Data Dependency Size Estimation for use in Memory Optimization

Per Gunnar Kjeldsberg, Francky Catthoor, and Einar J. Aas

Abstract— **A novel storage requirement estimation methodology is presented for use in the early system design phases when the data transfer ordering is only partly fixed. At that stage, none of the existing estimation tools are adequate, as they either assume a fully specified execution order or ignore it completely. This paper presents an algorithm for automated estimation of strict upper and lower bounds on the individual data dependency sizes in high level application code given a partially fixed execution ordering. In the overall estimation technique, this is followed by a detection of the maximally combined size of simultaneously alive dependencies, resulting in the overall storage requirement of the application. Using representative application demonstrators, we show how our techniques can effectively guide the designer to achieve a transformed specification with low storage requirement.**

Keywords— **system level design, memory, estimation.**

Nomenclature

DP Dependency Part
DV Dependency Vector
DVP Dependency Vector Polytope
LB Lower Bound
LR Length Ratio
ND Nonspanning Dimension
SD Spanning Dimension
UB Upper Bound
See set definitions in Fig. 6 for remaining abbreviations.

I. Introduction

MANY integrated circuit 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. Even for systems with caches, the overall storage requirement has vital impact on the performance and power consumption, since it greatly influences the number of slow and power expensive cache misses. For the system development process, the designer must hence concentrate first on exploring the data transfer and storage to achieve a cost optimized end product [8], [9]. At the system level, no detailed information is available about the size of the memories required for storing data in the alternative realizations of the application. To guide the designer and assist in choosing the best solution, estimation techniques for the storage requirements are needed, very early in the system design trajectory.

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. A straightforward way of estimating the storage requirement is for each array to multiply the size of its dimensions, and then add together the sizes of the different arrays. This will normally result in a huge overestimate however, since not all the arrays, and possibly not all parts of one array, are alive at the same time. In this context an array element is alive from the moment it is written, or produced, and until it is read for the last time. This last read is said to consume the element. To achieve a more accurate estimate, we have to take into account these non-overlapping lifetimes and their resulting opportunity for mapping arrays and parts of arrays in the same place in memory, the so called in-place mapping problem [50]. To what degree it is possible to perform in-place mapping depends heavily on the order in which the elements in the arrays are produced and consumed. This is mainly determined by the execution ordering of the loop nests surrounding the instructions accessing the arrays. In [23] this context and a sketch of a high-level estimation methodology was introduced. This work was continued in [25], presenting an early version of a CAD algorithm for size estimates of individual data dependencies. Finally, [26] discussed detection of simultaneously alive dependencies, leading to estimates of the overall storage requirement of an application.

In this paper, the CAD algorithm for size estimates of individual data dependencies is substantially extended. Some details are also given for the other steps in the estimation methodology. Towards the end, representative application demonstrators are used to illustrate the feasibility and usefulness of the methodology. Finally, we present our conclusions and directions for further work.

II. Previous Work

A. Scalar-Based Estimation

By far the major part of all previous work on storage requirement has been scalar-based. 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

P.G. Kjeldsberg and E. J. Aas are with the Norwegian University of Science and Technology, Trondheim, Norway. E-mail: pgk/aas@fysel.ntnu.no.

F. Catthoor is with IMEC, Leuven, Belgium and also at Katholieke Universiteit Leuven, Belgium. E-mail: catthoor@imec.be.

considered a separate scalar. Using scheduling techniques like the left-edge algorithm, the lifetime of each scalar is found so that scalars with non-overlapping lifetimes can be mapped to the same storage unit [29]. Techniques such as clique partitioning are also exploited to group variables that can be mapped together [48]. In [35], a lower bound for the register count is found without the fixation of a schedule, through the use of As-Soon-As-Possible and As-Late-As-Possible constraints on the operations. A lower bound on the register cost can also be found at any stage of the scheduling process using Force-Directed scheduling [39]. Integer Linear Programming techniques are used in [18] to find the optimal number of memory locations during a simultaneous scheduling and allocation of functional units, registers and busses. A thorough introduction to the scalar-based storage unit estimation can be found in [17]. Common to all scalar based techniques is that they break down when used for large multi-dimensional arrays. The problem is NP-hard and its 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.

B. Estimation Techniques for Multi-Dimensional Arrays

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 is treated separately. Due to the code's loop structure, 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.

In [49], a production time axis is created for each array. This models the relative production and consumption time, or date, of the individual array accesses. The difference between these two dates equals the number of array elements produced between them. The maximum difference found for any two depending instances gives the storage requirement for this array. The total storage requirement is the sum of the requirements for each array. An Integer Linear Programming approach is used to find the date differences. To be able to generate the production time axis and production and consumption dates, the execution ordering has to be fully fixed. Also, since each array is treated separately, only in-place mapping internally to an array (intra-array in-place) is consider, not the possibility of mapping arrays in-place of each other (inter-array in-place).

Another approach is taken in [20]. Assuming procedural execution of the code, the data-dependency relations between the array references in the code are used to find the number of array elements produced or consumed by each assignment. The storage requirement at the end of a loop equals the storage requirement at the beginning of the loop, plus the number of elements produced within the loop, minus the number of elements consumed within the loop. The upper bound for the occupied memory size within a loop is computed by producing as many array elements as possible before any elements are consumed. The lower bound is found with the opposite reasoning. From this, a memory trace of bounding rectangles as a function of time is found. The total storage requirement equals the peak bounding rectangle. If the difference between the upper and lower bounds for this critical rectangle is too large, better estimates can be achieved by splitting the corresponding loop into two loops and rerunning the estimation. In the worst-case situation, a full loop-unrolling is necessary to achieve a satisfactory estimate.

Reference [52] describes a methodology for so-called exact memory size estimation for array computation. It is based on live variable analysis and integer point counting for intersection/union of mappings of parameterized polytopes. In this context, as well as in our methodology, a polytope is the intersection of a finite set of half-spaces and may be specified as the set of solutions to a system of linear inequalities. It is shown that it is only necessary to find the number of live variables for one statement in each innermost loop nest to get the minimum memory size estimate. The live variable analysis is performed for each iteration of the loops however, which makes it computationally hard for large multi-dimensional loop nests.

In [42], a reference window is used for each array in a perfectly nested loop. At any point during execution, the window contains array elements that have already been referenced and will also be referenced in the future. These elements are hence stored in local memory. The maximal window size found gives the memory requirement for the array. The technique assumes a fully fixed execution ordering. If multiple arrays exists, the maximum reference window size equals the sum of the windows for individual arrays. Inter-array in-place is consequently not considered.

In contrast to the methods described so far in this subsection, the storage requirement estimation technique presented in [4] does not take execution ordering into account at all. It starts with an extended data dependency analysis resulting in a number of non-overlapping basic sets and the dependencies between them. The basic sets and dependencies are described as polytopes, using *linearly bounded lattices* (LBLs) of the form

$$x = T \bullet i + u | A \bullet i \geq b$$

where $x \in Z^m$ is the coordinate vector of an m-dimensional array, and $i \in Z^n$ is the vector of n loop iterators. The array index function is characterized by $T \in Z^{m \times n}$ and $u \in Z^m$, while the polytope defining the set of iterator vectors is characterized by $A \in Z^{2n \times n}$ and $b \in Z^{2n}$. The basic set sizes, and the sizes of the dependencies, are found using an efficient lattice point counting technique. The dependency size is the number of elements from one basic set that is read while producing the depending basic set. The total storage requirement is found through a greedy traversal of the corresponding data flow graph. The maximal combined size of simultaneously alive basic sets gives the storage requirement. Since no execution ordering is taken into account, all elements of a basic set are assumed

produced before the first element is consumed. This gives rise to an overestimate compared to all but the worst-case ordering.

In summary, all of the previous work on storage requirement entails a fully fixed execution ordering to be determined prior to the estimation. The only exception is the last methodology, 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 recent papers [46], it is necessary to start from a non-fully fixed ordering. In the next section we will present our new estimation technique which addresses this much more complex problem, allowing a partially fixed execution ordering.

As will be demonstrated in Section V, the main usage of storage requirement estimation, is for optimization of an application's overall memory size and for guiding source code transformations to achieve this goal. In addition to the obvious chip size reduction, this will also reduce the application's power consumption. Smaller size means that more data can be located in small memories close to the data path, where the power cost of each memory access is low [51]. Examples of work in the field of memory size optimization through use of in-place mapping include [15], [30], and [41]. All of them require a fully fixed execution ordering. Much research has also gone into estimation and optimization of size, performance and power consumption of memory hierarchies in cache based systems. Examples of this are [10], [22], and [36]. A thorough presentation of data and memory optimization techniques is also given in [37]. Further details are outside the scope of this paper.

III. Estimation with Partially Fixed Execution Ordering

A. Motivation and Context

The new estimation methodology employs the data flow graph generation presented in [4]. In this paper we mainly focus on data dependency size estimation however, which can be utilized for most polyhedral dependency descriptions. The principle improvement compared to previous methods is the possibility to avoid overestimates by taking available execution ordering information (based mainly on loop interchanges) into account. Using an estimation methodology that requires a fully fixed ordering necessitates a full exploration of all alternatives for the unfixed parts. 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 dimensionalities of the individual arrays are smaller. Since it is the number of loop dimensions that determine the complexity, a full exploration is very time consuming and hence not feasible when the designer needs fast feedback regarding an application's storage requirement when the execution ordering is not fully fixed. Instead, our methodology presents precise upper and lower bounds to the designer as guidance during the early high level design trajectory. Throughout the application code conflicting dependency considerations may exist, so an automatic tool is needed to lead the designer to a globally efficient solution.

