Polyhedral space generation and memory estimation from interface and memory models of real-time video systems

Benny Thörnberg[a,*], Qubo Hu[b], Martin Palkovic[c], Mattias O'Nils[a], Per Gunnar Kjeldsberg[b]

[a]Mid-Sweden University, Holmgatan 10, 851 70 SUNDSVALL, Sweden, {benny.thornberg, mattias.onils}@mh.se

[b]Norwegian University of Science and Technology, 7491 Trondheim, Norway, {qubo.hu, per.gunnar.kjeldsberg}@iet.ntnu.no

[c]IMEC, B-3001 Leuven, Belgium, +32 1628 1679 Martin.Palkovic@imec.be

ABSTRACT

We present a tool and a methodology for estimating the memory storage requirement for synchronous real-time video processing systems. Typically, a designer will use the feedback information from this estimation to select the most optimal execution order for software processors or space to time mapping for hardware. We propose to start from a conceptual interface and memory model that captures memory usage and data transfers. This high-level modeling is provided as an extension library of SystemC called IMEM. A common polyhedral iteration space is generated from the model, where polytopes are placed using a new placement algorithm based on simple heuristics. This algorithm will ensure maximum freedom of selecting executing order as all negative dependencies are removed to the length of zero. A demonstration is given regarding how the polytopes and dependency vectors can then be used as input to a memory storage estimation tool called STOREQ.

Keywords

Memory storage estimation, polyhedral, polytope placement, modeling, real-time, video, SystemC

1. INTRODUCTION

Real-time video processing systems (RTVPS) use a very large amount of data storage and transfers. This will become a major design bottleneck for embedded systems, since memories and bus transfers will consume a large amount of power, Wuytack et al. (1998). Initial work in the area of high-level synthesis of RTVPS, Thörnberg and O'Nils (2003), has indicated the need for memory storage estimation in order to efficiently map a model onto an architecture. STOREQ, Kjeldsberg et al. (2004), is a tool that can be used for the estimation of memory STOrage REQuirements for data intensive digital signal processing systems, namely DSP systems. STOREQ memory estimation includes selecting the most optimal processing order or space to time mapping with respect to storage requirements, which can guide the high level synthesis to a better result. This estimation is done using a polytope model as the input, Kjeldsberg et al. (2004).

A polytope in this model is a multidimensional body of iteration nodes when the statements with data write or data read are executed. A statement that uses data produced by an earlier statement is data dependent on the earlier statement. These dependencies are represented as dependency vectors between the iteration nodes, which can be calculated from the index expressions. Figure 1A shows an example code with a 2-dimensional loop nest. The set of iteration nodes when the statement *S1* with data array *A* write is executed is represented as polytope *A* in Figure 1B. Polytope *B* consists of the set of iteration nodes when the statement *S2* with data array *B* write and data array *A* read is executed. We simply say there are dependencies between these two polytopes. One of them is depicted as vector $\vec{v}_{12A}$ in Figure 1B. The dependency between these two polytopes is also depicted in Figure 1C

*Corresponding author: Tel: +46-60-148917, Fax: +46-60-148456, Email: benny.thornberg@mh.se

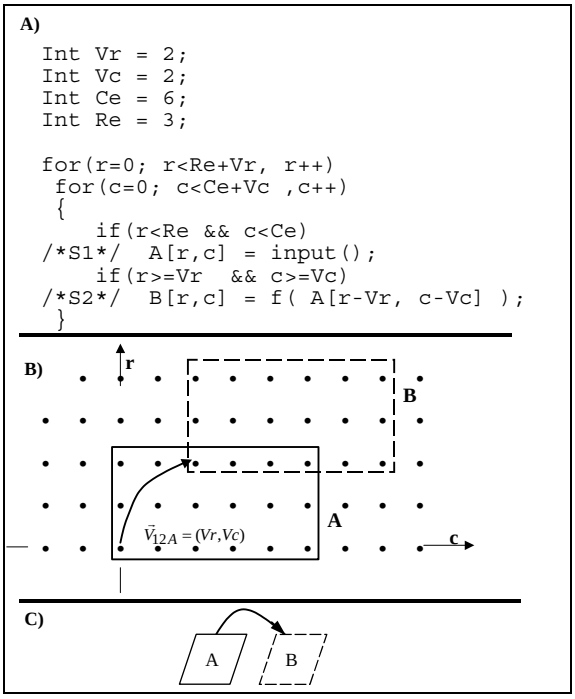

Figure 1. Example of a common polyhedral iteration space.

as a polytope dependency graph. In this directed graph, the vertices correspond to the polytopes and the single edge to the data flow dependency going from polytope *A* to *B*.

The STOREQ tool has been developed to be a part of the Data Transfer and Storage Exploration methodology from IMEC, Catthoor et al. (2002). When used in that context, a platform independent transformation step identifies the corresponding polytopes and data dependency vectors from a given application code. This step also include polytope placement into a common polyhedral iteration space, Dankaert et al. (2000). The placement algorithm used becomes complicated since both regularity and locality in data accesses are considered. Verdoolaege et al. (2003) only looks at locality and can achieve a reasonable result even for complex multimedia applications. There are also some other works in the area of polytope placement, which are referred to in section 2. However, the complexity of all these algorithms, still necessitate a complex tool for the RTVPS we are looking at. An even simpler placement algorithm would be sufficient enough and is thus developed and implemented in this work.

This paper presents the integration of STOREQ into the IMEM development workflow, Thörnberg at al. (2002). IMEM is a library extension of SystemC, Panda (2001), for modeling of neighborhood oriented multi-rate synchronous real-time video processing systems. In IMEM, no loops are initially specified and thus details related to the final implementation are excluded. In addition, the specification of coarse-grained data flow dependencies is separated from the functional specification of image processing operators, which makes code pruning unnecessary. Code pruning is a source code pre-processing step at which array accesses are separated from the computational statements. Code pruning is applied on C-code or Data Flow Language in the DTSE-methodology, Catthoor et al. (1998). The integration of STOREQ requires the development of an algorithm and a tool to support the transformation of an IMEM model into a polytope model. The fact that image frame sizes and data dependencies are captured as C++ objects in IMEM, simplifies the extraction of polytopes and dependency vectors. Neighborhood oriented RTVPS are perfectly regular in their data accesses, simplifying the polytope placement algorithm that we have developed.

This work has resulted in a new polytope placement algorithm suitable for IMEM models and implemented in the tool IMEM Projector. IMEM Projector can import an IMEM model and map this model onto a TriMedia DSP prototyping platform, Thörnberg and O'Nils (2003). To our knowledge, no similar approach exists, where

application specific, extended SystemC-based modeling is combined with the geometrical polyhedral modeling into a memory storage estimation methodology. This application specific methodology takes advantage of simplifications such as the regularity in RTVPS, the separation of data dependencies from computation and the powerful system modeling provided by SystemC. This is also the reason why we are doing polytope placement before integrating STOREQ, as opposite to other approaches. This is discussed in Section 8.3.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 describes modeling of a real-time video system in IMEM. Section 4 describes our tool IMEM Projector and motivates the existence of memory estimation in a system development workflow. Section 5 explains the polyhedral iteration space and how polytopes and dependencies can be generated from an IMEM model. The polytope placement algorithm is explained in section 6. A real-life video system is presented in section 7. Section 8 further discusses this work and compares it with others. Conclusions are finally drawn in section 9.

2. RELATED WORK

Related work includes studies of high-level synthesis and rapid prototyping of DSP applications. These studies more or less address memory storage estimation. A memory estimation method developed at Princeton is also presented.

2.1 Previous work on memory estimation

Li and Wolf (1999) have studied co-synthesis of hardware/software systems. Memory hierarchies are considered, as cyclic task graphs are mapped onto several processing elements. Their synthesis algorithm assumes that a number of parameters are extracted for each task. Memory storage requirement is one of them. This extraction is done by C program level estimation or simulation. The authors do not refer to any preprocessing step where the memory storage requirements of each task are optimized.

