Guidance of Loop Ordering for Reduced Memory Usage in Signal Processing Applications

Per Gunnar Kjeldsberg[1], Francky Catthoor[2], Sven Verdoolaege[3], Martin Palkovic[4], Arnout Vandecappelle[4], Qubo Hu[1] and Einar J. Aas[1]

Abstract

Data dominated signal processing applications are typically described using large and multi-dimensional arrays and loop nests. The order of production and consumption of array elements in these loop nests has huge impact on the amount of memory required during execution. This is essential since the size and complexity of the memory hierarchy is the dominating factor for power, performance and chip size in these applications. This paper presents a number of guiding principles for the ordering of the dimensions in the loop nests. They enable the designer, or design tools, to find the optimal ordering of loop nest dimensions for individual data dependencies in the code. We prove the validity of the guiding principles when no prior restrictions are given regarding fixation of dimensions. If some dimensions are already fixed at given nest levels, this is taken into account when fixing the remaining dimensions. In most cases an optimal ordering is found for this situation as well. The guiding principles can be used in the early design phases in order to enable minimization of the memory requirement through in-place mapping. We use real life examples to show how they can be applied to reach a cost optimized end product. The results show orders of magnitude improvement in memory requirement compared to using the declared array sizes, and similar penalties for choosing the suboptimal ordering of loops when in-place mapping is exploited.

I. Introduction

Many signal processing systems, particularly in the multi-media and telecom domains, are inherently data dominant. For this class of applications, data transfer and storage largely determine cost and performance parameters. This is the case for chip size, since large memories are usually needed, performance, since accessing the memories may very well be the main bottleneck, and power consumption, since the memories and buses consume large quantities of energy. Due to the large memory requirement, both on-chip and off-chip memories are usually needed. The accessing of the off-chip memories in particular should be minimized. Even for systems with caches, the overall storage requirement has vital impact on the performance and power consumption, since it greatly influences the

1: Norwegian University of Science and Technology, Trondheim, Norway. E-mail: pgk@iet.ntnu.no, qubo.hu@gmail.com, ejaas@iet.ntnu.no.

2: IMEC, Leuven, Belgium and also at Katholieke Universiteit Leuven, Belgium. E-mail: catthoor@imec.be.

3: Leiden Institute of Advanced Computer Science, The Netherlands and also at Katholieke Universiteit Leuven, Belgium. E-mail: Sven.Verdoolaege@cs.kuleuven.be.

4: IMEC, Leuven, Belgium. E-mail: palkovic@imec.be, arnout@mind.be.

number of slow and power expensive cache misses. During the system development process, the designer must hence concentrate first on exploring the memory subsystem to achieve a cost optimized end product [1], [2].

For our target classes of data dominant applications the high level description is typically characterized by large multi-dimensional loop nests and arrays with mostly manifest index expressions and loop bounds. An expression is manifest if it does not have data dependent index variables. It is only dependent on loop iterators and constants. Fig. 1 shows a simple example. Three statements, `S1`, `S2` and `S3`, produce elements of three arrays, *A*, *B*, and *C*. Elements from array *A* are consumed when elements of array *B* and *C* are produced. This gives rise to flow dependencies, also called true data dependencies, between statements `S1` and `S2` and between `S1` and `S3` [3], [4]. In this example the array index functions are very simple, but the techniques presented in this paper work for any affine and manifest index functions. For code that is non-affine, this is achievable in most cases by a proper array data flow analysis preprocessing [5], [6]. For non-manifest code, preprocessing and use of scenarios can be used to remove data dependent constructs [7]. Alternatively, worst case values can be used. The preprocessing is also able to distinguish between different types of data dependencies, so that we can focus solely on the flow dependencies that are determining the size of the memory required for the application.

```
for (x=0; x<=5; x++)
 for (y=0; y<=5; y++)
   for (z=0; z<=2; z++) {
S1: A[x][y][z] = f1( ... );
S2: if (x>=1 && y>=2) B[x][y][z] = f2( A[x-1][y-2][z] );
S3: if (x>=1 && y>=1) C[x][y][z] = f3( A[x-1][y][z-1] );
   }
```

Fig. 1. Code example for concept definitions.

Loop transformations are important system level design techniques used to enhance the locality and regularity of data accesses in the application code bringing their production and consumption closer together. This reduces the lifetime of data elements, thereby increasing the possibility of reuse of memory locations. Data with non-overlapping lifetimes can be mapped to the same physical memory locations. This transformation is called in-place mapping [8] and includes both intra-signal reuse of memory locations within one array, and inter-signal in-place between multiple arrays. The overall runtime storage requirement of the application is reduced accordingly. Loop interchange is a transformation with very large impact on the lifetimes of data elements within the loop. With the worst case ordering of the loop nest of Fig. 1, that is y outermost, x second outermost, and z innermost (y, x, z), the storage requirement for the dependency between `S1` and `S2` is 33 memory locations. For the best case ordering, (z, x, y), the number of memory locations required is 6. For this simple example, the absolute numbers are small. The ratio between them is large, however, and this relationship will hold also for large real life examples.

It is far from trivial to find the optimal loop ordering, in particular when we have thousands of data dependencies with possibly conflicting ordering. This is often the case in our target application domain. The optimal ordering for the dependency between `S1` and `S3` in Fig. 1 is for example not (z, x, y). The best case ordering here is the worst case ordering for the other dependency. The globally best solution is somewhere in between, i.e., (x, z, y). The storage requirements for the two dependencies are summarized in Tab. I. The complexity of investigating for example all loop interchange solutions for a single loop nest without any constraints is N!, where N is the number of dimensions in the loop nest. For realistic examples, N can be as large as six even if the number of dimensions in the individual arrays is smaller. Since it is the number of loop dimensions that determines the complexity, a full exploration is very time consuming for applications with many data dependencies. In this paper we present an alternative approach using a set of guiding principles for the ordering of loops in a loop nest. We prove their optimality for practical cases, and show how they can be used by the designer both as stand-alone techniques and as integrated parts of a memory size estimation tool. As loop interchange must typically be combined with other transformations (e.g., loop fusion, loop shifting and loop reverse), use of the guiding principles as part of estimation is very relevant. Exploration speed is then even more crucial to give the designer fast feedback during the early system level design stages. By using our technique, the very time consuming full exploration can be avoided. At this point of the design trajectory, it is also common that parts of the execution ordering are fixed, but not all. One or more loops may for instance be fixed outermost or innermost, while the ordering of the other loops can be freely interchanged. In these cases, our technique finds the optimal ordering of the still unfixed part. For the special situation that a given loop is fixed internally in the nest, some rare cases may occur where the guiding principles do not guarantee to find the optimal solution. We present a simple test that detects these cases. To summarize, we do not claim to find the optimal solution to the problem in its most general form. This would be intractable. Instead we prove that our guiding principles are optimal for some very important classes of the problem, which occur most of the time in practical contexts.

The loop transformations we are guiding improve temporal locality, bringing data production and consumption closer in time. The temporal nature makes the optimization platform independent, which is very important at the early design phases.

	ordering	`S1-S2`	`S1-S3`
a)	(x, y, z)	18	13
b)	(y, x, z)	33	3
c)	(x, z, y)	14	18
d)	(y, z, x)	31	6
e)	(z, x, y)	6	36
f)	(z, y, x)	11	31

TABLE I

DEPENDENCY SIZE IN FIG. 1 WITH ALTERNATIVE EXECUTION ORDERINGS

The rest of this paper is organized as follows. We start with an introduction to previous work in this field and to the geometrical model we use for our technique. This is followed by the section with the main contribution of this paper, the presentation and proof of our guiding principles for loop ordering. We then show how these guiding principles are used in our memory size estimation technique. Towards the end, representative application demonstrators are used to illustrate the feasibility and usefulness of the methodology. Finally, we present our conclusions.

II. Previous Work

Loop transformations have been extensively used to optimize application code to given architectures. In particular, the vector machine, systolic array, and parallel compiler communities have used this for a long time, e.g., [4], [9], [10], [11], [12]. Their main goal is, however, to reveal and exploit code and data parallelism, so that multiple instantiations of (parts of) a loop nest can be executed simultaneously. Our goal is to minimize the memory space required to store data during execution by improving data locality and regularity. In [13], data locality is optimized with loop transformations. A cost model for spatial and temporal reuse of cache lines is used to find a desirable loop organization in a way with similarities to ours. The goal is again different from ours, but gives evidence of the generality of the approach.

The loop transformations can successfully be used as enablers for memory size optimization through in-place mapping. In [14], [15], the authors describe transformation techniques with this objective. Our memory size estimates can be used to steer their transformations. [16], [17], [18], [19] are examples of work concerned with actual final in-place mapping. See [19] for a good overview of the work in this field. They require a fully fixed execution ordering, while our work focuses on finding the optimal fixation to enable in-place mapping. Much research has also gone into estimation and optimization of size, performance and power consumption of embedded memory hierarchies in cache and scratch pad based systems, e.g., [20], [21], [22], [23]. Further details here are outside the scope of this paper.

Since use of our guiding principles is so important in our memory size estimation technique, and because we describe this technique in later sections, we will now also present other work in this field. The main part of all previous work on storage requirement estimation has been scalar-based [24], [25], [26], [27], [28]. The number of scalars, also called signals or variables, is then limited, and if arrays are treated, they are flattened and each array element is considered a separate scalar. Common to all scalar based techniques is that they break down when used for large multi-dimensional arrays. The problem is NP-hard and the solution complexity grows exponentially with the number of scalars. When the multi-dimensional arrays present in the applications of our target domain are flattened, they result in many thousands or even millions of scalars.

To overcome the shortcomings of the scalar-based techniques, several research teams have tried to split the arrays into suitable units before or as a part of the estimation. Typically, each instance of array element accessing in the code (e.g., a statement reading or writing data) is treated separately. Due to the loop structure of the code, large parts of an array can be produced or consumed by the same code instance. This reduces the number of elements the estimator must handle compared to the scalar approach. [29], [30], [31], [32] present different approaches for size estimation given a fully fixed execution ordering. In contrast to this, the storage requirement estimation technique presented in [33] does not take execution ordering into account at all. It traverses a dependency graph based on an extended data dependency analysis resulting in a potentially very large number of non-overlapping array sections (so called basic sets) and the dependencies between them. The maximal combined size of simultaneously alive basic sets found through a greedy graph traversal gives the storage requirement. [34] brings the work in [33] back to the fully fixed domain. Using advanced algebraic techniques they find the exact size of each of the basic sets. To avoid overly complex address generation, the final implementation will most likely not be able to use the splitting of arrays into basic sets. Their results are then not realistic but can instead be seen as very useful lower bounds.