To enable this global optimization scope of the methodology, all dependencies are placed in a common iteration space [11], [26]. This can be regarded as representing one loop nest surrounding the complete application code, but is done in such a way that the already fixed part of the execution ordering can be kept. The common iteration space opens for estimation of dependencies between statements in different loop nests in the original code. Optimization of loop interchange and the ordering between statements can have the global view for the unfixed part of the execution ordering.

Our algorithm is useful for a large class of applications. Certain restrictions exist on the code that can be handled in the present version however, some of which will be alleviated through future work. The main requirements are that the code is single assignment and has affine array indexes. This is achievable by a good array data flow analysis preprocessing [16], [40]. Also the resulting Dependency Parts, see below, must be orthogonal, or made orthogonal, as described in [23]. When in the sequel the words upper and lower bounds are used, the bounds after orthogonalization is meant. The lower bounds may not be factual, since some approximations tend to result in overestimates so that realizations with lower storage requirements may exist. For a large class of applications, where the Dependency Parts are already orthogonal, and when the generation of the best-case and worst-case common iteration space is possible, both bounds are correct. A proof of the correctness can be found in [27] together with a discussion of the estimation error induced by the orthogonalization.

Let us take a look at the simple application code example shown in Fig. 1. Two statements, S.1 and S.2, produce elements of two arrays, A and B. Elements from array A are consumed when elements of array B are produced. This gives rise to a flow type data dependency between the statements [5].

```
for i=0 to 5 {
  for j=0 to 5 {
    for k=0 to 2 {
S.1   A[i][j][k] = f1( input );
S.2   if(i>=1)&(j>=2) B[i][j][k] = f2( A[i-1][j-2][k] );
}}}
```

Fig. 1. Code example for concept definitions.

The loops around the statements define an *iteration space*, [5], as shown in Fig. 2. Each node within this space represents one execution of the statements inside the loop nest. For our example, at each of these *iteration nodes* one A-array element and, when the if clause condition is

true, one B-array element is produced. These nodes also constitute the *iteration domains* of the two statements, as indicated by the rectangles *A(0)* and *B(0)* in Fig. 2. To avoid unnecessarily complex figures, two-dimensional rectangles are used. All nodes within the two rectangles are part of the corresponding iteration domains. Fig. 2 also shows the corresponding data-flow graph for the code example in Fig. 1. The nodes are the iteration domains, while the dependencies between them are the branches.

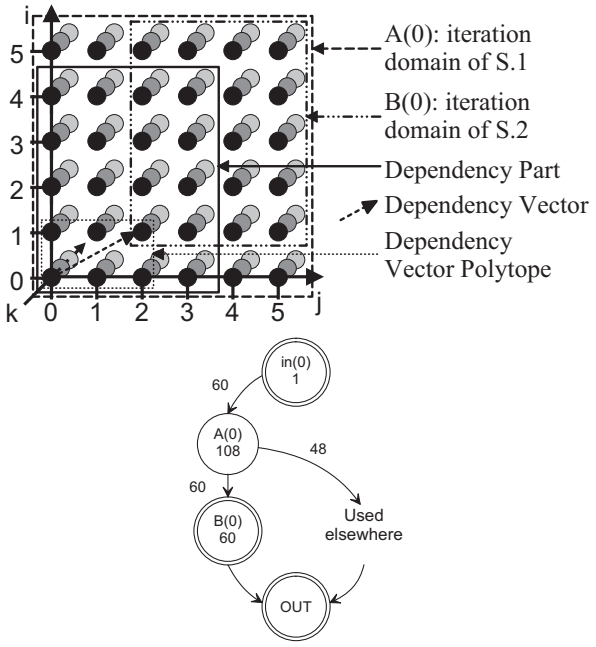


Fig. 2. Iteration space and data-flow graph of example in Fig. 1.

In general not all elements produced by one statement are read by a 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. A *Dependency Vector* (DV) is drawn from an iteration node in the DP producing an array element to the iteration node producing the depending element. This DV spans a *Dependency Vector Polytope* (DVP) and its dimensions are defined as *Spanning Dimensions* (SD). Since the SD normally only comprises a subset of the iterator space dimensions, the remaining dimensions are denoted *Nonspanning Dimensions* (ND). The largest value for each dimension in the DVP is denoted *Spanning Value* (SV) of that dimension. For the DVP in Fig. 2, i and j are SDs while k is ND. Furthermore we have $SV_i = 1$ and $SV_j = 2$.

The DP and DVP can also be represented using an LBL description as shown in Fig. 3. The vertical line in the LBLs divides the description into an array index function and a restriction part. The restricted function for the u index of $DP_{A(0)-B(0)}$ can thus be read as $u = i \; for \; 0 \leq i \leq 4$. Note that for comparison with the DP a three-dimensional LBL is used for the DVP, even though it only has two dimensions, i and j.

In many cases, the end point of the DV falls outside the DP. This happens when no overlap exists between the two depending iteration domains. The DVP is then intersected

$$DP_{A(0)-B(0)} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \left| \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 0 \\ -4 \\ 0 \\ -3 \\ 0 \\ -2 \end{bmatrix} \right.$$

$$DVP_{A(0)-B(0)} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \left| \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 0 \\ -1 \\ 0 \\ -2 \\ 0 \\ 0 \end{bmatrix} \right.$$

Fig. 3. LBL description of DP and DVP for A(0) with respect to B(0) of example in Fig. 1.

with the DP. For reasons that will become apparent in Section IV-B, the DVP is further extended with one in all dimensions for which the DV falls outside the DP.

When a dependency is not uniform, the DVs for different iteration nodes within its DP can differ in both length and direction. The DVP is then generated using the extreme DVs as introduced in [13]. Use of the extreme DVs is a simple approach to uniformization suitable for estimation when a short run time is required to give fast feedback to the designer. Alternative techniques for uniformization of affine dependency algorithms are presented in [44]. The complexity of these procedures is however much larger.

With the dependency size estimation methodology from [4], the size of the dependency between S.1 and S.2 in Fig. 1 will be the number of iteration nodes in the full DP, that is 60. This is an overestimate even for the worst-case ordering since the basic sets are overlapping. Inside this overlap both A-array elements and B-array elements are produced, and the B-array elements produced by S.1 can thus be mapped in-place of the A-array elements consumed by S.2. This overlap should therefore be removed, reducing the dependency size estimate to 36. An even more accurate estimate can be achieved by also taking available partially fixed execution ordering into account. The new estimation methodology presented in this paper is able to do this, producing upper and lower bounds on the dependency sizes. Even though this ability is its prime advantage, our technique is also useful even for a fully fixed ordering. At this point of the design trajectory other techniques, for example the one presented in [15], could however be as effective due to the reduced search space.

B. Overall Estimation Strategy

The storage requirement estimation methodology can be divided into four steps as shown in Fig. 4. The first step uses the technique from [4] to generate a data-flow graph reflecting the data dependencies in the application code. Any complete polyhedral dependency model can be used however, and work is in progress to integrate the estimation methodology with the ATOMIUM tool [1], [6]. The second step places the DPs of the iteration domains in a common iteration space according to their dependencies and the partially fixed execution ordering. Best-case and worst-case placements may be used to achieve lower and upper bounds respectively. References [11] and [12] de-

scribes work that can be utilized for this step. The third step focuses on the size of individual dependencies between the iteration domains in the common iteration space. The general principles were introduced in [23] with a first version of an estimation algorithm in [25]. A mature and detailed algorithm for estimation of upper bounds (UB) and lower bounds (LB) will now be presented in the following sections of this paper. The final step searches for simultaneously alive dependencies in the common iteration space, and calculates their maximal combined size. See [26] for details. A prototype estimation and optimization tool, STOREQ, has been developed for main parts of the steps [27].

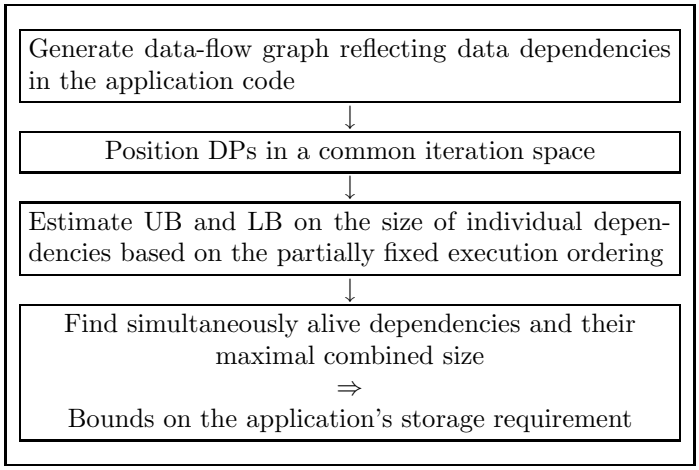


Fig. 4. Overall storage requirement estimation strategy.

IV. Size Estimation of Individual Dependencies

A most crucial step of the storage requirement estimation methodology is the estimation of the size of individual data dependencies in the code. This information can be used directly to investigate the largest and thus globally determining dependencies, or it can be used as input to the subsequent search for the maximal size of simultaneously alive dependencies.

Our estimation algorithm uses a number of principles for the best-case and worst-case ordering of dependency dimensions. These principles are presented first, followed by the actual algorithm and a small demonstrating example.

A. Ordering principles

In [24] we introduced a number of guiding principles for the ordering of the dimensions of a dependency. Table I shows the typical consequences of fixing SDs and NDs innermost and outermost in a loop nest. The size of a dependency is minimized if the execution ordering is fixed so that its NDs are fixed at the outermost nest levels and the SDs are fixed at the innermost nest levels. The proof of this and the other guiding principles discussed here can be found in [27]. 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 fix the *shortest* NDs inside the SDs. The length of ND d_i is determined by the length of the DP in this dimension, $|DP_{d_i}|$, and equals the number of iteration nodes covered by a projection of the DP onto this dimension. Both the start and end node of a dimension is included in its length.

