BUFFER MERGING TECHNIQUE FOR MINIMIZING MEMORY FOOTPRINTS OF SYNCHRONOUS DATAFLOW SPECIFICATIONS

Karol Desnos[⋆] *Maxime Pelcat*[⋆] *Jean-François Nezan*[⋆] *Slaheddine Aridhi*[†]

[⋆] IETR, INSA Rennes, UMR CNRS 6164, UEB
Rennes, France
email: kdesnos, mpelcat, jnezan@insa-rennes.fr

[†] Texas Instruments France
Cagnes-Sur-Mer, France
email: saridhi@ti.com

ABSTRACT

This paper introduces and assesses 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 SDF actors, and script-based specifications of merging opportunities between input and output buffers of actors. An automated optimization process is used to exploit these buffer merging opportunities and to minimize the memory footprints of applications. Experimental results on a computer vision application show a reduction of the memory footprint by 34% compared to state-of-the-art minimization techniques.

Index Terms— Dataflow, Memory, Buffer Merging

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 [1], the multimedia and the telecommunication domains [2]. 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 [3].

This paper presents a new memory optimization technique for applications specified with the Synchronous Dataflow (SDF) Model of Computation (MoC). The SDF MoC models an application as a directed graph of computational entities, called actors, that exchange data through a network of First-In First-Out queues (FIFOs) [4]. Each time an actor is executed, or fired, it consumes and produces a fixed quantum of data, called data token, on the FIFOs to which it is connected. An example of SDF graph with 5 image processing actors is given in Figure 1. Edges of this graph are labeled with their consumption and production rates. The popularity of the SDF MoC is due to its great analyzability and its natural expressivity of the parallelism of DSP applications which makes it particularly suitable to exploit the parallelism offered by Multiprocessor Systems-on-Chips (MPSoCs).

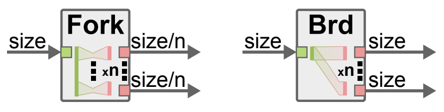
SDF actors are considered as "black boxes" within the model whose internal behavior can be implemented in any programming language. To simplify the description of this internal behavior, it is convenient to assume that the memory consumed and produced on each FIFO during the firing of an actor constitutes a contiguous memory space called a buffer [5]. To reveal these buffers, 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 equal (Figure 2). Each single-rate FIFO is a buffer of fixed size accessed by two actors. In most SDF programming frameworks [6, 7], memory optimization consists of graph-level minimization of the memory allocated to the FIFOs [8, 9]. Because internal data dependencies of actors are generally unknown to these frameworks, these dependencies cannot be used for optimization purposes. In particular, because the order in which an actor accesses its input and output buffers is unknown, these buffers are assumed to contain valid data simultaneously, and must always be allocated in non-overlapping memory spaces.

The purpose of the technique presented in this paper is to relax this constraint by allowing the application developer to explicitly specify merging opportunities between input and output buffers of actors. Examples of merging opportunities are presented in Figure 3. The purpose of *Fork* actors (Figure 3a), that are inserted during single-rate transformations, is to distribute equal parts of the data received in their input buffer to their output buffers. By allocating each 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. The purpose of the *Broadcast* actor (Figure 3b) is to copy the content of its input buffer into each output buffer. By merging n output buffers with the input buffer, the memory allocated for a *Broadcast* actor can be divided by *n+1*.

Previous work on buffer merging techniques for SDF graphs is presented in Section 2. Section 3 introduces new graph annotations enabling the specification of buffer merging opportunities, and Section 4 presents the automated minimization process that uses these opportunities. Finally, an experimental evaluation of the buffer merging technique on a state-of-the-art computer vision application is presented in Section 5.

2. RELATED WORK

To our knowledge, minimizing the memory footprint of SDF applications is usually achieved by using FIFO sizing techniques [8, 10] that consist of finding a schedule that minimizes the memory space allocated to each FIFO of an SDF graph. Contrary to the technique



(a) Fork actor (b) Broadcast actor

Fig. 3: Internal data dependencies of SDF actors.