Grape-II is a system level prototyping environment for DSP applications developed at Katholieke Universiteit Leuven, Lauwereins et al. (1995). In Grape, a data flow graph (DFG) is drawn using a graphical design entry, while the functionality of each actor in the graph is specified using DFL/Silage, Dace (1993), C or any other language supported by a third party compiler. The workflow is well defined for this approach and memory estimation can be identified at a resource estimation phase. The memory estimation relies on commercial compilers. However, neither the relationship between memory requirements nor space to time mapping for each actor are addressed in this study. The optimization problem involved in selecting a scheduling scheme for a coarse-grained DFG in order to reduce buffers is presented, Adé (1999).

Ying and Malik (2000) at Princeton university have developed an algorithm to perform integer counting on parameterized polytopes mapped through affine indexes. A traversal of the array write- and read accesses is used to calculate the exact number of live variables during any iteration.

Grun et al. (1998) propose that the data dependency relations could be used to find the number of array elements produced or consumed by each assignment. Then, a memory trace of upper and lower bounds as a function of time is found with the peak bounding rectangle yield the total memory requirement. If the difference between the upper and lower bounds for this critical rectangle is too large, the corresponding loop is split into two loops and the estimation is rerun.

Common for the memory estimation, presented by Ying and Malik (2000) and Grun et al. (1998), is the assumption of a fixed execution order, which is one major difference in comparison with STOREQ, Kjeldsberg et al. (2004).

Additional discussions of work related to memory estimation can be found in publications written by Thörnberg and O'Nils (2003) and Kjeldsberg et al. (2003).

2.2 Previous work on polytope placement

The polytope placement has previously been studied with loop transformation techniques, such as loop fusion and loop shifting in order to improve locality and reduce memory requirement. Loop fusion is a code transformation that takes related loop nests having the same iteration space traversal and combine their bodies into a single loop nest. This transformation enhances data locality, reduces the memory requirement and enables parallelism.

However, illegal dependencies occur when fused loop instructions from previous loops depend on instructions from the following loops. Loop shifting is then introduced to solve these illegal dependencies. As loop shifting for array contraction is proven to be NP-complete even for acyclic dependency graphs with uniform dependencies, heuristics are generally used, Darte and Huard (2002). Manjikian and Abdelrahman (1995) only consider loop shifting as soon as loop fusion is illegal and therefore it can only reach partial locality improvement. Fraboulet et al. (1999) consider minimizing the memory accesses in loop nests by data temporal locality optimization. They present a polynomial algorithm for memory access optimization in a single loop and a heuristic based on this algorithm for the multidimensional case. However, it does not guarantee optimal memory access minimization since only the last accesses to the given array are optimized when several statements read this array. Song et al. (2001) use a greedy heuristic to incrementally fuse loop nests when the fusion results in less register spilling and cache misses. Otherwise, the fusion will not be done. Opposite of these works mostly looking at uniform dependencies, Verdoolaege et al. (2003) propose a method for incremental loop fusion which does not restrict to any specific types of dependencies including uniform, non-uniform and cyclic dependencies. It can handle applications with relatively larger set of dependencies. With their greedy algorithm, only the heaviest dependency (which means the highest number of data elements involved in the corresponding dependency) has the priority for fusion. It is expected to achieve a reasonable result for memory and locality optimization within a reasonable time complexity. Besides, there are other work considering locality improvement or focusing on parallelism with other loop transformation techniques, i.e. Lim et al. (2001) improve data locality using affine partitioning.

3. MODELING AND DESIGN WITH IMEM

The parallel modeling capabilities of object-oriented mechanisms, such as encapsulation, classes, objects and inheritance have resulted in the present emerging object oriented hardware description languages such as SystemC, Panda (2001). SystemC can model hardware at Register Transfer Level, RTL, allowing automatic synthesis using commercial tools. Fully synchronous RTL-models can easily be combined with un-timed models, allowing a mix of transactional and clocked simulations. This rather nice mechanism will allow a system designer to incrementally refine a model. Remote Procedure Call, Panda (2001), RPC, is the model of computation currently used in SystemC for transactional level simulation. IMEM is an extension of SystemC capable of modeling real-time video processing systems. These systems are captured in IMEM as coarse-grained multi-rate synchronous data flow graphs. All actors in this data flow graph are captured using functional modeling. This means that all implementation related details are excluded in the model. Design entry is done using object instantiations of design entities (C++ classes) that are linked together as a dynamic data structure. We prefer to call it conceptual modeling to distinguish IMEM from functional or applicative languages that requires its own compilers, such as DFL, Dace (1993). This conceptual modeling has been found to be very modeling efficient, in relation to the number of source code lines, Thörnberg et al. (2002).

Figure 2 shows one IMEM module, (actor), two input- and one output video streamers linked together in a coarse-grained DFG. The streamers are test-benches that provide and capture simulation data to and from the model. The behavior of an actor in this IMEM DFG is defined by a conceptual interface and memory model in combination with a functional description of the filter kernel. Figure 3A 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, Gonzales and Woods (1993), with the exception of image boundaries. A neighborhood of input pixels is taken as the input for the calculation of each output pixel. Figure 3B shows an example of how the same pixel neighborhood is modeled in IMEM by a collection of design entities. These entities are depicted as a UML deployment diagram. This UML diagram is only shown as a graphical illustration of object instantiations made in the C++-code. IM_LINE(0,1,1) in Figure 3B, 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 Figure 3A indicates this line.

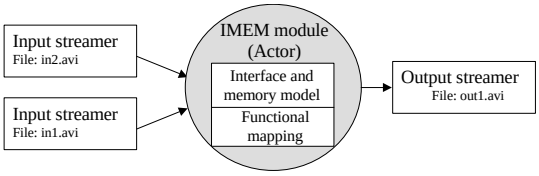

Figure 2. Data flow graph in IMEM

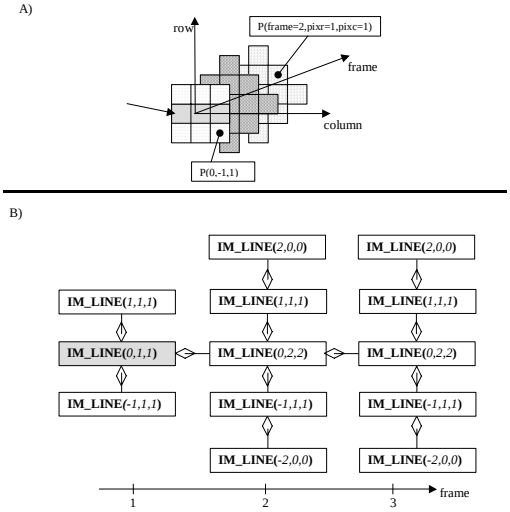

Figure 3. A 3-dimensional collection of pixels.

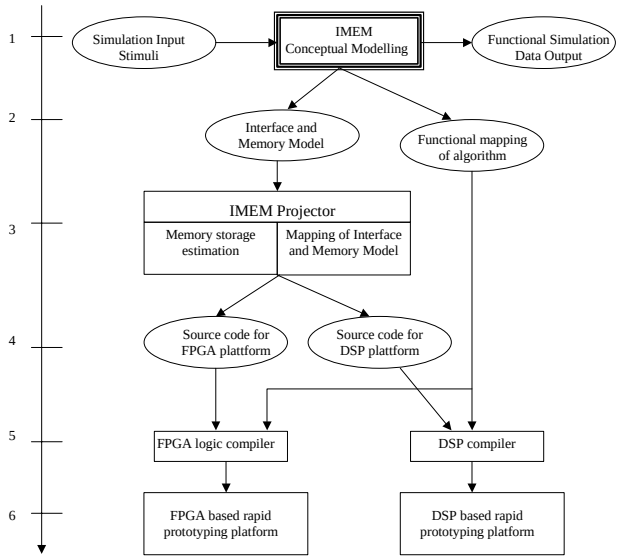

Figure 4. The IMEM system development workflow.

The first group of entities, indicated by the horizontal axis in Figure 3B corresponds to the first frame in the neighborhood in Figure 3A. Two examples of how individual pixels can be relatively addressed within the neighborhood are shown in Figure 3A. Video stream interfaces are modeled using a similar technique.

