Journal of Systems Architecture 93 (2019) 1–19

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

An integrated hardware/software design methodology for signal processing systems

Lin Li[a,*], Carlo Sau[b], Tiziana Fanni[b], Jingui Li[c], Timo Viitanen[c], François Christophe[e], Francesca Palumbo[d], Luigi Raffo[b], Heikki Huttunen[c], Jarmo Takala[c], Shuvra S. Bhattacharyya[a,c]

[a] University of Maryland, ECE Department, College Park, MD 20742, United States
[b] University of Cagliari, Department of Electrical and Electronic Engineering, Italy
[c] Tampere University, Finland
[d] University of Sassari, PolComIng-Information Engineering Unit, Italy
[e] Department of Computer Science, University of Helsinki, Finland

ARTICLE INFO

Keywords:
Dataflow
Model-based design
Hardware/software co-design
Low power techniques
Deep learning
Signal processing systems

ABSTRACT

This paper presents a new methodology for design and implementation of signal processing systems on system-on-chip (SoC) platforms. The methodology is centered on the use of lightweight application programming interfaces for applying principles of dataflow design at different layers of abstraction. The development processes integrated in our approach are software implementation, hardware implementation, hardware-software co-design, and optimized application mapping. The proposed methodology facilitates development and integration of signal processing hardware and software modules that involve heterogeneous programming languages and platforms. As a demonstration of the proposed design framework, we present a dataflow-based deep neural network (DNN) implementation for vehicle classification that is streamlined for real-time operation on embedded SoC devices. Using the proposed methodology, we apply and integrate a variety of dataflow graph optimizations that are important for efficient mapping of the DNN system into a resource constrained implementation that involves co-operating multicore CPUs and field-programmable gate array subsystems. Through experiments, we demonstrate the flexibility and effectiveness with which different design transformations can be applied and integrated across multiple scales of the targeted computing system.

1. Introduction

Model-based design has been widely studied and applied over the years in many domains of embedded processing. Dataflow is well-known as a paradigm for model-based design that is effective for embedded digital signal processing (DSP) systems [1]. In dataflow-based modeling, signal processing applications are represented as directed graphs (dataflow graphs), and computational functions are modeled as vertices (actors) in these graphs. Actors exchange data packets (tokens) through unidirectional, first-in, first-out (FIFO) communication channels that correspond to dataflow graph edges. Many dataflow-based design methods for DSP systems have been explored in recent years to support various aspects of design and implementation, including modeling and simulation; scheduling and mapping of actors to heterogeneous platforms; and buffer management (e.g. see [1,2]).

The diversity of design scales and dataflow techniques that are relevant to signal processing systems poses major challenges to achieving the fully potential that is offered by signal processing platforms under stringent time-to-market constraints. While automated techniques, such as those referred to above for scheduling and buffer mapping, are effective for specialized combinations of platforms and dataflow models (e.g., multicore CPUs and synchronous dataflow, respectively), they are limited in their ability to support more comprehensive assessment of the design space, where the models and target platforms themselves have great influence on addressing implementation constraints and optimization objectives. System designers must therefore resort to ad-hoc methods to explore design alternatives that span multiple implementation scales, platform types, or dataflow modeling techniques.

In this work, we propose a design methodology and an integrated set of tools and libraries that are developed to help bridge this gap. We refer to this methodology as the STMC Methodology or STMCM, which is named after the different institutions across which it is developed (Sassari, Tampere, Maryland, Cagliari). STMCM focuses on enabling experimentation across different levels of abstraction throughout

* Corresponding author.
E-mail addresses: lli12311@umd.edu (L. Li), carlo.sau@diee.unica.it (C. Sau), tiziana.fanni@diee.unica.it (T. Fanni), jingui.li@tuni.fi (J. Li), timo.2.viitanen@tuni.fi (T. Viitanen), francois.christophe@helsinki.fi (F. Christophe), fpalumbo@uniss.it (F. Palumbo), raffo@unica.it (L. Raffo), heikki.huttunen@tuni.fi (H. Huttunen), jarmo.takala@tuni.fi (J. Takala), ssb@umd.edu (S.S. Bhattacharyya).

https://doi.org/10.1016/j.sysarc.2018.12.010
Received 27 April 2018; Received in revised form 4 November 2018; Accepted 31 December 2018
Available online 31 December 2018

the design process, and allowing designers to experiment productively and iterate rapidly on complex combinations of design options, including dataflow models, heterogeneous target platforms, and integration with platform-specific languages and back-end tools. Special emphasis is placed on enabling effective experimentation with hardware/software design trade-offs, as well as trade-offs involving performance, resource utilization, and power consumption. These are trade-offs that are especially important and challenging to navigate efficiently in design processes for system-on-chip implementation of signal process systems.

The utility of STMCM is facilitated by the use of lightweight dataflow (LWDF) programming [3], and its underlying core functional dataflow (CFDF) model of computation [4]. LWDF provides a compact set of application programming interfaces (APIs) that allows one to apply signal-processing-oriented dataflow techniques relatively easily and efficiently in the context of existing design processes, target platforms, and simulation- and platform-oriented languages, such as MATLAB, C, CUDA, and VHDL. Additionally, CFDF is a general form of dataflow that accommodates more specialized forms of dataflow, such as Boolean dataflow [5], cyclo-static dataflow [6], synchronous dataflow [7], and RVC-CAL [8] as natural special cases. This accommodation of different dataflow models in turn provides potential to integrate designs with other dataflow frameworks and DSP libraries, such as those described in [8–13]. Furthermore, LWDF is granularity-agnostic, in the sense that actor complexity does not limit the applicability of the framework.

To demonstrate the capabilities of STMCM in addressing the challenges of mapping practical dataflow-based structures on heterogeneous signal processing platforms, we explore different implementations of a deep neural network (DNN) for vehicle classification on a heterogeneous, embedded system-on-chip (SoC), the Xilinx Zynq Z-7020 SoC. DNN applications pose great challenges in their deployment on embedded devices. Investigation of DNN implementations on embedded SoC devices is challenging due to the limited resources for processing and storage in these devices, and especially, due to the high computational complexity of DNNs. They involve very large and complex signal flow structures that involve intensive computation, data exchange, and multilayer processing. These characteristics make embedded DNN implementation highly relevant as a case study for STMCM.

2. Related work

Dataflow provides valuable model-based design properties for signal processing systems, and has been adopted in a wide variety of tools for both software and hardware design. For example, LWDF APIs for CUDA and C have been targeted in the DIF-GPU tool for automated synthesis of hybrid CPU/GPU implementations [14]. The CAL programming language and the Open RVC-CAL Compiler (Orcc) toolset provide a dataflow environment for generating dataflow implementations in a number of languages, such as C, Jade, and Verilog [8,9,15] (note that the Verilog backend of Orcc has been discontinued and Xronos synthesizer has been replaced). The CAPH language and framework generate hardware description language (HDL) code from high-level dataflow descriptions [10].

The work in [16] presents an integrated design flow and tools for the automatic optimization of dataflow specifications to generate HDL designs. The Multi-Dataflow Composer (MDC) tool is a dataflow-to-hardware framework able to automatically create multi-functional reconfigurable architectures. In addition to this baseline functionality, MDC offers three additional features: (1) a structural profiler to perform a complete design space exploration, evaluating trade-offs among resource usage, power consumption and operating frequency [17]; (2) a dynamic power manager to perform, at the dataflow level, the logic partitioning of the substrate to implement at the hardware level, and apply a power saving strategy [18]; (3) a coprocessor generator to perform the complete dataflow-to-hardware customization of a Xilinx compliant multi-functional IP [16].

All of the methodologies and tools described above are limited by the programming language, adopted dataflow description, or implementation target. For example, HDL code can be highly optimized for a given target (such as a Xilinx FPGA) but not usable for an application specific integrated circuit (ASIC) flow (e.g., see [15,19,20]). Automatic methods and tools require significant effort in development and maintenance of graph analysis and code generation functionality, and may be too costly for models and design approaches that are not mature. Such scenarios may arise for emerging applications or platforms that do not match effectively with the models or methods supported by available tools.

STMCM is complementary to these efforts that emphasize dataflow design automation. By applying LWDF APIs in novel ways, STMCM facilitates implementation of and iterative experimentation with new dataflow-based hardware/software architectures and design optimization techniques. LWDF is applied as an integral part of STMCM because of LWDF's minimal infrastructure requirements and its potential for rapid retargetability to different platforms and actor implementation languages. Furthermore, LWDF does not have any restriction in terms of actor granularity and can be extended with different combinations of dataflow graph transformations, as well as other forms of signal processing optimizations (e.g., see [1]).

In [21], we presented an efficient integration of the LWDF methodology with hardware description languages (HDLs). Building on this HDL-integrated form of LWDF, we developed methods for low power signal processing hardware implementation, and system-level trade-off exploration. In this paper, we apply the hardware design techniques introduced in [21] as part of a general methodology that spans software, hardware, and mixed hardware/software design, implementation, and trade-off exploration. Thus, while the focus in [21] is on rigorous integration across digital hardware design, lightweight dataflow programming, and low power optimization, the emphasis in this paper is on a methodology for applying LWDF concepts in an integrated manner across complete hardware/software development processes for embedded signal processing systems.

In summary, STMCM provides methods to seamlessly and comprehensively integrate LWDF-based actor implementation techniques with design processes for real-time, resource-constrained signal processing systems. STMCM can be used as an alternative to or in conjunction with more conventional automated dataflow tools (e.g., for disjoint subsystems). STMCM requires more effort in programming compared to fully automated toolchains, however it provides more agility in terms of retargetability and experimentation, as described above. This is a useful trade-off point to have available for model-based design of complex signal processing systems.

3. Proposed design methodology

Our proposed methodology STMCM is illustrated in Fig. 1. As motivated in Section 1 and Section 2, STMCM is a design methodology that emphasizes LWDF concepts, and is specialized for SoC-based signal processing systems. The upper part of Fig. 1 represents application-specific and algorithmic aspects, while the lower part represents the general part of the methodology that is reusable across different applications. The upper part is illustrated concretely in the context of DNN system design; this part can be replaced with other application/algorithm level design aspects when applying STMCM to other applications.

In STMCM, we apply the LWDF programming model through the Lightweight Dataflow Environment (LIDE). LIDE is a software tool for dataflow-based design and implementation of signal processing systems [3,22]. LIDE is based on a compact set of application programming interfaces (APIs) that is used for instantiating, connecting, and executing dataflow actors. These APIs have been implemented in a variety of implementation languages. For example, LIDE-C [22] and LIDE-V [21] provide C and Verilog language implementations of the LIDE APIs, respectively.

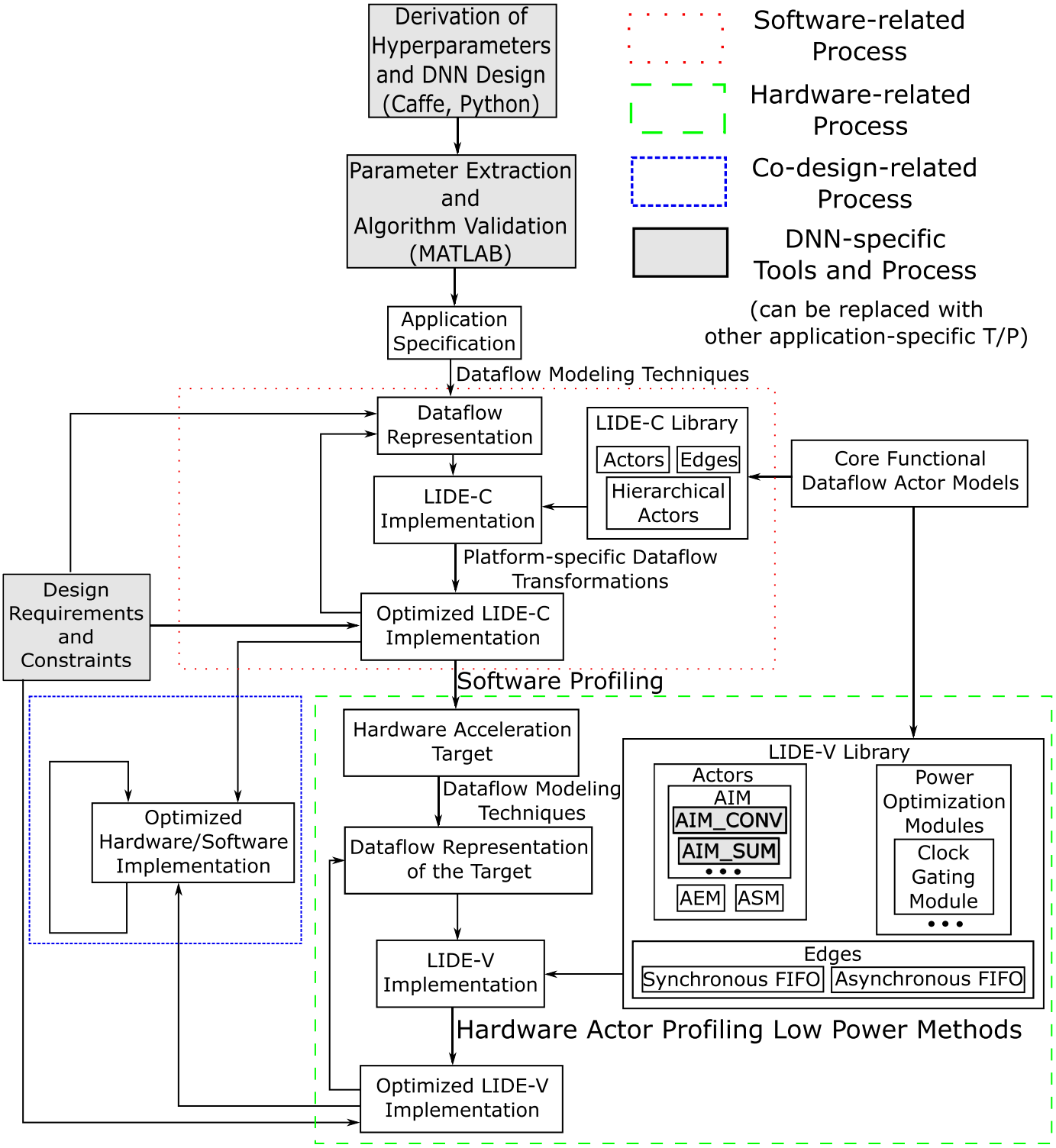
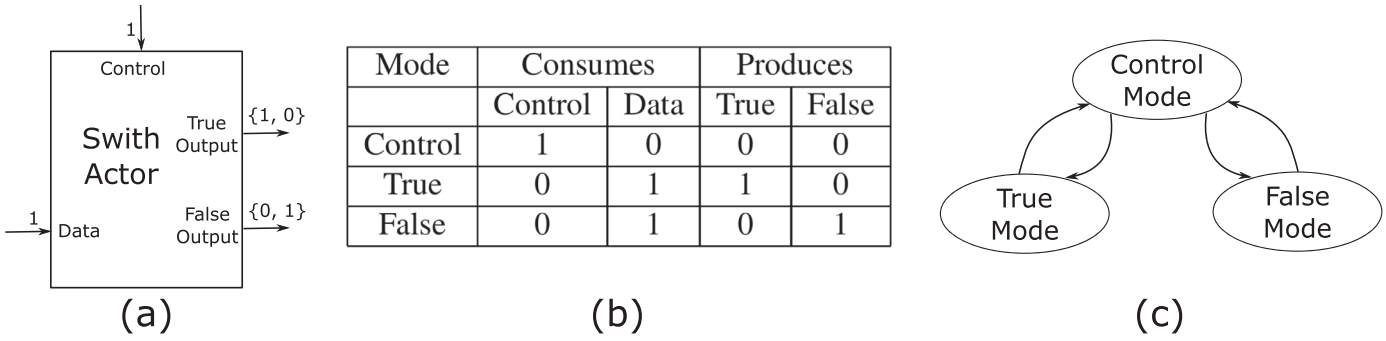


Fig. 1. An illustration of STMCM in the context of DNN system design.

