

# PRUNE: Dynamic and Decidable Dataflow for Signal Processing on Heterogeneous Platforms

Jani Boutellier, *Member, IEEE*, Jiahao Wu, *Member, IEEE*, Heikki Huttunen, *Member, IEEE*, Shuvra S. Bhattacharyya, *Fellow, IEEE*

**Abstract**—The majority of contemporary mobile devices and personal computers are based on heterogeneous computing platforms that consist of a number of CPU cores and one or more Graphics Processing Units (GPUs). Despite the high volume of these devices, there are few existing programming frameworks that target full and simultaneous utilization of all CPU and GPU devices of the platform.

This article presents a dataflow-flavored Model of Computation (MoC) that has been developed for deploying signal processing applications to heterogeneous platforms. The presented MoC is dynamic and allows describing applications with data dependent run-time behavior. On top of the MoC, formal design rules are presented that enable application descriptions to be simultaneously dynamic and decidable. Decidability guarantees compile-time application analyzability for deadlock freedom and bounded memory.

The presented MoC and the design rules are realized in a novel Open Source programming environment “PRUNE” and demonstrated with representative application examples from the domains of image processing, computer vision and wireless communications. Experimental results show that the proposed approach outperforms the state-of-the-art in analyzability, flexibility and performance.

**Index Terms**—Dataflow computing, design automation, signal processing, parallel processing

## I. INTRODUCTION

ADVANCES in signal processing have enabled new technologies that greatly affect our everyday lives. Progress in wireless communications, video coding, and recently, computer vision, has provided us previously impossible applications. However, simultaneously, the signal processing behind MIMO radios, H.265 video coding and Convolutional Neural Networks has reached considerable computational complexity, even though these applications are often executed on performance-constrained mobile devices.

Enabling real-time performance for such signal processing algorithms often means resorting to computation acceleration by fixed-function ASICs (Application Specific Integrated Circuit) or by programmable accelerators such as GPUs. However, due to strict design time requirements of the industry,

programmable accelerators are becoming increasingly popular compared to ASICs.

The efficiency of programmable computation accelerators is based on the fact that their architectures have been tuned to accelerate specific classes of algorithms. Unfortunately, this means that accelerators are only suitable for executing certain parts of application code, leaving the rest of the execution burden to the general purpose CPU cores of the computation platform. Hence, both general purpose CPU cores and accelerators have a significant role in the total application performance. To this extent, tapping the full performance potential of a heterogeneous computing platform requires a programming approach that can efficiently and simultaneously use all available CPU cores and accelerators.

The mainstream approach for programming the most popular compute accelerators of today, GPUs, involves the C-like languages CUDA from NVidia and OpenCL by Khronos. Whereas the former is intended for offloading computations to NVidia GPU devices, the latter provides a common Application Programming Interface (API) for both CPU cores and GPUs. Unfortunately, the OpenCL API operates on a low level and requires the programmer to take care of synchronization and memory transfers between devices, which is tedious and requires specialized expertise.

As it has been observed in many previous works [1], [2], [3], [4], dataflow Models of Computation provide a remarkably suitable abstraction for signal processing algorithms. Previous work [5] has also shown that programming frameworks based on the dataflow abstraction allow the application programmer to concentrate on developing the application, as concurrency and memory related low-level tasks are managed by the dataflow abstraction and by the programming framework.

This article describes a dataflow-flavored Model of Computation that

- captures the functionality of *data dependent* signal processing algorithms,
- enables design time analysis for deadlock freedom and bounded memory use (decidability) through formal design rules, and
- provides a basis for efficient concurrent computation on heterogeneous platforms.

The MoC presented in this article has been published [6] recently, and in this article the MoC is complemented with design rules that enable decidability analysis.

Based on the MoC, the article describes a novel Linux-

arXiv:1802.06625v1 [cs.DC] 19 Feb 2018

J. Boutellier is with the Laboratory of Pervasive Computing, Tampere University of Technology, Finland, e-mail: jani.boutellier@tut.fi.

J. Wu is with the Department of Electrical and Computer Engineering, University of Maryland, USA, e-mail: jiahao@terpmail.umd.edu.

H. Huttunen is with the Laboratory of Signal Processing, Tampere University of Technology, Finland, e-mail: heikki.huttunen@tut.fi.

S. S. Bhattacharyya is with the Department of Electrical and Computer Engineering, University of Maryland, USA, and the Laboratory of Pervasive Computing, Tampere University of Technology, Finland, email: ssb@umd.edu.

Manuscript received — xx, 2017; revised — xx, 2017.

based Open Source<sup>1</sup> programming framework PRUNE (PSM Runtime Environment) targeted for high-performance signal processing applications. PRUNE

- 1) implements application consistency analysis,
- 2) provides an efficient runtime memory- and concurrency management framework for heterogeneous platforms,
- 3) presents a compile-time translator that allows importing programs from previous similar run-time frameworks.

Out of these, items 1) and 3) are novel compared to [6].

To the best knowledge of the authors, the PRUNE dataflow framework is the first to simultaneously provide a) flexibility for describing signal processing applications with data-dependent token rates, b) a decidable Model of Computation, and c) experimental results that demonstrate high performance.

## II. BACKGROUND

In the dataflow abstraction [7], an application is described as a *graph* that consists of *actors* (nodes) and communication *channels* (edges). Actors perform computations on data that is quantized into *tokens*. Actors acquire tokens from their input *ports* and produce computation results to their output ports. Token communication between actors is handled by order-preserving FIFO (First-In-First-Out) channels that are attached to actor ports. A dataflow actor performs a computation by *firing*, which can include consuming tokens from input ports, and producing tokens to the actor output ports. A central feature of the dataflow abstraction is that computations are triggered by the availability of data, in contrast to, for example, time-triggered abstractions [8].

In the literature, a wide variety of dataflow Models of Computation (MoC) has been presented. One of the most important factors that differentiates a dataflow MoC from another concerns the token communication rates (*dataflow rates*) — that is, the rates at which an actor reads from or writes to the channels that are connected to it. In this sense, the most restricted dataflow MoC is *homogeneous synchronous dataflow* (HSDF) [7], where for each actor the token rate of each input port and each output port must be exactly one. *Synchronous dataflow* (SDF) [7] is more expressive as it allows token rates larger than one, as is *cyclo-static dataflow* (CSDF) [2], which goes beyond SDF by allowing tokens rates to vary in repetitive cycles.

The aforementioned MoCs (HSDF, SDF, CSDF) are restricted in the sense that they disallow *data dependent* changes to the token rates, which is a required feature as, for example, video decoders [9] and Software Defined Radio applications [10] introduce behavior that cannot be captured by static token rates. To achieve this, *dynamic dataflow* MoCs are required. Examples of dynamic dataflow MoCs are *Boolean dataflow* (BDF) [11], *enable-invoke dataflow* (EIDF) [1] and *dataflow process networks* (DPN) [4]. Dynamic dataflow MoCs allow port token rates to change based on values of input tokens. Some formulations [12], [13] also allow interpreting Kahn process networks (KPN) [14] as a kind of a dynamic dataflow MoC.

The BDF MoC is related in some ways to that of PRUNE, however PRUNE and its design rules impose some additional restrictions that make it more analyzable than BDF. For example, PRUNE requires that (descriptions of SWITCH and SELECT can be found in [11])

- 1) each SWITCH type actor needs to have a corresponding SELECT actor, unlike BDF;
- 2) data streams that control a pair of SWITCH and SELECT actors need to be identical, which is not the case in BDF;
- 3) the input token rate and output token rate on each end of a FIFO need to be identical.

These three differences are related to minimal examples presented in [11] that break either the *strong consistency* or *bounded memory* assumptions in BDF and preclude decidability.

### A. Analyzability of Dynamic Dataflow MoCs

The disadvantage of dynamic dataflow MoCs is their limited analyzability. A promising approach to provide analyzability and structure to dynamic behavior is parameterization, which restricts the allowed dynamic application behavior to a certain extent. An example of a well-known MoC belonging to this class is Parameterized Synchronous Dataflow (PSDF) [15].

Parameterization, however, does not imply any guarantee on decidability, which means compile-time application analyzability for deadlock freedom and bounded memory. A dataflow MoC that is both decidable and dynamic is Scenario Aware Dataflow (SADF) [16], where each operation mode (scenario) of an application is expressed as a separate SDF graph. Unfortunately, in some situations, such as when there is an arbitrary sample rate change in a signal processing application, the use of the SADF model yields an unwieldy number of scenarios. *Parameterized sets of modes* (PSMs) [17] is a modeling approach that addresses this shortcoming via *mode sets* that enable compact management of changes both in dataflow graph topology and sample rates. Intuitively, a PSM is a modeling abstraction that groups together a collection of related operating modes, where one or more parameters are used to select a unique mode from the collection at compile-time or run time. For details on the PSM modeling approach, we refer the reader to [17].

