Bit-Width Constrained Memory Hierarchy Optimization for Real-Time Video Systems

Benny Thörnberg, Martin Palkovic, Qubo Hu, Leif Olsson, Per Gunnar Kjeldsberg,
Mattias O'Nils and Francky Catthoor

***Abstract*— The great variety of pixel dynamics of real-time video processing systems, ranging from color, grayscale or binary pixels, means that a careful design and specification of bit-widths is required. It is obvious that the bit-width specification will affect the total memory storage requirement. However, what is not so obvious is that the bit-width specification will also affect the design of the memory hierarchy, an impact similar for both hardware and software implementations. We have developed an Integer Non Linear Program (INLP) formulation for the optimization of the memory hierarchy of real-time video processing systems. An active surveillance video camera is introduced as a test case. We demonstrate how the optimization model can reduce the on-chip memory storage by 61 percent compared to a non optimal memory hierarchy.**

***Index Terms*— Bit-width, memory hierarchy, polyhedral, video**

I. Introduction

In Digital Signal Processing (DSP) systems huge amounts of information are processed in real-time. Memory accesses are the biggest contributor to power consumption in these systems, which makes the optimization of memory structures and memory access the key design challenge in achieving cost effective implementations for embedded applications [1], [2].

Embedded digital signal processing systems in general and video processing systems in particular, are often described using a data flow graph (DFG) as the dominant programming model. The nodes in a DFG capture the computation while the edges capture the data flow dependencies. This is sometimes known as functional modeling because no control information (i.e. in what sequence the DFG is traversed) is included in the model. When the number of data tokens produced and consumed on all edges is known at compile-time, the DFG is said to be synchronous (SDFG). The SDFG model was developed and extensively used by Lee [3]. Fig. 1A depicts an example of a simple SDFG. Operations *A* and *B* are both applied to the input data stream. The resulting intermediate data streams are fed into the last operation *C* where the output data stream is computed.

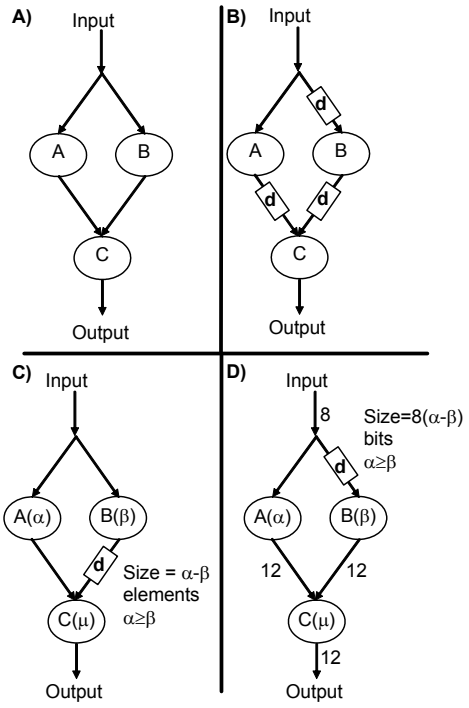


Fig. 1. The SDFG programming model and buffering of intermediate results.

The problem of calculating intermediate data storage and retiming an SDFG will now be exemplified for a simple one-dimensional DSP-system. However, the work presented in this paper targets multidimensional DSP-systems. For example, allow three operations to be scheduled in the order *A*-*B*-*C*. See Fig. 1B. This means that the input data read into operation *A* must also be stored in a buffer attached to the input of operation *B*. The intermediate results from operations *A* and *B* must then be stored in buffers attached to the inputs of operation *C* (all buffers are labeled *d*). This sequential traversal of the SDFG allows for an aggressive sharing of computational resources among the operations [6].

Manuscript received October X, 200X. The work in this paper was supported by the Swedish KK-foundation.

B. Thörnberg, Leif Olsson and Mattias O'Nils are with the Mid Sweden University, Sundsvall, Sweden. (e-mail: benny.thornberg@miun.se; leif.olsson@miun.se; mattias.onils@miun.se).

Qubo Hu and P. G. Kjeldsberg are with the Norwegian University Science and Technology, Trondheim, Norway. (e-mail: qubo.hu@iet.ntnu.no; per.gunnar.kjeldsberg@iet.ntnu.no).

M. Palkovic and Francky Catthoor are with IMEC, Leuven, Belgium. (e-mail: palkovic@imec.be; catthoor@imec.be).

In Fig. 1C, we now assume that the operations A, B, and C can be pipelined and executed in parallel. The latencies, given in number of clock cycles, are annotated to the operations as α,β and μ. The throughput of all operations is assumed to be one data token per clock cycle, which means that a complex scheduler is unnecessary. Thus, all nodes are executed at every clock cycle. The input data stream fed into both operations *A* and *B* must propagate and appear at operation *C* at exactly the same clock cycle. In this example, $\alpha \geq \beta$, and a buffer (shift register) of size α-β elements is inserted at the output of operation *B* as shown in Fig. 1C. The sequential scheduling problem has thus, for this parallel execution, been turned into a timing problem where data elements must appear on the operation inputs at exactly the right clock cycle. Hence, the buffers are now delay elements used to solve this timing problem. These buffers must be placed at locations within the SDFG such that the total memory storage requirement is minimized.

In Fig. 1D, bit-widths are also annotated to the input-intermediate- and output streams as 8 and 12 bits. This bit-width information motivates a change of the buffer location from *B*'s output stream in Fig. 1C to its input stream in Fig. 1D. This allocation is chosen because the buffer size will then be reduced by a third, from $12(\alpha-\beta)$ to $8(\alpha-\beta)$ bits. This storage size reduction is an example of a drastic effect on the hardware implementation caused by a crucial design refinement step [4]. At this refinement step, floating-point data types are converted into there fixed-point representation, where the number of used bits must be carefully selected. At the same time, the gain in hardware complexity must be balanced against subjective- and objective quality aspects inherited from precision- and truncation effects [5].

The main contribution of this paper is an optimization model that minimizes the storage size of all layers in a memory hierarchy for Real-Time Video Processing Systems (RTVPS) taking the bit-width information into account. The next section will introduce the reader to related techniques used for system modeling and memory optimization.

A. Related modeling and design automation techniques

IMEM [7],[8], is an extension of SystemC [9] that supports the SDFG programming model for RTVPS. An RTVPS is captured in IMEM as a coarse-grained multi-rate synchronous data flow graph. All actors in this data flow graph are captured using functional modeling. This means that all implementation related details are excluded in the model. We have used IMEM to model the experimental RTVPS presented in this paper.

The pre-processing parts of an RTVPS are usually neighborhood oriented. Examples of 2-D neighborhood operations are convolution, histogram, spatial and gray-level transforms, erosion, dilation, and component labeling [10]. Spatio-temporal RTVPS will operate on a 3-D neighborhood, which will also increase data storage and transfer intensity. Examples of 3-D operations are optical flow calculations and scene change detection [11]. Fig. 2 depicts an example of a 3-by-3-pixel spatial neighborhood. The neighborhood slides over the video operation's input frame in a progressive scan order [11]. One output pixel is calculated at every neighborhood position. Consequently, the data flow dependencies are regular, meaning that they are the same for every pixel position *A* except for the frame boundaries, exemplified by positions *B* and *C*. Data dependency vectors are drawn within the neighborhood for all three positions.

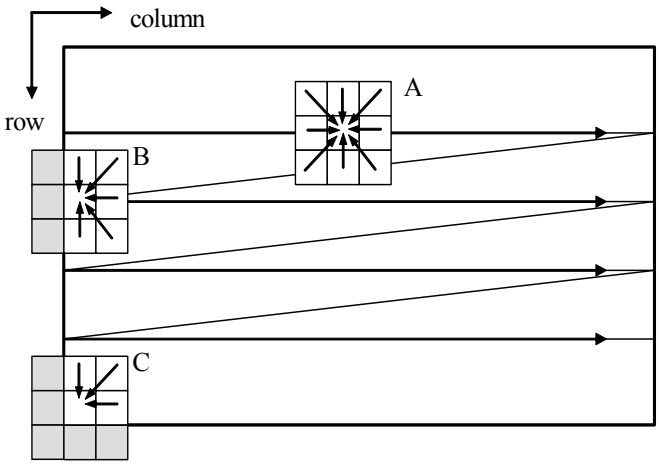


Fig. 2. A 2-D pixel neighborhood slides over the input frame.

The High abstraction-Level Synthesis (HLS) of neighborhood oriented RTVPS that we present in this paper exploits the simplification from that only regular data flow dependencies need to be considered. Regular dependencies are thus a necessary condition for this work. However, for a general HLS tool, regularity of data accesses can be improved by the linear part of affine source code transformations [25].

HLS will require that a CAD-tool can automatically calculate the length of intermediate buffers and also optimize their placement in a memory hierarchy, such that the total memory storage requirement is minimized. As shown above, the optimization of placement and sizes of intermediate buffers is possible when bit-widths are considered. This optimization is a crucial task that goes beyond what is traditionally meant by HLS. This is because our approach also includes background memory management. Optimization of the background memory was not previously considered in HLS and was not the target of HLS before.

Memory requirement minimization has previously motivated us to interface the IMEM model with the memory storage estimation tool STOREQ [12], [13]. The input to STOREQ is a polyhedral model of a DSP system. The output of STOREQ is the optimal loop nest ordering and an estimation of the required memory storage for individual data dependencies, assuming a single homogeneous bit-width. A more accurate estimation of the memory resources requires the correct bit-width information to be included for each dependency.

Bit-widths can also be considered at register retiming, a technique known to reallocate registers or memory, in order to either find the fastest clock speed of a synchronous circuit or

minimize the amount of used registers or memory storage [14], [15]. Further review of related work is outlined in Section II.

B. Contributions

Utilizing the regularity of data accesses in RTVPS and the coarse granularity of the IMEM model, we present in this paper a formal model that captures the optimization problem described above. The coarse granularity of our IMEM model enables our technique to calculate the intermediate storage, optimize its placement and also accurately estimate the total memory storage requirement within fractions of a second. The novel scientific contributions of this paper are:

- We show how SystemC modeling can be combined with polyhedral program modeling for data dependency analysis.
- We define an optimization model that minimizes storage size of all layers in a memory hierarchy for RTVPS.
- To our knowledge, retiming has not earlier been incorporated in a general polyhedral model. We show in this paper, for RTVPS, how retiming is applied as a polyhedral operation together with loop fusion and loop shifting.

The remainder of this paper is structured as follows. The next section reviews related work. Section III reviews our previous research in the area of modeling and HLS of RTVPS. Section IV describes the polyhedral modeling that the INLP is based on. Section V explains how data reuse is exploited in a perfect memory hierarchy at three levels. Section VI explains the INLP optimization model. Section VII describes and discusses an active video based surveillance camera and the results achieved from optimization of its memory hierarchy. Results from a synthetic RTVPS are presented and discussed in Section VIII. The model is discussed in Section IX. Finally, some conclusions are drawn in Section X.

II. Related work

Woodruff and Stonebraker present a comparison of different heuristics that can be used to minimize the average response time for a DFG implementation [17]. This comparison was made for randomly generated data flow graphs. Data types or bit-widths were not considered in this work.

Williamson and Lee present a framework for the implementation of SDFGs on hardware [18]. The output system is modeled in the hardware description language VHDL. A fixed scheduler that defines the execution order without timing is developed. Latencies of the operations are discussed for the future but are not included in this work. Bit-widths or the size of the buffers for storage of intermediate data are not discussed in this paper.

Adé, Lauwereins and Peperstraete have developed a theory for determining the minimal sizes of buffers in a SDFG that still guarantee a deadlock-free static schedule [19]. This work cannot be applied directly to the parallel implementation architecture that we target. Only consistency of a DFG is considered. Latency of operations and bit-width of data tokens must be incorporated to make this theory applicable to our work.

Leiserson et al. [15] use a graph theoretic framework for transforming synchronous circuitry. The minimum clock period is altered through a relocation of registers. This transformation aims to find the lowest possible clock period. An alternative cost function that minimizes the total number of registers is also given. The circuit optimization can then be transformed into a minimum cost flow problem. A key observation on this work is that it requires the order of computation to be defined prior to optimization. This is a major difference compared to our work where multidimensional data flow dependencies are modeled in a polyhedral space without defining the order of computation.

Passos and Sha [16] present a heuristic technique for retiming of multidimensional DFGs. This retiming can be considered as a multidimensional pipelining of loop bodies with the goal of achieving full parallelism. This can be achieved if all output edges on the DFG has a nonzero delay. The work is restricted to homogeneous constant dependencies.

ATOMIUM is an environment that supports the Data Transfer and Storage Exploration (DTSE) methodology developed at IMEC in Belgium [1], [2]. ATOMIUM can take a DSP algorithm all the way from a high abstraction-level specification to customized hardware and processor level implementations. The main focus throughout the entire design process is towards memory- usage and transactions. Bit-widths are considered as input for a data flow transformation called delayline shifting [14]. This work provides a limited formalism but ATOMIUM does not automate delay line shifting. ATOMIUM is reducing the memory storage size using global loop transformations without considering bit-widths. However, the subsequent optimization steps do consider bit-widths. In contrast, the work presented in this paper reduces storage size using loop- fusion and shifting in combination with a bit-width constrained retiming of the SDFG.

Source code transformations for the compiler optimization of programs are often described using a polyhedral model of the program. This geometrical modeling of loop nests by polyhedrons is a compact and a well-adopted mathematical program description [20], [21]. It has its origin in systolic design, where it is used for the analysis and improvement of parallelism in programs compiled for multiprocessor architectures [22], [23].

Equally, polyhedral modeling can be used to describe source code transformations for the improvement of data locality in multidimensional DSP-systems. Many different research groups have published automatic loop transformations for the improvement of data locality. These transformations improve data locality in single loop nests [24], or better globally over many loop nests [25].

Until now, nobody tackled the combination SystemC modeling, polyhedral modeling, memory storage estimation and optimization, re-timing and bit-widths.

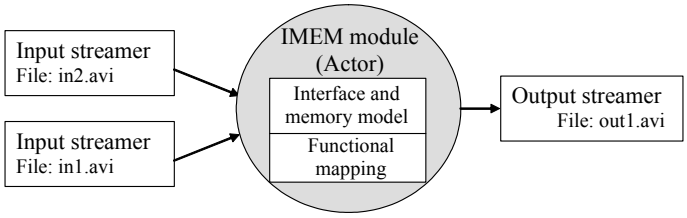


Fig. 3. Data Flow Graph in IMEM.