4. IMEM PROJECTOR

IMEM Projector is a tool that currently can import an IMEM model and map this model onto a TriMedia DSP prototyping platform, Thörnberg and O'Nils (2003). The work presented in this paper adds memory estimation as a new feature to this tool. IMEM Projector has a graphical user interface and the user is allowed to optimize the memory storage requirement by interactively choosing different possible assignments of processing order. The IMEM prototyping workflow depicted in Figure 4 demonstrates how memory estimation 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 direct mapping process at level 3. This mapping can be made more efficient by using the outcome from an interactive memory estimation. The most optimal processing order or space to time mapping is selected in order to reduce the memory storage requirement. The output from the direct mapping is either source code targeted for a field programmable gate array, FPGA, Norell et al. (2003), or a digital signal processor, DSP, Thörnberg and O'Nils (2003). The source code can be generated by mapping the IMEM interface and memory model with a parameterized and generic target specific model. The functional mapping of an algorithm can then interface directly with the generated interface and memory model and be compiled at level 5. Level 6 corresponds to either a FPGA- or DSP-based prototyping platform.

This work enables the use of memory storage estimation to steer the mapping process. The steering has not yet been automated within the tools, but the designer can interactively use the estimation results and by ordering guidelines can find an optimal mapping. Automation of the steering is part of our plans for future work.

5. GEOMETRICAL MODELING IN THE COMMON POLYHEDRAL ITERATION SPACE

Digital signal processing algorithms are typically described by a set of non-perfectly nested loop nests. A 2-dimensional loop nest is shown in Figure 1A. As illustrated previously, polytope A in Figure 1B consists of a multi-dimensional body of iteration nodes where the statement with data array A write is executed. Polytope B consists of a set of iteration nodes where the statement with data array A read and array B write is executed. Thus, 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*, an operation in a sequel referred to as polytope placement. During placement the polytopes are transformed to guarantee legacy and optimize for locality and regularity of the program code's data flow dependencies, Danckaert et al. (2000). 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 the storage size requirements, Kjeldsberg et al. (2004).

The rest of this section is dedicated to the definitions necessary for the understanding of the concepts of geometrical modeling in the *common polyhedral iteration space*. This geometrical model is the input to the memory storage estimation tool STOREQ.

Definition 1**:** The *common polyhedral iteration space* S_{cis} is an integer space of *n* dimensions used for the geometrical modeling of *n* iterators in a nested loop of *n* levels.

$$S_{cis} = Z^n \tag{1}$$

Definition 2 Each iteration in a loop correspond to a node in the common polyhedral iteration space and are identified by its *loop iterator vector* $\vec{I} \in S_{cis}$, Verdoolaege et al. (2003).

$$\vec{I} = (I_1, I_2, \cdots, I_n) \tag{2}$$

I_n is the value of the *n*:th iterator in the loop nest, counting from outermost to innermost iterator.

Definition 3: 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, Wilde (1993).

$$P = \left\{ \vec{I} \mid I_{1L} \le I_1 \le I_{1H} \wedge I_{2L} \le I_2 \le I_{2H} \wedge \cdots \wedge I_{nL} \le I_n \le I_{nH} \wedge \vec{I} \in S_{cis} \right\} \tag{3}$$

$\vec{I}_L = (I_{1L}, I_{2L}, \cdots, I_{nL}) \in S_{cis}$ 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, [reference?].

Definition 4: The set of iteration nodes in the common polyhedral iteration space at which a statement is executed is defined to be the statement's *iteration domain* and is denoted as a polytope, De Greef (1998).

For example, the iteration domain D_2^{iter} of statement 2 in Figure 1A is denoted as,

$$D_2^{iter} = \left\{ (r,c) \mid 2 \le r \le 4 \wedge 2 \le c \le 7 \wedge (r,c) \in Z^2 \right\} \tag{4}$$

This iteration domain simply referred to as polytope is depicted as a rectangle in Figure 1B.

Definition 5: The *index space* S_{ind} is an integer space of *m* dimensions used for the geometrical modeling of accesses to an *m*-dimensional array of variables.

$$S_{ind} = Z^m \tag{5}$$

Definition 6: The *variable domain* is a mathematical description of an array of variables. Each index node $\vec{A}$ with integer coordinates in this domain corresponds to the array index of exactly one variable in the array.

$$D_A^{\text{var}} = \left\{ \vec{A} \mid A_{1L} \le A_1 \le A_{1H} \wedge A_{2L} \le A_2 \le A_{2H} \wedge \cdots \wedge A_{mL} \le A_m \le A_{mH} \wedge \vec{A} \in S_{ind} \right\} \tag{6}$$

$\vec{A}_L = (A_{1L}, A_{2L}, \cdots, A_{mL}) \in D_A^{\text{var}}$ are the lower bounds for the array index in each dimension, and $\vec{A}_H = (A_{1H}, A_{2H}, \cdots, A_{mH}) \in D_A^{\text{var}}$ are the upper bounds, De Greef (1998).

For example, the variable domain D_B^{var} for array *B* in Figure 1A equals,

$$D_B^{\text{var}} = \left\{ (b_1, b_2) \mid 0 \le b_1 \le 9 \wedge 0 \le b_2 \le 9 \wedge (b_1, b_2) \in Z^2 \right\} \tag{7}$$

Definition 7: An index expression F_{sAp}^{ie} is an affine function $F_{sAp}^{ie} : \vec{I} \rightarrow A_p$, that describes the relation between statement *s* iteration node $\vec{I}$ and one of the dimensions *p* of the accessed array *A*'s index node $\vec{A}$, $\vec{A} = (A_1, A_2, \cdots, A_p, \cdots, A_m) \in D_A^{\text{var}}$.

Definition 8: Data are consumed from one or more arrays and produced to one array at each node in a statement's iteration domain. Index expressions are mapping each iteration node $\vec{I} \in S_{cis}$ in the iteration domain to an index node $\vec{A} \in D_A^{\text{var}}$ in the array's variable domain. For data consumption these mappings are defined as *operand mappings* M^{op} and for data production, *definition mapping* M^{def}, De Greef (1998).

For statement *s* and array *A*, this mapping is denoted as,

$$M_{sA} = \left\{ \vec{I} \rightarrow \vec{A} \mid A_1 = \mathrm{F}_{\mathrm{sA1}}^{\mathrm{ie}}(\vec{I}) \wedge A_2 = \mathrm{F}_{\mathrm{sA2}}^{\mathrm{ie}}(\vec{I}) \wedge \cdots \wedge A_m = \mathrm{F}_{\mathrm{sAm}}^{\mathrm{ie}}(\vec{I}) \wedge \vec{A} \in D_A^{\text{var}} \right\} \tag{8}$$

Definition 9: The *definition- and operand domains* of a statement describe which elements of the array are being accessed (read/written) during all possible executions. The *definition domain* D^{def} is the result of applying the definition mapping on the statement's iteration domain and the *operand domain* D^{op} is the result of applying the operand mapping on the statement's iteration domain, De Greef (1998). For statement *s* and array *A*, this domain is denoted as,

$$D_{sA} = M_{sA}(D_s^{iter}) = \left\{ \vec{A} \mid \exists \vec{I} \in D_s^{iter} \ s.t. \ A_1 = \mathrm{F}_{\mathrm{sA1}}^{\mathrm{ie}}(\vec{I}) \wedge A_2 = \mathrm{F}_{\mathrm{sA2}}^{\mathrm{ie}}(\vec{I}) \wedge \cdots \wedge A_m = \mathrm{F}_{\mathrm{sAm}}^{\mathrm{ie}}(\vec{I}) \wedge \vec{A} \in D_A^{\text{var}} \right\} \tag{9}$$

For example, the operand domain of array A in statement S2, Figure 1A equals

$$\begin{aligned} D_{2A}^{op} &= \left\{ (a_1, a_2) \mid \exists (r,c) \in D_2^{iter} \ s.t. \ \mathrm{a}_1 = r - 2 \wedge \mathrm{a}_2 = c - 2 \wedge (a_1, a_2) \in Z^2 \right\} \\ &= \left\{ (a_1, a_2) \mid \exists (r,c) \in Z^2 \ s.t. \ 2 \le r \le 4 \wedge 2 \le c \le 7 \wedge \mathrm{a}_1 = r - 2 \wedge \mathrm{a}_2 = c - 2 \wedge (a_1, a_2) \in Z^2 \right\} \end{aligned} \tag{10}$$

In this example the operand domain D_{2A}^{op} of statement *S2* equals the definition domain D_{1A}^{def} of statement *S1* and is depicted as polytope *A* in Figure 1B.

