

Model-Based Synthesis and Optimization of Static Multi-Rate Image Processing Algorithms

Joachim Keinert*, Hritam Dutta†, Frank Hannig†, Christian Haubelt† and Jürgen Teich†

*Fraunhofer IIS, Digital Cinema Department, Erlangen, Germany

Email: ket@iis.fraunhofer.de

†Hardware/Software Co-Design, Department of Computer Science, University of Erlangen-Nuremberg, Germany

Email: {dutta,hannig,haubelt,teich}@cs.fau.de

Abstract—High computational effort in modern image processing applications like medical imaging or high-resolution video processing often demands for massively parallel special purpose architectures in form of FPGAs or ASICs. However, their efficient implementation is still a challenge, as the design complexity causes exploding development times and costs. This paper presents a new design flow which permits to specify, analyze, and synthesize complex image processing algorithms. A novel buffer requirement analysis allows exploiting possible tradeoffs between required communication memory and computational logic for multi-rate applications. The derived schedule and buffer results are taken into account for resource optimized synthesis of the required hardware accelerators. Application to a multi-resolution filter shows that buffer analysis is possible in less than one second and that scheduling alternatives influence the required communication memory by up to 24% and the computational resources by up to 16%.

I. INTRODUCTION

As design complexity is becoming a major barrier for technical progress because of expensive and error-prone development, new design methodologies raising the level of abstraction are becoming increasingly popular. Simulink [1] or SystemC based high-level synthesis [2] tools for instance permit to compose complex systems by communicating blocks. However, these approaches do not allow for system-level analysis like determination of required communication buffer sizes, as the blocks can contain arbitrarily complex operations. Alternative approaches like [3], [4] are restricted to a subset of sequential languages like C. However, extraction of the contained parallelism is challenging, especially as analysis on individual statements can get computationally expensive [5].

In order to address these aspects, this paper presents a novel design flow for high-level synthesis of complex multi-rate image processing applications containing up- and downsamplers. It extends existing previous work by usage of lattice-based buffer analysis which considers different scheduling alternatives for multi-rate systems. As the obtained results are directly taken into account during hardware synthesis, we are able to exploit tradeoffs between required communication memory and computational logic. Furthermore, in contrast to many other approaches, analysis of the overall system does not rely on solving *Integer Linear Programs (ILPs)* in case of acyclic problems. Instead ILPs are only required for local analysis like actor synthesis or dependency calculation in order to assure good scaling properties of our design flow.

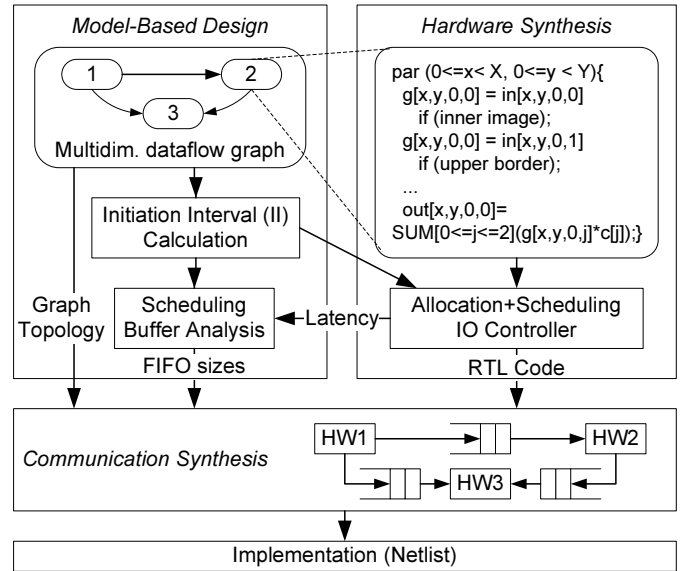


Fig. 1. Proposed design flow

Application to a multi-resolution filter used in medical imaging [6] shows the benefits of our approach.