	Fixed innermost	Fixed outermost
SD	*Small* increase in LB *Big* reduction in UB	*Big* increase in LB *Small* reduction in UB
ND	*Big* increase in LB Unchanged UB	Unchanged LB *Big* reduction in UB

TABLE I
Typical consequences of fixation of spanning and nonspanning dimensions.

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

$$LR_{d_i} = \frac{|DVP_{d_i}| - 1}{|DP_{d_i}| - 1}$$

or the length of the DVP in dimension d_i minus one divided by the length of the DP in dimension d_i minus one.

For the code of Fig. 1, the calculation of the LRs for the i and j dimensions are illustrated in Fig. 5. According to the guiding principles, the j dimension should be fixed innermost giving a dependency size of six as will be shown in Section IV-D.1. The alternative ordering, with the i dimension innermost, would give a dependency size of eleven.

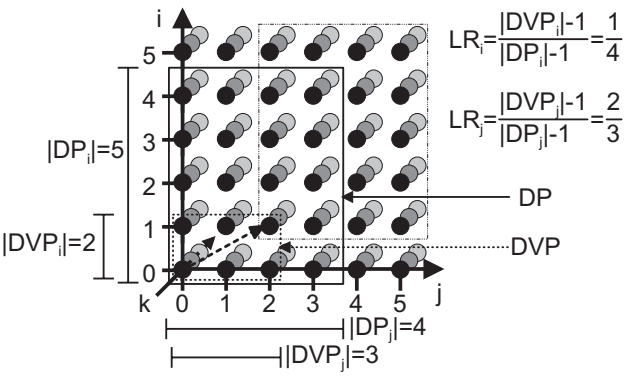


Fig. 5. Calculation of LRs for the code example of Fig. 1.

B. Algorithm for Estimation of Bounds

The ordering principles presented in the previous section are used to find a best-case and worst-case ordering of the dimensions of the dependency. Using these two orderings, exact upper and lower bounds on the storage requirement are calculated. Any dimensions fixed at given nest levels in the common loop nest are taken into account.

Fig. 6 describes an algorithm for automated estimation of the UB and LB on the storage requirement for individual dependencies. The algorithm also outputs the best case

ordering of the unfixed dimensions. The input to the algorithm consists of the LBL descriptions of the DP and DVP in addition to the ordered set of *Fixed Dimensions* (FD) starting with the outermost nest level. The unfixed dimensions of the FD set are represented by X. A five dimensional iteration space with dimension d_3 fixed second outermost and d_4 fixed innermost is thus presented as FD=$\{$X,d_3,X,X,$d_4\}$. For increased readability of the algorithm, the result of a multiplication operation over an empty set is assumed to return 1 ($\prod_{\forall d_i \in S} |A_{d_i}| = 1 \; if \; S = \emptyset$) and thus can be ignored.

The algorithm starts by defining a number of sets that are used during the calculation of the LB and UB. They divide the dimensions into fixed and unfixed Spanning Dimensions (SDs) and Nonspanning Dimensions (NDs). The unfixed SDs are ordered according to increasing LR for the LB calculation and decreasing LR for the UB calculation. The unfixed NDs are ordered according to decreasing $|DP_{d_i}|$ for the LB calculation and increasing $|DP_{d_i}|$ for the UB calculation. The reasoning behind this ordering is based on the guiding principles.

Starting with the outermost nest level of the FD set, the contributions of each dimension to the LB and UB are calculated. If the current FD set element, c_i, is unfixed, represented by an X, the sets containing unfixed SDs and NDs are used for the contribution calculation. As long as unfixed NDs exist, nothing is added to the LB. The only action taken is the removal of the next dimension from the set of unfixed NDs. If no more unfixed NDs are left, the next unfixed SD is removed from the LB set of unfixed SDs and used for the LB contribution calculation. Its $|DVP_{d_i}|$ minus 1 is multiplied with the $|DP_{d_i}|$ of all other remaining dimensions. This is demonstrated in Fig. 7 for the code example of Fig. 1. The contribution of the j dimension if placed outermost is indicated. The number of iteration nodes within the contribution rectangle equals $(|DVP_j| - 1) * |DP_k| * |DP_i| = (3 - 1) * 3 * 5 = 30$. Note that this is not the ordering that would have been used for the LB calculation, but it demonstrates nicely how the contribution is calculated.

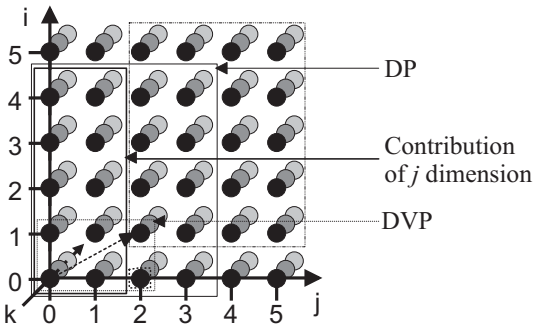


Fig. 7. Contribution of j dimension when placed at the outermost nest level.

In some cases, the LR of a dimension is larger than one, indicating that no overlap exists between the DP and the depending iteration nodes. The calculated contribution of this dimension will then include all elements of the remaining dimensions within the DP. Consequently, the remaining dimensions will not contribute further, and the LB calculation can be terminated.

The calculation of the contribution of c_i to the UB is performed in a similar way as the LB calculation, except that unfixed SDs are placed outside the unfixed NDs since this corresponds to the worst case ordering. Since the SD with the largest LR is used first, any unfixed non-overlapping dimensions will immediately cause all elements in the DP to be included in its contribution, and hence the UB calculation can be terminated. Note that the LB calculation must continue, so the function does not terminate.

If the current FD set element c_i is fixed, represented by the dimension fixed at this position, the dimension itself is used for the contribution calculation. If it is an ND, it is simply removed from the sets containing the fixed NDs for the UB and LB calculations. If it is an SD, its $|DVP_{c_i}|$ minus 1 is multiplied with the $|DP_{d_i}|$ of all other remaining dimensions, as demonstrated in Fig. 7. This is done for both the UB and LB contribution calculation but unless the current dimension is fixed outermost, the content of the sets used for the calculation of the two contributions may be different. The resulting UB and LB contributions may then also be different. The contributions to both the UB and LB are terminated immediately if the contribution of a non-overlapping SD is calculated.

In addition to returning the upper and lower bounds on the storage requirement, the algorithm also returns an ordered set, *Fixed Dimensions Best Case* (FDBC), with the best case ordering of the unfixed dimensions. For each element c_i in the FD set, the corresponding element in FDBC is the dimension used for calculating the contribution to the LB. If the current element in FD is fixed, its dimension is directly inserted at the same location in FDBC. If the current element in FD is unfixed, the same dimension selected for LB calculation is inserted as the current element in FDBC. The FDBC can guide the designer towards an ordering of dimensions that is optimal for one or for a number of dependencies.

In a few rare situations, the guiding principles do not result in a guaranteed optimal ordering. The estimated bounds would then not necessarily be correct. This may happen if internally fixed dimensions exist. In the following FD set, $FD = \{X, d_3, X, X, d_4\}$, dimension d_3 is fixed internally since it has an X on both sides. In most cases the guiding principles can be used even with internally fixed dimensions. The function *GuidingPrinciplesOK( )* in Fig. 6 checks for certain properties of the internally fixed dimensions, and returns FALSE if the guiding principles can not be used with a guaranteed result. See [27] for a detailed discussion of the situations where this is the case.

C. Estimation with partial ordering among dimensions

In section IV-B it is assumed that dimensions are fixed at a given position in the FD set and thus at certain nest levels in the loop nest. It is however also interesting to be able to estimate upper and lower bounds on the dependency

