Research Article

Exploiting the Expressiveness of Cyclo-Static Dataflow to Model Multimedia Implementations

Kristof Denolf,[1] Marco Bekooij,[2] Johan Cockx,[1] Diederik Verkest,[1, 3, 4] and Henk Corporaal[5]

[1] Nomadic Embedded Systems (NES), Interuniversity Micro Electronics Centre (IMEC), Kapeldreef 75, 3001 Leuven, Belgium
[2] NXP Research, Systems and Circuits, Prof. Holstlaan 4, 5656 AE Eindhoven, The Netherlands
[3] Department of Electrical Engineering, Katholieke Universiteit Leuven (KU-Leuven), 3001 Leuven, Belgium
[4] Department of Electrical Engineering, Vrije Universiteit Brussel (VUB), 1050 Brussels, Belgium
[5] Faculty of Electrical Engineering, Technical University Eindhoven, Den Dolech 2, 5612 AZ Eindhoven, The Netherlands

The design of increasingly complex and concurrent multimedia systems requires a description at a higher abstraction level. Using an appropriate model of computation helps to reason about the system and enables design time analysis methods. The nature of multimedia processing matches in many cases well with cyclo-static dataflow (CSDF), making it a suitable model. However, channels in an implementation often use for cost reasons a kind of shared buffer that cannot be directly described in CSDF. This paper shows how such implementation specific aspects can be expressed in CSDF without the need for extensions. Consequently, the CSDF graph remains completely analyzable and allows reasoning about its temporal behavior. The obtained relation between model and implementation enables a buffer capacity analysis on the model while assuring the throughput of the final implementation. The capabilities of the approach are demonstrated by analyzing the temporal behavior of an MPEG-4 video encoder with a CSDF graph.

1. INTRODUCTION

The increasing complexity and concurrency in digital multi-processor systems used to build modern multimedia codecs or wireless communications require a design flow covering different abstract layers that evolve gradually towards a final, efficient implementation. Describing the system first at higher level of abstraction, using a model of computation (MoC), permits the designer to model and reason about the system.

Dataflow MoCs have proven to be useful for describing multimedia processing applications [1] as they enable a natural visual representation exposing the parallelism and allowing an evaluation of the temporal behavior. Cyclo-static dataflow (CSDF) [2] is particularly interesting because this variant is one of the most expressive dataflow models while still being fully analyzable at design time (e.g., consistency checks, dead-lock analysis).

An implementation on a multiprocessor platform has optimized communication channels, often based on shared buffers, to improve the efficiency. Examples are a sliding window for data reuse or a circular buffer with multiple consumers. Also, due to implementation restrictions, buffer sizes are limited. As it is not always clear how the behavior of such channels can be expressed in a CSDF model, the designer could judge it as an unsuited MoC, thus losing its analysis potential.

This paper studies how such implementation aspects can be represented in a CSDF model within its current definition. Its main contribution is the modeling of special behavior on channels, such as data reuse or shared buffers, used in an implementation to improve the efficiency. The proposal of a short-hand notation for these special channels provides an intuitive expression of shared memory related aspects in CSDF without requiring extensions of the MoC. As a result, the enriched CSDF graph remains fully analyzable at design time and allows reasoning about the temporal behavior. The capabilities of the approach are demonstrated by describing a power-efficient custom implementation of an MPEG-4 part 2 video encoder using these special channels.

The special channels and the limited buffer sizes are modeled in CSDF by representing them by two edges, one

forward edge assuring the synchronization and one backward edge monitoring the free buffer space. Conditions are formulated on those two edges to assure functional correctness of the modeled application (i.e., no overwriting of live data) and these conditions are verified for every special channel. A basic technique for the buffer capacity calculation through life-time analysis is presented.

Other works only mention using extensions to (C)SDF to describe image [3] and video [4] applications without a formal description of these extensions. Reference [5] integrates CSDF in a parameterized dataflow model to allow dynamic data production and consumption rates. The modeling of buffer bounds by using a feedback edge is introduced in [1] for interprocessor communication graphs (a type of homogenous synchronous dataflow graph) and in [6] to explore the tradeoff between throughput and buffer requirements. To deal with global parameters, [7] describes a synchronous piggybacked dataflow model.

This paper is organized as follows. After summarizing dataflow theory and introducing the basics of CSDF in the next section, the modeling of an implementation including its special edges is discussed in Section 3. In Section 4, an approach for the buffer capacity calculations is presented. After the case study on an MPEG-4 part 2 video encoder in Section 5, conclusions close this document.

2. DATAFLOW MODELS

In the application specific domain, specialized models of computation like dataflow models aid in identifying and exploring the parallelism, and in the manual or automatic derivation of optimized implementations [8]. The choice of the model of computation is a tradeoff between its expressiveness and well-behavior [3]. In this work, a dataflow model is chosen as it combines the expressivity of block diagrams and signal flow charts while preserving the semantics for system design and analysis tools [9]. More specifically, a cyclo-static dataflow model is chosen as it is one of the most expressive while keeping all analysis potentials at design time.

2.1. Definitions of dataflow theory

A comprehensive introduction to dataflow modeling is included in [1, 10]. This subsection gives a summary to introduce the dataflow definitions and terminology. In dataflow, the application is described as a directed graph G. The vertices of this graph are called actors and correspond to the tasks of the application transforming input data into output data. They are by definition atomic (i.e., indivisible). The edges (arcs) represent channels carrying tokens between the communicating actors. The edges act as First-In-First-Out (FIFO) queues with a theoretically unlimited depth. A token is a synchronizing communication object. It can be used to represent a container or just to model synchronization. Containers are fixed-size data structures.

The actor execution is data-driven: it is enabled to fire as soon as sufficient tokens are available on all inputs (i.e, its firing-rule, a boolean expression in the number and/or the value of tokens, turns true). An actor consumes tokens from its input edges in one atomic action at the start of the firing and writes tokens on its output edges in one atomic action at the end of the firing. The number of tokens consumed and produced is, respectively, given by the consumption and production rules on the corresponding edges. The response time (RT) of an actor is the elapsed time between its enabling and the end of the firing.

The data-driven operation of a dataflow graph allows synchronization between the actors: an actor cannot be executed prior to the arrival of its input tokens. When a graph can run without a continuous increase or decrease of tokens on its edges (i.e., with finite queues) it is said to be consistent. A dataflow graph is called nonterminating or live if it can run forever.

For a DSP-application, both the liveness and consistency of the graph are required to get a proper execution. A forever running execution can be obtained by repeating one iteration of a periodic schedule [11]. To keep the number of tokens on the edges limited, the number of tokens produced on an edge during one period must equal the number of tokens consumed from it. The number of actor firings in one period can be derived from this consistency requirement. The existence of a deadlock-free schedule for one iteration [11] is a sufficient condition for a graph to be live. Any such schedule is called a valid static schedule of the graph.

Depending on how the consumption and production together with the firing rules are specified, different classes of graphs are distinguished [2]: homogeneous synchronous dataflow (HSDF), synchronous dataflow (SDF), cyclo-static dataflow (CSDF), and dynamic dataflow (DDF). This paper concentrates on the CSDF model.

2.2. Temporal monotonic behavior

The data-driven operation of a dataflow graph allows its execution in a selftimed manner: actors start as soon as they are enabled. Additionally, the FIFO ordering of the tokens assures they cannot overtake each other. The FIFO ordering of the tokens is automatically respected on the edges of a dataflow graph as these edges act as queues. In the actors, the FIFO ordering is guaranteed if autoconcurrency is excluded by a selfcycle with a single token forcing sequential firing of this actor or by making the response time of the actors constant.

These two properties are a sufficient condition for the definition in [12–14] of the monotonic execution of a dataflow graph G as follows: if firing i of actor A consumes token t, then G executes monotonically if no decrease in response time of any firing of any actor can lead to a later enabling of firing i of actor A. It is shown that a dataflow graph with selftimed execution that maintains the FIFO ordering of the tokens possesses this important property of monotonic behavior in time. As a result, a decrease in response time can only lead to earlier token production and consequently to an equal or earlier actor enabling. Overall, this could possibly lead to a higher throughput.