A different approach for providing decidability to dynamic dataflow is presented by Gao et al. in *well-behaved dataflow* (WBDF) [18]: the use of dynamic actors is restricted to pre-defined actor patterns that have been shown to provide decidability. Advantages of this approach are that it can be adapted to various dataflow MoCs, and extended with new patterns, as needed.

The PRUNE MoC presented in Section III is accompanied with *design rules* (Section IV) in the spirit of WBDF, in order to formulate necessary conditions for guaranteeing decidability, while still maintaining support for dynamic application behavior. The PRUNE MoC also draws from the PSM concepts for compact representation of token rate changes. PRUNE applications undergo a consistency analysis at compile time (Section V), and are executed under an efficient runtime system (Section VI) that targets heterogeneous platforms.

<sup>1</sup><https://gitlab.com/jboutell/Prune>

TABLE I: Comparison to related dataflow models and languages.

| Work              | Decidable | Dynamic | High-performance |
|-------------------|-----------|---------|------------------|
| SADF [16]         | +         | +       | ?                |
| BDF [11]          | -         | +       | ?                |
| DAL [5]           | -         | +(-)    | +                |
| RVC-CAL/Orcc [19] | -         | +       | +                |
| StreamIt [20]     | +         | -       | +                |
| PRUNE             | +         | +       | +                |

### B. Related Programming Frameworks

A number of programming frameworks that target heterogeneous platforms have emerged in the last several years. The frameworks described in [21], [22] and [23] represent *task-based* programming approaches, where tasks are spawned, executed and finished, and their interdependencies are expressed as a directed acyclic graph. The proposed approach, in contrast, is based on actors that are created once at initialization and run as independent entities, communicating with each other until termination of the application.

Concerning actor based programming frameworks, a recent article [20] presents a framework that enables deploying applications written in the StreamIt language [3] to GPUs. Compared to this work, the significant difference is that the StreamIt language heeds the SDF MoC, which does not allow run-time changes in token rates. The same token rate restriction applies to two recent works [24], [25] that discuss deployment of RVC-CAL dataflow programs to heterogeneous architectures.

The DAL framework [26] is based on Kahn process networks and also has an extension [5] for targeting heterogeneous systems with OpenCL enabled devices. In terms of OpenCL / GPU acceleration, this framework is limited to the SDF MoC, which disallows dynamic token rates.

Representative previous works are compared to the proposed PRUNE framework in Table I in terms of decidability, support for dynamic token rates, and experimentally demonstrated high performance on heterogeneous platforms.

## III. PROPOSED MODEL OF COMPUTATION

A common feature in signal processing oriented MoCs is the use of semantic restrictions that a) enhance the potential for application analysis and optimization, while b) being compatible with specialized classes of signal processing applications. The PRUNE MoC has been designed for capturing the behavior of high-performance signal processing applications that can be viewed as having configurable-topology, symmetric-rate dataflow behavior.

Here by *symmetric-rate dataflow*, we mean a restricted form of SDF in which the token production rate is equal to the consumption rate on every FIFO channel. However, the PRUNE MoC is significantly more flexible than SDF in that the connections between actors (graph topology) can be changed at run-time. At the same time, the design rules (Section IV) that are imposed on the construction of PRUNE graphs ensure that important decidability properties are maintained, including analyzability for bounded memory and deadlock-free operation.

### A. Connections of a PRUNE Graph

In the PRUNE MoC, an application is described as a graph  $G = (A, F)$ , where  $A$  is a set of actors and  $F$  is a set of FIFO communication channels that interconnect the actors. Each actor  $a \in A$  may have zero or more input ports and zero or more output ports. If an actor  $a$  has no input ports it is called a *source actor*, and if it has no output ports it is called a *sink actor*. If an actor  $a$  contains port  $p$ , we say that  $a$  is the *parent* of  $p$ , denoted  $parent(p)$ . When needed for clarity, we denote with a superscript +/- the output/input direction of a port, and with a subscript number/letter the index/parent of a port. For example,  $p_{a1}^+$  denotes output port #1 of actor  $a$ .

Each FIFO  $f \in F$  is connected to an output port  $p^+$  of some actor  $parent(p^+)$ , and to an input port  $p^-$  of some actor  $parent(p^-)$ . The ports  $p^+$  and  $p^-$  are referred to, respectively, as the *source port* and *sink port* of  $f$ . We say that the ports  $p^-$  and  $p^+$  are *connected* when they are source and sink ports of the same FIFO — that is, when  $fifo(p^-) = fifo(p^+)$ .

A given output port  $p^+$  can be connected to multiple FIFOs. However, each FIFO has a unique source port and sink port, and each input port  $p^-$  has a unique FIFO that connects to it. When a source port  $p^+$  is connected to multiple FIFOs, and  $parent(p^+)$  writes a token through  $p^+$ , the token is written (i.e., broadcasted) into all of the FIFOs connected to  $p^+$ .

Each port  $p$  has a type that is either a *control input port* (“control port”), a *static regular port* (SRP) or a *dynamic regular port* (DRP). SRPs have a single, fixed, positive token consumption rate (for input ports), or token production rate (for output ports). DRPs, in contrast, have two fixed token rates that are referred to as the *active token rate* ( $atr$ ) of  $p$ , and denoted as  $atr(p)$ , and the *inactive token rate* ( $itr$ ), which is always zero. The consumption rate of a control port is always equal to unity.

A distinguishing aspect of the PRUNE MoC is that each FIFO has a single, positive-integer token rate, denoted by  $fiforate(f)$ , that is associated with it. In conjunction with this association of token rates with FIFOs, the PRUNE MoC imposes the restriction for each port  $p$ ,  $atr(p) = fiforate(fifo(p))$ . In other words, it is a semantic error to have a port that is connected to a FIFO if there is a mismatch between the  $atr$  of the port and the token rate of the FIFO.

Similar to KPN [14], in the PRUNE MoC, blocking reads and blocking writes are assumed for all actors. Under blocking a read, the execution of an actor stalls if it has an input port  $p$  such that the number of tokens in  $fifo(p)$  is less than the number of tokens in  $fiforate(p)$ . The same holds for output FIFOs and their free token slots.

### B. Types of Actors

Computations related to a PRUNE application are performed in firings of actors. A firing  $\phi$  of actor  $a$  consumes tokens from the input ports (FIFOs) of  $a$  and produces tokens to the output ports (FIFOs) of  $a$ . In the PRUNE MoC, each actor has a type that is either *static processing actor*, *dynamic actor*, or *configuration actor*. The allowed port types and their associated firing behavior depend on the type of the actor.

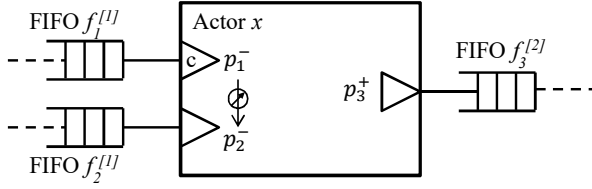


Fig. 1: An illustration of a dynamic actor in the PRUNE MoC.

A dynamic actor  $x$  has at least one DRP, any number of SRPs, and a unique control input port, denoted  $cport(x)$ . When dynamic actor  $x$  performs a firing  $\phi$ , a *control token* is consumed from  $cport(x)$ . The control token sets the token rate for each DRP  $p$  of  $x$  to  $atr(p)$  or  $itr(p)$  for the duration of  $\phi$ . By definition, the token rate of each SRP of  $x$  remains at its fixed positive token rate regardless of the value of the corresponding control token.

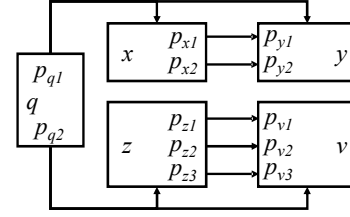
Fig. 1 depicts an example of a dynamic actor. This actor, denoted  $x$ , is connected to three FIFO channels,  $f_1$ ,  $f_2$  and  $f_3$ , through its ports  $p_1^-$ ,  $p_2^-$  and  $p_3^+$ . FIFOs  $f_1$ ,  $f_2$  and  $f_3$  have token rates of 1, 1, and 2 (shown in brackets), respectively. The annotation of port  $p_1^-$  with “c” indicates that this is the control port of the dynamic actor. Port  $p_2^-$  is a DRP, while  $p_3^+$  is an SRP. Values of the tokens consumed from the control port set the token rate of input port  $p_2^-$  to either 0 or 1; in other words,  $atr(p_2^-) = 1$  and  $itr(p_2^-) = 0$ .