III. Modeling and Synthesis with IMEM

This section is dedicated to the introduction and overview of our previous research activities in the area of modeling and synthesis of RTVPS.

A. Modeling

We model neighborhood oriented RTVPS using an extension library to SystemC that we call IMEM [7],[8],[9]. Fig. 3 shows one IMEM module, (actor), two input- and one output video streamers linked together in a coarse-grained SDFG. The streamers are test-benches that provide and capture simulation data to and from the model. The behavior of an actor in this IMEM SDFG is defined by a conceptual interface and memory model in combination with a functional description of the filter kernel. Fig. 5A depicts a 3-dimensional collection of pixels, a neighborhood that IMEM uses as an abstraction. This abstraction expresses perfectly regular data accesses, which is true for neighborhood-oriented image processing operations [7], with the exception of frame boundaries. We have chosen not to model boundary

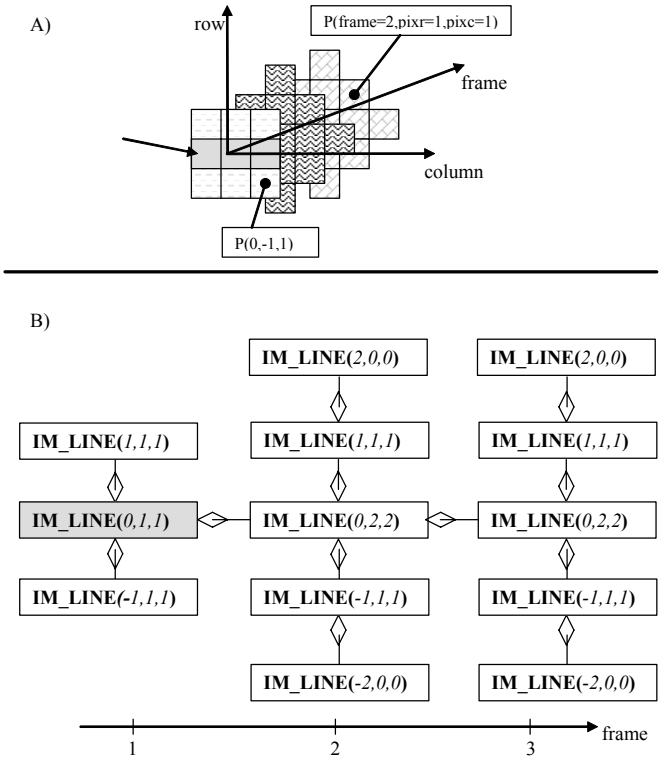


Fig. 5. A 3-dimensional collection of pixels.

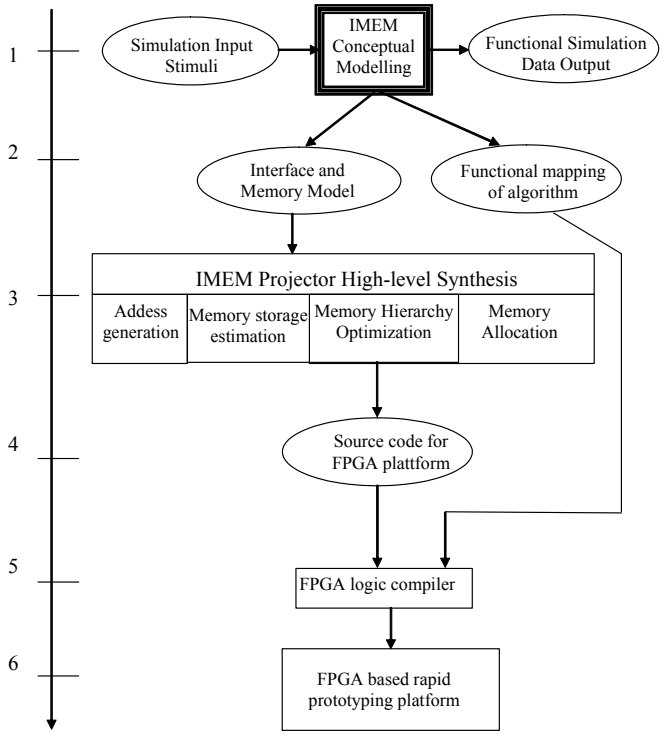


Fig. 4. The IMEM system development workflow.

conditions in IMEM because their impact on the size of the final memory hierarchy is assumed to be negligible. Fig. 2 exemplifies two different border conditions labeled *B* and *C* where the values of the outer pixel positions of the neighborhood must be estimated based on the interior pixels. This estimation is typically based on linear interpolation techniques. Fig. 2 shows how the sets of data flow dependency vectors for the border positions *B* and *C* are subsets of the vectors for position *A*. Thus, control- and data flow of the computation will be affected by border conditions, but not the memory hierarchy. The chosen interpolation technique must be additionally specified and fully taken into account at synthesis [26].

Fig. 5B shows an example of how the pixel neighborhood in Fig. 5A is modeled in IMEM by a collection of design entities. These entities are depicted as an object deployment diagram using a UML-style notation. This diagram is only shown as a graphical illustration of object instantiations made in the C++-code.

IM_LINE(0,1,1) in Fig. 5B, corresponds to a line with relative row address *0* and with one pixel to the left and one pixel to the right. An arrow in Fig. 5A indicates this line. The first group of entities, indicated by the horizontal axis in Fig. 5B corresponds to the first frame in the neighborhood in Fig. 5A. Two examples of how individual pixels can be relatively addressed within the neighborhood are shown in Fig. 5A. Video stream interfaces are modeled using a similar technique.

B. Synthesis

IMEM Projector is a tool that currently can import an IMEM model and perform an early memory storage

A)

```
for(R=0; R<L_R; R++)
 for(C=0; C<L_C; C++)
 {
   if( R>=O_AR && R<O_AR+N_R && C>=O_AC && C<O_AC+N_C )
     A[R,C] = input();
   if( R>=O_BR && C>=O_BC && R<O_BR+N_R && C<O_BC+N_C)
     B[R,C] = Operator( A[R-O_BR+O_AR,C-O_BC+O_AC],
                        A[R-O_BR+O_AR,C-O_BC+O_AC-1],
                        A[R-O_BR+O_AR,C-O_BC+O_AC+1],
                        A[R-O_BR+O_AR-1,C-O_BC+O_AC],
                        A[R-O_BR+O_AR-1,C-O_BC+O_AC-1],
                        A[R-O_BR+O_AR-1,C-O_BC+O_AC+1],
                        A[R-O_BR+O_AR+1,C-O_BC+O_AC],
                        A[R-O_BR+O_AR+1,C-O_BC+O_AC-1],
                        A[R-O_BR+O_AR+1,C-O_BC+O_AC+1] )
 }
```

B)

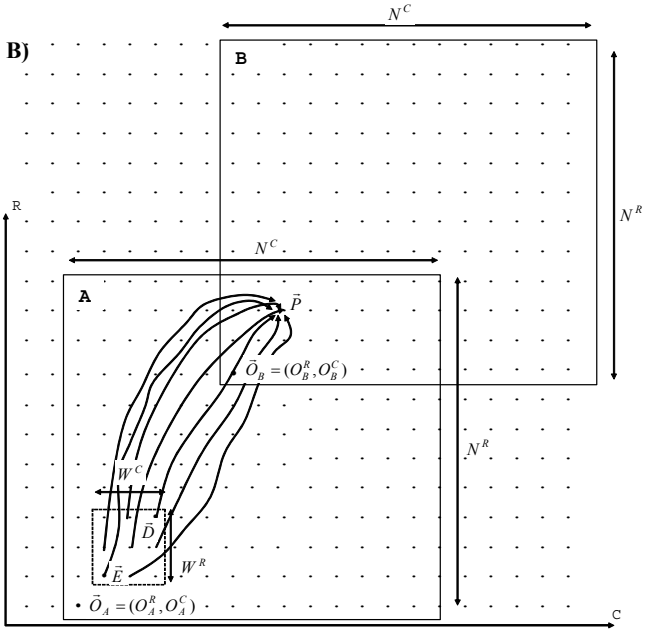


Fig. 6. The polyhedral iteration space.

estimation by interfacing to the STOREQ tool [12],[13]. An optimization model that captures the allocation of RTVPS FIFO buffers to dual ported FPGA block RAMs will in future be refined into a heuristic and included into the synthesis tool [27]. Two different approaches for generating the necessary addresses for the implementation of a set of RTVPS FIFO buffers are compared [28]. The IMEM prototyping workflow depicted in Fig. 4 demonstrates how our research on modeling and high level synthesis fits into an implementation trajectory. This workflow is defined at six different levels along the left-hand axis. The video-processing algorithm is developed and simulated using IMEM at level 1. This executable model can then be verified through functional simulation. Data dependency information, frame sizes, composition of the 3-dimensional neighborhoods and color space models are exported into an interface and memory model at level 2. This text file is the input to IMEM Projector and a synthesis process at level 3. This is where memory estimation, hierarchy optimization, allocation and address generation fits into the trajectory. The synthesis of an interface and memory model can be made more efficient by using the results from the memory hierarchy optimization presented in this paper.

The functional mapping of an algorithm can interface directly with the optimized interface and memory model and be compiled at level 5. Level 6 corresponds to an FPGA-based prototyping platform.

The prototype synthesis tool IMEM Projector is developed in Microsoft VC++ using the MFC library. This programming environment provides us with a quick path to a friendly graphical user interface. IMEM Projector is currently able to import a data flow dependency model from the SystemC based modeling library IMEM [7]. The synthesis tool can generate the necessary Matlab input files for the memory estimation tool STOREQ and also execute a compiled version of STOREQ [12]. The memory synthesis approach presented in this paper is only implemented as a script of equations that can be executed in the optimization tool Lingo [34]. The plan for future work on IMEM Projector is to fully integrate the presented memory synthesis by incorporating source code for a commercially or freely available ILP solver. We have also published work on subsequent synthesis steps such as FPGA memory allocation [27] and address generation for FPGA memories [28]. These two steps are also single scripts or programs that will be integrated into IMEM Projector.

Our research on this application specific CAD-tool aims to provide a non- hardware skilled engineer with a quick path to the computing power of FPGAs. We believe that an FPGA can be an attractive platform for implementing RTVPS. FPGAs provide a combination of programming flexibility and large amount of both logic and memory resources. The synthesis technique presented in this paper is therefore targeting FPGAs and the neighborhood oriented and thus data intensive parts of an RTVPS.

IV. Polyhedral Data Dependency Analysis

This Section will introduce sufficient details regarding the polyhedral model for our discussions later in the paper. This polyhedral model is used to capture RTVPS, operating on three-dimensional video data. We also develop a number of equations, later used to define the INLP we use for our optimizations.

A. Modeling in the common polyhedral iteration space

Digital signal processing algorithms are typically described by a set of non-perfectly nested loop nests. Fig. 6A depicts a C-coded loop-nest where a video processing operation called *Operator* is called within the loop body. The code in Fig. 6A traverses the *C* dimension first and secondly the *R* dimension. $R \in Z^+$ and $C \in Z^+$ are loop variables. It is also legal to change the loop nest levels so that firstly dimension *R* and then *C* are traversed. Fig. 6B shows the corresponding iteration space regardless of the order of traversal.

Definition 1 An iteration in a loop corresponds to an *iteration node* in the *n*-dimensional common polyhedral iteration space $S_{cis} = Z^n$ and is identified by its *loop iterator*

vector $\vec{I} \in S_{cis}$ [30].

$$\vec{I} = (I_1, I_2, \cdots, I_n) \quad (1)$$

I_j is the value of the *j*:th iterator. I_n is the innermost iterator.

The enclosed set of iteration nodes labeled *B* in Fig. 6B, are the nodes where output pixels are written to array *B* and pixels are read from array *A*. Set *A* corresponds to the iteration nodes where input pixel data is written to array *A*. These sets are integral polytopes of equal size, further referred to as just polytopes.

Definition 2: A group of iteration nodes in the common polyhedral iteration space, bounded by a set of constraints is defined as an *integral polytope P* and is formally denoted as a set [21].

$$P = \begin{Bmatrix} \vec{I} \mid I_{1L} \le I_1 \le I_{1H} \wedge I_{2L} \le I_2 \le I_{2H} \wedge \cdots \\ \wedge I_{nL} \le I_n \le I_{nH} \wedge \vec{I} \in S_{cis} \end{Bmatrix} \quad (2)$$

$\vec{I}_L = (I_{1L}, I_{2L}, \cdots, I_{nL}) \in S_{cis}$ are the lower bounds for each iterator and $\vec{I}_H = (I_{1H}, I_{2H}, \cdots, I_{nH}) \in S_{cis}$ are the upper bounds.

Polytopes, consisting of iteration nodes, where the statements are executed, represents each statement in each loop nest. Global loop transformations mean mapping all polytopes, representing execution of all statements in a program, to a *common polyhedral iteration space*. This operation is in the sequel referred to as polytope placement. During placement the polytopes are transformed to guarantee legacy optimized for locality and regularity of the program code's data flow dependencies [25]. The order in which the dimensions are traversed is not fixed in the common polyhedral iteration space. This freedom is used in the STOREQ methodology, which determines the best ordering with relation to storage size requirements [29].

B. Iteration space traversal

The enclosed smaller set of six iteration nodes within polytope *A* in Fig. 6B corresponds to the neighborhood of iteration nodes where pixel data is read by *Operator* and one output pixel is computed and written to array *B* at iteration node $\vec{P}$.

Definition 3: A set of iteration nodes N_A where data is written to array *A* and that surrounds a central iteration node $\vec{C}$ is called a *neighborhood of iteration nodes* and we define it as an integral polytope.

$$N_A = \begin{Bmatrix} \vec{I} \mid C_1 - \frac{W_1 - 1}{2} \le I_1 \le C_1 + \frac{W_1 - 1}{2} \wedge \\ C_2 - \frac{W_2 - 1}{2} \le I_2 \le C_2 + \frac{W_2 - 1}{2} \wedge \cdots \\ \wedge C_n - \frac{W_n - 1}{2} \le I_n \le C_n + \frac{W_n - 1}{2} \wedge \vec{I} \in S_{cis} \end{Bmatrix} \quad (3)$$

$$\vec{W} = \{W_1, W_2, ..., W_n\} \in ODD^{+n} \wedge ODD^{+} = \{x \mid x = 2 \cdot y + 1 \wedge y \in Z^{+}\}$$

is a tuple of variables that defines how many iteration nodes the neighborhood spans in each dimension of the common iteration space S_{cis}.