Fig. 1 illustrates the major steps of the proposed design flow. The application is modeled by help of a hierarchical multi-dimensional data flow graph. This not only helps to handle the system complexity, but also naturally represents the application parallelism. From this model the minimum execution periodicity, also called *initiation interval*, is calculated for each actor. The determined values are used during synthesis in order to exploit resource sharing while meeting throughput requirements. In order to allow for high clock frequencies, a pipelined RTL implementation is generated. Its latency is taken into account during graph scheduling which is performed as a prerequisite for buffer analysis. The latter one enables quick determination of the required communication memory by usage of lattice representations. In the final step, the overall system is assembled by instantiation of the generated hardware accelerators interconnected by FIFOs having the calculated size.

The remainder of the paper is as follows: Section II introduces the challenges of multi-rate image processing systems and discusses further related work. Sections III and IV describe

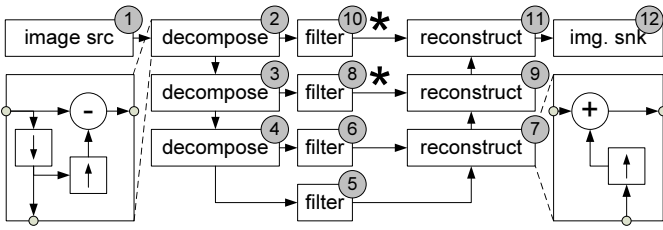


Fig. 2. Block diagram of multi-resolution filter

how application modeling allows for efficient buffer analysis. Hardware synthesis is discussed in Section V, followed by presentation of the obtained results in Section VI.

II. PROBLEM FORMULATION AND RELATED WORK

In order to show the benefits of our design methodology we have chosen a *multi-rate* image processing algorithm in form of a *multi-resolution filter* as it requires complex filters interconnected by a non-trivial communication topology. This not only leads to huge calculation efforts attaining ≈ 44 giga-operations per second for 2048×2048 images at $30fps$, but also makes determination of the required buffer sizes challenging.

Fig. 2 shows the block-diagram of a 4-stage multi-resolution filter. On each level, the input image is decomposed into a difference image and a downsampled version. After filtering with a *bilateral filter* [7], the *reconstruct block* recombines the corresponding images. However, due to the successive decomposition and filtering operations, the downsampled image arrives much later at the input of the reconstruct block than the image of bigger size. The latter one has hence to be delayed on the edge marked by an asterisk leading to huge memory requirements.

Unfortunately determination of the required buffer sizes is pretty time consuming even for an experienced designer, because the occurring pixel latency is not only influenced by the bilateral filter, but also by the occurring up- and downsampling. Even worse, as shown later on in Section IV-C, these operations allow deployment of different implementation alternatives for the filters influencing both the chip size of the hardware accelerators and the required communication memories. This however makes not only buffer analysis very hard, but also hardware synthesis, as different alternatives of the accelerators have to be provided.

Consequently, in order to make implementation less error prone and more efficient, several new design methodologies have been proposed. Data flow models of computation allow for efficient determination of system throughput [8] and required buffer sizes [9], [10]. However, these approaches restrict to one-dimensional streams of data whereas image processing applications communicate multi-dimensional image arrays. In particular for buffer analysis this information can be advantageously exploited [11], [12], [13], [14]. However, none of these approaches considers synthesis of hardware accelerators and the special properties of multi-rate systems. Ref. [15] explicitly addresses multi-rate systems, but restricts to one-dimensional down- and upsamplers and does not consider

buffer size determination.

The most similar work to our approach has been proposed in [16]. However, in contrast to this paper they do not consider multi-rate systems. Furthermore, their approach requires ILPs whose size increases with the number of processes contained in the system. As their solution have exponential complexity, we deploy a heuristic which only requires local and thus small ILP programs.