All the techniques above estimate the memory size assuming a single memory. [35] performs hierarchical memory size estimation, taking data reuse and memory hierarchy allocation into account. In-place mapping is not incorporated in the current version, but is indicated as part of future work.

In summary, all of the previous work on storage requirement estimation requires a fully fixed execution ordering to be determined prior to the estimation. The only exception is [33], which allows any ordering not prohibited by data dependencies. None of the approaches permit the designer to specify partial ordering constraints. The dominance of work on fully fixed orderings is not surprising, since this is easier to solve and because in present industrial design, the design entry usually includes a full fixation of the execution ordering. When the abstraction level of the design entry is raised, as motivated in numerous papers [36], it is necessary to start from a non-fully fixed ordering. We have previously presented an estimation technique which allows any degree of fixed ordering (unfixed, partially fixed, or fully fixed) [37], [38]. This makes it useful during the early system design stages as motivated in the previous section. Our previous work refers to the guiding principles but does not prove their optimality as we do here. We now also extend the discussion of the guiding principles to include partially fixed ordering with dimensions fixed internally, not only outermost or innermost.

III. The Geometrical Model

For the target class of data intensive applications, data, operations, dependencies, and the order in which they are handled, are very important. It is therefore vital to be able to model this accurately. It has been shown (e.g., by [39]) that a geometrical modeling of groups of data and groups of operations is a good solution. Several geometrical modeling approaches exist (e.g., Pressburger formulas, polyhedra, LBLs). In this work we use a somewhat simpler approach which is more appropriate for our purpose. This section gives a brief introduction to this geometrical model with an emphasis on giving an intuitive understanding of the main concepts. More details can be found in [37]. In our discussion we use the terms loop dimensions and loop iterators. A dimension should be understood as a geometrical concept, while an iterator is related to the actual variable over which the dimension iterates.

Let us return to the example code of Fig. 1 and look at the dependency between statements `S1` and `S2` (ignoring the other dependency for the time being). The loops around the statements define an *iteration space*, [3], as shown in Fig. 2. Each node within this space represents one execution of the statements inside the loop nest and can be identified by its loop iterator vector $\mathbf{i} = [i_1, i_2, ...i_n]$, where i_k is the value of the kth iterator in the loop nest, counting from outermost to innermost. For $\mathbf{i}$ we use the iterator sequence given in the original application code. Note, however, that since the execution ordering is not fixed, this sequence does not necessarily reflect the order in which the iterators will eventually be traversed. As long as no data dependencies are violated, i.e., data is never consumed before it is produced, any ordering can be used. For a given loop nest with n dimensions, $D = (i_k)$ contains the ordered set of iterators in the loop nest. For two iterators in D we have that i_l is fixed inside i_m iff $\mathrm{p}os(i_l) > \mathrm{p}os(i_m)$. Here $\mathrm{p}os(i_k)$ is a function that returns the position of i_k in D. In Fig. 1, we have $\mathbf{i} = [x, y, z]$ based on the order of the iterators in the application code. If this ordering is used during execution, we get $D = (x, y, z)$. $\mathrm{p}os(x)$ is then 1. If iterators x and z are interchanged, we would get $D = (z, y, x)$ and $\mathrm{p}os(x) = 3$.

For our example in Fig. 1, at each of the *iteration nodes* one *A*-array element is produced. When the if-clause condition of statement `S2` is true, one *A*-array element is consumed and one *B*-array element is produced. These nodes also constitute the *iteration domains* (IDs) of the two statements, as indicated by the rectangles $ID_{\mathtt{S1}}$ and $ID_{\mathtt{S2}}$ in Fig. 2. To avoid unnecessarily complex figures, two-dimensional rectangles are used. All nodes within a rectangle are part of the corresponding ID. We say that $ID_{\mathtt{S1}}$ and $ID_{\mathtt{S2}}$ are the production and consumption IDs, respectively. In this simple example, the IDs are rectangular. This is, however, not necessarily the case. We then adopt a so called bounding box approach. A bounding box is a rectangular approximation of the ID and it is specified by the lower and upper bounds in each dimension. This is an efficient and appropriate approximation since most shapes in our domain are of rectangular nature, images, blocks, etc. For other shapes, e.g., a triangle, the bounding box can increase the number of nodes inside the ID. In order to avoid too complex address generation, techniques with similar influence on the memory size are however also used in most real life implementations of applications in our domain (e.g., the Memory Compaction tool from the Atomium tool set [16], [40]). The actual memory space allocated for a triangle would for example be very close to the corresponding bounding rectangle. Their use therefore reflects the actual memory size requirement. Using the bounding boxes we can define the ID for a statement X as

$$ID_X = \{\, \mathbf{i} \mid \mathbf{l}_X \le \mathbf{i} \le \mathbf{u}_X \,\}, \tag{1}$$

where the comparison is applied componentwise and the kth elements of $\mathbf{l}_X$ and $\mathbf{u}_X$ are the minimum and maximum values in dimension k, respectively. Fig. 2 also shows the resulting IDs in our running example using our bounding box notation. For each dimension, ordered from top to bottom according to $\mathbf{i}$, the lower and upper bounds are listed.

A *Dependency Vector* (DV) is the (vector) difference of an iteration node in the consumption ID, ID_C, reading from an array element, and the iteration node in the production ID, ID_P, producing the same array element. This is also known as a dependency distance vector [12]. The DV is depicted as an arrow from the production iteration

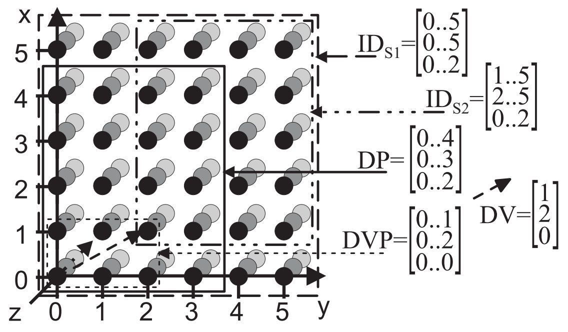
x
y
z
0
1
2
3
4
5
$ID_{S1}=\begin{bmatrix}0..5\\0..5\\0..2\end{bmatrix}$
$ID_{S2}=\begin{bmatrix}1..5\\2..5\\0..2\end{bmatrix}$
$DP=\begin{bmatrix}0..4\\0..3\\0..2\end{bmatrix}$
$DV=\begin{bmatrix}1\\2\\0\end{bmatrix}$
$DVP=\begin{bmatrix}0..1\\0..2\\0..0\end{bmatrix}$

Fig. 2. Iteration space and iteration domains of example in Fig. 1.

node to the consumption iteration node. If a dependency between two statements has multiple DVs with different length and direction, we combine these into (normally) one single extreme DV, having the maximal length, v_k, of any DV in each dimension [38]. This is a simple approach to uniformization. The overestimates of the memory requirements this may incur can again be accepted since similar techniques are used for real implementations. It is also very fast, making it suitable for estimation purposes. More advanced techniques for uniformization can alternatively be used [41].

In general not all elements produced by one statement are read by a given depending statement. A *Dependency Part* (DP) is therefore defined containing the iteration nodes at which elements are produced that are read by the depending statement. The DP for a dependency between the corresponding ID_P and ID_C is hence given by

$$DP = \{\, \mathbf{i} \mid \max(\mathbf{l}_C - \mathbf{v}, \mathbf{l}_P) \leq \mathbf{i} \leq \min(\mathbf{u}_C - \mathbf{v}, \mathbf{u}_P) \,\}. \tag{2}$$

The (extreme) DV, $\mathbf{v}$, spans a *Dependency Vector Polytope* (DVP) defined by

$$DVP = \{\, \mathbf{i} \mid \mathbf{l}_P \leq \mathbf{i} \leq \min(\mathbf{l}_{DP} + \mathbf{v}, \mathbf{u}_{DP} + \mathbf{1}) \,\}, \tag{3}$$

where $\mathbf{l}_{DP}$ and $\mathbf{u}_{DP}$ are the minimum and maximum vertex for the DP respectively. Fig. 2 illustrates the DV and DVP for the dependency between `S1` and `S2` in Fig. 1. When there is no overlap between ID_P and ID_C in one or more dimension, the DVP is extended with one outside the DP in these dimensions. This is done to enable a general use of the DVP when calculating dependency sizes as shown in Section IV-A.

In situations were multiple DVs for a given dependency are in opposite directions (e.g., one goes from 0 to N and the other from N to 0 in dimension x), the DVP can be generated from multiple extreme DVs. The resulting DVP is spanned by all the extreme DVs. Examples of this can be found in the cavity detection application in Section VI-A. The non-zero dimensions of the DVP are defined as *Spanning Dimensions* (SD). Since the set of SD normally only comprises a subset of the iterator space dimensions, the remaining dimensions are denoted *Nonspanning Dimensions* (ND). For the DVP in Fig. 2, *x* and *y* are SDs while *z* is ND. A list of these and other important abbreviations can be found at the end of the paper.

As mentioned in Section I, the ability to exploit in-place mapping is essential to minimize the storage requirement of an application. A data element is alive from its production and until it is consumed. The memory location at which it was saved is then reused by the new data element being produced (or at least freed for other use). Our definition of dependency size is tightly linked to this.

Definition 1: The size of a dependency between two iteration domains is the (maximum) number of live data elements in the dependency. ■

Since we assume uniform dependencies and rectangular domains, the dependency size is equal to the number of iteration nodes visited in the DP before the first depending iteration node is visited. Section IV-A describes how this size can be calculated mathematically. For now, let us see what sizes this definition gives for the dependency depicted in Fig. 2. The first depending iteration node is $\mathbf{i} = [1, 2, 0]$. If we assume $D = (x, y, z)$, we will first traverse the z dimension, then the y dimension, and finally the x dimension. We would then visit 18 iteration nodes inside the DP before $\mathbf{i} = [1, 2, 0]$ is visited. At each of these iteration nodes an A-array element is produced that has to be saved. When $\mathbf{i} = [1, 2, 0]$ is reached, memory locations can start being reused by the B-array, and the size of the dependency between `S1` and `S2` will therefore not increase further and is hence equal to 18. Tab. I shows the dependency size for all ordering alternatives. Note in particular that the smallest dependency size is achieved with ordering $D = (z, x, y)$.