In this work, the focus is on cyclo-static dataflow [2] as it is deterministic and allows checking conditions such as deadlocks and bounded memory execution at compile/design time. This is not always possible for DDF. Additionally, if dynamic dataflow concepts are required to model a multimedia application, this is often only needed for a part of the graph and can sometimes be reduced to CSDF by considering worst-case scenarios [15].

After introducing the elements and properties of CSDF in the next subsection, it will be shown that there exists a consistent relation between CSDF model and implementation. As a result, containers will not arrive later in an implementation with selftimed execution than the corresponding tokens in the CSDF model. If worst-case response times are used while building this schedule, the worst-case throughput is known and guaranteed.

2.3. Basics of CSDF

Cyclo-static dataflow modeling was first proposed by Bilsen et al. [2] as extension of SDF. In CSDF, each actor A has an execution sequence of length L_A, called the actor period. Consequently, the production and consumption are also sequences of constant integers noted on the corresponding side of the edge e_u as $\{p_P^u(0), p_P^u(1), \ldots, p_P^u(L_P - 1)\}$ for the producer P and $\{c_C^u(0), c_C^u(1), \ldots, c_C^u(L_C - 1)\}$ for the consumer C. The $(i+1)$th firing of actor P produces $p_P^u(i \bmod L_P)$ tokens on edge e_u. Similarly, the $(j+1)$th firing of actor C consumes $c_C^u(j \bmod L_C)$ tokens from the same edge. The firing rule of an actor A becomes true for its $(j + 1)$th firing if all inputs contain at least $c_A^u(j \bmod L_A)$ tokens. Also for CSDF, the consistency can be evaluated through the balance equations and a valid static schedule can be found [2] at compile time.

The rest of this subsection briefly explains how the consistency and liveliness of a CSDF graph are evaluated. More details are given in [1, 2]. The following notation are used in the rest of the text:

(i) L_A actor period or cycle length of the sequences of actor A;

(ii) $p_A^u(i)$ number of tokens produced on edge e_u by actor A during its $(i + 1)$th firing

$$p_A^u(i) = \begin{cases} (i + 1)\text{th element in the} \\ \text{production sequence} & \text{if } 0 \le i \le L_A - 1, \\ p_A^u(i \bmod L_A) & \text{if } i \ge L_A; \end{cases} \tag{1}$$

(iii) $c_A^u(j)$ number of tokens consumed from edge e_u by actor A during its $(j + 1)$th firing

$$c_A^u(j) = \begin{cases} (j + 1)\text{th element in the} \\ \text{production sequence} & \text{if } 0 \le j \le L_A - 1, \\ c_A^u(j \bmod L_A) & \text{if } j \ge L_A; \end{cases} \tag{2}$$

(iv) $P_A^u(k)$ number of tokens produced on edge e_u by actor A after k firings

$$P_A^u(k) = \sum_{i=0}^{k-1} p_A^u(i); \tag{3}$$

(v) $C_A^u(l)$ number of tokens consumed from edge e_u by actor A after l firings

$$C_A^u(l) = \sum_{j=0}^{l-1} c_A^u(j); \tag{4}$$

(vi) q_A^b basic repetition rate of actor A (see below).

A CSDF graph G is compactly represented by its topology matrix Γ containing one column for each actor and one row for each edge. Its (i, j)th entry corresponds to the total number of tokens produced/consumed by the actor with number j on the edge with number i during one period. If the actor with number j produces tokens, the entry is positive while for a consuming actor, the entry is negative. The actor period matrix L contains one row with the actor periods. Its jth entry holds the actor period of the actor with number j.

A period balance vector r is a positive solution of the balance equations

$$\Gamma \cdot r^T = 0. \tag{5}$$

Such a period balance vector only exists if

$$\text{rank}(\Gamma) = N_G - 1 \tag{6}$$

with N_G the number of actors in the CSDF graph. A repetition vector q is the product of a period balance vector r with the actor periods

$$q = r \cdot L_{\text{diag}} \tag{7}$$

with L_{diag} the diagonal version of L. The basic repetition vector q^b can be derived from any arbitrary repetition vector q as

$$q^b = \frac{q}{s}, \quad \text{with } s = \gcd_{y \in G}\left(\frac{q_y}{L_y}\right). \tag{8}$$

The existence of a repetition vector is a necessary condition for bounded memory execution (consistency) but is not sufficient to guarantee the existence of a valid static schedule (liveliness). To check if such a schedule with repetition vector q actually exists for a consistent (C)SDF graph, [2, 11] propose the construction of a single-processor schedule for one iteration, that is, one in which each actor A fires at least q_A^b times.

3. USING CSDF TO MODEL IMPLEMENTATIONS

The implementation of an application can be represented as a directed task graph [14] consisting of tasks communicating through FIFO buffers with fixed capacity, called regular channels (see Figure 1(a)). Only containers, communication
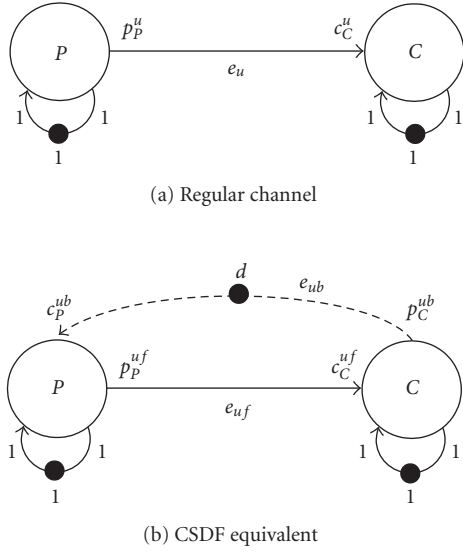
(a) Regular channel

(b) CSDF equivalent

FIGURE 1: The feedback edge e_{ub} limits the size of edge e_u to d.

units holding a fixed amount of data, are communicated over these FIFOs. These containers can be free or completed. Note the difference with a dataflow model where a token can represent a container or just synchronization. Tasks have production and consumption sequences and can only start if sufficient completed containers are present on its input FIFOs and sufficient free containers are available in its output FIFOs. More specifically, executing a task consists of the following steps: (i) acquire: check the availability of the completed input containers and free output containers, (ii) execute the code of the function describing the task behavior (accessing the data in the container), and (iii) release: signal the completion of the production of the output containers and the finishing of the consumption of the input containers. The elapsed time between the successful acquiring and releasing in a task execution is bounded by the worst-case response time, known at design time. Finally, it is assumed that at most one instance of a task can execute at any time. This is important when the task keeps an internal state with data that is needed during a next execution and to maintain the FIFO ordering of the containers.

In a real implementation, also other communication types than the regular channel are deployed, often to optimize the data transfer. Examples are a sliding window for data reuse or a shared buffer with multiple consuming tasks. Such communication types are called special channels. The next subsections describe how the regular channel and which types of special channels can be expressed with a CSDF graph. Their CSDF representation is essential to be able to use the design time analysis techniques of CSDF.

3.1. Blocking write and blocking read

In the modeling of such an implementation task graph as a CSDF graph, a task corresponds to an actor with a response

time equal to the task's worst-case response time. The acquire and release of containers in the implementation are, respectively, represented by the removal and arrival of tokens on the edges in the CSDF model. While a container is always represented by tokens in the dataflow model, the inverse is not necessarily true, as tokens can also express synchronization only. For example, a selfcycle on each actor models that no two instances of a task can execute simultaneously.

The blocking read behavior of a FIFO queue (i.e., the stalling of the consuming task because the queue is empty) is modeled by the data-driven operation of the actors. Because of the fixed depth of the FIFO queue, it also has a blocking write: the producing task is halted as long as the FIFO is full. This blocking read and blocking write behavior can be represented by a pair of queues in opposite direction [1, 6] in the CSDF graph (see Figure 1(b)). The tokens on the forward queue e_{uf} (from producer P to consumer C) represent completed containers while the tokens on the feedback queue e_{ub} indicate the free containers. The fixed size of the FIFO buffer (i.e., its depth expressed as a number of containers it can maximally hold) is modeled by the number of initial tokens d on e_{ub} for an initially empty FIFO.

