Automated Data Dependency Size Estimation with a Partially Fixed Execution Ordering

Per Gunnar Kjeldsberg[1], Francky Catthoor[2], and Einar J. Aas[1]

[1]Norwegian University of Science and Technology, Trondheim, Norway, {pgk|aas}@fysel.ntnu.no
[2]IMEC, Leuven, Belgium, also at Katholieke Universiteit Leuven, catthoor@imec.be

Abstract

For data dominated applications, the system level design trajectory should first focus on finding a good data transfer and storage solution. Since no realization details are available at this level, estimates are needed to guide the designer. 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. Previous work has either not taken execution ordering into account at all, resulting in large overestimates, or required a fully specified ordering which is usually not available at this high level. The usefulness of the methodology is illustrated on representative application demonstrators.

1. Introduction

Many integrated circuit systems, especially 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. During the system development process, the designer must hence concentrate first on exploring the data transfer and storage to achieve a cost optimized end product [4]. 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 help in choosing the best solution, we therefore need estimation techniques for the storage requirements, 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. 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. 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.

At the beginning of the design process, no information about the execution order is known, except what is given from the data dependencies between the instructions in the code. As the process progresses, the designer takes decisions that gradually fix the ordering, until the full execution ordering is known. To steer this process, estimates of the upper and lower bounds on the storage requirement are needed at each step, given the partially fixed execution ordering. In [8] this context and our high level approach was sketched without the inclusion of the actual way to automate it.

In this paper we present a CAD algorithm for estimation of these upper and lower bounds on the size of individual data dependencies in high level code, even when only a partial execution ordering is fixed. This information is then used to estimate the application's total storage requirement. Using representative application demonstrators, we illustrate the feasibility and usefulness of the methodology. At the end we present our conclusions and directions for further work.

2. Previous Work

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 considered a separate scalar. Through the use of 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 [9]. Techniques such as clique partitioning are also exploited to group variables that can be mapped together [12]. A good introduction to the scalar based storage unit estimation can be found in [6]. Common to all of them is that they break down when used for large multi dimensional arrays, due to the huge number of scalars present.

To overcome this shortcoming, 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 [13] a production time axis is used to find the maximum difference between the production and consumption time for any two depending instances, giving the storage

requirement for one array. The total storage requirement is the sum of the requirements for each array. Only in-place mapping internally to an array is considered, not the possibility of mapping arrays in-place of each other. In [7] 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. From this, a memory trace of upper and lower bounding rectangles as a function of time is found with the peak bounding rectangle indicating the total storage requirement. If the difference between the upper and lower bounds for this critical rectangle is too large, the corresponding loop is split into two and the estimation is rerun. In the worst-case situation a full loop unrolling is necessary to achieve a satisfactory estimate. [14] describes a methodology based on live variable analysis and integer point counting for intersection/union of mappings of parameterized polytopes. They show that it is only necessary to find the number of live variables for one instruction 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. All of these techniques require a fully fixed execution ordering.

In contrast to the methods described in the previous paragraph, the storage requirement estimation technique presented by Balasa et al. in [1] does not take execution ordering into account at all. They start 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

$$\{ x = T \bullet i + u \mid A \bullet i \geq b \}$$

where $x \in Z^m$ is the coordinate vector of an m-dimensional array, and $i \in Z^n$ is the vector of 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, which is really essential during the early exploration of the system level code transformations. In [8] the outline is given of an overall methodology that takes a partial execution ordering into account. In the following section we present a detailed CAD algorithm for finding upper and lower bounds on the dependency sizes of individual signals, given a partly fixed execution ordering. This supports the most crucial step in our new methodology.

3. Estimation with Partially Fixed Execution Ordering

3.1 Motivation and Context

The new estimation methodology employs the data flow graph generation presented in [1]. In this paper we mainly focus on data dependency size estimation however, which can be utilized for most polyhedral dependency descriptions. The main improvement compared to previous methods is the possibility to avoid overestimates by taking whatever execution ordering information available into account. Compared to estimations with a fully fixed execution ordering, a full exploration of all combinations of unspecified orderings is avoided. Instead precise upper and lower bounds are presented to the designer as guidance during the early high level design trajectory. Throughout the application code there may be conflicting dependency considerations, so an automatic tool is needed to lead the designer to a globally efficient solution.