EstimateDependencySize(DependencyPart (DP), DependencyVectorPolytope(DVP), FixedDimensions (FD))
define set AllDimensions (AD) = (all dimensions in DP)
define set SpanningDimensions (SD) = (all dimensions in DVP)
define set NonspanningDimensions (ND) = AD - SD
define set LowerboundFixedSpanningDimensions (LFSD) = SD ∩ FD
define ordered set LowerboundUnfixedSpanningDimensions (LUSD)= SD - LFSD
define set LowerboundFixedNonspanningDimensions (LFND) = ND ∩ FD
define ordered set LowerboundUnfixedNonspanningDimensions (LUND) = ND - LFND
order $\forall d_i \in LUSD \left| \left(d_i \prec d_{i+1} \Leftrightarrow LR_{d_i} < LR_{d_{i+1}} \right) \right.$ /* According to increasing LR */
order $\forall d_i \in LUND \left| \left(d_i \prec d_{i+1} \Leftrightarrow \left|DP_{d_i}\right| > \left|DP_{d_{i+1}}\right| \right) \right.$ /* According to decreasing $\left|DP_{d_i}\right|$ */
define set UpperboundFixedSpanningDimensions (UFSD) = LFSD
define ordered set UpperboundUnfixedSpanningDimensions (UUSD) = LUSD in opposite order
define set UpperboundFixedNonspanningDimensions (UFND) = LFND
define ordered set UpperboundUnfixedNonspanningDimensions (UUND) = LUND in opposite order
define ordered set FixedDimensionsBestCase (FDBC) = [ ]
define value LowerBound (LB) = 0
define value UpperBound (UB) = 0
define value LBFinished = 0
define value UBFinished = 0
if GuidingPrinciplesOK(DP, DVP, FD, SD, LUSD) {
for each element $c_i \in FD$ {
if FD_{c_i} == X { /* The current dimension is unfixed */
if LBFinished ≠ 1 { /* No non-overlapping SDs so far used in LB calculation */
if LUND ≠ ∅ { /* Remaining unfixed ND exists for use in LB calculation */
d_i = (next dimension in LUND)
LUND = LUND - d_i /* Remove the next dimension from LUND */
}
else { /* Use an unfixed SD for LB calculation */
d_i = (next dimension in LUSD)
if $LR_{d_i} > 1$ /* The dimension is non-overlapping */
LBFinished = 1
LUSD = LUSD - d_i /* Remove the next dimension from LUSD */
$LB = LB + \left(\left|DVP_{d_i}\right| - 1\right) * \prod_{\forall d_j \in LUSD} \left|DP_{d_j}\right| * \prod_{\forall d_j \in LFSD} \left|DP_{d_j}\right| * \prod_{\forall d_j \in LFND} \left|DP_{d_j}\right|$
}
$FDBC_{c_i} = d_i$ /* Insert dimension used for LB calculation in FDBC */
}
else /* Non-overlapping SD already used in LB calculation */
$FDBC_{c_i} = 0$ /* Insert pseudo-dimension in FDBC */
if UBFinished ≠ 1 { /* No non-overlapping SDs so far used in UB calculation */
if UUSD ≠ ∅ { /* Remaining unfixed SD exists for use in UB calculation */
d_i = (next dimension in UUSD)
if $LR_{d_i} > 1$ /* The dimension is non-overlapping */
UBFinished = 1
UUSD = UUSD - d_i /* Remove the next dimension from UUSD */
$UB = UB + \left(\left|DVP_{d_i}\right| - 1\right) * \prod_{\forall d_j \in UUSD} \left|DP_{d_j}\right| * \prod_{\forall d_j \in UUND} \left|DP_{d_j}\right| * \prod_{\forall d_j \in UFSD} \left|DP_{d_j}\right| * \prod_{\forall d_j \in UFND} \left|DP_{d_j}\right|$
}
else /* Use an unfixed ND for UB calculation */
UUND = UUND - (next dimension in UUND) /* Remove the next dimension from UUND */
} }
else /* The current dimension is fixed */
$d_i = FD_{c_i}$
$FDBC_{c_i} = d_i$ /* Insert the current fixed dimension in FDBC */
if $d_i \in$ ND { /* Use the fixed ND for LB and UB calculation */
LFND = LFND - d_i /* Remove the fixed ND from LFND */
UFND = UFND - d_i /* Remove the fixed ND from UFND */
}
else { /* Use the fixed SD for LB and UB calculation */
LFSD = LFSD - d_i /* Remove the fixed SD from LFSD */
UFSD = UFSD - d_i /* Remove the fixed SD from UFSD */
if LBFinished ≠ 1 /* No non-overlapping SDs so far used in LB calculation */
$LB = LB + \left(\left|DVP_{d_i}\right| - 1\right) * \prod_{\forall d_j \in LUSD} \left|DP_{d_j}\right| * \prod_{\forall d_j \in LUND} \left|DP_{d_j}\right| * \prod_{\forall d_j \in LFSD} \left|DP_{d_j}\right| * \prod_{\forall d_j \in LFND} \left|DP_{d_j}\right|$
if UBFinished ≠ 1 /* No non-overlapping SDs so far used in UB calculation */
$UB = UB + \left(\left|DVP_{d_i}\right| - 1\right) * \prod_{\forall d_j \in UUSD} \left|DP_{d_j}\right| * \prod_{\forall d_j \in UUND} \left|DP_{d_j}\right| * \prod_{\forall d_j \in UFSD} \left|DP_{d_j}\right| * \prod_{\forall d_j \in UFND} \left|DP_{d_j}\right|$
if $LR_{d_i} > 1$ /* The current dimension is non-overlapping */
UBFinished = LBFinished = 1
} } }
return LB, UB, FDBC

Fig. 6. Pseudo-code for dependency size estimation algorithm.

size when the only information available is a partial ordering 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 may for example happen when a dependency has three SDs, and one of them is negatively directed through use of the following array index in the consuming array access: [i+1]. If this dimension is fixed outermost among the SDs, the whole dependency will become negative, which is illegal for any flow dependency [28]. The designer still has full freedom in placing the three dimensions in the FD set and also regarding which one of the others (or both) to place outside the negative dimension. The main difference between a partially fixed ordering and a partial ordering among dimensions is thus that the partial ordering does not include fixation of dimensions at given nest levels in the FD set.

It is possible to use a different algorithm for the dependency size estimation when these kinds of constraints are given. An alternative approach using the same algorithm has been chosen here since it is usually only necessary to activate the existing algorithm twice with two different FD sets to obtain the results, one for the upper bound and one for the lower bound. The typical situation is that the partial ordering prohibits one dimension to be fixed innermost or outermost among a group of dimensions. Currently these dimensions have to follow each other in the best case and worst case ordering. The extension to allow other dimensions to have an LR that is larger than some and smaller than other of the LRs of the group is subject to future work. For the typical situation, where the partial ordering is among the dependency's entire set of SDs, these dimensions will in any case follow each other in the best case and worst case orderings. To find which FD sets to use, the LRs of the dimensions in the partial ordering is compared with the best-case and worst-case orderings. Depending on the result of this comparison, FDs are generated as follows:

1. If the partial ordering is in accordance with both the best case and worst case ordering, the estimation algorithm is simply activated with a fully unfixed FD (FD={...,X,X,X,...}). The resulting LB and UB returned from the algorithm is immediately correct for this partial ordering.
2. If dimension d_i with the largest LR among a group of dimensions is not to be ordered outermost among these dimensions, the LB is found through an activation of the estimation algorithm with a fully unfixed FD (FD={...,X,X,X,...}). If n dimensions with larger LR than d_i are present, the UB is found through an activation of the estimation algorithm with an FD where d_i is preceded by n+1 X's (FD={X,X,d_i,X...} if n=1).
3. If dimension d_i with the largest LR among a group of dimensions is not to be ordered innermost among these dimensions, the UB is found through an activation of the estimation algorithm with a fully unfixed FD (FD={...,X,X,X,...}). If n dimensions with larger LR than d_i are present, the LB is found through an activation of the estimation algorithm with an FD where d_i is followed by n+1 X's (FD= {...X,d_i,X,X} if n=1).
4. If dimension d_i with the smallest LR among a group of dimensions is not to be ordered outermost among these dimensions, the UB is found through an activation of the estimation algorithm with a fully unfixed FD (FD={...,X,X,X,...}). If n dimensions with smaller LR than d_i are present, the LB is found through an activation of the estimation algorithm with an FD where d_i is preceded by n+1 X's (FD= {X,X,d_i,X...} if n=1).
5. If dimension d_i with the smallest LR among a group of dimensions is not to be ordered innermost among these dimensions, the LB is found through an activation of the estimation algorithm with a fully unfixed FD (FD={...,X,X,X,...}). If n dimensions with smaller LR than d_i are present, the UB is found through an activation of the estimation algorithm with an FD where d_i is followed by n+1 X's (FD= {...X,d_i,X,X} if n=1).

	d_i not outermost in group	d_i not innermost in group
d_i largest LR in group	FD_{LB} = {...,X,X,X,...}	FD_{LB} = {...,X,d_i,X,X}
	FD_{UB} = {X,X,d_i,X,...}	FD_{UB} = {...,X,X,X,...}
d_i smallest LR in group	FD_{LB} = {X,X,d_i,X,...}	FD_{LB} = {...,X,X,X,...}
	FD_{UB} = {...,X,X,X,...}	FD_{UB} = {...,X,d_i,X,X}

TABLE II
FDs USED FOR DEPENDENCY SIZE ESTIMATION WITH PARTIAL ORDERING AMONG DIMENSIONS. N=1 IN ALL CASES.

Table II summarizes the use of FDs for the four situations where the partial ordering is not in accordance with both the best case and worst case ordering.

The reasoning behind the use of these FD sets is based on the guiding principles. When the partial order is in accordance with both the best case and worst case ordering a single fully unfixed FD can be used to find both the UB and LB since both the worst case and best case ordering used by the estimation algorithm is legal. For the other cases, the partial ordering is in accordance with either the best case or the worst case ordering. The fully unfixed FD can then be used to find the UB or LB respectively since the ordering used by this part of the estimation algorithm is legal. The other ordering used by the estimation algorithm places dimension d_i illegally outermost or innermost among the dimensions in the group. The legal best case or worst case ordering can be reached if d_i is swapped with the dimension in the group ordered next to it. This is achieved through a fixation of d_i at this dimension's position in the worst case or best case ordering. The resulting FDs are shown in Table II for the cases where d_i is normally placed second innermost or second outermost (n=1) in the ordered set of all dimensions in the Dependency Part.

Given that the dimensions are already sorted according to their LR, it would also be possible to use fully fixed FDs to find the UB and LB. This would however prohibit estimation with a combination of the partially fixed ordering methodologies presented in section IV-B and the partial ordering among dimensions presented here. To have the

freedom of this combination for future work, the FDs are kept as unfixed as possible also for estimations with a partial ordering among dimensions.

D. Automated Estimation on Code Examples

In this section, estimates are performed on two code examples. First, the three dimensional example presented in Fig. 1 is used. It is quite simple, so the full details of the estimation algorithm can be illustrated. This is followed by a six dimensional example that shows the effect of a number of partially fixed execution orderings. Estimations using larger and realistic design examples can be found in section V.