Definition 10: Let statement *r* be data flow dependent on statement *w* through read- and write accesses to array *A* of variables. Then the relation between the *r* statement's iteration domain and the *w* statement's iteration domain is called *data dependency relation* and is described by a mathematical mapping M_{rwA}^{flow}, Pugh (1991) and De Greef (1998).

$$\begin{aligned} M_{wrA}^{flow} &= \left\{ \vec{I}_w \rightarrow \vec{I}_r \mid \exists \vec{A} \in D_A^{\text{var}} \ s.t. \ M_w^{def}(\vec{I}_w) = \vec{A} = M_r^{op}(I_r) \wedge \vec{I}_r \in D_r^{iter} \wedge \vec{I}_w \in D_w^{iter} \right\} \\ &= \left\{ \begin{array}{l} \vec{I}_w \rightarrow \vec{I}_r \mid A_1 = \mathrm{F}_{\mathrm{wA1}}^{\mathrm{ie}}(\vec{I}_w) \wedge A_2 = \mathrm{F}_{\mathrm{wA2}}^{\mathrm{ie}}(\vec{I}_w) \wedge \cdots \wedge A_m = \mathrm{F}_{\mathrm{wAm}}^{\mathrm{ie}}(\vec{I}_w) \wedge \\ A_1 = \mathrm{F}_{\mathrm{rA1}}^{\mathrm{ie}}(\vec{I}_r) \wedge A_2 = \mathrm{F}_{\mathrm{rA2}}^{\mathrm{ie}}(\vec{I}_r) \wedge \cdots \wedge A_m = \mathrm{F}_{\mathrm{rAm}}^{\mathrm{ie}}(\vec{I}_r) \wedge \vec{I}_w \in D_w^{iter} \wedge \vec{I}_r \in D_r^{iter} \end{array} \right\} \end{aligned} \tag{11}$$

The iteration domain for array *A* write, D_w^{iter} and for array *A* read, D_r^{iter} are simply referred to as source polytope and destination polytope.

For example, the data dependency relation for array *A* from statement *S1* to *S2*, Figure 1A equals

$$M_{12A}^{flow} = \left\{ (r,c) \to (r',c') \mid \exists (a_1,a_2) \in D_A^{\text{var}} \ s.t.\ M_1^{def}(r,c) = (a_1,a_2) = M_2^{op}(r',c') \wedge (r,c) \in D_1^{iter} \wedge (r',c') \in D_2^{iter} \right\}$$
$$= \left\{ (r,c) \to (r',c') \mid r' = r - Vr \wedge c' = c - Vc \wedge 0 \le r \le 2 \wedge 0 \le c \le 5 \wedge 2 \le r' \le 4 \wedge 2 \le c' \le 7 \wedge (r,c) \in Z^2 \wedge (r',c') \in Z^2 \right\} \quad (12)$$

Definition 11: 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*. As defined by Verdoolaege et al. (2003),

$$\vec{V}_{wrA} = \vec{I}_r - \vec{I}_w \text{ , if } \vec{I}_w \text{ depends on } \vec{I}_r \text{ through array A} \quad (13)$$

Definition 12: A *dependency* between two iteration domains carried by array *A*, is defined as the set of data dependency distance vectors going from the iteration domain of statement *w* to the iteration domain of statement *r* and is denoted as,

$$D_{wrA}^{dep} = \left\{ \vec{V} \mid \exists \vec{I} \in D_w^{iter} \ s.t.\ \vec{\mathrm{V}} = M_{wrA}^{flow}(\vec{I}) - \vec{I} \wedge \vec{V} \in S_{cis} \right\} \quad (14)$$

Definition 13: If all dependency vectors $\vec{V}_{wrA}$ such that $\vec{V}_{wrA} \in D_{wrA}^{dep}$ are all equal, the dependency vectors $\vec{V}_{wrA}$ are said to be *uniform*, Verdoolaege et al. (2003).

For example, at the statement *S2* in Figure 1A, array *B* is written and array *A* is read at the same iteration node (r,c). Thus $\vec{I}_2 = (r,c)$. In addition, array *A* is read from an iteration node that was written by statement *S1* at iteration $(r - V_r, c - V_c)$ and thus $\vec{I}_1 = (r - V_r, c - V_c)$. The data flow dependency vector then becomes $\vec{V}_{12A} = \vec{I}_2 - \vec{I}_1 = (V_r, V_c)$. In Figure 1B, a dependency vector $\vec{V}_{12A}$ is then drawn from the source polytope to the destination polytope. There are actually one or more vectors going from every iteration node of the source polytope to the destination polytope. But since all vectors are equal and thus uniform, they are drawn only for one node. This is true for all positions apart from the frame boundaries, with reference to the regular types of RTVPS being considered here. For simplicity, boundary conditions are not considered.

Definition 14: A *dependency vector polytope* DVP_{wrA} is a geometrical model of the data dependencies carried by array *A* from statement *w* to *r*. It is defined as the smallest possible orthogonal set of iteration nodes, but large enough to enclose all dependency distance vectors $\vec{V}_{wrA} \in D_{wrA}^{dep}$,

$$DVP_{wrA} = \left\{ \begin{array}{l} \vec{I} \mid \forall \vec{V} \in D_{wrA}^{dep} \wedge \exists \vec{P} \in S_{cis} \wedge \exists \vec{Q} \in S_{cis} \ s.t.\ \text{minimize} \left| \vec{P} \right| \wedge \vec{P} \ge \vec{V} \wedge \text{minmize} \left| \vec{Q} \right| \wedge \vec{Q} \le \vec{V} \\ \vec{0} \le \vec{I} \le \vec{P} - \vec{Q} \wedge \vec{I} \in S_{cis} \end{array} \right\} \quad (15)$$

Thus, the *dependency vector polytope* is an orthogonal model and consequently a uniform approximation of a set of non-uniform data flow dependencies, Kjeldsberg et al. (2003).

Definition 15: For a loop nest with *s* number of statements, the set of statements equal $S = \{k \mid 1 \le k \le s\}$. A *polytope dependency graph* $G = (V,E)$, is a directed graph with a set *V* of vertices and a set *E* of edges. The vertices equal the set of iteration domains, $V = \left\{ D \mid D = D_w^{iter} \forall w \in S \right\}$. The edges equal the set of dependencies between the iteration domains, $E = \left\{ D \mid D = D_{wr}^{dep} \ \forall \text{ statements r that depends on w} \wedge w \in S \wedge r \in S \right\}$.

For example, Figure 1C depicts the polytope dependency graph drawn from the polytopes and the single dependency shown in Figure 1B.

Section 5.1 explains the polyhedral input model to the memory estimation tool STOREQ. Section 5.2 and 5.3 will explain how this polyhedral model can be extracted from an IMEM model.

5.1 STOREQ input model

There are two main input matrices to STOREQ. One defines a set of statement's iteration domains where each iteration domain is defined according to Definition 4. The other matrix defines a set of dependency vector

polytopes according to Definition 14. Each dependency vector polytope in this matrix is representing the set of non-uniform data dependency distance vectors according to Definition 11.