IV. Guiding Principles

We will now present a number of guiding principles for the ordering of loops in a loop nest. The goal is to minimize the storage requirement of a single dependency. Sections V and VI show how this is used to estimate the overall runtime memory size for a given application. A short introduction to the main principles has been presented by us in [37]. In the following subsections we will give proof of their optimality for the cases that appear most in real applications. We also extend the principles to include a discussion and proof for the situation where we have a partially fixed ordering including internally fixed dimensions.

The size of a dependency is minimized if the execution ordering is fixed so that its NDs and SDs are placed at the outermost and innermost nest levels respectively. The order of the NDs is of no consequence as long as they are all fixed outermost. If one or more SDs are ordered in between the NDs, it is however better to place the shortest NDs inside the SDs. The length of ND i_k is determined by the length of the DP in this dimension $|DP_k|$. The ordering of SDs is important even if all of them are fixed innermost. The SD with the largest Length Ratio (LR) should be ordered innermost. The LR is defined as follows:

$$LR_k = \frac{|DVP_k| - 1}{|DP_k| - 1}.$$

For the special case when $|DP_k| = 1$, care must be taken to avoid division by zero. An $|LR_k| = \infty$ can still be assumed, since such dimensions should be ordered innermost.

Returning to the dependency between `S1` and `S2` in Fig. 1, the calculation of the LRs for the *x* and *y* dimensions are illustrated in Fig. 3. According to the guiding principles, the *y* dimension should be fixed innermost giving a dependency size of six as shown to be optimal in Tab. I.

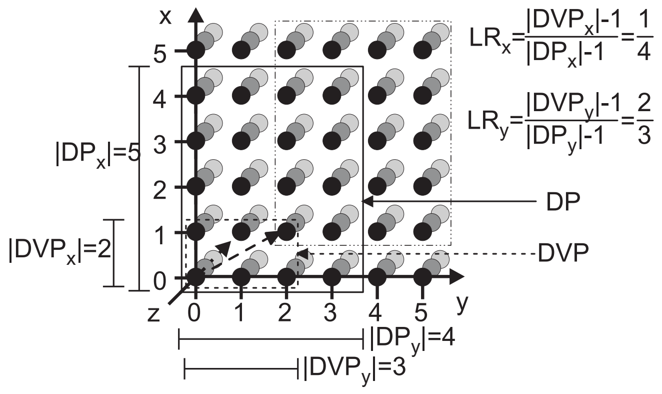
x
y
z
$LR_x=\frac{|DVP_x|-1}{|DP_x|-1}=\frac{1}{4}$
$LR_y=\frac{|DVP_y|-1}{|DP_y|-1}=\frac{2}{3}$
DP
DVP
$|DP_x|=5$
$|DVP_x|=2$
$|DP_y|=4$
$|DVP_y|=3$

Fig. 3. Calculation of LRs for the code example of Fig. 1.

The guiding principles can be used directly in the cases where one or a few major dependencies determine the total storage requirement of a loop nest. Furthermore, dimensions that are NDs for all dependencies should be fixed outermost to optimize the in-place mapping opportunity and minimize the memory requirement. For more general cases with multiple and possibly conflicting optimal orderings, an automated tool is needed. This was illustrated in Sec I and will be further discussed in Section V.

Before presenting the proofs of the guiding principles, we derive formulas for calculating the size of a dependency, given a certain loop ordering.

A. Dependency size

The guiding principles determine the loop ordering which minimizes the size of a single dependency. This size is given by the formulas derived in this section. A geometrical model is assumed with the approximations as presented in Section III.

A single dependency involves only two statements, one producing and one consuming array elements. We can shift the loops around the production and consumption statement so that the production loops start at 0. Assuming a dependency given as a DP and DVP as defined in Section III, we therefore get a general code template as shown in Fig. 4. The iteration nodes at which array elements are produced have an upper bound of $P_k = |DP_k|$ for each dimension. Similarly, the iteration nodes at which array elements are consumed have a lower bound of $Q_k = |DVP_k| - 1$ and an upper bound of $P_k + Q_k$ for each dimension. SDs have Q values larger than zero while NDs have Q values equal to zero. For the dependency between `S1` and `S2` in the code example of Fig. 1, we would have $n = 3, i_1 = x, i_2 = y, i_3 = z, P_1 = 5, P_2 = 4, P_3 = 3, Q_1 = 1, Q_2 = 2$, and $Q_3 = 0$ in Fig. 4.

The flow dependency information allows us to determine, for a certain consumption iteration reading a value, which was the production iteration that wrote that value. The actual place in the memory where this element is stored is not relevant. Therefore, the code used for our proofs represents the flow dependency information by mapping the production iterations one-to-one on the array indices where the value is stored. This results in the index expressions as given in Fig. 4.

```
for (i1=0; i1<P1+Q1; i1++)
 for (i2=0;  i2<P2+Q2; i2++)
  ...
   for (in=0;  in<Pn+Qn; in++) {
     ...
     if (i1<P1 && i2<P2 &&...&& in<Pn)
       A[i1][i2]...[in] = ...;
     ...
     if (i1>=Q1 && i2>=Q2 &&...&& in>=Qn)
       ... = f( A[i1-Q1][i2-Q2]...[in-Qn] );
   }
```

Fig. 4. Code example with n dimensions.$P_k = |DP_k|$ and $Q_k = |DVP_k| - 1$.

Note that the actual application code can have a much more complex form than Fig. 4. After having transformed the dependency information in the code to DPs and DVPs, this simplified code structure is however sufficient for our argumentation. Its DP and DVP are identical to the DP and DVP generated from the actual application code.

Given a certain ordering of the loops $D = (i_k)$ (see Section III), Equation 4 counts the number of iteration nodes I visited within the DP before a specific node $\mathbf{i} = [i_1, i_2, ...i_n]$ of the DP is visited. The equation sums the node contribution of each iterator, starting outermost.

$$I(\mathbf{i}) = \sum_{j=1}^{n} \left(i_j \cdot \prod_{k \in F_j} P_k \right), \tag{4}$$

where

$$F_j = \{ i_k \in D \mid \mathrm{p}os(i_k) > \mathrm{p}os(i_j) \}. \tag{5}$$

For a given iterator i_j, F_j is the set of iterators that is fixed inside iterator i_j in the loop nest. Assuming a procedural execution of Fig. 1, $F_x = \{y, z\}, F_y = \{z\}, F_z = \emptyset$. Consequently, for each iterator i_j, we multiply its iteration value with the DP-length of all dimensions fixed inside it. This gives the iterator's contribution to the total number of visited nodes within the DP. This total number is equal to the number of elements produced up to the moment when node $\mathbf{i} = [i_1, i_2, ...i_n]$ is visited.

Eq. 4 assumes that $\mathbf{i}$ lies in the production iteration domain, i.e. $\mathbf{i} \leq \mathbf{P}$. If this is not the case, the outermost dimension without overlap will force all iteration nodes of the dimensions in its F_j to be visited. The complete number of nodes visited is hence found, and the summation should be terminated.

The required size of array A varies across iterations. Because of our use of extreme DVs when generating the DVP, the consumption considered here is the last consumption of that value. Elements can therefore be discarded as soon as they are consumed. The size is consequently equal to the number of additional elements produced between the iteration that produced the currently consumed element, $\mathbf{i}_p = [i_1 - Q_1, i_2 - Q_2, ...i_n - Q_n]$, and the current iteration, $\mathbf{i}_c = [i_1, i_2, ...i_n]$. The size grows during the first iterations of a loop, because elements are produced without any elements being consumed. Then it stays steady for a while, and diminishes again at the end of the loop since elements are being consumed but no new ones are produced. The maximum size is therefore certainly reached for $\mathbf{i}_c = \mathbf{Q}$ and $\mathbf{i}_p = \mathbf{0}$. This corresponds to Definition 1 and the dependency size is thus given by Eq. 6. Since $i_j = 0$ for all i_j in $I(\mathbf{0})$, the size is equal to the number of nodes visited within the DP before **Q**, i.e., $I(\mathbf{Q})$.

$$S = I(\mathbf{Q}) - I(\mathbf{0}) = \sum_{j=1}^{n} \left(Q_j \cdot \prod_{k \in F_j} P_k \right) \tag{6}$$

If there is no overlap between ID_P and ID_C in one or more dimensions, we have the situation where the consuming **i** lies outside the production iteration domain. As soon as a dimension without overlap is reached while performing the summation in Eq. 6, the complete size of the dependency is found. We will come back to the consequences of this below.

B. Ordering of Nonspanning Dimensions Outermost

The following proposition uses equation 6 to show that all NDs should be placed outermost.

Proposition 1: The dependency size is minimized if the SDs are placed innermost.

Proof: The dependency size is given by Eq. 6. The number of products within each sum is determined by the number of elements in the corresponding F_j. $Q_j = 0$ for NDs and $Q_j > 0$ for SDs. It is consequently only SDs that will result in sum terms. The overall sum is minimized if the F_j of the SDs have as few elements as possible. This is achieved by placing the SDs inside all NDs. ■

There must be overlap between the two IDs for all NDs. If there is no overlap in a dimension, it has to be an SD since array elements produced at a given iterator value must be consumed at another value. The conclusion of Prop. 1 will therefore hold even in this situation.

C. Ordering Among Spanning Dimensions

In the previous section, it was proven that SDs should be ordered innermost. There may however be more than one SD, and the internal ordering among them is very important for the size of the dependency. In the following, a derivation and proof of the guiding principle based on LR will be given using the general code of Fig. 4. As was shown in the previous section, the NDs will not influence the dependency size if they are placed outermost. They can therefore be ignored without loss of generality, and it is now assumed that there are n SDs in the dependency. Eq. 6 still gives the dependency size.

Starting with any ordering of SDs, it is possible to reach any alternative ordering by moving dimensions outwards, starting with the dimension that will be outermost in the target ordering. The movement can be performed through a stepwise swapping of neighbors similar to the technique used in the bubble sort algorithm [42]. We will now use this to prove that an ordering of dimensions according to increasing LR is optimal with respect to dependency size.