D.1 Simple Three Dimensional Example

Let us go back to the example code in Fig. 1. If no execution ordering is specified, FD={X,X,X}, the algorithm in Fig. 6 orders all SDs according to their LR. $LR_i = (2-1)/(5-1) = 1/4$ and $LR_j = (3-1)/(4-1) = 2/3$, so LUSD={i,j} for the LB calculation and UUSD={j,i} for the UB calculation (see Fig. 6 for an explanation of the set abbreviations). Since k is the only ND, the orderings of the LUND and UUND sets are trivial. With $c_1 = X$ the LB calculation simply removes the unfixed ND k from the LUND set. The UB calculation selects the first UUSD dimension, j. The UFSD and UFND sets are empty, so together with the remaining UUSD element, i, and the UUND element, k, the resulting temporary UB value is $UB_{tmp} = UB_{tmp} + (|DVP_j| - 1) * |DP_i| * |DP_k| = 0 + 2*5*3 = 30$. With $c_2 = X$, the LUND set is empty, and the LB calculation selects the first LUSD dimension, i. The only nonempty set for the LB calculation is LUSD resulting in a temporary LB value of $LB_{tmp} = LB_{tmp} + (|DVP_i| - 1) * |DP_j| = 0 + 1 * 4 = 4$. The UB calculation selects the next UUSD dimension, i, resulting in a temporary UB value of $UB_{tmp} = UB_{tmp} + (|DVP_i| - 1) * |DP_k| = 30 + 1 * 3 = 33$. With $c_3 = X$ the LB calculation selects the next LUSD dimension, j, resulting in a final LB value of $LB = LB_{tmp} + (|DVP_j| - 1) = 4 + 2 = 6$. The UB calculation simply removes the unfixed ND k from the UUND set leaving the previously calculated UB as the final UB. A graphical description of the A-array elements that make up the upper and lower bounds is given in Fig. 8.

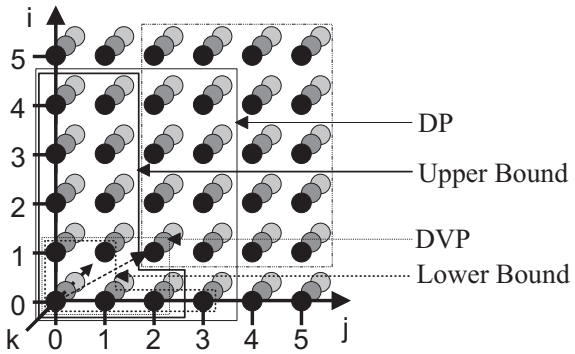


Fig. 8. UB and LB with no execution ordering fixed.

Assume now that the k dimension is specified as the outermost dimension while the ordering between the i and j dimensions are still unresolved, FD={k,X,X}. The LUSD and UUSD sets are then initially ordered the same way as before while the k dimension is moved from the LUND and UUND to the LFND and UFND sets respectively. With $c_1 = k$, the LB and UB calculations simply removes k from the LFND and UFND sets. During the subsequent calculation of the UB, all but the UFSD set is empty and the calculation is therefore reduced to $UB_{tmp} = UB_{tmp} + (|DVP_j| - 1) * |DP_i| = 0 + 2 * 5 = 10$, and $UB = UB_{tmp} + (|DVP_i| - 1) = 10 + 1 = 11$. The LB calculation already assumed NDs to be placed outermost. Consequently, placing the k dimension outermost gives an unchanged LB and a decreased UB.

For the next partial fixation, the j dimension is placed innermost and the k dimension second innermost, FD={X,k,j}. The resulting sets are UFSD = LFSD = {j}, UFND = LFND = {k}, UUSD = LUSD = {i}, and UUND = LUND = $\emptyset$. With $c_1 = X$ and LUND = $\emptyset$ the LB calculation is $LB_{tmp} = LB_{tmp} + (|DVP_i| - 1) * |DP_j| * |DP_k| = 0 + 1 * 4 * 3 = 12$. Since UUSD $\neq \emptyset$ the UB calculation is $UB_{tmp} = UB_{tmp} + (|DVP_i| - 1) * |DP_j| * |DP_k| = 0 + 1 * 4 * 3 = 12$. $c_2 = k$ results in removal of k from the LFND and UFND sets. Finally $c_3 = j$ gives $LB = LB_{tmp} + (|DVP_j| - 1) = 12 + 2 = 14$ and $UB = UB_{tmp} + (|DVP_j| - 1) = 12 + 2 = 14$. Note that given what is actually a fully fixed ordering since only one dimension is unfixed, the UB and LB calculations are identical.

Consider the situation where the j dimension is fixed internally, FD= {X,j,X}. The resulting sets are UFSD = LFSD = {j}, UUSD = LUSD = {i}, UUND = LUND = {k}, and UFND = LFND = $\emptyset$. With $c_1 = X$ the LB calculation simply removes the k dimension from the LUND set. The UB calculation uses the i dimension from the UUSD set to find $UB_{tmp} = UB_{tmp} + (|DVP_i| - 1) * |DP_j| * |DP_k| = 0 + 1 * 4 * 3 = 12$. With $c_2 = j$ the LB calculation is $LB_{tmp} = LB_{tmp} + (|DVP_j| - 1) * |DP_i| = 0 + 2 * 5 = 10$ and the UB calculation is $UB_{tmp} = UB_{tmp} + (|DVP_j| - 1) * |DP_k| = 12 + 2 * 3 = 18$. Finally with $c_3 = X$ the LB calculation uses the i dimension from the LUSD set to find $LB_{tmp} = LB_{tmp} + (|DVP_i| - 1) = 10 + 1 = 11$. The UB calculation simply removes the k dimension from the UUND set leaving the previously calculated UB as the final UB.

A partial ordering is now given forcing the j dimension not to be placed innermost among the j dimension and i dimension. Since $LR_j > LR_i$ this is not in accordance with the best case ordering and FDs are generated as described in clause 3 of section IV-C. No dimensions with LR larger than j exist, so n=0 giving $FD_{LB} = \{X, j, X\}$ and $FD_{UB} = \{X, X, X\}$. Both of these have been detailed above, and the resulting LB found using $FD_{LB} = \{X, j, X\}$ is 11, while the UB found using $FD_{UB} = \{X, X, X\}$ is 33.

Table III summarizes the estimation results for the different execution orderings and methodologies; upper and lower bounds from the new methodology described here,

Fixed Dimension(s)	LB	UB	[4]	Exact BC/WC
FD = {X,X,X}	6	33	60	6/33
FD = {k,X,X}	6	11	60	6/11
FD = {X,k,j}	14	14	60	14/14
FD = {X,j,X}	11	18	60	11/18
j not innermost among i and j	11	33	60	11/33
FD = {X,X,j}	6	14	60	6/14
FD = {X,i,j}	6	6	60	6/6
FD = {k,i,j}	6	6	60	6/6

TABLE III
DEPENDENCY SIZE ESTIMATES OF CODE EXAMPLE IN FIG. 1.

the methodology presented in [4], and manually calculated exact worst-case (WC) and best-case (BC) numbers. Since no orthogonalization is needed for this example, these numbers are identical with the estimates. For larger real-life code, the exact numbers can of course not be computed manually any longer since they require a full exploration of all alternative orderings. The table also includes the estimation results of a stepwise fixation of the optimal ordering; j innermost, i second innermost, and k outermost.

Assume now that the example code of Fig. 1 is extended with a third statement:

S.3 if (i > 1) C[i][j][k] = h(A[i-2][j][k]);

The new dependency has only one SD, the i dimension. As was discussed in section IV-A, the lowest dependency size is reached when the SDs are placed innermost. Table IV gives the estimation results of the dependency between S.1 and S.3 with no execution ordering fixed, with i innermost, and with j innermost. It also shows the size estimates of the dependency between S.1 and S.2 with i fixed innermost. The size penalty of fixing i innermost for this dependency is smaller (11-6=5) then that of fixing j innermost for the S.1-S.3 dependency (12-2=10). Using the new dependency size estimation algorithm, it is therefore already with such small amount of information available possible to concluded that the i dimension should be ordered innermost. As can be seen from column [4] in Table III and Table IV, not taking the execution ordering into account results in large overestimates. If a technique requiring a fully fixed execution ordering is used, N! alternatives have to be inspected where N is the number of unfixed dimensions. N can be large (>4) for real-life examples. Note that N is not the number of dimensions in the arrays, but the number of dimensions in the loop nests surrounding them. Even algorithms with only two- and three-dimensional arrays often contain nested loops up to six levels deep.

E. Six Dimensional Example

Assume a DP and DVP as in Fig. 9. The length of each DP and DVP dimension is selected so that the function *GuidingPrinciplesOK( )* in Fig. 6 will return TRUE regardless of the placement of partly fixed dimensions. It is hence always possible to use automated dependency size estimation.