The data dependency distance vectors in Fig. 6B, drawn from all iteration nodes within the neighborhood of array *A* to the output node $\vec{P}$ in array *B* corresponds to the data flow dependencies. These dependencies are regular and thus will be the same for all nodes within polytope B, except for all the frame borders.

Definition 4: A *data dependency distance vector* $\vec{V}_{wrA}$ is defined as going from the iteration node of data write, $\vec{I}_w$ to the iteration node of data read, $\vec{I}_r$. The data dependency distance vector $\vec{V}_{wrA}$ is thus constrained by the time period of the write and read of the same memory location within array *A* [30],

$$\vec{V}_{wrA} = \vec{I}_r - \vec{I}_w \text{ , if } \vec{I}_R \text{ depends on } \vec{I}_W \text{ through array A} \quad (4)$$

Iteration node $\vec{O}_A$ in Fig. 6B is the origin (lower left corner) of polytope A and $\vec{O}_B$ the origin of polytope B. Node $\vec{D}$ is the upper right corner of the neighborhood and $\vec{E}$ its lower left corner. The number of iteration nodes traversed within polytope X before node $\vec{P} = (P^R, P^C)$ is denoted $I_X(\vec{P})$, where polytope X can be a dependency's source polytope A or its destination polytope B, $X \in \{A, B\}$.

$$I_X^{RC}(\vec{P}) = N^C \cdot (P^R - O_X^R) + P^C - O_X^C \quad (5)$$

Equation (5) holds true if the *C*-dimension is traversed firstly and secondly the *R*-dimension. For the opposite traversal,

$$I_X^{CR}(\vec{P}) = N^R \cdot (P^C - O_X^C) + P^R - O_X^R \quad (6)$$

At each iteration node within polytope *A*, data is written to array *A*. At each iteration node within polytope *B*, a neighborhood of data items are read from data array *A* and one calculated data item is written to array *B*. The neighborhood of data items read from array *A* are previously written to array *A* at an iteration node within polytope *A*. (They must all be previously written in order to make all data flow dependencies legal.) Regardless of the selected order of traversal, $\vec{E}$ will always be the first node where pixel data is written to array *A* within the neighborhood and also later read from array *A* at iteration node $\vec{P}$. Equally, node $\vec{D}$ will always be the last node within the neighborhood where pixel data is written to array *A* and also later read from array *A* at iteration node $\vec{P}$.

C. Memory storage

The total memory storage S_{AB}^{CR} required by the data flow dependencies from all iteration nodes within the neighborhood of polytope *A* to iteration node $\vec{P}$ within *B* can be calculated as the number of traversed iteration nodes within polytope *A* at node $\vec{P}$ minus the traversed iteration nodes within polytope *A* at node $\vec{E}$ and multiplied with the bit-width. This is because all

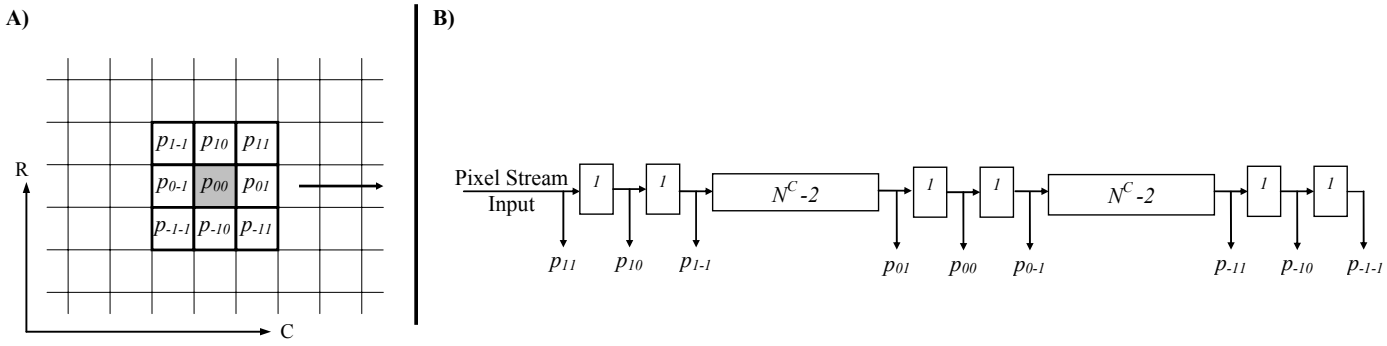


Fig. 7. Minimal neighborhood implementation.

nodes traversed before node $\vec{E}$ will never be used as input again for the calculation of output pixels. However, all nodes traversed from node $\vec{E}$ to $\vec{P}$ will definitely be used again and therefore the pixel data written to array *A* at all those nodes between $\vec{E}$ and $\vec{P}$ must be kept in memory. From Fig. 6 we see that iteration node $\vec{E}$ can be calculated as,

$$\vec{E} = \vec{P} - \vec{O}_B + \vec{O}_A - \left(\frac{W^R - 1}{2}, \frac{W^C - 1}{2}\right) \tag{7}$$

The total memory storage size of a dependency then becomes,

$$S_{AB}^{RC} = BTW_A \cdot \left(I_A^{RC}\left(\vec{P}\right) - I_A^{RC}\left(\vec{E}\right)\right) \tag{8}$$
$$= BTW_A \cdot \left(N^C \cdot \left(O_B^R - O_A^R + \frac{W^R - 1}{2}\right) + O_B^C - O_A^C + \frac{W^C - 1}{2}\right)$$

Equation (8) applies for a first-C-then-R traversal of the iteration space. $BTW_A \in Z^+$ is the bit-width of each data element in array A. The memory storage S_{AB}^{CR} for a first-R-then-C traversal and can equally be developed.

D. Data dependency legacy constraint

Obviously, the memory storage according to Equation (8) is reduced to zero if the distance vector between the two polytope origins are set to $\vec{O}_B - \vec{O}_A = \left(-\left(W^R - 1\right)/2, -\left(W^C - 1\right)/2\right)$. But the question that must be answered is how close to each other can the two polytopes be placed without violating the data flow dependencies? The answer is that the pixel data at node $\vec{D}$ within the neighborhood must be written to array *A* before the pixel data at node $\vec{P}$ can be computed and written to array *B*. Or equally we can say that the number of traversed nodes within polytope *A* at node $\vec{P}$ must be greater than or equal to the number of traversed nodes at $\vec{D}$. We see from Fig. 6B that iteration node *D* is equal to,

$$\vec{D} = \vec{P} - \vec{O}_B + \vec{O}_A + \left(\frac{W^R - 1}{2}, \frac{W^C - 1}{2}\right). \tag{9}$$

The data dependency legacy constraint then becomes,

$$I_A^{RC}\left(\vec{P}\right) \geq I_A^{RC}\left(\vec{D}\right) \tag{10}$$
$$N^C \cdot \left(O_B^R - O_A^R\right) + O_B^C - O_A^C \geq N^C \cdot \frac{W^R - 1}{2} + \frac{W^C - 1}{2}$$

Equation (10) expresses that the system must be causal. This means that all pixels within the neighborhood must be written to memory before the video processing operator calculates the output pixel. This constraint can equally be developed for a first row-then-column traversal.

E. Minimum memory storage

If the result of Equation (10) is substituted into Equation (8), we get the storage size of the neighborhood as the minimum memory storage $\min\left(S_{AB}\right)$,

$$NSZ_{AB}^{RC} = \min\left(S_{AB}^{RC}\right) = BTW_A \cdot \left(N^C \cdot \left(W^R - 1\right) + W^C - 1\right) \tag{11}$$

Equation (11) expresses the lower limit of the storage size of a dependency S_{AB}. This limit is set by the data dependency legacy constraint in equation (10). The correctness of this equation can be justified by studying the neighborhood implementation depicted in Fig. 7. Fig. 7B shows a delay network of FIFO-registers that implements the neighborhood modeled in Fig. 7A and satisfies Equation (11) for a first-*C*-then-*R* traversal. The labeled pixels within the neighborhood in Fig. 7A can be identified as taps in the delay network in Fig. 7B. Each FIFO is labeled with its number of elements.

The neighborhood storage size for a first-*R*-then-*C* traversal can equally be developed.

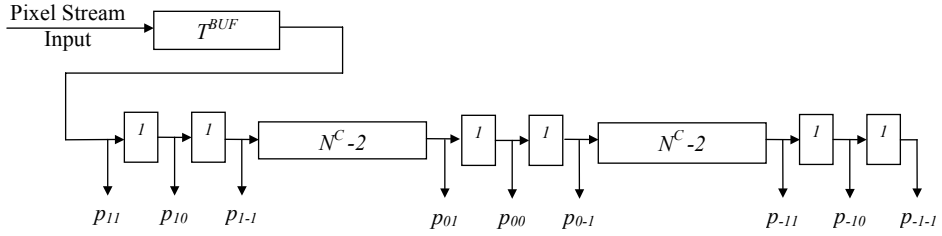


Fig. 8. Neighborhood implementation with additional buffer.

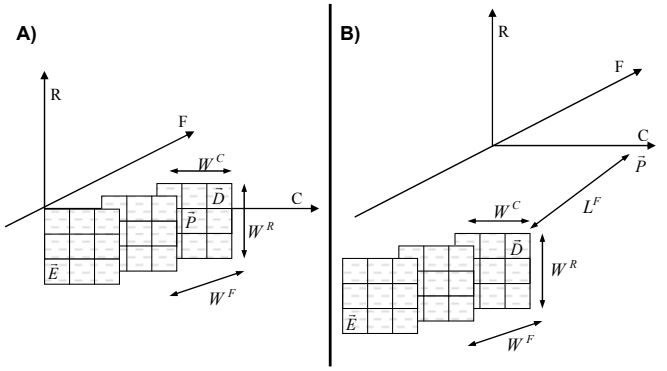


Fig. 9. Spatio-temporal neighborhood.

F. Size of intermediate buffer

Any memory storage calculated by Equation (8) greater than the neighborhood storage calculated by Equation (11) results in an additional intermediate delay buffer on the neighborhood input. The storage size of this buffer for a first-C-then-R traversal can then consequently be calculated as,

$$BUF_{AB}^{RC} = S_{AB}^{RC} - NSZ_{AB}^{RC} \tag{12}$$
$$= BTW_A \cdot \left(N^C \cdot \left(O_B^R - O_A^R - \frac{W^R - 1}{2} \right) - \frac{W^C - 1}{2} + O_B^C - O_A^C \right)$$

The data dependency legacy constraint in equation (10) allows the total memory storage S_{AB} of a dependency to be greater but not less than then the size of the neighborhood. Thus, equation (12) expresses memory storage additional to the memory required by the neighborhood implementation.

Fig. 8 depicts a neighborhood as modeled in Fig. 6 but with an intermediate buffer on its input. The buffer is labeled with its length $T^{BUF} \in Z^+$, given in number of elements. The buffer size given in number of bits can be calculated according to Equation (12). The buffer size for a first-*R*-then-*C* traversal can equally be developed.

G. Spatio-temporal neighborhoods

A neighborhood-oriented video processing operation sometimes has a temporal behavior where pixels from several consecutive frames are inputs as output pixels are computed. Examples of such operations are optical flow calculations or spatio-temporal filters.

Fig. 9A depicts an example of a spatio-temporal neighborhood where the number of frames, rows and columns are set to $W^F=W^R=W^C=3$. The number of frames $W^F=3$ means that the neighborhood contains the current frame plus two older frames. Thus, $W^F = 1$ means that only the current frame is accessed. A parameter L^F is used to explicitly model an additional frame delay as shown in Fig. 9B.

Equation (5) and (6) defining the number of traversed iteration nodes will now be expanded with the frame dimension.

$$I_X^{FRC}\left(\vec{P}\right) = N^C N^R \cdot (P^F - O_X^F) + N^C \cdot \left(P^R - O_X^R\right) + P^C - O_X^C \tag{13}$$

$$I_X^{FCR}\left(\vec{P}\right) = N^C N^R \cdot (P^F - O_X^F) + N^R \cdot \left(P^C - O_X^C\right) + P^R - O_X^R \tag{14}$$

The memory storage S_{AB} required by a spatio-temporal dependency is given by expanding Equation (8) with the frame dimension. This expansion is developed based on that node D in the 2-dimensional iteration space is positioned at L^F in the frame dimension. See Fig. 6B and equation (9), Node D then becomes,

$$\vec{D} = \vec{P} - \vec{O}_B + \vec{O}_A + \left(L^F, \frac{W^R - 1}{2}, \frac{W^C - 1}{2} \right) \tag{15}$$

Node *E* in the 2-dimensional iterations space, see Equation (7), is positioned at L^F-W^F+*1* in the frame dimension. Node *E* then becomes,

$$\vec{E} = \vec{P} - \vec{O}_B + \vec{O}_A - \left(W^F - 1 - L^F, \frac{W^R - 1}{2}, \frac{W^C - 1}{2} \right) \tag{16}$$

Node *D*, *E* and *P* are indicated in Fig. 9. With a neighborhood that spans W^F frames, positioned L^F frames earlier than the output frame, the memory storage requirement S_{AB} for a spatio-temporal dependency becomes,

$$S_{AB}^{FRC} = BTW_A \cdot \left(I_A^{FRC}\left(\vec{P}\right) - I_A^{FRC}\left(\vec{E}\right) \right) \tag{17}$$

A complete definition of the dependency storage size S_{AB}, the storage size of the neighborhood NSZ_{AB} and the size of the intermediate buffer BUF_{AB}, where the temporal dimension is incorporated, can be found in the Appendix.

H. Pipelined computational logic

At each clock cycle, all neighborhood pixels are used as inputs to calculate a corresponding output pixel. The throughput of the system is thus one pixel per clock cycle. The design of the computational logic is therefore most likely divided into several pipeline stages in order to cope with this real-time constraint.