Proposition 2: The dependency size is minimized if the SDs are ordered with increasing LR from the outermost to the innermost SD.

Proof: We first consider the effect on the relative dependency size of interchanging the positions of two adjacent dimensions. In the formula for the dependence size (6), such an interchange only has an effect on the terms corresponding to the two interchanged dimensions, since for all other dimensions the set of inner dimensions is unaffected by this interchange. From the two affected terms, we may further factor out the factors corresponding to the dimensions that are placed inside both interchanged dimensions, since this (non-negative) factor does not influence the relative dependency size. Without loss of generality, we may therefore assume that the two interchanged dimensions are the only dimensions. Eq. 6 is then reduced to two terms, $Q_{outerdim} \cdot P_{innerdim} + Q_{innerdim}$, where $outerdim$ and $innerdim$ are the outermost and innermost of the two dimensions, respectively.

The original order of these two dimensions should be kept if the dependency size before interchange is smaller than (or equal to) the dependency size after interchange, i.e.,

$$Q_1 \cdot P_2 + Q_2 \leq Q_1 + Q_2 \cdot P_1 \tag{7}$$

or

$$\frac{Q_1}{P_1 - 1} \leq \frac{Q_2}{P_2 - 1}.$$

Substituting $P_k = |DP_k|$ and $Q_k = |DVP_k| - 1$ yields

$$\frac{|DVP_1| - 1}{|DP_1| - 1} \leq \frac{|DVP_2| - 1}{|DP_2| - 1}$$

or

$$LR_1 \leq LR_2.$$

That is, the original ordering should be kept if it has dimensions ordered according to increasing LR.

Now consider any ordering of the dimensions. The dependency size can be improved (or kept constant) by successively interchanging adjacent dimensions where, prior to the interchange, the outer dimension has a larger LR than the inner dimension. This “bubble sort” [42] continues until all dimensions are sorted according to increasing LR with a dependency size that is better than or equal to the initial size. Since the initial ordering was arbitrary, the sorted ordering has the optimal dependency size. This completes the proof. ■

If dimensions without overlap exist, they will have an LR larger than one. All dimensions with overlap will have LR smaller or equal to one. Consequently, a dimension without overlap should be placed innermost. If there is more than one dimension without overlap, their internal ordering is of no consequence, since all iteration nodes of all of them will be visited as soon as one of these dimensions is encountered. The conclusion of Prop. 2 will therefore hold even in this situation.

D. Partially Fixed Ordering

The proofs given in the previous sections assume that the designer has full freedom to choose the ordering. This is often not the case, and guiding principles are needed for the remaining dimensions when the execution ordering is partially fixed.

Proposition 3: With dimensions fixed outermost, the dependency size is minimized if the remaining NDs are ordered outermost among the remaining dimensions.

Proof: According to Eq. 6, the contribution of a dimension only depends on the dimensions that are fixed inside it, not the ordering of these dimensions. The contribution of a dimension fixed outermost is therefore not influenced by the ordering of the dimensions inside it. For the contribution of the remaining dimensions, see Prop. 1. ■

Proposition 4: With dimensions fixed innermost, the dependency size is minimized if the remaining NDs are ordered outermost among the remaining dimensions.

Proof: According to Eq. 6, the contribution of a dimension does not depend on the dimensions fixed outside it. The contribution of a dimension fixed innermost is therefore not influenced by the dimensions outside it. For the contribution of the remaining dimensions, see Prop. 1. ■

Having proved use of the guiding principles for NDs when dimensions are fixed outermost or innermost, we will now do the same for SDs. In this case, the proof is the same for both.

Proposition 5: With dimensions fixed outermost and/or innermost the dependency size is minimized if the remaining SDs are ordered with increasing LR from the remaining outermost to the remaining innermost SD.

Proof: The dimensions fixed outermost and/or innermost are among the dimensions factored out in Prop. 2. Prop. 2 can hence be used for the ordering of the remaining SDs. ■

If one or more dimensions are fixed but neither innermost nor outermost, the situation becomes more complex. The fixation of one dimension at one given interior location influences the optimal ordering among other dimensions. Ordering of the remaining dimensions in accordance with their LR is not necessarily optimal. Such a situation would for example occur if we in Fig. 4 have n=3, using dimensions *x*, *y*, and *z*, and P_x=5, P_y=14, P_z=32, Q_x=1, Q_y=3, Q_z=1. Since $P_k = |DP_k|$ and $Q_k = |DVP_k| - 1$, the LRs of the three SDs are $LR_x = 0.25$, $LR_y = 0.23$, and $LR_z = 0.03$. Accordingly the optimal ordering is $D = (z, y, x)$. As Tab. II shows, this is indeed the ordering with the smallest dependency size (row f)). What is more interesting for the current discussion, is that when z is fixed innermost (rows a and b) or outermost (rows e and f), the optimal internal ordering among dimensions x and y is in accordance with the guiding principles, with y outside x (rows b and f). When z is fixed internally however (rows c and d), the optimal ordering among dimensions x and y changes. Now the smallest dependency size is obtained with dimension x outside y (row c).

In most cases, the LR can still be used for the ordering, even when some dimensions are fixed internally. Proposition 6 discusses the requirements that need to be fulfilled for the guiding principles to be used. To obtain a truly optimal ordering in the rare occasions that they can not be used, it is necessary to perform a full exploration

	Ordering	Dependency size, S
a)	(x, y, z)	$Q_x \cdot P_y \cdot P_z + Q_y \cdot P_z + Q_z = 1 \cdot 14 \cdot 32 + 3 \cdot 32 + 1 = 545$
b)	(y, x, z)	$Q_y \cdot P_x \cdot P_z + Q_x \cdot P_z + Q_z = 3 \cdot 5 \cdot 32 + 1 \cdot 32 + 1 = 513$
c)	(x, z, y)	$Q_x \cdot P_z \cdot P_y + Q_z \cdot P_y + Q_y = 1 \cdot 32 \cdot 14 + 1 \cdot 14 + 3 = 465$
d)	(y, z, x)	$Q_y \cdot P_z \cdot P_x + Q_z \cdot P_x + Q_x = 3 \cdot 32 \cdot 5 + 1 \cdot 5 + 1 = 486$
e)	(z, x, y)	$Q_z \cdot P_x \cdot P_y + Q_x \cdot P_y + Q_y = 1 \cdot 5 \cdot 14 + 1 \cdot 14 + 3 = 87$
f)	(z, y, x)	$Q_z \cdot P_y \cdot P_x + Q_y \cdot P_x + Q_x = 1 \cdot 14 \cdot 5 + 3 \cdot 5 + 1 = 86$

TABLE II

STORAGE REQUIREMENT FOR CODE EXAMPLE OF FIG. 4 WITH N=3 AND DEFINED P_k AND Q_k

of the ordering of the undetermined dimensions. These will normally be very few, so the complexity will not be intractable even in these cases.

Proposition 6: With dimensions fixed internally, the guiding principles can be used to minimize the dependency size under the following assumption regarding the resulting ordering. For each pair of dimensions s and t placed on different sides of internally fixed dimensions, we have

$$|DP_s| \geq |DP_t| \quad \text{and} \quad |DVP_s| \leq |DVP_t|.$$

Proof: From Proposition 2 we know that inside each block of dimensions without any fixed dimensions, the dimensions should be ordered according to increasing LR. It remains to show that we cannot improve the dependency size by interchanging two dimensions from different blocks. As in the proof of Proposition 2 we may assume that the dimensions under consideration are the outermost and innermost dimensions. Without loss of generality, we may further assume that $s < t$ and that the remaining dimensions are numbered from $s + 1$ to $t - 1$, i.e., $pos(i_k) = k$ for all k, $s = 1$, $D = (i_1, i_2, i_3, ..., i_t)$, and dimensions i_2 through i_{t-1} are fixed.

When s and t are interchanged, the terms in the formula for the dependency size (6) change as follows: the factor of Q_t changes from 1 to the product of the P_k of all other dimensions and vice versa for Q_s, while for the remaining dimensions, the factor P_t is replaced by P_s. The order of these two dimensions should be kept if the change in dependency size is non-negative, i.e., if

$$Q_s \left(1 - \prod_{s<k\leq t} P_k\right) + Q_t \left(\prod_{s\leq k<t} P_k - 1\right) + \sum_{s<j<t} Q_j (P_s - P_t) \prod_{j<k<t} P_k \geq 0.$$

The last term is positive if $P_s - P_t \geq 0$, i.e., $|DP_s| \geq |DP_t|$. This corresponds to the first part of the proposition. It only remains to show that

$$Q_s(1 - P_t G) + Q_t(P_s G - 1) \geq 0, \tag{8}$$

where

$$G = \prod_{s<k<t} P_k.$$

Since $LR_s \leq LR_t$, we have (7)

$$Q_t - Q_s \leq Q_t P_s - Q_s P_t.$$

With $|DVP_s| \leq |DVP_t|$ (the second part of the proposition), we further get $Q_t - Q_s \geq 0$ Then also $Q_t P_s - Q_s P_t \geq 0$ and since $G \geq 1$ we have

$$(Q_t P_s - Q_s P_t)G \geq Q_t P_s - Q_s P_t \geq Q_t - Q_s,$$

which is equivalent to (8). This shows that if $|DP_s| \geq |DP_t|$ and $|DVP_s| \leq |DVP_t|$ then the dependency size is not decreased by interchanging s and t, which completes the proof. ■

Proposition 7: With SDs fixed internally at a position where the optimal ordering fixes NDs both on the outside and inside, the dependency size is minimized if the NDs with the largest $|DP_k|$ are ordered outside the internally fixed dimensions.

Proof: According to Eq. 6 the size contribution of a dimension does not depend on the dimensions fixed outside it. The P_k of the NDs outside the internally fixed dimension does hence not influence the size of the dependency. According to Eq. 6 the size contribution of a dimension does depend on the P_k of dimensions fixed inside it. The smaller the P_k of the NDs inside the internally fixed dimension, the smaller the size of the dependency will be. ■

E. Worst Case Ordering

In many situations, for example memory size estimation, it is interesting to also know the worst case ordering. All propositions above can be directly used with opposite reasoning. To maximize the dependency size, it is for instance necessary to place all NDs innermost, and to order SDs with decreasing LR from the outermost to the innermost SD.

