On Memory Reuse Between Inputs and Outputs of Dataflow Actors

Karol Desnos, Maxime Pelcat, Jean François Nezan, Slaheddine Aridhi

On Memory Reuse Between Inputs and Outputs of Dataflow Actors

KAROL DESNOS, MAXIME PELCAT and JEAN-FRANÇOIS NEZAN, IETR, INSA Rennes
SLAHEDDINE ARIDHI, Texas Instruments France

This article introduces a new technique to minimize the memory footprints of Digital Signal Processing (DSP) applications specified with Synchronous Dataflow (SDF) graphs and implemented on shared-memory Multiprocessor Systems-on-Chips (MPSoCs). In addition to the SDF specification, which captures data dependencies between coarse-grained tasks called actors, the proposed technique relies on two optional inputs abstracting the internal data dependencies of actors: annotations of the ports of actors, and script-based specifications of merging opportunities between input and output buffers of actors. Experimental results on a set of applications show a reduction of the memory footprint by 48% compared to state-of-the-art minimization techniques.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems ; D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms: Design

Additional Key Words and Phrases: Dataflow, memory optimization, buffer merging

ACM Reference Format:
Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi, 2015. On memory reuse between inputs and outputs of dataflow actors. *ACM Trans. Embedd. Comput. Syst.* 15, 2, Article 30 (February 2016), 25 pages.
DOI: http://dx.doi.org/10.1145/2871744

1. INTRODUCTION

Over the last decade, the popularity of data-intensive image and Digital Signal Processing (DSP) algorithms in embedded systems has rapidly grown, with many applications in the automotive [Arndt et al. 2013], the multimedia and the telecommunication domains [Pelcat et al. 2012]. When developing data-intensive applications for embedded systems, addressing the memory challenges is an essential task as it can dramatically impact the quality and performance of a system. Indeed, the silicon area occupied by the memory can be as large as 80% of a chip and may be responsible for a major part of its power consumption [Zorian 2002]. Despite the large silicon area allocated to memory banks, the amount of internal memory available on most embedded Multiprocessor Systems-on-Chips (MPSoCs) is still limited, thus the development of data-intensive applications remains a challenging objective. For example, when executing a computer vision algorithm on high-definition (1080p) RGB video stream, several frames of 2 MPixels may be processed simultaneously, each frame requiring 6 MBytes of memory to be stored in the RGB format. As most modern MPSoCs embed at most a few tens of MBytes, minimizing the memory footprints of such applications has become an essential step in the development of an embedded system, often determining the feasibility of this system.

Author's addresses: K. Desnos (kdesnos@insa-rennes.fr), M. Pelcat (mpelcat@insa-rennes.fr) and J.-F. Nezan (jnezan@insa-rennes.fr), IETR, INSA Rennes, UMR CNRS 6164, UEB, 20 avenue de Buttes de Coësmes, 35708 Rennes, France; S. Aridhi (saridhi@ti.com), Texas Instruments France, 5 chemin des Presses, 4 allée Technopolis, 06800 Cagnes-Sur-Mer, France.

DOI: http://dx.doi.org/10.1145/2871744

This article introduces a new optimization technique to minimize the memory footprint of applications described with a dataflow Model of Computation (MoC). Representing an application with a dataflow graph [Lee and Parks 1995] consists of dividing this application into a set of processing entities, named actors, interconnected by a set of First-In First-Out queues (FIFOs). FIFOs allow the transmission of data tokens between actors. An actor starts its preemption-free execution (that is, it fires) when its incoming FIFOs contain enough data tokens. The number of data tokens consumed and produced during the execution of an actor is specified by a set of firing rules. The possibility of analyzing dataflow graphs due to their natural expressivity of parallelism makes them particularly popular both for research [Lee and Parks 1995; Pelcat et al. 2012] and commercial purposes [Kalray 2013]. Indeed, it is this parallelism that makes dataflow an attractive MoC to fully exploit the computing power offered by MPSoCs [Pelcat et al. 2012], Graphics Processing Units (GPUs), and manycore architectures [Kalray 2013].

In the literature, most memory optimization techniques for dataflow MoCs rely solely on an analysis of the high-level data dependencies specified in dataflow graphs. The memory reuse opportunities exploited by these optimization techniques is thus limited by the high abstraction level considered and by the lack of information about the internal data dependencies of actors. In this article, a new set of annotations for dataflow graphs is introduced to allow the developer of an application to specify memory reuse opportunities between input and output buffers of an actor.

The numerous memory reuse opportunities revealed by considering internal data dependencies of dataflow actors are detailed in Section 2. Then Section 3 presents related work on this topic. The new graph annotations enabling the description of actor-level memory reuse opportunities are introduced in Section 4. Section 5 shows how these new memory reuse opportunities can be exploited to reduce the memory footprint allocated for dataflow graphs. Finally, Section 6 assesses the efficiency of the proposed minimization technique on a set of dataflow graphs of real applications.

2. MOTIVATIONS: MEMORY REUSE OPPORTUNITIES FOR DATAFLOW ACTORS

This section introduces the context of our paper with a presentation of the semantics of the dataflow Model of Computation (MoC) used and a presentation of memory reuse opportunities for both pre-defined and user-defined dataflow actors.

2.1. Synchronous Dataflow

Synchronous Dataflow (SDF) [Lee and Messerschmitt 1987] is the most commonly used dataflow Model of Computation (MoC). An SDF graph $G = \langle A, F \rangle$ is a directed graph containing a set of actors A that are interconnected by a set of FIFOs F. An actor $a \in A$ comprises a set of data ports $\langle P^{in}_{data}, P^{out}_{data} \rangle$ where P^{in}_{data} and P^{out}_{data} respectively refer to a set of data input and output ports. Data ports are used as anchors for FIFO connections. Functions $src : F \rightarrow P^{out}_{data}$ and $snk : F \rightarrow P^{in}_{data}$ associate source and sink data ports to a given FIFO. For each port, a data rate is specified by the function $rate : A \times F \rightarrow \mathbb{N}$. The rate of a port corresponds to the fixed number of data tokens consumed or produced on this port for each execution (or firing) of the parent actor. This property allows a static analysis of an SDF graph during the application compilation. Static analyses can be used to ensure consistency and schedulability properties that imply deadlock-free execution of the application and bounded FIFO memory needs. If an SDF graph is consistent and schedulable, a fixed sequence of actor firings can be repeated indefinitely to execute the graph. This minimal sequence is deduced from the token exchange rates of the graph and is called graph iteration [Lee and Parks 1995].

An example of an SDF graph with 6 actors is given in Figure 1. FIFOs are labeled with their data token production and consumption rates. A FIFO marked with a dot

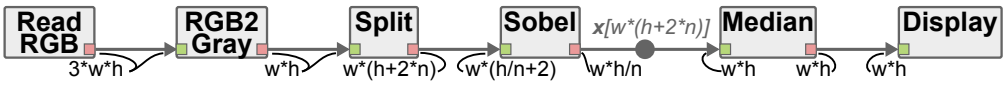


Fig. 1: Image processing SDF graph

signifies that initial tokens are present in the FIFO queue when the system starts to execute. The number of initial tokens is specified by the $\times[N]$ label. Initial tokens are a semantics element of the SDF MoC that makes communication possible between successive iterations of the graph execution; they are often used to pipeline the execution of applications described with SDF graphs [Lee and Messerschmitt 1987]. Actors have no states in the SDF MoC, consequently if enough data tokens are available, an actor can start several executions in parallel. For example in Figure 1, actor *Split* produces enough data tokens for actor *Sobel* to be executed n times in parallel. Hence, the SDF MoC naturally expresses the parallelism of an application. Assigning a static order to the firings of actors on the cores of an architecture is called scheduling the application.

Within the SDF MoC, actors are considered as "black boxes" whose internal behavior can be implemented in any programming language. To simplify the description of this internal behavior, it is convenient to assume that the memory consumed and produced on each FIFO during the firing of an actor constitutes a contiguous memory space called a buffer, and referenced by a pointer [Wipliez and Raulet 2010]. Indeed, by using contiguous memory spaces, the developer of an application avoids the multiple jumps in memory that would have a negative impact on the system performance. By doing so, the developer also avoids writing complex pointer operations that would decrease the source code readability [Cudennec et al. 2014].

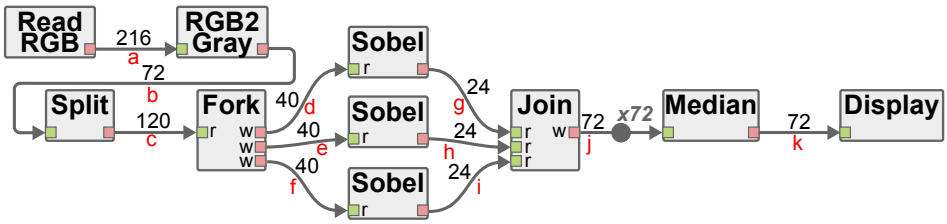


Fig. 2: Single-rate SDF graph derived from the SDF graph of Figure 1 (with h=9, w=8, and n=3).

To reveal these buffers of fixed sizes, an SDF graph can be transformed into an equivalent single-rate graph where each FIFO is replaced with single-rate FIFOs whose consumption and production rates are identical. Each single-rate FIFO is thus a buffer of fixed size accessed by two actors, assuming that executions of successive graph iterations never overlap. For example, Figure 2 presents the single-rate SDF graph that can be derived from SDF graph of Figure 1.

From the SDF programming framework perspective, actors firings are thus seen as function calls accessing indivisible buffers. Since the internal behavior of actors is unknown, the worst-case scenario must be assumed. In this case, the memory allocated for input and output buffers of actors is thus reserved for the complete execution time of actors. Each input and output buffer must be allocated in a separate memory space. To allow memory reuse between input and output buffers of actors, new information must be provided about the internal behavior of actors and the way in which they access the data contained in their input and output buffers. The next section presents

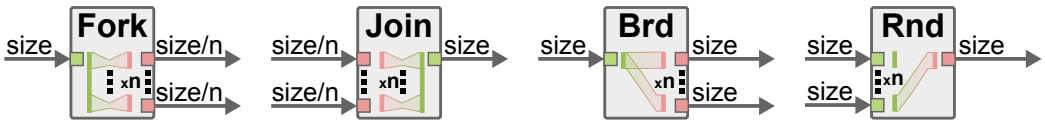
the memory reuse opportunities offered by SDF graphs through the example of the dummy image processing application presented in Figure 2.