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, see [5] and [11]. Also the resulting Dependency Parts, see below, must be orthogonal, or made orthogonal, as described in [8].

Let us take a look at the simple application code example shown in Figure 1. Two instructions, I.1 and I.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 instructions [2].

```
 for (i=0; i<=5; i++)
   for (j=0; j<=5; j++)
     for (k=0; k<=2; k++){
I.1    A[i][j][k] = f( in[i][j][k] );
I.2    if ((i > 0) & (j > 1)) B[i][j][k] = g( A[i-1][j-2][k] );
     }
```

Figure 1: Simple application code example in C

The loops around the instructions define an iteration space, [2], as shown in Figure 2. Each point within this space represents one execution of the instructions inside the loop nest. For our example, at each of these iteration points one A-array element and, when the *if clause* condition is true, one B-array element is produced. In general not all elements produced by one instruction are read by a depending instruction. A Dependency Part (DP) is therefore defined containing the iteration points for which elements are produced that are read by the depending instruction. A Dependency Vector (DV) is drawn from an iteration point in the DP producing an array element to the iteration point 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). For the DVP in Figure 2, i and j are SDs while k is ND. The DP and DVP can also be represented using an LBL description as shown in Figure 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 $DP_{I.1\text{-}I.2}$ 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*.

Using the dependency size estimation methodology from [1], the size of the dependency between I.1 and I.2 will be the number of lattice points in the full DP. 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 A-array elements produced by I.1 can thus potentially be mapped in-place of the A-array elements consumed by I.2. This overlap should therefore be removed from the dependency size. Figure 4 presents an alternative methodology for calculating upper and lower bounds on dependency sizes that among its features include the ability of taking these kinds of overlap into account. Using this algorithm, the resulting lower and upper bounds are 4 and 36 locations respectively as shown in Section 3.3.

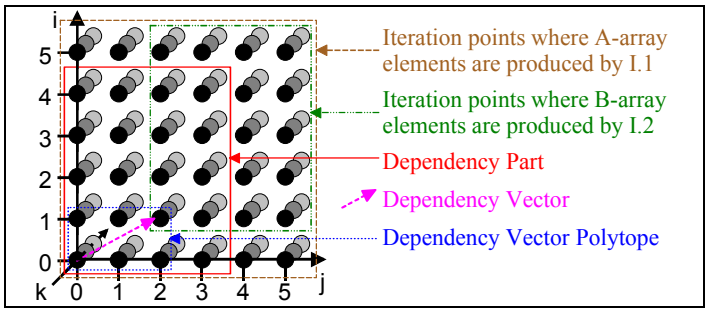

Figure 2: Iteration space with Dependency Part, Dependency Vector, and Dependency Vector Polytope

$DP_{I.1\text{-}I.2}$	$DP\begin{bmatrix}u\\v\\w\end{bmatrix} = \begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix} \left\| \begin{bmatrix}1&0&0\\-1&0&0\\0&1&0\\0&-1&0\\0&0&1\\0&0&-1\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix} \geq \begin{bmatrix}0\\-4\\0\\-3\\0\\-2\end{bmatrix} \right.$
$DVP_{I.1\text{-}I.2}$	$DVP\begin{bmatrix}u\\v\\w\end{bmatrix} = \begin{bmatrix}1&0&0\\0&1&0\\0&0&0\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix} \left\| \begin{bmatrix}1&0&0\\-1&0&0\\0&1&0\\0&-1&0\\0&0&0\\0&0&0\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix} \geq \begin{bmatrix}0\\-1\\0\\-2\\0\\0\end{bmatrix} \right.$