V. Memory Size Estimation

High level memory size estimation is very important during the system level design stages. This gives the designer feedback on the consequences of transformations performed. In particular, upper and lower bounds on the memory size are useful since a given transformation normally does not influence all parts of an application's execution ordering. The estimates should therefore focus on the impact of this transformation on the upper and lower size bounds. The span between them reflects the remaining available execution freedom.

Our estimation methodology is outlined in Fig. 5. The first step is performed by the Atomium tool [43], based on the IDs of the statements and the index functions in the application code. The iteration domains are placed in a common iteration space to enable a global estimation scope even for applications with loops that are not perfectly nested. In [38], we give an example of a common iteration space generation. The third step performs size estimation for individual dependencies while the last step finds the overall memory requirement of the application. Some more details regarding these steps are given in the next sections.

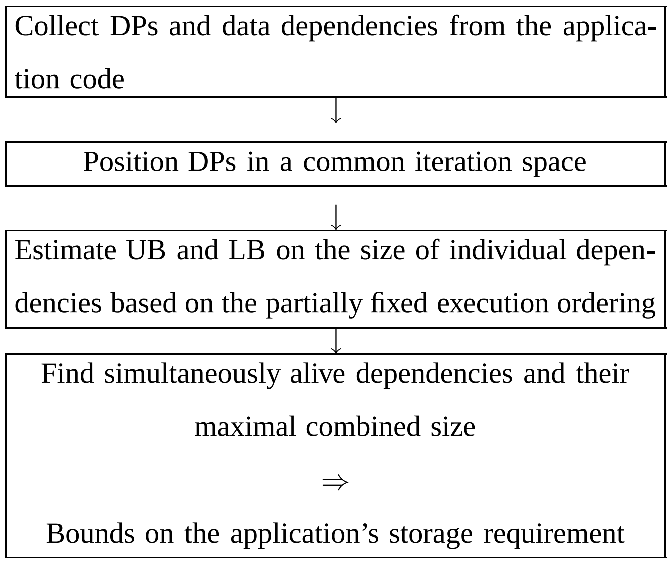
Collect DPs and data dependencies from the application code
Position DPs in a common iteration space
Estimate UB and LB on the size of individual dependencies based on the partially fixed execution ordering
Find simultaneously alive dependencies and their maximal combined size
⇒
Bounds on the application's storage requirement

Fig. 5. Overall storage requirement estimation strategy.

A. Estimation of Individual Dependencies

The size of a dependency is given by Eq. 6. This assumes a fully fixed ordering, which is typically not available at the early design stages. By applying the guiding principles for best case and worst case ordering introduced in Section IV, it is however possible to find the orderings that will result in the lower and upper bounds. Any ordering already fixed will be taken into account. Fig. 6 shows the main principles of the estimation algorithm. A complete description can be found in [37], which also includes a technique for estimation when we are only given a partial order among some of the dimensions. The constraint may for instance be that one dimension is not fixed outermost among (a subset of) the dimensions in the iteration space. This can for example be specified to prohibit a negative dependency to be fixed outermost among the SDs since this will result in data consumption before production (which is not valid).

Let us take a quick look at how this algorithm would perform estimates on the dependency between `S1` and `S2` in the example code from Fig. 1. Assuming a fully unfixed ordering, the SDs x and y would be ordered (x, y) for the LB calculations, since $LR_y > LR_x$. For the UB calculation, they would be ordered (y, x). Since the example only has one ND, its ordering is trivial. It is now checked if the requirements of Prop. 6 are fulfilled so that the LRs can be used to find the optimal ordering. It is only for very rare cases with internally fixed dimensions that this can fail. A somewhat more advanced estimation algorithm must then be used, which is outside the scope of this paper. In this case there is no problem, so the actual size estimation starts with the outermost dimension. Since this dimension is not fixed, the ND z is used for the LB calculation. It does not contribute to the dependency size, so $LB_{tmp} = 0$. For the UB calculation, the SD ordered outermost in the worst case set is selected, that is y. No dimensions are fixed outside it, so all the other dimensions are used to calculate its contribution. This gives $UB_{tmp} = Q_y \cdot P_x \cdot P_z = 2 \cdot 5 \cdot 3 = 30$. The size estimation now continues with the second outermost dimension. For the LB calculation, x is now selected from the ordered set of SDs. Since z is already fixed outside x, it is not

```
define sets of fixed and unfixed SDs and NDs needed for calculation of upper and lower bounds
order sets of unfixed SDs according to their $LR$
order sets of unfixed NDs according to their $|DP|$
check if it is OK to use guiding principles; if not return
for each dimension, currdim, starting outermost
  if currdim is fixed
    use currdim and dimensions not outside it to calculate its contribution to LB and UB
  else
    use best case dimension from ordered sets and dimensions not outside it to calculate contribution to LB
    use worst case dimension from ordered sets and dimensions not outside it to calculate contribution to UB
return LB, UB, and the best case ordering found
```

Fig. 6. Overview of algorithm for size estimation of individual dependencies

included in the calculation of the contribution of x. Hence, we get $LB_{tmp} = LB_{tmp} + Q_x \cdot P_y = 0 + 1 \cdot 4 = 4$. For the UB calculation, x is selected, giving in a similar fashion $UB_{tmp} = UB_{tmp} + Q_x \cdot P_z = 30 + 1 \cdot 3 = 33$. Finally, the size contribution of the innermost dimension is calculated. For the LB, the contribution of y without any dimension fixed inside it gives $LB = LB_{tmp} + Q_y = 4 + 2 = 6$. For the UB, ND z is used, which does not contribute, giving $UB = UB_{tmp} = 33$. In addition to returning these two bounds, the algorithm also returns the best case ordering $D = (z, x, y)$.

In case one dimension is already fixed, for example x outermost, both sets of ordered unfixed dimensions become trivial. Both LB and UB calculations start by calculating the contribution of x, giving $LB_{tmp} = UB_{tmp} = Q_x \cdot P_y \cdot P_z = 1 \cdot 4 \cdot 3 = 12$. For the second outermost dimension we get $LB_{tmp} = LB_{tmp}$ since the noncontributing ND z will be used. We also get $UB_{tmp} = UB_{tmp} + Q_y \cdot P_z = 12 + 2 \cdot 3 = 18$. Finally, for the innermost dimension we get $LB = LB_{tmp} + Q_y = 12 + 2 = 14$ and $UB = UB_{tmp} = 18$. In this case, the best case ordering $D = (x, z, y)$ is returned, taking the partial ordering into account. Note that as the ordering is gradually fixed, the bounds converge. With a fully fixed ordering they will be equal.

This simplified algorithm overview does not include termination if the ID_P and ID_C are not overlapping. A check of this is performed individually for the UB and LB contribution at each iteration of the for loop. As soon as a dimension without overlap is used for one of them, the calculation of this bound is terminated. The calculation of the other bound has to continue until a dimension without overlap is reached for this ordering as well.

An automated CAD tool, called STOREQ, has been developed for this part of the estimation methodology, size estimation of individual dependencies. Details regarding the tool can be found in [38].

B. *Simultaneously Alive Dependencies*

After having found the upper and lower bounds on the size of each dependency in the common iteration space, it is necessary to determine if two or more of them are alive simultaneously. The combined size of simultaneously alive dependencies gives the current storage requirement at any point during the execution of the application.

Definition 2: The maximal combined size of simultaneously alive dependencies over the lifetime of an application, gives the total storage requirement of the application. ■

Two dependencies can potentially be alive simultaneously if their DPs overlap in one or more dimensions in the common iteration space. Depending on whether the overlap occurs only for NDs, for a subset of the dimensions including at least one SD, or for all dimensions, they will alternate in being alive, be alive simultaneously for certain execution orderings, or be alive simultaneously regardless of the chosen execution ordering. A similar reasoning can be used for groups of multiple dependencies. The estimation methodology uses a two-step procedure. First, groups of potentially simultaneously alive dependencies are detected, followed by an inspection to reveal those actually simultaneously alive for a given partially fixed execution ordering. When multiple dependencies cover the same array elements, this is taken into account during the calculation of the combined size of the dependencies found to be alive simultaneously. Details regarding this part of the methodology are presented in [44]. It will not be elaborated further in this paper.

Current work is integrating the two last steps in the overall estimation methodology, Fig. 5, to find the globally best ordering.

VI. Estimation on Real Life Application Demonstrators

In this section the usefulness of the guiding principles and estimation methodology is demonstrated on several real-life applications from the multi-media and wireless domains. The STOREQ prototype CAD tool has been used to reach the results reported.

A. *Cavity Detection*

The estimation methodology has been applied to a cavity detection algorithm used for detection of cavity perimeters in medical imaging. We have previously reported results for the same application in [38].

1) Code description and external constraints: Figure 7 shows an extract from the code. A pseudo t dimension is inserted to generate a common iteration space. As a starting point, the pseudo t dimension is defined so that statements placed in separate loop nests in the original code are executed sequentially. A data flow graph for major parts of the code is given in Fig. 8. The complete code used to produce this DFG can be found in [38]. Each node depicts the ID of a statement. The edges represent dependencies between the IDs. The dependencies are enumerated to ease the following discussion. The vertical lines divide the IDs into groups that are executed at different iterator values of the t dimension. The t dimension opens opportunities for global transformations such as loop merging. The storage requirement of the transformed code with alternative placements and orderings can thus be compared

```
for (t=0; t<=5; t++)
 for (x=0; x<=N-1; x++)
  for (y=0; y<=M-1; y++)
   for (k=0; k<=NB-1; k++) {
    if (t==0 && x>=1 && x<=N-2 && y>=1 && y<=M-2 && k<=2)
S1:  gauss_x_compute[x][y][k+1] = f1(gauss_x_compute[x][y][k],
                                          image_in[x+k-1][y]);
    if (t==0 && k==2 && x>=1 && x<=N-2 && y>=1 && y<=M-2)
S2:  gauss_x_image[x][y] = f2(gauss_x_compute[x][y][TOT]);

    if (t==1 && x>=1 && x<=N-2 && y>=1 && y<=M-2 && k<=2)
S3:  gauss_xy_compute[x][y][k+1] = f3(gauss_xy_compute[x][y][k],
                                    gauss_x_image[x][y+k-1]);
    ...
   }
```

Fig. 7. Extract ($\sim 30\%$) of the code for Cavity Detection example. N=480, M=640, NB=8

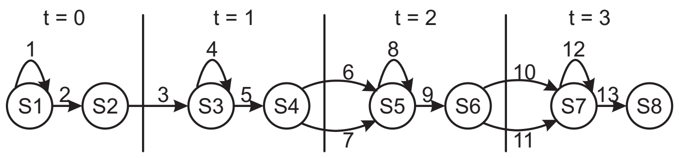

Fig. 8. Data-flow graph for Cavity Detection example.

to the storage requirement of the original ordering. In this example, the focus is on parts with major impact on the storage requirement, so boundary conditions are ignored. It is realistically assumed that the input image can be presented at the input of the application in any order.

