

APPLICATION OF HIGH-LEVEL MEMORY SIZE ESTIMATION FOR GUIDANCE OF LOOP TRANSFORMATIONS IN MULTIMEDIA DESIGN

Per Gunnar Kjeldsberg¹, Francky Catthoor², and Einar J. Aas¹

¹Norwegian University of Science and Technology, Trondheim, Norway, pgk@fysel.ntnu.no

²IMEC, Leuven, Belgium, also at Kath. Univ. Leuven, catthoor@imec.be

ABSTRACT

In this paper, we demonstrate how a novel technique for high-level memory requirement estimation can be used in system level synthesis for data-dominated multimedia applications. Using a polyhedral description of partitioned arrays and their dependencies, guiding hints for the loop ordering are presented to the designer. Our key contribution consists of estimates on the upper and lower bounds of the memory size requirement with a partially fixed execution ordering. These are used in the early system design trajectory to find an implementation with low memory requirement. The methodology is demonstrated using a representative multimedia application.

1. INTRODUCTION

Many signal processing systems, especially in the multimedia 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 in more and more cases forms the main bottleneck (especially at the board level), 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 [3]. 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. Estimation techniques for the storage requirements are therefore needed, very early in the system design trajectory. A good design tool should also be able to give the designer hints that will guide the synthesis to achieve low storage requirements.

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 to find the size of each array by multiplying the size of each dimension, and then add together 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 more accurate estimates, we must 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 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 which gradually fix the ordering, until it is fully specified. To guide this process, hints of an optimal ordering and estimates of the upper and lower bounds on the storage requirement are needed at each step, given the partially fixed execution ordering.

In this paper we show how a novel technique for estimation of the storage requirements can be used to guide the design process for data intensive applications. The technique also gives the designer hints for a good execution ordering that can be synthesized into an implementation with low storage requirements. The rest of this paper starts with a presentation of previous work on storage requirement estimation. Section 3 focuses on how to use our new size estimation technique to guide the exploration of loop transformations in multimedia applications. Experimental results on a representative application demonstrator are discussed in section 4. At the end we present our conclusions.

2. PREVIOUS WORK

Most of the 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. A good introduction to the scalar-based storage unit estimation can be found in [4]. Common to all of the techniques 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 accesses 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. [5], [7], and [8] present different approaches given a fully fixed execution ordering. In contrast, the storage requirement estimation technique presented in [1] does not take execution ordering into account at all, except what is directly given by the data dependencies in the code. None of the approaches permit the designer to specify partial ordering constraints, which is really essential during the early exploration of the code transformations. In [6] we propose a new technique that allows partial ordering constraints. In the following section this is taken further, especially in the direction of system level synthesis.

3. SYSTEM LEVEL ESTIMATION AND LOOP TRANSFORMATION EXPLORATION

3.1 Estimation Methodology

The estimation methodology splits the arrays into suitable parts, denoted basic sets, which are represented using a polyhedral description. The dependencies between the basic sets are then detected, and a data-flow graph is generated where the basic sets are the nodes and the dependencies between them are the branches. Currently the partitioning into basic sets and the data-flow graph generation are based on [1], but also other detailed polyhedral descriptions of arrays and their dependencies can be

used. There are some restrictions to the code that can be handled, where the most important are the affine index and single assignment requirements. These can be partly overcome but we assume for the moment that the designer (re)writes the initial code in that form. This is anyway very beneficial for the freedom to apply the actual loop transformations [3]. The theoretical background for the new estimation methodology is explained in [6]. We will in this paper focus on how the methodology can be utilized for system level synthesis on a representative multi-media related example.

```
(i: 0 .. 15)::
(j: 0 .. 15)::
(k: 0 .. 7)::
begin
S.1 A[i][j][k] = f( in );
S.2 B[i][j][k] = if (j > 0) & (k > 1) ->
g( A[i][j-1][k-2] ) fi;
end;
```

Figure 1: Three-dimensional example (Silage language)

We will first present some important definitions using the code example of Figure 1. It shows a three-dimensional loop-nest with two arrays, $A[u][v][w]$ and $B[u][v][w]$, A and B for short. The loop iterators i , j , and k span a three-dimensional iterator space, and the array index functions u , v , and w define the points in this space for which an array element is produced or read. Array A is produced by statement S.1 and partly consumed while array B is produced in statement S.2. The rest of array A is used elsewhere in the code not shown here. The part of array A consumed by statement S.2, denoted basic set $A(0)$, can be described as a Linearly Bounded Lattice (LBL) as shown in Figure 2. Correspondingly, the LBL for array B produced by statement S.2, basic set $B(0)$, is shown in Figure 3. The vertical line in the LBLs divide the description in an array index function and a restriction part. The restricted function for the u index of $A(0)$ can thus be read as $u = i$ "for" $0 \leq i \leq 15$. Together with the dependency between them, $A(0)$ and $B(0)$ make out the main part of the data-flow graph of Figure 4.