Figure 3: LBL descriptions for the DP and DVP of the example code in Figure 1

3.2 Algorithm for Estimation of Bounds

The input to the algorithm in Figure 4 consists of the LBL descriptions of the DP and DVP in addition to the ordered set of any Fixed Dimensions (FD). The dimensions can be specified starting with the outermost or innermost dimension and the algorithm is split into two fairly similar parts, each handling one of these two situations. If no dimensions are specified, that is the FD set is empty, any of the two parts can be used. Let us now take a closer look at the part of the code used when dimensions are specified starting with the outermost dimension. The fixed dimensions, if any, are dealt with first. Nonspanning dimensions are simply removed from the DP until the first spanning dimension is reached. The contribution to the Upper Bound (UB) from this spanning dimension is calculated through multiplication of all remaining dimensions and the length of the DVP in this dimension minus one. $|DVP_{d_i}|$ denotes the length of the DVP projected onto the d_i dimension. Similarly, $|DP_{d_i}|$ denotes the length of the DP projected onto the d_i dimension. If the spanning dimension is followed by nonspanning dimensions, they are again removed, until a new spanning dimension is reached in which case its contribution to the UB is calculated based upon itself and the remaining dimensions. This continues until all specified dimensions have been dealt with. So far the calculations are exact, so the Lower Bound (LB) equals the UB.

The remaining dimensions can be ordered in a best-case or worst-case sequence. The next step in the algorithm is to calculate their contribution to the lower bound. In the best-case ordering, the spanning dimensions are placed innermost and the remaining nonspanning dimensions can then be removed from the DP and ignored. The minimal combination of one spanning dimension extended to one lower than the DV and the rest extended to the border of the DP is a lower bound on their contribution to the total dependency size. This lower bound is not reachable unless only one spanning dimension is remaining or if no overlap exists between the DP and the depending basic set. This is however often the case, and even when this is not so, this small underestimate is the price we have to pay for using relatively simple calculations.

After having found the contribution of the remaining dimensions to the LB, the contribution to the UB is calculated. The worst case ordering of the spanning dimensions is to place them outside the remaining nonspanning dimensions. The upper bound on the unfixed dimensions' contribution to the total UB equals the size of the DP with all fixed dimensions removed. Any overlap between the DP and the depending basic set is also removed. It is calculated by placing all unfixed nonspanning dimensions innermost, and adding the contribution of each unfixed spanning dimension one at a time. When a spanning dimension has been treated, it is removed from the Spanning Dimension set (SD) and added to the Used Unfixed Spanning Dimensions set (UUSD). This ensures that no elements are counted more than once in the remaining calculations. Again the UB is only reachable when no overlap is present between the DP and the depending basic set or when only one unspecified spanning dimension remains.

The algorithmic principles for fixation starting with the innermost dimension is similar to the above description, except that nonspanning dimensions are added to the DVP instead of removed from the DP.

3.3 Illustration on Representative Example

Let us go back to the example code in Figure 1. If no execution ordering is specified, the FD set is empty, and we go directly to the second *for each* loop in the estimation algorithm in Figure 4. The lower bound on the cost of ordering one of the spanning dimensions outermost among the spanning dimensions, $d_{i_{LB}}$, is calculated for $d_i = i$ and $d_i = j$. The results are (2-1)*4 = 4 and (3-1)*5 = 10 respectively. The result for $d_i = i$ is consequently used and we get LB = 4. For the UB calculation in the third *for each* loop in Figure 1, the order of the spanning dimensions is of no consequence. If we choose first to calculate the contribution of $d_i = i$ and then that of $d_i = j$, the results are UB_{tmp} = (2-1)*4*3 = 12, and UB = UB_{tmp} + (3-1)*(5-(2-1))*3 = 36. A graphical description of the A-array elements that make up the upper and lower bounds is given in Figure 5.