Dependency	Fixed innermost	LB	UB	[4]	Exact BC/WC
S.1 → S.3	None	2	36	90	2/36
S.1 → S.3	i	2	2	90	2/2
S.1 → S.3	j	12	36	90	12/36
S.1 → S.2	i	11	31	60	11/31

TABLE IV
DEPENDENCY SIZE ESTIMATES OF EXTENDED CODE EXAMPLE OF FIG. 1.

$$DP\begin{bmatrix}u\\v\\w\\x\\y\\z\end{bmatrix}=\begin{bmatrix}1&0&0&0&0&0\\0&1&0&0&0&0\\0&0&1&0&0&0\\0&0&0&1&0&0\\0&0&0&0&1&0\\0&0&0&0&0&1\end{bmatrix}\begin{bmatrix}i\\j\\k\\l\\m\\n\end{bmatrix}\left|\begin{bmatrix}1&0&0&0&0&0\\-1&0&0&0&0&0\\0&1&0&0&0&0\\0&-1&0&0&0&0\\0&0&1&0&0&0\\0&0&-1&0&0&0\\0&0&0&1&0&0\\0&0&0&-1&0&0\\0&0&0&0&1&0\\0&0&0&0&-1&0\\0&0&0&0&0&1\\0&0&0&0&0&-1\end{bmatrix}\begin{bmatrix}i\\j\\k\\l\\m\\n\end{bmatrix}\geq\begin{bmatrix}0\\-15\\0\\-15\\0\\-5\\0\\-7\\0\\-3\\0\\-15\end{bmatrix}\right.$$

$$DVP\begin{bmatrix}u\\v\\w\\x\\y\\z\end{bmatrix}=\begin{bmatrix}1&0&0&0&0&0\\0&1&0&0&0&0\\0&0&1&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\end{bmatrix}\begin{bmatrix}i\\j\\k\\l\\m\\n\end{bmatrix}\left|\begin{bmatrix}1&0&0&0&0&0\\-1&0&0&0&0&0\\0&1&0&0&0&0\\0&-1&0&0&0&0\\0&0&1&0&0&0\\0&0&-1&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\end{bmatrix}\begin{bmatrix}i\\j\\k\\l\\m\\n\end{bmatrix}\geq\begin{bmatrix}0\\-2\\0\\-1\\0\\-2\\0\\0\\0\\0\\0\\0\end{bmatrix}\right.$$

Fig. 9. Dependency Part and Dependency Vector Polytope for the six dimensional example.

Table V summarizes the estimation results for a number of partially fixed execution orderings. Note in particular that the estimates converge to the same number when the ordering is fully fixed.

V. ESTIMATION ON REAL LIFE APPLICATION DEMONSTRATORS

In this section the usefulness and feasibility of the 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.

Fixed Dimension(s)	LB	UB
FD = {X,X,X,X,X,X}	110	279040
FD = {i,X,X,X,X,l}	98368	115200
FD = {X,m,k,X,X,X}	4114	102528
FD = {j,m,l,n,i,k}	49166	49166
j not outermost among i, j, and k	200	279040

TABLE V
DEPENDENCY SIZE ESTIMATES OF SIX DIMENSIONAL EXAMPLE.

A. MPEG-4 motion estimation kernel

A.1 Code description and external constraints

MPEG-4 is a standard from the *Moving Picture Experts Group* for the format of multi-media data-streams in which audio and video objects can be used and presented in a highly flexible manner [34], [45]. An important part of the coding of this data-stream is the motion estimation of moving objects. See [7] for a more detailed description of the motion estimation part of the standard. This real-life application will now be used to show how storage requirement estimation can be used during the design trajectory. A part of the code where major arrays are produced and consumed is given in Fig. 10.

The sad-array is the only one both produced and consumed within the boundaries of the loop nest. It is produced by instruction S.1, S.2, and S.3, and consumed by instructions S.2, S.3, and S.4. For this example, the dependency analysis technique presented in [4] is utilized. This results in a number of basic sets and their dependencies as shown in the corresponding data-flow graph in Fig. 11. These basic sets can then be placed in a common iteration space.

```
for (y_s=0; y_s<=31; y_s++)
  for (x_s=0; x_s<=31; x_s++)
    for (y_p=0; y_p<=15; y_p++)
      for (x_p=0; x_p<=15; x_p++)
S.1     if ((x_p == 0)&(y_p == 0)) sad[y_s][x_s][y_p][x_p]=
            f1(curr[y_p][x_p], prev[y_s+y_p][x_s+x_p]);
S.2     else if ((x_p == 0)&(y_p != 0)) sad[y_s][x_s][y_p][x_p]=
                        f2(sad[y_s][x_s][y_p-1][15], curr[y_p][x_p],
                                                prev[y_s+y_p][x_s+x_p]);
S.3     else sad[y_s][x_s][y_p][x_p]=
                        f3(sad[y_s][x_s][y_p][x_p-1], curr[y_p][x_p],
                                                prev[y_s+y_p][x_s+x_p]);

for (y_s=0; y_s<=31; y_s++)
  for (x_s=0; x_s<=31; x_s++)
S.4   result[y_s][x_s] = f4(sad[y_s][x_s][15][15]);
```

Fig. 10. MPEG-4 motion estimation kernel.

For the remainder of the MPEG-4 example, some typical examples of external constraints are assumed. The curr[y_p][x_p] pixels are presented sequentially and row first (curr[0][0], curr[0][1], ... curr[0][15], curr[1][0] ...) at the input. The prev-array is already stored in local memory from the previous calculation and does not need to be taken into account. There are no output constraints, so the result can be presented at the output in any order.

A.2 Storage Size Estimation for Arrays

Due to the input constraint, the storage requirement for the curr-array can be made as small as possible if *y_p* is fixed outermost and *x_p* second outermost. Indeed, all calculations for each pixel can be completed before the next pixel arrives, and only one storage location is needed for the curr-array. In any of the other implementations, the curr-array elements must be buffered, since they are needed in bursts. This will require up to 16x16=256 storage locations. As a first design step, it is therefore natural to investigate the storage requirement of the sad-array dependencies given that the execution order is optimized for the curr-array. *y_p* is then first fixed at the outermost nest level. Applying the algorithm in Fig. 6 on each dependency with an FD= {y_p,X,X,X} we get estimation results as listed in Table VI. The table also lists the estimation results without any execution ordering fixed and the Spanning Dimensions (SDs) for each dependency. For the two dependencies with *y_p* and *x_p* as SDs, *y_p* is an SD due to boundary conditions causing negative dependencies.

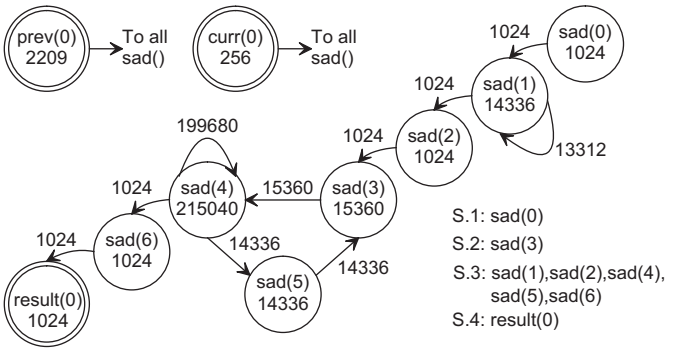


Fig. 11. Data-flow graph for MPEG-4 motion estimation kernel.

		FD={X,X,X,X}		FD={y_p,X,X,X}	
	SD	LB	UB	LB	UB
sad(0) → sad(1)	x_p	1	1024	1	1024
sad(1) → sad(1)	x_p	1	1024	1	1024
sad(1) → sad(2)	x_p	1	1024	1	1024
sad(2) → sad(3)	y_p, x_p	1	1024	1024	1024
sad(3) → sad(4)	x_p	1	15360	1	1024
sad(4) → sad(4)	x_p	1	15360	1	1024
sad(4) → sad(5)	x_p	1	14336	1	1024
sad(5) → sad(3)	y_p, x_p	1	1024	1024	1024
sad(4) → sad(6)	x_p	1	1024	1	1024
sad(6) → result(0)	-	1	1	1	1
curr(0)		1	256	1	256

TABLE VI
ESTIMATION RESULTS FOR THE SAD-ARRAY AND CURR-ARRAY OF THE MPEG-4 MOTION ESTIMATION KERNEL.

The maximal increase in the Lower Bound (LB) on the storage requirement for a dependencies with *y_p* as the outermost loop (1024 - 1 = 1023), exceeds the storage requirement for the entire curr-array (256). Consequently, it is possible to rule out any ordering with *y_p* as the outermost loop, since the lower bound storage requirement with no ordering constraints indicates that a better solution exists.

Focusing now on the size of the sad-array, we see from Table VI that *y_p* and *x_p* are the only Spanning Dimension (SD) for any of the dependencies. Following the ordering principles of Section IV-A they should hence be fixed innermost. Ignoring for the moment ordering constraints implied by the negative dependencies in the code, we use the algorithm of Fig. 6 to estimate the sizes of each dependency with *y_p* and *x_p* fixed innermost respectively. The results are given in Table VII. As can be seen, the minimum storage requirement can be achieved with *x_p* fixed innermost. This is also in accordance with constraints from

		FD={X,X,X,y_p}		FD={X,X,X,x_p}	
	SD	LB	UB	LB	UB
sad(0) → sad(1)	x_p	1	1024	1	1
sad(1) → sad(1)	x_p	1	1024	1	1
sad(1) → sad(2)	x_p	1	1024	1	1
sad(2) → sad(3)	y_p, x_p	1	1	1	1024
sad(3) → sad(4)	x_p	15	15360	1	1
sad(4) → sad(4)	x_p	15	15360	1	1
sad(4) → sad(5)	x_p	14	14336	1	1
sad(5) → sad(3)	y_p, x_p	1	1	1	1024
sad(4) → sad(6)	x_p	1	1024	1	1
sad(6) → result(0)	-	1	1	1	1
curr(0)		256	256	256	256

TABLE VII

ESTIMATION RESULTS FOR THE SAD-ARRAY AND CURR-ARRAY WITH *y_p* AND *x_p* INNERMOST RESPECTIVELY.

the negative dependencies.