A configuration actor has one or more *control output ports*. A control output port of a configuration actor must be an SRP, have a token production rate of unity, and be connected to the control input port of a dynamic actor. Thus, the control tokens consumed by control input ports are always produced by configuration actors: if  $p_x^-$  is a control input port of a dynamic actor  $x = parent(p_x^-)$ , then there is a unique control output port  $p_q^+$  of a configuration actor  $q = parent(p_q^+)$  such that  $p_x^-$  is connected to  $p_q^+$ . In addition to its one or more control output ports, a configuration actor has zero or more *data ports* of type SRP. A data port can be either an input port or output port.

Finally, all ports of a static processing actor  $a$  are of the type SRP and hence active during all firings of  $a$ . Thus, for all firings  $\phi$  of a static processing actor  $a$ , and all ports  $p$  of  $a$ ,  $tokrate(p, \phi) = atr(p)$ , where  $tokrate(p, \phi)$  denotes the token rate of a port  $p$  during a firing  $\phi$  of  $parent(p)$ .

### C. Control and Firing of a Dynamic Actor

Since each control output port  $p_q^+$  of a configuration actor  $q$  is required to connect to a control input port  $p_x^-$  of a dynamic actor  $x$ , and in turn  $p_x^-$  sets the token rate of each DRP of  $x$ , we say that the port  $p_q^+$  *controls* the DRPs of  $x$ . This control relationship can be represented as part of a data structure called the *control table*. The control table for a PRUNE graph or subgraph  $G$  is a matrix whose rows are indexed by control output ports, and columns are indexed by DRPs. The size of the matrix is  $h \times w$ , where  $h$  and  $w$  are the number of control output ports and DRPs, respectively, in  $G$ . Fig. 2 provides an example of a control table.



|          | $p_{x1}$ | $p_{x2}$ | $p_{z1}$ | $p_{z2}$ | $p_{z3}$ | $p_{y1}$ | $p_{y2}$ | $p_{v1}$ | $p_{v2}$ | $p_{v3}$ |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| $p_{q1}$ | 1        | 2        | 0        | 0        | 0        | 1        | 2        | 0        | 0        | 0        |
| $p_{q2}$ | 0        | 0        | 1        | 2        | 3        | 0        | 0        | 1        | 2        | 3        |

Fig. 2: A PRUNE graph and its control table. Configuration actor  $q$  controls the DRPs of dynamic actors  $x$ ,  $y$ ,  $z$  and  $v$ .

When a dynamic actor  $x$  fires, it first consumes a control token from its control port. A control token in turn encapsulates a *control value*  $\bar{v}$ , which is a vector  $\bar{v}[1], \bar{v}[2], \dots, \bar{v}[K]$  of Boolean elements, and  $K$  is the number of DRPs in  $x$ . In other words, there is one element in  $\bar{v}$  for each DRP of  $x$  (For reasons of clarity, here  $K$  is assumed to equal the DRP count of  $x$ . A more general formulation is presented in Section V). A control value  $\bar{v}$  produced on a port  $j$  is related to the control table in that each entry  $T[j][i]$  of the control table indicates the element index in  $\bar{v}$  that is used to configure the DRP  $i$ . For example,  $T[j][i] = 3$  means that the third element of each control token produced on port  $j$  is used to control DRP  $i$ . If port  $j$  is not used to configure DRP  $i$ , then  $T[j][i] = 0$ .

Based on the control value of the consumed control token, the token rate of each DRP  $p$  of  $x$  is fixed to either 0 or  $atr(p)$  for the duration of the current firing  $\phi$ . More specifically, the token rate on DRP  $p$  for firing  $\phi$  is determined by

$$tokrate(p, \phi) = BTOI(\bar{v}_\phi[p]) \times atr(p), \quad (1)$$

where  $\bar{v}_\phi$  denotes the control value consumed in firing  $\phi$ , and  $BTOI$  (Boolean to integer) represents a simple function that maps Boolean values to integer values: that is,  $BTOI(false) = 0$ , and  $BTOI(true) = 1$ .

The computation associated with a given firing  $\phi$  must adhere to the dynamically-adjusted dataflow constraints imposed by Equation 1. Implementation of actor firing functions in PRUNE is discussed in Section VI-A.

### D. Token Delays

Each FIFO channel  $f \in F$  has a non-negative integer *delay* associated with it, which specifies the number of initial tokens that are placed in the channel at system setup time (before the graph is executed). Such delays can be used, for example, to implement the  $z^{-1}$  operator in signal processing (e.g., see [27]). If  $p^+$  and  $p^-$  are two ports that are connected to a common FIFO  $f$ , then (with a minor abuse of notation) we denote the delay associated with  $f$  by  $delay(p^+, p^-)$  or by  $delay(f)$ . It is of high importance to notice that the presence of delays on FIFOs combined with PRUNE’s *symmetric-rate* dataflow behavior can lead to unaligned FIFO accesses. This is discussed in Section VI-B.

## IV. DESIGN RULES

Arbitrary connections of dynamic actors can lead to inconsistent dataflow behavior [11]. Such inconsistencies can lead to deadlock or unbounded accumulation of tokens within FIFOs, which are problematic when a signal processing system must operate on very large, possibly unbounded streams of data [27].

PRUNE imposes a small set of concrete design rules to ensure that dynamic activation of ports in a given graph is performed in a manner that maintains matched patterns of token production and consumption across each FIFO in a PRUNE graph. Thus, tokens that need to be consumed are ensured to have corresponding producers (producing actors), and similarly, production of tokens is matched with a corresponding “demand” to consume the produced data. The design rules therefore ensure consistency in the dynamic dataflow behavior of PRUNE graphs.

In the remainder of this section, we formulate the design rules of PRUNE. The precise formulations provided here in terms of fundamental dataflow and graph-theoretic concepts allow the rules to be checked automatically within design tools. Indeed, in our prototype analysis tool for PRUNE, which we report on in Section V, the design rules are checked automatically to aid the designer in iteratively refining a design as needed until all design rules are satisfied.

As necessary background we first review some graph theoretic concepts in the context of PRUNE graphs. We say that two PRUNE actors  $a$  and  $b$  are *adjacent* if there is a FIFO that connects a port of  $a$  to a port of  $b$ . A *chain* in a PRUNE graph is a non-empty sequence  $S = (a_1, a_2, \dots, a_N)$  of actors in the graph such that for each  $i = 1, 2, \dots, (N-1)$ ,  $a_i$  and  $a_{i+1}$  are adjacent. We say that the chain  $S$  *connects*  $a_1$  and  $a_N$ . The chain  $S$  is a *simple chain* if all of the  $a_i$ ’s are distinct. Given two actors  $x$  and  $y$ , an actor  $z \notin \{x, y\}$  is said to *connect* actors  $x$  and  $y$  if there is a chain  $S$  that connects  $x$  and  $y$  such that  $S$  contains  $z$ .

Now suppose that  $p_x$  and  $p_y$  are distinct ports within two actors  $x$  and  $y$ , respectively, of a PRUNE graph  $G$ . We say that  $p_x$  and  $p_y$  are *linked ports* if (a)  $\text{fifo}(p_x) = \text{fifo}(p_y)$  or (b) there is a simple chain  $(x, a_1, a_2, \dots, a_N, y)$  of actors, where  $p_x$  is connected to a port of  $a_1$ , and  $p_y$  is connected to a port of  $a_N$ . The sequence of  $a_i$ ’s in (b) is referred to a *connecting subchain* associated with the linked ports  $\{p_x, p_y\}$ . Note that for the same linked ports  $\{p_x, p_y\}$ , there can be multiple connecting subchains. If  $p_x$  and  $p_y$  are linked ports, and they are both DRPs, then we say that they are *linked DRPs*.

The **first design rule**, called the *linked port control rule*, is that for each pair  $\{p_x, p_y\}$  of linked DRPs, the ports must be controlled by the same control output port  $p_q$ , and by the same element of the associated control token — that is,  $T[p_q][p_x] = T[p_q][p_y] > 0$ .

The **second design rule**, called the *balanced delay rule*, states that if a control output port  $p_q$  controls DRPs  $cport(x)$  and  $cport(y)$ , then  $\text{delay}(p_q, cport(x)) = \text{delay}(p_q, cport(y))$ . In other words, the control input ports of  $x$  and  $y$  must be connected to  $p_q$  with the same delay.