2.2. Memory Reuse Opportunities

2.2.1. Special Actors with Pre-Defined Internal Behavior.

As presented in [Pelcat et al. 2012], graph transformations are often applied to the SDF graph of an application in order to reveal its intrinsic parallelism before the multicore mapping, scheduling process and the memory allocation process. The single-rate transformation of SDF graphs and the flattening of hierarchical SDF graphs are two graph transformations that are important to our method.

During these two graph transformations, *special* actors are introduced to ensure the equivalence with the original SDF graph. For example, Figure 2 illustrates the new *Fork* and *Join* actors that were introduced during the single-rate transformation of the SDF graph of Figure 1. The purpose of these *special* actors is to gather, distribute, discard, and duplicate data tokens without performing actual computations on the handled data tokens. The following list illustrates the memory reuse opportunities offered by some of these *special* actors:



(a) *Fork* actor (b) *Join* actor (c) *Broadcast* actor (d) *Roundbuffer* actor

Fig. 3: Special actors: graphs inside actors illustrate internal data movements.

— ***Fork* and *Join* actors:** The purpose of a *Fork* actor (Figure 3a), is to distribute equal parts of the data received in its input buffer to its output buffers. For an input buffer of $size$ bytes, $2 * size$ bytes will be needed for the allocation of input and output buffers of a *Fork* actor in non-overlapping memory spaces. By allocating an output buffer in its corresponding range from the input buffer, half the memory allocated for this actor can be saved. A symmetrical optimization is possible for *Join* actors (Figure 3b) whose purpose is to assemble data from several input buffers into a single contiguous output buffer.

— ***Broadcast* actors:** The purpose of a *Broadcast* actor (Figure 3c) is to produce a copy of the content of its input buffer into each of its output buffer. For n output buffers and an input buffer of $size$ bytes, $n * size$ bytes will be needed for the non-overlapping allocation of input and output buffers. By merging the n output buffers with the input buffer, the memory allocated for a *Broadcast* actor can be divided by *n+1*.

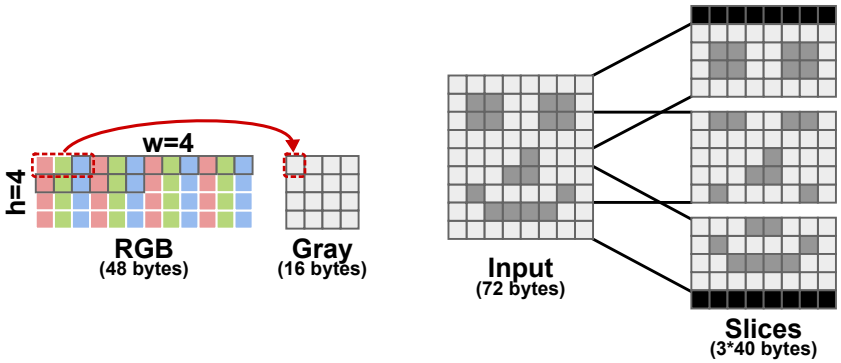
— ***Roundbuffer* actors:** The purpose of a *Roundbuffer* (Figure 3d) actor is to forward the data tokens received on its last input buffer to its output buffer, and to discard data tokens received on all other input buffers. For n input buffers and an output buffer of $size$ bytes, $n * size$ bytes will be needed for the non-overlapping allocation of input and output buffers. To take advantage of this internal behavior the allocation of the last input buffer can be matched directly in the output buffer memory. Moreover, since data tokens contained in all other input buffers are discarded by the *Roundbuffer* actor, it does not matter whether the data contained in these buffers are valid or not. Hence, these input buffers can be allocated in overlapping memory spaces, thus overwriting the content of each others. By doing so, the memory footprint of a

Roundbuffer actor can be reduced to a footprint only twice as large as the $size$ of the output buffer.

In addition to reducing the memory footprint of the *Fork*, *Join*, *Broadcast*, and *Roundbuffer* actors, these optimizations also improve the performance of applications since copying data from input to output buffers is no longer required. The memory reuse opportunities presented in this section can be applied systematically since the actors concerned are *special* actors whose behavior is not defined by the application developer. The next section presents memory reuse opportunities resulting from the internal behavior of user-defined actors.

2.2.2. Actors with User-Defined Internal Behavior.
The description of the internal behavior of a dataflow actor is not part of the SDF MoC. Consequently, many SDF programming frameworks have no knowledge of the internal data dependencies of actors [Buck et al. 1994; Stuijk et al. 2006b]. The last section showed that advanced memory optimization is possible for input and output buffers of *special* actors whose behaviors are pre-defined. Although their behavior is defined by the application developer, user-defined actors also present internal data dependencies that may favor the use of advanced memory reuse techniques.

Information on the internal data dependencies between the input and output buffers of an actor can be used to reveal the **disjoint lifetimes of sub-ranges of the buffers**. For example, if a sub-range of an input buffer is last read before a sub-range of an output buffer of equal size is first written, these two sub-ranges might be allocated in the same memory space. In such a case, corrupting the input buffer by writing the result into it is not an issue since the data contained in this input buffer will no longer be used. An allocation technique which takes advantage of the data dependencies exposed by an automatic analysis of application data access is presented in [De Greef et al. 1997].



(a) *RGB2Gray*: Input RGB bytes with a dark border can be used to store output Gray bytes.

(b) *Split*: Slices can be read directly from the input buffer. Black pixel lines of the slices are not part of the input image.

Fig. 4: Memory reuse opportunities for custom actors.

Figure 4a illustrates the internal data dependencies of the *RGB2Gray* actor for a 4x4 pixel image. In this example, 3 bytes, the R, G and B values, are read to produce each output byte, the luminance value. Assuming that a raster scan of the input pixels is used, the input bytes will never be read twice and the result can be stored directly in the input. Hence, as illustrated in Figure 4a, the 16 output bytes can be stored in a contiguous sub-range of the input buffer. The sub-range must be chosen to ensure

that output byte does not overwrite an unread input byte. In the example, the 16 bytes of the *Gray* output buffer are stored in the $[2; 17]$ sub-range of the *RGB* input buffer. Another valid solution would be to store these 16 bytes in the $[0; 15]$ sub-range.

A special case of mergeable buffers is the *Split* actor presented in Figure 4b. The purpose of the *Split* actor is to divide an input image into several overlapping slices. In this example, each slice has a 1-pixel line overlap with both the previous and the next slices. Provided that the consumer of the slices do not write within the slice memory, output slices can be allocated directly within the input buffer memory. As illustrated in Figure 4b, border slices of the image may require the allocation of extra lines of pixel. If each slice is allocated in its corresponding contiguous memory space of the input buffer, then these extra lines will be allocated before and after the memory allocated for the input buffer.

In-place internal behavior is a another special case of data access pattern that a developer might take advantage of to optimize the memory footprint of his application [Hsu et al. 2005]. An in-place algorithm is an algorithm that can write its results directly in its input buffer during its execution. In addition to the input buffer memory, the execution of an in-place algorithm requires a small working memory whose size is independent of the size of the processed data. Examples of in-place signal processing algorithms can be found in the literature such as Fast Fourier Transform (FFT) algorithms [Burrus and Eschenbacher 1981], and sorting algorithms [Geffert and Gajdoš 2011].

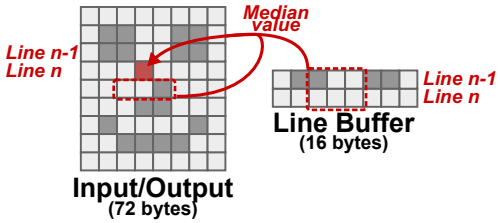


Fig. 5: Internal behavior of *Median* actor with in-place results..

Figure 5 illustrates the in-place behavior of the *Median* actor from the SDF graph of Figure 1. The purpose of the *Median* is to replace each pixel with the median value derived from the neighboring 3x3 pixels. Its implementation is based on a sliding-window of 3x3 pixels which successively scan the lines of the filtered image. To compute the filtered value of a pixel, the median filter must keep the original values of the 8 surrounding pixels in memory. If the result of the filter is written directly in the input image, the algorithm behavior will be corrupted since the new value assigned to a pixel will be used for the computation of its neighbor pixels. By buffering two pixel lines of the input image in a working memory, the *Median* actor can be implemented so that its results are written directly in its input buffer. As illustrated in Figure 5, to compute the value of a pixel, 3 pixels are read from the original input image and 6 pixels are read from the line buffer. The line buffer stores the original value of pixels that have already been overwritten in the input image by the application of the median filter on previous lines of pixels. Note that the *Median* actor is not an in-place algorithm in the strict sense as it requires a working memory (i.e. the line buffer) whose size is proportional to the width of the processed input image.

The next section presents related work on memory optimization techniques which take advantage of application data access and internal behavior of actors.

3. RELATED WORK

The analysis of data dependencies to optimize the memory allocation of an application has been an important research subject for many years and has been studied at many levels of abstraction.

3.1. Compiler Optimizations

Data dependency analysis is an important optimization process of modern compilers. This optimization consists of automatically analyzing the source code of an application to identify data accesses and expose variables with disjoint lifetimes. Graph coloring [Quong and Chen 1993] and clique partitioning [Kim and Shin 2002] techniques are then used to perform the memory allocation of these variables with a minimal memory footprint. Such approaches have been widely used to optimize the memory allocation for procedural programming languages such as C [De Greef et al. 1997] and Fortran [Fabri 1979].

For example, in [De Greef et al. 1997], the order in which array elements are last read and first written in a C program is exposed through an automated analysis of the source code. Using this information, an allocation algorithm is proposed to partially merge the memory allocated for arrays with non-overlapping lifetimes.

Keywords have been introduced in programming languages to explicitly specify the type of access for a given variable or array. For example, the `const` keyword in C or the `final` keyword in Java both specify that the associated primitive object will be read but never written. There exist other keywords such as the `restrict` or the `volatile` keywords in C language [Magee 2005]. Using these keywords, the developer of an application gives information to the compiler that can not be deduced from code analysis because of complex nested function calls or call to external libraries. This information is used by the compiler to minimize the allocated memory footprint of applications.

3.2. Dataflow Optimizations

Minimizing the memory footprint of SDF applications is usually achieved by using FIFO sizing techniques [Stuijk et al. 2006a; Benazouz et al. 2010] that consist of finding a schedule to minimize the memory space allocated to each FIFO of an SDF graph. In contrast to our technique, FIFO sizing techniques do not consider merging opportunities between input and output buffers. The technique presented in [Geilen et al. 2005] finds a schedule that minimizes the maximum number of tokens ever stored in FIFOs of the graph during an iteration. In this method, consumption and production of tokens are considered as synchronous events, thus allowing storage of tokens produced and consumed by an actor firing in the same memory space, but ignoring the memory space needed to store these tokens during actor firings. For this reason, token consumption and production were made asynchronous events when implementing this method in Section 6.