Assume now that the *k*-dimension is specified as the

outermost dimension while the ordering between the *i*- and *j*-dimensions is still unresolved. In the first *for each* loop of the estimation algorithm, *k* is removed from the set of Nonspanning Dimensions (ND). During the subsequent calculation of the UB, the ND set is empty and the last product is removed from the equation. The UB calculation is therefore reduced to UB_{tmp} = (2-1)*4 = 4, and UB = UB_{tmp} + (3-1)*(5-(2-1)) = 12. The LB calculation already assumed nonspanning dimensions to be placed outermost. Consequently, placing *k* outermost gives an unchanged LB and a decreased value of UB.

```
void 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 UsedUnfixedSpanningDimensions (UUSD) = Ø
  define set FixedSpanningDimensions (FSD) = Ø
  define set NonspanningDimensions (ND) = (AD - SD)
  define set FixedNonspanningDimensions (FND) = Ø
  define value LowerBound (LB) = 0
  define value UpperBound (UB) = 0
  if fixation starts from outermost dimension {
    for each dimension d_i ∈ FD {
      if d_i ∈ ND            /* Remove Nonspanning Dimension from DP */
        ND = ND - d_i
      else {
        SD = SD - d_i      /* Calculate contribution of Spanning Dimension to UB */
        UB = UB + (|DVP_{d_i}| - 1) * ∏_{∀d_j∈SD} |DP_{d_j}| * ∏_{∀d_j∈ND} |DP_{d_j}|
    } }
    LB = UB                          /* Previous calculations are accurate */
    for each dimension d_i ∈ SD    /* Only unfixed spanning dimensions left  */
      d_{i_LB} = (|DVP_{d_i}| - 1) * ∏_{∀d_j∈SD−d_i} |DP_{d_j}|
    LB = LB + min( d_{i_LB} | d_i ∈ SD)    /* Lower bound on the contribution to the LB */
    for each dimension d_i ∈ SD {       /*Upper bound on the contribution  to the UB */
      SD = SD - d_i
      UB = UB + (|DVP_{d_i}| - 1) * ∏_{∀d_j∈UUSD} (|DP_{d_j}| - (|DVP_{d_j}| - 1)) * ∏_{∀d_j∈SD} |DP_{d_j}| * ∏_{∀d_j∈ND} |DP_{d_j}|
      UUSD = UUSD + d_i
  } }
  if fixation starts from innermost dimension {
    for each dimension d_i ∈ FD {
      if d_i ∈ ND              /* Add Nonspanning Dimension to the DVP */
        FND = FND + d_i
      else {                   /* Calculate contribution of Spanning Dimension to LB */
        SD = SD - d_i
        LB = LB + (|DVP_{d_i}| - 1) * ∏_{∀d_j∈FSD} |DP_{d_j}| * ∏_{∀d_j∈FND} |DP_{d_j}|
        FSD = FSD + d_i
    }}
    UB = LB                          /* Previous calculations are accurate */
    for each dimension d_i ∈ SD   /* Only unfixed spanning dimensions left */
      d_{i_LB} = (|DVP_{d_i}| - 1) * ∏_{∀d_j∈SD−d_i} |DP_{d_j}| * ∏_{∀d_j∈FSD} |DP_{d_j}| * ∏_{∀d_j∈FND} |DP_{d_j}|
    LB = LB + min( d_{i_LB} | d_i ∈ SD)   /* Lower bound on the contribution to the LB */
    for each dimension d_i ∈ SD {     /*Upper bound on the contribution to the UB */
      SD = SD - d_i
      UB = UB + (|DVP_{d_i}| - 1) * ∏_{∀d_j∈UUSD} (|DP_{d_j}| - (|DVP_{d_j}| - 1)) * ∏_{∀d_j∈SD} |DP_{d_j}| * ∏_{∀d_j∈FSD} |DP_{d_j}| * ∏_{∀d_j∈ND} |DP_{d_j}|
      UUSD = UUSD + d_i
  } }
  return UB, LB
```

Figure 4: Pseudo code for dependency size estimation algorithm

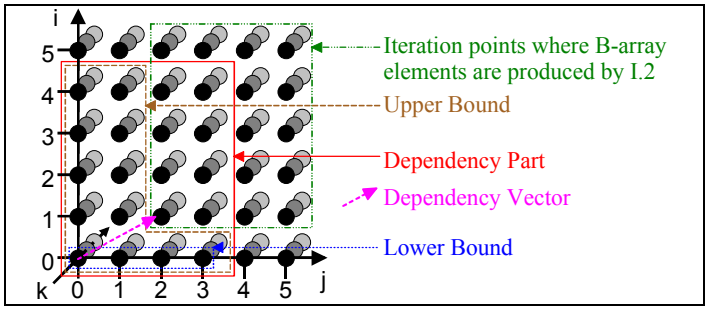

Figure 5: UB and LB with no execution ordering fixed.