The **third design rule**, called the *connecting subchain rule*, states that if  $x$  and  $y$  are dynamic actors,  $\{p_x, p_y\}$  are

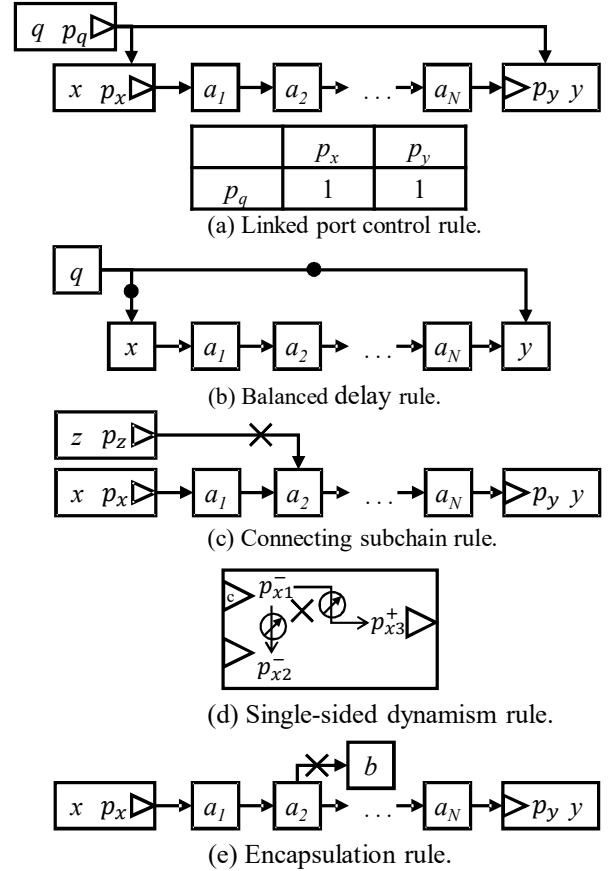


Fig. 3: An illustration of design rules in PRUNE.

linked DRPs with  $\text{parent}(p_x) = x$  and  $\text{parent}(p_y) = y$ ,  $S = (a_1, a_2, \dots, a_N)$  is a connecting subchain associated with  $\{p_x, p_y\}$ , then (1) actor  $a_i \in S$  must be a static processing actor, and (2) each connecting subchain, to which actor  $a_i \in S$  belongs, must be associated with the two dynamic actors  $x$  and  $y$ .

The **fourth design rule**, called the *single-sided dynamism rule*, states that a dynamic actor may only have dynamic input ports, or dynamic output ports, but not both.

The **fifth design rule**, called the *encapsulation rule*, states that if  $x$  and  $y$  are dynamic actors,  $\{p_x, p_y\}$  are linked DRPs with  $\text{parent}(p_x) = x$  and  $\text{parent}(p_y) = y$ ,  $S = (a_1, a_2, \dots, a_N)$  is a connecting subchain associated with  $\{p_x, p_y\}$ , and  $b \notin S$  is an actor that connects to an actor  $a_i \in S$  through an SRP of  $b$ , then  $b$  must belong to a chain that connects  $x$  and  $y$ .

Fig. 3 provides illustrations of the five design rules in PRUNE. The control table shown in the lower part of Fig. 3(a) represents relationships between the control output port  $p_q$  and two DRPs — DRP  $p_x$  of actor  $x$ , and DRP  $p_y$  of actor  $y$ . The control table shows that since DRPs  $p_x$  and  $p_y$  are controlled by the same control output port  $p_q$  and the same element of the associated control token, the linked port control rule is satisfied. Fig. 3(b) shows a graph that satisfies the balanced delay rule, whereas Fig. 3(c) shows an example that violates the connecting subchain rule: actor  $a_2$  of the connecting subchain  $(a_1, a_2, \dots, a_N)$  associated with

the dynamic actors  $x$  and  $y$  also belongs to another connecting subchain, associated with dynamic actors  $z$  and  $y$ . In Fig. 3(d), the dynamic actor  $x$  violates the single-sided dynamism rule, as it contains both input and output DRPs. Finally, Fig. 3(e) depicts a violation of the encapsulation rule. Here, actor  $a_2$  belongs to a chain that connects dynamic actors  $x$  and  $y$ . Actor  $a_2$  is adjacent to actor  $b$ , but  $b$  is not part of a chain that connects  $x$  and  $y$ .

## V. COMPILE TIME GRAPH ANALYSIS

The aim of the proposed design rules is to ensure that deadlock freedom and bounded memory analysis of a PRUNE graph can be completed in finite time, i.e. these problems remain decidable. This section establishes this decidability result for the PRUNE model of computation.

The design rules require that DRPs and their dynamic actor parents  $x$  and  $y$  always appear in pairs and both of these are controlled by the same configuration actor  $q$ . Therefore, in a PRUNE graph  $G$  we can identify zero or more *dynamic processing graphs* (DPGs) that each consist of a) one configuration actor  $q$ , b) exactly two dynamic actors,  $x$  and  $y$ , and c) any number of chains that connect  $x$  and  $y$ . These chains form the *dynamic components* (DCs) of the DPG. Given a DPG  $D$ , the set of DCs of  $D$  is denoted  $Z_c(D)$ , and the pair of dynamic actors contained in  $D$  is denoted  $\delta(D)$ .

Consider a DPG that contains dynamic actor  $x$  with  $K$  output DRPs  $p_{xi}$  ( $i = 1, 2, \dots, K$ ), and dynamic actor  $y$  with  $L$  input DRPs  $p_{yj}$  ( $j = 1, 2, \dots, L$ ),  $\{x, y\} = \delta(D)$ ; we require that each  $p_{xi}$  is a linked DRP with at least one of  $p_{yj}$ . Our procedure for finding the DCs  $Z_c(D)$  associated with a given DPG  $D$  can be expressed as follows:

1. For each linked DRP  $\{p_{xi}, p_{yj}\}$ , where  $\text{fifo}(p_{xi}) = \text{fifo}(p_{yj})$ , insert a dummy actor  $d$  such that  $\text{fifo}(p_{xi}) = \text{fifo}(p_d^-)$  and  $\text{fifo}(p_d^+) = \text{fifo}(p_{yj})$ .
2. Remove  $q$ ,  $x$ ,  $y$ , and all FIFOs  $\text{fifo}(p_q)$ ,  $\text{fifo}(p_x)$  and  $\text{fifo}(p_y)$  in the DPG. This removal procedure decomposes the DPG into a set of connected components that form the DCs. Thus,  $Z_c(D) = \{Z_1, Z_2, \dots, Z_M\}$ , where  $M \in [1, \min(K, L)]$  is an integer constant.

As an example, consider the DPG in Fig. 4 that consists of the configuration actor  $q$ , the pair of dynamic actors  $\delta(D) = \{x, y\}$ , and actors  $a_1$  through  $a_4$ . It can be seen that the linked DRPs of this DPG are  $\{p_{x1}, p_{y1}\}$ ,  $\{p_{x2}, p_{y1}\}$ ,  $\{p_{x3}, p_{y2}\}$ , and  $\{p_{x4}, p_{y3}\}$ .

The linked port-pair  $\{p_{x3}, p_{y2}\}$  is connected over a one-actor chain of  $S = (a_4)$ , whereas the ports  $\{p_{x4}, p_{y3}\}$  are linked directly, i.e.,  $\text{fifo}(p_{x4}) = \text{fifo}(p_{y3})$ . Here, the DC analysis procedure inserts the dummy actor  $d$ . Finally, for  $\{p_{x1}, p_{y1}\}$  and  $\{p_{x2}, p_{y1}\}$ , the adjacent actors  $a_1, a_2$ , and  $a_3$  form one DC. Hence, in this example, the DPG consists of three DCs  $Z_c(D) = \{Z_1, Z_2, Z_3\}$ , where  $Z_1 = \{a_1, a_2, a_3\}$ ,  $Z_2 = \{a_4\}$ , and  $Z_3 = \{d\}$ . The dummy actor  $d$  exists only for the duration of the graph analysis and is not carried to the final implementation.

For a DPG to be *valid*, we require that a) each DC in  $Z_c(D)$  is connected to at least one DRP of  $x$  and to at least one DRP of  $y$  ( $\{x, y\} = \delta(D)$ ), and b) there must be exactly one

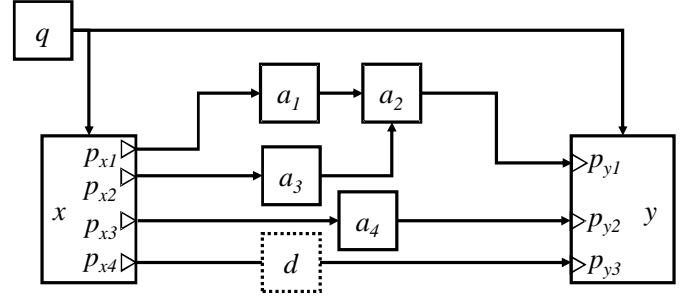


Fig. 4: PRUNE compile time analysis example for one DPG. Connections to the complete PRUNE graph  $G$  are not shown.