Fig. 10 shows an example of a pixel neighborhood with a three-stage pipeline on the computational logic. There is one register after each computational step. Hence, the output pixel is then additionally delayed with the number of clock cycles, $T^{PL} \in Z^+$, equal to the number of pipeline stages. This delay can be added to the polyhedral dependency analysis by rewriting the dependency legacy constraint as previously defined by Equation (10). The number of traversed iteration nodes at node $\vec{P}$ must now be increased equal to the number of pipeline stages.

$$I_A^{FRC}\left(\vec{P}\right) \geq I_A^{FRC}\left(\vec{D}\right) + REC \cdot T_B^{PL} \tag{18}$$

$REC_{AB} \in \{0,1\}$ is set to *0* if the dependency is recursive, otherwise it is set to *1*. This means that for a non-recursive dependency, the number of traversed nodes at node $\vec{P}$ must be equal or larger than the number of traversed nodes at $\vec{D}$ plus the modeled pipeline stages.

Definition 5: At the destination node $\vec{I}_r$ of a data dependency distance vector $\vec{V}_{wrA}$, the data flow dependent statement performs computation. If and only if the result of that computation is written to the same data array *A* as from where data is also read and used as input for the computation, we define the vector from node $\vec{I}_w$ to node $\vec{I}_r$ as a *recursive*

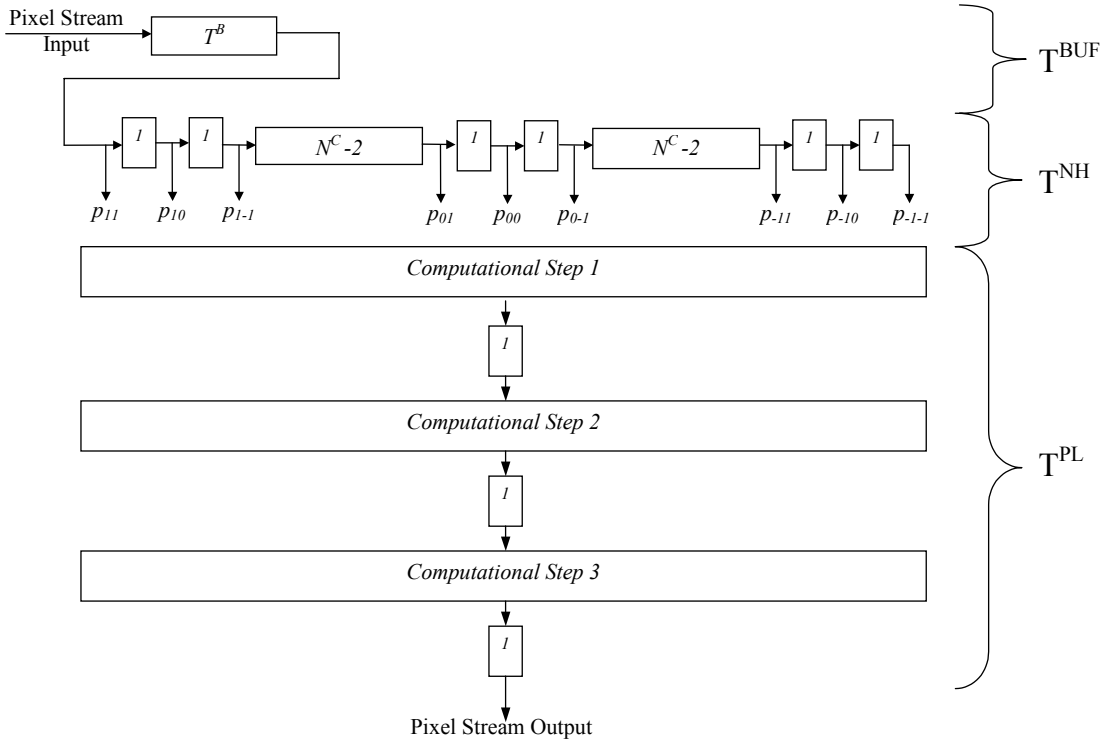


Fig. 10. Neighborhood implementation with pipelined computational logic.

data dependency distance vector. This means that elements that are computed and written into array *A* is data flow dependent on a neighborhood of elements that are previously written to the same array *A*.

Definition 6: We define a *dependency* D_A within the scope of neighborhood oriented RTVPS as the set of dependency distance vectors stemming from a neighborhood of iteration nodes N_W, where pixel data is written to array *A,* to a destination node $\vec{I}_R$, where pixel data is read from array *A*.

$$D_A = \left\{ \begin{matrix} \vec{V}_{wrA} \mid \exists \vec{I}_R \in S_{cis} \wedge \forall \vec{I}_W \in N_A \; s.t. \vec{V}_{wrA} = \vec{I}_R - \vec{I}_W \\ \wedge \vec{I}_R \; depends \; on \; \vec{I}_W \; through \; array \; A \end{matrix} \right\} \quad (19)$$

If the dependency distance vectors belonging to D_A are recursive, we say that dependency D_A is a *recursive dependency*. As a consequence of the selected SDFG programming model, we assume that all elements in a given data array can only be produced by one single SDFG node output.

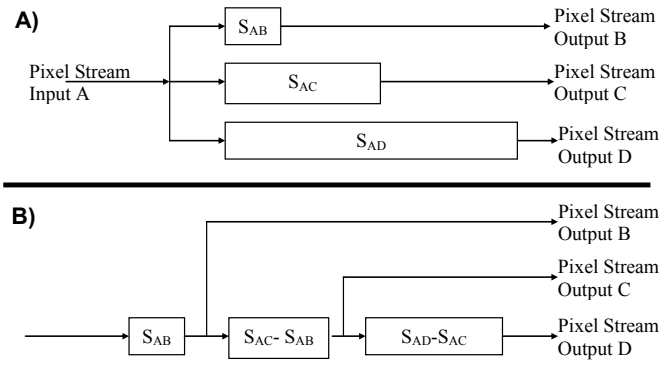


Fig. 11. Sharing of data storage between three buffers.

For a recursive dependency, the source and destination polytopes are thus the same, and the pipeline equally delays both output and input pixels. Hence, the number of traversed iteration nodes between node $\vec{D}$ and node $\vec{P}$ are not changed by the introduction of a pipeline. The intermediate buffer must be equally reduced with the number of elements corresponding to the pipeline stages.

$$BUF_{AB}^{FRC} = BTW_A \cdot \begin{pmatrix} N^C \cdot N^R \cdot \left(O_B^F - O_A^F - L^F \right) + \\ N^C \cdot \left(O_B^R - O_A^R - \frac{W^R - 1}{2} \right) + \\ O_B^C - O_A^C - \frac{W^C - 1}{2} - T_B^{PL} \end{pmatrix} \quad (20)$$

If a dependency is recursive, the parameter L^F must be less or equal to minus one. Otherwise, the dependency will always be non causal and thus illegal. The storage size NSZ, required by the neighborhood implementation will remain unchanged due to pipelining. This, because the minimum size required by the data dependencies of a neighborhood will not change because of an additional output delay caused by pipelined logic. However, the additional data storage introduced by the pipelined computational logic can now be estimated to,

$$PS_B = T_B^{PL} \cdot BTW_B \quad (21)$$

I. Sharing data storage between dependencies

Let us assume that three video streams *B, C* and *D* are data flow dependent on the same source stream *A* and that the sizes of these dependencies S_{AB}, S_{AC} and S_{AD} are calculated in the polyhedral space as defined in Section IV-C. $S_{AB} < S_{AC} < S_{AD}$.

Then a direct implementation of the corresponding buffers will be as depicted in Fig. 11A, where the rectangular nodes correspond to the storage requirements of the dependencies and the edges are video streams. However, for the RTVPS that we are modeling, we can assume that all data items produced at the iteration nodes of the source polytope are also read by all dependent destination polytopes. This means that the *A* data required to calculate *B* is a subset of the *A* data required to calculate *C*, and likewise between *D* and *C*. Thus, sharing of buffers are possible. This sharing is exploited in Fig. 11B so that the total storage size for the three dependencies equals $\max(S_{AB}, S_{AC}, S_{AD}) = S_{AD}$.

J. Latency of a dependency

Definition 7 We define the latency of a dependency $T_{AB}^{DEP} \in Z^+$, stemming from polytope *A* and going to polytope *B,* as the number of clock cycles between the writing of the pixel with coordinate *(f,r,c)* to array *A* and the writing of the computed output pixel with coordinate *(f,r,c)* to array *B*.

T_{AB}^{DEP} is the sum of latencies caused by the intermediate buffer, $T_{AB}^{BUF} \in Z^+$, and by the image processing operation $T_{AB}^{OP} \in Z^+$.

$$T_{AB}^{DEP} = T_{AB}^{OP} + T_{AB}^{BUF} \tag{22}$$

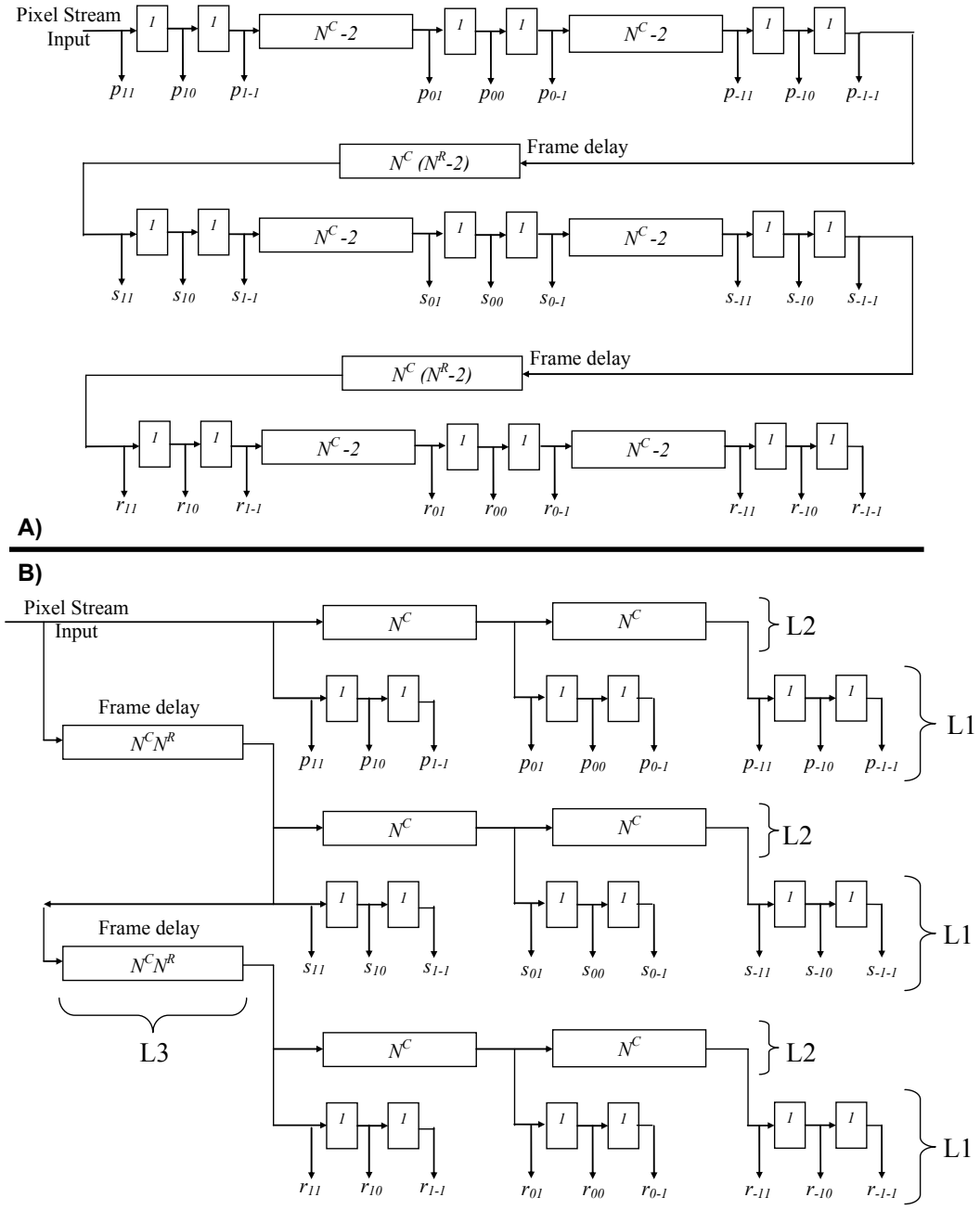


Fig. 12. A) Neighborhood implementation without memory hierarchy.
B) Neighborhood implementation with a 3-level custom memory hierarchy.

The pipelining of computational logic $T_B^{PL} \in Z^+$ and the neighborhood delay $T_{AB}^{NH} \in Z^+$, in turn cause the latency of the image processing operation.

$$T_{AB}^{OP} = T_{AB}^{NH} + T_B^{PL} \tag{23}$$

Fig. 10 gives an example of how these latencies can appear in the RTVPS circuitry. T_{AB}^{NH} can be calculated in the polyhedral model as the number of traversed iteration nodes in polytope A at node $\vec{D}$, see Fig. 9 and Equation (15), minus the number of traversed iteration nodes in polytope A at the center of the neighborhood, $\vec{P} - \vec{O}_B + \vec{O}_A + (L^F, 0, 0)$.

$$T_{AB}^{FRC,NH} = N^C \cdot \left(\frac{W^R - 1}{2} \right) + \frac{W^C - 1}{2} \tag{24}$$

Equation (24) expresses the number of clock cycles required for a pixel value to pass the FIFO-buffers from the input tap of the neighborhood to the center tap. This is equal to the number of clock cycles the output frame of a video processing operator is delayed by the neighborhood with reference to the input frame.

T_{AB}^{BUF} can be calculated from the size of the intermediate buffer as defined by Equation (20) by simply dividing the buffer size with its bit-width.

$$T_{AB}^{FRC,BUF} = \frac{BUF_{AB}^{FRC}}{BTW_A} \tag{25}$$

Equation (25) expresses the length of the intermediate buffer, or equally its latency given in number of clock cycles.