The tight coupling between the tokens and the containers is expressed by requiring that a producing or consuming task releases at the end of the task execution all containers acquired at the start of the task invocation,

$$\forall i, j \in \mathbb{N} : p_P^{uf}(i) = c_P^{ub}(i), \quad c_C^{uf}(j) = p_C^{ub}(j). \qquad (9)$$

Consuming c_C^{uf} tokens from e_{uf} releases the corresponding containers, but only at the end of the firing with the production of the same number of tokens p_C^{ub} on e_{ub}. To produce p_P^{uf} tokens representing completed containers at the end, the same number c_P^{ub} of them is consumed at the start of the firing, expressing the acquiring of the containers. Consequently, the tokens on the two edges represent correctly how the containers are used in the task graph: acquiring at the start of the execution and releasing at the end of the execution.

Note that the presence of a selfcycle with one initial token is assumed but not drawn in the following CSDF graphs of this text.

3.2. Decoupling tokens from containers

The tight coupling of tokens and containers in a regular channel represents the most common interpretation of the behavior of an edge in a dataflow model: a container is released from/to the edge after a single firing. Figure 2 illustrates the data reuse in the overlapping regions of the search area data during the motion estimation of a video encoder [16]. Such sliding window behavior cannot be modeled with the common CSDF interpretation since the complete dashed search area is required as firing condition and consequently, it will be released entirely from the edge after the first execution of the motion estimation task.

FIGURE 2: Data reuse in the overlapping regions of the search area data for motion estimation.

(a) Special channel

(b) CSDF equivalent

FIGURE 3: Nondestructive reads between a producer P with period L_P and production sequence $p = \{p_P^u(0), \ldots, p_P^u(L_P - 1)\}$ and a consumer C with period L_C and sequences $r = \{r_C^u(0), \ldots, r_C^u(L_C - 1)\}$ and $c = \{c_C^u(0), \ldots, c_C^u(L_C - 1)\}$ for which $c_C^u(j) \leq r_C^u(j)$.

Similarly, the production of a container over multiple task executions cannot be expressed in the common CSDF interpretation as the acquired containers at the start are released to the consuming task at the end of the same invocation. Finally, edges represent point-to-point communication, hindering the expression of shared containers between multiple tasks.

Relaxing the requirement in (9) allows breaking this tight relation between tokens and containers and enables the modeling of special data communication. During a firing of the producer, the number of produced tokens p_P^{uf} on e_{uf} can differ from the number of consumed tokens c_P^{ub} from e_{ub}. Similarly, a consumer firing can consume a different number of tokens from e_{uf} than the number produced on e_{ub}.

In the example of Figure 2, this decoupling of tokens and containers allows releasing only the left, nonoverlapping part of the search area (p_C^{ub}), while the complete search area was required to enable the execution of the motion estimation (c_C^{uf}), with $p_C^{ub} < c_C^{uf}$. The next subsection discusses the behavior of this special channel and other types (dealing with the other restrictions listed above) in detail.

Bounded memory condition

To maintain bounded memory execution, during one period of the producing task, the sum of acquired containers at the producer should equal the sum of completed containers (first equality of (10)). Similarly, during one period of the consumer, the sum of released containers has to equal the sum of consumed completed containers (second equality of (10)).

$$P_P^{uf}(L_P) = C_P^{ub}(L_P), \qquad C_C^{uf}(L_C) = P_C^{ub}(L_C). \qquad (10)$$
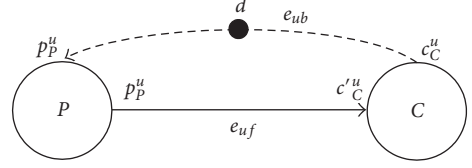
Mutual exclusiveness condition

Additionally, at any moment at the producing task, the sum of completed containers should not be larger than the sum of acquired containers to avoid writing in a nonfree container.

$$\forall k \in \mathbb{N}_0 : C_P^{ub}(k) \geq P_P^{uf}(k). \qquad (11)$$
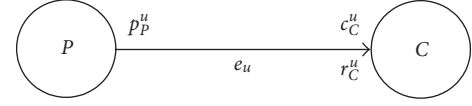
Data preservation condition

Similarly at any moment at the consuming task, the sum of released containers should not be larger than the sum of acquired new containers to avoid loss of data.

$$\forall k \in \mathbb{N}_0 : P_C^{ub}(k) \leq C_C^{uf}(k). \qquad (12)$$

The number of free containers f in the buffer of edge e_u after k firings of P and l firings of C is

$$f = d - C_P^{ub}(k) + P_C^{ub}(l). \qquad (13)$$

3.3. Modeling special channels

Using the decoupling of tokens and containers, the following subsections present some interesting cases of modeling special behavior on edges of the task graph. For each of these special channels, a CSDF equivalent is given when possible. If the equivalent exists, the special channel becomes a shorthand notation for the CSDF graph.

3.3.1. Nondestructive read

An edge e_u with nondestructive reads (see Figure 3(a)) allows a consuming task C to acquire during its $(j + 1)$th invocation $r_C^u(j)$ containers of which only $c_C^u(j)$ containers are released, with

$$\forall j \in \mathbb{N} : r_C^u(j) \geq c_C^u(j). \qquad (14)$$

This special channel enables data reuse: the same container is accessed over multiple invocations of the same task. Because this container remains available on the special channel, the number of acquired containers $r_C^u(j)$ consists of a number of reused containers and a number of additionally acquired containers. Note that during the first task invocation, all acquired containers are additionally acquired containers.

The number of containers $r(j)$ that is reused from the current invocation j during the next task execution $j + 1$

is obtained with (15) as the difference between the number of acquired containers and the number of released containers. When the number of acquired containers $r_C^u(j)$ is smaller than the number of reused containers $r(j-1)$ from the previous invocation, this equation calculates $r(j)$ recursively,

$$r(j) = \begin{cases} r_C^u(0) - c_C^u(0) & \text{if } j = 0, \\ r_C^u(j) - c_C^u(j) & \text{if } j > 0, \ r_C^u(j) > r(j-1), \\ r(j-1) - c_C^u(j) & \text{otherwise.} \end{cases} \tag{15}$$

To avoid an accumulation of containers in the channel that would lead to unbounded memory requirements (i.e., an inconsistent graph), the sum of additionally acquired containers during a repetition of the task should equal the number of released containers (bounded memory condition of (10)). This requires that the number of reused containers of the last firing of the repetition (q_C) is zero. Consequently, at least all reused containers $r(q_C - 2)$ of the one but last firing of the repetition should be acquired, and all acquired containers need to be released:

$$r_C^u(q_C - 1) = c_C^u(q_C - 1) \geq r(q_C - 2). \tag{16}$$

Proof of (16). In order to prove (16), both cases of (15) are considered for $j = (q_C - 1) > 0$ while requiring that $r(q_C - 1) = 0$.

(1) When $r_C^u(q_C - 1) > r(q_C - 2)$ with $r(q_C - 1) = 0$ in (15),

$$c_C^u(q_C - 1) = r_C^u(q_C - 1). \tag{17}$$

(2) When $r_C^u(q_C - 1) \leq r(q_C - 2)$ with $r(q_C - 1) = 0$ in (15),

$$c_C^u(q_C - 1) = r(q_C - 2). \tag{18}$$

Combining this with (14),

$$r_C^u(q_C - 1) \leq c_C^u(q_C - 1),$$
$$r_C^u(q_C - 1) \geq c_C^u(q_C - 1) \implies r_C^u(q_C - 1) = c_C^u(q_C - 1). \tag{19}$$

Overall,

$$r_C^u(q_C - 1) = c_C{}^u(q_C - 1) \geq r(q_C - 2). \tag{20}$$

$\square$

The above condition on the last firing of the repetition also applies to the last firing of the actor period, or

$$r_C^u(L_C - 1) = c_C{}^u(L_C - 1) \geq r(L_C - 2). \tag{21}$$

This condition can sometimes be met by setting the actor period appropriately. In video processing for instance, extending the actor period from a row basis to a frame basis allows the correct releasing of all reused containers at the frame border, when no data reuse dependencies exist between frames.