Boolean element in  $\bar{v}$  for each DC in  $Z_c(D)$ . Thus, the number of elements in  $\bar{v}$  must equal  $M$ .

Expressed through binary relations of sets, the relation between DCs of  $Z_c(D)$  and the elements of  $\bar{v}$  is required to be a *bijection*. In contrast, the relation  $\text{drps}(x) \rightarrow Z_c(D)$  (and  $\text{drps}(y) \rightarrow Z_c(D)$ ) is surjective, but not necessarily injective. Here, the set of all DRPs of  $x$  is denoted as  $\text{drps}(x)$ . In other words, the DRPs associated with a specific  $Z_k$  must all simultaneously be configured with their respective *attrs*, or they all must be configured with their *itrs*.

Consequently, in a valid DPG  $D$ , each control token (which encapsulates  $\bar{v}$ ) produced by  $q$  sets each DRP  $p_x$  of  $x$  and each DRP  $p_y$  of  $y$  either to its *atr* or its *itr*, where  $\{x, y\} = \delta(D)$ . A control token effectively sets each DC in  $Z_c(D)$  as *active* or *inactive*. In this context, we say that the DC  $Z_k \in Z_c(D)$  is active if it will be provided with tokens through DRPs of  $x$ , and the actors within  $Z_k$  will consequently fire producing tokens that are consumed by actor  $y$ . Inactiveness of  $Z_k$ , in contrast, means that DRPs of  $x$  will not provide tokens to  $Z_k$ , and consequently no actor in  $Z_k$  will fire, and actor  $y$  does not demand tokens from  $Z_k$ .

Returning to Fig. 4: due to Design Rules 1 and 2, for a valid DPG, for example the linked DRP pairs  $\{p_{x1}, p_{y1}\}$  and  $\{p_{x2}, p_{y1}\}$  are all simultaneously set either to their *atr* or to their *itr*; a deadlock would follow if any of the three ports would be set to its *itr*, while the other would be set to its *atr* (and vice-versa).

Above, the necessary background information has been given for our discussion on the decidability of PRUNE graphs. As the design rules of Section IV restrict dynamic actors to exist within DPGs, actors outside DPGs have fixed data rates and have dataflow relationships with DPGs that are simple, and can readily be validated using standard SDF techniques. Thus, in the remainder of this section, the decidability discussion concentrates on DPGs.

In general, a PRUNE graph  $G$  may contain multiple DPGs. However, our design rules require the DPGs of  $G$  to be independent of each other. Since the number of distinct DPGs is finite, our proof of decidability can be reduced to proving that consistency analysis for a single DPGs is decidable.

**Definition 1** (Consistency). A PRUNE graph is consistent if it can be scheduled with guarantees of bounded memory requirements and deadlock-free operation, regardless of what inputs are applied.

**Theorem V.1.** *The consistency analysis of a PRUNE graph is decidable.*

*Proof.* Let  $Z_c(D) = Z_1, Z_2, \dots, Z_M$  be the set of DCs of DPG  $D$ . Since Design Rule 3 requires that all actors in DCs are static processing actors, there is a finite number of different SDF graphs  $Z_1, Z_2, \dots, Z_M$  that can be active during execution of the DPG  $D$ .

Since each  $Z_k, k \in [1, M]$  is an SDF graph, it is decidable to determine whether or not the graph is consistent [7]. If any of the  $Z_k$ 's is not consistent, then deadlock-freedom and bounded memory scheduling for the DPG  $D$  cannot be guaranteed, and therefore  $D$  is inconsistent. On the other hand, if all  $Z_k$ 's are consistent, then there exists a valid, periodic schedule  $P(Z_k)$  for each  $Z_k$  [7].  $P(Z_k)$  provides a schedule for *actors*( $Z_k$ ), where *actors*( $X$ ) represents the set of actors that are contained in a given DC  $X$ .

For each FIFO  $f$  connected to an actor  $a \in \text{actors}(Z_k)$ , there is a buffer bound  $B_k(f)$  which gives the maximum number of tokens on  $f$  during an execution of  $P(Z_k)$ . The existence of this buffer bound follows from the properties of consistent SDF graphs [7]. There is then a finite maximum  $\beta(f) = \max(B_k(f) \mid Z_k \in Z_c(D))$ .

Any execution of the DPG  $D$  can be carried out by a sequence of schedules  $\Omega = (O_1, O_2, \dots)$  where for each  $O_k$ , there is an  $H_k \in Z_c(D)$  such that  $O_k = P(H_k)$ .  $H_k$  can be viewed as the  $k$ th active graph during execution of the enclosing DPG.

Since each  $Z_k$  is assumed to be consistent and have a valid, periodic schedule  $P(Z_k)$ ,  $O_k$  produces no net change in the token populations of the buffers between *actors*( $Z_k$ ), and consequently, the number of tokens on a FIFO  $f$  during an execution of  $O_k$  is bounded by  $B_k(f)$ . It follows that the number of tokens on  $f$  during execution of  $\Omega$  is bounded by  $\beta(f)$ .

In summary, the consistency of the DPG  $D$  can be determined by analyzing the consistency of the elements of  $Z_c(D)$ . Since the elements of  $Z_c(D)$  can be analyzed in finite time (due to the decidability of SDF and the fact that  $Z_c(D)$  has finite cardinality), it follows that consistency analysis for DPG  $D$  is decidable.  $\square$

The design rules presented in Section IV and the analysis presented in this section allow construction and verification of DPGs within a PRUNE application graph  $G$ . A DPG can be seen as a generalization of the *conditional schema* of [18] in terms of the number and topology of conditional branches, and the fact that the branches are not mutually exclusive. Following the same logic, it is possible to formulate useful design rules for other kinds of dynamic constructs on top of the PRUNE MoC, which is a useful direction for future work.

## VI. THE PRUNE FRAMEWORK

The previously described design rules and compile time graph analysis have been implemented to the *PRUNE compiler and analyzer* as shown in Fig. 5. The PRUNE compiler takes three types of input files: the application graph, the platform graph, and the actor-to-platform mapping.

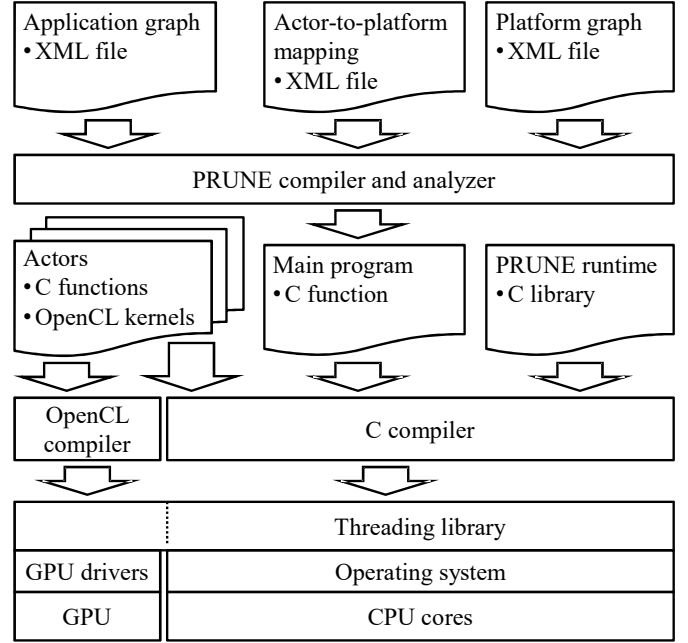


Fig. 5: Overview of the PRUNE framework.

The formats of the application graph and the platform graph have directly been adopted from the DAL framework [26], but some extensions have been introduced to provide support for execution of dynamic actors on GPUs, multi-dimensional OpenCL workloads and static data for OpenCL actors.

The PRUNE compiler and analyzer transforms the XML input files into an internal representation, which is suitable for performing graph analysis and verification. If the sanity checks and the compile-time analysis (Section V) for the input files pass, the compiler outputs the main program file of the application, which takes care of initializing and terminating OpenCL device access, memory allocations, actors and FIFOs.

After the PRUNE compiler and analyzer has successfully produced the main C file of the application, the application is ready to be compiled with the target platform specific C and OpenCL compilers.

Besides the main C file, this compilation step requires the functional description of each actor. The PRUNE actor API (application programming interface) closely follows the DAL API except for GPU-mapped actors that are described in OpenCL (in DAL a small translator converts appropriately formatted C actors to their OpenCL equivalents). The PRUNE actor API essentially provides functions for inter-actor communication, such as `fifoWriteStart`, `fifoWriteEnd`, etc.

Finally, compilation with the target-specific C compiler requires the PRUNE run-time library, which contains application independent actor wrappers, FIFO implementations and OpenCL support. The following detailed description of the PRUNE runtime framework contains some extension compared to our preliminary work [6], where it was first presented. For example, Equation 2 has been generalized to support token delays  $> 1$ .