Estimates with a fully fixed ordering of FD={y_s,x_p,y_p, x_p} result in converging upper and lower bounds of 1 for each individual dependency. Using the techniques presented in [26] for the final step of the overall estimation methodology, we find that non of the dependencies are alive simultaneously with this ordering. Using this for the simultaneously alive columns, Fig. 12 summarizes estimated upper and lower bounds for a stepwise fixation of the dimensions. The leftmost column shows the combined declared size of the sad- and curr-arrays. This is the memory size (262400) the designer would have to assume if no estimation and optimization tools were available. The next column shows the result found using the estimation technique described in [4] (45296). This result will be the same independently of the execution ordering. Finally the columns marked a) through e) show the size estimates found for a number of partially fixed execution orderings using the methodology presented in this paper. Note that the lower bound reported in a) only considers the combined size of dependencies with individually optimal orderings. The conflicting optimal ordering of dimension *y_p* is hence not taken into account. The consequences of fixing this dimension outermost are revealed in column b). Using the guiding principles and feedback from the estimation tool, the designer is finally able to reach a storage requirement of 257 when the full execution ordering is fixed as indicated in e).

B. SVD updating algorithm for beamforming

The results from the storage requirement estimation can also be used for feedback during interactive or tool driven global loop reorganization, [11]. This will now be demonstrated using the *Updating Singular Value Decomposition* (USVD) algorithm [33] for instance used in beamforming for antenna systems. The algorithm continuously updates matrix decompositions as new rows are appended to a matrix. Fig. 13 shows the two major arrays, *R* and *V*, and the important loop nest 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.

```
    for (k=0; k<=n-2; k++){
S.1   theta[k] = f1( R[...][...][2*k] );
S.2   phi[k] = f2( R[...][...][2*k], theta[k] ) ;
      for (i=0; i<=n-1; i++)
        for (j=0; j<=n-1; j++)
          ...
S.3       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++) {
          ...
S.4       else R[i][j][2*k+2] = f4( R[i][j][2*k+1], phi[k] );
          ...
S.5       else V[i][j][k+1] = f5( V[i][j][k], phi[k] );
      } }
```

Fig. 13. USVD algorithm, diagonalization loop nest

	S.1→ S.3	S.2→ S.5	S.3→ S.4	S.4→ S.3	S.5→ S.5
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 VIII

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.

Table VIII a) shows estimation results for a number of the 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. However, due to data dependencies not explicitly shown in Fig. 13, the *k* dimension must be placed outermost during the production of the *R*-array. Table VIII 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 S.1 → S.3, S.2 → S.5, S.4 → S.3, and S.5 → S.5 are alive simultaneously. This gives rise to a storage requirement of 202 memory locations, as shown in Fig. 14.

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 have 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 ordering for these dependencies. The current version of the methodology and tool does not indicate where such a loop body split should be performed apart from reporting that a given enforced ordering results in a large increase in the storage requirement for certain dependencies. This is however among our topics for future work. 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 avail-

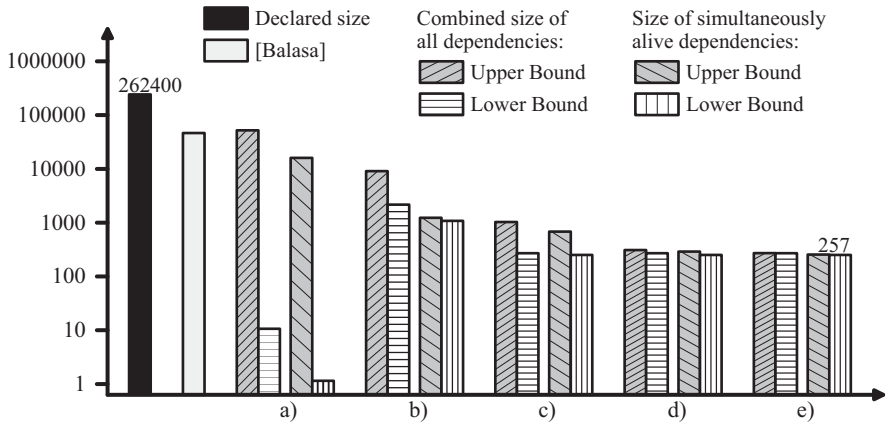


Fig. 12. Estimation results for MPEG-4 motion estimation using various techniques and partially fixed execution orderings. a) No ordering, b) y_p outermost, c) y_s outermost, d) x_s second outermost, e) y_p third outermost (fully fixed).

able. A comparison of the resulting storage requirement for an alternative loop organization, where a loop body splitting places the V-array in a separate loop nest, is therefore needed. Table VIII 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 Table VIII d) shows the estimation results for a fully fixed legal ordering with loop body split.

Due to the loop body split, dependencies S.4 → S.3 and S.5 → S.5 are not alive simultaneously anymore. The global storage requirement is hence approximately halved to 110 memory locations as shown in Fig. 14. The V-array is placed alongside the R-array, since they are not alive simultaneously. The data dependency size estimates thus guides the designer through the early steps of the design trajectory towards a solution with low storage requirements.

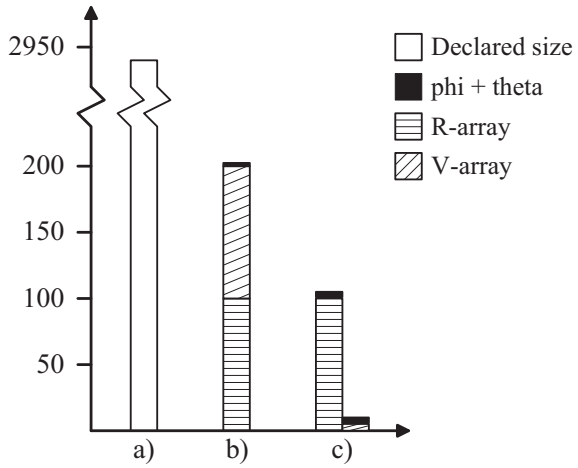


Fig. 14. Estimated storage requirement for the USVD algorithm: a) Declared size, b) without loop body split, c) with loop body split (n=10).

VI. Conclusions

We have presented a formalized algorithm for size estimation of individual data dependencies. The methodology is supported with a prototype CAD tool. Using representative data intensive applications we have shown how it can be used for storage requirement estimation and optimization during the early system design steps. Compared to previous work in the field, the technique allows a partially fixed execution ordering and produces upper and lower bounds on the storage requirement. As the execution ordering is gradually fixed, the upper and lower bounds on the data dependencies converge. This is a very useful and unique property of the methodology.

Acknowledgments

The work in this paper was supported in part by the Norwegian Research Council through research project 131359 CoDeVer.

References

[1] —, Atomium web site `http://www.imec.be/atomium/`.
[2] —, The International Technology Roadmap for Semiconductors web site, 2000 Update, http://public.itrs.net/
[3] —, "MATLAB, The Language of Technical Computing, Using MATLAB Version 5", The Math Works Inc., Natick, MA, USA, Jan. 1999
[4] F.Balasa, F.Catthoor, H.De Man, "Background Memory Area Estimation for Multi-dimensional Signal Processing Systems", *IEEE Trans. on VLSI Systems*, Vol.3, No.2, pp.157-172, June 1995.
[5] U.Banerjee, "Dependence Analysis for Supercomputing", Kluwer Acad. Publ., Boston, 1988.
[6] J.Bormans, K.Denolf, S.Wuytack, L.Nachtergaele and I.Bolsens, "Integrating System-Level Low Power Methodologies into a Real-Life Design Flow", *Proc. IEEE Wsh. on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Kos, Greece, pp.19-28, Oct. 1999.
[7] E.Brockmeyer, L.Nachtergaele, F.Catthoor, J.Bormans, H.De Man, "Low power memory storage and transfer organization for the MPEG-4 full pel motion estimation on a multi media processor", *IEEE Trans. on Multi-Media*, Vol.1, No.2, pp.202-216, June 1999.
[8] F.Catthoor, S.Wuytack, E.De Greef, F.Balasa, L.Nachtergaele, A.Vandecappelle, "Custom Memory Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design", ISBN 0-7923-8288-9, Kluwer Acad. Publ., Boston, 1998.
[9] F.Catthoor, K.Danckaert, C.Kulkarni, E.Brockmeyer, P.G.Kjeldsberg, T. Van Achteren, and T.Omnes, "Data Access and Storage Management for Embedded Programmable Processors", ISBN 0-7923-7689-7, Kluwer Acad. Publ., Boston, 2002.
[10] C.Chakrabarti, "Cache design and exploration for low power embedded systems", *Proc. Intnl. Conf. on Performance, Computing, and Communications*, pp.135 -139, 2001.

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

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

[13] K.Danckaert, "Loop transformations for data transfer and storage reduction on multiprocessor systems", *Doctoral dissertation*, ESAT/EE Dept., K.U.Leuven, Belgium, May. 2001.

[14] E.De Greef, "Storage size reduction for multimedia applications", *Doctoral dissertation*, ESAT/EE Dept., K.U.Leuven, Belgium, Jan. 1998.

[15] E.De Greef, F.Catthoor, H.De Man, "Array Placement for Storage Size Reduction in Embedded Multimedia Systems", *Proc. Intnl. Conf. on Applic.-Spec. Array Processors*, Zurich, Switzerland, pp.66-75, July 1997.

[16] P.Feautrier, "Dataflow analysis of array and scalar references", *Intnl. J. of Parallel Programming*, Vol.20, No.1, pp.23-53, 1991.

[17] D.Gajski, F.Vahid, S.Narayan, J.Gong, "Specification and design of embedded systems", Prentice Hall, Englewood Cliffs NJ, 1994.

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

[19] R.Gonzalez, M.Horowitz, "Energy dissipation in general-purpose microprocessors", *IEEE J. of Solid-state Circ.*, Vol.SC-31, No.9, pp.1277-1283, Sep. 1996.

[20] P.Grun, F.Balasa, and N.Dutt, "Memory Size Estimation for Multimedia Applications", *Proc. ACM/IEEE Wsh. on Hardware/Software Co-Design (Codes)*, Seattle WA, pp.145-149, March 1998.