Figure 3(b) shows how this data reuse behavior is expressed in CSDF using the decoupling of tokens and containers. Only containers that are no longer reused are released as indicated by the production $p_C^{ub} = c_C^u$ on the feedback edge e_{ub}. The forward edge e_{uf} assures the correct synchronization between the actors P and C.

The number c_C^{uf} on this forward edge expresses the number of additionally acquired containers $c_C'^u$, that is, the required number of new completed containers. $c_C^{uf} = c_C'^u$ is calculated in (22) so that actor C can only start firing j if the sum of reused containers $r(j-1)$ and additionally acquired containers $c_C'^u(j-1)$ at least equals $r_C^u(j)$,

$$c_C^{uf} = c_C'^u(j) = \begin{cases} r_C^u(0) & \text{if } j = 0, \\ r_C^u(j) - r(j-1) & \text{if } j > 0, \ r_C^u(j) > r(j-1), \\ 0 & \text{otherwise.} \end{cases} \tag{22}$$

Of the bounded memory, mutual exclusiveness and data preservation conditions (see (10), (11), (12)) of the special channel, only those at the consumer side need to be checked. The ones at the producer are automatically fulfilled as $p_P^{uf} = c_P^{ub}$ (since the producer behavior is like a regular channel).

Proof of the requirements in (12) *and* (10). The data preservation condition of (12) becomes

$$P_C^{ub}(l) \leq C_C^{uf}(l) \implies C_C^u(l) \leq C_C'^u(l). \tag{23}$$

In order to use (22), two cases are distinguished as follows.
(1) $r_C^u(l-1) > r(l-2)$

$$C_C^u(l) \leq C_C'^u(l),$$
$$C_C^u(l) \leq C_C'^u(l-1) + c_C'^u(l-1). \tag{24}$$

Using (22) to replace $c_C'^u(l-1)$,

$$C_C^u(l) \leq C_C'^u(l-1) + r_C^u(l-1) - r(l-2). \tag{25}$$

If $r_C^u(j) \leq r(j-1)$ for $l - x < j < l - 1$ and $x > 1$, then according to (15), $r(l-2) = r_C^u(l-x) - \sum_{j=2}^{x} c_C^u(l-j)$ and according to (22), $c_C'^u(j) = 0$ making $C_C'^u(l-1) = C_C'^u(l-x+1)$,

$$C_C^u(l) \leq C_C'^u(l-x+1) + r_C^u(l-1) - r_C^u(l-x) + \sum_{j=2}^{x} c_C^u(l-j),$$
$$C_C^u(l-x) + c_C^u(l-1) \leq C_C'^u(l-x+1) + r_C^u(l-1) - r_C^u(l-x). \tag{26}$$

With $c_C'^u(l-x) = r_C^u(l-x) - r(l-x-1)$,

$$C_C^u(l-x) + c_C^u(l-1) \leq C_C'^u(l-x) + r_C^u(l-1) - r(l-x-1). \tag{27}$$

If $r^u_C(j) \leq r(j-1)$ for $l-y < j < l-x-1$ and $y > x$, then $c'^u_C(j) = 0$ and $r(l-y-1) = r^u_C(l-y) - \sum^y_{j=x+1} c^u_C(l-j)$,

$$C^u_C(l-y)+c^u_C(l-1) \leq C'^u_C(l-y)+r^u_C(l-1) - r(l-y-1). \tag{28}$$

Assume that $l - y - 1 = 0$,

$$c^u_C(0) + c^u_C(l-1) \leq c'^u_C(0) + r^u_C(l-1) - r(0). \tag{29}$$

With $r(0) = r^u_C(0) - c^u_C(0)$ (see (15)),

$$c^u_C(0) + c^u_C(l-1) \leq r^u_C(0) + r^u_C(l-1) - \left(r^u_C(0) - c^u_C(0)\right),$$
$$c^u_C(l-1) \leq r^u_C(l-1). \tag{30}$$

(2) $r^u_C(l-1) \leq r(l-2)$

$$C^u_C(l) \leq C'^u_C(l). \tag{31}$$

If $r^u_C(j) \leq r(j-1)$ for $l-x < j \leq l-1$ with $x > 1$, according to (15), $r(l-1) = r^u_C(l-x) - \sum^x_{j=1} c^u_C(l-j)$ and according to (22), $c'^u_C(j) = 0$ making $C'^u_C(l) = C'^u_C(l-x+1)$,

$$C^u_C(l) \leq C'^u_C(l-x+1),$$
$$C^u_C(l) \leq C'^u_C(l-x) + c'^u_C(l-x). \tag{32}$$

Using (22) to replace $c'^u_C(l-x)$,

$$C^u_C(l) \leq C'^u_C(l-x) + r^u_C(l-x) - r(l-x-1). \tag{33}$$

With $r^u_C(l-x) = r(l-1) + \sum^x_{j=1} c^u_C(l-j)$ (see above),

$$C^u_C(l) \leq C'^u_C(l-x) + r(l-1) + \sum^x_{j=1} \left(c^u_C(l-j)\right) - r(l-x-1),$$
$$C^u_C(l-x) \leq C'^u_C(l-x) + r(l-1) - r(l-x-1). \tag{34}$$

If $r^u_C(j) \leq r(j-1)$ for $l-y < j \leq l-x-1$ and $y > x$, then $c'^u_C(j) = 0$ and $r(l-y-1) = r^u_C(l-y) - \sum^y_{j=x+1} c^u_C(l-j)$,

$$C^u_C(l-y) \leq C'^u_C(l-y) + r(l-1) - r(l-y-1). \tag{35}$$

Assume that $l - y - 1 = 0$,

$$c^u_C(0) \leq c'^u_C(0) + r(l-1) - r(0). \tag{36}$$

With $c'^u_C(0) = r^u_C(0)$ (see (22)),

$$c^u_C(0) \leq r^u_C(0) + r(l-1) - r(0). \tag{37}$$

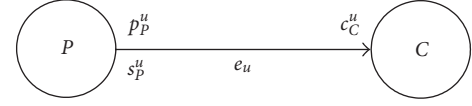With $r(0) = r^u_C(0) - c^u_C(0)$ (see (15)),

$$0 \leq r(l-1). \tag{38}$$

To check the bounded memory condition of (10), L_C firings are considered or $l = L_C$
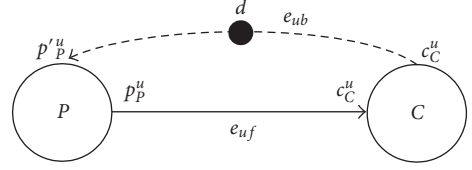
$$C^u_C(L_C) = C'^u_C(L_C). \tag{39}$$

Because of (21), $r^u_C(L_C-1) \geq r(L_C-2)$. This matches the first case of the proof above. Substituting l by L_C and replacing the inequality by an equality yields

$$c^u_C(L_C - 1) = r^u_C(L_C - 1). \tag{40}$$

This is true because of (21). $\qquad\square$

(a) Special channel

(b) CSDF equivalent

FIGURE 4: Partial updates between a producer P with period L_P and sequences $p = \{p^u_P(0), \ldots, p^u_P(L_P - 1)\}$ and $s = \{s^u_P(0), \ldots, s^u_P(L_P - 1)\}$ for which $p^u_P(i) \leq s^u_P(i)$ and a consumer C with period L_C and sequence $c = \{c^u_C(0), \ldots, c^u_C(L_C - 1)\}$.

3.3.2. Partial update

An edge e_u with partial updates (see Figure 4(a)) allows the acquiring of $s^u_P(i)$ containers by the producing task during the $(i+1)$th invocation of which only $p^u_P(i)$ containers are full and released at the end of the task execution, with

$$\forall i \in \mathbb{N} : s^u_P(i) \geq p^u_P(i). \tag{41}$$

This enables the production of data in a container over multiple invocations. Because this container remains available on the special channel, the number of acquired containers $s^u_P(i)$ consists of a number of uncompleted containers and a number of additionally acquired containers. Note that during the first task invocation, all acquired containers are additionally acquired containers. An example of partial updating is a task that completes the data in a container over 2 invocations: data on the even positions is written during the first execution, while the data on the odd positions is produced during the second execution.