The STOREQ CAD tool has been used extensively during the exploration of the cavity detection algorithm. As allowed by the tool, some of the dependencies in the code are non-uniform. Multiple extreme DVs are then used for their description. Statement `S3` has for example a non-uniform accessing of array gauss_x_image. Depending on the *k* value, an array element is accessed that was produced at an iteration point with a *y* value smaller, equal, or larger than the current *y* value. This gives rise to two extreme DVs, one in each direction along the *y* dimension and with different lengths in the *k* dimension. Furthermore, a number of negative dependencies exist in the code. Tab. III summarizes the negative dimensions, the SDs, and the NDs for each DV of the dependencies. This information is automatically dumped from the STOREQ tool. The dependency enumeration is in accordance with Fig. 8.

2) Array Size Estimation and Guidance of Ordering: It will now be shown how the designer can use the guiding principles and feedback from size estimation to rearrange the common loop nest and to determine an execution ordering that gives an optimized storage requirement for the application. The first run of the STOREQ estimation

Dep.	DV1	DV2	DV3	DV4
1	$\underline{t}$ $\underline{x}$ $\underline{y}$ k			
2	$\underline{t}$ $\underline{x}$ $\underline{y}$ $\underline{k}$			
3	t $\underline{x}$ y -k	t $\underline{x}$ -y $\underline{k}$		
4	$\underline{t}$ $\underline{x}$ $\underline{y}$ k			
5	$\underline{t}$ $\underline{x}$ $\underline{y}$ $\underline{k}$			
6	t $\underline{x}$ $\underline{y}$ -k	t $\underline{x}$ $\underline{y}$ k		
7	t -x -y -k	t -x y $\underline{k}$	t x -y k	t x y k
8	$\underline{t}$ $\underline{x}$ $\underline{y}$ k			
9	$\underline{t}$ $\underline{x}$ $\underline{y}$ $\underline{k}$			
10	t $\underline{x}$ $\underline{y}$ -k			
11	t -x -y -k	t -x y -k	t x -y -k	t x y $\underline{k}$
12	$\underline{t}$ $\underline{x}$ $\underline{y}$ k			
13	$\underline{t}$ $\underline{x}$ $\underline{y}$ $\underline{k}$			

TABLE III

SDs, NDs (UNDERLINED), AND NEGATIVE DIMENSIONS (INDICATED WITH -) FOR EACH DV OF THE DEPENDENCIES IN THE CAVITY DETECTION CODE.

tool without any ordering fixed shows that all dependencies that do not cross a vertical *t*-line in Figure 8, have the *k* dimension placed innermost in their optimal ordering. This is due to the fact that all other dimensions are NDs for these dependencies. They should hence be fixed outside the *k* dimension according to the guiding principles of Section IV.

For dependencies with *t* as an SD, that is all dependencies crossing a *t*-line in Fig. 8, the upper and lower bounds converge at the previous upper bound if *t* is fixed outermost. This is again in accordance with the guiding principles which indicate a large penalty on placing SDs outermost. Due to the loop splitting caused by the *t* dimension, none of these dependencies is alive simultaneously if *t* is fixed outermost. Their individual sizes hence determine the overall storage requirement. Since the *t* dimension is an artificial dimension used for generation of a common loop nest, it must always be fixed outermost. To reduce the storage requirement, the common iteration space must therefore be rearranged so that the dependencies do not have *t* as an SD (do not cross a *t*-line). The removal of the *t* dimension from the set of SDs for dependencies corresponds to a loop merging. This may cause dependencies to be alive simultaneously, so that their combined sizes determine the global storage requirement.

We will exemplify this repositioning using statement `S3`. The goal is to make the *t* dimension an ND for dependency 3. This corresponds to a loop merging between the first and second loop nest in the original code. Dependency 3 has two extreme DVs, as seen in Tab. III. In DV1 the *k* dimension is negative. This DV has two other SDs, *t* and *y*. It is hence possible to fulfill the requirement of having another SD outside a negative dimension, even if *t* is transformed into an ND. The requirement is fulfilled if *y* is placed outside *k*. For DV2, the *t* dimension is the only SD in addition to the negative *y* dimension. For a repositioning of `S3` to be valid, it must also result in a non-negative *y* dimension in dependency 3. A form of *skewing* [45] can be used so that the array elements of statement `S3` are produced one iteration node further out along the *y* axis of the common iteration space compared to their original production. When using the guiding principles to find the optimal ordering for dependency 3 we must now take into account the fixed partial ordering of *y* outside *k*. Running STOREQ reveals that the lower bound of dependency 3 is increased from 1 to 2 (achievable with ordering $D = (t, x, y, k)$) while the upper bound is reduced from 304964 to 956 (for example achievable with ordering $D = (t, y, x, k)$).

The next dependencies with *t* as SD are dependencies 6 and 7. As can be seen from Tab. III, DV1 of dependency 6 has only one SD apart from *t*, and this dimension, *k*, is negative. The *k* dimension must hence be made non-negative if *t* is to be made an ND. The situation is more complicated for dependency 7. It has four DVs, all with different combinations of SDs, NDs and negative dimensions. For DV1, *x*, *y*, and *k* are all negative. It is hence not possible to perform the loop merging and transform *t* into an ND without at the same time repositioning statement `S5` in such a way that either all negative SDs are removed or so that at least one of the SDs is made positive. Using the guiding principles, we find that the size of dependency 7 is minimized if *k* is fixed innermost, *x* second innermost, and *y* outermost (except for *t*). The *y* dimensions should hence be made positive and the resulting size estimate for this dependency is 1435. As shown in the previous paragraph, the ordering enforced by this loop merging and skewing has negative consequences for dependency 3, however. It now requires a memory size equal to its upper bound of 956. If instead *x* is made positive and fixed outermost, dependency 3 requires 2 memory locations, while dependency 7 requires 1915. Dependencies 10 and 11 have the same structure as 6 and 7. Their sizes for different orderings and placements are hence also the same. Furthermore, because of the loop merging, all dependencies are alive simultaneously, and it is their combined size that gives the overall size. The memory elements carried by dependency 6 (10) are, however, a subset of those carried by dependency 7 (11) so only the size of 7 and 11 needs to be included.

An alternative to making *x* or *y* positive for dependency 7 and 11, is to make all their SDs non-negative. Tab. IV summarizes the STOREQ estimation results for dependencies 3, 7, and 11. Each row shows the storage requirement for alternative valid placements and execution orderings. The storage requirement of the original code without loop merging is shown for comparison in the first row. Row e) holds the globally best solution. It has a storage requirement over two orders of magnitude lower than the original solution, and substantially lower than the other alternatives.

If the same experiment is performed using a different image format, for instance the panoramic format of the *Advanced Photo System*, the conclusion turns out to be quite different. The previously best solution is now second to worst as shown in Fig. 9. The storage required for buffering full image lines along the *y* dimension for dependency 7 and 11 are now larger than that of buffering full image lines along the *x* dimension for dependency 3. These somewhat surprising results demonstrate how important the storage requirement estimation tool is for the

		Dep. 3	Dep. 7	Dep. 11	Combined
a)	Original code, t outermost	304964	304964	304964	304964
	t=0 and t=1 merged, y outside k	2/956	304964	304964	304964
b)	y positive, (t, y, x, k)	956	1435	1435	3826
c)	x positive, (t, x, y, k)	2	1915	1915	3832
d)	No neg. dim., (t, y, x, k)	956	959	959	2874
e)	No neg. dim., (t, x, y, k)	2	1279	1279	2560

TABLE IV

ESTIMATED DEPENDENCY SIZES RESULTING FROM ALTERNATIVE TRANSFORMATIONS (N = 480, M = 640). FOR ROW B) THROUGH E), A FULL LOOP MERGE IS PERFORMED. UB = LB WHERE ONLY ONE NUMBER IS REPORTED.

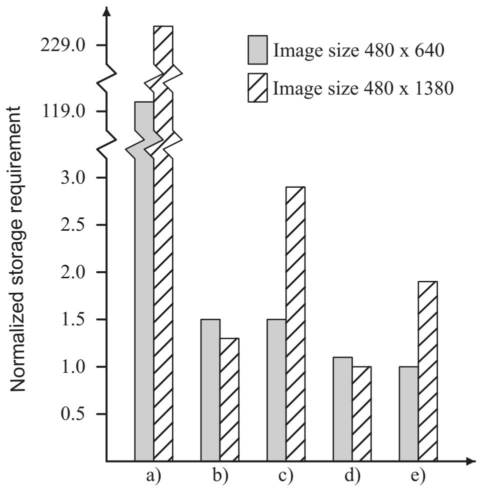

Fig. 9. Combined storage requirement for alternative cavity detection implementations. Normalized to best solution for each image size. Transformations as in Tab. IV.

optimization of the memory usage. It also illustrates the importance of taking the actual values of the parameters into account instead of only considering them as symbols during the optimization exploration. This is different from what happens in many state-of-the-art optimization techniques.

B. Updating Singular Value Decomposition Algorithm

We will now look at the *Updating Singular Value Decomposition* (USVD) algorithm [46] used, for instance, in beamforming for antenna systems. We have previously reported results for the same application in [37]. We show how the guiding principles and the results from the memory size estimation can be used to steer interactive or tool driven global loop reorganization, [47].