As mentioned in Section 1 and illustrated in Fig. 1, core functional dataflow (CFDF) [4], is the form of dataflow that LWDF is based on. In CFDF, each actor is specified as a set of modes. Each actor firing operates according to one of the specified modes (called the "current mode" associated with the firing), and determines a unique next mode, which will be the current mode for the next firing. The production and consumption rates (*dataflow rates*) for the actor ports are constant for a given mode. However, different modes of the same actor can have different rates, which allows actors to exhibit dynamic dataflow behavior. We present the switch actor as an example of CFDF actor. Switch actor has three modes: Control, True and False. In Control mode, the switch actor consumes one token from Control port. In True or False mode, the switch actor consumes one token from Data port and forward that token to True or False Output port accordingly. The dataflow table and mode transition diagram between CFDF modes of switch actor are illustrated in Fig. 2.



Mode	Consumes		Produces	
	Control	Data	True	False
Control	1	0	0	0
True	0	1	1	0
False	0	1	0	1

(b)

Fig. 2. Switch actor in CFDF. (a) Switch Actor, (b) Dataflow Table, (c) Mode Transition Diagram between CFDF Modes.

The definition of a CFDF actor includes two functions called the *enable function* and *invoke function* of the actor. The enable function checks whether there is sufficient data available on the actor's input edges and sufficient empty space available on the output edges to fire the actor in its next mode. The invoke function executes an actor firing according to the actor's current mode, consuming and producing amounts of data that are determined by the fixed dataflow rates of the current mode. The invoke function also determines the actor's next mode, as described above.

In the remainder of this section, we discuss in detail the application-, software-, and hardware-specific processes illustrated in Fig. 1.

3.1. Application-specific tools and processes

In Fig. 1, application-specific tools and associated design processes are illustrated by gray blocks. Throughout this paper, we adopt a DNN application as a concrete demonstration of how such application-specific aspects are used as an integral part of STMCM. The DNN-focused design process illustrated in Fig. 1 starts with the derivation of DNN hyperparameters and the network configuration. Then the parameters associated with the derived DNN structure are extracted and the DNN algorithm is carefully validated to ensure that target levels of accuracy are satisfied.

The block labeled "Design Requirements and Constraints" refers to the application- and platform-specific requirements and constraints on the DNN implementation. Examples of these include the accuracy and throughput requirements for image classification DNN systems, and constraints on available power and hardware resources for a targeted SoC platform.

In the remainder of this section, we introduce the software-related and hardware-related design processes that provide the core of STMCM. These processes are applied in an integrated manner for hardware/software co-design, as represented by the lower left hand part of Fig. 1. Detailed explanations of the major components in STMCM are provided in Section 4.3.

3.2. Software-related process

In the next main phase of the proposed design methodology, the DNN network configuration derived using application-specific, algorithm-level tools is mapped to a software implementation using LIDE-C. Note that LIDE-C is in no way restricted to DNN systems, and is instead designed to support a broad class of dataflow-based signal and information processing systems. For example, in the work of [23], the design space exploration of a digital predistortion system for wireless communication is based on implementation using LIDE-C. In [24], LIDE-C is extended to support parameterized synchronous dataflow [25] modeling and applied to the implementation of an adaptive wireless communication receiver. In [26], optimized vectorization techniques are applied to LIDE-based actors for throughput optimization, and demonstrated using an Orthogonal Frequency Division Multiplexing (OFDM) receiver. For more details about LIDE-C and the development of DNN components in LIDE-C, we refer the reader to [22,27].

Working with the LIDE-C implementation of the DNN, a number of optimization processes are carried out iteratively to streamline the software implementation in terms of the relevant design objectives and constraints. This iterative optimization process is illustrated in Fig. 1 by the cyclic path that involves the blocks labeled *Dataflow Representation, LIDE-C Implementation, and Optimized LIDE-C Implementation*. The proposed approach supports efficient application of commonly-used DNN software optimization methods such as for-loop tiling and buffer memory sharing among dataflow graph edges. We refer the reader to Section 4.1 for more details about these optimization methods and the integration of them with the LIDE-C implementation.

Next, software profiling is performed on the optimized LIDE-C implementation of the DNN system to extract profiling data. This data is extracted for each dataflow component of the DNN architecture. In the profiling process applied in STMCM, the memory sizes of the buffers and execution time of the actors in the graph are measured. According to the characteristics of DNN architecture, the DNN system is divided into multiple computation layers. In our application of STMCM, software profiling is specialized to DNN implementation by measuring the total memory sizes for the buffers both inside each layer and between pairs of adjacent layers. We also measure the total time complexity of each DNN layer.

3.3. Hardware-related process

The dataflow model of the subgraph to accelerate is implemented in hardware using LIDE-V. Hardware profiling based on the specific implementation platform is performed on the LIDE-V implementation. This profiling is used to collect measurements on hardware performance and help identify possible optimizations. Details on hardware profiling are demonstrated concretely through the case study presented in Section 4.2. Like the software implementation, the hardware implementation will in general go through multiple optimization iterations before it is finalized.

In LIDE-V, the hardware implementation of a dataflow actor is decomposed into implementations of its enable function and invoke function. These components are implemented as two coupled Verilog modules — the actor enable module (AEM), and actor invoke module (AIM). Dataflow edges are implemented as dataflow edge modules (DEMs); we informally refer to DEMs also as "FIFOs".

To provide fully distributed scheduling of actors, one can connect a LIDE-V *actor scheduling module* (*ASM*) to each actor. The ASM initiates a new firing of its associated actor any time the actor is not already in the firing mode, has sufficient data on its input edges, and has sufficient empty space on its output edges. Scheduling of LIDE-V actors is not restricted to such a fully distributed scheduling approach. For example, with appropriately-designed control logic, subsets of actors can be serialized to allow sharing of resources within the subsets. In this paper,

however, we restrict our attention to fully distributed scheduling. Fully distributed scheduling of dataflow graphs has been analyzed in various contexts. For example, Ghamarian el al. have developed methods for throughput analysis of synchronous dataflow graphs that are scheduled in a fully distributed manner [28]. Such analysis techniques can be applied to hardware subsystems in STMCM.

The orthogonality (separation of concerns) among actor, edge, and scheduler design in LIDE-V lays a valuable foundation for rigorous integration of power-management within the associated APIs. In particular, we demonstrated in [29] and [21] that methods for asynchronous design, Globally Asynchronous Locally Synchronous (GALS) design, and clock gating can be applied efficiently through natural extensions of the LIDE-V APIs. We also demonstrated the use of these extensions to power optimization.

To manage complexity and improve reuse of subsystems within and across designs, one can encapsulate subgraphs in *LIDE-V* within *hierarchical actors* (*HAs*). An HA in LIDE-V appears from the outside as a regular (non-hierarchical) LIDE-V actor with an associated AEM, AIM, and ASM. Execution of an HA as an actor in the enclosing dataflow graph is coordinated by the *external scheduler* associated with the HA. When an HA is fired by its external scheduler, the *internal scheduler* of the HA coordinates the firings of actors that are encapsulated within the HA (nested actors). The internal scheduler carries out the set of nested actor firings that must be completed for a given firing of the HA. An example of an HA with internal and external schedulers is discussed in detail and illustrated in Fig. 6.

Since it appears from the outside as a regular actor, an HA can be clock gated in exactly the same way, allowing the designer to efficiently switch off the whole subgraph at appropriate times during operation.

4. Case study: A deep neural network for vehicle classification

As a concrete demonstration of STMCM, we adopt a DNN use case for automatic discrimination among four types of vehicles — bus, car, truck, and van. This implementation is based on a neural network design presented in [30], where a network configuration — i.e., the number and types of layers and other DNN hyperparameters — was carefully derived and demonstrated to have very high accuracy. The accuracy of the methods was validated with a database of over 6500 images, and the resulting prediction accuracy was found to be over 97%. The work in this paper and the work in [30] have different focuses. The work of [30] focuses on deriving hyperparameters, network design, and demonstrating network accuracy, and does not address aspects of resource-constrained implementation or hardware/software co-design. In this paper, we go beyond the developments of [30] by investigating resource constrained implementation on a relevant SoC platform, and optimized hardware/software co-design involving an embedded multicore processor and FPGA acceleration fabric that are integrated on the platform. In [30], the proposed DNN architectures are evaluated based on the classification accuracy, while in our work on STMCM, the objectives that we are trying to optimize are system throughput, memory footprint and power efficiency. In addition, our work in this paper can be generalized to the design and implementation of arbitrary DNN architectures, and also it can be generalized beyond DNN applications to other signal and information processing applications; the architecture of [30] is selected as a case study to concretely demonstrate the usage of the methodology proposed in this paper.

In relation to Fig. 1, we apply the results from [30] in the block labeled “derivation of hyperparameters and DNN design” as part of the design methodology that is demonstrated in this paper. Fig. 3 illustrates the complete DNN architecture that we implement in this work. For more details about this use case, such as the dataset, the derivation of the DNN architecture and the application of the use case in vehicle classification, we refer the reader to [30].

The DNN network design is composed of two convolutional layers, two dense layers and one classifier layer, as depicted in Fig. 3. The first convolutional layer takes an RGB image ($3\times96\times96$) as input, and produces 32 feature maps, each with dimensions (48×48). The second convolutional layer takes these 32 feature maps as input and produces 32 smaller feature maps, each having dimensions (24×24). We refer to a subsystem that processes multiple input images to produce a single feature map as a *branch*. Thus, the first and second convolutional layers have 32 branches each. The two dense layers combine to transform the feature maps into a (1×100) vector, which is then multiplied in the classifier layer by a (100×4) matrix to determine the (1×4) classification result. Each of the four values in the result corresponds to the likelihood that the vehicle in the input image belongs to one of the four vehicle types (i.e., bus, car, truck and van).

The studied use case is relatively easy to solve compared to common image recognition benchmarks, such as MSCOCO [31], or ImageNet [32]. Therefore, one can reach high accuracy with a relatively simple network requiring significantly lower resources than common network topologies intended for mobile use (such as Mobilenets). As such, the focus of our work is not in mobile devices (e.g., smartphones), but in simpler IoT devices targeted to solving less complex machine learning problems at low cost. For further details on the DNN network design and hyperparameter specifications, we refer the reader to [30].

The specific platform and associated platform-based tools that we employ are based on the Xilinx Zynq Z-7020 SoC. The remainder of this Section focuses on details associated with STMCM and its associated design processes. These details are presented concretely through the development of this DNN case study.

4.1. Software implementation and optimization

In this section, we discuss dataflow-graph- and actor-level optimizations and associated design iterations, as illustrated in Fig. 1 by the blocks labeled Dataflow Representation, LIDE-C Implementation, and Optimized LIDE-C Implementation. We start with a dataflow graph implementation that is derived using LIDE-C [3,22], which provides a C-language implementation of the LWDF APIs so that CFDF-based actors and dataflow graphs can be implemented in a structured manner using C. The initial (sequential) LIDE-C design is developed in a design phase that corresponds to the block labeled LIDE-C Implementation in Fig. 1.

After validating the correct, dataflow-based operation of the initial DNN dataflow graph implementation in LIDE-C, we experiment with various transformations at the actor, subgraph, and dataflow graph levels. Here, we exploit the orthogonality of actor, edge, and graph implementation in LIDE-C, which allows designers to flexibly and efficiently perform experimentation with a wide variety of transformations, and with different combinations of applied transformations. The actor-level transformations performed here are focused on optimization methods applied to the convolution actor, which is a major performance bottleneck in the design. The subgraph-level transformations involve memory management optimizations performed on FIFOs both inside each subgraph (DNN layer) and between pairs of adjacent layers.

4.1.1. Actor-level optimization

We demonstrate actor-level optimization at this stage of the design process using the convolution actor in our DNN example. In our LIDE-C implementation of this actor, we apply a transformation of the convolution computation that is commonly used to simplify the design, and improve classification speed. The transformation involves loop tiling to reduce the cache miss rate. The utility of loop tiling in DNN implementation has been demonstrated previously, for example, in [33]. Using loop tiling, we decompose the main loop of the convolution computation into an inner loop that iterates within contiguous “strips” of data, and an outer loop that iterates across strips. Applying loop tiling in this way allows one to enhance cache reuse based on an array size (strip length) that fits within the cache.