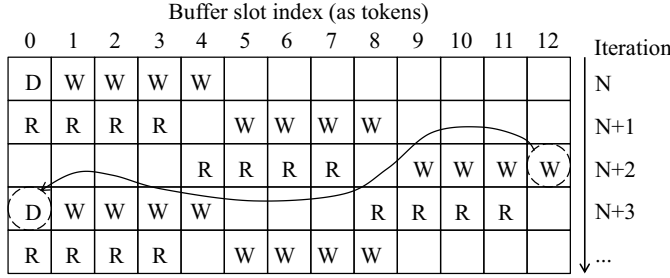


Fig. 6: FIFO channel token access pattern in the case of one delay token for token rate 4 and  $C = 3$ .

#### A. Description of Actors

In the PRUNE runtime framework the description of each actor consists of the mandatory *fire* function, and optional *init*, *control*, and *finish* functions. The *fire* function describes the actor's behavior upon firing and comprises the reading of SRP and DRP input ports, computation and writing to SRP and DRP output ports. The optional *init* and *finish* functions are only executed once on application initialization and termination, and are mainly useful for source and sink actors to start and end interfacing with I/O. The *control* function is only required for dynamic actors and is executed once for each firing of the actor, right before invoking the *fire* function. The *control* function is responsible for reading the control input port and setting the token rates of DRPs.

This formulation, where actors consist of *init*, *fire*, and *finish* functions is identical to the DAL [26] framework. However, the *control* function, especially required for enabling dynamic data rate actors on OpenCL / GPU devices, is specific to PRUNE. The *control* function takes one control token as input and sets the token rate (to  $itr(p)$  or  $atr(p)$  as defined in Section III) of each DRP for the duration of one firing.

#### B. Communication Channels

A communication channel in the proposed framework connects an output port of an actor to an input port of another actor, heeding FIFO behavior. In contrast to other (e.g. [26]) programming frameworks, the capacity  $\Gamma_f$  of a communication channel  $f$  cannot be arbitrarily chosen by the programmer, but is exactly specified as

$$\Gamma_f = \begin{cases} B * (r * C + Q), & Q \text{ not an integer multiple of } r \\ B * \max(r * C, Q), & \text{otherwise,} \end{cases} \quad (2)$$

where  $r = \text{fiforate}(f)$ ,  $B$  is the size (e.g. in bytes) of one token of FIFO  $f$ ,  $Q = \text{delay}(f)$ , and  $C$  is a compile-time constant. For example, setting  $C = 2$  creates a double buffer, and  $C = 3$  creates a triple buffer.

A regular channel (the *otherwise* case in Equation 2) is double or a triple buffer, which allows simultaneous reading and writing of tokens to the channel. On each write,  $f$  assumes to receive  $r$  tokens, and on each read the channel outputs  $r$  tokens. However, for channels that contain initial tokens (delay), the channel is implemented as a slightly more complex buffer that implements a specific access pattern to

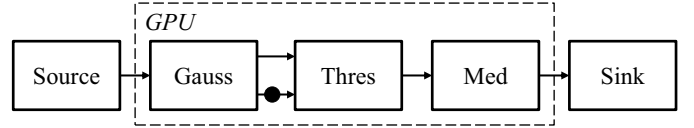


Fig. 7: The Motion Detection application.

enable simultaneous reads and writes to the channel. This is exemplified in Fig. 6 with  $r = 4$ .

At application initialization the initial token in the channel, displayed with  $D$  in Fig. 6, resides in buffer slot 0. The first write to the channel occupies slots 1 ... 4, whereas the first read consumes tokens from slots 0 ... 3 and so forth. The third write to the channel reaches the end (slot 12) of the buffer, followed by an explicit data copy from slot 12 to slot 0, and the access pattern starts to repeat. The access pattern is repetitive and can be generalized to any token rate beyond one.

Looking at Fig. 6, it is evident that this solution does not minimize the memory footprint, but it was chosen as it offers 1) uncompromised throughput and 2) transparency to the application programmer. Ring buffers were considered inadequate, as OpenCL / GPUs offer the best combination of performance and ease of programming when input and output data to actors is provided as contiguous arrays.

#### C. Concurrency, Scheduling and Actor-to-Core Mapping

The proposed framework has been designed to enable maximally parallel operation. Parallelism is based on threading, such that each actor runs on an operating system (OS) thread of its own, regardless whether the actor is targeted to OpenCL / GPU devices or to one of the CPU cores. Each actor thread is created once at application startup, and is canceled after the application has terminated. Similar to the DAL framework [26], [5], synchronization of data exchange over FIFO channels is based on *mutex* locks and blocking communication: if an actor attempts to read a channel that has less tokens than the actor requires, the reading actor blocks until sufficient data is available. This enables very efficient multiprocessing, but on the other hand makes the MoC somewhat more restricted than e.g. that of DPNs [13].

As each actor is instantiated as a separate thread using the GNU/Linux *pthread*s library, the scheduling of actor firings (heeding data availability) is left to the OS. If the programmer so chooses, the framework allows fixing of actors to specific CPU cores, otherwise the OS chooses the core on which the actor is executed.

It is necessary to state that alternatively to the adopted OS threading based concurrency, it would also have been possible to build concurrency and synchronization on top of OpenCL events, however this would have limited the applicability to platforms where both the CPU cores and GPUs have OpenCL drivers. The adopted OS threading based solution, however, is beneficial due to its backwards compatibility: with this solution it is possible to jointly synchronize and run also non-OpenCL compatible CPU cores with GPUs.



TABLE II: Platforms used for experiments.

| Tag     | CPU                                     | OpenCL Device                               | Operating System        |
|---------|---|---|-------------------------|
| Carrizo | AMD Pro A12-8800B (2.1 GHz, 4 cores)    | AMD Radeon R7, OpenCL 2.0, driver 15.30.3   | Ubuntu 14.04, g++ 4.8.4 |
| i7      | Intel Core i7-6700HQ (2.6 GHz, 4 cores) | CPU, Intel OpenCL 1.2 driver 6.2.0.1760     | Ubuntu 16.04, g++ 4.8.5 |
| RX      | Intel Core i7-4770 (3.5 GHz, 4 cores)   | AMD Radeon RX 460, OpenCL 1.2, driver 16.40 | Ubuntu 16.04, g++ 4.8.5 |

TABLE III: Motion Detection performance in HD 1080p frames/s.

| Tag     | DAL Multicore | PRUNE Multicore | DAL Heterogen. | PRUNE Heterogen. |
|---------|---------------|-----------------|----------------|------------------|
| Carrizo | 15.9          | 17.3            | 132            | 140              |
| i7      | 39.2          | 39.8            | -              | 113              |
| RX      | 41.1          | 42.9            | 696            | 772              |

TABLE IV: Digital Predistortion performance in complex float megasamples/s.

| Tag     | DAL Multicore | PRUNE Multicore | DAL Heterogen. | PRUNE Heterogen. |
|---------|---------------|-----------------|----------------|------------------|
| Carrizo | 6.15          | 6.64            | n/a            | 39.5             |
| i7      | 6.78          | 7.34            | n/a            | 25.1             |
| RX      | 17.2          | 19.2            | n/a            | 106              |

TABLE V: Adaptive Deep Neural Network performance in frames/s.

| Tag     | DAL Multicore | PRUNE Multicore | DAL Heterogen. | PRUNE Heterogen. |
|---------|---------------|-----------------|----------------|------------------|
| Carrizo | 8.63          | 11.5            | n/a            | 160              |
| i7      | 9.37          | 9.86            | n/a            | 299              |
| RX      | 26.2          | 37.8            | n/a            | 1033             |

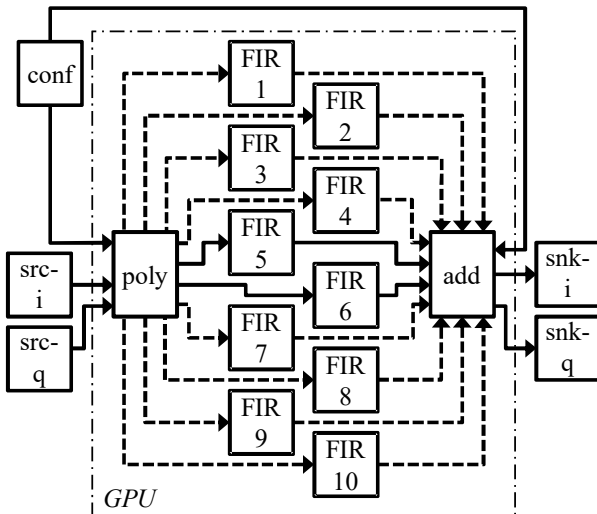


Fig. 8: The Dynamic Predistortion application.