V. Data Reuse Exploitation

In DSP applications such as RTVPS, there is a high level of data reuse. This means that a data item, previously computed and written to memory, is later read, not once, but twice, or even more, until it is finally consumed at the last read operation. The normal design procedure is then to take a local copy of the data item to a smaller memory or register at the first time it is read. At the successive read operations, the data item is read from the smaller memory instead of from the larger memory. Speed or power performance can then be gained [31]. If in addition, the DSP algorithm is manifest and thus the control flow is known at compile time, it is possible to design a custom memory hierarchy [31]. A custom memory hierarchy interleaves all data fetches with the computation based on a compile time analyzis of memory transactions. Since neighborhood oriented RTVPS are manifest, we have chosen this attractive solution. In comparison with a direct mapped or set associative cache memory, the custom memory hierarchy requires less hardware. This is mainly because the additional tag memories can be avoided.

The set of FIFO buffers depicted in Fig. 7 implements a spatial neighborhood. As an example, this spatial neighborhood can be repeated for three consecutive frames as depicted in Fig. 12A. We then get a spatio-temporal neighborhood that spans three pixels in both the frame-, row- and column dimensions. We see that two additional large frame buffers are now inserted into the series of delay lines. The groups of taps labeled *p*, *s* and *r* correspond to the pixel neighborhood of three consecutive frames. The storage size of this neighborhood implementation can be calculated according to Section IV-G. Theoretically, whole this implementation can be implemented in a large off-chip memory. However, there will be intensive data transactions with this large memory, since the written data at the pixel input stream will be read back 26 times at the different taps. The functional equivalent neighborhood implementation shown in Fig. 12B is arranged in a custom memory hierarchy with three levels. Which FIFO-buffers that belongs to which hierarchy level is indicated by L1 to L3. L1 are registers close to the data path. L2 are line buffers and L3 are frame buffers. We see that the line buffers at L2 contain pixel data that is a copy of a fraction of the pixel data within the frame buffers at L3. Equally, the pixel data in the registers at L1 is a copy of a fraction of the data within the line buffers at L2. This compile time customized memory hierarchy guarantees pixels to always be available at L1 exactly at the time of computation. This approach of copying pixel data using a fixed level assignment for neighborhood oriented image processing is well known [32]. We will use this type of 3-level memory hierarchy when the optimization technique presented in this paper is used for the memory synthesis of RTVPS.

VI. Memory Hierarchy Optimization

The memory hierarchy optimization model described in this section minimizes the total memory storage required by a Video Processing Data Flow Graph (VPDFG). Fig. 13 summarizes the input and output parameters. These parameters are described in detail in the Appendix. The optimization model is an Integer Non Linear Program (INLP) that is based upon the polyhedral model presented in Section IV and the directed graph in Fig. 14.

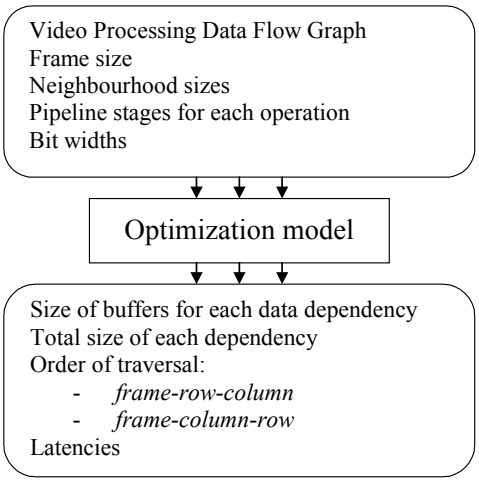


Fig. 13. Optimization model input- and output parameters.

A. Video Processing Data Flow Graph

A VPDFG, see Fig. 14, reads a number of input video streams, which then results in the computation of a number of intermediate- and output video streams. A VPDFG is a directed graph $G = (S, D)$ with a set S of input-, output-, and intermediate video streams and a set D of data flow dependencies. $SI = \{1,2,\cdots, NV \mid NV \in Z^+\}$ is an index of NV number of video streams such that $S = \{S_k \mid k \in SI\}$ and $D = \{D_{sd} \mid s \in SI \wedge d \in SI\}$. The nodes in this graph are the video streams and the edges are the data flow dependencies.

B. Objective function

Equation (26) below defines the objective function that must be minimized.

$$Minimize \sum_{Q \in SI} NS_Q \tag{26}$$

$NS_Q \in Z^+$ is the memory storage requirement associated with all data flow dependencies stemming from the video stream indexed by Q. We assume that sharing of data storage among all these dependencies is fully exploited as explained in Section IV-I. The INLP optimization is performed by placing the polytopes in a common iteration space with their relative origins $\vec{O}$ such that the total memory storage requirement is minimized.

C. Constraints

A limited selection of the optimization model's constraints, which are necessary to understand the INLP concept, is presented in this section. A complete listing of all constraints can be found in the Appendix.

$$\forall A \in SI \wedge \forall B \in SI \tag{27}$$
$$NS_A \geq NSZ_{AB} + BUF_{AB}$$

Equation (27) expresses that the node storage NS_A is the maximum of all storage requirements associated with all dependencies stemming from node A. See Section IV-E and IV-F for an explanation of *NSZ* and *BUF*. Remember that the objective function, Equation (26), minimizes the sum of all node storage NS_A.

The mechanism to handle two possible orders of execution, *frame-row-column* or *frame-column-row* is represented by two binary variables, $BIN = \{0,1\}$, $FRC \in BIN$ and $FCR \in BIN$ together with Equation (28). *FRC=1* and *FCR=0* means that *frame-row-column* is considered as an optimal solution, or *FRC=0* and *FCR=1* means that *frame-column-row* is considered.

$$FRC + FCR = 1 \tag{28}$$

It is then possible to model the memory storage requirement of the neighborhood implementation, Equation (29), as well as the intermediate buffering, Equation (30), regardless of execution order.

$$NSZ_{AB} = FRC \cdot NSZ_{AB}^{FRC} + FCR \cdot NSZ_{AB}^{FCR} \tag{29}$$

$$BUF_{AB} = FRC \cdot BUF_{AB}^{FRC} + FCR \cdot BUF_{AB}^{FCR} \tag{30}$$

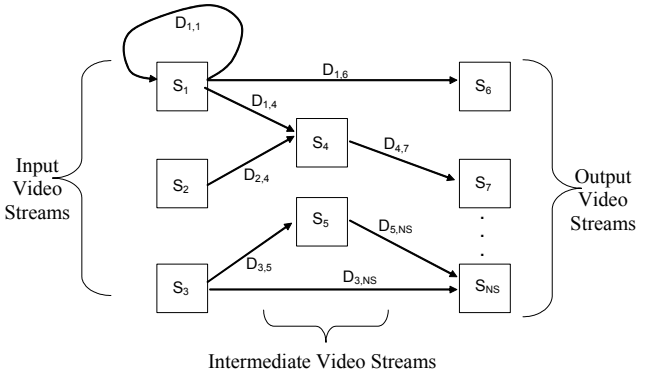


Fig. 14. Video Processing Data Flow Graph (VPDFG).

D. How to solve the optimization model

The optimization model presented in Section VI-B and VI-C is an INLP and has to be solved using techniques from global optimization calculating many local optimum and then finding out which one of these that is globally optimal [33]. This can be done directly with the LINGO software we use [34]. The solution time is several minutes for our test problem however, and it is probably not possible to solve larger problems at all. Below we will therefore present a way to solve our problem indirectly by transforming the INLP to an Integer Linear Program (ILP) in the modeling phase. Hence, we do not have to care about how to directly solve the INLP formulated in Section VI-B and VI-C.

The key observation, that allows us to reformulate the INLP, is that since the binary decision variables FRC and FCR could not be set simultaneously the modeling will be much more efficient if we change the model in the following way.

1. Define an input parameter vector *[FRC, FCR]* that take the values *[1,0]* or *[0,1]*.
2. Skip constraint (28) in Section VI-C, since *FRC* and *FCR* no longer are binary variables but constants.
3. Compare the solutions from setting *[FRC, FCR]=[1,0]* or *[0,1]*.
4. Pick the one that gives the lowest objective value of the ILP.

This procedure transforms the INLP to an ILP that has to be solved twice (one for each combination in 3.) and we also get rid of the binary variables. Another advantage of this procedure is that ILPs are frequently solved problems in operations research. Hence, the availability of software and algorithms is much larger for this problem class then for INLPs.

The LINGO software, that we have chosen, is a modeling language that includes several solvers for linear and non-linear programming where the variables can be continuous, binary or integer. The ILP is solved in the following standard way by LINGO.

1. The linear solver is used to calculate an LP that gives a lower bound to the ILP relaxing all variables to be continuous instead of integer. The solution to this LP is obtained through the simplex method [35].
2. The integer pre-solver adds cutting planes through constraints to tighten the feasible area for the ILP [34][35][36]. The integer solver searches the tightened feasible area, starting from the estimated lower bound, with a method called branch and bound [34][35][36]. It continues until the solver reaches the optimal solution to the ILP.

Note that the branch and bound procedure in general can be very costly since these types of problems in general are non-convex [37]. However, if cutting planes can be added in an efficient manner to approximate the feasible area of the ILPs constraint set, the solution will be swiftly obtained. If we want to stop the search to get a near-optimal solution this is also always possible and an upper bound of the solution gap will always be available.

In our case the solution time for the ILP is only fractions of a second instead of some minutes for the INLP. This indicates the possibility that through transforming the INLP to an ILP in the modeling phase, we can solve much larger problems then the one we present here.

VII. Experiments on a Video Based Surveillance System

One example of an RTVPS is a smart camera, a term normally used for a video camera with built-in computing power [38]. Smart cameras are, for instance, used in robotic vision where the information leaving the camera can be statistics based on various objects [39]. The most important idea behind the smart camera architecture is to collect video data and then analyze, interpret, and reduce the information before it leaves the camera.

The recent terror attacks have definitely made active video camera based surveillance systems a "hot" research area [40]. This has motivated a set up of an experiment using a surveillance camera for the observation of a car park [41]. A captured video sequence from this video camera is used as input stimuli for simulation of a smart camera [38]. A video analysis algorithm that tracks and highlights moving objects is simulated. The simulation is performed using the C++ based modeling library, IMEM [8]. The goal of this experiment is to apply the optimization model presented in Section VI, analyze the resulting memory hierarchy, and thereby demonstrate the effectiveness of the selected approach. The next section will give a system description of the modeled surveillance camera.

A. Surveillance camera system description

Fig. 15A depicts an SDFG of the modeled surveillance system. The input video camera source is labeled *A*. Nodes *B* to *L* are all neighborhood oriented video operations. Fig. 15B depicts a series of sample output frames from nodes *A, C, E, H* and *K*. An intuitive description now follows of the computation of intermediate- and output video streams.

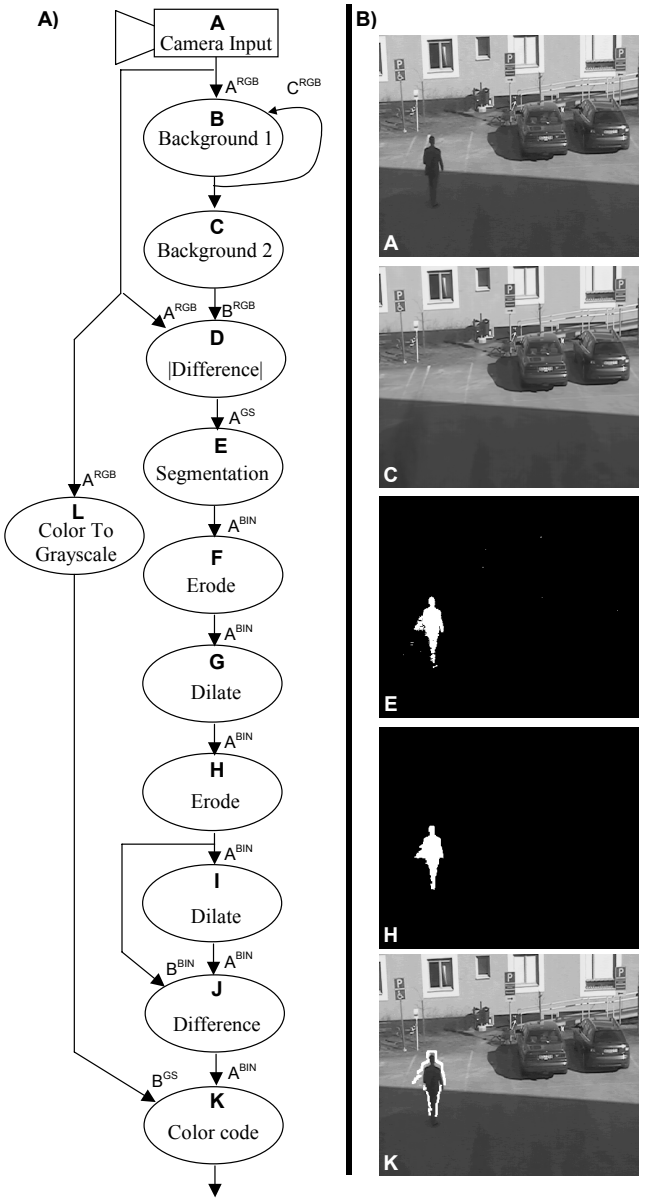


Fig. 15. Surveillance system data-flow graph and output sample frames.

Sample frame *A* in Fig. 15B shows a car park captured by the color video camera. The monitored space has been free of moving objects for a considerable amount of time, but suddenly someone is approaching the cars.

Sample frame C shows the color background without the moving person. The background frame is computed from a temporal low-pass Infinite Impulse Response (IIR) filter at node *B* and *C*. The input frame is then subtracted from the background at node *D* and a threshold is applied at node *E* resulting in a binary frame. The corresponding sample frame *E* in Fig. 15B depicts a white object on a black background. The system is thus able to extract the moving object from the background.

Operations *F, G,* and *H* implement a morphological noise reduction filter by means of a sequence of erode, dilate and erode again [10]. The broken white areas on the white object in sample frame *E* are reduced, as can be seen in frame *H*.

Operations *I* and *J* implement a morphological edge detection [10]. Operation *L* converts the input color video stream into a grayscale video stream. At operation *K*, the detected edge of the moving object is merged with the input grayscale video stream such that the edge is highlighted red. Colors are not visible in this paper but a white highlighted edge can be seen surrounding the moving person in sample frame *K*.