Fig. 4 shows a segment of code from our application of the tiling transformation to the convolution actor.

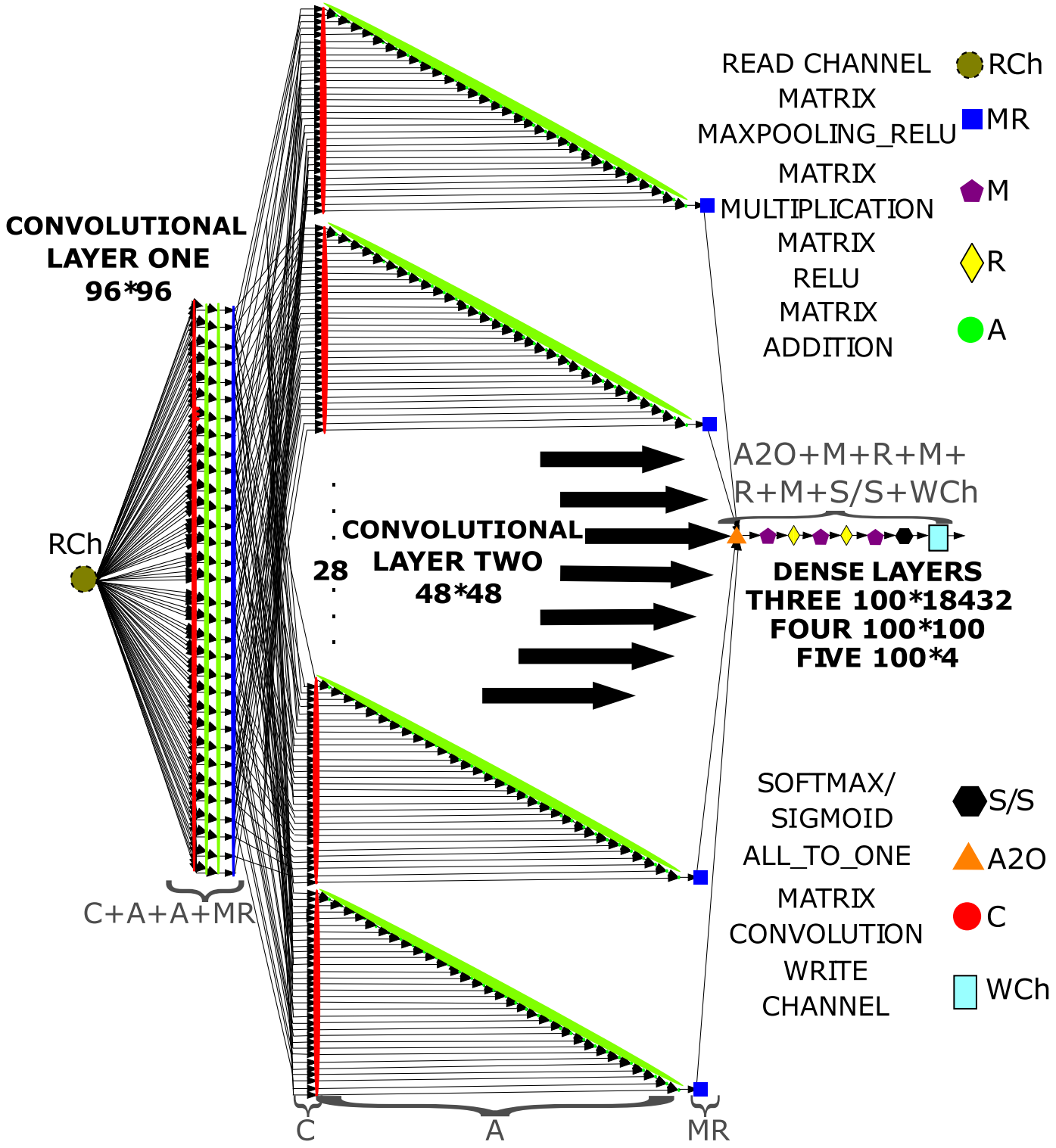


Fig. 3. DNN for automatic discrimination of four types of vehicles.

Through the orthogonality provided by the model-based design rules in LIDE-C, this transformation can be applied at a late stage in our design process, in a way that is interoperable with previously applied transformations, and in a way that requires no modifications to other actor or edge implementations. In this case, no modification is needed to the dataflow graph scheduler implementation as well, although for some transformations, scheduler adjustments can be useful to integrate transformed actors into the overall system in an optimized way. The CFDF-based APIs (enable and invoke functions) in LIDE-C for scheduler implementation allow the designer to experiment efficiently with such scheduling adjustments as needed.

4.1.2. *Buffer memory management*

A major challenge in resource-constrained implementation of a DNN architecture is managing the large volume of data transfers that are carried out during network operation. Each DNN layer typically processes

```
for (row = 0; row < L; row += tile_num)  //tiling for row
  for (col = 0; col < L; col += tile_num)  //tiling for col
    for (row_tile = row; row_tile < ((row + 1) < L ? (row + 1) : L); row_tile++)
      for (col_tile = col; col_tile < ((col + 1) < L ? (col + 1) : L ) ; col_tile++)
        for (i = 0; i < n; i++)   //sliding window
          for (j = 0; j < n; j++)
            *(output+row_tile*L+col_tile) +=
              *(zero_pad+(row_tile+i)*(L+4)+(col_tile+j)) *
                (*(conv_wgt+ (conv_num-1-i)*conv_num +(conv_num-1-j));
```

Fig. 4. The code segment that implements loop tiling within the LIDE-C actor for convolution.

a large amount of data, and requires memory to store the input data from the previous layer or subsystem, the intermediate data during the computation processing, and the computation results that will be transmitted to the following layer or subsystem.

Consider, for example, the buffer memory costs (the storage costs associated with the dataflow graph edges) for the DNN of Fig. 3. In our LIDE-C implementation, the second convolutional layer requires the most buffer memory. In this layer, each of the 32 branches is composed of 32 convolution actors, 31 addition actors and one actor performing both maxpooling and ReLU (Rectified Linear Unit). Given that the size of the input feature map processed by each branch is 48×48 pixels, the buffer memory required for actor communication inside each branch is $image\_size \times (number\_of\_conv\_actors + number\_of\_output\_feature\_maps)$, which is $48 \times 48 \times (32 + 1) = 76{,}032$ pixels. Thus, the total buffer memory inside the second convolutional layer is $76{,}032 \times 32 = 2{,}433{,}024$ pixels. The buffer memory required for data communication between the first and the second layer can be computed as $48 \times 48 \times 32 = 73{,}728$ pixels.

In STMCM, we apply a buffer memory optimization technique that is useful for resource-constrained DNN implementation. In particular, we incorporate a new FIFO abstract data type (ADT) implementation in LIDE-C, called *shared FIFO*, that enables multiple dataflow edges in a graph to be implemented through FIFO ADT instances that share the same region of memory. Such *buffer sharing* in dataflow implementations has been investigated in different forms for various contexts of automated scheduling and software synthesis (e.g., see [34–36]). In STMCM, we make it easy for the system designer to apply buffer sharing *explicitly* within her or his implementation rather than depending on its implicit support through the toolset that is used. This is an example of the agility that is supported in STMCM, as described at the end of Section 2.

Again, by exploiting the orthogonality among dataflow components, buffer sharing in STMCM is performed only on the targeted dataflow edges and requires no modification to other actors or subgraphs. Through the support for such separation of concerns in LIDE-C, different ADT implementations for a FIFO or group of FIFOs can be interchanged without affecting overall system functionality.

There are three key aspects to our application of shared FIFOs in our LIDE-C DNN implementation. First, at the input of each convolutional layer L, input data from the previous layer is stored centrally instead of being copied separately into each branch of L. Second, edges in different layers share the same memory so that the memory is time-division multiplexed between the layers — the processing of a given layer overwrites memory in its shared FIFOs without introducing conflicts that affect the computation results. Third, actors operate on data from shared input FIFOs directly through their read pointers into the FIFO (rather than first copying the data locally within the actor's internal memory). This kind of copy-elimination is similar to dataflow memory management techniques introduced by Oh and Ha [35].

Improvements resulting from our application of shared FIFOs are demonstrated quantitatively in Section 5.1.

Table 1
Layer-level software profiling. Here, the row labeled "T" gives the execution time of each layer, and the row labeled "T%" gives the percentage of the total DNN execution time that is attributed to each layer.

	Layer					Total
	1	2	3	4	5	
T [ms]	18.71	22.08	0.0149	0.0034	0.0036	40.812
T%	45.84	54.10	0.04	0.01	0.01	100

Table 2
Actor-level software profiling.

Layer	Convolutional layer 1			Convolutional layer 2		
Actor	Conv	Add	M&ReLU	Conv	Add	M&ReLU
T_{ic} [μs]	230.10	0.03	0.025	59.77	0.005	0.006
Layer	Dense Layer 3		Dense Layer 4		Output Layer 5	
Actor	Mult	ReLU	Mult	ReLU	Mult	Softmax
T_{ic} [μs]	5.1	0.0012	0.029	0.0012	0.0023	0.0031

4.1.3. Software profiling

In this subsection, we demonstrate the process of software profiling, as illustrated in Fig. 1, in the context of our optimized LIDE-C implementation of the DNN architecture. The implementation platform is an Intel i7-2600K running at 3.4GHz. Table 1 and Table 2 show layer- and actor-level software profiling measurements, respectively.

In Table 2, T_{ic} denotes the *invoke to firing completion time* of a given actor. This is the average time that elapses between the time that an actor firing is initiated and when the firing completes. We also refer to T_{ic} as the *average execution time* of the associated actor. The abbreviations Add, Conv, Mult, and M&ReLU stand, respectively, for Addition, Convolution, Multiplication, and Maxpool-and-ReLU.

Layer- and actor-level software profiling provide insight into the processing complexity of actors in each layer. According to Table 1, the convolutional layers account for 99.94% of the system execution time. Also, the execution time of Layer 2 is very close to that of Layer 1. In both convolutional layers, the Conv actors account for most of the processing time compared with the other two actors — Add and M&ReLU — in the convolutional layers. Additionally, the average execution time of the Conv actors in Layer 2 is only about a quarter of that of the Conv actors in Layer 1. This is primarily because each of the Conv actors in Layer 1 processes input images of size 96×96, while the Conv actors in Layer 2 process input feature maps that have size 48×48.

4.2. Hardware implementation and design exploration

In this section, we describe the main capabilities of the design flow depicted in Fig. 1 with respect to design and implementation of hardware accelerators. These capabilities are represented by the blocks in the region labeled "Hardware-related Process".

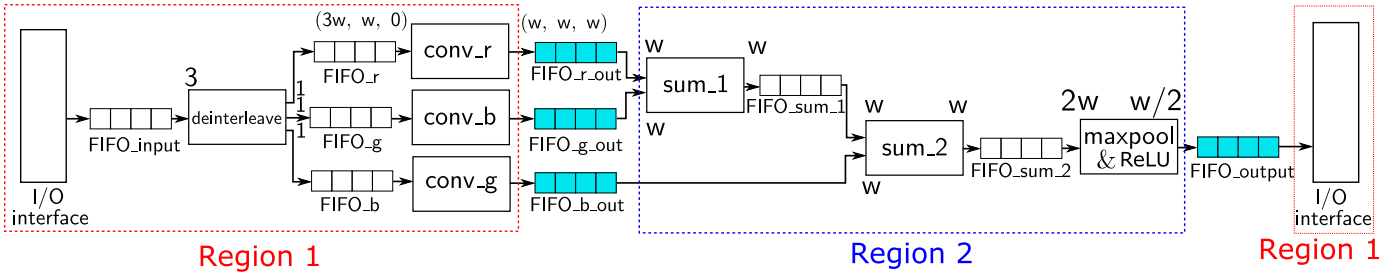


Fig. 5. LIDE-V implementation for the accelerated SFM.

For example, through a preliminary hardware profiling phase of the DNN application described in Section 4.2.1, we can identify three hardware design aspects that are interesting to investigate in detail — the adoption of clock gating techniques, exploitation of asynchrony that is inherent in dataflows, and exploration of different levels of actor granularity.

We demonstrate the hardware-related design process of STMCM using a hardware accelerator that is introduced in [21]. The accelerator provides a subsystem for producing feature maps from the first convolutional layer of the DNN application. In the remainder of this paper, we refer to this subsystem as the *Subtree for Feature Map* (SFM).

Due to the interfacing consistency that is maintained across LIDE actor implementations in different languages, one can readily convert the LIDE-C based SMF subsystem implementation into hardware by replacing each software actor with a hardware module that is designed in LIDE-V, and by connecting the derived hardware actors with LIDE-V FIFOs. Following the general approach of realizing LIDE actors in hardware, each LIDE-V actor implementation is decomposed into an AEM and AIM. The AEM is reusable among different actors in our implementation, although in general it can be useful to have specialized AEM implementations that are streamlined for the specific requirements of individual actors [21].

The hardware implementation diverges from the LIDE-C design in two major ways. First, we feed the input data in an interleaved format, reducing the complexity of the hardware interface and driver software since there is only one input FIFO to manage. Second, the hardware actors are designed to produce one row per firing instead of entire images. This reduces the FIFO size requirements in the first layer from 96×96 pixels to only 96 pixels. The hardware actors in our implementation are scheduled using a fully distributed approach.

The resulting SMF is shown in Fig. 5. The implemented hardware is verified against reference outputs extracted from the LIDE-C implementation. In this Figure, production and consumption rates (dataflow rates) are annotated next to actor ports, and w is the input image width. The convolution actor has multiple operating modes (CFDF modes) with different consumption rates.

4.2.1. Hardware profiling

We employ hardware profiling in STMCM to extract execution time data, which is later used to guide the process of iterative design optimization. In this section, we demonstrate hardware profiling in the context of our DNN application. Profiling is performed using the target platform, which in our demonstration is the Zynq Z-7020 SoC. We profile the LIDE-C implementation on the ARM A9 MPCores provided by the target platform and develop a first version implementation of the SFM on this platform and extract execution time data from this implementation.