Several solutions to *broadcast* data tokens can be found in the literature. Nondestructive reads, also called FIFO peeking, is a well-known way to read data tokens without popping them from FIFOs, hence avoiding the need for *Broadcast* actors [Fischaber et al. 2007]. Unfortunately, this technique cannot be applied without considerably modifying the underlying SDF MoC. Indeed, the use of FIFO peeking means that an actor does not have the same behavior for all firings. Otherwise, tokens of peeked FIFOs would never be consumed and would accumulate indefinitely. Another solution to this issue is to use a single-writer, multiple-readers FIFO that discards data tokens only when all readers have consumed them [Mamidala et al. 2011]. An evaluation of this technique is presented in Section 6.

In [Cudennec et al. 2014], a technique is proposed to enable buffer merging for a set of actors with pre-defined behavior. In contrast to the method presented in this article, this technique does not allow buffer merging for actors with a user-defined behavior.

Few articles can be found in the literature on the topic of allocation of input and output buffers of an actor in overlapping memory spaces. In [Murthy and Bhattacharyya 2004], an annotation system is introduced to specify a relation between the number of data tokens produced and consumed for a pair of input and output buffers of an actor. This relation is then used jointly with scheduling information to enable the merging of annotated buffers. In [Hsu et al. 2005; Forget et al. 2013], other annotation systems are introduced to specify buffers that may be used for in-place execution of actors. The advantage of these annotation-based techniques is that no modification of the underlying SDF MoC is required. Despite the fact that SDF FIFOs must be replaced with buffers to benefit fully from these annotations, a regular SDF graph can still be obtained by ignoring these annotations. The major drawback of these two annotation systems is that they only allow pairwise merging of input and output buffers. Hence, these annotation systems are unable to model the behavior of *Fork* or *Broadcast* actors, and so require another method for merging several output buffers into a single input. Moreover, the optimization technique presented in [Murthy and Bhattacharyya 2004] relies on a monocore scheduling of the application graph. The extension of this optimization technique to multicore architectures and schedules is not straightforward.

Like existing annotation systems, the technique presented in this article does not require any modification of the SDF semantics. In contrast to existing techniques, our buffer merging technique can be used for any number of input and output buffers. Although the work presented in this article only applies to the static SDF MoC, it could easily be extended to dynamic dataflow MoCs, using technique similar to that presented in [Wipliez and Raulet 2010] where certain FIFOs are replaced with buffers.

This paper extends our previous work on buffer merging technique [Desnos et al. 2015] by providing a set of notations and properties to formally model a buffer merging problem. Using these notations and properties enables the automated verification of the validity of buffer merging patterns created by a developer. These notations are also used to formally characterize the different steps of the iterative buffer merging optimization process in which the application of a merging pattern may invalidate other buffer merging opportunities specified by the developer (Section 5). This paper also introduces new port annotations and associated optimization process to handle the memory optimization of *Roundbuffer* and similar actors. Finally, the paper presents new experimental results on a broader set of dataflow specifications of real DSP applications.

4. GRAPH ANNOTATIONS

In addition to the single-rate SDF graph which specifies application behavior, the buffer merging technique presented in this paper relies on two additional inputs abstracting the internal behavior of actors: a script-based specification of mergeable input and output buffers, and annotations of the ports of SDF actors. Advantages of this technique are:

— **No impact on SDF graphs:** Annotating an application with these new inputs does not require any modification of the underlying SDF graph. These new optional inputs are only used by the development framework as part of a compile time optimization process. If an annotated application were to be implemented on a target that does not support these optimizations, these annotations could simply be ignored without any impact on the application behavior. Conversely, the proposed annotations are not indispensable for the description of a functional application. Annotations can be added

to an application description only in late stages of development, when optimization of the application memory footprint is needed.

— **Independence from the *host* language:** The *host* language is the language used to specify the internal behavior of actors. The proposed optimization technique is not based on an automated analysis of the host code of actors. Instead, a script-based description of the mergeable buffers is provided by the application developer. This abstract description can be suitable for several implementations of a given actor in different *host* languages.

— **Lightweight and intuitive information:** The scripts specifying the memory reuse opportunities for an actor are never longer than ten lines of code, and can be tested independently from the internal computations performed by this actor. The use of a scripting language instead of mathematical notations is also more natural and intuitive for software developers.

— **Semi-automated graph annotation:** Since the behavior of some "special" actors (*Fork*, *Join*, *Broadcast*, *Roundbuffer*) is pre-defined, annotations associated to with actors can be added automatically during the graph transformations.

4.1. Actors Annotations with Memory Scripts

The first input used to abstract the internal behavior of actors is called a memory script. The objective of memory scripts is to allow the application developer to to specify which input buffer can be merged with which output for a given actor, and the relative position of the resulting merged buffers. To this purpose, each actor of the SDF graph can optionally be associated with a memory script.

Memory scripts are interpreted at compile time for each actor of the single-rate graph. For each actor, the script inputs are: a list of the input buffers, a list of the output buffers, and a list of parameters influencing the behavior of the actor. The script execution produces a list of *matches* between the input and output buffers of the actor. Each match associates a sub-range of bytes from an input buffer with a sub-range of bytes from an output buffer. Applying a match consists of merging the memory allocated to the two sub-ranges in a unique address range of the memory.

4.1.1. Memory Script Example.

Figure 6 presents the memory script associated with the user-defined *Split* actor and illustrates the matches resulting from its execution. Memory scripts are written with a derivative of the Java language called BeanShell [Niemeyer 2014]. The h, w, and n parameters correspond to the height and width of the sliced image and the number of slices produced by the actor respectively. Figure 6b illustrates the matches obtained for the case of $h = 9$, $w = 8$, and $n = 3$. The output buffer of the *Split* actor is divided into three sub-ranges of equal size $(h/nb_{slice}+2)*w$. Each output sub-range is matched to the corresponding sub-range of the input buffer, thus creating overlaps between the matched input sub-ranges. It is interesting to note that the first and the last sub-ranges of the output buffer are partially matched outside the boundaries of the input buffer initial byte range $r_{bytes} = [0, 72[$.

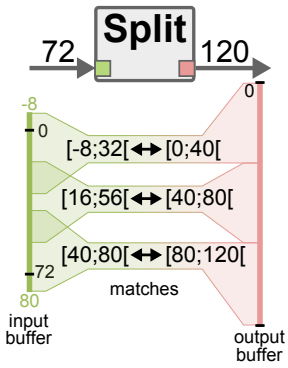
4.1.2. Definitions.

The following definitions give the formal notations associated with the buffers and matches concepts used in this article.

Definition 4.1 (*Buffers*). For an actor $a \in A$ of a single-rate SDF graph $G = \langle A, F \rangle$, B_a is the set of input and output buffers of actor a. $B_a^{in} \subseteq B_a$ and $B_a^{out} \subseteq B_a$ are the sets of input and output buffers respectively corresponding to ports P_{data}^{in} and P_{data}^{out} of actor a. Each buffer $b \in B_a$ is defined as a 1-tuple $b = \langle r_{bytes} \rangle$ where:

```
/* Input buffer:  bIn
 * Output buffer: bOut
 * Parameters:    h, w, n
 */
int s=w*(h/n+2);//Slice size
for(int i=0; i<n ;i++){
  bOut.match(  //Match source
    i*s,         //Start index in bOut
    bIn,         //Match destination
    (i*h/n-1)*w,//Start index in bIn
    s);          //Matched range size
}
```

(a) Memory Script



(b) Created matches

Fig. 6: Memory script for *Split* actor (with h=9, w=8, and n=3).

– r_{bytes} is the range of bytes associated with buffer $b \in B_a$. By definition $r_{bytes} = [start, end[$ with $start, end \in \mathbb{Z}$ and $start < end$. The **initial range of bytes** r_{bytes} for buffer $b \in B_a$ associated with port $p \in P^{in}_{data} \cup P^{out}_{data}$ is $b.r_{bytes} = [0, rate(p)[$. The amount of memory needed to allocate a given buffer is deduced from the range of bytes $b.r_{bytes}$ associated with this buffer: $size := b.r_{bytes}.end - b.r_{bytes}.start$.

Definition 4.2 (*Matches*). For an actor $a \in A$ of an SDF graph $G = \langle A, F \rangle$:
M_a is the set of matches associated with buffers B_a of actor a. Each match $m \in M_a$ is defined as a tuple $m := \langle b_{src}, r_{src}, b_{dst}, r_{dst} \rangle$ where:
– $b_{src}, b_{dst} \in B_a$ are the source and the destination buffers of match $m \in M_a$.
– r_{src} and r_{dst} are the matched sub-ranges of bytes from buffers b_{src} and b_{dst}.
It is important to note that a match $m \in M$ can be applied in both directions: the destination sub-range $m.r_{dst}$ can be merged into the source buffer $m.b_{src}$ or the source sub-range $m.r_{src}$ can be merged into the destination buffer $m.b_{dst}$.

4.1.3. Matching Rules.

As shown in Figure 6, it is possible to match a contiguous sub-range of a buffer into non-contiguous sub-ranges. This example also illustrates the possibility of matching sub-ranges of a buffer partially outside the initial range of bytes of another buffer. Although memory scripts offer great liberty for defining custom matching patterns, a set of rules must be respected to ensure the correct behavior of an application.

R1. Both sub-ranges r_{src} and r_{dst} of a match $m \in M_a$ must cover the same number of bytes: $r_{dst}.end - r_{dst}.start = r_{src}.end - r_{src}.start$.
R2. A match $m \in M_a$ can only be created between an input buffer $b_{src} \in B^{in}_a$ and an output buffer $b_{dst} \in B^{out}_a$.
R3. A sub-range of bytes of an output buffer $b_{dst} \in B^{out}_a$ can not be matched several times by overlapping matches. Formally, if $\exists\ m = \langle b_{src}, r_{src}, b_{dst}, r_{dst} \rangle \in M_a$ then $\nexists\ m' = \langle b'_{src}, r'_{src}, b'_{dst}, r'_{dst} \rangle \in M_a | b'_{dst} = b_{dst}$ and $r'_{dst} \cap r_{dst} \neq \emptyset$
R4. All matches $m \in M_a$ must involve at least one byte from the initial byte range $b_{src}.r_{bytes}$ and one byte from the initial byte range $b_{dst}.r_{bytes}$. Formally, the following condition must always be true: $\forall m = \langle b_{src}, r_{src}, b_{dst}, r_{dst} \rangle \in M_a$, $r_{src} \cap b_{src}.r_{bytes} \neq \emptyset$ and $r_{dst} \cap b_{dst}.r_{bytes} \neq \emptyset$.
R5. Only bytes within the initial byte range $b.r_{bytes}$ of their buffer $b \in B_a$ can be matched with bytes falling outside the initial byte range of the matched buffer. Formally, con-

sidering a match $m = \langle b_{src}, r_{src}, b_{dst}, r_{dst} \rangle \in M_a$, for each byte $n \in r_{src}$ and its matched byte $n' \in r_{dst}$, if $n \notin b_{src}.r_{bytes}$ then $n' \in b_{dst}.r_{bytes}$.