5.2 Polytopes from an IMEM model

Figure 5 depicts an UML deployment diagram that illustrates how an output stream of an actor can be modeled conceptually. This UML diagram is only shown as a graphical illustration. All video output streams are uniquely defined by an index since one IMEM actor can contain several output streams. The spatial size of a frame in a stream is modeled by two parameters, number of rows *NOR* and number of columns, *NOC.* An IM_LAYER is used to model each dimension in a color space model such as RGB or YUV. This means for example that only one IM_LAYER is used to model a gray scale pixel whereas three are used to model Red, Green and Blue. The index of IM_LAYER provides a unique identification of the layer and the semantics is a string that explains its use. Input streams are similarly modeled in IMEM. This model of an input- or output video stream contains enough information to define a polytope *P* as being an iteration domain in accordance with Definition 4,

$$P = \left\{ \begin{array}{l} \vec{\chi} \mid \vec{\chi}=(\chi_1,\chi_2,\chi_3,\chi_4) \wedge 0 \le \chi_1 < NOF \wedge 0 \le \chi_2 < NOR \wedge 0 \le \chi_3 < NOC \wedge 0 \le \chi_4 < NOL \wedge \\ \vec{\chi} \in Z^4 \wedge NOF \in Z \wedge NOR \in Z \wedge NOC \in Z \wedge NOL \in Z \end{array} \right\} \quad (16)$$

$\vec{\chi}$ is an iteration node within the polytope. The parameter *NOL* means number of layers. Together with *NOR* and *NOC*, it is taken directly from the stream model. *NOF* means the number of frames and is actually infinite for a continuous stream of video. *NOF* is set equally for all polytopes to a value greater then the projection of all modeled pixel neighborhoods in the frame dimension.

All input streams that are produced by an input streamer and all output streams from all actors will result in the definition of a polytope *P* in a common polyhedral iteration space.

5.3 Dependency distance vectors from IMEM

A dependency distance vector describes the relationship between production and consumption of data elements. In image processing, a neighborhood of pixels will describe the input set of pixels from which an output pixel is computed. This is equal to the non-uniform set of dependency distance vectors V, which are drawn in Figure 6 for a simple spatial neighborhood. The regularity of neighborhood-oriented image processing algorithms makes the set of dependency distance vectors to be the same for each pixel position. Since the neighborhood is modeled as a data structure of design entities, this set of dependency distance vectors *V* can be derived from the IMEM model as,

$$V = \left\{ \vec{V} \mid \vec{V} = (frame_i, pixr_i, pixc_i, dNOL) \cup (frame_i, pixr_i, pixc_i, -sNOL) \ s.t.\ i = 1,\cdots,noOfPixels \wedge \vec{V} \in Z^4 \right\} \quad (17)$$

The parameters *frame, pixr* and *pixc* are the relative coordinates of each pixel *i* in a neighborhood of *noOfPixel* number of pixels. See Figure 3A. For each pixel *i* there is one or more dependency distance vectors going from array write, defined in the source polytope, to array read, defined in the destination polytope. *dNOL* is the number of layers in each pixel for the array read in the destination polytope and *sNOL* for the data write in the source polytope. Each layer in a color space model is assumed to be dependent on all others. *dNOL* and *–sNOL* are thus used to model the extreme projections in the layer dimension. The set of dependency distance vectors defined by Equation (17) represents a dependency vector polytope in accordance with Definition 14.

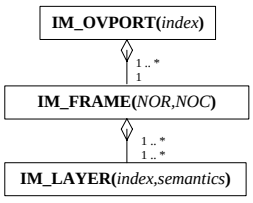

Figure 5. An IMEM model of an output stream.

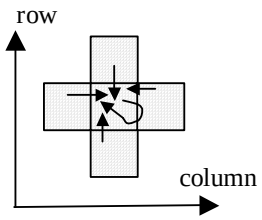

Figure 6. Dependency vectors from a neighborhood.

6. PLACING THE POLYTOPES IN THE COMMON POLYHEDRAL ITERATION SPACE

This section describes our polytope placement algorithm both formally and illustrated by an example. This example may not be the most relevant for image processing applications. However, it is selected in order to clearly describe how the algorithm works in an intuitive way. We have chosen to depict C-code for a common loop nest for different situations during the polytope placement process. This code representation will make this example easier to understand although the C-code representation never exists in our model. A C-coded loop nest is depicted in Figure 7A, again ignoring border conditions. This loop nest implements two spatial image processing operations, *f* and *g*, from which polytopes *B* and *C* are produced. Polytopes *A* and *B* correspond to two input video streams. A polytope dependency graph is depicted in Figure 7C. The polygons *A*, *B*, *C* and *D* correspond to the statements respectively, in which data arrays *A, B ,C* and *D* are produced. The vectors in this graph show how polytopes are data flow dependent on each other. See Definition 15 for a formal definition. Figure 7B depicts the common polyhedral iteration space. The polytopes and dependency vectors in this common iteration space and the corresponding nodes and dependencies in Figure 7C are dashed equally. The steps of the placement algorithm are:

1. Define all polytopes, see section 5.1, and place them with their relative origins aligned with the absolute origin of the common polyhedral iteration space. The absolute origin of the polyhedral iteration space equals the first iteration node being traversed within the common loop nest. The relative origin of a polytope equals the first iteration node being traversed within the polytope.
2. Define all dependency vectors between each data array production and each data array consumption as equal to the relative coordinates of the corresponding pixel neighborhood, see section 5.3.

The initial generation of a common polyhedral iteration space with polytopes and dependency vectors is shown in Figure 7B. This placement is now illegal in the sense that a negative dependency exists. Polytope *B* is dependent on *A[r+2,c+2]*, which makes this vector negative in both the *r*- and *c*-dimension. A negative dependency, which means a dependency vector that has a negative projection in one or more of its dimensions, may express a non-casual system depending on the space to time mapping. That is, data must be produced before it can be consumed. So the next step will be:

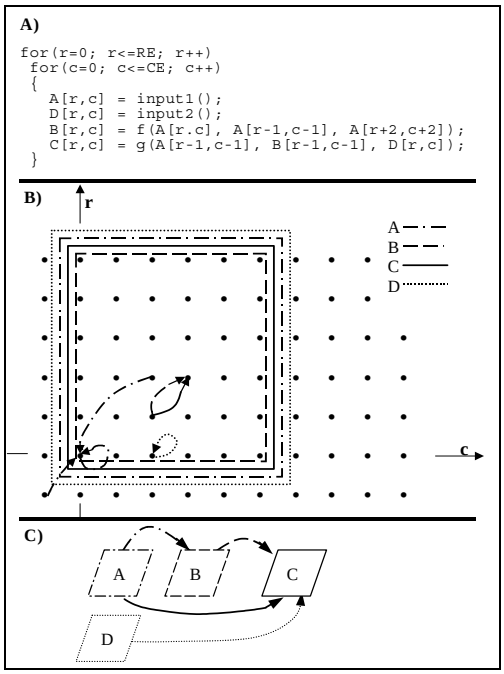

Figure 7. Initial placement of polytopes.

3. Search for a negative dependency vector. If this is not found, proceed to step 7.
4. Move the destination polytope. That is the polytope to which the negative dependency vector goes. This movement must be equal to minus the projection of the dependency vector for each dimension. That means, to let the dependency vector in those dimensions, which have a negative projection, to be zero. However if the dependency vector in any dimension has a positive projection, the movement in that dimension must be zero.
5. Recalculate all dependency vectors that go to the destination polytope just moved in step 4. This must also include dependencies stemming from other polytopes other than the negative vector found at step 3. The movement will be added to all mentioned vectors.
6. Recalculate all dependency vectors that stems from the destination polytope just moved in step 4. The movement must be subtracted from the dependency vectors. Go to step 3.
7. No more negative dependencies now exist.

During the search for negative dependency vectors, one of the dependency vectors from polytope *A* to *B* is found to be negative in Figure 7B. This vector (r,c)=(-2,-2) will cause the algorithm to move polytope *B* +2 in both dimensions. This is depicted in Figure 8B with the corresponding C-code in Figure 8A. The dependency vector going from polytope *A* to polytope *B* will be recalculated as well as the vector going from polytope *B* to *C*. Before the movement of *B*, the dependency vector from *B* to *C* was equal to (r,c)=(1,1) and after the movement of *B*, (r,c)=(-1,-1). Hence the algorithm has now caused another vector to become negative which will be detected at step 3. So polytope *C* will be moved +1 in both dimensions at the next iteration. This is depicted in Figure 9B with the corresponding C-code in Figure 9A. The movement of polytope *C*, which was +1 in both dimensions, is now added to the vectors going from polytope *A* to *C* as well as *B* to *C* and *D* to *C*. Now, no more negative dependency vectors exist, which will be detected at step 3.