Table 3 depicts various data associated with execution times and waiting times for the SFM hardware accelerator illustrated in Fig. 5. Here, the symbol t_{tot} represents the total time necessary to execute the *SFM;* T_{ic} is the average time period between an actor invocation and its corresponding firing completion; T_{ci} is the average time period that an actor has to wait to be fired after its previous firing completion; *firings* is the number of firings of a given actor during execution of *SFM; Tot*, calculated as $(T_{ic}) \times (firings)$, gives the total execution time of a given actor during the execution of *SFM*; $T_{ii} = (T_{ic} + T_{ci})$ denotes the average time period between the beginning of one invocation to the beginning of the next; and the ratio T_{ii}/T_{ic} measures the extent of actor idleness.

Table 3
Measured data associated with actor execution times and waiting (idle) times.

SFM t_{tot}	232,831				
	T_{ic}	T_{ci}	*firings*	*Tot* (*Tot*%)	T_{ii}/T_{ic}
Deinterleave	3	2	9216	27,648 (11.87)	1.67
Convolution	2402	2	96	230,592 (99.04)	1.00
Sum	107	2297	96	10,272 (4.41)	22.46
Maxpool&ReLU	195	4613	48	9360 (4.02)	24.66

This rich collection of metrics, which is supported by the underlying CFDF model computation, provides various insights on the dataflow-based system architecture and its implementation. For example, the T_{ii}/T_{ic} ratio provides insight on differences in processing speed that are useful in exploiting the inherent asynchrony between dataflow actors.

From analysis of our hardware profiling results (Table 3), we can derive different versions of the SFM hardware accelerator with different trade-offs among power consumption, system throughput, and hardware resource cost. Firstly, looking at column *Tot*%, we see that all of the actors except for Convolution are inactive throughout most of the execution time. The maximum proportion of active time among these actors is 11.87%, reached by Deinterleave. Gating the clock of these frequently inactive actors can provide more energy efficient accelerator operation by eliminating dynamic power consumption during idle phases.

Furthermore, the Deinterleave and Convolution actors have relatively small idleness levels (T_{ii}/T_{ic}), with a waiting time T_{ci} equal to 2 clock cycles for both of them. On the other hand, Sum and Maxpool&ReLU exhibit much larger waiting times and idleness levels. An important hint coming from the T_{ci} values is that, thanks to the inherent asynchrony of dataflow actors, it is possible to partition the design into different clock regions working at different frequencies, thus obtaining a GALS design. In particular, the *Deinterleave* and *Convolution* actors can be placed in one clock region (Region 1), driven by *clock 1*, while *Sum* and *Maxpool&ReLU* can be placed in another region (Region 2), driven by *clock 2*. On the basis of the measured T_{ii}/T_{ic} values, we can set *clock 2* to be 20 times slower than *clock 1*.

Moreover, the subgraph included in Region 2 can be encapsulated into a hierarchical actor (see Section 3.3). This actor, seen from the "outside", is like any other LIDE-V actor. The actor and its encapsulated subsystem can be clock gated or clocked with a different frequency, providing additional candidate solutions for SFM accelerator optimization.

4.2.2. SFM Exploration

Based on the hardware profiling analysis discussed in Section 4.2.1, we explored six different variants of the SFM design:

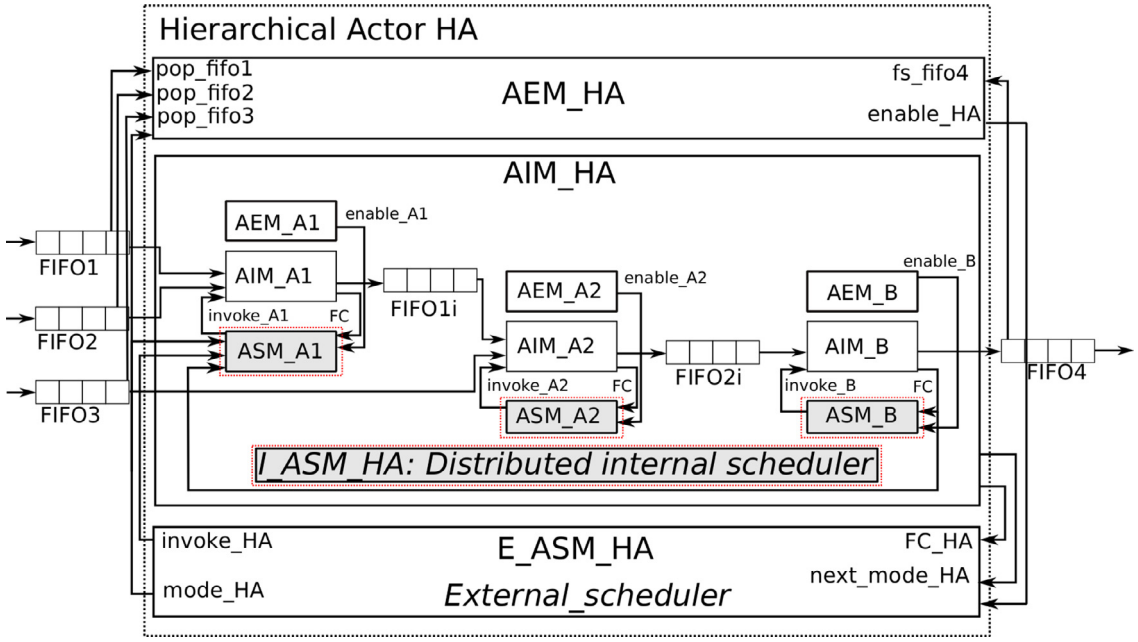


Fig. 6. An illustration of the hierarchical actor associated with Design SFM_h.

- SFM_a: This is an asynchronous design where actors belonging to different logic regions run at different clock frequencies. In particular, the clock frequency for *clock 1* is set to 100 MHz, and the clock frequency for *clock 2* is set to 5 MHz. Referring to Fig. 5, the only modification required in the design is the replacement of FIFOs that are placed *between* the two clock regions. These FIFOs need to be replaced with asynchronous FIFOs — for this purpose, we employ the clock domain crossing (CDC) FIFOs presented in [21]. CDC FIFOs are designed with read and write logic that can be driven by different clocks. At the same time, their module interfaces conform to standard LIDE-V edge interfaces so they can replace other FIFO implementations without requiring changes to actors that communicate with them.
- SFM_{CG}: Based on our hardware profiling results, we apply clock gating to the Deinterleave, Sum and Maxpool&ReLU actors. To be clock gated, a LIDE-V actor needs only the instantiation of a clock gating module (CGM) [21]. The CGM involves a BUFG primitive that physically enables/disables the clock signal in the target SoC. Thus for each clock gated actor A in SFM_{CG}, a CGM is instantiated and connected to the clock inputs of A and to the read- and write-clock inputs, respectively, of the FIFOs that A reads from and writes to.
- SFM_{aCG}: This design incorporates both asynchronous design and clock gating techniques. As in SFM_a, the FIFOs between the two clock regions are replaced with CDC FIFOs. Additionally, the Deinterleave, Sum and Maxpool&ReLU actors are clock gated as in SFM_{CG}, and a CGM is instantiated for each of these actors.
- SFM_h: This is a hierarchical SFM design, which can be viewed as a baseline for evaluating our enhanced hierarchical design SFM_{hCG} (defined below). In SFM_h, Region 2 (see Fig. 5) is encapsulated in a hierarchical actor H. An illustration of this hierarchical actor is provided in Fig. 6. The subgraph that is encapsulated by H contains three actors *A1, A2* and *B*. We denote this subgraph by G_H. Actors *A1* and *A2* correspond to *Sum 1* and *Sum 2*, respectively, which are two actors that add outputs from the three convolution actors. Actor *B* corresponds to the Maxpool&ReLU actor.
  When H is viewed as a single actor from the outside, a firing of H starts when the internal scheduler *I_ASM_HA* for G_H receives the *invoke_HA* signal from the external scheduler *E_ASM_HA*. Inside the subgraph G_H, the *invoke_HA* signal is received by *ASM_A1*, which is the ASM of actor *A1*. Once *ASM_A1* receives the *invoke_HA* signal, the firing of the subgraph G_H starts.
- SFM_{hCG}: This design is the same as SFM_h, except that the Deinterleave actor and the hierarchical actor are clock gated. It is important to highlight that the application of clock gating at the region level is advantageous if the execution times of the actors within the region are overlapped. In this design, however, the execution times of the three actors are not overlapped. When one actor is executed, the others wait in an idle state and waste power. Therefore, we expect that this configuration would not be really effective in reducing power consumption as SFM_{CG} in the targeted DNN case. However, we include the test in our explorations to present the complete wide variety of options made available by STMCM (even if some of them may be less efficient than others for this particular application scenario).
- SFM_{auto}: This is a version of the SFM that is synthesized and implemented by enabling the automatic power optimization available within the adopted Xilinx Vivado environment. This design applies fine-grain clock-gating and fine-grain logic-gating at the Verilog level and *excludes* all of the higher-level, dataflow-based optimizations (coarse-grain asynchronous design, clock-gating, and hierarchical decomposition) that are applied in the other five investigated designs. Thus, SFM_{auto} is useful as a common baseline to assess the higher-level models and transformations provided by *STMCM* compared to existing off-the-shelf synthesis techniques.

4.3. Joint hardware/software implementation and optimization

This section shows how the proposed design flow (summarized in Fig. 1) provides a variety of interesting hardware/software co-design implementation choices and optimization possibilities. In particular, these features are represented by the “Co-design-related Process” area of Fig. 1. For a given high-level LWDF model, the interaction between software (see Section 4.1) and hardware (see Section 4.2) actors or subgraphs can be shaped and refined depending on the specific constraints and requirements of the application.

In particular, we demonstrate two main implementation aspects that can be efficiently explored with STMCM: parallelism across actor execution, and the adopted communication interfaces. The degree of parallelism can be tuned depending on the number of software and/or hard-

ware cores adopted for the execution of a certain computational step, while different communication interfaces allow different levels of coupling between hardware and software actors. Both of these dimensions for exploration therefore represent important sources of trade-offs to consider during the implementation process.

For the purpose of our co-design explorations, the DNN application has been split into two parts to be executed respectively in software (PS) and hardware (PL). Here, PS and PL stand for Processing System and Programmable Logic, respectively. In our experiments, we consider the SFM subsystem introduced in Section 4.2 as the portion of DNN application that will be accelerated in the PL, while the remaining part, involving the second convolutional layer, two dense layers and final classification layer, will be executed by the PS.

Note that the first convolutional layer constitutes only one of the main computationally intensive steps of the DNN application. According to software profiling results that are based on the SoC platform that we applied for hardware/software co-design (see Table 8), the first convolutional layer only accounts for about 27% of the prediction time. For this reason, the speedup brought by hardware acceleration to the overall DNN application is not dramatic, as will be discussed further in Section 5. However, the results concretely demonstrate how STMCM can be applied to perform extensive design space exploration across a variety of diverse designs to achieve system performance enhancement under highly-constrained hardware resource availability.

The SFM accelerator has been integrated into the LIDE-C design presented in Section 4.1 by replacing the SFM software implementation with function calls to driver software that is capable of offloading the computation to the PL. We have experimented with using a Linux kernel driver based on the Userspace I/O (UIO) framework [37], and a driver that is independent of the Linux kernel and operates by directly accessing memory with the mmap system call. The UIO approach is more suitable for production use, while mmap works well for prototyping, and this latter approach has been used in this work for evaluation. The PS and PL can communicate by means of AXI interfaces exploiting General Purpose (GP) ports; 32-bit width PS master or slave ports with 600 Mbps bandwidth for both read and write channels; High Performance (HP) ports or Accelerator Coherency Ports (ACP); and 64-bit width PS slave ports with 1200 Mbps bandwidth for both read and write channels.

Fig. 7 depicts the reference configuration for the co-design explorations. In order to integrate the accelerator into the SoC, a generic AXI wrapper for hardware dataflow subgraphs has to be provided. The wrapper is compliant with the adopted AXI interface and lets the programmer access the input and output FIFOs of the dataflow graph and monitor their populations. For this purpose, the wrapper includes all the necessary logic for the communication management.

In our hardware acceleration approach, we map the SFM subsystem to hardware. This subsystem produces a 48x48 feature map on each execution. Thus, in order to perform the entire first convolutional layer of the DNN application, which must produce 32 48x48 feature maps, the SFM accelerator has to be executed 32 times with the appropriate convolution coefficients. For each of these SFM executions, the PS will send the corresponding convolution coefficients to the accelerator. The input image, which remains the same across all 32 executions, is sent only once from the PS and stored within a local buffer within the accelerator. All 32 executions of the SFM access the input image from this local buffer. In this way, we avoid the large amount of data transfer that would be required if the input image had to be sent separately from the PS to the PL for each SFM execution. Upon completion of each SFM execution, the PS retrieves the resulting feature map from the accelerator.

In the remainder of this section, we discuss in detail three different sets of co-design implementations and optimizations that are facilitated by STMCM:

- the amount of parallelism that is exploited in the software and hardware subsystems;
- two alternative communication interfaces that offer different trade-offs in terms of resource requirements and execution speed; and
- local buffering to avoid redundant transmission of common data across different branches of the SFM accelerator.

These three sets of co-design explorations are discussed further in Section 4.3.1, Section 4.3.2, and Section 4.3.3, respectively.

4.3.1. Exploiting parallelism

STMCM allows the designer to experiment efficiently with the amounts of parallelism that are exploited in both the hardware and software subsystems (see the dashed squares in Fig. 7). In particular, depending on the specific application requirements, multiple parallel instances of software cores or hardware accelerators can be utilized. While software cores are able to execute all DNN application steps, hardware accelerators can only perform the steps that they have been conceived for. Generally speaking, hardware accelerators achieve higher efficiency than software cores when executing a given computational step, both in terms of execution time and resource efficiency (resource utilization and consumption).