III. APPLICATION MODELING

Similar to the block diagram shown in Fig. 2, our design flow depicted in Fig. 1 represents the application as a *hierarchical multi-dimensional data flow graph* $G = (V, E)$ [17]. Its *vertices* $v \in V$ represent the *actors* incorporating the *system functionality* while the edges $e \in E$ model *inter-module communication*. The separation of these two aspects is very beneficial for the design of complex systems, as the vertices can be synthesized independently of each other. Furthermore, system analysis is simplified significantly because it is possible to use an abstract view provided by the *communication semantics* of the data flow model instead of considering internal details of the vertices.

a) Communication Semantics: Each edge $e \in E$ of the multi-dimensional data flow graph represents the transport of multi-dimensional arrays of size $\vec{u} \in \mathbb{N}^n$ from the edge *source* to its *sink*, $n \in \mathbb{N}$ being the number of image dimensions. These arrays however are not produced and consumed as a whole. Instead each source *invocation* generates a so called *token* which consists of $\vec{p} \in \mathbb{N}^n$ pixels. The so produced array is sampled by the sink with possibly overlapping windows whose size is given by $\vec{c} \in \mathbb{N}^n$. The distance between two consecutive windows is defined by $\Delta\vec{c} \in \mathbb{N}^n$.

Fig. 3 illustrates the communication parameters per edge assuming a vertical downsampler. For each invocation, the source generates a single data element ($\vec{p} = (1, 1)^T$). The resulting image is sampled by the downsampler with a sliding window of size $\vec{c} = (1, 3)^T$. It moves by one pixel in horizontal direction, but by two pixels in vertical direction ($\Delta\vec{c} = (1, 2)^T$) in order to achieve the downsampling. As in realistic applications the sliding window can transcend the image extensions, the pixel array can be virtually extended with a border as illustrated in Fig. 3 by the gray shading. This border is not produced by the source, but the actual pixel values are determined by the deployed border processing algorithm like symmetric mirroring or constant value extension.

b) System Functionality: Each vertex of the graph represents a process, also called *actor*, whose functionality is described by a nested loop program. The kernel of the loop nest contains the computations which read from the input ports, processes the data, and provides the results on the output ports. Each execution of the kernel corresponds to one position of the sliding window and output token described above. Hence, it can only be executed when enough input data and free space for the output data are available.

In order to select the currently processed pixels, the input and output ports are accessed with a vector having $2 \cdot n$

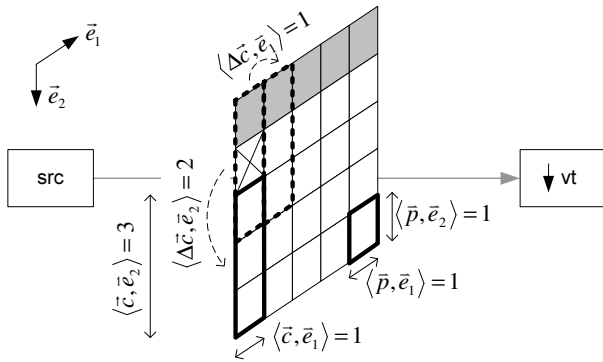


Fig. 3. Communication parameters

dimensions when processing n -dimensional images. The first n dimensions indicate the current window position, while the next n parameters correspond to the pixel coordinates relative to the window borders. Taking for instance Fig. 1, $\text{in}[0, 0, 0, 1]$ addresses the first pixel in the second row of the upper left sliding window as illustrated by the cross in Fig. 3.

IV. BUFFER ANALYSIS

Based on the introduced communication semantics, this section presents a method for fast determination of the required buffer sizes. As these not only depend on the size of the sliding windows, but also on the time instance an actor can execute, buffer analysis requires determination of an *actor schedule*.

This paper proposes a lattice-based approach which exploits the regularity of the problem for fast analysis. Its major idea is to embed all actors into a common *invocation grid* such that their relative execution time is determined. This allows for efficient dependency analysis and thus determination of the required buffer sizes. Note that the final hardware does not directly implement the determined actor schedule. Instead, a *self-timed schedule* is deployed in which the actor invocations are controlled by the availability of input data and free output space. The property of monotonic execution for data flow graphs [9] guarantees that no deadlock occurs and that the throughput attains at least the value determined during analysis.

The following subsections detail the individual steps for buffer analysis consisting in (i) lattice embedding including grid scaling and (ii) buffer size calculation. Subsection IV-C takes special care about the possible actor schedule alternatives in multi-rate systems.