The estimation algorithm can also be used when the execution ordering is specified starting with the innermost dimension. Assume a partly fixed ordering placing the *i*-dimension innermost and the *k*-dimension second innermost (this is actually a fully fixed ordering since only one dimensions is unspecified, but this does not change the computations). We now use the second part of the estimation algorithm; *fixation starts from innermost dimension*. In its first *for each* loop we enter the *else* clause and calculate the contribution of *i* to the LB. Since we so far have no fixed spanning or nonspanning dimension, FSD=FND=Ø, it is simply LB_{tmp} = (2-1) = 1. Afterwards, *i* is added to FSD. When the same *for each* loop is executed with $d_i = k$, *k* is added to the FND set since it is a nonspanning dimension. So far, the calculations are exact, and $UB_{tmp} = LB_{tmp}$. The second *for each* loop is then entered. Only one spanning dimension, *j*, is remaining, and its contribution to the LB is calculated and added to LB_{tmp}; LB = LB_{tmp}+ (3-1)*5*3 =31. The contribution of the *j*-dimension to the UB is calculated using the last *for each* loop of the estimation algorithm: UB = UB_{tmp} + (3-1)*5*3 = 31. The UB and LB converge, and as can be seen through a closer inspection of Figure 5, the estimated dependency size is fully accurate in this case.

Table 1 summarizes the estimation results for the different execution orderings and methodologies; upper and lower bounds from the new methodology described here, the methodology presented in [1], and manually calculated exact worst case (WC) and best case (BC) numbers. For larger real-life code, the exact numbers can of course not be computed any longer. The table also includes the results of a stepwise fixation of the optimal ordering; *j* innermost, *i* second innermost, and *k* outermost.

Fixed Dimension(s) Outermost Innermost	Lower bound	Upper bound	[1]	Exact BC/WC
None	4	36	60	6/33
k	4	12	60	6/11
k,i	31	31	60	31/31
j	6	14	60	6/14
i,j	6	6	60	6/6
k,i,j	6	6	60	6/6

Table 1: Dependency size estimates of code in Figure 1

Let us extend the example code of Figure 1 with a third instruction:

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

The new dependency has only one SD, the *i*-dimension. The corresponding DP and DVP are shown in Figure 6. As Table 1 indicates, the lowest dependency size is reached when the spanning dimensions are placed innermost. Table 2 gives the estimation results of the dependency between I.1 and I.3 with no execution ordering fixed, with *i* innermost, and with *j* innermost. It also shows the size estimates of the dependency between I.1 and I.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 I.1-I.3 dependency (12-2=10). Using our 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 [1] in Table 1 and Table 2, not taking the execution ordering into account results in large overestimates. If a technique requiring that 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.

$DP_{I.1\text{-}I.3}$	$B\begin{bmatrix}u\\v\\w\end{bmatrix} = \begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix} \left\| \begin{bmatrix}1&0&0\\-1&0&0\\0&1&0\\0&-1&0\\0&0&1\\0&0&-1\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix} \geq \begin{bmatrix}0\\-3\\0\\-5\\0\\-2\end{bmatrix} \right.$
$DVP_{I.1\text{-}I.3}$	$B\begin{bmatrix}u\\v\\w\end{bmatrix} = \begin{bmatrix}1&0&0\\0&0&0\\0&0&0\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix} \left\| \begin{bmatrix}1&0&0\\-1&0&0\\0&0&0\\0&0&0\\0&0&0\\0&0&0\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix} \geq \begin{bmatrix}0\\-2\\0\\0\\0\\0\end{bmatrix} \right.$

Figure 6: LBL descriptions for the DP and DVP of dependency between I.1 and I.3 in the extended example code of Figure 1

Dependency	Fixed innermost	Lower bound	Upper bound	[1]	Exact BC/WC
I.1 - I.3	None	2	36	72	2/36
I.1 - I.3	i	2	2	72	2/2
I.1 - I.3	j	12	36	72	12/36
I.1 - I.2	i	11	31	60	11/31

Table 2: Dependency size estimates of extended example of Figure 1

3.4 Global Estimation

The previous section explains how the size of individual data dependencies in the code can be estimated. This can be used directly to find the optimal execution ordering for dependencies within perfectly nested loops. For more complex dependencies between loop nests and in imperfectly nested loops, an advanced data dependency analysis, similar to the one described in [1], followed by a traversal of the resulting data flow graph, is needed. This is however outside the scope of this paper.