[21] K-W. Kim, K-H. Baek, N. Shanbhag, C.L. Liu, S-M. Kang, "Coupling-Driven Signal Encoding Scheme for Low-Power Interface Design", *Proc. IEEE Int. Conf. Comp. Aided Design*, San Jose, USA, pp.318-321, Nov. 2000.

[22] D. Kirovski, C. Lee, M. Potkonjak, W. H. Mangione-Smith, "Application-driven synthesis of memory-intensive systems-on-chip" *IEEE Trans. on Comp.-aided Design*, Vol.CAD-18, No.9, pp.1316-1326, Sept. 1999.

[23] P.G.Kjeldsberg, F.Catthoor, E.J.Aas, "Storage requirement estimation for data-intensive applications with partially fixed execution ordering", *Proc. ACM/IEEE Wsh. on Hardware/Software Co-Design (Codes)*, San Diego CA, pp.56-60, May 2000.

[24] P.G.Kjeldsberg, F.Catthoor, E.J.Aas, "Application of high-level memory size estimation for guidance of loop transformations in multimedia design", *Proc. Nordic Signal Proc. Symp.*, Norrkvping, Sweden, pp.371-374, June 2000.

[25] P.G.Kjeldsberg, F.Catthoor, E.J.Aas, "Automated data dependency size estimation with a partially fixed execution ordering", *Proc. IEEE Intnl. Conf. on Comp. Aided Design*, Santa Clara CA, pp.44-50, Nov. 2000.

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

[27] P.G.Kjeldsberg, "Storage requirement estimation and optimisation for data-intensive applications", *Doctoral dissertation*, Norwegian Univ. of Science and Technology, Trondheim, Norway, March 2001.

[28] D.Kulkarni, M.Stumm, "Loop and Data Transformations: A Tutorial", Technical Report CSRI-337, Computer Systems Research Inst., Univ. of Toronto, pp.1-53, June 1993.

[29] F.J.Kurdahi, A.C.Parker, "REAL: a program for register allocation", *Proc. 24th ACM/IEEE Design Automation Conf.*, Miami FL, pp.210-215, June 1987.

[30] V.Lefebvre, P.Feautrier, "Optimizing storage size for static control programs in automatic parallelizers", *Proc. EuroPar Conf.*, Passau, Germany, Aug. 1997. "Lecture notes in computer science" series, Springer Verlag, Vol.1300, 1997.

[31] C.Mead, L.Conway, "Introduction to VLSI systems", Addison-Wesley, 1980.

[32] R.Gonzalez, M.Horowitz, "Energy dissipation in general-purpose microprocessors", *IEEE J. of Solid-state Circ.*, Vol.SC-31, No.9, pp.1277-1283, Sep. 1996.

[33] M.Moonen, P.Van Dooren, J.Vandewalle, "An SVD updating algorithm for subspace tracking", *SIAM J. Matrix Anal. Appl.*, Vol.13, No.4, pp.1015-1038, 1992.

[34] —, The ISO/IEC Moving Picture Experts Group Home Page, http://www.cselt.it/mpeg/

[35] S.Y.Ohm, F.J.Kurdahi, N.Dutt, "Comprehensive lower bound estimation from behavioral descriptions", *IEEE/ACM Intnl. Conf. on Computer-Aided Design*, pp.182-7, 1994.

[36] P.R.Panda, N.D.Dutt, A.Nicolau, "Local memory exploration and optimization in embedded systems", *IEEE Trans. on Comp.-aided Design*, Vol.CAD-18, No.1, pp.3-13, Jan. 1999.

[37] P.R.Panda, F.Catthoor, N.D.Dutt, K.Danckaert, E.Brockmeyer, C.Kulkarni, A.Vandercappelle, and P.G.Kjeldsberg, "Data and Memory Optimization Techniques for Embedded Systems", *ACM Trans. on Design Automation of Electronic Systems*, Vol.6, No.2, pp.149-206, April 2001.

[38] D.Patterson, T.Anderson, N.Cardwell, R.Fromm, K.Keeton, C.Kozyrakis, R.Thomas, K.Yelick, "A case for intelligent RAM", *IEEE-Micro*, Vol.17, No.2, pp.34-44, March-April 1997.

[39] P.G.Paulin, J.P.Knight, "Force-directed scheduling for the behavioral synthesis of ASICs", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*,Vol.8, No.6, pp.661-679, June 1989.

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

[41] F.Quillere, S.Rajopadhye, "Optimizing memory usage in the polyhedral model", *ACM Trans. on Programming Languages and Systems*, Vol.22, No.5, pp.773-815, Sept. 2000.

[42] J.Ramanujam, J.Hong, M.Kandemir, A.Narayan, "Reducing memory requirements of nested loops for embedded systems", *38th ACM/IEEE Design Automation Conf.*, Las Vegas NV, pp.359-364, June 2001.

[43] K.H.Rosen, "Discrete Mathematics and its Applications", McGraw-Hill, Inc., New York, USA, 1995 (Third edition).

[44] W.Shang, E.Hodzic, Z.Chen, "On uniformization of affine dependence algorithms", *IEEE Trans. on Computers*, Vol.45, No.7, pp.827-839, July 1996.

[45] T.Sikora, "The MPEG-4 video standard verification model", *IEEE Trans. on Circuits and Systems for Video Technology*, Vol.7, No.1, pp.19–31, Feb. 1997.

[46] A. Smailagic (guest editor) "Special Issue on System Level Design", *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, Vol.9, No.6, Dec. 2001.

[47] P.P. Sotiriadis and A. Chandrakasan, "Bus Energy Minimization by Transition Pattern Coding (TPC) in Deep Sub- Micron Technologies", *Proc. IEEE Int. Conf. Comp. Aided Design*, San Jose, USA, pp.322-327, Nov. 2000.

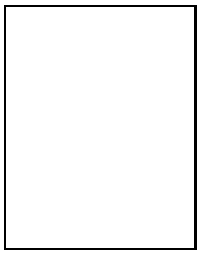
[48] C-J.Tseng, D.Siewiorek, "Automated synthesis of data paths in digital systems", *IEEE Trans. on Comp.-aided Design*, Vol.CAD-5, No.3, pp.379-395, July 1986.

[49] I.Verbauwhede, C.Scheers, J.Rabaey, "Memory estimation for high-level synthesis", *Proc. 31st ACM/IEEE Design Automation Conf.*, San Diego, CA, pp.143-148, June 1994.

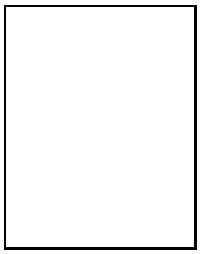
[50] I.Verbauwhede, F.Catthoor, J.Vandewalle, H.De Man, "Background memory management for the synthesis of algebraic algorithms on multi-processor DSP chips", *Proc. VLSI'89, Intnl. Conf. on VLSI*, Munich, Germany, pp.209-218, Aug. 1989.

[51] S.Wuytack, J.P.Diguet, F.Catthoor, H.De Man, "Formalized methodology for data reuse exploration for low-power hierarchical memory mappings", *IEEE Trans. on VLSI Systems*, Vol.6, No.4, pp.529-537, Dec. 1998.

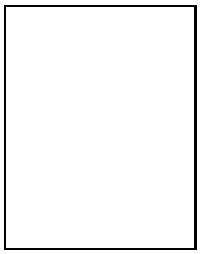
[52] Y.Zhao and S.Malik, "Exact Memory Size Estimation for Array Computation without Loop Unrolling", *36th ACM/IEEE Design Automation Conf.*, New Orleans LA, pp.811-816, June 1999.



Per Gunnar Kjeldsberg (S'98-M'01) received his M.Sc. in electrical engineering in 1992 from the Norwegian Institute of Technology in Trondheim, Norway. In 2001 he received the Ph.D. degree from the same place (now Norwegian University of Science and Technology, NTNU). Between 1992 and 1996 he worked as design engineer at Eidsvoll Electronics AS. During his doctoral studies, he focused on storage requirement estimation and optimization for data intensive applications. Kjeldsberg has published a number of conference and journal papers, and has been coauthor of a book in his field of interest. Currently he is Associate Professor at NTNU, focusing on hw/sw codesign and system level design in general, and on data transfer and storage exploration in particular.



Francky Catthoor (S'86-M'87-SM'98) received the engineering degree and a Ph.D. in electrical engineering from the Katholieke Universiteit Leuven, Belgium in 1982 and 1987 respectively. Since 1987, he has headed several research domains in the area of high-level and system synthesis techniques and architectural methodologies, all within the Design Technology for Integrated Information and Telecom Systems (DESICS - formerly VSDM) division at the Inter-university Micro-Electronics Center (IMEC), Heverlee, Belgium. He is part-time full professor at the EE department of the K.U.Leuven. In 1986 he received the Young Scientist Award from the Marconi International Fellowship Council. He has been associate editor for several IEEE and ACM journals, like Trans. on VLSI Signal Processing, Trans. on Multi- media, and ACM TODAES. He was the program chair of several conferences including ISSS'97 and SIPS'01.



Einar J. Aas (M'87) received the M.Sc. and Ph.D. degrees in electrical engineering from the Norwegian Institute of Technology in 1968 and 1972, respectively. He worked for the research organization ELAB, SINTEF for nine years before joining the university as professor in 1981. At ELAB, he worked with R&D projects for government and industry, including CAD systems for electronic design and layout. His current research interests include VLSI design methodology, in particular design quality models and metrics, design synthesis and verification, design for testability including BIST, and probabilistic methods in testing. Dr. Aas has published more than 200 scientific and technical papers. He is member of the Norwegian Academy of Technological Sciences, and The Royal Norwegian Society of Sciences and Letters.