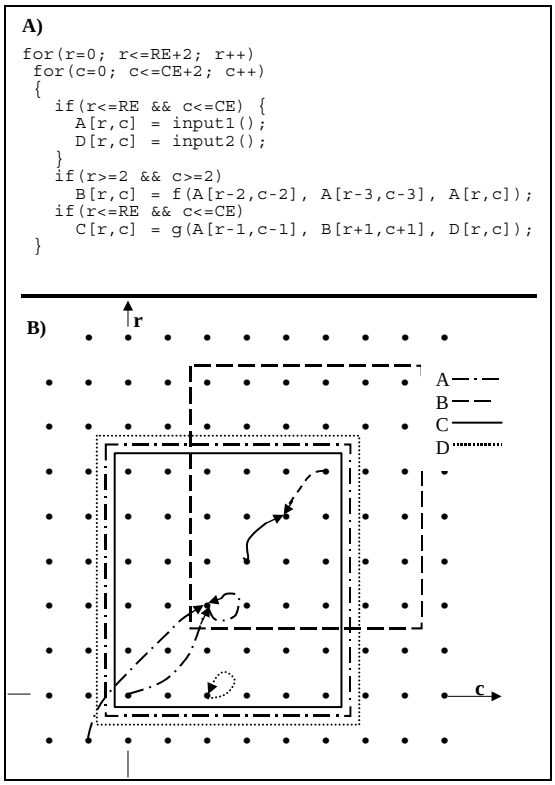

Figure 8. Polytope B has been moved (r,c)=(2,2).

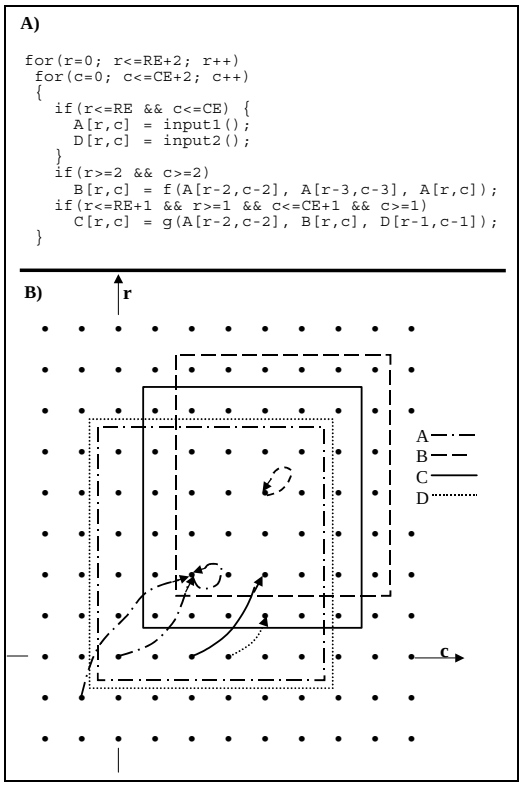

Figure 9. Polytope C has been moved (r,c)=(1,1).

7. A REAL-LIFE APPLICATION

We have selected a machine vision application in order to demonstrate how memory estimation can be used together with IMEM modeling. Figure 10 shows a simplified sketch of an inspection chamber for cellulose fibers. A mixture of water and cellulose fibers flows through this chamber. A monochrome video camera monitors the fibers through a window in the chamber. Figure 11 shows one video frame with cellulose fibers. Typical parameters requiring to be measured include distributions of length, width and curvature of the fibers.

Figure 15A shows a data flow graph of the modeled machine vision system. The first operation performed on the video stream from the camera is pre-filtering, where light shading and dirt on the glass window can be removed, by creating a reference frame through temporal low pass filtering. This reference frame is subtracted from the video stream. An example from this pre-filtering can be seen in Figure 12. The next step is to segment the frames into fibers and background through grayscale thresholding. A near optimal threshold level can be automatically calculated from the histogram of each frame, Otsu (1979). This results in binary frames as depicted in Figure 13. The smallest fragments of fibers are not of interest for the measurements in this application. So the next step will be to remove the smallest black image components by morphological filtering, Gonzales and Woods (1993). This can be seen in Figure 15A as a sequence of three operations, erode, dilate and erode again. The filtered example frame is depicted in Figure 14.

We have chosen not to demonstrate the final steps of the image analysis, which corresponds to the calculation of length, width and curvature distribution. Until the morphological filtering, the data flow graph is multi-rate synchronous. The last steps of the data flow will be asynchronous and can be modeled and refined using SystemC in conjunction with IMEM. There are currently no additional constructs available in IMEM for asynchronous data flow modeling.

Figure 15B depicts a data dependency graph of polytopes. The capitals correspond to the video streams in Figure 15A. The size of each data dependency, as reported by the estimation tool, is depicted in Table 1. The results are based on a frame size of 410 rows by 512 columns. The size for some of the dependencies will differ with execution order. Two situations are represented, frames-rows-columns and frames-columns-rows. The layer dimension is omitted since only one layer exists for a gray-scale pixel. Dependency 2 is recursive as the pre-filtering is done with an infinite impulse response filter, IIR. The estimation results and their use during the design trajectory are discussed in Section 8.2.

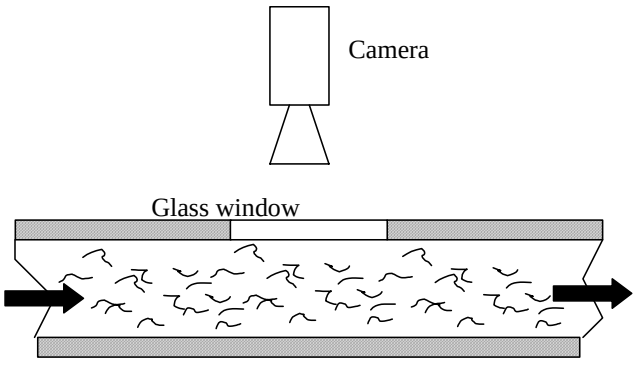

Figure 10. Cellulose inspection chamber.

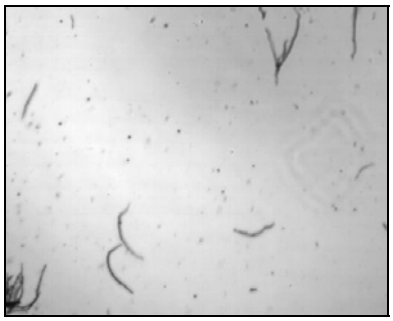

Figure 11. Input frame from camera.

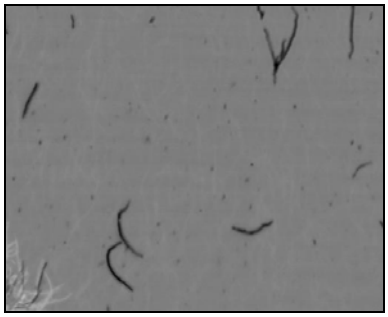

Figure 12. After temporal pre-filtering.

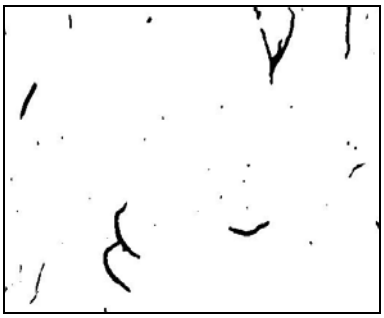

Figure 13. After segmentation.

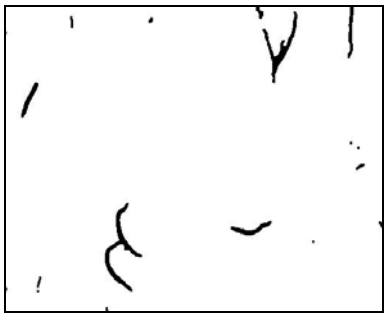

Figure 14. After morphological filtering.