4. Estimation on Real-Life Application Demonstrators

4.1 MPEG-4 Motion Estimation Kernel

MPEG-4 is a standard for the format of multi-media data streams in which audio and video objects can be used and presented in a highly flexible manner. An important part of the coding of this data stream is the motion estimation of moving objects. See [3] for a more detailed description of this part of the standard. We will now use this real life application to demonstrate the effectiveness of the new dependency size estimation algorithm. A part of the code is given in Figure 7. The sad-array is the only one both produced and consumed within the boundaries of the loop nest. It is produced by instruction I.1, I.2, and I.3, and consumed by instruction I.2, I.3,

and I.4. We will use the estimation technique to determine the bounds on the size of the data dependencies between these instructions.

```
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++)
I.1     if ((x_p == 0)&(y_p == 0)) sad[y_s][x_s][y_p][x_p]=
                          f(curr[y_p][x_p], prev[y_s+y_p][x_s+x_p]);
I.2     else if ((x_p == 0)&(y_p != 0)) sad[y_s][x_s][y_p][x_p]=
                          g(sad[y_s][x_s][y_p-1][15], curr[y_p][x_p],
                                                  prev[y_s+y_p][x_s+x_p]);
I.3     else sad[y_s][x_s][y_p][x_p]=
                          g(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++)
I.4  result[y_s][x_s] = h(sad[y_s][x_s][15][15]);
```

Figure 7: MPEG-4 motion estimation kernel

As can be seen from the *if clauses* in Figure 7, by far the largest number of sad-array elements are produced and also consumed by instruction I.3. An estimation of the size of this data dependency is therefore an obvious starting point. Figure 8 gives an LBL description of the elements that are both produced and consumed by instruction I.3, that is the DP for the self dependency of instruction I.3. The corresponding Dependency Vector starts at (y_s,x_s,y_p,x_p)-point (0,0,0,1) and ends at (0,0,0,2). An LBL description of its DVP is given in Figure 9.

$$DP_{I.3-I.3}\begin{bmatrix} m \\ n \\ u \\ v \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y\_s \\ x\_s \\ y\_p \\ x\_p \end{bmatrix} \left\| \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} y\_s \\ x\_s \\ y\_p \\ x\_p \end{bmatrix} \geq \begin{bmatrix} 0 \\ -31 \\ 0 \\ -31 \\ 0 \\ -15 \\ 1 \\ -14 \end{bmatrix} \right.$$

Figure 8: LBL description of the DP containing elements both produced and consumed by instruction I.3

$$DVP_{I.3-I.3}\begin{bmatrix} m \\ n \\ u \\ v \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y\_s \\ x\_s \\ y\_p \\ x\_p \end{bmatrix} \left\| \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} y\_s \\ x\_s \\ y\_p \\ x\_p \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ -2 \end{bmatrix} \right.$$

Figure 9: LBL description of the DVP for the self dependency of instruction I.3

Without any execution ordering fixed and for the time being also ignoring any ordering implied by the data dependencies in the code, the lower and upper bounds on the size of this dependency can be calculated using the second and third *for each* loop in the estimation algorithm in Figure 4. The results are listed in the I.3→I.3 row of Table 3. Placing spanning dimensions innermost results in general in lower dependency sizes. Estimates of the bounds with the partly fixed execution ordering of having the spanning dimension *x_p* innermost are also listed in Table 3. Similar estimates can be performed for the other dependencies in the code. For the elements consumed by instruction I.2, *y_p* is the spanning dimension. Consequently we have possibly competing optimal orderings. Estimates with *y_p* and *x_p* placed innermost are therefore performed for all dependencies, as listed in Table 3.

dependency	No ordering	x_p innermost	y_p innermost
I.1→I.3	1/1024	1/1	1/1024
I.2→I.3	1/15360	1/1	15/15360
I.3→I.2	1/1024	1/1024	1/1
I.3→I.3	1/16384	1/1	16/16384
I.3→I.4	1/1	1/1	1/1

Table 3: Estimated LB/UB for sad array dependency sizes in MPEG-4 motion estimation kernel

Since the code example is written in single assignment form, no overlap can exist between partitions of a single array. This implies that the lower bound values are reachable. The ordering with *x_p* innermost is therefore optimal for the sad-array, since even the dependency between instructions I.3 and I.2 can have the size of one array element. This visualizes how the estimation technique can be exploited during the early phases of the design trajectory.