A. Lattice Embedding

Starting from the multi-dimensional data flow graph, it is possible to derive for each actor the number of invocations in each dimension. Taking for instance the example given in Fig. 3, we can see that the source actor executes 5×4 times whereas the sink performs only 5×2 invocations. Each of these invocations can be represented as a point in an n -dimensional grid, also known as *bounded lattice*.

Fig. 4(a) illustrates the principles assuming the example given in Fig. 3. The gray shaded semicircles belong to the

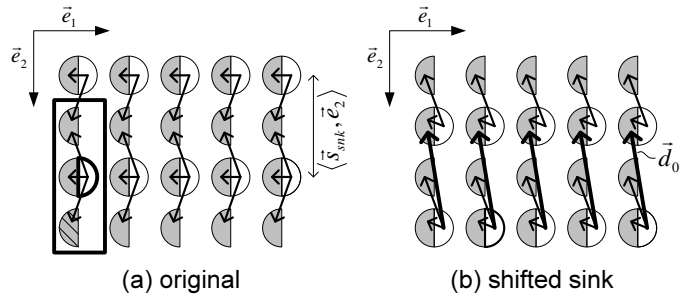


Fig. 4. Lattice representation of Fig. 3

source invocations while the sink invocations are represented by white color. The arrows illustrate data dependencies and will be discussed later on. Note that the extended image border is not represented in this picture, as it is not produced by the source (see Section III). All invocations are executed in row major order from left to right and from top to bottom.

The embedded lattices thus define the relative execution order required for correct buffer analysis. However, in order to obtain reasonable actor schedules, the lattices belonging to the different actors have to be scaled correctly. In case of the example shown in Fig. 3, the downsampler actor performs only half the invocations in vertical direction compared to the source. This is because the sliding window moves by two pixels while the source generates only one pixel per invocation. Consequently, the sink invocation points have to be scaled by factor two in comparison to the source grid as already done in Fig. 4(a). In general the corresponding scaling factor in dimension i can be calculated from the window movement $\Delta \vec{c}$ and the produced token size \vec{p} :

$$\langle \vec{s}_{snk}, \vec{e}_i \rangle = \frac{\langle \Delta \vec{c}, \vec{e}_i \rangle}{\langle \vec{p}, \vec{e}_i \rangle} \in \mathbb{Q}$$

After grid scaling, the *dependency vectors* can be determined as shown in Fig. 4(a). They define which source invocations a given sink invocation depends on. The rectangle corresponds to the lower left sliding window depicted in Fig. 3 and belongs to the bold sink invocation in Fig. 4(a). The dependency vectors point to the source invocations which generate the corresponding window pixels. Note that due to border processing the first row of sink invocations only require two input dependencies as border pixels are not produced by the source.

Based on those dependency vectors it is possible to construct valid schedules by taking care that no pixel is read before being produced. Unfortunately, this is not automatically the case. In Fig. 4(a) for instance, the bold sink execution depends on the striped source invocation although the latter one is executed in the future due to the row-major execution order of the lattice. Mathematically this corresponds to *anti-lexicographically positive* dependency vectors $\vec{d} \in \mathbb{Q}^n$ for which the following holds: $\exists i : \langle \vec{d}, \vec{e}_i \rangle > 0 \wedge \forall j > i : \langle \vec{d}, \vec{e}_j \rangle \geq 0$.

Thus, in order to construct valid schedules, the sink has to be delayed such that no dependency vector is anti-

lexicographically positive. Fig. 4(b) exemplarily depicts the result of this operation when applied to Subfigure 4(a). Additional shifting is possible in order to take the actor latency derived by the hardware synthesis (see Section V) into account: Assuming for instance that the result of the source actor is only available after the time equivalent of two invocations, we would additionally shift the sink lattice by two in direction \vec{e}_1 in order to avoid reading of illegal data.¹ Furthermore, this operation allows parallel execution of the coinciding sink and source lattice points.

If a sink actor disposes of several input edges, then it must be embedded such into the common lattice that none of the corresponding dependency vectors is anti-lexicographically positive. For a graph containing cycles however this is anything but easy, as actors have to be placed into the lattice while not all of their predecessors are already embedded. Typically this is solved by complex heuristics like [18] or even solution of ILPs [16].