In the targeted Xilinx Zynq Z-7020 SoC platform, a pair of homogeneous cores is available, so that the maximum degree of software parallelism in our implementations is 2. The available cores are both ARM A9 MPCores with two levels of cache and access to a 512 Mb off-chip DDR RAM. In our experiments, we have exploited software parallelism for the two most computationally intensive steps of the application — the two convolutional layers.

When using FPGA fabric, designers have the possibility to utilize as much parallelism as the FPGA resources allow. In this work, we have investigated three alternative designs that utilize 1, 2 or 4 parallel SFM instances, respectively, in the same hardware accelerator. In the first case, the accelerator is executed 32 times in order to complete the 32 branches of the first convolutional layer. This design executes a different branch with different convolution coefficients for each accelerator invocation. In the second case (2 parallel SFM instances), the accelerator execution time is halved, but for each run, two new sets of convolution coefficients are necessary. Finally, with 4 parallel SFM instances, only 8 accelerator executions are needed, with each requiring the updating of four different sets of coefficients.

4.3.2. Communication interfaces

During the process of co-design exploration, STMCM gives the designer significant flexibility to select interfaces for communicating data between the hardware and software subsystems. This flexibility is provided by the general dataflow model of computation that underlies STMCM. Flexibility in selecting a communication interface can be very useful in the context of resource- or performance-constrained design. This is demonstrated, for example, by the work of Silva et al., which analyzes trade-offs among the different AXI interface options [38].

We investigated the usage of two different AXI communication interfaces located at the extremes of the resource-versus-performance trade-off:

- the memory-mapped AXI4-lite (*mm-lite*) interface; and
- FIFO-based AXI4-stream (*stream*) interface.

Compared to the stream interface, the mm-lite interface has lower resource requirements, but it also exhibits lower performance. The mm-lite interface uses memory-mapped, one-by-one transfer of data items. The interface is particularly intended for control signals and small-scale data accesses. It does not need any additional modules beyond those depicted in Fig. 7, and it uses only one of the PS master GP ports. For example, the execution of one branch of the first layer requires the input images (3 RGB images with 96×96 pixels each) and kernel coefficients (3 kernels with 5×5 coefficients each). Since the mm-lite interface uses a separate data transfer operation for each pixel, this results in a total of $3 \times 96 \times 96 + 3 \times 5 \times 5$ data transfer operations. Once the accelerator

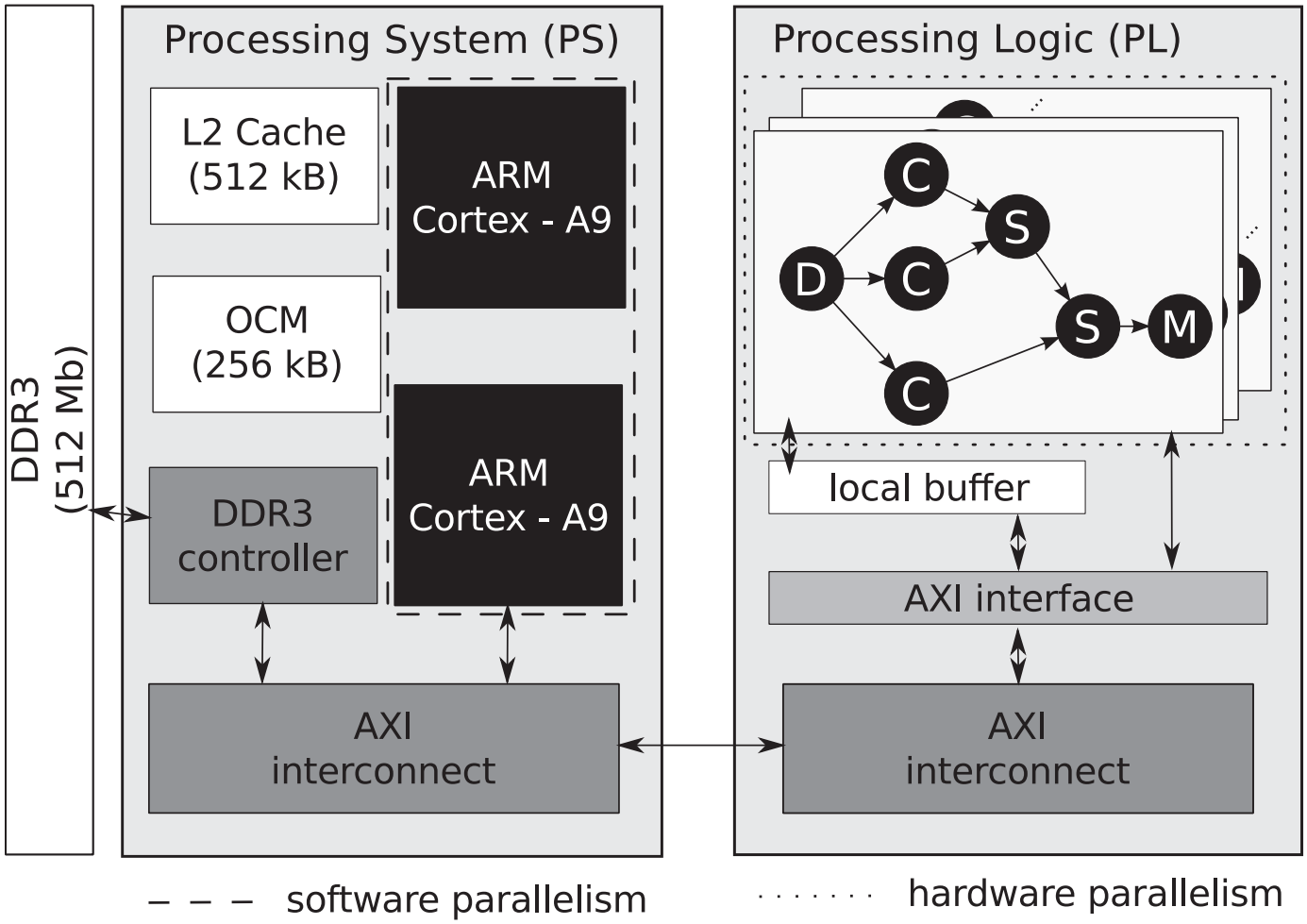


Fig. 7. Reference configuration for hardware/software co-design exploration in our experiments.

completes its computation, the mm-lite interface requires 48×48 data transfer operations to enable the processor to read the output feature map.

Unlike the mm-lite interface, which performs data transfers one-by-one, the stream interface employs a DMA engine that transfers data between processor memory and the accelerator in blocks, where the block size can be up to 256 bytes. Successive data items within a block are transferred in consecutive clock cycles. The stream interface requires a DMA engine, as mentioned above, and additional FIFO buffers, and therefore incurs significant overhead in terms of resource requirements. Note that the additional hardware required by the stream interface is not depicted in Fig. 7. The DMA engine is configured through one of the PS master GP ports, and requires two different PS slave HP ports to directly access the memory where data to be transferred to/from the accelerator is stored.

To execute one branch of the first DNN layer, the stream interface performs (a) 96 memory-to-accelerator DMA operations to send the input images, with 96×3 pixels for each DMA operation, and (b) one memory-to-accelerator DMA operation to send $5 \times 5 \times 3$ kernel coefficients. Additionally, the stream interface needs 48 accelerator-to-memory DMA operations to retrieve the computed feature map, with 48 pixels for each DMA operation.

4.3.3. Local buffering

As mentioned previously, we incorporate local buffering of image pixels in the SFM accelerator to avoid redundant transmission of common data across different branches of the accelerator. This local buffering optimization is applied to both the mm-lite-interface- and stream-interface-based accelerator implementations.

For an accelerator configuration with a single SFM instance, the input image data is transferred to the accelerator only during execution of the first branch. After being transferred, this data is retained in a local buffer within the accelerator for reuse by the remaining 31 executions. For accelerator configurations that have multiple (parallel) SFM instances, the input image is also transferred only once to the accelerator. For these configurations, the image data is reused by the remaining executions of all of the SFM instances. Thus, our incorporation of local buffering optimization eliminates input image data transfers for all branches except the first one.

5. Results

In this section, we present experimental results to demonstrate the design and implementation methods provided by STMCM based on the detailed case study presented in Section 4. The main contribution of this section is to demonstrate that the proposed methodology facilitates efficient experimentation with alternative dataflow-based architectures, design optimization methods, and implementation trade-offs.

5.1. Embedded software implementation

In this section we present results of our experimentation using STMCM to explore alternative embedded software implementations. We focus specifically on the optimized application of loop tiling and buffer memory management.

Table 4
Memory requirements (in pixels) for the first two layers. In bracket in the last column: the percentage of memory requirement of DNN with shared FIFOs with respect to that of DNN with common FIFOs.

	FIFOs	Convolutional layer 1			Convolutional layer 2			Total
		Conv.	Add	Maxpool&ReLU	Conv.	Add	Maxpool&ReLU	
Common FIFOs	6,875,136	1,806,720	1,179,648	368,640	5,128,192	2,433,024	92,160	17883520
Shared FIFOs	2,525,184	921,984	0	0	2,768,896	0	0	6,216,064 (34.8)

5.1.1. Loop tiling

As introduced in Section 4.1.1, in the optimization of our LIDE-C implementation of the DNN application, we explored loop-tiled convolution actor designs with different tile sizes. Specifically, we measured the number of cache load misses and the cache load miss rates during execution of a convolution actor. The valid tile sizes for each convolution actor were those within the range of 1 to D, where D is the dimension of input images to the actor. For example, for the convolution actors in Layer 1, which process input images with size 96×96 pixels, we explored tile sizes within the range of 1–96.

Fig. 8 shows the number of cache load misses and cache load miss rate under different tile sizes for convolution actors with different input image dimensions (48×48, 96×96, 750×750, and 1500×1500). As we can see from the results, the cache load miss rates are very small for image dimensions $D \in \{48, 96, 750\}$. This indicates that the data can be fully stored or almost fully stored in the cache with any valid tile size.

For $D = 1500$, however, there is significant variation in the cache load miss rate across different tile sizes. The rate reaches its lowest value when the tile size is approximately 400. With careful setting of the tile size, loop tiling significantly reduces the cache miss rate for convolution actors that have relatively large image dimensions.

Additionally, we can see that there is a large average CPU cycle count for small tile sizes in all figures. We expect that this is due to the overhead caused by the additional `for` loops that are introduced by the loop tiling transformation.

In summary, based on our simulation analysis for small image dimensions (96×96 and 48×48), loop tiling does not help to reduce the cache miss rate on the target platform, and furthermore, it introduces overhead due to the additional for loops. Thus, loop tiling should not be applied to this DNN application for low image dimensions. However, our experiments also show that for larger image dimensions, loop tiling does help to improve the efficiency by reducing the cache load miss rate.

5.1.2. Buffer memory management

Fig. 9 shows the amount of memory required for data storage in each DNN layer. We report memory requirements in this section in terms of pixels. In our experiments, we used a 4-byte floating point data type for each pixel. Fig. 9 also shows the amount of data communication that is needed between adjacent layers, and the amount of memory that must be active simultaneously during the computation associated with each layer. The memory needed for input is calculated as $input\_image\_size \times number\_of\_input\_images$. The memory needed for execution of each layer is calculated as $input\_image\_size \times (number\_of\_input\_images + 1) \times number\_of\_output\_feature\_maps$.

As we can see from Fig. 9, the processing in Layer 2 requires the largest amount of active memory, and a minimum of 2,525,184 pixels must be allocated for buffer storage. The memory size can be optimized subject to this constraint through the application of shared FIFOs, which were introduced in Section 4.1.2. The buffer memory allocation that we propose for this DNN application based on shared FIFOs is illustrated in Fig. 10.

Table 4 summarizes the memory requirements for dataflow edges (FIFO buffers) and actors in the two convolutional layers, which require most of the memory among the five layers. These memory requirements are shown both with and without the use of shared FIFOs. As discussed in Section 4.1.2, actors operate on data from shared input FIFOs directly without copying data to its internal memory. Thus, convolution actors only need memory for its intermediate computation results. Add and Maxpool&ReLU actors do not require additional memory. The results presented in this table quantitatively demonstrate the utility of shared FIFOs for this application. In particular, the application of shared FIFOs reduces the memory requirements by 65%.

Table 5
Resource utilization. In parentheses: the percentage of utilization with respect to the resources available on the targeted FPGA.

Available	LUTs 53200	REGs 106400	BUFGs 32	BRAMs 140	DSPs 220
SFM	5188 (9.75)	3472 (3.26)	1 (3.1)	11 (7.9)	13 (5.9)
SFM_a	5430 (10.20)	3687 (3.47)	2 (6.3)	11 (7.9)	13 (5.9)
SFM_{CG}	5206 (9.79)	3496 (3.29)	5 (15.6)	11 (7.9)	13 (5.9)
SFM_{aCG}	5479 (10.30)	3704 (3.48)	6 (18.8)	11 (7.9)	13 (5.9)
SFM_h	5170 (9.72)	3472 (3.26)	1 (3.1)	11 (7.9)	13 (5.9)
SFM_{hCG}	5198 (9.77)	3480 (3.27)	3 (9.4)	11 (7.9)	13 (5.9)
SFM_{auto}	5230 (9.83)	3472 (3.26)	1 (3.1)	11 (7.9)	13 (5.9)

5.2. Hardware implementation

In this section, we investigate trade-offs among the variants of the SFM design that were introduced in Section 4.2.2. STMCM and the underlying LIDE-V approach allow one to perform such trade-off exploration, based on different combinations of high-level optimization techniques, in a systematic manner. In particular, STMCM allows the designer to focus on different strategies for instantiating, configuring, and coordinating different combinations of actor and buffer (edge) implementations, and eliminates the need for modification inside the actor and edge implementations. We exploited these advantages of STMCM when deriving the results presented in this section.