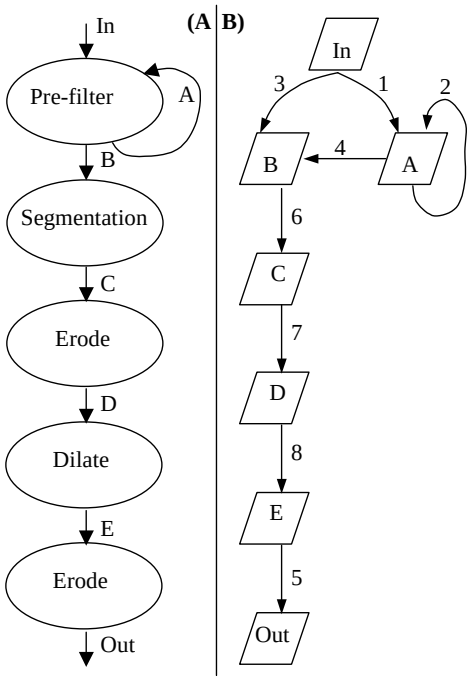

Figure 15. Data flow and dependency graph.

Dependency number	Number of iteration nodes. Exec order = f-r-c	Number of iteration nodes. Exec order = f-c-r
1	0	0
2	419840	419840
3	0	0
4	419840	419840
5	1026	822
6	0	0
7	1026	822
8	2052	1644

Table 1. Result from storage estimation.

8. ANALYSIS

In this section a twofold discussion of the results is presented. Firstly, the polytope placement algorithm developed to work with IMEM is analyzed. Secondly, the use of memory storage estimation will be analyzed with respect to the IMEM workflow.

8.1 The polytope placement algorithm

As the function of the placement algorithm illustrated previously is to remove negative dependencies and optimize memory storage requirement, the minimum memory requirement and the optimal execution ordering required to achieve it may change during the procedure of polytope placement. The effect of the placement algorithm on the memory storage requirement and on the flexibility of execution ordering is thus evaluated here.

The placement algorithm makes negative dependencies legal by moving the destination polytope so that the dependency is no longer negative in any dimension. In Figure 7B one of the dependencies between polytope A and polytope B is negative in both the r and c dimensions. In Figure 8B it has been made positive, or rather of length zero, in both dimensions. The projection of both of the other two dependency vectors between A and B increases, thus increasing their memory storage requirement according to Definition 1. This movement is only one of several ways to make a negative dependency legal. As long as one dimension is positive, the other dimensions can remain negative, if the positive dimension is ordered outermost, as explained below. As depicted in Figure 16, if polytope B is moved one step further out along dimension c for instance, the dependency can remain negative in the r dimension, as long as the execution ordering is fixed so that we first iterate up along the r dimension, before we iterate out along the c dimension. This will result in different, and perhaps smaller, memory storage requirements for each of the three dependencies between A and B. Now, however, we will show that removing all negative dimensions is the optimal solution compared to all other legal placements.

1. The storage requirement of a dependency equals the number of iteration nodes visited in the source polytope before the first iteration node is visited in the destination polytope, Kjeldsberg et al. (2004).
2. In Figure 16, one required array element is produced at the iteration node IN_0 where our methodology places the relative origin of the destination polytope.
3. An optimal execution ordering can be found so that the number of iteration nodes visited in source polytopes before IN_0 is at a minimum, Kjeldsberg et al. (2003).
4. If the relative origin of the destination polytope is placed at any other iteration node i.e. IN_1 while ensuring the legality of the dependency, it is still necessary to visit IN_0 before IN_1.
5. The number of iteration nodes visited in the source polytope before IN_1 is therefore at least one larger than the number visited before IN_0.
6. The storage requirement of the relative origin of the destination polytope placed at IN_0 is consequently smaller than the storage requirement of any other placement. QED.

A placement, which includes one or more negative dimensions, will enforce restrictions on the possible execution ordering. The size penalty of such placements can therefore be large. As illustrated, the placement algorithm is based on a simple heuristic by positioning the destination polytope as close as possible to the source polytope(s) in order to minimize the memory storage requirement while ensuring the legacy of the dependencies at the same time. This is done by following the coarse-grained data flow graph, by initially aligning all input polytopes with the absolute origin, thus the reading of all input streams will have no phase shift.

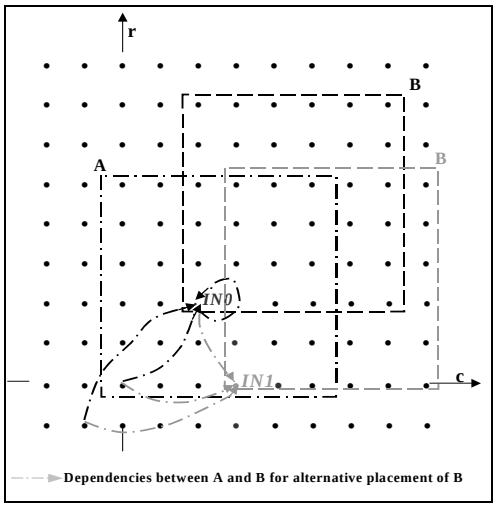

Figure 16. Possible alternative placement for polytope B

However, as shown in Figure 7, it is possible to move the input polytope D as close as possible to polytopes C in order to reduce the memory storage requirement. Moving the polytopes D introduces a buffer requirement, since the input stream will still appear at the input with the same phase with respect to all other input streams. Since the approach of moving input polytopes does not guarantee profit, this placement algorithm assumes that all input polytopes are aligned at the absolut origin. Thus at each iteration, only one destination polytope is considered to be placed as close as possible to the source polytope(s) if any negative dependencies exist. A positive dependency can be unavoidable and occurs for instance as we remove negative dependencies. It will exist when there are several dependencies between a pair of destination and source polytopes (as shown in Figure 9, the dependencies between polytope A and polytope B) or between one destination polytope and several source polytopes (as shown in Figure 9, the dependencies between polytope C and polytopes A and B). However, for applications modeled in IMEM, no positive dependency has been found to be eliminable through a repositioning of polytopes without introducing negative dependencies except with input polytopes(s) as illustrated previously. This results in the current placement algorithm only searching for any negative dependencies and hence repositioning related polytopes.

The result of the placement is that all negative dependencies are removed. At each iteration, we only consider the possible placement of the current destination polytope according to the selected negative dependency. The effect of new negative dependencies introduced by the current polytope placement will not be taken into account until next iteration. This makes the algorithm efficient and is sufficient for the coarse-grained data flow graphs used in the IMEM approach. As a part of future work, we will investigate the effect of simultaneously alive dependencies for more complex sets of dependency graphs within possible video processing applications. By removing all negative dimensions of a dependency, the methodology ensures maximal freedom for the subsequent selection of an optimal execution ordering.

It has been demonstrated that this algorithm is simple compared with previous work but efficient for real-time video-processing applications which contains a limited number of polytopes. The placement algorithm results in optimal or near-optimal memory storage requirement by removing all negative dependencies after placement while ensuring the functionality of the model.

8.2 From model to memory estimation

A real-life example of a machine vision system was presented in order to demonstrate how memory estimation can be used together with IMEM modeling. The response from the estimation tool with respect to the modeled data flow graph is therefore being evaluated here.

The estimation tool STOREQ responds to the size for each inter- or intra- polytope dependency. No further analysis of whether dependencies are alive simultaneously, or if one dependency is equal to, or subset of another is performed. Dependencies 2 and 4 in Figure 15B and Table 1 are equal and should for accuracy be counted only once. This can be explained as the same delay elements of two frames are used to delay both the filter input and the feedback loop in a linear recursive filter. These dependencies are dominant in size and independent of the selected execution order. Their size makes them most likely to be stored in a background memory. Since there is no reuse and small locality in the data accesses, there is no point in introducing a smaller on-chip cache-memory for these dependencies. However, combined spatial and temporal dependencies would require an on-chip cache memory to reduce the external memory bandwidth. It has been demonstrated by Oelmann et.al. (1999) how the size of such a smaller cache-memory can then be considerably reduced by a tiling loop transformation. This is however outside the scope of this paper.

Dependencies 1, 3 and 6 in Figure 15B and Table 1 are all of length zero. This is correct since the size of each of these pixel neighborhoods is one pixel. A simplification has been applied to the segmentation step that corresponds to dependency 6. The histogram calculated for frame *N* is used as the input for the threshold selection of frame *N+1*. This simplification is reasonable since the situation for thresholding is unlikely to change during the time of one frame. Complete accuracy would otherwise require the additional storage of one frame.