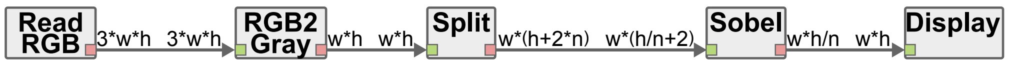


Fig. 1: Image processing SDF graph

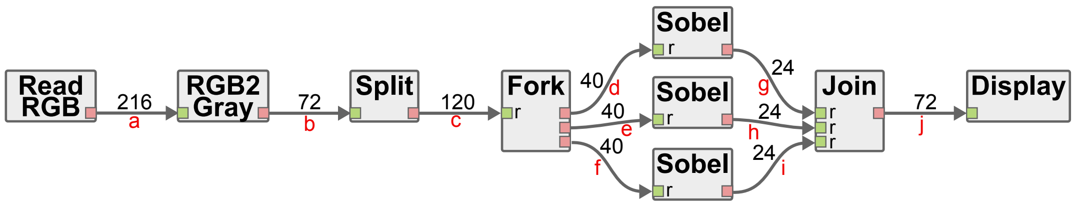


Fig. 2: Single-rate SDF graph derived from the SDF graph of Figure 1 (with w=8, h=9, and n=3).

presented in this paper, FIFO sizing techniques do not consider merging opportunities between input and output buffers.

Several solutions to *broadcast* data tokens can be found in the literature. Non-destructive reads, or FIFO peeking, is a well-known way to read data tokens without popping them from FIFOs, hence avoiding the need for *Broadcast* actors [11]. 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 [12]. The drawback of this solution is that it also requires a modification of the SDF MoC semantics.

In [5], a technique is proposed to enable buffer merging for a set of actors with pre-defined behavior. Contrary to the method presented in this paper, this technique does not allow buffer merging for actors with a user-defined behavior. The allocation of input and output buffers of an actor in overlapping memory spaces has been studied in [13, 14]. In [13], 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 [14, 15], another annotation system is 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 that require merging several output buffers into a single input. Moreover, the optimization technique presented in [13] 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 buffer merging technique presented in this paper does not require any modification of the SDF semantics. Contrary to existing techniques, this buffer merging technique can be used for any number of input and output buffers.

3. GRAPH ANNOTATIONS

In addition to the application single-rate SDF graph, 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 buffers, and annotations of the ports of SDF actors.

3.1. Memory Scripts

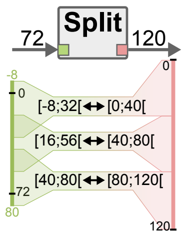
The objective of memory scripts is to allow the application developer to specify for a given actor which input buffer can be merged with which output buffer, and what the relative position of the merged buffers is. To this purpose, each actor of the SDF graph can optionally be associated with a memory 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.