4.2 SVD Updating Algorithm

The Singular Value Decomposition (SVD) algorithm, [10], continuously updates matrix decompositions as new rows are appended to a matrix. Figure 10 shows the two major arrays and the important loop nest for production and consumption of their data elements. Instructions I.1, I.2, I.4, I.5, I.7, and I.8 all require two array accesses (numbered in order of appearance) for each array element produced, giving rise to two dependencies between these instructions and other (or the same) instructions.

```
for (l = 0; l < n-2; l++)
  {
  for (i = 0; i < n-1; i++)
    for (j = 0; j < n-1; j++)
I.1   if (i==l & j>=l) R[i][j][2*l+1] = f1(R[i][j][2*l], R[i+1][j][2*l]);
I.2   else if (i==l+1 & j>=l)
                       R[i][j][2*l+1] = f2(R[i-1][j][2*l], R[i][j][2*l]);
I.3   else             R[i][j][2*l+1] = f3( R[i][j][2*l] );

  for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
      {
I.4   if (j==l & i<=l+1)
            R[i][j][2*l+2] = f4( R[i][j][2*l+1], R[i][j+1][2*l+1] );
I.5   else if (j==l+1 & i<=l+1)
            R[i][j][2*l+2] = f5( R[i][j-1][2*l+1], R[i][j][2*l+1] );
I.6   else  R[i][j][2*l+2] = f6( R[i][j][2*l+1] );

I.7   if ( j==l )          V[i][j][l+1] = f7( V[i][j][l], V[i][j+1][l] );
I.8   else if ( j==l+1 )   V[i][j][l+1] = f8( V[i][j-1][l], V[i][j][l] );
I.9   else                 V[i][j][l+1] = f9( V[i][j][l] );
  }  }
```

Figure 10: USVD algorithm, diagonalization loop nest

With no execution ordering fixed and not taking the data dependencies in the code into account the lower and upper bounds on dependency sizes for the R- and V-array are estimated as shown in Table 4. Elements are produced by the instructions in the leftmost column and read by the array accesses to the right of the arrows. Note that the DPs have been orthogonalized. Assume now that an external constraint requires the *i*-dimension to be ordered innermost. This partially fixed execution ordering will change the dependency size estimates as shown in Table 5. The constraint results in a substantial increase

in the storage requirement. This is detected without having to investigate all alternative orderings with *i* as the innermost dimension.

→	I.4.1	I.4.2	I.5.1	I.5.2	I.6.1
I.1	1/1	1/1	1/1	1/1	1/8
I.2	1/1	1/1	1/1	1/1	1/8
I.3	1/8	1/8	1/8	1/8	1/100
→	I.1.1	I.1.2	I.2.1	I.2.2	I.3.1
I.4					1/9
I.5	1/1		1/1		1/8
I.6	1/8	1/9	1/8	1/9	1/100
→	I.7.1	I.7.2	I.8.1	I.8.2	I.9.1
I.7					1/10
I.8	1/10		1/10		
I.9		1/10		1/10	1/100

Table 4: Estimated LB/UB for dependency sizes in USVD algorithm with no execution ordering fixed (n=10)

I.3→I.4.1	I.3→I.4.2	I.3→I.5.1	I.3→I.5.2	I.3→I.6.1	I.4→I.3.1	I.5→I.3.1
8/8	8/8	8/8	8/8	10/100	9/9	8/8
I.6→I.3.1	I.7→I.9.1	I.8→I.7.1	I.8→I.8.1	I.9→I.7.2	I.9→I.8.2	I.9→I.9.1
10/100	10/10	10/10	10/10	10/10	10/10	10/100

Table 5: Changes in estimated LB/UB for dependency sizes in USVD algorithm with *i*-dimension fixed innermost (n=10)

Returning to Table 4 the upper bound on three dependencies are notably larger then the others; I.3→I.6.1, I.6→I.3.1, and I.9→I.9.1. For all of these, *l* is the only SD and should be placed innermost to minimize the storage requirement. Estimates with this partially fixed execution ordering result in converging lower and upper bounds of 1 for all three dependencies. Due to other data dependencies in the code, the R-array elements must be produced with *l* outermost resulting in converging lower and upper bounds size estimates of 100 for the I.3→I.6.1 and I.6→I.3.1 dependencies. Table 6 a) shows the estimation results for the V-array dependencies with *l* outermost. In this case however no dependencies enforce this partial ordering. Table 6 b) and c) present the estimation results for the gradual fixation of an alternative ordering. The reduced dependency sizes rationalize the generation of a separate loop nest for instruction I.7, I.8, and I.9, that is a loop body split. The example shows how the data dependency size estimates systematically guides the designer through the early steps of the design trajectory towards a solution with low storage requirements.