Table 5 depicts resource utilization data that is extracted from the post-place and route reports generated by the Xilinx Vivado tool using the targeted Zynq Z-7020 SoC. From the results in Table 5, we see that the different design variants all exhibit similar levels of resource cost. The asynchronous designs SFM_a and SFM_{aCG} incur the highest resource costs due to the additional logic required by the CDC FIFOs. The number of BUFGs varies significantly among the different designs, depending on the number of clock domains and the number of clock gated actors.

Each of the implemented designs has been simulated in order to generate a switching activity file, which has been back-annotated to Vivado Power Estimation to extract power consumption data. Since the designs have different execution times, the energy consumption levels do not vary in the same proportions as the power consumption levels. Table 6 summarizes the power consumption, execution time and energy consumption of the six alternative designs.

In these experiments, the clock frequencies of the synchronous designs and of Region 1 (CLK 1) in the asynchronous designs are all set to 100 MHz, which is the maximum achievable frequency for the targeted platform. For Region 2 (CLK 2) in the asynchronous designs, the frequency is set to 5 MHz. This setting of 5 MHz is derived from the hardware profiling data (see Table 3) as 1/20 of CLK 1. These clock frequencies are specified in Table 6 with the suffix _F, where F represents the frequency value in MHz.

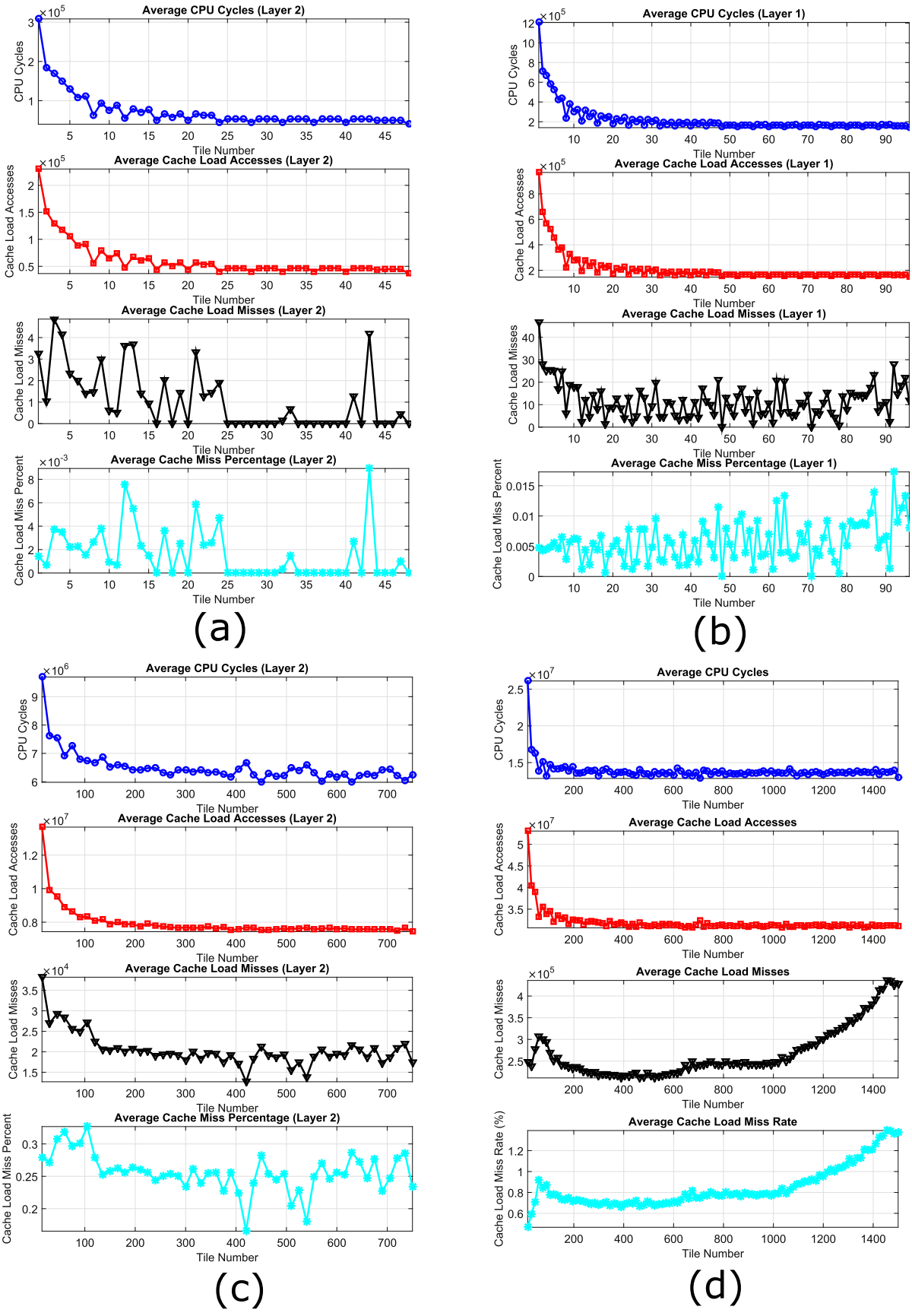


Fig. 8. Performance evaluation of convolution actors with different image dimensions: (a) 48×48, (b) 96×96, (c) 750×750, (d) 1500×1500.

FIFO	Memory (Pixels)
Layer 1 Input	96×96×3 =27648
Layer 1	96×96×4×32 =1179648
Layer 1 Output	48×48×32 =73728
Layer 2	48×48×33× 32 =2433024
Layer 2 Output	24×24×32 =18432
Layer 3	24×24×32 =18432
Layer 3 Output	100
Layer 4	0
Layer 4 Output	100
Layer 5	0
Layer 5 Output	4

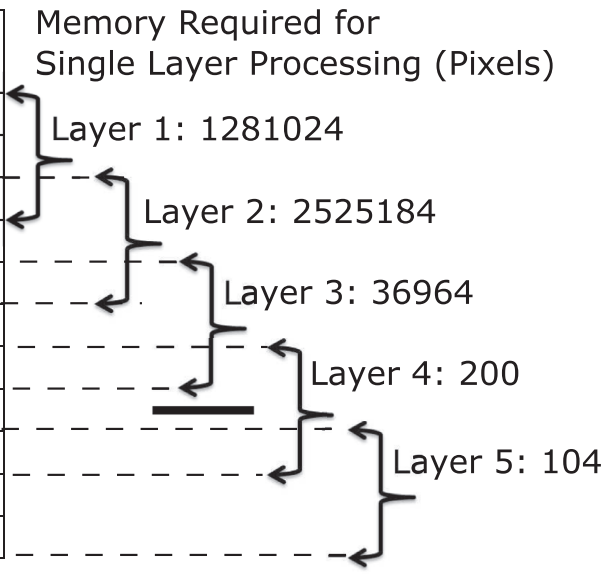


Fig. 9. Buffer memory and communication requirements in the DNN architecture.

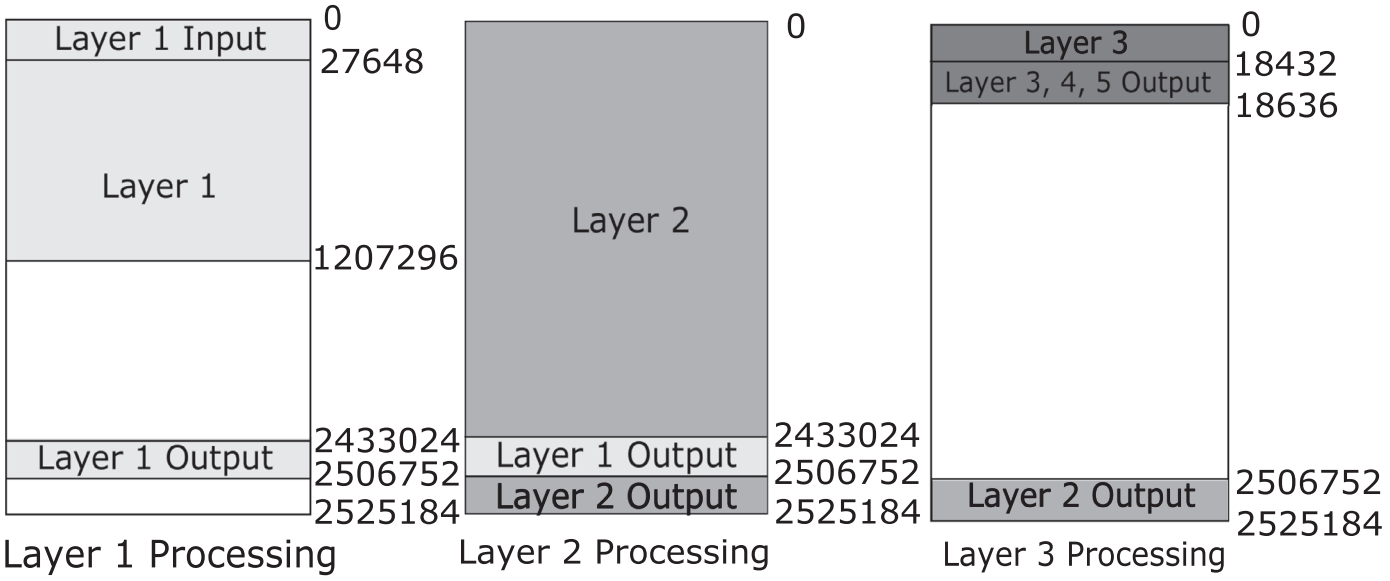


Fig. 10. Buffer memory allocation for the DNN application.

Table 6
Dynamic power consumption, execution time and energy consumption of the different SFM variants. In parentheses: the percentage difference with respect to the baseline *SFM*.

	Power [mW]	Time [ns]	Energy [μJ]
SFM	115	2,329,165	268
SFM_{a_5}	89 (-22.61)	2,407,300 (+3.354)	214 (-20.01)
SFM_{CG}	89 (-22.61)	2,329,245 (+0.003)	207 (-22.61)
SFM_{aCG_5}	88 (-23.48)	2,408,100 (+3.389)	212 (-20.89)
SFM_{h}	117 (+1.74)	2,329,155 (-0.000)	273 (+1.74)
SFM_{hCG}	105 (-8.70)	2,329,175 (+0.000)	244 (-8.70)
SFM_{auto}	113 (-1.74)	2,329,165 (+0.000)	263 (-1.74)

According to Table 6, the clock gated designs SFM_{CG} and SFM_{aCG_5} have the best capabilities for saving energy, reducing the total energy consumption by 22.61% and 20.89%, respectively. Design SFM_{aCG_5} saves less energy than SFM_{CG} since the former employs one more BUFG. Furthermore, in SFM_{aCG_5}, the actors in the slower domain (Region 2) are active for a relatively large portion of the execution time, and thus, they cannot be switched off for large proportions of time. In contrast, according to Table 3, the Deinterleave actor in Region 1 can be switched off for almost 90% of the total execution time.

The designs SFM_{aCG_5} and SFM_{a_5}, both of which employ two clock domains with CLK 1 at 100 MHz and CLK 2 at 5 MHz, have similar capabilities to save energy. The former design is slightly more energy efficient compared to the latter. The results for these two designs show that the energy saved by switching off the actors, when inactive, and also the saving of the unused logic in the CDC FIFOs counterbalance the energy overhead due to the additional circuitry.

As expected, SFM_h has a small amount of energy overhead due to the logic necessary to encapsulate Sum1, Sum2 and Maxpool&ReLU into the hierarchical actor. The design SFM_{hCG}, among the clock gated designs, is not as advantageous as the previously analyzed designs in terms of energy saving. This is because even though it employs only three BUFGs, the hierarchical actor is switched off only when none of the underly-

ing actors are working. This means that, for instance, while Sum1 is active, the actors Sum2 and Maxpool&ReLU will have an active clock even when the actors are in an idle state (so that they keep wasting energy). Finally SFM_{auto} is the design with the smallest energy saving, only 1.74% compared to *SFM*. Even considering the same optimization technique (clock gating), the level on which it is applied turns out to be fundamental: at a low level (single flip-flops in SFM_{auto}) only the dynamic power of a restricted number of gates can be saved. On the other hand, at a coarse-grain level (groups of dataflow actors in SFM_{CG}), it is possible to act also on the clock tree, which is highly effective for improving power saving.

5.3. Hardware/software co-design results

In this section, we investigate different hardware/software co-design configurations. As anticipated in Section 4.1, depending on the portion of the application that is accelerated in hardware and on the given requirements and constraints, different design choices regarding the hardware/software communication interface lead to different trade-offs between resource requirements and performance. For the SFM accelerator, we investigated several implementation and optimization solutions, exploring three key aspects: exploiting parallelism, communication interfaces and local buffering (see Section 4.3). In this section, by an *SFM accelerator*, we mean specifically a hardware accelerator.

Different software and hardware configurations that we explored in our co-design exploration are summarized as follows.

- *SW1* — The application runs in software on a single ARM core. This design can be viewed as a baseline design without any optimization or hardware acceleration. Comparisons between this baseline design and alternative designs are discussed in the remainder of this section.
- *SW2* — The application runs in software by using both of the ARM cores on the target platform.
- *HW1* — A single-branch SFM accelerator is employed to execute the first convolutional layer.
- *HW2* — An SFM accelerator with two parallel branches. In this configuration, a local buffer is shared between the branches.
- *HW4* — An SFM accelerator with four parallel branches. Again, a local buffer is shared among the branches.

For multicore software implementations and hardware implementations with multiple branches, the layer or layers that are executed in parallel (i.e., intra-layer parallelism is exploited) are indicated in parentheses. Similarly, hardware configurations are annoated with *-mm* or *-s* depending, respectively, on whether a memory-mapped AXI-lite communication interface is used, or a FIFO-based AXI-stream interface is used.

For example, SW2(L1, L2) represents a software-only implementation in which layer 1 and layer 2 are executed in parallel. As another example, SW2(L2)/HW2(L1)-mm represents a hardware/software implementation based on configurations SW2 and HW2; in this implementation, layer 2 is executed across multiple cores, layer 1 is parallelized in hardware with 2 parallel branches, and AXI-lite is used as the communication interface.

Note that the SFM accelerators are able to execute only the first convolutional layer. Thus, in all of the DNN system implementations, the accelerators are coupled with one of the software configurations.