These five rules are checked for each match created during execution of a memory script and errors are immediately reported to the developer. Rule R1 enforces the validity of the matches. Indeed, it is impossible to allocate a contiguous sub-range of n bytes within a memory range of m bytes if $m \neq n$. Rules R2 and R3 prevent scripts from generating matching patterns that would result in a *destination merge issue* (see Section 5.1). Rules R4 and R5 regulate the creation of matches with bytes falling outside the initial range of bytes of buffers. A byte is called "virtual" if it does not belong to the initial range of bytes of a buffer, otherwise the byte is called "real". All "virtual" and "real" bytes of a buffer are mapped in memory when a buffer is allocated. As illustrated by the *Split* actor (Figure 6), memory scripts can be used to match a sub-range of a buffer partially outside the range of "real" bytes of another buffer. Without further limitations, this feature could be used to merge a buffer completely out of the "real" byte range of another buffer, thus resulting in no memory reuse between the two buffers. Such matches are made impossible by forcing matches to have at least one "real" byte on both sides of the match (R4), and by forbidding matches between "virtual" bytes (R5).

4.2. Data Ports Annotations

As illustrated by the *Split* actor, memory scripts allow the creation of overlapping matches. Applying overlapping matches results in merging several sub-ranges of output buffers in the same input buffer. Hence, actors reading data from the merged output buffers are accessing the same memory. To ensure the correct behavior of the application, the memory optimization process must check that actors accessing the merged buffer do not write in this shared memory. If one of the consumer actor does not respect this condition, its corresponding output buffer should not be merged and it should be given a private copy of the data. Indeed, writing in the shared memory would modify the input data read by other actors, which might change their behavior.

By default, the most flexible actor behavior is assumed; all actors are supposed to be both writing to and reading from all their buffers during their execution. Since this assumption excludes matches with overlapping sub-ranges, a set of graph annotations has been introduced to specify how actors access their input and output buffers. Each data port $p \in P^{in}_{data} \cup P^{out}_{data}$ can be annotated with the following keywords:

— **Read-Only:** The actor possessing a *read-only* input port can only read data from this port. Like a `const` variable in C or a `final` variable in Java, the content of a buffer associated to a *read-only* port can not be modified during the computation of the actor to which it belongs.
— **Write-Only:** The actor possessing a *write-only* output port can only write data on this port. During its execution, an actor with a *write-only* buffer is not allowed to read data from this buffer, even if data in the buffer was written by the actor itself.
— **Unused:** The actor possessing an *unused* input port will never write nor read data from this port. Like the `/dev/null` device file in Unix operating systems, an *unused* input port can be used as a sink to consume and immediately discard data tokens produced by another actor.

Developers should note that if a match created by a memory script results in matching an input buffer into an output buffer where data is written by the actor, then the corresponding input port must not be marked as *read-only*. For example, the input port of the *Split* actor presented in Figure 6 cannot be marked as *read-only* because it is matched in an output buffer whose first and last lines of pixel will be written by the

Split actor. In the SDF graph of Figure 2, an `r` mark is associated with each *read-only* port and a *w* mark is associated with each *write-only*.

An example of use-case for the *write-only* and *unused* port annotations are the *Roundbuffer* actors. As presented in Section 2.2.1, the only purpose of *Roundbuffer* actors is to forward on their output port the last data tokens consumed on their input ports. All input ports of a *Roundbuffer* actor except the last one can thus be marked as *unused*. If a single-rate FIFO connects an *unused* input port to a *write-only* output port, the memory allocated for this buffer will never be read by any actor. Hence, the memory allocated for such a FIFO can be overwritten at any time, including during the execution of the producer actor of the FIFO, without changing the behavior of the application. The process taking advantage of *write-only* and *unused* annotations is presented in Section 5.3.2 and does not depend on matches created by memory scripts.

Port annotations can easily be automated for the data ports of the pre-defined *Fork*, *Join*, *Broadcast*, and *Roundbuffer* actors that are inserted during graph transformations. All output ports of these special actors are thus automatically marked as *write-only* ports, and all input ports, except the first ports of the *Roundbuffer* actors, are marked as *read-only* ports. For user-defined actors, it is the developer's responsibility to make sure that the internal behavior of actors is consistent with port annotations.

The memory scripts and port annotations presented in this section allow the developer of an application to specify how the input and output buffers can be merged, and how actors access these buffers. In the next section, an optimization process is presented to make use of these inputs to reduce the memory footprints of SDF graphs.

5. MEMORY MINIMIZATION PROCESS

The execution of memory scripts during the compilation of an SDF graph produces a list of matches that represent merging opportunities for the input and output buffers of actors. Some matching pattern specified by memory scripts may result in a corruption of the application behavior (see Sections 4.1.3 and 5.1). The purpose of the memory minimization process is to apply as many matches as possible without corrupting the application behavior. For this purpose, the memory minimization process first builds Directed Acyclic Graph (DAG) of buffers and matches by chaining the lists of matches produced by the memory scripts.

Definition 5.1. A match DAG derived from a single-rate acyclic SDF graph $G = \langle A, F \rangle$ is a Directed Acyclic Graph (DAG) denoted by $T = \langle B, M \rangle$ where:

- B is the set of vertices of the DAG. Each vertex is a buffer $b \in B_a$ of an actor a from the SDF graph (see Definition 4.1). In the match DAG, a single buffer $b \in B$ is used to represent an output buffer $b_o \in B^{out}_{prod}$ and an input buffer $b_i \in B^{in}_{cons}$ linked by a single-rate FIFO.
- M is the set of directed edges of the DAG. Each edge $m \in M$ is a match produced by the memory script of an actor (Definition 4.2).

The combination of all the buffers and matches of an application results in the creation of a forest (that is the creation of several unconnected match DAGs). An example of match DAG is given in Figure 7 below the corresponding single-rate SDF graph. In this figure, the single-rate FIFOs between actors *RGB2Gray* and *Split*, and actors *Split* and *Fork* each are represented by a single buffer.

The match DAGs derived from an SDF graph are used by the optimization process to identify the matches that can be applied without corrupting the behavior of the application.

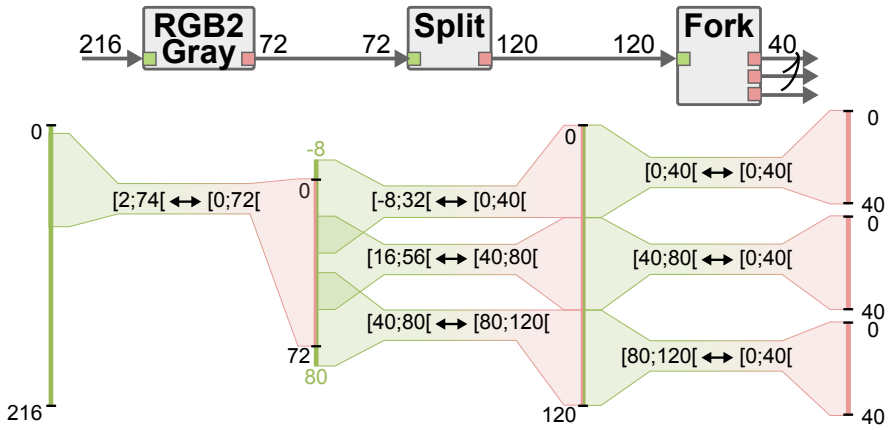


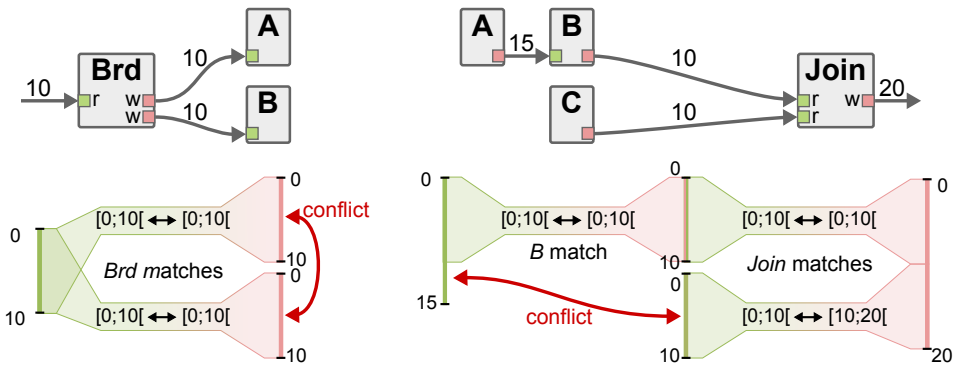
Fig. 7: Match DAG associated to buffers of actors *RGB2Gray*, *Split*, and *Fork*.

5.1. Match Applicability Issues

A match $m \in M_a$ is said to be applicable if it can be applied without changing the behavior of the application. The 5 matching rules presented in Section 4.1.3 are necessary conditions to ensure the applicability of a created match. However, following these rules is not sufficient to guarantee the applicability of the created matches.

5.1.1. Match Overlap Issues.

Figure 8 gives examples of matches that respect the matching rules presented in Section 4.1.3 but that can not be applied without corrupting the application behavior.



(a) Source overlap issue (b) Destination overlap issue

Fig. 8: Matching patterns with inapplicable matches.

— **Source overlap issue:** Matches with overlapping source sub-ranges can be applied only if their destination buffers are *read-only* or *unused*. In Figure 8a, only one of the *Broadcast* matches can be applied because neither actor A nor actor B have a *read-only* input port and it is thus implicitly assumed that both actors may write in their input buffers. If both matches were applied, actors A and B would write in each other input buffer and corrupt the application behavior.

— **Destination overlap issue:** A chain of matches cannot be applied if it results in merging several source sub-ranges in overlapping destination sub-ranges. In Figure 8b, if all matches were applied, the output buffers of actors A and C would be

merged in ranges $[0, 15[$ and $[10, 20[$ respectively of the output buffer of the *Join* actor. Hence, if all matches were applied and actor *A* was executed after actor *C*, then actor *A* would partially overwrite the data tokens produced by actor *C*, thus corrupting the application behavior.

In order to detect and avoid applying the matches involved in a merging issue, a set of new properties is introduced by the memory optimization process.