The number of uncompleted containers $s(i)$ in task invocation i that are continued during the next invocation $i+1$ is calculated with (42) as the difference between the number of acquired containers and the number of completed containers. When the number of acquired containers $s^u_P(i)$ is smaller than the number of reused containers $s(i-1)$ from the previous invocation, this equation calculates $s(i)$ recursively,

$$s(i) = \begin{cases} s^u_P(0) - p^u_P(0) & \text{if } i = 0, \\ s^u_P(i) - p^u_P(i) & \text{if } i > 0, \ s^u_P(i) > s(i-1), \\ s(i-1) - p^u_P(i) & \text{otherwise.} \end{cases} \tag{42}$$

To avoid the loss of partially produced data, the number of containers acquired during the last invocation has to include the remaining uncompleted ones from the previous executions(s) (calculated with (42)) and all of them need to be released

$$s^u_P(n-1) = p^u_P(n-1) \geq s(n-2). \tag{43}$$

Similar to the nondestructive read, this condition can sometimes be met by setting the actor period appropriately. If this is not possible, the channel is misused as scratchpad. Such temporal data should be stored in a local buffer of the task.

The partial update behavior is represented in Figure 4(b) using the decoupling of tokens and containers. Only the completed containers are released to be used by the consumer, as indicated by the production $p_P^{uf} = p_P^u$ on the forward edge e_{uf}. Consequently, this edge e_{uf} synchronizes the producer and the consumer. Equation (44) makes sure that the sum of uncompleted containers $s(i-1)$ and additionally acquired containers $p_P^{ub} = p'_P^u(i)$ at least equals the number of acquired containers $s_P^u(i)$ for data production during firing i,

$$c_P^{ub} = p'_P^u = \begin{cases} s_P^u(0) & \text{if } i = 0, \\ s_P^u(i) - s(i-1) & \text{if } i > 0, \ s_P^u(j) > s(i-1), \\ 0 & \text{otherwise.} \end{cases}$$

(44)

Of the bounded memory, mutual exclusiveness and data preservation conditions (see (10), (11), (12)) of the special channel, only the ones at the producer need to be checked. The conditions at the consumer are automatically fulfilled as $c_C^{uf} = p_C^{ub}$. The proof is similar to the nondestructive read one.
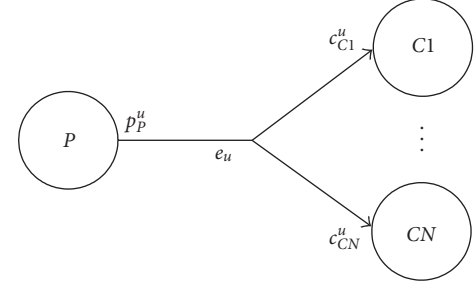
3.3.3. Multiple consumers

An edge e_u with multiple consumers (see Figure 5(a)) allows N consuming tasks $C1 \cdots CN$ to consume the same containers produced by a task P. Each consumer Cy can have its own actor period L_{Cy} as long as there exists a solution for their combined balance equations in (45) to obey the consistency condition,

$$r_P \cdot P_P^u(L_P) = \begin{cases} r_{C1} \cdot C_{C1}^u(L_{C1}), \\ \vdots \\ r_{CN} \cdot C_{CN}^u(L_{CN}). \end{cases}$$

(45)

A multiple consumer edge works with a composed consume: a container can only be released at the consume side if all actors $C1 \cdots CN$ have released this container. Equation (46) calculates the composed consume $cc^u(j_c)$ after l_y firings of the tasks Cy (with $1 \le y \le N$). The index j_c counts the composed consumes by incrementing j_c whenever a consuming task Cy executes. To make sure all consumers no longer need the container(s), this equation looks for the consuming task with the minimum sum of consumed containers and subtracts the sum of previously composed consumed containers,

$$cc^u(j_c) = \min_{1 \le y \le N} \left(C_{Cy}^u(l_y) \right) - C_{cc}^u(j_c), \quad \text{with } j_c = \left(\sum_{y=1}^N l_y \right) - 1.$$

(46)

(a) Special channel
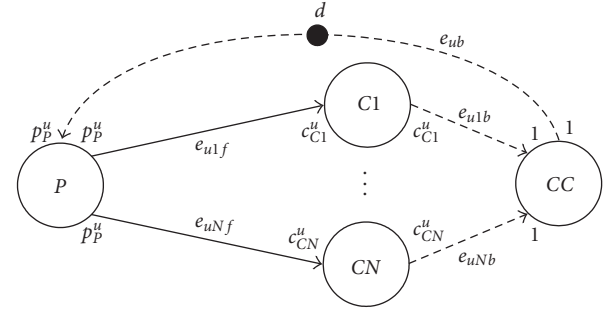
(b) CSDF equivalent

Figure 5: Multiple consumers on an edge between a producer P with period L_P and sequence $p = \{p_P^u(0), \dots, p_P^u(L_P - 1)\}$ and N consumers $C1, \cdots, CN$ with periods $L_{C1}, \dots, L_{CN}$ and sequences $c1 = \{c_{C1}^u(0), \dots, c_{C1}^u(L_{C1} - 1)\}, \dots, cN = \{c_{CN}^u(0), \dots, c_{CN}^u(L_{CN} - 1)\}$.

Such a multiple consumer edge is represented in CSDF using the decoupling of tokens and containers in Figure 5(b). On each of the N forward edges e_{uyf}, the same number of tokens p_P^u representing the available completed containers is produced during a firing of the producer. The number of tokens consumed from these forward edges can vary for the N consumers, including the consume sequence length, as long as the balance condition of (45) is met. The composed consume is modeled by the CC actor with a zero response time. Only when all consuming actors have released a container, it is made available as free container on the backward edge e_{ub}.

As the size of the container, buffer d is shared over all edges, the number of free containers f (in the shared buffer) equals the number of initially free containers d decreased with the number of acquired containers after k firings of the producer and incremented with the number of composed consumed containers after l_c composed consumptions,

$$f = d - C_P^{ub}(k) + C_{CC}^u(l_c).$$

(47)

Using (46), $C_{CC}^u(l_c)$ can be rewritten and the number of free containers f becomes

$$f = d - C_P^{ub}(k) + \min_{1 \le y \le N} \left(P_{Cy}^{uyb}(l_y) \right),$$

(48)

where the minimum over all edges assures the containers remain available until the last consumer has released them.
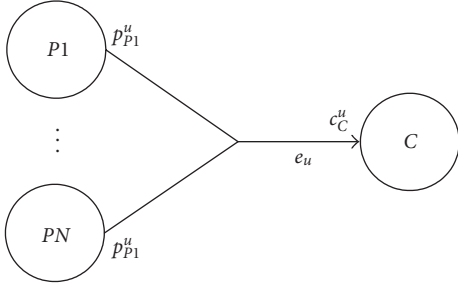
FIGURE 6: The multiple producers special channel with producers $P1, \ldots, PN$ has no CSDF equivalent as the token order depends on the response time.

(a) Special channel

(b) Expressed as multiple consumers with non-destructive reads

FIGURE 7: Special case of the multiple consumers with nondestructive read: a nondestructive read-back at the producer side.

FIGURE 8: Notation of a global buffer.

FIGURE 9: Some actors do not fire concurrently due to the schedule or the graph topology.

The bounded memory, mutual exclusiveness conditions (see (10), (11)) of the special channel are met as for all edges $p_P^{uyf} = c_P^{ub}$, $c_C^{uyf} = p_C^{uyb}$ and the CC actor has all ones as consumption and production rates. The data preservation condition (12) is satisfied because the composed consume can only lead to a later releasing of a container that was still needed by another consuming task.

3.3.4. Multiple producers

An edge e_u with multiple producers (see Figure 6) allows N producing tasks $P1 \cdots PN$ to produce containers. This special channel has no CSDF equivalent, as the token arrival depends on the actual response time of the producer, leading to nondeterministic behavior. Consequently it is invalid.

Multiple producers with partial updates on a single edge would allow these tasks to produce their part of the token. Still, this is equivalent to separate edges between the producers and the consumers and does not offer the protection of the data that is produced like in the equivalent.

3.3.5. Combinations

All valid previous special channels can be combined, like an edge with partial updates and nondestructive reads, an edge with partial updates and multiple consumers, and so forth. An interesting combination is multiple consumers with nondestructive reads as it allows a producing task P to read previously produced containers back (see Figure 7(a)) by considering the producer also as a consumer on the same special channel (see Figure 7(b)).