5.3.1. Resource costs of accelerator implementations

Table 7 depicts the resource occupancy in the targeted Zynq Z-7020 device for the different SFM accelerator implementations that we experimented with. As expected, a higher level of parallelism (going from HW1-mm to HW4-mm) requires more resources, and our experiments here help to quantify the associated trends. For example, fine-grained and computation-related resources (LUTs, REGs and DSPs) increase linearly with the number of parallel branches placed in the accelerator (about +100% with one more branch and about +300% with three more branches), while coarse-grained memory resources (BRAMs) exhibit a gentler slope. We expect that this gentler slope results because the primary BRAM-demanding module, the local buffer, is shared across parallel branches.

Table 7
Resource occupancy for different SFM accelerator implementations. In parentheses: the percentage of utilization with respect to the resources available on the targeted FPGA. The bottom part of the table depicts the percentage of variation with respect to HW1-mm.

Available	LUTs 53200	REGs 106400	BRAMs 140	DSPs 220
HW1-mm	5395(10.14)	4668(4.39)	43 (30.71)	13 (5.91)
HW2-mm	10890 (20.47)	8197 (7.70)	54 (38.57)	26 (11.82)
HW4-mm	21474 (40.36)	16331(15.35)	76 (54.29)	52 (23.64)
HW2-mm	+101.85	+75.60	+25.58	+100.00
HW4-mm	+298.04	+249.85	+76.74	+300.00

Table 8
Performance of different co-design solutions. The top part of the table depicts execution time in milliseconds (ms). The bottom part depicts the percentage of execution time variation for each configuration with respect to SW1.

	input	Layer			Prediction
		1	2	3:5	
SW1	118.9	640.3	1594.7	34.4	2388.2
SW2(L1)	118.7	368.3	1609.8	34.0	1639.5
SW2(L1,L2)	117.4	354.7	842.1	33.8	1348.0
SW2(L2)/HW1(L1)-mm	118.9	118.5	856.7	35.4	1129.5
SW2(L2)/HW2(L1)-mm	118.4	74.6	866.5	35.1	1094.5
SW2(L2)/HW4(L1)-mm	117.9	54.5	859.0	35.4	1066.8
SW2(L1)	-0.13	-42.48	+0.95	+1.12	-10.75
SW2(L1,L2)	-1.22	-44.61	-47.19	-1.72	-43.56
SW2(L2)/HW1(L1)-mm	-0.00	-81.49	-46.28	+2.89	-52.71
SW2(L2)/HW2(L1)-mm	-0.40	-88.36	-45.66	+2.09	-54.17
SW2(L2)/HW4(L1)-mm	-0.81	-91.48	-46.13	+2.85	-55.33

The results above indicate that when the DNN architecture is made deeper (i.e., as the number of convolutional layers is increased), the biggest restriction will be the hardware resource limitations. Usually, as a DNN is made deeper, more parallel branches are needed to complete the computation without compromising the processing speed and more memory resources are needed to store the intermediate feature maps. However, deeper networks do not necessarily imply more computational complexity. For example, the well-known ResNet101, which has 101 layers, needs less computation than the 16-layer VGG16 because the VGG layers are significantly larger [39,40].

5.3.2. Comparison of co-design solutions

Table 8 presents performance results for different software-only and hardware/software solutions that we investigated using STMCM. In particular, the table reports the execution time in terms of milliseconds (ms) for different execution phases: reading the input file (column input), computing the first and the second layers, and computing the deep layers (Layers 3, 4 and 5). The table also reports the execution time of the overall application (prediction) for different degrees of software and hardware parallelism.

The reference time is given by the execution of the entire DNN application on a single ARM core (SW1), which is capable of completing the prediction in about 2.4 seconds. From this reference configuration, it is also possible to appreciate the computational load of the different application phases. The heaviest part is Layer 2, which is responsible for more than 65% of the overall execution time, while most of the remaining load is attributable to Layer 1 (around 25%), and to reading of the input file (about 5%). For this reason, software parallelization has been evaluated only on Layer 1 (SW2(L1)), and on both Layers 1 and 2 together (SW2(L1,L2)).

Table 9
Differences in resource costs between communication interfaces when applied to HW1. In parentheses: percentage of utilization with respect to the resources available on the targeted FPGA. The bottom part of the table depicts the percentage utilization variation with respect to HW1-mm.

Available	LUTs 53200	REGs 106400	BRAMs 140	DSPs 220
HW1-mm	5395(10.14)	4668(4.39)	43 (30.71)	13 (5.91)
(1) HW1-s	5784 (10.87)	4357 (4.09)	43 (30.71)	13 (5.91)
(2) FIFOs (stream)	212 (0.40)	242 (0.23)	10 (7.14)	0 (0.00)
(3) DMA (stream)	1490 (2.80)	1881 (1.77)	3 (2.14)	0 (0.00)
(1)+(2)+(3)	7486 (14.07)	6480 (6.09)	56 (40.00)	13 (5.91)
HW1-s	+7.21	-6.66	+0.00	+0.00
(1)+(2)+(3)	+38.75	+38.81	+53.49	+0.00%

The execution time needed by each of the major execution phases is almost halved when two cores are adopted. A precise 50% reduction is not reached because of the software overhead necessary to manage multitasking. With software parallelization only, the overall execution time is reduced to 1.13 seconds, about 44% less than the SW1 configuration. Hardware acceleration and related parallelization are only applied to the first convolutional layer, while only software parallelization is applied to Layer 2. If we consider only the execution time of layer 1, then SW2(L2)/HW1(L1) reduces execution time by more than 80% compared to SW1, and more than 65% compared to SW2(L1,L2).

If multiple branches of Layer 1 are processed in parallel, the hardware accelerator achieves further performance benefits — a time saving up to 88% for a 2-branch configuration (SW2(L2)/HW2(L1)-mm), and up to 91% for a 4-branch configuration (SW2(L2)/HW4(L1)-mm). These performance improvements are with respect to SW1. Note that the speed-up obtained by doubling the number of branches (going from 1 to 2 and from 2 to 4) is less than 2 in either case (1.6 from 1 to 2 and 1.4 from 2 to 4). This is due to the software overhead related to managing multiple branches. Due to the limited computational load of Layer 1, the benefits of hardware acceleration and parallelization on the overall system are somewhat limited. The best solution, SW2(L2)/HW4(L1)-mm, requires 1.07 seconds to perform the whole application, 55% less than a full software execution on a single core (SW1) and 21% less than a full software execution on two cores (SW2(L1,L2)).

Another aspect that has been studied in our co-design experiments is the interfacing between system components. As discussed in Section 4.3.2, the adopted communication interface between software and hardware portions of a design can have a significant impact on overall system performance. In our co-design experiments, we have applied two very interfaces — mm-lite and stream, which are discussed in Section 4.3.2.

Table 9 helps to understand differences between the resource costs of these two interfaces. The first row of this table shows resource availability on the target platform. The second and third rows show resource costs for the HW1-mm accelerator, and HW1-s accelerator. The fourth and fifth rows show resource costs for FIFO and DMA modules (external to the accelerator) that are necessary for the stream interface. The sixth row shows total resource costs induced by use of the stream interface (the sums of the costs in the preceding three rows). The last two rows of the table represent percentage increases in resource costs relative to the HW1-mm accelerator.

From Table 9, we see that the HW1-mm and HW1-s accelerators alone require approximately the same amount of resources: HW1-s requires 7.21% more LUTs and 6.66% less REGs compared to HW1-mm. However, when the overhead due to the DMA and FIFO modules necessary for AXI-stream communication is considered, significantly more resources are required when the stream interface is used: about 38% more LUTs and REGs are required by the overall stream design ((1)+(2)+(3)), while over 50% more BRAM cost is incurred.

To make the stream interface a useful option in our system design, its significant increase in resource costs should be accompanied by tangible advantages in execution performance. Table 10 shows results pertaining to the impact of the communication interface on execution time. In order to better expose the effects of the selected communication interface, details on data transfers (input, convolution coefficients and outputs) between the hardware (accelerator) and software subsystems is reported. For the HW1-s design, two different sets of results are reported depending on whether program data is directly accessible by the DMA engine. For one set, the program data is located in a memory that is not directly accessible by the DMA. This scenario corresponds to the design that we have implemented. It requires an additional copy of the program data in a memory that is accessed directly by the DMA. For the other set, the program data is located in a memory that is directly accessible by the DMA. This set is indicated in Table 10 using the annotation *HW1-s dir*. We have not implemented HW1-s dir; instead, we have estimated the corresponding results to gain some idea about the maximum achievable performance. Details on the estimation approach are omitted for brevity.

The results in Table 10 demonstrate the utility of the resource-hungry HW1-s design, and quantify its clear ability to outperform HW1-mm. In particular, the input data and transmission of convolution coefficients are respectively about 84% and 67% faster when the AXI-stream protocol is adopted. This leads to an estimated time saving of up to 92% and 80%, respectively, when the DMA has direct access to the program data (HW1-s dir).

On the other hand, the output data transmission time is the same among all of the reported configurations. We expect that this is because the outputs are produced in a row-by-row fashion (48 data units at a time), and the timing of output production is determined by the computation latency, which is greater than the communication latency for all of the interfacing configurations. However, looking at the total Layer 1 and DNN application execution times, we see that the advantages of adopting the stream interface are no longer visible. Indeed, for the considered SFM accelerator, the input data is transmitted only during the first branch execution due to our use of local buffering. Additionally, even though the coefficients are transmitted for each branch, their transmission requires a relatively small amount of time.

These results involving communication interface selection illustrate the importance of comprehensive system-level evaluation of alternative design options, which is one of the key parts of the design process that is facilitated by STMCM.

Table 10
Results pertaining to the impact of the communication interface on execution time. The top part of the table depicts the execution time of the different DNN application steps. The bottom part depicts the execution time variation of each configuration with respect to HW1-mm.

	File input [ms]	Layer 1				Layer 2 [ms]	Layers 3, 4, 5 [ms]	Prediction [ms]
		input tx [μs]	coeffs tx [μs]	output tx [μs]	total [ms]			
HW1-mm	118.9	5222	15	2339	118.5	856.7	35.4	1129.5
HW1-s	117.5	854	5	2333	114.6	864.3	34.9	1131.3
HW1-s dir	118.3	418	3	2333	114.1	869.5	35.0	1137.0
HW1-s	-1.17	-83.65	-66.67	-0.26	-3.27	+0.87	-1.39	+0.16
HW1-s dir	-0.49	-92.00	-80.00	-0.26	-3.69	+1.49	-1.22	+0.66

6. Conclusion

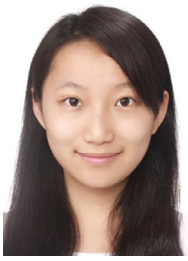
In this paper, we have introduced a design methodology, called the STMC Methodology or STMCM, and an integrated set of tools and libraries that support the application of this methodology. STMCM is developed to assist designers of signal processing systems in exploring complex design alternatives that span multiple implementation scales, platform types, and dataflow modeling techniques. We have demonstrated the capabilities of STMCM through a detailed case study involving a deep neural network (DNN) for vehicle classification. The demonstration encompasses dataflow-based application modeling, profiling, embedded software optimization, hardware accelerator design, hardware/software co-design, and hardware/software interface design, all in the context of mapping the given DNN into an efficient implementation on a resource-constrained, system-on-chip platform. Through this case study, it is shown how STMCM provides a unified, model-based framework for conducting comprehensive empirical evaluations of diverse hardware/software design alternatives. Through its application of lightweight dataflow techniques, STMCM is complementary to dataflow tools that emphasize specialized design flows and high degrees of automation. Useful directions for future work involve applying STMCM in novel ways that exploit these complementary relationships. Additionally, we believe that an automatic code generator producing the corresponding hardware/software co-design code given the hyperparameters such as the number of layers and/or number of feature maps would be very impactful. Implementation criteria could be integrated such that the generated network can be optimized based on different constraints and objectives.

Acknowledgments

This research was supported in part by Business Finland (FiDiPro project StreamPro1846/31/2014); US National Science Foundation (CNS1514425); H2020 Program CERBERO (# 732105), ALOHA (# 780788), FitOptiVis (# 783162) Projects; and the Sardinian Regional Project PROSSIMO (POR FESR 2014/20-ASSE I).