The size of dependencies 5 and 7 in Figure 15B and Table 1 are dependent on the selected execution order or space to time mapping. A spatial square neighborhood of 3 by 3 pixels gives a dependency size of 2 rows plus 2

pixels, alternatively 2 columns plus 2 pixels. The same difference in storage size can be observed for dependency 8, which has the spatial neighborhood of 5 by 5 pixels. The spatial size of a frame is 410 rows by 512 columns. The smaller size of dependencies 5, 7 and 8 makes them more likely to be stored in a cache memory close to the data path. Their size can be further reduced with the same tiling transformation as for dependencies 2 and 4, Oelmann (1999).

Dependencies 5, 7 and 8 are for binary frames and will, of course, in a final implementation have lower storage requirements than for a gray-scale frame. The output from the estimation tool is a count of number of simultaneously live variables for each inter- or intra polytope dependency. These variables are of general size and have no assigned bit-width. Hence, to obtain a more accurate estimate of the final storage requirement, bit-width analysis is needed as an additional design refinement step.

The consequence of the performed memory storage estimation is that the processing order or space to time mapping, frame-column-row should be selected for the final implementation. This because the on-chip memory storage for dependencies 5, 7 and 8 will be reduced by 20 percent compared to the non-optimal execution order frame-row-column.

8.3 Motivation of our design procedure

In our method, all polytopes are firstly placed with their relative origins aligned by just doing loop fusion. Then our heuristic-based placement algorithm is used to ensure all dependencies are legal by doing loop fusion and loop shifting. Following is the integration of STOREQ, which functions try different loop interchanges on the whole common iteration space. Loop interchange has been universally used and is categorized as linear loop transformations, Dankaert et al. (2000), Kandemir et al. (1999) and etc. We do not perform any other loop transformations aimed at enhancing the regularity, simply because the RTPS are regular in its nature. Good regularity simplifies the control flow and thus the implementation of the system. In other's approaches, linear loop transformations are done first to improve data locality and regularity globally before placement with loop fusion and loop shifting. This is not necessary in our case. The reason is that the RTPS are perfectly regular in their data accesses and also synchronized. Our placement algorithm, as proven in section 8.1, results in optimal memory storage requirement for each dependency. Doing placement first will not increase the global memory storage requirement with the limited number of polytopes considered, no matter which execution ordering is chosen. Then, STOREQ can take a global estimation of different execution ordering and determines the best one with regards to the memory storage requirement.

9. CONCLUSION

The new polytope placement algorithm presented in this paper, formalizes the method by which an interface and memory model of a real-time video processing system are transformed into a projection in a common polyhedral iteration space. This projection, defined by polytopes and dependency vectors can then be used as input to a memory storage estimation tool STOREQ.

A machine vision application demonstrates how a designer can use this memory estimation feature in order to optimize the storage requirements by means of selecting the optimal processing order or space to time mapping. The on-chip memory storage for the selected application could be reduced by 20 percent compared to a non optimal execution order.

From the results we can conclude that the presented work will form the backend for design transformations such as bit-width refinement, loop transformations and physical memory mapping.

10. ACKNOWLEDGMENTS

The Mid-Sweden University and the KK-foundation are greatly acknowledged for their financial support. Eurocon Analyzer AB is greatly acknowledged for their support of real-life images.

REFERENCES

Adé, M., Lauwereins, R., Peperstraete, J.A., 1999. Buffer Memory Requirements in DSP Applications. Computer Systems Science and Engineering, 14 (3), 155-165.

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

Catthoor, F., Wuytack S., De Greef E., Balasa F., Nachtergaele L., Vandecappelle A., 1998. Custom Memory Management Methodology -- Exploration of Memory Organisation for Embedded Multimedia System Design, Kluwer Academic Publishers, ISBN 0-7923-8288-9

Dace, C.A., 1993. An applicative high-level language for dsp system design. IEE Colloquium on General-Purpose Signal-Processing Devices, 2/1-2/4, (Digest No.085).

Danckaert, K., Catthoor, F., De Man, H., 2000. 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.

Darte, A., Huard, G., 2002. New results on array contraction. Proceedings of the 13th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP), 359-370, San Jose, CA, USA.

De Greef, E., 1998. Storage Size Reduction For Multimedia Applications. Doctoral dissertation at Katholieke Universiteit Leuven.

Fraboulet, A., Huard, G., Mignotte, A., 1999. Loop Alignment for Memory Accesses Optimization. Proceedings of the Twelfth International Symposium on System Synthesis (ISSS'99), 71-77, San Jose, CA, USA.

Gonzales, R.C., Woods, R.E., 1993. Digital Image Processing, Addison Wesley, ISBN 0-201-50803-6.

Grun, P., Balasa, F., Dutt, N., 1998. Memory Size Estimation for Multimedia Applications. Proceedings of ACM/IEEE Workshop on Hardware/Software Co-Design (Codes)}, 145-149, Seattle WA, USA.

Kandemir, M., Ramanujam J., Choudhary A., 1999. Improving cache locality by a combination of loop and data transformations. IEEE Transaction. on Computers 48(1), 159-167

Kjeldsberg, P.G., Catthoor, F., Aas, E.J., 2003. Data Dependency Size Estimation for use in Memory Optimization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 22 (7), 908-921.

Kjeldsberg, P.G., Catthoor, F., Aas, E.J., 2004. Storage Requirement Estimation for Optimized Design of Data Intensive Applications. Accepted for publication in ACM Transactions on Design Automation of Electronic Systems.

Lauwereins, R., Engels, M., Adé M., Peperstraete, J.A., 1995. Grape-II: A System-Level Prototyping Environment for DSP Applications, Computer, 28 (2), 35-43.

Li, Y., Wolf, W.H., 1999. Hardware/Software Co-Synthesis with Memory Hierarchies. IEEE transactions on computer-aided design of integrated circuits and systems, 18 (10), 1405-1417.

Lim, A. W., Liao, S. W., Lam, M. S., 2001. Blocking and array contraction across arbitrarily nested loops using affine partitioning. ACMSIGPLAN notices, 36 (7) , 103-112.

Manjikian, N., Abdelrahman, T. S., 1997. Fusion of Loops for Parallelism and Locality. IEEE Transactions on Parallel and Distributed Systems, 8 (2), 193-209.

Norell, H., Thörnberg, B., O'Nils, M., 2003. Automatic Hardware Synthesis of Spatial Memory Models for Real-Time Image Processing Systems. Proceedings of the 21th Norchip Conference, Riga, Latvia.

Oelmann, B., Norell, H., Andersson, R., Xu, Y., 1999. Design of Real-Time Signal Processing ASIC for Noise Reduction in Moving Video Images. Proceeding of the 17th Norchip Conference, 228-33, Oslo, Norway.

Otsu, N., 1979. A Threshold Selection Method from Gray-Level Histograms. IEEE Transactions on Systems, Man and Cybernetics, 9 (1), 62-66.

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

Pugh, W., 1991. The omega test: a fast and practical integer programming algorithm for dependency analysis. Proceedings of Supercomputing ´91, 4-13

Song, Y., Xu, R., Wang, C., Li, Z., 2001. Data locality enhancement by Memory Reduction. Proceedings of the 15th international ACM conference on supercomputing, 50-64, Sorrento, Italy.

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

Thörnberg, B., O'Nils, M., 2003. Automated implementation of memory models for real-time video processing systems. Proceedings of the 21th Norchip Conference, Riga, Latvia.

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

Wilde, D.K., 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.

Wuytack, S., Diguet, J.Ph., Catthoor, F., De Man, H., 1998. Formalized methodology for data reuse exploration for low-power hierarchical memory mappings. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 6 (4), 529-537.

Ying, Z., Malik, S., 2000. Exact memory size estimation for array computations. Transactions on Very Large Scale (VLSI) Systems, 8 (5), 517-521.

Benny Thörnberg received the Licentiate and B.Sc. EE degrees from Mid Sweden University. He is a lecturer and a Ph.D. student at the department of Information Technology and Media in the same 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. Thörnberg has also been working with camera designs for intra-oral x-ray imaging at Regam Medical Systems AB from 1990 to 1997.

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.

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.

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

Per Gunnar Kjeldsberg 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.