3.4. Other implementation aspects

All special channels described above represent a synchronizing communication. The implementation of an application can also use nonsynchronizing communication, to pass for instance parameters or if synchronization becomes obsolete when tasks never execute concurrently due to ordering constraints.
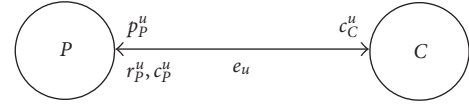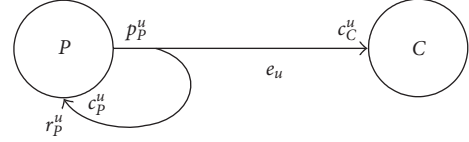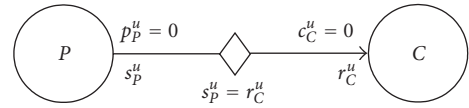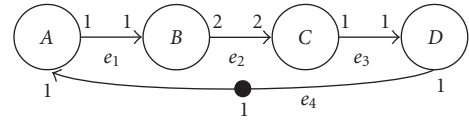
3.4.1. Global parameters

Global parameters are used in an implementation to pass the most recent settings to a task. Through a global buffer with an updating mechanism, the consuming tasks only see the new parameters when the producer completed the new data in a container. The nonsynchronizing behavior of such a communication (see Figure 8) and its dynamic consumption and production pattern cannot be modeled in CSDF.

On the other hand, these global parameters do not influence the temporal behavior (since they are a form of nonsynchronizing communication) nor need to be considered during the buffer capacity calculation as their size is fixed at design time (depending on the number and the size of the parameters).

3.4.2. Serialized actors

In some cases, actors will never fire concurrently due to ordering constraints, either in their schedule or in the graph topology. The schedule ordering constraint can also be represented in the graph by adding an edge to indicate this. In Figure 9 actors A, B, C, and D can only fire sequentially due

to the graph topology. A schedule ordering constraint (e.g., a sequential schedule A, B, C, D) of the same graph but without edge e_4 can be represented by adding edge e_4. Using a global buffer allows the sharing of container space between such serialized actors. In the literature, this approach is combined with lifetime analysis for memory optimized software synthesis [17, 18].

4. BUFFER CAPACITY CALCULATION

The (minimum) buffer capacities d are calculated at design time by manually constructing a (desired) static periodic schedule and combining this with a life-time analysis of the tokens using the worst-case actor response times. The schedule needs to cover at least a complete iteration in the periodic phase. As a result, it is constructed from the start and also includes the transient phase before reaching the periodic phase. As no dead-lock is allowed in this periodic schedule to assure the liveliness of the graph, the minimum buffer size is found if the number of free tokens f on the feedback edge is zero when the difference between the total number of consumed and produced tokens on this edge reaches a maximum. The buffer capacity d_u of edge e_u is derived from (48), the generic case for the all valid special channels, by setting f to zero and considering the life-time analysis from start until one period in steady state (periodic phase) is completed. Assuming the desired schedule reaches the periodic phase after k_{SS} firings of the producer P and $l_{y,SS}$ firings of the consumers Cy

$$d_u = \max_{0 \leq k < k_{SS}+q_P^b; \, 0 \leq l_y < l_{y,SS}+q_{Cy}^b} \left(C_P^{ub}(k) - \min_{1 \leq y \leq N} \left(P_{Cy}^{uyb} \right)(l_y) \right). \tag{49}$$

The throughput of the constructed static schedule relates to μ^{-1}, with μ being the iteration period (or total execution time of one period) of this periodic schedule. The temporal monotonic behavior guarantees that moving to a selftimed execution after the buffer sizing yields an implementation with at least this throughput.

Practically, the life-time analysis monitors the number of tokens on the forward and the backward edge of all edges e_u in the CSDF graph G: the forward one for the evaluation of the firing condition, the backward one for the buffer capacity calculation. Consequently, the evaluation $P_P^{uyf}(k) - C_C^{uyf}(l_y)$ on e_{uyf} is made at the end of each firing of its producer or consumer. The evaluation $C_P^{uyb}(k) - P_C^{uyb}(l_y)$ on e_{uyb} is made at the start of each firing of its producer or consumer. The maximum over all e_{uy} during the transient phase and one iteration period in the periodic phase of the desired schedule yields the buffer size d_u.

The formula for d_u (see (49)) and the practical approach presented above only provide a basic buffer sizing technique to find the minimum buffer capacity for the given desired schedule. For an efficient multiprocessor implementation, four related elements need to be considered in the tradeoff:

(a) Example nondestructive read FIFO
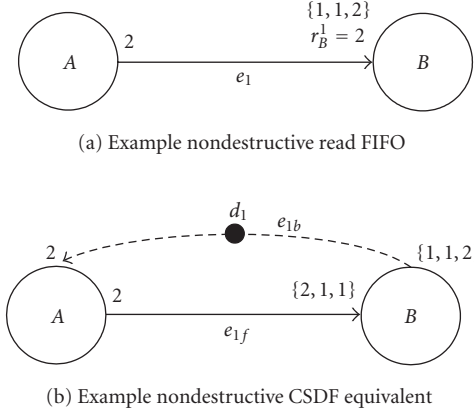

(b) Example nondestructive CSDF equivalent

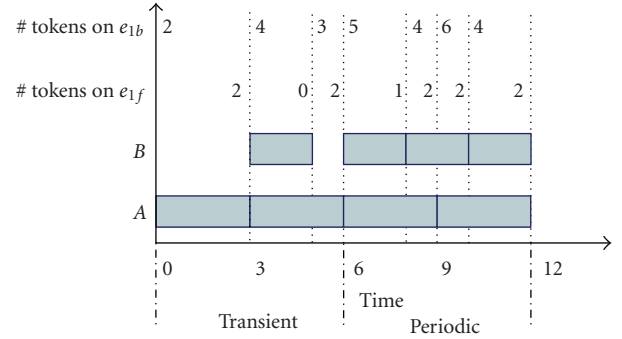Figure 10: Example nondestructive read keeping one container for data reuse.

Figure 11: Schedule and life-time analysis of the buffer capacity.

throughput, response times, schedule settings, and buffer capacities. Optimization algorithms exploring these tradeoffs are outside the scope of this paper.

Example 1. Consider the nondestructive read edge of Figure 10(a) with its CSDF equivalent in Figure 10(b). The basic repetition vector q^b is calculated from the topology matrix Γ and the actor periods. Assume the worst case response times are known, $RT_A = 3$ and $RT_B = 2$ and the desired schedule is a pipelined parallel operation of both actors,

$$\Gamma = \begin{pmatrix} 2 & -4 \end{pmatrix}; \quad L = \begin{pmatrix} 1 & 3 \end{pmatrix}; \quad r = \begin{pmatrix} 2 & 1 \end{pmatrix}; \quad q = q^b = \begin{pmatrix} 2 & 3 \end{pmatrix}. \tag{50}$$

The corresponding schedule with the lifetime analysis on the edges e_{1f} and e_{1b} is shown in Figure 11. The number of tokens on e_{1f} is calculated at the end of a firing of one of the actors while the number of tokens on edge e_{1b} is calculated at the start of a firing. The desired schedule reaches steady state (periodic phase) at time 6 and one period has $q_A^b = 2$ firings of actor A and $q_B^b = 3$ firings of actor B. This period
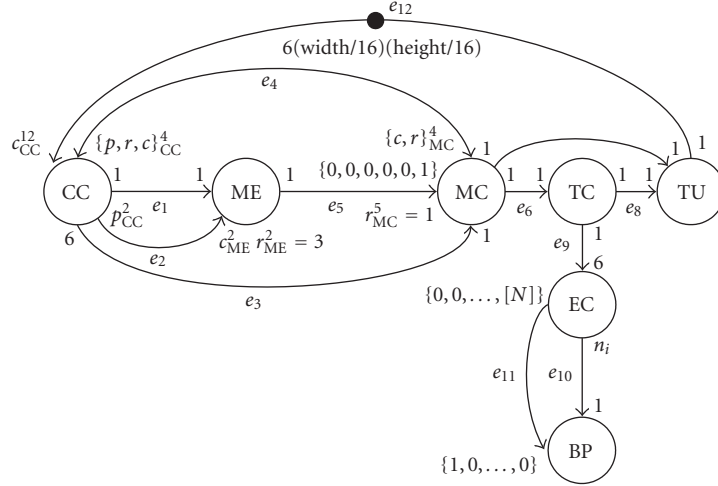
FIGURE 12: CSDF graph representing the partitioning of the MPEG-4 part 2 SP encoder scheme.