References

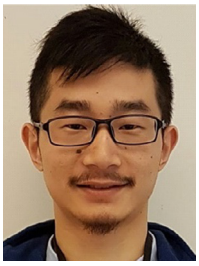
[1] S.S. Bhattacharyya, E. Deprettere, R. Leupers, J. Takala (Eds.), Handbook of signal processing systems, Springer, 2013.
[2] S. Ha, J. Teich (Eds.), Handbook of hardware/software codesign, Springer, 2017.
[3] C. Shen, W. Plishker, H. Wu, S.S. Bhattacharyya, A lightweight dataflow approach for design and implementation of SDR systems, in: Proceedings of the Wireless Innovation Conference and Product Exposition, 2010, pp. 640–645.
[4] W. Plishker, N. Sane, M. Kiemb, S.S. Bhattacharyya, Heterogeneous design in functional DIF, in: Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation, Samos, Greece, 2008, pp. 157–166.
[5] J.T. Buck, E.A. Lee, Scheduling dynamic dataflow graphs using the token flow model, in: Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, 1993.
[6] G. Bilsen, M. Engels, R. Lauwereins, J.A. Peperstraete, Cyclo-static dataflow, IEEE Trans. Signal Process. 44 (2) (1996) 397–408.
[7] E.A. Lee, D.G. Messerschmitt, Synchronous dataflow, Proc. IEEE 75 (9) (1987) 1235–1245.
[8] J. Eker, J.W. Janneck, Dataflow programming in CAL — balancing expressiveness, analyzability, and implementability, in: Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers, 2012, pp. 1120–1124.
[9] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, M. Raulet, Orcc: multimedia development made easy, in: Proceedings of the ACM International Conference on Multimedia, 2013, pp. 863–866.
[10] J. Sérot, F. Berry, S. Ahmed, CAPH: A Language for Implementing Stream-processing Applications on FPGAs, in: P. Athanas, D. Pnevmatikatos, N. Sklavos (Eds.), Embedded Systems Design with FPGAs, Springer, 2013.
[11] J. Mcallister, R. Woods, R. Walke, D. Reilly, Multidimensional DSP core synthesis for FPGA, J VLSI Signal Process Syst Signal Image Video Technol 43 (2–3) (2006).
[12] M. Pelcat, P. Menuet, S. Aridhi, J.-F. Nezan, Scalable compile-time scheduler for multi-core architectures, in: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2009, pp. 1552–1555.
[13] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, J. Teich, A systemc-based design methodology for digital signal processing systems, EURASIP J. Embed. Syst. 2007 (2007) 22. Article ID 47580.
[14] S. Lin, Y. Liu, W. Plishker, S.S. Bhattacharyya, A design framework for mapping vectorized synchronous dataflow graphs onto CPU–GPU platforms, in: Proceedings of the International Workshop on Software and Compilers for Embedded Systems, Sankt Goar, Germany, 2016, pp. 20–29.
[15] S. Casale-Brunet, M. Wiszniewska, E. Bezati, M. Mattavelli, J.W. Janneck, M. Canale, TURNUS: An open-source design space exploration framework for dynamic stream programs, in: Proceedings of the Conference on Design and Architectures for Signal and Image Processing, 2014, pp. 1–2.
[16] C. Sau, et al., Automated design flow for multi-functional dataflow-based platforms, J. Signal Process. Syst. (2015) 1–23. doi: 10.1007/s11265-015-1026-0.
[17] F. Palumbo, T. Fanni, C. Sau, P. Meloni, Power-awarness in coarse-grained reconfigurable multi-functional architectures: a dataflow based strategy, J. Signal Process. Syst. 87 (1) (2017) 81–106.
[18] T. Fanni, C. Sau, P. Meloni, L. Raffo, F. Palumbo, Power and clock gating modelling in coarse grained reconfigurable systems, in: Proceedings of the ACM International Conference on Computing Frontiers, 2016, pp. 384–391.
[19] S.C. Brunet, E. Bezati, C. Alberti, M. Mattavelli, E. Amaldi, J.W. Janneck, Multi-clock domain optimization for reconfigurable architectures in high-level dataflow applications, in: Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers, 2013, pp. 1796–1800.
[20] E. Bezati, S.C. Brunet, M. Mattavelli, J.W. Janneck, Coarse grain clock gating of streaming applications in programmable logic implementations, in: Proceedings of the Electronic System Level Synthesis Conference, 2014, pp. 1–6.
[21] T. Fanni, L. Li, T. Viitanen, C. Sau, R. Xie, F. Palumbo, L. Raffo, H. Huttunen, J. Takala, S.S. Bhattacharyya, Hardware design methodology using lightweight dataflow and its integration with low power techniques, J. Syst. Archit. 78 (2017) 15–29.
[22] S. Lin, Y. Liu, K. Lee, L. Li, W. Plishker, S.S. Bhattacharyya, The DSPCAD Framework for Modeling and Synthesis of Signal Processing Systems, in: S. Ha, J. Teich (Eds.), Handbook of Hardware/Software Codesign, Springer, 2017, pp. 1–35.
[23] L. Li, A. Ghazi, J. Boutellier, L. Anttila, M. Valkama, S.S. Bhattacharyya, Evolutionary multiobjective optimization for digital predistortion architectures, in: Proceedings of the International Conference on Cognitive Radio Oriented Wireless Networks, 2016, pp. 498–510.
[24] L. Li, A. Sapio, J. Wu, Y. Liu, K. Lee, M. Wolf, S.S. Bhattacharyya, Design and implementation of adaptive signal processing systems using Markov decision processes, in: Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors, Seattle, Washington, 2017, pp. 170–175.
[25] B. Bhattacharya, S.S. Bhattacharyya, Parameterized dataflow modeling for DSP systems, IEEE Trans. Signal Process. 49 (10) (2001) 2408–2421.
[26] S. Lin, J. Wu, S.S. Bhattacharyya, Memory-constrained vectorization and scheduling of dataflow graphs for hybrid CPU-GPU platforms, ACM Trans. Embedded Comput. Syst. 17 (2) (2018) 50:1–50:25.
[27] R. Xie, H. Huttunen, S. Lin, S.S. Bhattacharyya, J. Takala, Resource-constrained implementation and optimization of a deep neural network for vehicle classification, in: Proceedings of the European Signal Processing Conference, Budapest, Hungary, 2016, pp. 1862–1866.
[28] A.H. Ghamarian, M.C.W. Geilen, S. Stuijk, T. Basten, A.J.M. Moonen, M.J.G. Bekooij, B.D. Theelen, M.R. Mousavi, Throughput analysis of synchronous data flow graphs, in: Proceedings of the International Conference on Application of Concurrency to System Design, 2006.
[29] L. Li, T. Fanni, T. Viitanen, R. Xie, F. Palumbo, L. Raffo, H. Huttunen, J. Takala, S.S. Bhattacharyya, Low power design methodology for signal processing systems using lightweight dataflow techniques, in: Proceedings of the Conference on Design and Architectures for Signal and Image Processing, Rennes, France, 2016, pp. 81–88.
[30] H. Huttunen, F. Yancheshmeh, K. Chen, Car type recognition with deep neural networks, ArXiv e-prints (2016) To appear in proceedings of IEEE Intelligent Vehicles Symposium 2016. ArXiv:1602.07125v2.
[31] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, C.L. Zitnick, Microsoft COCO: Common objects in context, in: Proceedings of the European Conference on Computer Vision, 2014, pp. 740–755.
[32] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, ImageNet: A large-scale hierarchical image database, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2009, pp. 248–255.
[33] T. Chen, et al., DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning, in: Symposium on Architectural Support for Programming Languages and Operating Systems, 2014, pp. 269–284.
[34] P.K. Murthy, S.S. Bhattacharyya, Shared buffer implementations of signal processing systems using lifetime analysis techniques, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 20 (2) (2001) 177–198.
[35] H. Oh, S. Ha, Memory-optimized software synthesis from dataflow program graphs with large size data samples, EURASIP J. Appl. Signal Process. 2003 (6) (2003).
[36] K. Desnos, M. Pelcat, J.-F. Nezan, S. Aridhi, Buffer merging technique for minimizing memory footprints of synchronous dataflow specifications, in: Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, 2015, pp. 1111–1115.
[37] H.-J. Koch, The Userspace I/O HOWTO, Linutronix, 2006.
[38] J. Silva, V. Sklyarov, I. Skliarova, Comparison of on-chip communications in zynq-7000 all programmable systems-on-chip, IEEE Embed. Syst. Lett. 7 (1) (2015) 31–34.
[39] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016.
[40] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, 2014 arXiv:1409.1556.



Lin Li is a Ph.D. student in DSPCAD research group in the Department of Electrical and Computer Engineering at the University of Maryland, College Park, USA. Lin Li received the Bachelor's degree in Electrical Engineering and Automation from Fudan University, Shanghai, China. Her research has focused on dataflow-based framework for design, implementation, and optimization of signal processing systems including wireless communication systems and machine learning systems.

Carlo Sau is currently Assistant Professor at the University of Cagliari. He received his degree in Electronic Engineering in 2012 at the University of Cagliari and his PhD in 2016 in the same university. Since 2012, he is working on automated methodologies for dataflow-based reconfigurable platforms generation. His main research focus is related to reconfigurable system design and development of code generation tools for advanced reconfigurable hardware architectures.

Tiziana Fanni is a Ph.D student at the Department of Electrical and Electronic Engineering of the University of Cagliari. She received her degree in Electronic Engineering in 2014 at the University of Cagliari. In June 2014 she started a 1 year research grant related to power saving methodologies in dataflow- based reconfigurable platforms. Her main research focus is related to reconfigurable systems design and development of code generation tools for low power reconfigurable hardware architectures.



Jingui Li received his Bachelor's degree (with honor) in Electronic Science and Technology from Hefei University of Technology, China. He received Master's degree at Tampere University of Technology (TUT). He has worked on a project which aims to apply dataflow techniques in hardware design using Verilog HDL.

Timo Viitanen received his M.Sc. degree in Embedded Systems from Tampere University of Technology (TUT) in 2013, and is now a graduate student at the Department of Pervasive Computing in TUT. He is the recipient of a TUT graduate school position and has been awarded the Nokia Scholarship in 2014. His research interests include computer architecture and computer graphics.

François Christophe works as University Researcher at the Department of Computer Science of University of Helsinki. His research interests include computational models for the simulation of complex systems, semi-formal modelling and Artificial Intelligence. In 2007, after receiving a double Master degree in Computer and Software Engineering from Brest National Engineering School (France) and in Artificial Intelligence and Image from University of Rennes I, he decided to pursue doctoral studies in Systems Engineering at Helsinki University of Technology, Finland. He received his Ph.D. degrees from Aalto University (Finland) and Nantes Centrale Engineering School (France) in 2012. He worked as post-doctoral researcher at Aalto University from 2012 to 2014 and in the Department of Pervasive Computing at Tampere University of Technology from 2014 to 2017.

Francesca Palumbo is Assistant Professor at the University of Sassari, within the Information Engineering unit of the Department of Political Sciences, Communication Sciences and Information Engineering. She received her summa cum laude "Laurea Degree" in Electronic Engineering in 2005 at the University of Cagliari, then attended the Master Advanced in Embedded System Design in 2006 at the Advanced Learning and Research Institute of the University of Lugano before starting her Ph.D. in Electronic and Computer Engineering at the University of Cagliari. Her research focus is related to reconfigurable systems and to code generation tools and design automation strategies for advanced reconfigurable hardware architectures. For her studies in the fields of dataflow-based programming and hardware customization, she received two Best Paper Awards at the Conference on Design and Architectures for Signal and Image Processing, respectively in 2011 and in 2015, with the works entitled "The Multi-Dataflow Composer tool: A runtime reconfigurable HDL platform composer" and "MPSoCs for real-time neural signal decoding: A low-power ASIP-based implementation". Dr. Palumbo serves in several different Technical Committee of international conferences and she is a permanent Steering Committee Member of the ACM Conference on Computing Frontiers and Associate Editor of the Springer Journal of Signal Processing Systems. At the moment, Dr. Palumbo is the scientific coordinator of the CERBERO H2020 European Project on Smart Cyber Physical System Design and the Scientific Director of Summer School entitled "Designing Cyber-Physical Systems - From concepts to implementation" that has been hold in Alghero in September 2017 and will be organized again in 2018.

Luigi Raffo is full professor of Electronics at the Department of Electrical and Electronic Engineering - University of Agliari (ITALY). He received the "Laurea degree" in Electronic Engineering at University of Genoa (ITALY) in 1989, the PhD degree in Electronics and Computer Science at the same university in 1994. In 1994 he joined the Department of Electrical and Electronic Engineering of University of Cagliari (ITALY) as assistant professor, in 1998 as associate professor and from 2006 as full professor of electronics. He teaches courses on system/digital and analog electronic design and processor architectures for the Courses of studies in Electronic and Biomedical Engineering. He was coordinator of the project EU IST- FET - IST-2001-39266 - BEST and he was unit coordinator of the project EU IST-FET - SHAPES - Scalable Software Hardware Architecture Platform for Embedded Systems. He has been local coordinator of industrial projects in the field (among others: ST-Microelectronics - Extension of ST200 architecture for ARM binary compatibility, ST-Microelectronics - Network on chip). He is responsible for cooperation programs in the field of embedded systems with several other European Universities. He was coordinator of the MADNESS EU Project (FP7/2007-2013) and local coordinator in the ASAM (ARTEMIS-JU) and ALBA projects (national founded project) and RPCT (regional founded project).

Heikki Huttunen is an associate professor at Tampere University of Technology, Finland. He holds M.Sc. and Ph.D degrees from University of Tampere and Tampere University of Technology in 1995 and 1999, respectively. He leads the Machine Learning Group and his research interests are in machine learning deployment, to bring real time machine learning into embedded and mobile devices.

Jarmo Takala received his M.Sc. (hons) degree in Electrical Engineering and Dr.Tech. degree in Information Technology from Tampere University of Technology, Tampere, Finland (TUT) in 1987 and 1999, respectively. From 1992 to 1995, he was a Research Scientist at VTT-Automation, Tampere, Finland. Between 1995 and 1996, he was a Senior Research Engineer at Nokia Research Center, Tampere, Finland. From 1996 to 1999, he was a Researcher at TUT. Since 2000, he has been Professor in Computer Engineering at TUT and currently Dean of the Faculty of Computing and Electrical Engineering of TUT. Dr. Takala is Co-Editor-in-Chief for Springer Journal on Signal Processing Systems. During 2007–2011 he was Associate Editor and Area Editor for IEEE Transactions on Signal Processing and in 2012–2013 he was the Chair of IEEE Signal Processing Society's Design and Implementation of Signal Processing Systems Technical Committee. His research interests include circuit techniques, parallel architectures, and design methodologies for digital signal processing systems.

Shuvra S. Bhattacharyya is a Professor in the Department of Electrical and Computer Engineering at the University of Maryland, College Park. He holds a joint appointment in the University of Maryland Institute for Advanced Computer Studies (UMIACS). He is also a part-time visiting professor in the Department of Pervasive Computing at the Tampere University of Technology, Finland, as part of the Finland Distinguished Professor Programme (FiDiPro). His research interests include signal processing, embedded systems, electronic design automation, wireless communication, and wireless sensor networks. He received the B.S. degree from the University of Wisconsin at Madison, and the Ph.D. degree from the University of California at Berkeley. He has held industrial positions as a Researcher at the Hitachi America Semiconductor Research Laboratory (San Jose, California), and Compiler Developer at Kuck & Associates (Champaign, Illinois). He has held a visiting research position at the US Air Force Research Laboratory (Rome, New York). He has been a Nokia Distinguished Lecturer (Finland) and Fulbright Specialist (Austria and Germany). He has received the NSF Career Award (USA). He is a Fellow of the IEEE.