In image processing, however, often multi-dimensional data flow graphs without feedback loops occur which can be analyzed much more efficiently. In this case, we establish a topological order such that each actor is visited after its sources. The numbers in Fig. 2 illustrate such a sequence when ignoring that some actors are hierarchical. Based on this topological order, the lattice grids of all actors can be easily scaled and shifted as described above, because all predecessors are processed before their successors.²

B. Buffer Size Calculation

Once the relative execution order of all actors is determined, we can derive for each edge its associated buffer size. The latter one is determined by the dependency vectors as they indicate for each sink lattice point the earliest source invocation whose data are still required. Thus, all pixels produced since this source execution up to the considered sink invocation have to be stored in the edge buffer. Hence, intuitively the required memory size is determined by that dependency vector which spans the most lattice points. Mathematically this corresponds to the anti-lexicographically minimal dependency vector \vec{d}_{min} . Note that $\langle \vec{d}_{min}, \vec{e}_n \rangle \leq 0$. The buffer size calculates as

$$m = \left(\sum_{i=1}^n \left[d_i^* \cdot \prod_{j=1}^{i-1} \langle \vec{r}_{src}, \vec{e}_j \rangle \right] + 1 \right) \cdot \underbrace{\prod_{i=1}^n \langle \vec{p}, \vec{e}_i \rangle}_{(2)} \quad (1)$$

$$d_i^* = \frac{\langle -\vec{d}_{min}, \vec{e}_i \rangle}{\langle \vec{s}_{src}, \vec{e}_i \rangle} \in \mathbb{N}$$

n represents the number of image dimensions. \vec{s}_{src} stands for the grid scaling factor of the source actor derived in Section IV-A. The division is necessary in order to take into

¹In order to increase efficiency of the generated schedule, our prototyping tool wraps around sink invocations which transcend the original grid. However, due to space restrictions further details are omitted.

²Graphs with multiple sources require an extended effort in order to obtain small buffer sizes. Although implemented in our prototyping tool, further details are omitted due to space restrictions.

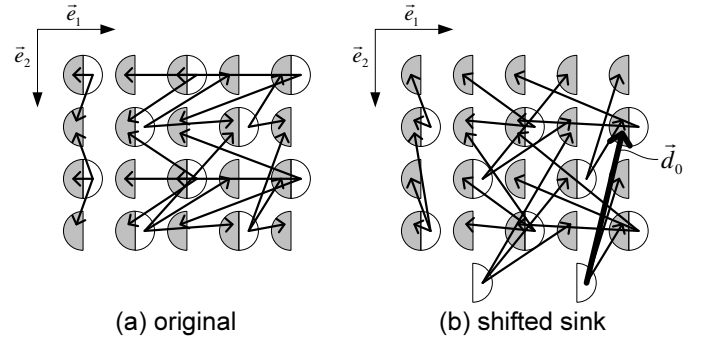


Fig. 5. Situation of Fig. 4 after load smoothing

account that due to grid scaling not all lattice points correspond to an actor invocation. Hence, they do not contribute to the required buffer size. \vec{r}_{src} represents the number of actor invocations in each dimension. The value of product (2) corresponds to the number of data elements which a produced token consists of.

Example 1: In Fig. 4(b) we find $\vec{d}_{min} = (0, -2)^T$ and $\vec{s}_{src} = (1, 1)^T$. Together with $n = 2$, $\vec{p} = (1, 1)^T$ and $\vec{r}_{src} = (5, 4)^T$ we obtain for the buffer size $m = \left(0 \cdot 1 + \frac{-(-2)}{1} \cdot 5 + 1 \right) = 11$. This corresponds to two lines and one pixel which is indeed the minimum buffer size for a 1×3 downsampler.

Note that for $d_i^* \notin \mathbb{N}$ a more complex formula has to be deployed. Due to space restrictions, this however is out of scope for this paper.

C. Multi-Rate Analysis

After having introduced the principles of lattice-based buffer analysis, this section will describe how these techniques can be extended in order to incorporate load smoothing for efficient synthesis.