Figure 4 presents the memory script associated with the *Split* actor and illustrates the matches resulting from its execution. Memory scripts are written with a derivative of the Java language called BeanShell [16]. Matches created for a *Join* actor and a *Broadcast* actor are presented in Figure 5.

```
/* 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. 4: Memory script for *Split* actor (with w=8, h=9, and n=3).

As illustrated in this example, it is possible to match a contiguous sub-range of a buffer into non-contiguous sub-ranges, and to match sub-ranges partially outside the original range of bytes of another buffer. The original range of bytes of a buffer corresponds to the bytes comprised between the first (`index=0`) and the last (`index=`$\texttt{buffer}_{size}$`-1`) bytes of this buffer, with $\texttt{buffer}_{size}$ the production or consumption rate of the corresponding SDF port. Any byte indexed outside this range does not belong to the original range of bytes.

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 of a match must cover the same number of bytes.
- **R2.** A match can only be created between an input buffer and an output buffer.
- **R3.** A sub-range of bytes of an output buffer can not be matched several times by overlapping matches.
- **R4.** A match must involve at least one byte from the original range of bytes of both buffers with which it connects.
- **R5.** Only bytes within the original range of bytes of their buffer can be matched with bytes falling outside the original range of bytes of the matched buffer.

Rule R1 enforces the validity of the matches. 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* (cf. Section 4.1). Rules R4 and R5 limit the creation of matches with bytes falling outside of the original range of bytes of buffers. Without these rules, an input buffer could be merged completely out of the original range of bytes of an output buffer, thus resulting in no memory reuse between the two buffers.

3.2. Read-only Ports

As illustrated by the *Split* and the *Broadcast* actors, 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, actors accessing the merged buffers must not write in this shared memory. If one of the consumer actors does not respect this condition, its corresponding output buffer should not be merged and it should be given a private copy of the data.

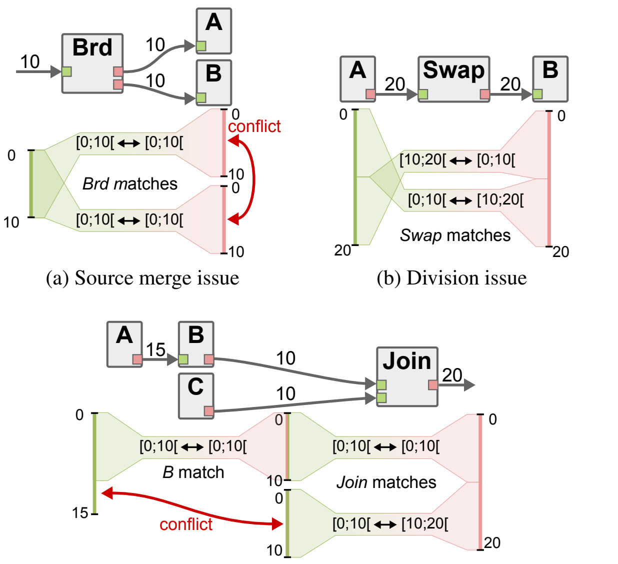
By default, the most flexible actor behavior is assumed and all actors are supposed to be both writing to and reading from all their input buffers. This assumption forbids the application of overlapping matches. A `read-only` annotation has been introduced, and can be associated with any input ports of an SDF graph by its developer. The actor possessing a `read-only` input port can only read data from this port. Like a `const` variable in C, 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. In the SDF graph of Figure 2, an `r` mark is associated with each `read-only` port.

4. MINIMIZATION PROCESS

The execution of memory scripts produces a list of matches that represent merging opportunities for the input and output buffers of actors. The purpose of the memory minimization process is to apply as many of these matches as possible.

4.1. Potential Merging Issues

A match is said to be applicable if its application does not change the behavior of the application. The 5 matching rules presented in Section 3.1 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.



(a) Source merge issue (b) Division issue

(c) Destination merge issue

Fig. 5: Conflicts preventing the application of matches.

Figure 5 gives examples of matches that respect the rules presented in Section 3.1 but that can not be applied without corrupting the application behavior.

Source merge issue: Matches with overlapping input sub-ranges can be applied only if their output buffers are connected to *read-only* ports. In Figure 5a, only one of the *Broadcast* matches can be applied because neither actor *A* nor actor *B* have a `read-only` input port. If both matches were applied, actors *A* and *B* would write in the input buffer of the other and corrupt the application behavior.

Destination merge issue: A chain of matches cannot be applied if it results in merging several input sub-ranges in overlapping output sub-ranges. In Figure 5c, if all matches were applied, input buffers of actors *B* and *Join* would be merged in range $[0, 15[$ and $[10, 20[$, respectively, of the output buffer of the *Join* actor. In this scenario, if actor *A* is fired after actor *C*, then actor *A* will partially overwrite the data tokens produced by actor *C*, thus corrupting the application behavior.