1) Code description and external constraints: Fig. 10 shows the two major arrays, *R* and *V*, and the important loop nests and statements for their production and consumption. Two smaller arrays, *phi* and *theta*, used during the production of the *R* and *V* arrays, are also shown. Due to data dependencies not explicitly shown in Fig. 10, the *k* dimension must be placed outermost during the production of the *R*-array.

```
   for (k=0; k<=n-2; k++){
S1: theta[k] = f1( R[...][...][2*k] );
S2: phi[k] = f2( R[...][...][2*k], theta[k] ) ;
    for (i=0; i<=n-1; i++)
     for (j=0; j<=n-1; j++)
      ...
S3:   else R[i][j][2*k+1] = f3( R[i][j][2*k], theta[k] );

    for (i=0;i<=n-1;i++)
     for (j= 0;j<=n-1;j++) {
      ...
S4:   else R[i][j][2*k+2] = f4( R[i][j][2*k+1], phi[k] );
      ...
S5:   else V[i][j][k+1] = f5( V[i][j][k], phi[k] );
    } }
```

Fig. 10. USVD algorithm, diagonalization loop nest

	S1-S3	S2-S5	S3-S4	S4-S3	S5-S5
a)	1/9	1/9	1/100	1/100	1/100
b)	1/1	1/1	100/100	100/100	100/100
c)	1/1	9/9	100/100	100/100	1/10
d)	1/1	9/9	100/100	100/100	10/10

TABLE V

LB/UB FOR DEPENDENCIES IN THE USVD ALGORITHM WITH A) NO EXECUTION ORDERING FIXED, B) *k* FIXED OUTERMOST WITHOUT LOOP BODY SPLIT, C) *k* FIXED OUTERMOST IN *R*-LOOP NEST AND *k* NOT FIXED OUTERMOST IN *V*-LOOP NEST (WITH LOOP BODY SPLIT), D) FULLY FIXED ORDERING FOR BOTH LOOP BODIES.

2) Array Size Estimation and Guidance of Ordering : Tab. V a) shows estimation results for a number of dependencies in the code with no execution ordering fixed (n=10). The *k* dimension is the only SD for all the dependencies, and according to the guiding principles, this dimension should therefore be fixed at the innermost nest level to minimize the storage requirement. As mentioned above, this is not valid, however, since it must be placed outermost for production of the *R*-array. Tab. V b) gives the estimation results for this partial ordering. The upper and lower bounds of the three largest dependencies converge at their previous upper bound of 100. An investigation of the global storage requirement reveals that the maximal combined size occurs while dependencies `S1-S3`, `S2-S5`, `S4-S3`, and `S5-S5` are alive simultaneously. This gives rise to a storage requirement of 202 memory locations, as shown in Fig. 11.

The large increase in the dependency sizes enforced by the fixation of the *k* dimension at the outermost nest level, would encourage the designer to take a closer look at the code. It might very well be that not all dependencies need to have *k* outermost. They could then benefit from a loop body split which would enable a different, more optimal, ordering for these dependencies. The outermost fixation of the *k* dimension is in our example not required for the production of the *V*-array, as long as the necessary phi-values are available. A comparison of the resulting storage requirement for an alternative loop organization, where a loop body split places the *V*-array in a separate loop nest, is therefore needed. Tab. V c) shows estimation results after this reorganization and with the partial ordering that *k* is not to be fixed outermost in the new *V*-array nest. Finally Tab. V d) shows the estimation results for a fully fixed valid ordering with loop body split.

Due to the loop body split, dependencies `S4-S3` and `S5-S5` are not alive simultaneously anymore. The global storage requirement is hence approximately halved to 110 memory locations as shown in Fig. 11. The *V*-array is placed alongside the R-array, since they are not alive simultaneously. The guiding principles and data dependency size estimation thus guides the designer through the early steps of the design trajectory towards a solution with low storage requirements. Compared to the solution using the declared arrays directly, the required memory size is reduced by 95%.

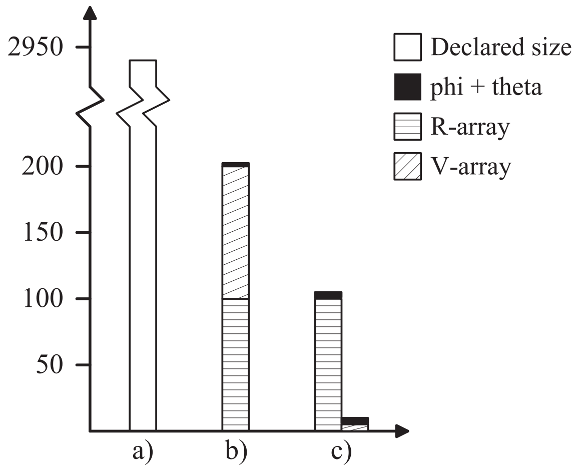

Fig. 11. Estimated storage requirement for the USVD algorithm (n=10): a) Declared size, b) without loop body split, c) with loop body split.

C. Image Processing Kernels

We have experimented with a number of other drivers selected from our application domain. The results are summarized in Tab. VI. We compare our estimates with results from the K2 tool developed at the University of Illinois at Chicago. K2 computes exactly the minimum data memory size required by the application [34]. The procedural execution ordering of the original application code is used. STOREQ in general gives estimates that are bigger than the calculated values found by K2. This is not due to errors in STOREQ, but rather to different approaches. K2 calculates the absolute minimum number of memory locations needed to store the application’s

Application (parameters)	Declared	Proced. exec.		After loop tr.	
		K2	STOREQ	DTSE	STOREQ
Regularity detection (MaxGrid=8, L=64)	4,752	2,304	4,153	21	21 / 129
Durbin algorithm (N=500)	252,499	1,249	2,002	1,502	2,002
2D Gauss blur filter (N=800,M=600)	5,260,027	480,604	958,809	1,809	1,809 / 1,915,008
MPEG-4 motion estimation	265,633	2,465	3,399	2,374	2,373 / 3,396

TABLE VI

ESTIMATION RESULTS AND COMPARISON WITH OTHER APPROACHES. STOREQ RESULTS AFTER LOOP TRANSFORMATIONS ARE GIVEN AS LOWER AND UPPER BOUNDS.

data. An implementation using this minimum number could require a very complex memory address generation unit, however. STOREQ instead tries to estimate the storage requirement of a likely implementation.

We have also performed merging of loops in the different applications. The rightmost column of Tab. VI lists the estimated upper and lower bounds of the optimized storage requirement assuming a fully unfixed execution ordering. As described for the two previous examples, the guiding principles can in each case be used by the designer to decide how to perform the necessary transformations in an optimized way. They can also be used to find the optimal ordering for the final solutions. For individual dependencies, the optimal ordering is also part of the output from the STOREQ tool. In following paragraphs we will compare these early system level estimates with likely implementations.

When estimating the size after loop transformations, we do not necessarily include the complete input and output arrays, as K2 does. We assume either streaming input and output or a multi-dimensional memory hierarchy where we focus on optimizing the memory closest to the data path. We then only include the input memory size required for efficient reuse of data that are read multiple times. The loop ordering has a big influence on the on-chip memory size requirement resulting from this data reuse exploration [48]. This is an example of an additional design step where our guiding principles and high level size estimates are crucial for efficient trade off between data reuse and in-place optimization.

In the two rightmost columns of Tab. VI we compare our early system level estimates with results obtained after performing a complete design using the DTSE methodology and tools. In this case our estimated lower bounds should be compared with the final DTSE results. The estimates are sufficiently accurate for our early exploration purposes. The overestimate in the Durbin algorithm is caused by a bounding box around array elements produced along a two-dimensional diagonal line in the iteration space. This is a situation that in some rare cases can happen in our application domain. The bounding box of the corresponding DP equals the rectangle bounded by the diagonal resulting in an overestimate of the size of the dependency compared to real implementations. We are currently working on techniques to handle this effect. Instead of using a bounding box around the depending iteration nodes of the production ID, the DP can be generated solely based on the DV. In this case the maximal DV length along any dimension decides the maximal storage requirement for different execution orderings. Similar approaches can be used for other iteration domains with diagonal like shapes. Our experiments show that these cases appear very rarely, so it is acceptable to integrate them in the overall technique using a case by case strategy. If we perform a manual generation of this particular DP in accordance with this new procedure, the estimation results are the same as for DTSE. For the main contribution of this paper, the guiding principles, it is in any case irrelevant how the DP is generated. A more computationally complex generation will slow down tools using the principles. The reduced estimation speed is not dramatic, however, since the generation of the DP is a preprocessing step that does not need to be repeated during estimation.

For the Durbin algorithm, the storage requirement of the original application code is identical with the solution after transformations. Before being able to come to this conclusion, the designer still needs to explore the huge number of different transformation alternatives. This requires the guiding principles and tool support described here. The other applications demonstrate how the original application code most often is far from optimal. Orders of magnitude reduction in memory size is achieved when techniques are applied that benefit strongly from use of the guiding principles and size estimation. We also see how important it is to select the optimal loop ordering for the final solution. Orders of magnitude difference between the best case and worst case solution can be seen here as well.

Even if the version of the STOREQ tool used only focuses on individual dependencies, it fully supports the guiding principles. Their low computational complexity can hence be seen from the speed of the tool. We have run STOREQ on the Matlab platform on a Linux server with a Dual 2.4 GHz Xenon Processor and 3 GB RAM. The maximal cputime reported for the examples in Tab. VI is 0.1s. [34] reports cputimes that are orders of magnitude slower for K2. This is to be expected and acceptable in their case, since they perform an exact calculation. A more detailed discussion of the computational complexity and estimation speed of STOREQ can be found in [38].

VII. Conclusions

We have presented a number of guiding principles for loop ordering. They can be used during the early design stages when the execution ordering of an application is unfixed or partially fixed. For the most relevant cases, we have given proof of their optimality for individual data dependencies in the application code. This ensures that the ordering indicated by the guiding principles enables maximal in-place mapping between data elements that are not alive simultaneously, thereby minimizing the storage requirement of the dependency. Using real life applications, we have demonstrated how the designer can use the guiding principles while performing loop transformations. We have also shown how the guiding principles are an integrated part of our memory size estimation technique. The demonstrator results show orders of magnitude improvement in memory requirement compared to using the declared array sizes. Similar penalties are also seen from choosing the suboptimal ordering of loops.