For motivation, we consider again Fig. 4. There we can clearly see that the workload of the downsampler is not equally distributed over time, but shows bursty behavior: The phases where each source invocation also induces execution of the downsampler are followed by idle lines. For complex actors like the bilateral filter however, this bursty behavior leads to increased hardware resources due to a resulting small *initiation interval* (Π). It corresponds to the time between two actor invocations and amounts to one for the actor schedule illustrated in Fig. 4. However, the same system throughput would be possible when deploying an initiation interval of two, thus increasing the potential for hardware resource sharing.

This effect can be taken into account during buffer analysis by redistributing the sink lattice points. Fig. 5 exemplarily illustrates the results for the downsampler actor. Its workload is now equally balanced, because it is executed only each second source invocation ($\Pi = 2$). However, this has to be paid by less regular dependency vectors which can be calculated by help of an ILP. As it only depends on the parameters of a single edge, the complexity scales well with increasing size of the multi-dimensional data flow graph. For Fig. 5, its solution leads to $\vec{d}_0 = (1, -3)^T$. By help of (1) we can thus

derive that the required communication memory has increased to $m = 3 \cdot 5 - 1 \cdot 1 + 1 = 15$ resulting in an increase of approx. 36%. Hence, the capability to analyze and synthesize both actor schedule alternatives permits to exploit a trade-off between communication memory and hardware requirements for the accelerator. To the best of our knowledge, this has not been done before.

V. HARDWARE SYNTHESIS

After presentation of the buffer analysis and its ability to incorporate load smoothing techniques, this section discusses the synthesis step of our design flow. It is steered by the determined initiation intervals in order to generate resource optimized hardware accelerators with the requisite throughput. This is done by applying a sequence of high-level transformations to the actor loop description (see Fig. 1), scheduling, and RTL generation as discussed in the next subsections.

A. Operation Scheduling

As the bilateral filter [7] is the most computational intensive algorithm in Fig. 2, it is chosen for explaining the principles of our high-level synthesis. The algorithm processes the image with a sliding window of size 3×3 using filter coefficients which depend on the image content. This leads to a large number of arithmetic operations (44 MUL, 16 ADD, 9 SUB, 9 EXP, 1 DIV, ...) for each output pixel. The boundary pixels need symmetric extension for acquiring the requisite inputs.

The corresponding calculation rules and the occurring data dependencies are described in form of an actor loop description as exemplified in the upper right corner of Fig. 1. It contains the iteration bounds representing the computation domain. The border processing is defined by conditional statements. From this loop description, the high-level synthesis must derive the required hardware resources and schedule the operations on them. For this purpose, the loop actor description is analyzed in order to build the dependence graph containing all variables and operations. Compact representation is possible by usage of a polytope model which allows for efficient parallelization and hardware generation. Based on this dependence graph the operation scheduling and allocation is performed via *mixed integer linear programming*. It considers resource constraints, data dependencies, and the required initiation interval (II) determined by model-based analysis as described in Section IV-A.

The obtained *operation schedule* gives the allocated resources as well as the execution times of the loop iterations and the contained operations. Furthermore, it allows calculating the iteration latency, also called actor latency, which corresponds to the number of clock cycles required to output the first result pixel after all necessary input data are available. The corresponding value is taken into account during buffer analysis as described in Section IV-A.

B. Architecture Synthesis

The allocation and scheduling information determined in the previous section is used to automatically derive the RTL implementation of the accelerator which is then retargeted

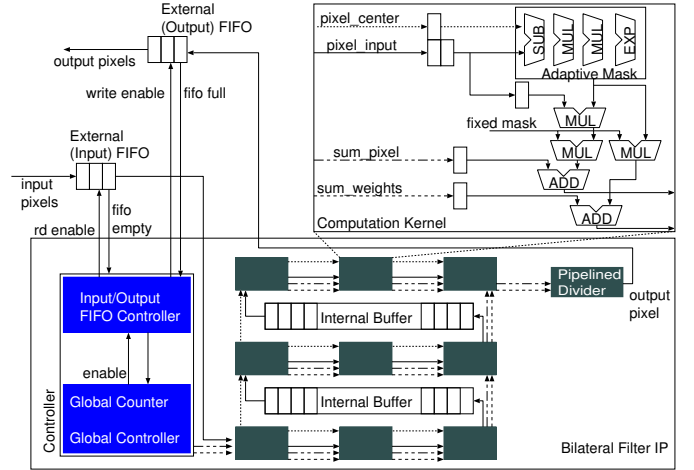


Fig. 6. Bilateral Filter Architecture (II=1)

to VHDL. Fig. 6 exemplarily shows the generated hardware architecture of the bilateral filter. It consists of three parts, namely (i) the computation kernel, (ii) the internal delay buffers, and (iii) the controller.