## VII. EXPERIMENTAL RESULTS

This section presents experimental evaluation, which shows that the PRUNE framework is efficient and suitable for running real-life signal processing workloads. The performance results are compared against the DAL framework [5], which in many ways resembles PRUNE.

### A. Use Case Applications

1) *Video Motion Detection*: The first application used in our experiments is 8-bit grayscale video Motion Detection that consists of five actors, as shown in Fig. 7. The source and sink actors are always executed on CPU cores and are essentially responsible for reading and writing data from/to mass storage. The Gauss actor performs  $5 \times 5$  pixels Gaussian filtering on input frames, followed by the Thres actor that subtracts consecutive frames and performs pixel thresholding against a fixed constant value. To avoid exceeding frame boundaries, the Gauss actor skips filtering for two pixel rows in the frame top and frame bottom. Finally, the Med actor performs 5-pixel median filtering to reduce noise from the generated motion map.

The distinguishing feature of this use case application is the use of delay tokens: one of the communication channels between the Gauss and Thres actors bears a dot in Fig. 7 and depicts an initial token, which is a one-frame delay that enables consecutive frame subtraction functionality. Out of the use case applications, Motion Detection is the only one that can be described using fixed token rates, and hence be GPU-benchmarked with both PRUNE and the DAL framework, which we use as a state-of-the-art reference.

The frame size used was  $1920 \times 1080$ , which resulted in the token size becoming 1.98 megabytes. Due to the large token size, the token rate was kept at 1 (in our previous publication [6] that uses a preliminary version of PRUNE, resolution was  $320 \times 240$  with a token rate of 4). GPU acceleration was applied to Motion Detection by mapping the Gauss, Thres and Med actors to the GPU.

2) *Dynamic Predistortion Filtering*: Dynamic Predistortion (DPD) filtering (Fig. 8) was used as the second application use case. The algorithm [28] is used in wireless communications to mitigate transceiver impairments, and essentially consists of 10 parallel 10-tap complex-valued floating-point FIR filters.

DPD significantly differs from the Motion Detection application in the sense that it features actors with dynamic token rates: Fig. 8 shows the configuration (conf) actor that at run-time periodically reconfigures the poly and Adder (add) actors to select which set of the FIR filters is used to process the input signal. The reconfiguration period was set to once every 65536 samples, and the number of active filter actors is allowed to change arbitrarily between 2 and 10. The run time

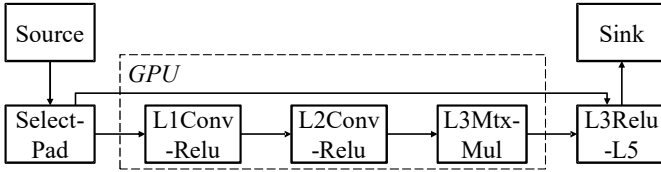


Fig. 9: The Adaptive Deep Neural Network application.

reconfiguration used here is defined by an external input and cannot be modeled e.g. by the CSDF MoC.

The DPD application computes on complex floating-point numbers, which were represented as a pair of single precision floats. To this end, all edges in Fig. 8 *inside* the "GPU" box represent a pair of edges, one for the real part and one for the imaginary part. Hence, the total number of FIFO channels is 46 in this application.

3) *Adaptive Deep Neural Network*: The Deep Neural Network (DNN) application for vehicle classification has first been presented in [29], [30]. The neural network consists of two convolutional layers followed by three dense layers. The PRUNE implementation of the DNN consists of seven actors, as depicted in Fig. 9. Here, the application has been extended with adaptiveness that allows dynamically enabling and disabling DNN processing for each frame to, e.g., save on power when deployed to an energy-limited device.

The three GPU-mapped actors (Fig. 9) represent the neural network core of the application and are demanding both in terms of memory footprint and computational complexity. Layer 1 convolution consists of 2400 floating-point weights, layer 2 of 25600 weights and layer 3 matrix multiplication of 1.8M weights. The input image is delivered in resolution  $96 \times 96$  (RGB) and separated to 32 feature maps that are convolved by  $5 \times 5$  pixel kernels in layers 1 and 2.

The GPU-accelerated convolution layers perform simultaneous convolution and ReLU non-linearity computation, and are processed as a 3-dimensional volume along feature map index, image width and image height axes. Data was mapped to tokens such that the data associated with one input image was mapped to one token. Hence, due to the nature of the algorithm, the token size varied considerably from one FIFO to another. Benchmarking was performed with  $atr = 24$  for each FIFO channel, as this token rate provided the highest throughput.

Adaptiveness was implemented to the DNN application by introducing a *bypass* channel from the actor Select-Pad to the actor L3Relu-L5. With the bypass channel the computationally demanding DNN processing can be omitted for selected frames, and instead of classification results, the bypass channel provides constant values to the output to indicate omitted classification.

In order to demonstrate the possibility for performing post-analysis application optimizations, the configuration actor was merged to the dynamic actor Select-Pad, preserving identical application functionality. For the moment, the actor merging needs to be done manually, however automatic approaches exist [31].

Besides lack of dynamic token rates, implementation of

GPU accelerated DNN in DAL turned out to be unfeasible for several reasons. First, DAL only supports 1-dimensional kernel processing (OpenCL NDRange), and second, DAL provides no direct means to deliver megabytes of fixed coefficients to GPU-accelerated kernels. For these reasons, no GPU accelerated DAL version of the application was created.

### B. The Platforms

Table II shows the platforms that were used to benchmark the PRUNE framework and the Distributed Application Layer, which was used as a reference. The *Carrizo* chip features 4 CPU cores and an integrated graphics processor, a solution that minimizes the data transfer times between the GPU and the CPU cores. *RX* represents a conventional desktop system with a 4-core CPU and a mid-range GPU that is connected to the CPU over a PCI express bus. Finally, *i7* represents a laptop processor with OpenCL drivers that allow accelerating data parallel workloads on the CPU cores.

### C. Experimental Setup

For all use case applications, the PRUNE run-time library was configured to implement triple-buffering of FIFOs ( $C = 3$ , see Eq. 2), which provided equal FIFO memory sizes as DAL. DAL applications were implemented to use high-speed *windowed FIFOs* in communication between CPU cores. For each platform and each use case application the execution time was calculated from the average of 8 successive application executions. Before measurements, it was ensured that the processor cores were almost idle by closing unnecessary applications in the OS.

In Motion Detection the OpenCL global work size was set to 76800 for *i7* and to 518400 for *Carrizo* and *RX*. The input data file was a grayscale 300 frame uncompressed sequence "Jockey" in resolution  $1920 \times 1080$ , which resulted in a file size of 593 MB.

For the DPD application, OpenCL global work size was equal to the actor input token rate, which was either 65527 or 65536 depending on the actor. The same work size was used for all platforms. The DPD input data stream consisted of 67 megasamples, altogether 537 MBs of size.

For Adaptive DNN the OpenCL work size dimensions were  $768 \times 52 \times 52$  for L1Conv-Relu,  $768 \times 24 \times 24$  for L1Conv-Relu, and  $24 \times 100$  for L3Mtx-Mul. The input sequence consisted of 384 RGB frames that had a file size of 40.5 MB due to their single-precision float data format.

### D. Results

Table III shows that with the Motion Detection application the PRUNE framework provided 2-9% higher throughput than DAL on each platform, when no OpenCL acceleration was used, but all processing was done by CPU cores (the columns labeled "Multicore"). With OpenCL acceleration ("Heterogen." columns) the Motion Detection application throughput increased  $3 \times$  to  $18 \times$  such that PRUNE was 6% to 11% faster than DAL, depending on the platform. Under DAL the Intel OpenCL drivers caused an error on the *i7* that prohibited benchmarking.

Multicore-only benchmarking of the Digital Predistortion application revealed 8%-12% higher throughput for PRUNE compared to DAL, as Table IV shows. OpenCL acceleration increased the application performance by  $3\times$  to  $6\times$  under PRUNE compared to multicore-only. DAL OpenCL results could not be acquired, as DAL is restricted to static token rates under OpenCL.

The Adaptive Deep Neural Network application revealed (Table V) larger differences in throughput: under multicore-only, PRUNE was 5% to 44% faster than DAL. OpenCL acceleration was also remarkably powerful, as the application speeded up between  $14\times$  to  $30\times$  by OpenCL on PRUNE. As mentioned in Section VII-A3, OpenCL results could not be acquired due to several restrictions of DAL.

### VIII. DISCUSSION AND FUTURE WORK

The use case applications introduced in Section VII-A demonstrate that the PRUNE Model of Computation is expressive enough for describing a wide variety of performance-intensive signal processing applications, which are highly relevant to the video processing, computer vision and wireless communications fields. In contrast, the state-of-the-art framework DAL could not provide means for OpenCL acceleration of Digital Predistortion or Adaptive DNN applications, or means for analyzing the consistency of the application graphs.