TABLE 1: Detailed information of the actors in the encoder CSDF graph.

Actor name	Acronym	Functionality	Actor period
Copy control	CC	Fill the memory hierarchy and the new video inputs	(width/16)(height/16)
Motion estimation	ME	Find the motion vectors	width/16
Motion compensation	MC	Get predicted block and calculate error	6(width/16)(height/16)
Texture coding	TC	Transform, quantization, and inverse	1
Texture update	TU	Add and clip compensated and predicted blocks	1
Entropy coding	EC	AC/DC, MV prediction and VLC coding	m
Bitstream packetization	BP	Add headers and compose the bitstream	N

contains 6 time units. The required buffer capacity for the desired schedule is 6 (the maximum on the # tokens on e_{1b} line).

5. MPEG-4 PART 2 VIDEO ENCODER EXAMPLE

To illustrate the expressiveness of a CSDF graph when tokens are decoupled from containers, an MPEG-4 part 2 video encoder [19] is presented as a case study. The constructed dataflow graph (see Figure 12) supports the partitioning phase of the implementation of a low-power, fully dedicated MPEG-4 part 2 encoder [20]. When the behavior of the data communication between two actors cannot be expressed by regular CSDF edges, special channels are inserted. In the video encoder example, this happens to maintain the effect of high-level memory optimizations, like data-reuse and the sharing of local buffers.

The dataflow graph is a combination of a CSDF graph with compile time parameters related to the maximum supported resolution (width × height) and a DDF part after the entropy coding (EC). The meaning of the variables m and N in the graph relate to this dynamic behavior and will be explained later. The regular and special channels are used in

TABLE 2: Production and consumption sequences instantiated for a resolution of 80×48 pixels.

Symbol	Sequence
p_{CC}^2	$\{3, 1, 1, 1, 1\}$
c_{ME}^2	$\{1, 1, 1, 1, 3\}$
p_{CC}^4	$\{7, 1, 1, 1, 0, 2, 1, 1, 1, 0, 0, 0, 0, 0, 0\}$
c_{MC}^4	$c(j \div 6)$ with $c = \{0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 0, 1, 1, 1, 7\}$
r_{MC}^4	$r(j \div 6)$ with $r = \{7, 8, 9, 10, 10, 11, 6, 6, 6, 11, 7, 6, 6, 6, 6\}$
r_{CC}^4	$\{6, 3, 4, 5, 10, 11, 6, 6, 6, 11, 7, 6, 6, 6, 6\}$
c_{CC}^4	$\{0, 0, 0, 0, 0, 2, 1, 1, 0, 1, 2, 1, 1, 0, 6\}$
c_{CC}^{12}	$\{42, 6, 6, 6, 0, 12, 6, 6, 6, 0, 0, 0, 0, 0, 0\}$

the dataflow graph as short-hand notation: every drawn edge represents a forward and a backward CSDF edge (to model the bounded buffer sizes). Remember that a selfcycle with one initial token is assumed on every actor.

The proposed dataflow graph (see Figure 12) consists of 7 actors connected by 12 edges numbered e_1 to e_{12}. Three edges, e_2, e_4, and e_5 are special channels. Edge e_2 is a nondestructive read special channel modeling the sliding window with the search area data repetitively accessed by the motion

TABLE 3: Buffer and token size of all edges.

Edge	Name	Buffer size (# of containers)	Container size (words/container)	Container width (bits/word)
e_1	New macroblock	2	256	8
e_2	Search area	6	768	8
e_3	Current macroblock	18	64	8
e_4	Buffer YUV	(width/8) + 5	384	8
e_5	Motion vectors	2	2	12
e_6	Error block	2	64	9
e_7	Compensated block	3	64	8
e_8	Texture block	2	64	12
e_9	Quantized macroblock	12	64	12
e_{10}	Data buffer	$VP_{max} + n_{max}$	1	1
e_{11}	Generate VP	1	1	1
e_{12}	Reconstructed frame	6(width/16)(height/16)	64	8
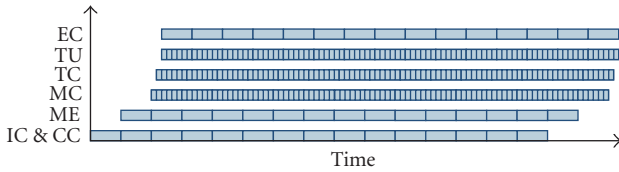
FIGURE 13: MPEG-4 part 2 SP encoder desired schedule with pipelined parallel operation for one frame with resolution of 80×48 pixels.

estimation. Edge e_4 is drawn as a bidirectional edge, as it represents nondestructive read back behavior at the producer side, sharing data between the copy controller and the motion compensation. Edge e_5 is a nondestructive read special channel passing the motion vectors to the motion compensation. These motion vectors are reused for the six blocks of the macroblock. Edge e_{12} is a regular channel with initial tokens (represented by the full dot and the number of initial tokens).

Table 1 details for every actor its full name, functionality, and actor period. The production/consumption sequences reflect the behavior of the video encoder. They are represented as compactly as possible in Figure 12 due to the long actor periods: (i) if the sequence contains a repeated pattern, only this pattern is listed and (ii) a symbolic representation is used if the cycle of the sequence spans more than 6 phases. As these symbols are a function of the compile time parameters width and height, they are instantiated for a maximum resolution of width = 80 and heigth = 48 in Table 2. Note that even this small resolution results in a sequence period of 90 for c_{MC}^4 and r_{MC}^4. A short notation is used for them. The real design [20] for which the CSDF graph is built has a supported resolution of 704×576.

The number of bits generated by the entropy coder varies depending on the type of sequence and the quantization degree (DDF). Edges e_{10} and e_{11} cooperate in a special way to deal with this. The compressed information is accumulated on edge e_{10} with the number of bits n_i varying per firing of the actor EC. When the size of a video packet is reached during the mth firing, the number of bits $N = \sum_{i=0}^{m-1} n(i)$ accumulated on edge e_{10} is written on edge e_{11} (noted as $[N]$ in the produce sequence, representing the value of the single token). Once this is completed, actor BP can fire and consumes 1 token from edge e_{11} containing a scalar with the total number of tokens to consume from edge e_{10}, resulting in N firings of BP that consume 1 token from e_{10}. As the maximum number of bits allowed in a video packet (VP_{max}) is defined by the levels of the MPEG-4 part 2 standard, this edge can be interpreted in worst-case conditions as CSDF to calculate the buffer bound of edge e_{10}.

To maximize the throughput while relaxing the response time requirements for the HW design, the desired schedule for a fully dedicated design is a pipelined and parallel operation (see Figure 13). This sets the goal of the buffer capacity calculation to: find the minimal buffer sizes that maximize the throughput while also maximizing the response times. There are no processing resource constraints as every actor is implemented as a separate hardware accelerator. Under those circumstances, the worst-case actor RT equals its critical RT, defined as

$$\mathrm{RT}_A^{\mathrm{crit}} = \frac{\mu}{q_A^b} \qquad (51)$$

and directly relates to the throughput required in the specification through the iteration period μ of the desired pipelined parallel schedule. The practical technique of the previous section now has the necessary givens for the life-time analysis of the edges. The resulting buffer sizes are summarized in Table 3, together with their name, their container size, the width of an element in a container, and the communication primitive type that is selected for the hardware implementation [20].

6. CONCLUSIONS

The CSDF model of computation matches in many cases well with the dataflow dominated behavior of multimedia

processing, making it a good abstraction means to reason about the parallelism required in an efficient implementation. Among different dataflow models of computation, CSDF is one of the most expressive MoCs while keeping the full analysis potential (e.g., consistency checks, dead-lock analysis, etc.).