The complete system description was captured by 970 lines of source code using the IMEM library. IMEM is very efficient in terms of code lines. Only about 40 percent code lines are required if compared with using SystemC untimed modeling style [7].

B. Results

The impact of the optimization model presented in Section VI on the surveillance camera design in Section VII is demonstrated in Table I. This table summarizes the input- and output parameters from the general INLP solver Lingo. The numbers in Table I are based on a frame size of *R=460* and *C=620* pixels. The optimal order of execution where found to be *frame-column-row*. The first column in Table I is a list of all nodes in the SDFG. In Fig. 16, we see that intermediate buffers labeled *M*, *N*, *O* and *P* are inserted as additional nodes in the SDFG. Column 2 indicates the input stream for each node. Column 3 and 4 are the size parameters for each operator. All neighborhoods are square sized in the spatial dimensions and thus, $W^R = W^C$. Column 5 indicates the occurrence of an additional frame delay L^F, which is the case for the single recursive dependency. Column 6 lists the latencies for all operators T^{OP}, as defined by Equation (23) and for all intermediate buffers T^{BUF}, defined by Equation (25). To simplify the analysis, the latencies T^{PL} caused by the pipeline stages of the computation are all assumed to be one. Normally, there will be a larger variety of the number of pipeline stages that reflects the variety of complexity of the computation being performed. Column 7 lists the memory storage requirement caused by the neighborhood for each operator input *NSZ*, see Equation (29), and for all intermediate buffers *BUF*, see Equation (30) and (20). The last column lists the storage required by the computational pipeline stages PS, see Equation (21). Nodes B and C have image-processing operators with pure temporal behavior. These dependencies are assumed to be implemented using a large off-chip memory as frame buffers. The sum of all off-chip storage at the bottom of Table I includes the maximum of the operators B and C and the buffer P. This is because these dependencies can share the same storage. The size of the buffer at node O is reduced by two thirds from 132912 to 44304 bits by re-allocating the buffer from the non-optimal Allocation 1 to the optimal Allocation 2. The size of the non-optimal Allocation 1 could be calculated when the bit-width of the output of operation L was increased to 25 bits. The total on-chip memory storage is summarized for Allocation 1 and Allocation 2. The storage requirement of buffer M is not included in these two sums because this dependency can be shared with the neighborhood of node I. We see that Allocation 2 reduces the total on-chip memory storage by 61 percent compared to Allocation 1.

TABLE I
TIMING AND MEMORY REQUIREMENT FOR ALLOCATION 1 AND 2

Node	In	$W^C=W^R$	W^F	L^F	T^{BUF}/T^{OP} [Cycles]	NSZ/BUF [Bits]	PS [Bits]
B	A	1	1	0	ξ=1	0	24
	B	1	2	-1	ξ=1	6844800	
C	A	1	3	0	α=1	13689600	24
D	A	1	1	0	β=1	0	8
	B	1	1	0	β=1	0	
E	A	1	1	0	φ=1	0	1
F	A	5	1	0	λ=923	1844	1
G	A	9	1	0	σ=1845	3688	1
H	A	5	1	0	λ=923	1844	1
I	A	9	1	0	σ=1845	3688	1
J	A	1	1	0	β=1	0	1
	B	1	1	0	β=1	0	
K	A	1	1	0	ρ=1	0	24
	B	1	1	0	ρ=1	0	
L	A	1	1	0	τ=1	0	8
M	-	-	-	-	σ=1845	1845	0
N	-	-	-	-	$\xi+\alpha$=2	48	0
P	-	-	-	-	μ= 285199	6844776	
O	-	-	-	-	χ=5538	132912	0
On-chip storage for Allocation 1						144024	
O	-	-	-	-	χ=5538	44304	0
On-chip storage for Allocation 2						55416	
Total off-chip storage						13689600	

Table II shows the timing and memory requirement for the same surveillance camera when a single bit-width of 24 bits is specified for all video streams. In this case, the chosen allocation for buffer *O* has no effect on the total memory storage size.

C. Analysis

In this section, we will analyze the optimized memory hierarchy that was calculated and summarized in Table I. The scheduling of the data flow graph is done with maximal parallelism of the computation. All nodes fire at every clock cycle such that the throughput of all video-processing operations is one pixel per clock cycle. This scheduling results in a timing constraint, as also described in Section I. For operators with more than one input, pixels have to arrive at the exactly right clock cycle. Otherwise, data dependencies will be violated. A manual timing analysis will be applied on the surveillance camera, that way verifying the correctness of the optimized memory hierarchy. This manual analysis is possible here because of the very simple pipeline assumptions used in our example. For more realistic pipelines with varying latencies, an automated technique like ours is necessary. It also assumes a detailed knowledge of the architecture and algorithms, which cannot be presumed for the user of our RTVPS tool suit. In Fig. 15A, we see that the timing

TABLE II
TIMING AND MEMORY REQUIREMENT USING A SINGLE BIT-WIDTH OF 24 BITS

Node	In	W^C= W^R	W^F	L^F	T^{BUF}/T^{OP} [Cycles]	NSZ/BUF [Bits]	PS [Bits]
B	A	1	1	0	ξ=1	0	24
	B	1	2	-1	ξ=1	6844800	
C	A	1	3	0	α=1	13689600	24
D	A	1	1	0	β=1	0	24
	B	1	1	0	β=1	0	
E	A	1	1	0	φ=1	0	24
F	A	5	1	0	λ=923	44256	24
G	A	9	1	0	σ=1845	88512	24
H	A	5	1	0	λ=923	44256	24
I	A	9	1	0	σ=1845	88512	24
J	A	1	1	0	β=1	0	24
	B	1	1	0	β=1	0	
K	A	1	1	0	ρ=1	0	24
	B	1	1	0	ρ=1	0	
L	A	1	1	0	τ=1	0	24
M	-	-	-	-	σ=1845	44280	0
N	-	-	-	-	ξ+α=2	48	
P	-	-	-	-	μ= 285199	6844776	0
O	-	-	-	-	χ=5538	132912	0
On-chip storage for Allocation 1						398496	
O	-	-	-	-	χ=5538	132912	0
On-chip storage for Allocation 2						398496	
Total off-chip storage						13689600	

constraint must be resolved for operations *D*, *J* and *K*. In Fig. 16, additional buffers labeled *M*, *N* and *O* are inserted. The latencies for each of the nodes *B* to *L* are denoted as ξ, α, β, φ, λ, σ, ρ and τ, $\xi \in Z^+$, $\alpha \in Z^+$, $\beta \in Z^+$, $\varphi \in Z^+$, $\lambda \in Z^+$, $\sigma \in Z^+$, $\rho \in Z^+$, $\mu \in Z^+$ and $\tau \in Z^+$. The latency for the buffer at node *M* equals the latency of node *I* in order for the frames of the two input streams to appear simultaneously without any phase shift at node *J*. Similarly, the latency for the buffer at node *N* equals the sum of the latencies of node *B* and *C* in order for the frames of the two input streams to appear simultaneously without any phase shift at node *D*. As for the buffer at node *O*, the first conclusion to be drawn is that $2\beta+\varphi+2\lambda+2\sigma \geq \tau$ and thus it is necessary for the buffer to be allocated to the left-side path in the dataflow depicted in Fig. 16. The timing constraint for the two input streams at node *K* can then be expressed as,

$$\chi + \tau = 2\beta + \varphi + 2\lambda + 2\sigma . \quad (31)$$

The right-hand side of Equation (31) corresponds to the chain of latencies of the right-hand side dataflow path in Fig. 16. The left-hand side of the equation corresponds to the left-hand side of the dataflow. Two different allocations are possible for node *O*. The buffer at node *O* will have the same latency, i.e. the number of elements, for both *Allocation 1* and *Allocation 2*. However, the fact that the bit-width of the *L*'s output stream is a third of the bit-width of its input stream makes *Allocation 2* favorable. Buffer *P* has the latency of one frame minus one clock cycle. This buffer is introduced because $L^F = -1$ for the recursive dependency. However, this frame buffer must be reduced in length equal to the number of pipeline stages of operation *B*.

Table II present the results from using one single bit-width for all video streams. 24 bits was chosen for this experiment since it is the maximum specified bit-width used for the results in Table I. This means that our optimization approach reduces the on-chip memory storage by 86 percent compared to existing tool sets based on global loop transformations. This is possible since we uniquely specify the bit-widths of all video streams within our polyhedral program model.

The manual timing analysis performed in this section verifies the correctness of the optimization model. The analysis is similar to the timing constraints applied in the work of Leiserson et al. [15]. However, this one-dimensional analysis can only be applied after that the order of execution is specified, in this case, *frame-column-row*. The presented optimization model will minimize the memory storage size for all three hierarchy layers. Lower storage requirement will

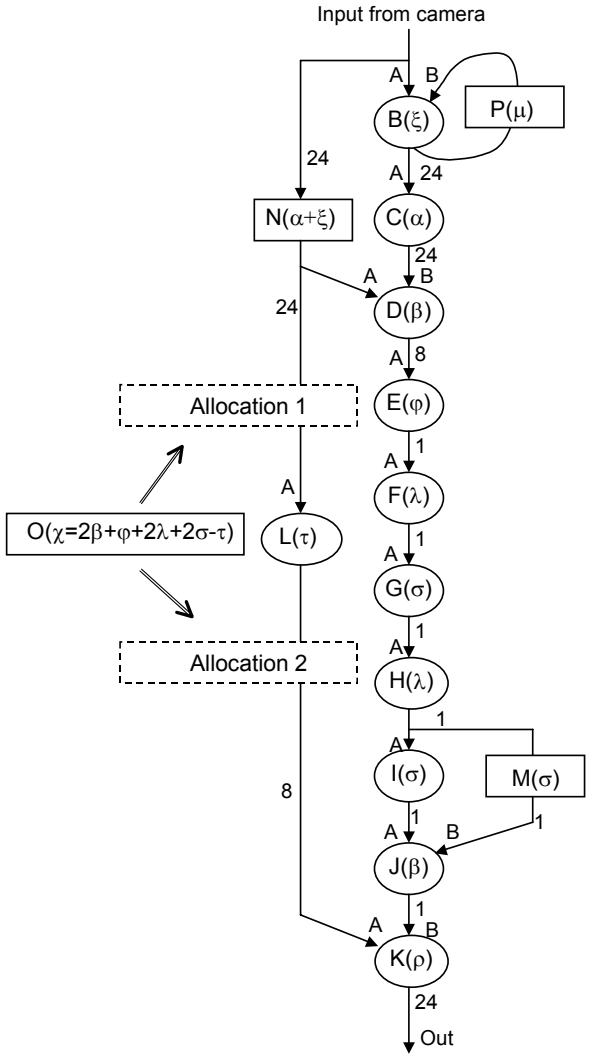


Fig. 16. Surveillance system timing diagram

TABLE III
OPTIMIZATION RESULTS FROM SYNTHETIC TEST CASES.

		INPUT PARAMETERS							OUTPUT PARAMETERS					
C	Ln	Stream	BTW [bits]	T^{PL} [cycles]	D	W^R	W^C	W^F	T^{BUF} [cycles]	T^{NH} [cycles]	NS [bits]	PS [bits]	(O^F,O^R,O^C)	(FRC, FCR)
1	1	1	8	-	1,2	3	3	1	0	461	11024	-	(0,0,0)	(0,1)
	2				1,3	1	1	1	1378	0				
	3	2	8	3	2,4	5	5	1	0	922	14752	24	(0,4,1)	
	4	3	24	8	3,4	1	1	1	0	0	0	192	(0,6,3)	
	5	4	8	5	-	-	-	-	-	-	-	40	(0,11,3)	
	6	Total minimized storage requirement for test case 1 →									26032 bits			
2	7	1	8	-	1,2	3	3	1	0	461	7376	-	(0,0,0)	(0,1)
	8				1,3	1	1	1	922	0				
	9	2	8	3	2,4	5	5	1	0	922	14752	24	(0,4,1)	
	10	3	4*	8	3,4	1	1	1	456	0	1824	32	(0,10,2)	
	12	4	8	5	-	-	-	-	-	-	-	40	(0,11,3)	
	13	Total minimized storage requirement for test case 2 →									24048			
3	14	1	8	-	1,2	1*	1*	1	0	0	0	-	(0,0,0)	(0,1)
	15				1,3	1	1	1	0	0				
	16	2	8	3	2,4	5	5	1	0	922	14752	24	(0,3,0)	
	17	3	4	8	3,4	1	1	1	917	0	3668	32	(0,8,0)	
	18	4	8	5	-	-	-	-	-	-	-	40	(0,10,2)	
	19	Total minimized storage requirement for test case 3 →									18516			
4	20	1	8	-	1,2	1	1	1	0	0	0	-	(0,0,0)	(0,1)
	21				1,3	1	1	1	0	0				
	22	2	8	3	2,4	5	5	1	0	922	14752	24	(0,3,0)	
	23	3	4	1*	3,4	1	1	1	924	0	3696	4	(0,1,0)	
	24	4	8	5	-	-	-	-	-	-	-	40	(0,10,2)	
	25	Total minimized storage requirement for test case 4 →									18516			
5	26	1	8	-	1,2	1	1	1	0	0	0	-	(0,0,0)	(1,0)
	27				1,3	1	1	1	0	0				
	28	2	8	3	2,4	1*	5	1	0	2	32	24	(0,0,3)	
	29	3	4	1	3,4	1	1	1	4	0	16	4	(0,0,1)	
	30	4	8	5	-	-	-	-	-	-	-	40	(0,0,10)	
	31	Total minimized storage requirement for test case 5 →									116			

consequently result in less used FPGA block-RAMs as well as a smaller background memory. This will in turn lead to reduced power dissipation [31]. However, the complete workflow as described in Section III is not implemented yet and FPGAs are not used in our experiments. Instead, we report the results from executing the presented optimization model using the Lingo software.

VIII. EXPERIMENTS ON A SYNTHETIC VIDEO SYSTEM

We have done additional experiments on a synthetic RTVPS that has a well specified VPDFG, but with an undefined arbitrary computation. The purpose of this experiment is to demonstrate how the optimization model responds to a selection of changes of the input parameters.