Definition 5.2. The tuple associated to a buffer $b \in B$ is extended as follows: $b = \langle r_{bytes}, r_{merge} \rangle$ where r_{merge} is the set of ranges of mergeable bytes of the buffer. Buffers corresponding to FIFOs connected to a *read-only* or an *unused* input buffer are initialized with a mergeable range $r_{merge} = \{r_{bytes}\}$, otherwise $r_{merge} = \{\emptyset\}$.

Here, r_{merge} is defined as a set of ranges because it can contain several non-contiguous ranges as a result of an update of the buffer properties (see Section 5.2).

A range of bytes of a buffer is said to be *mergeable* only if the consumer actor associated to the corresponding FIFO does not write in this buffer (that is, sink port of the FIFO is either *read-only* or *unused*). If all ranges of bytes involved in a *source overlap issue* are *mergeable*, then the corresponding matches can be applied without corrupting the application behavior. $()^{dst \xrightarrow{m} src}$ denotes the translation of a range of bytes of the destination buffer into the corresponding range of the source buffer according to match m.

PROPERTY 5.1 (SOURCE OVERLAP ISSUE). *Two matches $m_1, m_2 \in M$ with overlapping source sub-ranges $m_1.r_{src}$ and $m_2.r_{src}$ with $m_1.b_{src} = m_2.b_{src}$ can both be applied if and only if $(m_1.r_{src} \cap m_2.r_{src}) \subseteq ((m_1.b_{dst}.r_{merge})^{dst \xrightarrow{m} src} \cap (m_2.b_{dst}.r_{merge})^{dst \xrightarrow{m} src})$. If this condition is not satisfied, the two matches are said to be **in conflict**, and at most one of them can be applied.*

PROPERTY 5.2 (DESTINATION OVERLAP ISSUE). *Two matches $m_1, m_2 \in M$ with overlapping destination sub-ranges $m_1.r_{dst}$ and $m_2.r_{dst}$ with $m_1.b_{dst} = m_2.b_{dst}$ are **in conflict** and can never be both applied. Formally, if $m_1.r_{dst} \cap m_2.r_{dst} \neq \emptyset$, then m_1 and m_2 mutually exclude each other.*

Properties 5.1 and 5.2 present the applicability rules for the source overlap issue and the destination overlap issue respectively.

5.1.2. Buffer Division Issue.

To ensure that the behavior of an application is not modified when buffer merging is applied, a necessary condition is that all actors must always have access to all their input and output buffers (Section 2.1). The example of Listing 1 shows that, provided that the memory allocated for its output buffer is merged into its input buffer memory, a *null* output pointer can be given to an *in-place* actor like *RGB2Gray*. In this example, the output and input buffers are allocated in a single memory space, hence there is no need to have a specific output pointer since all buffer accesses made by the actor can be made using the input pointer.

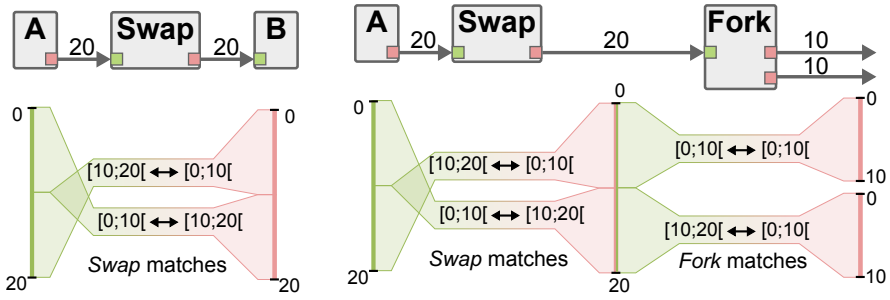
```
// Call to RGB2Gray actor
// 16 output bytes are merged in byte range [2, 17] of the input buffer.
rgb2gray(4 /*height*/, 4 /*width*/,
         ptr /*pointer to 48 bytes*/, NULL /*null pointer*/);
```

Listing 1: Possible call to RGB2Gray actor if buffers are merged as in Figure 4a.

As illustrated by the memory script of actor *Split* in Figure 6, contiguous sub-ranges of bytes can be matched in non-contiguous ranges of bytes. The application of this

matching pattern requires the division of the output buffer of actor *Split* into several sub-ranges, each matched in a distinct location.

Since a buffer divided and allocated into non-contiguous sub-ranges can no longer be accessed through a simple pointer, a condition for dividing a buffer is that all actors accessing this buffer can still access all its sub-ranges. A divided buffer remains accessible to an actor only if the memory script of this actor matches all the sub-ranges of this buffer into other buffers accessible by this actor. In Figure 9a, neither actor *A* nor actor *B* is associated with a memory script, and hence neither actor would be permitted to keep an access to the sub-ranges if the buffer division was applied. Since applying the *Swap* matches is only possible by dividing the output buffer of actor *A* or the input buffer of actor *B*, these matches cannot be applied.



(a) Division issue (b) No division issue

Fig. 9: Divisibility of buffers

In Figure 9b, the *Swap* actor is followed by a *Fork* actor. In this case, all sub-ranges of the input buffer of actor *Fork* are matched within its output buffers. Consequently, actors *Swap* and *Fork* both expect a division of the buffer corresponding to the FIFO between them. In such a case, the buffer can be divided into two non-contiguous ranges of bytes merged and accessible into the input buffer of actor *Swap* and into the output buffers of actor *Fork* respectively.

In order to detect and check the applicability of matching patterns which require the division of a buffer, a set of new properties is introduced by the memory optimization process.

Definition 5.3. The tuple associated to a buffer $b \in B$ is extended as follows: $b = \langle r_{bytes}, r_{merge}, r_{div} \rangle$ where r_{div} is a set of *indivisible* ranges of bytes of the buffer. If a range of bytes is *indivisible* it will compulsorily be allocated in a contiguous memory space.

Considering the set of matches $S \subset M$ involving a buffer $b \in B$, the set of indivisible ranges r_{div} of this buffer is initialized as the *lazy union* of the matched sub-ranges.

Definition 5.4 (*Lazy union*). Considering two ranges $r_1 = [a, b[$ and $r_2 = [c, d[$, the *lazy union* of these ranges, noted $\overset{lazy}{\cup}$, is computed as follows:

$$r_1 \overset{lazy}{\cup} r_2 = \begin{cases} \text{if } r_1 \cap r_2 \neq \emptyset & \text{then } [min(a,c); max(b,d)[\\ \text{if } r_1 \cap r_2 = \emptyset & \text{then } \{[a,b[; [c,d[\} \end{cases}$$

The lazy union of two sets of ranges is computed like a classic union $\cup$ of two sets of ranges, but using with the lazy union $\overset{lazy}{\cup}$ as the basic operation.

Considering two consecutive ranges of bytes $r_1 = [a, b[$ and $r_2 = [b, c[$, contrary to the standard union operator whose result is $r_1 \cup r_2 = [a, c[$, the result of the *lazy union* is $r_1 \overset{lazy}{\cup} r_2 = \{[a, b[; [b, c[\}$.

Definition 5.5. For a buffer $b \in B$ and the set S of associated matches

$$S = \{m \in M | m.b_{dst} = b \vee m.b_{src} = b\}$$

Let $m.r_b$ be the matched sub-range (source or destination) of $m \in S$ associated with buffer b. The indivisible range r_{div} associated with buffer b is initialized as follows:

$$b.r_{div} = \begin{cases} \text{if } \underset{m \in S}{\cup} m.r_b = b.r_{bytes} & \text{then } \overset{lazy}{\underset{m \in S}{\cup}} m.r_b \\ & \text{else } b.r_{bytes} \end{cases}$$

In Figure 9b, the input buffer of actor *Swap* and the output buffer of actor *Swap* (and input buffer of actor *Fork*) both have several indivisible ranges $r_{div} = \{[0, 10[; [10, 20[\}$. The two output buffers of actor *Fork* each have a single indivisible range that covers their complete range of bytes: $r_{div} = r_{bytes} = [0; 10[$.

If only part of the range of bytes of a buffer $b \in B$ is matched, then this buffer is indivisible, and its indivisible range is $b.r_{div} = b.r_{bytes}$. For example, in Figure 7, the indivisible range of bytes associated to the input buffer of actor *RGB2Gray* is $[0; 216[$.

Definition 5.5 specifies how the initial indivisible ranges of a buffer can be computed. The following property gives the conditions that must be satisfied for a buffer to be divided into these indivisible sub-ranges.

PROPERTY 5.3. *For a buffer $b \in B$ and the set of associated matches $S \subset M$ (as defined in Definition 5.5), buffer b can be divided into sub-ranges $b.r_{div}$ and matched into non-contiguous ranges of bytes if and only if all the following conditions are met:*

(*1*) *Matches in S with $m.b_{src} = b$ completely cover the range of bytes of buffer b and have no overlap. Formally, with $S^{src} = \{m \in S | m.b_{src} = b\}$, the condition is* $(\underset{m \in S^{src}}{\cup} m.r_{src} = b.r_{bytes})_1 \wedge (\forall m_1, m_2 \in S^{src} : m_1 \neq m_2 \Rightarrow m_1.r_{src} \cap m_2.r_{src} = \emptyset)_2$.

(*2*) *Matches in S with $m.b_{dst} = b$ completely cover the range of bytes of buffer b and have no overlap. Formally, with $S^{dst} = \{m \in S | m.b_{dst} = b\}$, the condition is* $(\underset{m \in S^{dst}}{\cup} m.r_{dst} = b.r_{bytes})_1 \wedge (\forall m_1, m_2 \in S^{dst} : m_1 \neq m_2 \Rightarrow m_1.r_{dst} \cap m_2.r_{dst} = \emptyset)_2$.

(*3*) *All matches in S are applicable under Properties 5.1 and 5.2*

(*4*) *All matches in S must match buffer b only with indivisible buffers. Formally, with $m.b_{remote}$ the second buffer ($\neq b$) matched by $m \in S$, the condition is* $\forall m \in S, m.b_{remote}.r_{div} = m.b_{remote}.r_{bytes}$.

When buffer $b \in B$ is divided, it can no longer be accessed as a contiguous memory space. Consequently, a unique reference (that is, a pointer) is not sufficient to access all the memory associated with this buffer. In order to preserve the behavior of the application, actors must find another access to the divided buffer. Consequently, all actors accessing a divided buffer must have matched all sub-ranges of the divided buffer with other buffers accessed by the actor. Hence, a buffer b can be divided only if both its producer and consumer actors completely match b with other buffers. This condition is expressed by the first parenthesis $()_1$ in the formal expression of conditions 1 and 2 of Property 5.3. It is the developer responsibility to ensure that when an actor accesses a divided buffer, it does so through the sub-ranges matched with other buffers.