Division issue: As illustrated by the matches of actor *Swap* in Figure 5b, contiguous sub-ranges of bytes can be matched with non-contiguous ranges of bytes, thus requiring a division of the buffer to apply these matches. 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. Hence, a buffer can be divided into non-contiguous sub-ranges only if all actors accessing this buffer can still access all its sub-ranges. In Figure 5b, for the matches to be applicable, either actor *A* or actor *B* should be associated with a memory script dividing the swapped buffer in two.

4.2. Selection of Applicable Matches

The compile-time minimization process responsible for selecting the matches to apply can be divided into the following steps:

1. Combine the results of the memory scripts for all actors into a tree of buffers and matches.
2. Select a subset M_{sel} of applicable matches of the tree that have no conflict with each other.
3. Apply matches in M_{sel}.
4. Remove from the tree all matches that were in conflict with matches from M_{sel}.

5. Repeat steps 2 to 4 until no match is applicable.
6. Allocate memory according to merging decisions.

Figure 6 gives an overview of the execution of the minimization process for the single-rate SDF graph from Figure 2. Figure 6a presents the match tree obtained by combining the 6 buffers and the 7 matches associated to actors *RGB2Gray*, *Split*, and *Fork*. Since matches created by the *RGB2Gray* and *Fork* actors are applicable and have no conflicts, they can be applied during the first iteration of the minimization process. The match tree with 2 merged buffers resulting from their application is presented in Figure 6b. During the second iteration of the process, matches created by the *Split* actor are applied. The result of the minimization process for this match tree is a unique merged buffer of 222 bytes.

4.3. Static Memory Allocation of Merged Buffers

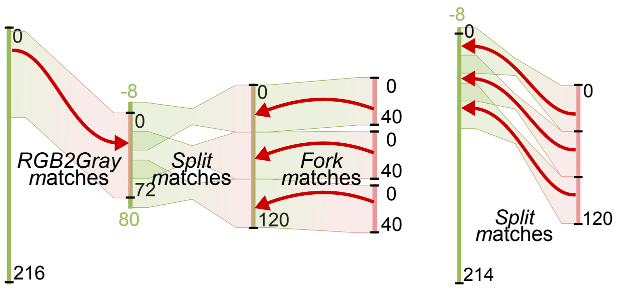
The static memory allocation of merged buffers in a shared-memory is realized using a memory reuse technique for SDF graphs presented in [9]. This technique relies on the construction of a Memory Exclusion Graph (MEG) whose weighted vertices are the memory objects that must be allocated in memory to support the execution of the application. Two vertices are connected with an exclusion if they can not be allocated in overlapping memory ranges.

Each memory object of the MEG presented in Figure 7a corresponds to a single-rate FIFO of the SDF graph from Figure 2. Updating a MEG with results from the optimization process simply consists of merging memory objects that correspond to merged buffers. For example, in Figure 7b, memory objects *a* to *f* are merged into a single memory object of 222 bytes as a result of the application of matches for the match tree presented in Figure 6. Partial exclusions are then added between the merged memory objects and the other memory objects from the exclusion graph. In the example of Figure 7b, 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 7c illustrates the MEG obtained when memory objects *g* to *j* are merged according to matches associated to the *Join* actor. The allocation of this MEG requires 222 bytes of memory since the exclusion between merged memory objects *a-f* and *g-j* is partial. Without the buffer merging technique, the allocation of the original MEG requires 288 bytes of memory. The allocation of the same application with the FIFO sizing technique presented in [8] requires 480 bytes. The next Section presents experimental results using a more complex computer vision application.

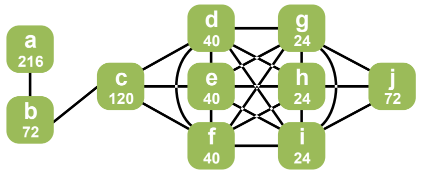
5. EXPERIMENTS

We implemented the buffer merging technique in the open-source rapid prototyping framework PREESM [17], and applied it to the SDF specification of a state-of-the-art stereo-matching algorithm [18]. A stereo-matching algorithm is a computer vision application whose purpose is to extract the 3D information from a pair of 2D images. All presented results are obtained for images of 450*375 pixels. The SDF graph of this application contains 16 distinct actors, some of which can be fired up to 60 times in parallel. The time spent by the developer to write the scripts for this application is less than 30 minutes. The MEG of this application contains 1000 memory objects requiring 1452 MB for their allocation without any optimization.

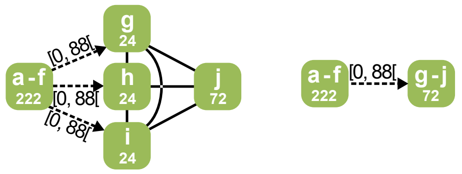


(a) Match tree and 1st set of applicable matches (b) 2nd set

Fig. 6: Application of matches



(a) Original MEG for the FIFOs from Figure 2



(b) 1st update (c) 2nd update

Fig. 7: Updates of the Memory Exclusion Graph (MEG).

Technique	[9]	[8]	Buff. Merg.	[8] + Brd. FIFO	[9] + Buff. Merg.
Footprint	1256 MB	373 MB	170 MB	35 MB	23 MB

Table 1: Memory footprints for different allocation techniques.