This paper shows that implementation specific aspects, like data reuse and shared buffers to improve the efficiency or restricted buffer sizes, can also be expressed in a CSDF graph that is used as an analysis model. Representing the optimized data communication behavior and memory limitations of such special channels, often related to the use of shared circular buffers, by two edges allows the correct modeling of the synchronization and the free buffer space between the communicating tasks. Consequently, the graph remains completely analyzable and allows reasoning about its temporal behavior. Additionally, the special channels are a short-hand notation and a more intuitive representation of this optimized data communication, enriching CSDF with the expression of shared memory aspects.

With worst-case response times and a desired schedule as given, a buffer capacity calculation at design time through a life-time analysis of the CSDF model is presented. The obtained consistent relation between the model and the implementation combined with the temporal monotonic behavior when moving to selftimed execution assures that the throughput of the final implementation is at least the one derived from the iteration period of the desired schedule.

A CSDF graph of an MPEG-4 part 2 video encoder using shared buffers and exploiting reuse is constructed. With this CSDF model, the correct buffer capacities are calculated for a fully dedicated encoder implementation operating as a video pipeline.

REFERENCES

[1] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Dekker, New York, NY, USA, 2000.

[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, 1996.

[3] A. Davare, Q. Zhu, J. Moondanos, and A. Sangiovanni-Vincentelli, "JPEG encoding on the intel MXP5800: a platform-based design case study," in *Proceedings of the 3rd IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTMED '05)*, pp. 89–94, New York, NY, USA, September 2005.

[4] H. Hwang, T. Oh, H. Jung, and S. Ha, "Conversion of reference C code to dataflow model H.264 encoder case study," in *Proceedings of the Asia and South Pacific Design Automation Conference (DAC '06)*, pp. 152–157, Yokohama, Japan, January 2006.

[5] F. Haim, M. Sen, D.-I. Ko, S. S. Bhattacharyya, and W. Wolf, "Mapping multimedia applications onto configurable hardware with parameterized cyclo-static dataflow graphs," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '06)*, vol. 3, pp. 1052–1055, Toulouse, France, May 2006.

[6] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *Proceedings of the 43rd Design Automation Conference (DAC '06)*, pp. 899–904, San Francisco, Calif, USA, July 2006.

[7] C. Park, J. Jung, and S. Ha, "Extended synchronous dataflow for efficient DSP system prototyping," *Design Automation for Embedded Systems*, vol. 6, no. 3, pp. 295–322, 2002.

[8] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: formal models, validation, and synthesis," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366–390, 1997.

[9] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[10] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Resynchronization for multiprocessor DSP systems," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 47, no. 11, pp. 1597–1609, 2000.

[11] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[12] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman, "Task-level timing models for guaranteed performance in multiprocessor networks-on-chip," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '03)*, pp. 63–72, San Jose, Calif, USA, October-November 2003.

[13] M. H. Wiggers, M. Bekooij, P. Jansen, and G. Smit, "Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure," in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, pp. 10–15, Seoul, Korea, October 2006.

[14] M. H. Wiggers, M. Bekooij, P. Jansen, and G. Smit, "Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure," in *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '07)*, pp. 281–292, Bellevue, Wash, USA, April 2007.

[15] J. Teich and S. S. Bhattacharyya, "Analysis of dataflow programs with interval-limited data-rates," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 43, no. 2-3, pp. 247–258, 2006.

[16] I. E. G. Richardson, *H.264 and MPEG-4 Video Compression: Video Coding for Next-Generation Multimedia*, John Wiley & Sons, New York, NY, USA, 2003.

[17] P. K. Murthy and S. S. Bhattacharyya, "Shared buffer implementations of signal processing systems using lifetime analysis techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 2, pp. 177–198, 2001.

[18] H. Oh and S. Ha, "Memory-optimized software synthesis from dataflow program graphs with large size data samples," *EURASIP Journal on Applied Signal Processing*, vol. 2003, no. 6, pp. 514–529, 2003.

[19] "Information technology—generic coding of audio-visual objects—part 2: visual," ISO/IEC 14496-2:2004, June 2004.

[20] K. Denolf, A. Chirila-Rus, and D. Verkest, "Low-power MPEG-4 video encoder design," in *Proceedings of IEEE Workshop on Signal Processing Systems (SIPS '05)*, pp. 284–289, Athens, Greece, November 2005.

Kristof Denolf received the M.Eng. degree in electronics from the Katholieke Hogeschool, Brugge-Oostende, Belgium, in 1998, the M.S. degree in electronic system design from Leeds Metropolitan University, Leeds, UK, in 2000 and is currently a Ph.D. candidate at the Eindhoven University of Technology, The Netherlands. He joined the Multimedia (MM) group of the Nomadic Embedded Systems devision, at the Interuniversity Micro Electronics Centre (IMEC), Leuven, Belgium, in 1998. His main research interests are the cost efficient design of advanced video processing systems and the end-to-end quality of experience.

Marco Bekooij received an M.S.E.E. degree from Twente University of Technology, in 1995 and a Ph.D. degree from the Eindhoven University of Technology, in 2004. He is currently a Senior Researcher at NXP Semiconductors. He has been involved in the design of a channel decoder IC for digital audio broadcasting and a compiler backend for VLIW processors with distributed register files. His current research interest is the design and analysis of predictable multiprocessor systems.

Johan Cockx received his degree in electrical engineering from the Katholieke Universiteit Leuven, Belgium, 1983. From 1983 to 1985 he was a member of the CAD research group at the ESAT laboratory of that university, working on modular circuit simulation. From 1986 to 1996, he worked for Silvar-Lisco, later renamed to EDC, on a wide range of electronic design tools including a schematic editor, the core data structure of DSP station behavioral synthesis tool suite, and a dynamic dataflow simulator. He was an early adopter of object oriented programming techniques in general and the C++ programming language in particular. In 1996, he joined the Design Technology for Integrated and Communication Systems (DESICS) division of the Interuniversity Micro Electronics Center (IMEC), Heverlee, Belgium, where he did research on C++-based concurrent timed simulation of embedded systems (TIPSY—comparable to but preceding SystemC), automated overhead removal from object oriented C++ programs, functional parallelization (SPRINT), translation of C++ code to readable *C* code, and *C* code cleaning for embedded application. He is author/coauthor of two patent applications and several papers on these subjects.

Diederik Verkest received the Master and Ph.D. degrees in applied sciences from the Katholieke Universiteit Leuven (Belgium) in 1987 and 1994, respectively. He has been working in the VLSI design methodology group of the IMEC laboratory (Leuven, Belgium) on several topics related to formal methods, system design, hardware/software codesign, reconfigurable systems, and multiprocessor systems. He is currently in charge of the research at IMEC on design technology for nomadic embedded systems. He is Professor at the University of Brussels (VUB) and at the University of Leuven (KU-Leuven).

He is Member of IEEE and a Golden Core Member of the IEEE Computer Society. He published and presented over 100 articles in International Journals and at International Conferences. Over the past years, he was a member of the programme and/or organization committees of several major international conferences such as ISSS, CODES, FPL, DATE, and DAC. He was the General Chair of the Design, Automation, and Test in Europe Conference, DATE'03.

Henk Corporaal has gained an M.S. degree in theoretical physics from the University of Groningen, and a Ph.D. degree in electrical engineering, in the area of computer architecture, from Delft University of Technology. He has been teaching at several schools for higher education, has been Associate Professor at the Delft University of Technology in the field of computer architecture and code generation, had a Joint Professor appointment at the National University of Singapore, and has been Scientific Director of the joined NUS-TUE Design Technology Institute. He also has been Department Head and Chief Scientist within the DESICS (design technology for integrated information and communication systems) division at IMEC, Leuven (Belgium). Currently Corporaal is Professor in embedded system architectures at the Eindhoven University of Technology (TU/e) in The Netherlands. He has coauthored over 200 journal and conference papers in the (multi)processor architecture and embedded system design area. Furthermore, he invented a new class of VLIW architectures, the transport triggered architectures, which is used in several commercial products. His current research projects are on the predictable design of soft and hard real-time embedded systems.