The computation kernel performs the arithmetic operations and deploys heavy pipelining such that several pixels can be processed in parallel. It instantiates parameterizable components like adders or dividers which are available in an IP library. Similarly to handcrafted designs, the internal delay buffers temporarily store the input pixels such that each of them is read only once in order to avoid I/O bottlenecks. Their size can be calculated from the static operation schedule and the underlying dependencies. The controller is responsible for keeping track of the image pixels being processed and orchestrates the correct computations and I/O. For this purpose, the global counter generates the coordinates of the currently processed pixel from which the global controller derives several control signals. They select which conditional statement to execute such that correct border processing and I/O access is possible.

Communication with the predecessor and successor actors is performed on pixel granularity by help of external FIFOs. Together with the internal delay buffers they build the required memory buffer whose size is determined by the analysis described in Section IV. The pixel coordinates provided by the global counter allow generation of the correct read and write enable signals for the input and output FIFOs. In case the input FIFO is empty or the output FIFO is full the accelerator is stopped by the I/O controller.

VI. RESULTS

In order to illustrate the benefits of the proposed design flow, this section presents its results when mapping the multi-resolution filter depicted in Fig. 2 to Xilinx FPGAs.

Table I shows the results of our buffer analysis for 512×512 images and varying number of filter instances by presenting the required memory size for all communication buffers. Two different schedule variants leading to *bursty* (Fig. 4) or *smoothed* (Fig. 5) actor load are distinguished. Depending on

TABLE I
BUFFER ANALYSIS RESULTS (OVERALL BUFFER)

#stages	bursty	smoothed	increase
2	22628	25712	13.6 %
4	94092	114096	21.2 %
5	182720	225744	23.5 %

TABLE II
FPGA RESOURCE CONSUMPTION

		Flip Flops	Look-up Tables	Multipliers
2	bursty	38229	37153	292
	smoothed	34770	35167	278
	increase	-9.0%	-5.3%	-4.8%
4	bursty	67351	71780	454
	smoothed	60144	67877	392
	increase	-10.7%	-5.4%	-13.7%
5	bursty	80494	88748	522
	smoothed	73335	86691	438
	increase	-8.9%	-2.3%	-16.1%

the number of filter blocks we measured up to 24% increased buffer requirements when switching to the smoothed schedule strategy.

On the other hand this offers a possibility to trade memory like Xilinx Block RAMs against the required computational hardware resources. The latter ones are shown in Table II by adding up the synthesis results for all filter, decompose, and reconstruct blocks when using 32 bit fixed point arithmetic. Evaluation has shown that 90% of the required hardware resources are assigned to the bilateral filters because they contain complex arithmetic operations like divisions and exponential functions. As this leads to important possibilities for resource sharing, the smoothed schedule reduces the FPGA resources between 2.3% and 16.1% while achieving the same system throughput. This is, because the bursty schedule requires initiation intervals of 1, 2, 4, . . . for the filters in the different levels whereas for the smoothed implementation 1, 4, 16, . . . is sufficient. Consequently more resources can be shared because the number of pixels which have to be processed per clock cycle decreases. The latency measured for the bilateral filter amounts 159 cycles for an initiation interval of one due to the presence of dividers and exponential units. It stays almost constant for the other initiation intervals.