To avoid ambiguities, all sub-ranges of a divided buffer must be matched exactly once with other buffers accessed by the actors. This condition is expressed by the second parenthesis $()_2$ in the formal expression of conditions 1 and 2 of Property 5.3.

Following conditions 1 and 2, each sub-range of a divided buffer is matched exactly once in buffers of its producer actor, and once in buffers of its consumer actor. If one of these matches is not applied, the corresponding actor will not be able to access the unmatched sub-range of the divided buffer. Consequently, all matches must be applicable for the buffer to be divided (condition 3).

Finally, sub-ranges of a divided buffer can only be accessed in the remote buffers with which they were merged. Consequently, these remote buffers cannot be divisible themselves since their content must remain accessible to the actors through a simple reference (condition 4).

In Figure 9b, the buffer corresponding to the output of the *Swap* actor satisfies all three conditions described in Property 5.3. Hence, if this buffer is divided, a *null* pointer will be given in its place to actors *Swap* and *Fork*. Listing 2 presents the function call of these two actors for the case when the buffer between is divided.

```
// Call to actor Swap
// Output range [0;10[ is accessible in range [10;20[ of the input
// Output range [10;20[ is accessible in range [0;10[ of the input
swap(ptrIn /*Input*/, NULL /*Output: null pointer*/);
// Call to actor Fork
// Input range [0;10[ is accessible in range [0;10[ of the first output
// Input range [10;20[ is accessible in range [0;10[ of the second output
fork(NULL /*Input: null pointer*/, ptrOut1 /*Output*/, ptrOut2 /*Output*/);
```

Listing 2: Call to actors *Swap* and *Fork* from Figure 9b if the buffer between the two actors is divided.

In Figure 7, the buffer corresponding to the output of the *Split* actor satisfies the conditions of Property 5.3.

5.2. Selection of Applicable Matches

Properties 5.1, 5.2, and 5.3 give the necessary conditions that a match must satisfy to be applicable. Using these properties, the purpose of the minimization process is to select and apply as many matches as possible in order to minimize the memory footprint of an application.

Algorithm 1 gives the pseudo-code of the optimization process. The purposes of the different parts of this optimization process are detailed in following sections.

ALGORITHM 1: Optimization process for the application of buffer matches.

Input: B - set of buffers initialized as specified in Definitions 4.1 and 5.5
M - set of matches generated by memory scripts and satisfying rules R1 to R5
Output: $M_{applied}$: a set of applied matches

Separate $T = \langle B, M \rangle$ into independent match DAGs $DAGSet = \{T_i = \langle B_i, M_i \rangle\}$;
for each $T_i \in DAGSet$ **do**
  // Fold match DAG T_i
  repeat
    $M_{sel} \leftarrow \emptyset$;
    // A match $m = \langle b_{src}, r_{src}, b_{dst}, r_{dst} \rangle$ can be selected only if:
    // $\nexists m' \in M_{sel}$ such that $m'.b_{dst} = m.b_{src}$ or $m'.b_{src} = m.b_{dst}$
    $M_{sel} \leftarrow$ Select applicable match(es) in M_i ;
    Apply selected matches M_{sel} in T_i;
    $M_{applied} \leftarrow M_{applied} \cup M_{sel}$;
  until $M_i = \emptyset$ *or* $M_{sel} = \emptyset$;
endfor

5.2.1. Applying a match.

The heart of the optimization process is to apply the selected matches at line 9 of Algorithm 1. Applying a match consists of merging a buffer or, for a divided buffer, a sub-range into a target buffer. The merged sub-range can either be part of the source or the destination buffer of a match, merged in a target sub-range at the other end of this match (see Definition 4.2).

Applying a set of matches is a complex operation that requires many transformations in the corresponding match DAG. The following list describes the transformations resulting from the application of a match $m = \langle b_{src}, r_{src}, b_{dst}, r_{dst} \rangle \in M$ where b_{src} is the target buffer and b_{dst} is the merged buffer. All notations can thus be reversed ($src \leftrightarrows dst$) when applying a match where the target is the destination range.

Applying a match consists of:

— **Updating the target buffer properties:** The mergeable ranges r_{merge}, the indivisible ranges r_{div}, and the range of bytes r_{bytes} of the target buffer must be updated with the properties of the merged sub-range. Formally,

$$b_{src}.r_{bytes} \leftarrow [min(b_{src}.r_{bytes}.start, r_{src}.start), max(b_{src}.r_{bytes}.end, r_{src}.end)[$$

$$(b_{src}.r_{merge} \cap r_{src}) \leftarrow [b_{src}.r_{merge} \cap (b_{dst}.r_{merge})^{dst \xrightarrow{m} src}] \cap r_{src}$$

$$(b_{src}.r_{div} \cap r_{src}) \leftarrow [b_{src}.r_{div} \overset{lazy}{\cup} (b_{dst}.r_{div})^{dst \xrightarrow{m} src}] \cap r_{src}$$

Where notation $b_{src}.r_{merge} \cap r_{src}$ on the left-hand part of an assignation is used to specify that only the sub-range r_{src} of the mergeable range $b_{src}.r_{merge}$ is updated (that is, r_{src} can be seen as mask for the assignation).

— **Reconnecting matches of the merged sub-range:** The merged buffer is not necessarily a leaf of the match DAG. In other words, the merged sub-range of the destination buffer b_{dst} may itself be the source of other matches $m' \in M$. In such case, these matches should be reconnected to the target buffer as follows:

$$m'.b_{src} \leftarrow b_{src} \qquad \text{and} \qquad m'.r_{src} \leftarrow (m'.r_{src})^{dst \xrightarrow{m} src}$$

— **Removing conflicting matches, applied matches, and merged buffers:** If the applied match $m \in M$ is in conflict with other matches (see Properties 5.1 and 5.2), these other matches will no longer be applicable once m is applied. Inapplicable matches are thus removed from the match DAG. The merged buffer and the applied match no longer exist in the match DAG, and are also removed from it. Although these elements are no longer part of the match DAG, the optimization process keeps track of them to identify future violations of Properties 5.1 and 5.2.

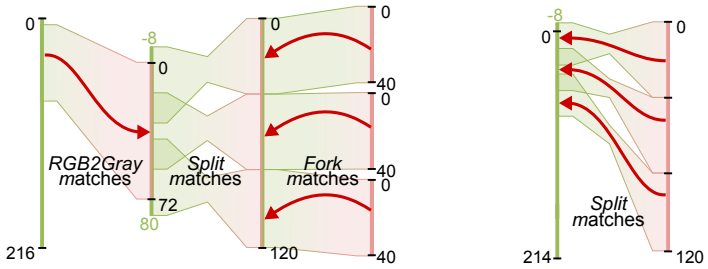
5.2.2. Match DAG Folding.

The match DAG folding algorithm is the iterative optimization process responsible for selecting the matches to apply (lines 2 to 11 of Algorithm 1). This process iterates until no applicable matches can be found in the match DAG.

The order in which the matches are applied is important to maximize the number of applied matches. In particular, matches that are not involved in any conflict should be applied first. Since applying such matches does not require the removal of any conflicting match from the match DAG, applying them first is a good way to maximize the number of applied matches. For example, if all but one output buffers of a *Broadcast* actor are mergeable, applying the match with the non-mergeable output first would be a bad choice since it would forbid the application of all other matches.

It is important to note that a match can be applied during an iteration only if neither its source nor its destination buffer is itself merged during the same iteration. Indeed,

when a first match is applied, the properties of the target buffer are updated in such a way that may prevent the application of a second match. For example, in the match DAG of Figure 8b, all matches are applicable during the first iteration of the folding algorithm. However, only one match connected to the output buffer of actor B can be applied at a time to avoid destination merge issue.



(a) 1^{st} iteration (b) 2^{nd} iteration

Fig. 10: Folding the match tree formed by *RGB2Gray-Split-Fork*

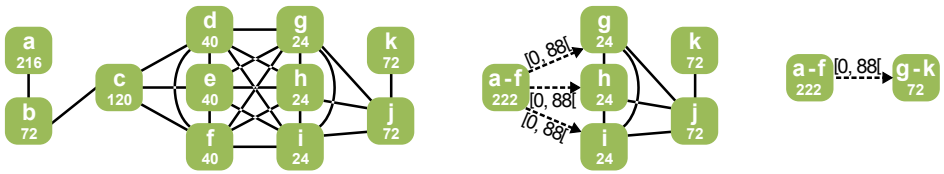
Figure 10 illustrates the processing of the folding algorithm on the match DAG from Figure 7. In the first iteration (Figure 10a), 4 matches can be applied simultaneously: the three output buffers of the *Fork* actor are merged into the output buffer of the *Split* actor, and the input buffer of the *RGB2Gray* actor is merged into the input buffer of the *Split* actor. As a result of this last merge, the input buffer of the *Split* actor is enlarged to cover a range of bytes $r_{bytes} = [0, 214[$. In the second iteration (Figure 10b), the output of the *Split* actor is divided into three sub-ranges, each corresponding to a previously merged output buffer of the *Fork* actor. These three sub-ranges are merged into overlapping sub-ranges of the input buffer of the *Split* actor. Since all matches of the match DAG were applied, the execution of the match DAG folding algorithm is terminated and a single buffer of 222 bytes remains in the folded match DAG. Without the merging technique, the allocation of each buffer in a separate memory space would require 528 bytes.

5.3. Static Memory Allocation of Merged Buffers

The static memory allocation of merged buffers in a shared-memory is realized using a memory reuse technique for SDF graphs presented in [Desnos et al. 2014]. This technique relies on the construction of a Memory Exclusion Graph (MEG) whose weighted vertices are the memory objects that must be allocated in memory to support the execution of the application. Two memory objects are connected with an exclusion if they can not be allocated in overlapping memory ranges. An exclusion is added between two memory objects if they have overlapping lifetime, that is, if they may store valid data simultaneously. For example two memory objects are linked with an exclusion if they represent two single-rate buffers from parallel data-paths that may contain data-tokens simultaneously. Two memory objects representing an input and an output buffer of the same actor may contain valid data simultaneously during the execution of this actor, they are thus linked with an exclusion.

5.3.1. Buffer Merging in the Memory Exclusion Graph (MEG).

Each memory object of the MEG presented in Figure 11a corresponds to a single-rate FIFO of the SDF graph from Figure 2. Updating a MEG with results from the buffer merging optimization process simply consists of merging memory objects that correspond to merged buffers. For example, in Figure 11b, memory objects a to f are merged into a single memory object of 222 bytes as a result of applying matches for the match



(a) MEG for the FIFOs from Figure 2 (b) 1^{st} update (c) 2^{nd} update

Fig. 11: Updates of the Memory Exclusion Graph (MEG).