	I.7→I.9.1	I.8→I.7.1	I.8→I.8.1	I.9→I.7.2	I.9→I.8.2	I.9→I.9.1
a)	10/10	10/10	10/10	10/10	10/10	100/100
b)	1/1	1/1	1/1	1/1	1/1	1/10
c)	1/1	1/1	1/1	1/1	1/1	10/10

Table 6: Estimated LB/UB for dependency sizes for the V-array with a) *l* outermost, b) *i* outermost, and c) *i* outermost and *l* second outermost (n=10)

5. Conclusion

This paper presents a novel technique for estimation of individual data dependency sizes in high level application code. It differs from previous work in that it allows a partial fixed execution ordering, which will usually be the case during the early steps in the system design trajectory. Design examples have shown how strict upper and lower bounds are easily calculated avoiding the overestimate of methodologies that do not take execution ordering into account. The full exploration of all alternative orderings of the unfixed dimensions necessary for methodologies requiring a fully fixed execution ordering is also avoided. The upper and lower bounds have the very desirable property that they converge to a single value as the dimensions are becoming fully fixed.

Acknowledgement

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

References

[1] F. Balasa, F. Catthoor, and H. De Man, "Background memory area estimation for multidimensional signal processing systems", IEEE Trans. on VLSI Systems, Vol. 3, No. 2, June 1995, pp. 157-72

[2] U. Banerjee, "Dependency Analysis for Supercomputing", Kluwer Academic Publishers, Boston/Dordrecth/London, 1988

[3] E. Brockmeyer, L. Nachtergaele, F. Catthoor, J. Bormans, and H. De Man, "Low Power Memory Storage and Transfer Organization for the MPEG-4 Full Pel Motion Estimation on a Multimedia Processor", IEEE Trans. on Multimedia, Vol. 1, No. 2, June 1999, pp. 202-16

[4] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, "Custom Memory Management Methodology Exploration of Memory Organization for Embedded Multimedia Systems Design", Kluwer Academic Publishers, 1998

[5] P. Feautrier, "Dataflow analysis of array and scalar references", International Journal of Parallel Programming, Vol. 20, No. 1, Feb. 1991, pp. 23-52

[6] D.D. Gajski, F. Vahid, S. Narayan, and J. Gong, "Specification and Design of Embedded Systems", Prentice Hall, 1994

[7] P. Grun, F. Balasa, and N. Dutt, "Memory Size Estimation for Multimedia Applications", Proc. Sixth Int. Workshop on HW/SW Codesign (CODES/CACHE), March 1998, pp. 145-9

[8] P.G. Kjeldsberg, F. Catthoor, E.J. Aas, "Storage requirement estimation for data intensive applications with partially fixed execution ordering", Proc. Int. Workshop on HW/SW Co-Design, CODES 2000, San Diego, May 2000, pp. 56-60

[9] F.J. Kurdahi and A.C. Parker, "REAL: A Program for REgister ALlocation", Proc. 24th DAC, 1987, pp. 210-5

[10] M. Moonen, P. Van Dooren, J. Vandewalle, "An SVD updating algorithm for subspace tracking", SIAM Journal on Matrix Analysis and Applications, Vol. 13, No. 4, 1992, pp. 1015-1038

[11] W. Pugh and D. Wonnacott, "An exact method for analysis of value-based array data dependences", Proc. 6th International Workshop on Languages and Compilers for Parallel Computing, Portland, USA, Aug. 1993, pp. 546-66

[12] C-J. Tseng, and D.P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems", IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, Vol. 5, No. 3, July 86, pp. 379-95

[13] I.M. Verbauwhede, C.J. Scheers, J.M. Rabaey, "Memory Estimation for High Level Synthesis", Proc. 31st DAC, 1994, pp. 143-8

[14] Y. Zhao and S. Malik, "Exact Memory Size Estimation for Array Computation without Loop Unrolling", Proc 36th DAC, 1999, pp.811-6