Table III finally shows the run-time analysis of the proposed buffer analysis and compares it against some related work. It demonstrates its capacity for fast analysis even when the number of graph actors and edges is large. This is due to the renouncement of integer linear programming for the determination of the overall schedule together with model-based analysis. [15] and [5] in contrast use fine-grained analysis together with solution of ILPs making it computationally more expensive. IMEM [16] achieves similar results, but performs ILP scheduling which scales less well. Ref. [11] also reports similar analysis speeds, but direct comparison is difficult as they do not deploy model-based design. Furthermore, both publications do not consider the particularities of multi-rate systems.

TABLE III
RUNTIME FOR BUFFER ANALYSIS

	# Actors	# Edges	Time	CPU
MMAAlpha [15]	6	8	0.4s	1.7GHz
MMAAlpha [15]	24	22	59.8s	1.7GHz
CRP [5]	7	10	73s	1.4GHz
IMEM [16]	12	16	<1s	?
ours	79	92	0.8s	3GHz

VII. CONCLUSIONS

This paper introduced a novel design flow which eases design of complex image processing systems by (i) increasing the level of abstraction, (ii) high-level synthesis of parallel hardware accelerators and (iii) automatic determination of the required buffer sizes. The underlying analysis permits to trade required buffer sizes against logic resources while attaining similar analysis speed than the best known algorithms. Future work entails extension of our design flow for MPSoC systems which also include dynamic algorithms.

REFERENCES

- [1] The MathWorks, "Simulink," www.mathworks.com/.
- [2] Forte Design Systems, "Cynthesizer," www.forted.com. [Online]. Available: <http://www.forted.com/products/cynthesizer.asp>
- [3] Synfora, "Pico Express," www.synfora.com.
- [4] H. Ziegler and M. Hall, "Evaluating heuristics in automatically mapping multi-loop applications to FPGAs," in *FPGA*, 2005, pp. 184–195.
- [5] P. Feautrier, "Scalable and structured scheduling," *Int. J. Parallel Program.*, vol. 34, pp. 459–487, 2006.
- [6] F. Zhang, Y. M. Yoo, L. M. Koh, and Y. Kim, "Nonlinear diffusion in laplacian pyramid domain for ultrasonic speckle reduction," *IEEE Trans. Med. Imaging*, vol. 26, pp. 200–211, 2007.
- [7] H. Dutta, F. Hannig, J. Teich, B. Heigl, and H. Hornegger, "A Design Methodology for Hardware Acceleration of Adaptive Filter Algorithms in Image Processing," in *Proceedings of IEEE 17th International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, Sep. 2006, pp. 331–337.
- [8] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *DAC*, 2006, pp. 899–904.
- [9] M. Wiggers, M. Bekooij, and G. Smit, "Efficient computation of buffer capacities for cyclostatic dataflow graphs," University of Twente, Tech. Rep., Nov. 2006.
- [10] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *J. of VLSI Signal Processing Systems*, vol. 21, pp. 151–166, 1999.
- [11] Q. Hu, A. Vandecappelle, P. G. Kjeldsberg, F. Catthoor, and M. Palkovic, "Fast memory footprint estimation based on maximal dependency vector calculation," in *DATE*, 2007, pp. 379–384.
- [12] F. Balasa, P. G. Kjeldsberg, M. Palkovic, A. Vandecappelle, and F. Catthoor, "Loop transformation methodologies for array-oriented memory management," in *ASAP*, 2006, pp. 205–212.
- [13] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Trans. on Comp.*, vol. 54, pp. 1242–1257, 2005.
- [14] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and automated multiprocessor system design, programming, and implementation," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 27, pp. 542–555, 2008.
- [15] F. Charot, M. Nyamsi, P. Quinton, and C. Wagner, "Modeling and scheduling parallel data flow systems using structured systems of recurrence equations," in *ASAP*, 2004, pp. 6–16.
- [16] N. Lawal, M. O'Nils, and B. Thörnberg, "C++ based system synthesis of real-time video processing systems targeting FPGA implementation," in *IPDPS*, 2007, pp. 1–7.
- [17] J. Keinert, C. Haubelt, and J. Teich, "Modeling and analysis of windowed synchronous algorithms," *ICASSP*, vol. III, pp. 892–895, 2006.
- [18] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor, "Multi-dimensional incremental loop fusion for data locality," *ASAP*, pp. 17–27, 2003.