DAG presented in Figure 10. Partial exclusions are then added between the merged memory objects and the other memory objects from the exclusion graph. In the example of Figure 11b, a partial exclusion is added between the first 88 bytes of the merged memory object *a-f* and memory objects *g*, *h*, and *i*.

Figure 11c illustrates the MEG obtained when memory objects *g* to *k* are merged according to matches associated with the *Join* actor and the in-place *Median* actor. The allocation of this MEG requires 222 bytes of memory since the exclusion between merged memory objects *a-f* and *g-k* is partial. Without the buffer merging technique, the allocation of the original MEG requires 288 byte of memory. The allocation of the same application with the FIFO sizing technique presented in [Stuijk et al. 2006a] requires 552 bytes.

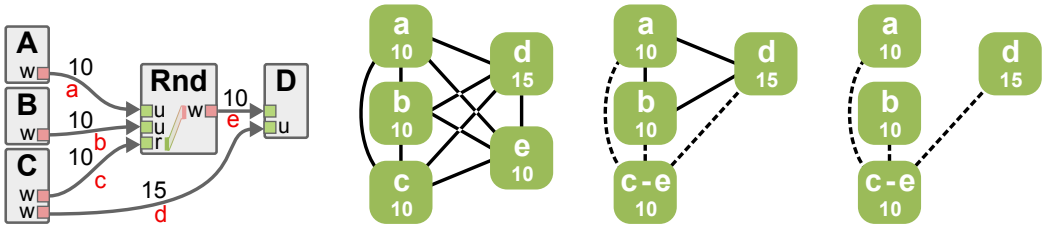
5.3.2. Removing Exclusions.

Port annotations *unused* and *write-only* can be used to specify that the data tokens of an input port of an actor will never be used, and to specify that the data tokens written on an output port will never be read again by their producer respectively. Hence, if a single-rate FIFO is connected both to a *write-only* output port and an *unused* input port, the data tokens written on this FIFO will never be used by the application and can be discarded as soon as they are produced. Even though single-rate FIFOs connected to a *write-only* and an *unused* port contain only useless data tokens, their allocation in memory is still required. Indeed, the actor producing the data tokens on this FIFO has no knowledge that these tokens will never be used, and so an output buffer must still be provided.

Since the allocation of *write-only* and *unused* FIFOs is mandatory, but their content is useless, all corresponding buffers can be allocated in overlapping memory spaces, regardless of their respective lifetimes. Indeed, if two *write-only* and *unused* buffers are allocated in overlapping memory ranges, the content of one buffer might be corrupted when the other content is written. However, this data corruption will not change the application behavior since the content of these buffers will never be used. This mechanism is especially useful for hierarchical SDF graphs [Piat et al. 2009] whose hierarchy flattening results in the insertion of many *Roundbuffer* actors with *unused* input ports.

To enable the allocation of memory objects corresponding to *write-only* and *unused* FIFOs in overlapping memory ranges, exclusions between them are simply removed from the MEG. Figure 12a gives an example of SDF graph with *unused* and *write-only* FIFOs. The MEG derived from this SDF graph is presented in Figure 12b. A first update of this MEG (Figure 12c) consists of merging memory objects c and e according to the matches created by the memory script of actor *Roundbuffer*. Then, a second update consists of removing exclusions between memory objects a, b and d that correspond to single-rate FIFOs connecting *write-only* ports to *unused* ports (Figure 12d).

Since all the memory objects of the original MEG exclude each other: they form a clique, so no memory reuse can be applied between them, thus requiring 55 bytes of memory for their allocation. After the first update, the remaining memory objects still form a clique and so 45 bytes are needed for their allocation. Finally, after the last update, memory objects a, b, and d are no longer linked by an exclusion and can be



(a) SDF graph. r, w, and u mark *read-only*, *write-only*, and *unused* ports respectively.

(b) Original MEG

(c) 1^{st} update: merging c and e.

(d) 2^{nd} update: removing exclusions between a, b, and d.

Fig. 12: Removing exclusion between *write-only* and *unused* memory objects of a MEG.

allocated in overlapping memory ranges. Hence, only 25 bytes of memory are needed to allocate the MEG of Figure 12d. On this application, applying the buffer merging technique leads to a reduction of the memory footprint by 55%.

The next Section presents experimental results of our buffer merging technique when applied to more complex applications.

6. EXPERIMENTS

The buffer merging technique presented in this article was implemented within the open-source rapid prototyping framework PREESM [Pelcat et al. 2014]. To assess the performance of this buffer merging technique, a set of SDF specifications of state-of-the-art computer vision, telecommunication and signal processing applications were used. Table I presents the characteristics of the SDF graphs of these applications. The *Sobel* application is the example used throughout this article and illustrated in Figure 1. The *Stereo* application is a stereo-matching computer vision algorithm whose purpose is to extract 3D information from a pair of 2D images [Mercat et al. 2014]. The *Chaotic* application is a generator of pseudo-random sequences used for cryptography purposes and described in [El Assad and Noura 2013]. The *LTE PRACH* application is a description of the preamble detection part of the Long Term Evolution (LTE) wireless communication standard [Pelcat et al. 2012]. Finally, the *Large FFT* application is an implementation of an FFT algorithm optimized for processing a very large number of samples (65536 complex values in our SDF implementation) [Takahashi 2000].

Table I: SDF graphs used in our experiments.

	SDF graph		Associated scripts		Single-rate SDF		Executed scripts	
Application	Actors	FIFOs	Custom	Special	Actors	FIFOs	Custom	Special
Sobel	6	5	3	0	10	11	3	2
Stereo	28	42	4	10	300	987	62	83
Chaotic	10	14	6	3	29	45	16	12
LTE PRACH	14	10	2	1	398	543	88	93
Large FFT	9	8	6	0	780	1544	771	6

In Table I, the *SDF graph* column presents the number of actors and FIFOs in the SDF graphs of the applications. The *Associated scripts* column gives the number of actors of the graphs that are associated with custom and special memory scripts. Custom scripts are the memory scripts written by the application developer to specify the memory reuse opportunities for user-defined actors written specifically for an application whereas special scripts corresponds to memory scripts associated with actors

with a pre-defined behavior such as *Fork*, *Broadcast*, and *Roundbuffer* actors. It is interesting to note that on average, more than half of the actors of an SDF graph can be associated with a memory script.

The *Single-rate SDF* column gives the number of actors and FIFOs obtained after applying the single-rate transformation. A large increase of the number of actors and FIFOs compared to the *SDF graph* column is indicative of a high degree of the data parallelism implicitly embedded in the original SDF graph, and explicitly revealed during the single-rate SDF transformation. Except for the *Sobel* application that is a simple example used throughout this paper, all other applications have a degree of parallelism greater than 8 for the most computation intensive actors. When deploying the applications on a multicore architecture, this degree of parallelism ensures that all processing elements will execute a fair amount of computations.

The *Executed scripts* column presents the number of executions of both custom and special memory scripts. The numbers presented in this column are much larger than those presented in the *Associated scripts* column because each script associated with an actor of the original SDF graph may be executed several times after the duplication of actors during the single-rate transformation. Special actors that are automatically inserted in SDF graphs during the single-rate transformation also increase the number of executed memory scripts.

Table II: Memory footprints for different memory optimization techniques.

Application	None	FIFO sizing[1]	FIFO sizing[1] + Brd. FIFO[2]	Buffer min.[3]	MEG[3]	Buff. Merg.	MEG[4] + Buff. Merg.
Sobel	672 B	552 B	552 B	288 B	288 B	294 B	222 B
Stereo	1452 MB	373 MB	35 MB	– MB	1256 MB	170 MB	23 MB
Chaotic	6.93 MB	2.15 MB	2.15 MB	1.67 MB	2.63 MB	1.92 MB	1.91 MB
LTE PRACH	5.81 MB	– MB	– MB	– MB	1.10 MB	2.12 MB	1.04 MB
Large FFT	3074 kB	1794 kB	1794 kB	512 kB	514 kB	257 kB	257 kB

[1]: [Stuijk et al. 2006a] [2]: [Mamidala et al. 2011] [3]: [Geilen et al. 2005] [4]: [Desnos et al. 2014]

Table II presents the memory footprints allocated for the mono-core execution of 5 applications with 6 different optimization techniques. Footprints given in the *None* column are obtained when no memory optimization technique is used and each buffer of the single-rate SDF graph is allocated in an dedicated memory space. The *FIFO sizing* and the *FIFO sizing + Brd. FIFO* columns are obtained with a state-of-the-art FIFO sizing technique [Stuijk et al. 2006a] implemented in the SDF For Free (SDF3) framework [Stuijk et al. 2006b]. For the results of the third column, broadcast FIFOs were also used [Mamidala et al. 2011] to further reduce the allocated memory footprints. Results of the fourth column were obtained with the buffer minimization technique allowing reuse of memory for the allocation of FIFOs [Geilen et al. 2005] (Section 3). Results of the second, third and fourth columns were obtained by applying the minimization techniques to the SDF graphs of application before applying the single-rate transformation. The results of the fifth column are obtained by exploiting only graph-level memory reuse opportunities [Desnos et al. 2014]. Results of the sixth and the seventh columns were allocated using the new buffer merging technique presented in this article. The memory reuse opportunities offered by the updated MEG were exploited only for the last column. This FIFO sizing and the buffer minimization techniques model the memory optimization problem as a state-space exploration problem that can become too large to be solvable for complex SDF graphs. For this reason, several cells of Table II have been left empty for the *LTE PRACH* and *Stereo* applications. The results presented in this table show the efficiency of the proposed buffer merg-

ing technique which allocates on average 48% less memory than the combination of state-of-the-art FIFO sizing and broadcast FIFOs techniques.

An application of special interest is the *Large FFT* application that can be executed *in-place* when the buffer merging technique is applied. Indeed, 256 kB of memory corresponds to the exact amount of memory needed to store the 65536 complex values, where each complex value is stored with two 16-bit fixed point numbers. The extra kilobyte allocated for the application corresponds to the memory needed to store iteration indexes and constants.

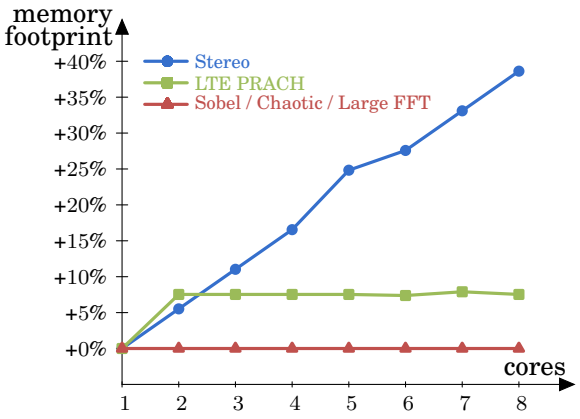


Fig. 13: Memory footprints of the 5 applications on a variable number of cores.