Table 1 presents the memory footprints allocated for the mono-core execution of the stereo-matching application with different optimization techniques. The first footprint is obtained by exploiting only graph-level memory reuse opportunities [9]. The second and the fourth footprints are obtained with a FIFO sizing technique [8]. For the fourth footprint, broadcast FIFOs were also used [12]. The third and the fifth footprints were allocated using the new buffer merging technique. The memory reuse opportunities offered by the MEG [9] were exploited only for the fifth footprint. The results presented in this table show the efficiency of the proposed buffer merging technique which allocates 34% less memory than the best combination of state-of-the-art techniques.

The buffer merging technique was also used to allocate memory for a mapping of the stereo-matching application on a quad-core MPSoC. Despite the additional data parallelism in this scenario, only 28 MB of memory were allocated, which is less than the footprint allocated by state-of-the-art techniques for a mono-core execution.

6. CONCLUSION

In this paper, we have proposed a new buffer merging technique to minimize the memory footprints of DSP applications specified with an SDF graph. Our technique is based on graph annotations that allow the developer to specify merging opportunities between input and output buffers of actors. Experiments on a computer vision application have shown that our technique results in a memory footprint 34% smaller than state-of-the-art optimization techniques. Future work on this subject will include the automated creation of memory-scripts through an analysis of the source code of actors.

7. REFERENCES

[1] O.J. Arndt, D. Becker, C. Banz, and H. Blume, "Parallel implementation of real-time semi-global matching on embedded multi-core architectures," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), International Conference on*, 2013.

[2] M. Pelcat, S. Aridhi, J. Piat, and J.-F. Nezan, *Physical Layer Multi-Core Prototyping: A Dataflow-Based Approach for LTE eNodeB*, Springer, 2012.

[3] Y. Zorian, "Embedded memory test and repair: infrastructure ip for soc yield," in *Test Conference, 2002. Proceedings. International*, 2002, pp. 340–349.

[4] E.A. Lee and D.G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, sept. 1987.

[5] L. Cudennec, P. Dubrulle, F. Galea, T. Goubier, and R. Sirdey, "Generating code and memory buffers to reorganize data on many-core architectures," *Procedia Computer Science*, vol. 29, pp. 1123–1133, 2014.

[6] Electronic Systems Group Technical University of Eindhoven, "Sdf for free (sdf3)," Mar. 2013, http://www.es.ele.tue.nl/sdf3/.

[7] C.-C. Shen, L.-H. Wang, I. Cho, S. Kim, S. Won, W. Plishker, and S. Bhattacharyya, "The dspcad lightweight dataflow environment: introduction to lide version 0.1," Tech. Rep., U. of Maryland, 2011.

[8] 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 annual Design Automation Conference (DAC)*. ACM, 2006, pp. 899–904.

[9] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi, "Pre-and post-scheduling memory allocation strategies on mpsocs," in *Electronic System Level Synthesis Conference (ESLsyn), Proceedings*, 2013.

[10] M. Benazouz, O. Marchetti, A. Munier-Kordon, and P. Urard, "A new approach for minimizing buffer capacities with throughput constraint for embedded system design," in *Computer Systems and Applications (AICCSA), IEEE/ACS*, 2010.

[11] S. Fischaber, R. Woods, and J. McAllister, "Soc memory hierarchy derivation from dataflow graphs," in *Signal Processing Systems, 2007 IEEE Workshop on*, 2007, pp. 469–474.

[12] A.R. Mamidala, D. Faraj, S. Kumar, D. Miller, M. Blocksome, T. Gooding, P. Heidelberger, and G. Dozsa, "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), IEEE International Symposium on*, May 2011, pp. 771–780.

[13] P. Murthy and S. Bhattacharyya, "Buffer merging: a powerful technique for reducing memory requirements of synchronous dataflow specifications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 9, no. 2, pp. 212–237, Apr. 2004.

[14] C.-J. Hsu, M.-Y. Ko, and S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the 2005 Workshop on Software and Compilers for Embedded Systems (SCOPES'05)*. 2005, pp. 37–49, ACM.

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

[16] P. Niemeyer, "Beanshell website," 2014, http://www.beanshell.org.

[17] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "PREESM: A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming," in *EDERC 2014 Proceedings*, Sept. 2014, p. 36.

[18] A. Mercat, J.-F. Nezan, D. Menard, and J. Zhang, "Implementation of a Stereo Matching Algorithm Onto a Manycore Embedded System," in *International Symposium on Circuits and Systems*, 2014.