ABBREVIATIONS

DP	Dependency Part	LR	Length Ratio
DV	Dependency Vector	ND	Nonspanning Dimension
DVP	Dependency Vector Polytope	SD	Spanning Dimension
ID	Iteration Domain	UB	Upper Bound
LB	Lower Bound		

REFERENCES

[1] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design*. Boston, USA: Kluwer Acad. Publ., 1998.

[2] F. Catthoor, K. D. andC. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. Van Achteren, and T. Omnes, *Data Access and Storage Management for Embedded Programmable Processors*. Boston, USA: Kluwer Acad. Publ., 2002.

[3] U. Banerjee, *Dependence Analysis for Supercomputing*. Boston, USA: Kluwer Acad. Publ., 1988.

[4] J. R. Allen and K. Kennedy, "Automatic loop interchange," Proc. of the SIGPLAN'84 Symposium on Compiler Construction, SIGPLAN Notices, vol. 19, pp. 233–246, June 1984.

[5] W. Pugh and D. Wonnacott, "An exact method for analysis of value-based array data dependences," in Proc. 6th Intnl. Wsh. on Languages and Compilers for Parallel Computing, (Portland OR, USA), pp. 546–566, Aug. 1993.

[6] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor, "Transformation to dynamic single assignment using a simple data flow analysis," in Proc. 3rd Asian Symp. on Programming Languages and Systems, APLAS'05, (Tsukuba, Japan), vol. 3780 of *Lecture Notes Comp. Sc.*, pp. 330–346, Springer Verlag, Nov. 2005.

[7] M. Palkovic, E. Brockmeyer, P. Vanbroekhoven, H. Corporaal, and F. Catthoor, "Systematic preprocessing of data dependent constructs for embedded systems," in Proc. 15th Intnl. Wsh. on Integrated Circuit and System Design, Power and Timing Modeling, Optimization and Simulation (PATMOS), (Leuven, Belgium), pp. 89–98, IEEE, Sept. 2005.

[8] I. Verbauwhede, F. Catthoor, J. Vandewalle, and H. De Man, "Background memory management for the synthesis of algebraic algorithms on multi-processor dsp chips," in Proc. VLSI'89, Intnl. Conf. on VLSI, (Munich, Germany), pp. 209–218, Aug. 1989.

[9] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," IEEE Trans. on Parallel and Distributed Systems, vol. 2, pp. 452–471, Oct. 1991.

[10] K. Kennedy and K. S. McKinley, "Optimizing for parallelism and data locality," in Proc. of the 6th international conference on Supercomputing, (Washington, D. C, USA), pp. 323–334, Aug. 1992.

[11] P. Clauss and V. Loechner, "Parametric analysis of polyhedral iteration spaces," J. of VLSI Signal Processing, vol. 19, pp. 179–194, July 1998.

[12] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. San Francisco, USA: KMorgan Kaufmann Publ., 2002.

[13] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," ACM Trans. Programming Languages and Systems, vol. 18, pp. 424–453, July 1996.

[14] K. Danckaert, F. Catthoor, and H. De Man, "A loop transformation approach for combined parallelization and data transfer and storage optimization," in Proc. ACM Conf. on Par. and Dist. Proc. Techniques and Applications, PDPTA'00, (Las Vegas NV, USA), pp. 2591–2597, June 2000.

[15] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor, "Multi-dimensional incremental loop fusion for data locality," in Proc. IEEE International Conference on Application-Specific Systems, Architectures, and Processors, ASAP'03, (Leiden, The Netherlands), pp. 17–27, June 2003.

[16] E. De Greef, F. Catthoor, and H. De Man, "Array placement for storage size reduction in embedded multimedia systems," in Proc. Intnl. Conf. on Applic.-Spec. Systems Arch. and Processors, (Zurich, Switzerland), pp. 66–75, July 1997.

[17] V. Lefebvre and P. Feautrier, "Optimizing storage size for static control programs in automatic parallelizers," in Proc. EuroPar Conf., vol. 1300 of *Lecture notes in computer science*, (Passau, Germany), pp. 356–363, Springer Verlag, Aug. 1997.

[18] F. Quillere and S. Rajopadhye, "Optimizing memory usage in the polyhedral model," ACM Trans. on Programming Languages and Systems, vol. 22, pp. 773–815, Sept. 2000.

[19] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," IEEE Trans. on Computers, vol. 54, pp. 1242–1257, Oct. 2005.

[20] C. Chakrabarti, "Cache design and exploration for low power embedded systems," in Proc. Intnl. Conf. on Performance, Computing, and Communications, (Phoenix, Arizona, USA), pp. 135–139, IEEE, Apr. 2001.

[21] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "A compiler-based approach for dynamically managing scratch-pad memories in embedded systems," IEEE Trans. on Comp.-aided Design, vol. 23, pp. 243–260, Feb. 2004.

[22] D. Kirovski, C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Application-driven synthesis of memory-intensive systems-on-chip," IEEE Trans. on Comp.-aided Design, vol. 18, pp. 1316–1326, Sept. 1999.

[23] P. R. Panda, N. D. Dutt, and A. Nicolau, "Local memory exploration and optimization in embedded systems," IEEE Trans. on Comp.-aided Design, vol. 18, pp. 3–13, Jan. 1999.

[24] F. Kurdahi and A. Parker, "Real: a program for register allocation," in Proc. 24th ACM/IEEE Design Automation Conf., (Miami FL, USA), pp. 210–215, June 1987.

[25] S. Y. Ohm, F. J. Kurdahi, and N. Dutt, "Comprehensive lower bound estimation from behavioral description," in IEEE/ACM Intnl. Conf. on Computer-Aided Design, (San Jose CA, USA), pp. 182–187, IEEE, Nov. 1994.

[26] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of asics," IEEE Trans. on Comp.-aided Design, vol. 8, pp. 661–679, June 1989.

[27] C.-J. Tseng and D. Siewiorek, "Automated synthesis of data paths in digital systems," IEEE Trans. on Comp.-aided Design, vol. 5, pp. 379–395, July 1986.

[28] C. H. Gebotys and M. I. Elmasry, "Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis," in Proc. of the 28th ACM/IEEE Design Automation Conf., (San Jose CA, USA), pp. 2–7, Nov. 1991.

[29] I. Verbauwhede, C. Scheers, and J. Rabaey, "Memory estimation for high-level synthesis," in Proc. 31st ACM/IEEE Design Automation Conf., (San Diego CA, USA), pp. 143–148, June 1994.

[30] P. Grun, F. Balasa, and N. Dutt, "Memory size estimation for multimedia applications," in Proc. ACM/IEEE Wsh. on Hardware/Software Co-Design (Codes), (Seattle WA, USA), pp. 145–149, Mar. 1998.

[31] Y. Zhao and S.Malik, "Exact memory size estimation for array computation without loop unrolling," in 36th ACM/IEEE Design Automation Conf., (New Orleans, USA), pp. 811–816, June 1999.

[32] J. Ramanujam, J. Hong, M. Kandemir, and A. Narayan, "Reducing memory requirements of nested loops for embedded systems," in 38th ACM/IEEE Design Automation Conf., (Las Vegas NV, USA), pp. 359–364, June 2001.

[33] F. Balasa, F. Catthoor, and H. De Man, "Background memory area estimation for multi-dimensional signal processing systems," *IEEE Trans. on VLSI Systems*, vol. 3, pp. 157–172, June 1995.

[34] F. Balasa, H. Zhu, and I. Luican, "Computation of storage requirements for multi-dimensional signal processing applications," *IEEE Trans. on VLSI Systems*, vol. 15, pp. 447–460, Apr. 2007.

[35] Q. Hu, A. Vandecappelle, M. Palkovic, P. G. Kjeldsberg, E. Brockmeyer, and F. Catthoor, "Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications," in Proc. of the 11th Asia and South Pacific Design Automation Conference, ASP-DAC 2006, (Yokohama, Japan), pp. 606–611, Jan. 2006.

[36] A. Smailagic (guest editor), "Special issue on system level design," IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 9, Dec. 2001.

[37] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas, "Detection of partially simultaneously alive signals in storage requirement estimation for data-intensive applications," IEEE Trans. on Comp.-aided Design, vol. 22, pp. 908–921, July 2003.

[38] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas, "Storage requirement estimation for optimized design of data intensive applications," ACM Trans. Design Automation of Electronic Systems, vol. 9, pp. 133–158, Apr. 2004.

[39] M. van Swaaij, F. Franssen, F. Catthoor, and H. De Man, "Modeling data flow and control flow for high level memory management," in Proc. of the European Conference on Design Automation, (Brussels, Belgium), pp. 8–13, Mar. 1992.

[40] E. De Greef, F. Catthoor, and H. De Man, "Memory size reduction through storage order optimization for embedded parallel multimedia applications," Elsevier Parallel Computing Journal, vol. 23, pp. 1811–1837, Dec. 1997.

[41] W. Shang, E. Hodzic, and Z. Chen, "On uniformization of affine dependence algorithms," IEEE Trans. on Computers, vol. 45, pp. 827–840, July 1996.

[42] D. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition*. Addison-Wesley, 1997.

[43] IMEC, "Atomium web site," 2007. http://www.imec.be/design/atomium/.

[44] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas, "Detection of partially simultaneously alive signals in storage requirement estimation for data-intensive applications," in 38th ACM/IEEE Design Automation Conf., (Las Vegas N, USA), pp. 365–370, June 2001.

[45] D. Kulkarni and M. Stumm, "Loop and data transformations: A tutorial," Tech. Rep. CSRI-337, Computer Systems Research Inst., Univ. of Toronto, June 1993.

[46] M. Moonen, P. V. Dooren, and J. Vandewalle, "An svd updating algorithm for subspace tracking," SIAM J. Matrix Anal. Appl., vol. 13, no. 4, pp. 1015–1038, 1992.

[47] K. Danckaert, F. Catthoor, and H. De Man, "A preprocessing step for global loop transformations for data transfer and storage optimization," in Proc. Intnl. Conf. on Compilers, Arch. and Synth. for Emb. Sys., (San Jose, CA, USA), pp. 34–40, Nov. 2000.

[48] S. Wuytack, J. P. Diguet, F. Catthoor, and H. De Man, "Formalized methodology for data reuse exploration for low-power hierarchical memory mappings," IEEE Trans. on VLSI Systems, vol. 6, pp. 529–537, Dec. 1998.