The buffer merging technique was also used to allocate memory for a mapping of the applications on an 8-core shared-memory MPSoC. Figure 13 shows the evolution of the memory footprints of the 5 applications when mapped and scheduled on a variable number of cores. In this graph, the memory footprints of the applications are expressed relative to the memory allocated for a mono-core execution.

The buffer merging process presented in this article is independent of scheduling considerations as it is solely based on the precedence relationships defined in the single-rate SDF graph. The growth of memory footprints that can be observed in Figure 13 is caused by the MEGs of applications that are updated with scheduling information to reflect the additional data parallelism that is exploited when applications are deployed on multiple cores. Since the buffer merging technique and the MEG-based memory optimization have no impact on scheduling decisions, all the parallelism embedded in SDF graphs is exploited to improve the performance of applications when mapping them on multicore architectures [Desnos et al. 2014]. Since this paper focuses on memory optimizations, the performance improvements of applications mapped on multiple cores is not within the scope of this paper. Nevertheless, it should be noted that since the degree of parallelism of these applications is higher than the number of cores to which they are mapped, each core receives an equitable amount of computation and the performance of these application is greatly increased when mapping them on a multicore architecture.

Despite the additional data parallelism of applications mapped on multiple cores, for a worst case scenario, up to 40% additional memory is required to allocate the applications from Table I on 8 cores. It is important to note that the buffer merging technique means that the memory footprint allocated for the *Sobel*, the *Chaotic*, and the *Large FFT* applications remains constant irrespective of the number of cores. This is due to the fact that large majority of buffers within the application have been merged by the

buffer merging technique, thus leaving few reuse opportunities to the later MEG-based optimization. Since the buffer merging technique is independent of scheduling considerations, the allocated memory footprints for these applications are not impacted by the mapping on multiple cores. For all applications, the memory footprint allocated on 8 cores remains lower than the footprints allocated by state-of-the-art FIFO sizing and broadcast FIFOs techniques for a mono-core execution. These results show that the proposed buffer merging technique is more efficient than state-of-the-art techniques to minimize the memory footprint of real applications specified with SDF graphs. The tools[1] and most applications[2] used for these experiments are open-source and can be freely downloaded in order to reproduce the presented results.

7. CONCLUSION

In this paper, a new buffer merging technique is introduced to minimize the memory footprints of DSP applications specified with an SDF graph. This technique is based on graph annotations that allow the developer to specify merging opportunities between input and output buffers of actors. This paper details how these annotations are converted into a formal specification of the buffer merging problem. This formal specification then serves as a basis both for checking the validity of the specified buffer merging patterns and for the memory minimization process whose purpose is to apply as many merging opportunities as possible while preserving the application behavior. Experiments on a set of real DSP applications have shown that our technique results on average in memory footprints 48% smaller than state-of-the-art optimization techniques. Experiments also showed that the proposed buffer merging technique limits the growth of the memory footprint of applications when deploying them on a multi-core architecture. Future work on this subject will include the automated creation of memory-scripts through an analysis of the source code of actors.

REFERENCES

Oliver Jakob Arndt, Daniel Becker, Christian Banz, and Holger Blume. 2013. Parallel implementation of real-time semi-global matching on embedded multi-core architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*. IEEE, 56–63.

Mohamed Benazouz, Olivier Marchetti, Alix Munier-Kordon, and Pascal Urard. 2010. A new approach for minimizing buffer capacities with throughput constraint for embedded system design. In *Computer Systems and Applications (AICCSA), 2010 IEEE/ACS*. IEEE, 1–8.

Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. 1994. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation* Vol. 4 (1994), 155–182.

C. Sidney Burrus and Peter W. Eschenbacher. 1981. An in-place, in-order prime factor FFT algorithm. *Acoustics, Speech and Signal Processing, IEEE Transactions on* 29, 4 (Aug 1981), 806–817.

Loïc Cudennec, Paul Dubrulle, François Galea, Thierry Goubier, and Renaud Sirdey. 2014. Generating Code and Memory Buffers to Reorganize Data on Many-core Architectures. *Procedia Computer Science* 29 (2014), 1123–1133.

Eddy De Greef, Francky Catthoor, and Hugo De Man. 1997. Array placement for storage size reduction in embedded multimedia systems. In *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*. IEEE, 66–75.

Karol Desnos, Maxime Pelcat, Jean-François. Nezan, and Slaheddine Aridhi. 2014. Memory Analysis and Optimized Allocation of Dataflow Applications on Shared-Memory MPSoCs. *Journal of Signal Processing Systems, Springer US* 80 (July 2014), 19–37.

Karol Desnos, Maxime Pelcat, Jean-François. Nezan, and Slaheddine Aridhi. 2015. Buffer Merging Technique for Minimizing Memory Footprints of Synchronous Dataflow Specifications. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP15)*. IEEE, 1111–1115.

[1] PREESM website: http://preesm.sf.net and SDF3 website: http://www.es.ele.tue.nl/sdf3/

[2] Stereo, Sobel and Large FFT PREESM applications: https://github.com/preesm/preesm-apps

S. El Assad and H. Noura. 2013. Generator of chaotic sequences and corresponding generating system. (Feb. 2013). Patent No. US 8781116 B2, Filed March 28th., 2011, Issued Jul. 15th., 2014.

Janet Fabri. 1979. *Automatic storage optimization*. Courant Institute of Mathematical Sciences, NY University.

Scott Fischaber, Roger Woods, and John McAllister. 2007. SoC Memory Hierarchy Derivation from Dataflow Graphs. In *Signal Processing Systems, Workshop on*. IEEE, 469–474.

J. Forget, C. Gensoul, M. Guesdon, C. Lavarenne, C. Macabiau, Y. Sorel, and C. Stentzel. 2013. *SynDEx v7 User Manual*. INRIA Paris-Rocquencourt. http://www.syndex.org/v7/manual/manual.pdf.

Viliam Geffert and Jozef Gajdoš. 2011. In-Place Sorting. In *SOFSEM 2011: Theory and Practice of Computer Science*, I. Černy, T. Gyimthy, J. Hromkovič, K. Jefferey, R. Krlović, M. Vukolić, and S. Wolf (Eds.). Lecture Notes in Computer Science, Vol. 6543. Springer Berlin Heidelberg, 248–259.

Marc Geilen, Twan Basten, and Sander Stuijk. 2005. Minimising Buffer Requirements of Synchronous Dataflow Graphs with Model Checking. In *Design Automation Conference*. ACM, NY, USA, 819–824.

Chia-Jui Hsu, Ming-Yung Ko, and Shuvra S. Bhattacharyya. 2005. Software Synthesis from the Dataflow Interchange Format. In *Proceedings of the 2005 Workshop on Software and Compilers for Embedded Systems (SCOPES'05)*. ACM, 37–49.

Kalray. 2013. Many-core processors - Dataflow. (Dec. 2013). http://www.kalray.eu/technology/dataflow/

Jong T. Kim and Dong R. Shin. 2002. New efficient clique partitioning algorithms for register-transfer synthesis of data paths. *Journal of the Korean Phys. Soc* 40 (2002), 754–758.

Edward A. Lee and David G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (sept. 1987), 1235 – 1245.

Edward A. Lee and Thomas M. Parks. 1995. Dataflow process networks. *Proc. IEEE* 83, 5 (1995), 773–801.

David P. Magee. 2005. Matlab Extensions for the Development, Testing and Verification of Real-time DSP Software. In *Proceedings of the 42nd Annual Design Automation Conference*. ACM, NY, USA, 603–606.

Amith R. Mamidala, Daniel Faraj, Sameer Kumar, Douglas Miller, Michael Blocksome, Thomas Gooding, Philiph Heidelberger, and Gabor Dozsa. 2011. Optimizing MPI Collectives Using Efficient Intra-node Communication Techniques over the Blue Gene/P Supercomputer. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), International Symposium on*. IEEE, 771–780.

Alexandre Mercat, Jean-François Nezan, Daniel Menard, and Jinglin Zhang. 2014. Implementation of a Stereo Matching Algorithm Onto a Manycore Embedded System. In *International Symposium on Circuits and Systems*. IEEE, 1296–1299.

Praveen K. Murthy and Shuvra S. Bhattacharyya. 2004. Buffer Merging: a Powerful Technique for Reducing Memory Requirements of Synchronous Dataflow Specifications. *Trans. Des. Autom. Electron. Syst.* 9, 2 (April 2004), 212–237.

Patrick Niemeyer. 2014. BeanShell website. (2014). http://www.beanshell.org.

Maxime Pelcat, Slaheddine Aridhi, Jonathan Piat, and Jean-Franois Nezan. 2012. *Physical Layer Multi-Core Prototyping: A Dataflow-Based Approach for LTE eNodeB*. Springer.

Maxime Pelcat, Karol Desnos, Julien Heulot, Clement Guy, Jean-François. Nezan, and Slaheddine Aridhi. 2014. PREESM: A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming. In *EDERC 2014 Proceedings*. IEEE, 36.

Jonathan Piat, Shuvra S. Bhattacharyya, and Mickaël Raulet. 2009. Interface-based hierarchy for synchronous data-flow graphs. In *SiPS Proceedings*. IEEE, 145 – 150.

Russel W. Quong and Shu-Ching Chen. 1993. *Register Allocation via Weighted Graph Coloring*. ECE Technical Reports. 232. Register Allocation via Weighted Graph Coloring.

Sander Stuijk, Marc Geilen, and Twan Basten. 2006a. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Design Automation Conference, Proceedings*. ACM, 899–904.

Sander Stuijk, Marc Geilen, and Twan Basten. 2006b. SDF^3: SDF For Free. In *Conference on Application of Concurrency to System Design, Proceedings (ACSD '06)*. IEEE Computer Society, 276–278.

Daisuke Takahashi. 2000. High-performance parallel FFT algorithms for the HITACHI SR8000. In *High Performance Computing in the Asia-Pacific Region. Proceedings.*, Vol. 1. IEEE, 192–199.

Matthieu Wipliez and Mickaël Raulet. 2010. Classification and transformation of dynamic dataflow programs. In *Design and Architectures for Signal and Image Processing (DASIP),*. IEEE, 303–310.

Yervant Zorian. 2002. Embedded memory test and repair: infrastructure IP for SOC yield. In *Test Conference, 2002. Proceedings. International*. IEEE, 340–349.

Received February 2015; revised July 2015; accepted October 2015