A. Results

Fig. 17 shows the simple VPDFG that we have selected for the synthetic RTVPS. The corresponding set of input- and output parameters are shown in Table III. The values in this table are based on a frame size of *R=460* and *C=620* pixels. The first column lists a sequence of 5 cases of different parameter inputs. The parameters that have changed from one case to the next is indicated with a star *. The second column *Ln,* is a list of table lines. The column labeled *Stream* is a list of the video streams, numbered according to the VPDFG shown in Fig. 17. The following columns, labeled *BTW* and T^{PL}, are input parameters as defined in Section IV. *D* is a column that lists the dependencies between video streams, as defined in the VPDFG. The following columns, labeled W^R, W^C and W^F, are the geometrical dimensions of a neighborhood, as depicted in Fig. 9. There are six columns of

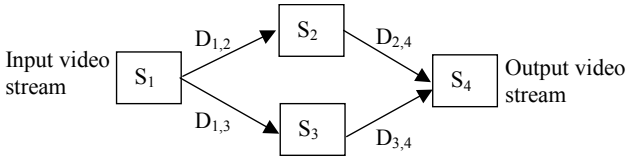


Fig. 17. VPDFG of a synthetic RTVPS.

output parameters labeled T^{BUF} , T^{NH}, *NS*, *PS*, (O_F,O_R,O_C) and *(FRC,FCR)*, which are defined in Sections IV and VI. These output parameters are the result of the execution of the optimization model defined in Section VI, taking the inputs from the VPDFG in Fig. 17 and the inputs from the parameters in Table III.

B. Analysis

The first case in Table III shows an example where intermediate data storage is calculated and placed in the dependency $D_{1,3}$, going from video stream *1* to *3*, see Fig. 17. The buffer must be positioned along the lower data path to make the sums of latencies for the upper- and lower data path equal. It is positioned in dependency $D_{1,3}$ because of the larger bit-width for video stream *3* and consequently larger buffer size for dependency $D_{3,4}$.

In case 2, due to the decrease in bit-width for stream *3*, to *4* bits, parts of the intermediate storage is shifted to dependency $D_{3,4}$. The neighborhood implementation of dependency $D_{1,2}$ can share data storage with the intermediate storage positioned in dependency $D_{1,3}$. This is also the reason why in this case, not all intermediate data storage is shifted. Note that the origins of the polytopes corresponding to video streams *1* and *4* have not changed between case *1* and *2*. This means that the shift of the origin of the polytopes, corresponding to stream *3*, corresponds to a retiming of the VPDFG. This retiming operation, performed within the polyhedral space, is constrained by the change of bit-widths.

In case *3*, the neighborhood parameters for dependency $D_{1,2}$ is reduced to $(W^F ,W^R, W^C)=(1,1,1)$. T^{NH} on line *14* is now zero, which means that neighborhood implementation is thus removed to zero length. The difference between T^{NH} on line *16* and T^{BUF} on line *17* is now equal to the difference of pipelining T^{PL} for stream *2* and *3* on line *16* and *17*. This can be explained by the timing, which requires that the sums of the latencies for the upper and lower data path in Fig. 17 are equal.

In case *4*, the number of pipeline stages for computing stream *3*, is reduced from *8* to *1*. As expected, the same reduction can also be observed on the inner dimension of the polytope origins, see O^R on line *17* and *23*. We can also observe that the total storage requirement has not changed from case *3* to *4*. This is because the mentioned reduction of pipeline stages is compensated for by the equally increased buffer size on dependency $D_{3,4}$. The total storage requirement is the sum of node storage *NS,* and the sum of pipeline register storage *PS* for all streams in the VPDFG, see Equation (49) in appendix. This can easily be verified for all five cases.

In case *5*, the geometrical dimensions of the neighborhood are changed such that the optimal order of execution is changed from *frame-column-row* to *frame-row-column.* This modified unsymmetrical neighborhood spans five columns and one row. Consequently, spanning those columns in the inner loop dimension reduces the storage cost, which motivates the change of optimal loop ordering.

IX. Discussion

In this section, we will discuss the presented formal optimization model with respect to related research.

The formal model presented in Section VI is an Integer Non Linear Program with its objective function tuned to minimize the total storage requirement. The target application RTVPS, operates on multidimensional data. A polyhedral model is thus chosen to express the relation between multidimensional iteration nodes, where data are produced and consumed. The polyhedral model allows us, regardless of loop nest ordering, to express constraints that guarantee the legacy of the data flow dependencies. The perfect regularity of the data accesses of RTVPS, further allows us to simplify its description. A set of dependency vectors is simply modeled by the spanning dimensions of the corresponding pixel neighborhood. This simplification reduces the complexity of a dependency into a bounding box model that can be compared with the orthogonal simplification of irregular dependencies, as used in [29].

The loop order *frame-row-column*, or *frame-column-row*, as expressed in our model, is equivalent of doing loop interchange of the common iteration space and is defined in [42] as loop ordering. In their work, it was proved that the incorporation of the loop ordering in the linear part of the affine transformations, results in lower complexity in the translation part of the affine transformations used for global memory optimization in the DTSE methodology. The linear transformation increases the regularity of the dependency vectors, enabling more space for data locality improvement at the translation step [25]. Since RTVPS are perfectly regular in their data accesses, except for frame boundaries, we only consider the ordering in the linear transformation part, which do not affect the perfect regularity. The linear transformation step is therefore not necessary in this case and can simply be ignored except of the loop interchange for loop ordering. To remove the linear transformation step except the loop ordering reduces the problem's cost complexity and makes the formalized INLP solving feasible. Since only loop ordering and translation is concerned, the INLP problem can be simplified to two instances of ILP problems, which are more computationally effective. By this transformation, we consider only one possible loop order at each problem instance by using different sets of equations. To express global loop ordering as a linear matrix operation as in [42], or using two sets of equations, is in our case just a matter of selected formalism. We further take into account the bit-width specification, which is not considered in their work but is shown to have a significant impact on the total storage requirement in this work.

The proposed model is the mathematical description of the storage size requirement for Real Time Video Processing Systems (RTVPS). The model is derived from a high abstraction-level model of the system described using the IMEM library. The IMEM library is an extension of the SystemC library. The model allows us to catch several

important concepts:

1. Intermediate buffer delay.
2. Neighborhood implementation delay.
3. Pipelining effects.
4. Bit-width

The optimization goal is to achieve the minimal storage size requirement w.r.t. all of those concepts. This is achieved by solving the INLP problem, where the ordering freedom and the translation freedom (i.e. loop fusion and loop shifting) is given and the bit-widths are taken into account. To our knowledge, nobody has previously combined the high abstraction-level system modeling for RTVPS with storage size estimation and optimization taking the four important concepts mentioned above into account.

X. Conclusion

We have presented a formal model that captures the optimization of the memory hierarchy for neighborhood oriented RTVPS. This formal model exploits simplifications inherited from the coarse granularity of the VPDFG and the almost perfect regularity of data accesses for RTVPS. This complexity reduction makes the model solvable using a general ILP-solver.

Using a surveillance camera design as an application example, we have demonstrated a reduction of the on-chip memory by as much as 61 percent compared to a non-optimal memory hierarchy. A retiming of the VPDFG performed this significant reduction of memory storage. We show that this retiming can be efficiently incorporated as a polyhedral operation, constrained by the specified bit-width information.

If compared with existing tool sets based on global loop transformations, where all data types are specified to a single bit-width, the reduction of the on-chip memory storage was 86 percent using our approach.

We expect the optimization result to be in the same order of magnitude for other RTVPS, having parallel paths in the dataflow graph and diversity in data types.

The memory storage optimization technique presented in this paper is developed and demonstrated for RTVPS. However, the same technique can be used for a broader range of applications that can be described using the SDFG model, such as wireless baseband, audio and image processing.

Appendix

The complete INLP optimization model is listed in this appendix. Firstly, the sets of, input- and output parameters are listed. Then the INLP is listed with its objective function and all its constraints.

A. Sets

Following is a list of all data sets that the model operates on.

- SI, S and D as previously defined in Section VI-A.
- $BIN \in \{0,1\}$, a binary variable space.
- $ODD^+ \in \{x \mid x = 2 \cdot y + 1 \wedge y \in Z^+\}$ are all positive odd integers.

B. Input parameters

Following is a list of all the model's input parameters.

- $MO \in Z^+$ is the maximum coordinate value in all dimensions for a polytope origin.
- $N^C \in Z^+$ is the number of columns in a video frame.
- $N^R \in Z^+$ is the number of rows in a video frame.
- $USE_{AB} \in BIN$ is a variable that is set to *1* if the data flow dependency stemming from the video stream indexed by $A \in SI$ going to video stream indexed by $B \in SI$ exists in the VPDFG, otherwise it is set to *0*.
- $REC_{AB} \in BIN$ is a variable that is set to *1* if the data flow dependency stemming from the video stream indexed by $A \in SI$ going to video stream indexed by $B \in SI$ and $A = B$, meaning that the dependency is recursive, else it is set to *0*.
- $W_{AB}^F \in ODD^+$ is the size of the pixel neighborhood in the frame dimension as defined in Section IV-G. This variable applies for the data flow dependency stemming from the video stream indexed by $A \in SI$ going to video stream indexed by $B \in SI$.
- $W_{AB}^R \in ODD^+$ is the size of the pixel neighborhood in the row dimension as defined in Section IV-A. $W_{AB}^C \in ODD^+$ is the size of the pixel neighborhood in the column dimension as defined in Section IV-A.
- $L_{AB}^F \in Z$ is the size of the additional frame delay as defined in Section IV-G.
- $BTW_Q \in Z^+$ is the bit-width of each element of the data array associated with the video stream indexed by $Q \in SI$.
- $T_Q^{PL} \in Z^+$ is the number of pipeline stages for the computation of each video stream indexed by $Q \in SI$.
- $IO_Q \in BIN$ is set to *0* if the video stream indexed by $Q \in SI$ is an input video stream, otherwise it is set to *1*.

C. Output parameters

Following is a list of the model's all output parameters.

- $PS_Q \in Z^+$ is the memory storage requirement caused by pipelining of the computation of the video stream indexed by $Q \in SI$, see Equation (21).
- $NS_Q \in Z^+$ is the total memory storage requirement represented by all data flow dependencies stemming from the source stream indexed by $Q \in SI$, to all other destination streams dependent on stream *Q*. NS_Q assumes that all data flow dependencies stemming from node *Q* share buffers, see Section IV-I.

- $NSZ_{AB} \in Z^+$ is the memory storage requirement corresponding to the neighborhood implementation of the data flow dependency from stream $A \in SI$ to stream $B \in SI$. NSZ_{AB} is thus associated with each edge in the VPDFG.
- $BUF_{AB} \in Z^+$ is the memory storage requirement of the intermediate buffer associated with the data flow dependency (edge) from stream $A \in SI$ to stream $B \in SI$, see Equation (20) and (37).
- $T_{AB}^{OP} \in Z^+$ is the latency caused by a video processing operation associated with the data flow dependency from stream $A \in SI$ to stream $B \in SI$, see Equation (23).
- $T_{AB}^{BUF} \in Z^+$ is the latency caused by an intermediate data storage buffer associated with the data flow dependency from stream $A \in SI$ to stream $B \in SI$.
- $FRC \in BIN$ is a binary variable that is set to *1* if the iteration space traversal is selected to *frame-row-column*, else it is set to *0*.
- $FCR \in BIN$ is a binary variable that is set to *1* if the iteration space traversal is selected to *frame-column-row*, else it is set to *0*.
- $STORE \in Z^+$ is the total estimated memory storage required by the whole VPDFG.
- $\vec{O}_Q = \left(O_Q^F, O_Q^R, O_Q^C\right) \in Z^3$ is a tuple of coordinate values of the origin for the polytope indexed by $Q \in SI$.

D. Polyhedral model

Following is a list of basic expressions for the latencies and memory storage sizes of intermediate buffers and neighborhoods developed from a polyhedral model.

$$NSZ_{AB}^{FCR} = BTW_A \cdot \begin{pmatrix} N^C N^R (W^F - 1) + N^R \cdot \left(W^C - 1\right) \\ + W^R - 1 \end{pmatrix} \tag{32}$$

$$NSZ_{AB}^{FRC} = BTW_A \cdot \begin{pmatrix} N^C N^R (W^F - 1) + N^C \cdot \left(W^R - 1\right) \\ + W^C - 1 \end{pmatrix} \tag{33}$$

$$T_{AB}^{FRC,NH} = N^C \cdot \left(\frac{W^R - 1}{2}\right) + \frac{W^C - 1}{2} \tag{34}$$

$$T_{AB}^{FCR,NH} = N^R \cdot \left(\frac{W^C - 1}{2}\right) + \frac{W^R - 1}{2} \tag{35}$$

$$T_{AB}^{FRC,BUF} = \begin{pmatrix} N^C \cdot N^R \cdot \left(O_B^F - O_A^F - L^F\right) + \\ N^C \cdot \left(O_B^R - O_A^R - \frac{W^R - 1}{2}\right) + O_B^C - O_A^C - \frac{W^C - 1}{2} - T_B^{PL} \end{pmatrix} \tag{36}$$

$$T_{AB}^{FCR,BUF} = \begin{pmatrix} N^C \cdot N^R \cdot \left(O_B^F - O_A^F - L^F\right) + \\ N^R \cdot \left(O_B^C - O_A^C - \frac{W^C - 1}{2}\right) + O_B^R - O_A^R - \frac{W^R - 1}{2} - T_B^{PL} \end{pmatrix} \tag{37}$$

E. Objective function

$$Minimize \sum_{Q \in SI} NS_Q \tag{38}$$

F. Constraints

$$\forall A \in SI \wedge \forall B \in SI \quad NS_A \geq NSZ_{AB} + BUF_{AB} \tag{39}$$

$$FRC + FCR = 1 \tag{40}$$

$$\forall A \in SI \wedge \forall B \in SI \quad T_{AB}^{BUF} = USE_{AB} \cdot \left(FRC \cdot T_{AB}^{FRC.BUF} + FCR \cdot T_{AB}^{FCR,BUF}\right) \tag{41}$$

$$\forall A \in SI \wedge \forall B \in SI \quad BUF_{AB} = BTW_A \cdot T_{AB}^{BUF} \tag{42}$$

$$\forall A \in SI \wedge \forall B \in SI \quad NSZ_{AB} = USE_{AB} \cdot \left(FRC \cdot NSZ_{AB}^{FRC} + FCR \cdot NSZ_{AB}^{FCR}\right) \tag{43}$$

$$\forall A \in SI \wedge \forall B \in SI \quad T_{AB}^{NH} = USE_{AB} \cdot \left(FRC \cdot T_{AB}^{FRC,NH} + FCR \cdot T_{AB}^{FCR,NH}\right) \tag{44}$$

$$\forall A \in SI \wedge \forall B \in SI \quad \begin{pmatrix} N^C \cdot N^R \cdot (O_B^F - O_A^F - L^F) + \\ FRC \cdot \begin{pmatrix} N^C \cdot \left(O_B^R - O_A^R - \frac{W_{AB}^R - 1}{2}\right) + \\ O_B^C - O_A^C - \frac{W_{AB}^C - 1}{2} \end{pmatrix} + \\ FCR \cdot \begin{pmatrix} N^R \cdot \left(O_B^C - O_A^C - \frac{W_{AB}^C - 1}{2}\right) + \\ O_B^R - O_A^R - \frac{W_{AB}^R - 1}{2} \end{pmatrix} \end{pmatrix} \geq REC_{AB} \cdot T_B^{PL} \tag{45}$$

$$\forall A \in SI \wedge \forall B \in SI \quad T_{AB}^{OP} = T_{AB}^{NH} + T_B^{PL} \tag{46}$$

$$\forall Q \in SI \quad PS_Q = T_Q^{PL} \cdot BTW_Q \tag{47}$$

$$\forall Q \in SI \quad (O_Q^F, O_Q^R, O_Q^C) \leq IO_Q(MO, MO, MO) \tag{48}$$

$$STORE = \sum_{P \in SI} (PS_P + NS_P) \tag{49}$$

REFERENCES

[1] F. Catthoor et al., *Custom Memory Management Methodology*. Kluwer Academic Publishers, 1998, ISBN 0-7923-8288-9.

