IEEE Computer Society, 2010, pp. 217–222.
[15] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*. Springer, 1974, pp. 362–376.
[16] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.
[17] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.
[18] H.-W. Park, H. Oh, and S. Ha, "Multiprocessor SoC design methods and tools," *IEEE Signal Processing Magazine*, vol. 26, no. 6, 2009.
[19] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity-the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.
[20] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "Orcc: Multimedia development made easy," in *Proceedings of the 21st ACM International Conference on Multimedia*, 2013, pp. 863–866.
[21] K. Jerbi, H. Yviquel, A. Sanchez, D. Renzi, D. De Saint Jorre, C. Alberti, M. Mattavelli, and M. Raulet, "On the Development and Optimization of HEVC Video Decoders Using High-Level Dataflow Modeling," *Journal of Signal Processing Systems*, vol. 87, no. 1, pp. 127–138, 2017.
[22] M. Chavarrias, F. Pescador, M. J. Garrido, E. Juarez, and M. Raulet, "A DSP-Based HEVC decoder implementation using an actor language dataflow model," *IEEE Transactions on Consumer Electronics*, vol. 59, no. 4, pp. 839–847, 2013.
[23] H. Yviquel, A. Sanchez, P. Jääskeläinen, J. Takala, M. Raulet, and E. Casseau, "Embedded multi-core systems dedicated to dynamic dataflow programs," *Journal of Signal Processing Systems*, vol. 80, no. 1, pp. 121–136, 2015.
[24] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele, "Scenario-based design flow for mapping streaming applications onto on-chip many-core systems," in *Proc. International Conference on Compilers Architecture and Synthesis for Embedded Systems (CASES)*, Oct 2012, pp. 71–80.
[25] L. Schor, A. Tretter, T. Scherer, and L. Thiele, "Exploiting the Parallelism of Heterogeneous Systems using Dataflow Graphs on Top of OpenCL," in *Proc. IEEE Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*. Montreal, Canada: IEEE, Oct 2013, pp. 41–50.
[26] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming," in *2014 6th European Embedded Design in Education and Research Conference (EDERC)*,, Sept 2014, pp. 36–40.
[27] J. Zhang, J.-F. Nezan, M. Pelcat, and J.-G. Cousin, "Real-time GPU-based local stereo matching method," in *2013 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, 2013, pp. 209–214.
[28] J. Boutellier, O. Silvén, and M. Raulet, "Automatic synthesis of TTA processor networks from RVC-CAL dataflow programs," in *2011 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 2011, pp. 25–30.
[29] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–87. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673
[30] J. Boutellier and O. Silvén, "Towards generic embedded multiprocessing for RVC-CAL dataflow programs," *Journal of Signal Processing Systems*, vol. 73, no. 2, pp. 137–142, 2013.
[31] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems," in *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, Nov. 2006.
[32] A. Tretter, J. Boutellier, J. Guthrie, L. Schor, and L. Thiele, "Executing Dataflow Actors As Kahn Processes," in *Proceedings of the 12th International Conference on Embedded Software*, ser. EMSOFT '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 105–114.
[33] M. Michalska, N. Zufferey, J. Boutellier, E. Bezati, and M. Mattavelli, *Efficient scheduling policies for dynamic data flow programs executed on multi-core*, ser. 11th International Meeting on Logistics Research, 2016, iD: unige:91453. [Online]. Available: https://archive-ouverte.unige.ch/unige:91453
[34] G. J. Sullivan, J. Ohm, W.-J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1649–1668, Dec. 2012.
[35] A. Norkin, G. Bjontegaard, A. Fuldseth, M. Narroschke, M. Ikeda, K. Andersson, M. Zhou, and G. Van der Auwera, "HEVC Deblocking Filter," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1746–1754, Dec. 2012.
[36] C. M. Fu, E. Alshina, A. Alshin, Y. W. Huang, C. Y. Chen, C. Y. Tsai, C. W. Hsu, S. M. Lei, J. H. Park, and W. J. Han, "Sample Adaptive Offset in the HEVC Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1755–1764, Dec. 2012.
[37] I. Hautala, J. Boutellier, J. Hannuksela, and O. Silvén, "Programmable low-power multicore coprocessor architecture for HEVC/H. 265 in-loop filtering," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 7, pp. 1217–1230, 2015.
[38] K. Mühlmann, D. Maier, J. Hesser, and R. Männer, "Calculating dense disparity maps from color stereo images, an efficient implementation," *International Journal of Computer Vision*, vol. 47, no. 1-3, pp. 79–88, 2002.
[39] M. Abdelaziz, A. Ghazi, L. Anttila, J. Boutellier, T. Lähteensuo, X. Lu, J. R. Cavallaro, S. S. Bhattacharyya, M. Juntti, and M. Valkama, "Mobile transmitter digital predistortion: Feasibility analysis, algorithms and design exploration," in *2013 Asilomar Conference on Signals, Systems and Computers*. IEEE, 2013, pp. 2046–2053.
[40] I. Hautala, J. Boutellier, and O. Silvén, "Programmable 28nm coprocessor for HEVC/H.265 in-loop filters," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2016, pp. 1570–1573.
[41] T. Nyländen, H. Kultala, I. Hautala, J. Boutellier, J. Hannuksela, and O. Silvén, "Programmable data parallel accelerator for mobile computer vision," in *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 2015, pp. 624–628.

Ilkka Hautala received his M.Sc. (Tech.) degree from the Department of Electrical and Information Engineering at the University of Oulu (Finland) in 2013. He is currently a doctoral student in the Center for Machine Vision Research at the University of Oulu. His research interests include low-power design, multicore processor architectures and video coding techniques.

Jani Boutellier received the M.Sc. and Ph.D. degrees from the University of Oulu, Finland, in 2005 and 2009, respectively. Currently he is an Associate Professor at the School of Technology and Innovations, University of Vaasa, Finland. His research interests include dataflow programming, design and implementation of deep learning algorithms, and heterogeneous computing. He is a member of the IEEE Signal Processing Society DISPS Technical Committee.

Olli Silvén received the M.Sc. and Ph.D. degrees in electrical engineering from the University of Oulu, Finland, in 1982 and 1988, respectively. Since 1996, he has been a professor of signal processing engineering at the University of Oulu. His main research interests are in embedded signal processing and machine vision system design. He has contributed to the development of numerous solutions from real-time 3-D imaging in reverse vending machines to IP blocks for mobile video coding.