The results in Section VII-D show that the PRUNE runtime framework is also remarkably efficient compared to the state-of-the-art DAL framework: application performance on CPU cores is up to 44% higher under PRUNE. PRUNE also enables highly efficient simultaneous use of OpenCL devices: performance increases up to  $30\times$  were measured compared to CPU-only.

As future work for PRUNE, interfaces for importing signal processing applications written for the DAL framework and the Open-RVC-CAL Compiler [19] will be developed. This will enable the novel capabilities for high performance signal processing in PRUNE to be leveraged by applications.

### IX. CONCLUSION

In this article the PRUNE Model of Computation and framework has been presented. The dataflow oriented PRUNE Model of Computation is expressive enough for describing signal processing applications with dynamic token rates, yet it provides at the same time decidable deadlock freedom and memory boundedness analysis of applications.

The expressiveness of PRUNE has been demonstrated by examples from three signal processing domains: computer vision, video processing and wireless communications. Experimental results have shown that besides decidability, PRUNE also provides more expressiveness and higher performance than the previously published state-of-the-art DAL framework.

### ACKNOWLEDGMENT

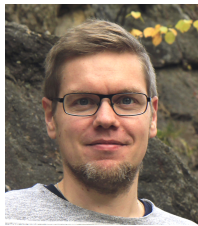
This work was partially funded by the Academy of Finland project 309693 UNICODE and by TEKES — the Finnish Technology Agency for Innovation (FiDiPro project StreamPro 1846/31/2014).

### REFERENCES

- [1] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for rapid prototyping," in *IEEE/IFIP International Symposium on Rapid System Prototyping (RSP)*, 2008, pp. 17–23.
- [2] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, Feb 1996.
- [3] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Compiler Construction*, ser. Lecture Notes in Computer Science, R. N. Horspool, Ed. Springer, 2002, vol. 2304, pp. 179–196.
- [4] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.
- [5] L. Schor, A. Tretter, T. Scherer, and L. Thiele, "Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL," in *IEEE Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, 2013, pp. 41–50.
- [6] J. Boutellier and I. Hautala, "Executing dynamic data rate actor networks on OpenCL platforms," in *IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2016, pp. 98–103.
- [7] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.
- [8] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Embedded Software: First International Workshop (EMSOFT)*, T. A. Henzinger and C. M. Kirsch, Eds., 2001, pp. 166–184.
- [9] M. Mattavelli, I. Amer, and M. Raulet, "The Reconfigurable Video Coding standard [standards in a nutshell]," *IEEE Signal Processing Magazine*, vol. 27, no. 3, pp. 159–167, 2010.
- [10] H. Berg, C. Brunelli, and U. Lucking, "Analyzing models of computation for software defined radio applications," in *International Symposium on System-on-Chip*, 2008, pp. 1–4.
- [11] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 1, April 1993, pp. 429–432 vol.1.
- [12] E. A. Lee and E. Matsikoudis, "The semantics of dataflow with firing," in *From Semantics to Computer Science: Essays in memory of Gilles Kahn*, G. Huet, G. Plotkin, J. Lévy, and Y. Bertot, Eds. Cambridge University Press, 2009.
- [13] A. Tretter, J. Boutellier, J. Guthrie, L. Schor, and L. Thiele, "Executing dataflow actors as Kahn processes," in *International Conference on Embedded Software (EMSOFT)*, 2015, pp. 105–114.
- [14] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing*, J. L. Rosenfeld, Ed. Stockholm, Sweden: North Holland, 1974, pp. 471–475.
- [15] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, 2001.
- [16] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *International Conference on Embedded Computer Systems (SAMOS)*. IEEE, 2011, pp. 404–411.
- [17] S. Lin, L.-H. Wang, A. Vosoughi, J. R. Cavallaro, M. Juntti, J. Boutellier, O. Silvén, M. Valkama, and S. S. Bhattacharyya, "Parameterized sets of dataflow modes and their application to implementation of cognitive radio systems," *Journal of Signal Processing Systems*, vol. 80, no. 1, pp. 3–18, 2015.
- [18] G. R. Gao, R. Govindarajan, and P. Panangaden, "Well-behaved dataflow programs for DSP computation," in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 5. IEEE, 1992, pp. 561–564.
- [19] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "Orcc: Multimedia development made easy," in *ACM International Conference on Multimedia*, 2013, pp. 863–866. [Online]. Available: <http://doi.acm.org/10.1145/2502081.2502231>
- [20] H. P. Huynh, A. Hagiescu, O. Z. Liang, W.-F. Wong, and R. S. M. Goh, "Mapping streaming applications onto GPU systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 9, pp. 2374–2385, 2014.
- [21] V. Boulos, S. Huet, V. Fristot, L. Salvo, and D. Houzet, "Efficient implementation of data flow graphs on multi-GPU clusters," *Journal of Real-Time Image Processing*, vol. 9, no. 1, pp. 217–232, 2014.
- [22] A. Shirlea, Y. Zou, Z. Budimlic, J. Cong, and V. Sarkar, "Mapping a data-flow programming model onto heterogeneous platforms," in *ACM*

*SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, 2012, pp. 61–70.

- [23] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, “XKaapi: A runtime system for data-flow task programming on heterogeneous architectures,” in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013, pp. 1299–1308.
- [24] W. Lund, S. Kanur, J. Ersfolk, L. Tsiopoulos, J. Lilius, J. Haldin, and U. Falk, “Execution of dataflow process networks on OpenCL platforms,” in *Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2015, pp. 618–625.
- [25] J. Boutellier and T. Nylander, “Programming graphics processing units in the RVC-CAL dataflow language,” in *IEEE Workshop on Signal Processing Systems (SiPS)*, 2015, pp. 1–6.
- [26] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele, “Scenario-based design flow for mapping streaming applications onto on-chip many-core systems,” in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2012, pp. 71–80.
- [27] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, 2nd ed. Springer, 2013, iISBN: 978-1-4614-6858-5 (Print); 978-1-4614-6859-2 (Online).
- [28] M. Abdelaziz, A. Ghazi, L. Anttila, J. Boutellier, T. Lähteensuo, X. Lu, J. R. Cavallaro, S. S. Bhattacharyya, M. Juntti, and M. Valkama, “Mobile transmitter digital predistortion: Feasibility analysis, algorithms and design exploration,” in *Asilomar Conference on Signals, Systems and Computers*, 2013, pp. 2046–2053.
- [29] H. Huttunen, F. S. Yancheshmeh, and K. Chen, “Car type recognition with deep neural networks,” in *IEEE Intelligent Vehicles Symposium*, 2016, pp. 1016–1021.
- [30] R. Xie, H. Huttunen, S. Lin, S. S. Bhattacharyya, and J. Takala, “Resource-constrained implementation and optimization of a deep neural network for vehicle classification,” in *24th European Signal Processing Conference (EUSIPCO)*. IEEE, 2016, pp. 1862–1866.
- [31] J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silvén, “Actor merging for dataflow process networks,” *IEEE Transactions on Signal Processing*, vol. 63, no. 10, pp. 2496–2508, May 2015.



**Jani Boutellier** received the M.Sc. and Ph.D. degrees from the University of Oulu, Finland, in 2005 and 2009, respectively. Currently he is an Assistant Professor at the Laboratory of Pervasive Computing of Tampere University of Technology, Finland. In 2007–2008, 2013 he was working as a visiting researcher with the EPFL, Switzerland. His research interests include dataflow programming, signal processing, hardware-software codesign and heterogeneous computing. He is a member of the IEEE Signal Processing Society Design and Implementation of Signal Processing Systems Technical Committee.



**Jiahao Wu** received the bachelor's degree from the University of Electronic Science and Technology of China (UESTC). He joined the Department of Electrical and Computer Engineering at the University of Maryland, College Park as a Ph.D. Student in 2014. His research interests include model-based design for parallel computing, dataflow implementations and synthesis of digital signal processing systems.



**Heikki Huttunen** was born in 1971 and received M.Sc. in mathematics from University of Tampere, Finland, in 1995 and Ph.D. degree in signal processing from Tampere University of Technology, Finland, in 1999. Between 2003–2005 he worked with Visy Oy, developing automatic license plate recognition systems. Since then he held various positions at Tampere University of Technology where he currently is an associate professor leading the machine learning group. His research interests are in the field of machine learning, in particular in deep learning, with research focus on efficient real-time and real-life implementations, and classifier error estimation. Dr. Huttunen has published ca. 100 journal and conference articles and he is an Associate Editor of *Journal of 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). He is an author of six books, and over 250 papers in the areas of 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.