[2] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockemeyer, P.G. Kjeldsberg, T. Van Achteren, T. Omnes, *Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, 2002, ISBN 0-7923-7689-7.

[3] E.A. Lee, "Consistency on dataflow graphs". *Proc. of the International Conference on Application Specific Array Processors*, Barcelona, Spain, 1991, pp. 355-369.

[4] C. Carreras, J.A. Lopez and O. Nieto-Taladriz, "Bit-width selection for data-path implementations". *Proc. of the 12th International Symposium on System Synthesis*, San Jose, CA, USA, 1999, pp. 114-119.

[5] Y. Cao and H. Yasuura, "Quality-driven design by bit-width optimization for video applications". *Proc. of the conf. of Asia and South Pacific Design Automation*, Kitakyushu, Japan, 2003, pp. 532-537.

[6] P. Bjuréus, M. Millberg and A. Jantsch, "FPGA Resource and Timing Estimation from Matlab Execution Traces". *Proc. of the 10th International Symposium on Hardware/Software Codesign*, Estes Park, CO, USA, 2002, pp. 31-36.

[7] B. Thörnberg, H. Norell and M. O'Nils, "IMEM: An object-oriented memory- and interface modelling approach for realime video systems". *Proceedings of the Forum on specification & Design Languages*, Marseille, 2002, ISSN 1636-9874.

[8] B. Thörnberg, H. Norell and M. O'Nils, "Conceptual Interface and Memory-Modelling for Real-Time Image Processing Systems". *Proc. of the 5th IEEE International Workshop on Multimedia Signal Processing*, St. Thomas, VI, USA, 2002, pp. 138-141.

[9] P.R. Panda, "SystemC – a modelling platform supporting multiple design abstractions". *Proceedings of the 14th International Symposium on System Synthesis*, Montreal, Quebec, Canada, 2001, pp. 75-80.

[10] R.C. Gonzales and R.E. Woods, *Digital Image Processing*. Addison Wesley, 1993, ISBN 0-201-50803-6.

[11] A. Bovik, *Handbook of Image&Video Processing*. Academic Press, 2000, ISBN 0-12-119790-5.

[12] B. Thörnberg, Q. Hu, M. Palkovic, M. O'Nils and P.G Kjeldsberg "Polyhedral space generation and memory estimation from interface and memory nodels of real-time video systems", *Journal of Systems and Software*, Vol 79, No. 2, pp. 231-245, Elsevier 2005.

[13] P.G. Kjeldsberg, F. Catthoor, E.J. Aas, "Storage Requirement Estimation for Optimized Design of Data Intensive Applications". *ACM Transactions on Design Automation of Electronic Systems,* 2004, Vol 1, No. 2, pp. 133-158.

[14] F. Catthoor et al., "System-level data-flow transformation exploration and power-area trade-offs demonstrated on video codecs". *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 1998, Vol. 18, No. 1, pp. 39-50.

[15] C. E. Leiserson, F. M. Rose and J. B. Saxe. "Optimizing Syncronous Circuitry by Retiming". *Proc. Of the 3rd Caltech Conference on Very Large Scale Integration*, Pasadena, CA, USA, 1983, pp. 87-116

[16] N. Passos and E. Sha. "Achieving Full Parallelism Using Multidimensional Retiming". *IEEE Transactions on Parallel and Distributed Systems,* 1996, Vol. 7, No. 11, pp. 1150-1163.

[17] A. Woodruff and M. Stonebraker, "Buffering of intermediate results in dataflow diagrams". *Proc. of the 11th IEEE International Symposium on Visual Languages*, Darmstadt, Germany, 1995, pp. 187-194.

[18] M.C. Williamson and E.A. Lee, "Synthesis of parallel hardware implementations from synchronous dataflow graph specification". *Conference Record of Thirtieth Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, USA, 1996, pp. 1340-1343.

[19] M. Ade, R. Lawereins and J.A. Peperstraete, "Buffer memory requirements in DSP applications". *Computer Systems Science and Engineering*, 1999, Vol. 14, No. 3, pp. 155-165.

[20] V. Loechner and D. Wilde, "Parameterized Polyhedra and Their Vertices", *International Journal of Parallel Programming*, 1997, Vol. 25, No. 6, pp. 525-549

[21] D.K Wilde, 1993. A library for doing polyhedral operations, Master's thesis, Oregon state university, Corvallis Oregon, Also published in IRISA technical report PI 785, Rennes, France 1993.

[22] V. Lefebrev and P. Feautrier, "Aotomatic storage management for parallel programs", *Parallel computing*, 1998, Vol. 24, No. 3-4, pp. 649-671.

[23] C. Polychronopoulos, "Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design", *IEEE Transactions on Computers,* 1998, Vol. 37, No. 8, pp. 991-1004.

[24] M. Kandemir, J. Ramanujam and A. Choudhary, "Improving cache locality by a combination of loop and data transformations", *IEEE Transactions on Computers*, 1999, Vol. 48, No. 2, pp. 159-167.

[25] K. Danckaert, F. Catthoor and H. De Man, "A preprocessing step for global loop transformations for data transfer optimization". *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, San Jose, USA, 2000.

[26] H. Norell and M. O'Nils, "A Generalized Architecture for Hardware Synthesis of Spatio-Temporal Memory Models for Image Processing Systems". Proceedings of the 12th International Workshop on Systems, Signals and Image Processing, IWSSIP'05, Chalkida, Greece 2005.

[27] B. Thörnberg, L. Olsson and M. O'Nils, "Optimization of Memory Allocation for Real-Time Video Processing on FPGA". *Proc. Of the 16th IEEE International Workshop on Rapid System Prototyping"*, Montreal, Canada, 2005, pp. 141-147.

[28] N. Lawal, B. Thörnberg and M. O'Nils, "Address Generation for FPGA RAMs for Efficient Implementation of Real-Time Video Processing Systems". *Proc. of the Conference on Field Programmable Logic and Applications,* Tampere, Finland, 2005.

[29] P.G. Kjeldsberg, F. Catthoor and E.J. Aas, "Data dependency size estimation for use in memory optimization". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 2003, Vol. 22, No 7, pp. 908-921.

[30] S. Verdoolaege, M. Bruynooghe, G. Janssens and F. Catthoor, "Multi-dimensional Incremental Loop Fusion for Data Locality". *Proceedings of the IEEE Conference on Application-Specific Systems*, *Architectures, and Processors,* The Hague, Netherlands, 2003, pp. 14-24.

[31] S. Wuytack, J.Ph. Diguet, F.Catthoor and H. De Man, "Formalized methodology for data reuse exploration for low-power hierarchical memory mappings". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1998, vol. 6, no. 4. pp. 529-537.

[32] B. Oelmann, H. Norell, R. Andersson and Y. Xu, "Design of Real-Time Signal Processing ASIC for Noise Reduction in Moving Video Images", *Proceeding of IEEE Norchip Conference*, 1999, pp. 228-33.

[33] R. Horst and H. Tuy, *Global optimization- Determinstic Approaches*, 3rd Edition, Springer Verlag, 1996, New York, USA.

[34] L. Schrage, *Optimization Modeling with Lingo*. Lindo Systems Inc, Chicago, Illinois, USA.
Available: http://www.lindo.com/lingo_textbook.html

[35] R.L. Rardin, *Optimization in Operations Research*. Prentice-Hall, 1998, Upper Saddle River, New Jersey, USA.

[36] L.A. Wolsey, *Integer Programming*. Wiley-Interscience Series in Discrete Mathematics and Optimization, 1998 New York, USA.

[37] D.P. Bertsekas, *Non-Linear programming*, Athena Scientific, 1995, Belmont, Masachuesetts, USA.

[38] W. Wolf, B. Ozer and T. Lv, "Smart cameras as embedded systems". *Computer*, 2002, Vol. 35 No. 9, pp. 48-53.

[39] D.S. Shrestha, B.L. Steward and S.J. Birrell, "Video Processing for Early Stage Maize Plant Detection". *Biosystems Engineering*, 2004, Vol. 89 No. 2, pp. 119-129.

[40] G.L. Foresti et al., "Active video-based surveillance system: the low-level image and video processing techniques needed for implementation". *IEEE Signal Processing Magazine*, 2005, Vol. 22 No. 2, pp. 25-37.

[41] B. Thörnberg, and M. O'Nils, "Impact of Bit-Width Specification on the Memory Hierarchy for a Real-Time Video Processing System". To be published in *Proceedings of Design, Automation and Test in Europe*, DATE06, Munich, Germany, 2006.

[42] S. Verdoolaege, F. Catthoor, M. Bruynooghe and G. Janssens, *Feasibility of Incremental Translation*, Report CW348, Katholieke University Leuven October 2002. http://www.cs.kuleuven.be/publicaties/rapporten/CW/2002/

Benny Thörnberg received the Certificate in electronic design in 1988, the B.Sc. EE in 2001 and the Licentiate of technology degree in 2004 from Mid Sweden University.

He has been working with a camera design for intra-oral x-ray imaging at Regam Medical Systems AB from 1990 to 1997. He is currently a lecturer and a Ph.D. student at the department of Information Technology and Media in Mid Sweden University. His main research interest is in design methods for embedded systems and in particular for real-time video processing. Currently he is working in a project involving automatic camera inspection of wood chips where new design methods are applied.

Martin Palkovic received the B.Sc. and M.Sc. degree in electrical engineering from the Slovak University of Technology in Bratislava, Slovakia.

Since 2001 he is a researcher at the Interuniversity MicroElectronics Center (IMEC) in Leuven, Belgium. His research interests include high-level optimizations in data dominated multimedia applications, related system design automation aspects and platform architectures for low power.

Qubo Hu received the M.Sc. degree in electronics system design from the Royal Institute of Technology in Stockholm, Sweden and the B.Sc degree in computer science from China.

He is currently a Ph.D student at the Norwegian University of Science and Technology in Trondheim, Norway. His research interests are in embedded systems design, with special emphasis on hw/sw codesign and the development of methods and algorithms for system-level optimizations and estimations of embedded multimedia applications for low power.

Leif Olsson received the B.Sc. degree in mathematics at the Mid Sweden University in 1998 and the PhD. degree in quantitative forest logistics at the Swedish University of Agricultural Sciences in the spring of 2004.

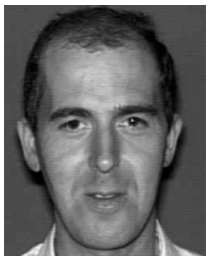
Currently he is a researcher at the Fibre Science and Communication Networks at Mid Sweden University in Sundsvall, Sweden. During his period as PhD. student he worked with the development of network optimization models for road investment- and wood supply chain coordination together with a major Swedish forest company. Ongoing research spans over the area of forest logistics, transport infrastructure management and assistance to researchers in other disciplines in there use of network optimization models. Olsson has published a number of journal and conference papers in his areas of interest.

Per Gunnar Kjeldsberg (S'98–M'01) received his M.Sc. in electrical engineering in 1992 from the Norwegian Institute of Technology in Trondheim, Norway. In 2001 he received the Ph.D. degree from the same place (now Norwegian University of Science and Technology, NTNU).

Between 1992 and 1996 he worked as design engineer at Eidsvoll Electronics AS. During his doctoral studies, he focused on storage requirement estimation and optimization for data intensive applications. Kjeldsberg has published a number of conference and journal papers, and has been coauthor of a book in his field of interest. Currently he is Associate Professor at NTNU, focusing on hw/sw codesign and system level design in general, and on data transfer and storage exploration in particular.

Mattias O'Nils received his B.Sc. EE degree from Mid Sweden University in 1993 and his Licentiate/PhD degrees in electronics system design from The Royal Institute of Technology in Stockholm, Sweden, in 1996 and 1999 respectively.

In 2006 he became full professor and leads a research group in embedded systems design at Mid Sweden University. His research interest spans over design methods and implementation of embedded systems with a specific interest in implementation of real-time video processing systems.



Francky Catthoor is an IMEC Fellow and a part time full professor in the Electrical Engineering Department of the Katholieke Universiteit Leuven. His research interests include architecture design methods and system-level exploration for power and memory footprints within real-time constraints. Catthoor has an MEng and a PhD in electrical engineering from the Katholieke Universiteit Leuven. He is an IEEE Fellow.