$$A(0) \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \left| \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 0 \\ -15 \\ 0 \\ -14 \\ 0 \\ -5 \end{bmatrix} \right.$$

Figure 2: LBL description for basic set $A(0)$

$$B(0) \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \left| \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 0 \\ -15 \\ 1 \\ -15 \\ 2 \\ -7 \end{bmatrix} \right.$$

Figure 3: LBL description for basic set $B(0)$

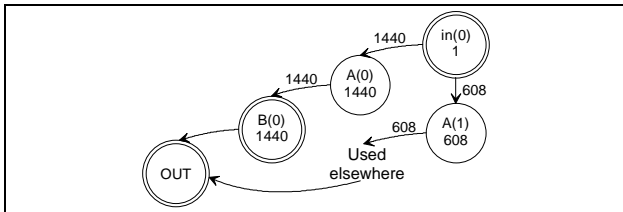


Figure 4: Data-flow graph for code example in Figure 1.

The part of $A(0)$ that $B(0)$ depends on, in this case the whole of $A(0)$, is identified as the *Dependency Part* (DP) of $A(0)$ in respect of $B(0)$. A *Dependency Vector* (DV) is defined from the smallest element of $A(0)$ to the depending element of $B(0)$, in this case from (i,j,k) -point $(0,0,0)$ to $(0,1,2)$. This vector spans a *Dependency Vector Polytope* (DVP), as shown in Figure 5. For comparison with the other LBL descriptions, it is written out

with three dimensions, even though it only has two dimensions, j and k . These dimensions are defined as *spanning dimensions* since they span the DVP, while other dimensions (only i in our example) are *non-spanning dimensions*. For each spanning dimension we define the *spanning value* (SV) as the end point of the DV for this dimension ($SV_{j=1}$, $SV_{k=2}$).

The storage requirement for the code equals the maximum number of simultaneously alive array elements. This number can be found through a data-flow graph traversal, where a node can be produced when all its ancestors are produced and consumed when its last child is produced. The sizes of dependencies between nodes are decided by the execution ordering and equal the number of elements produced in one node before the first element in the depending node is produced. The size of a node is the sum of its outgoing dependencies. With no ordering specified for the example in Figure 1, the upper bound for the dependency between $A(0)$ and $B(0)$, equals the size of $(A(0) - (A(0) \cap B(0)))$, that is 544. The subtracted part equals the overlap between the two basic sets. The lower bound equals the size of the DVP. Its elements are produced no matter what execution ordering is chosen in the final implementation, that is 6.

$$DVP \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \\ -2 \end{bmatrix}$$

Figure 5: DVP for DV between $A(0)$ and $B(0)$

3.2 Guiding the Design Process

An important part of system level synthesis is to decide on which execution ordering to use for the different loop nests in the code. The execution ordering can be specified starting with the innermost or the outermost dimension. Through an inspection of the DVs, DPs, and DVPs defined in [6] and illustrated in this paper, it is possible to give the designer hints for a good ordering. If a non-spanning dimension is used outermost, it signifies that all elements produced for one value of the corresponding iterator are also consumed for the same value of the iterator. Consequently this dimension can be removed from the DP and the upper bound is reduced. If on the other hand it is used as the innermost dimension, a full set of its elements is produced between each increment of any of the other dimensions. Consequently this dimension must be included in the DVP with an increased lower bound as result. Figure 6 shows the restriction parts of the LBL descriptions for the DP and DVP with i as either the outermost or innermost dimension respectively. In the first case, the upper bound is reduced from 544 to 34, and in the second case the lower bound is increased from 6 to 96.

a)	b)
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ -14 \\ 0 \\ -5 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 0 \\ -15 \\ 0 \\ -1 \\ 0 \\ -2 \end{bmatrix}$

Figure 6: Restriction part of LBL description for a) DP with i outermost, b) DVP with i innermost

The opposite reasoning can be used for a spanning dimension. If it is specified as the outermost dimension, the full set of all other dimensions is produced for all values of the corresponding iterator smaller than the spanning value. The new lower bound equals the size of the union of the original DVP and these elements. If the spanning dimension is used as the innermost dimension, only the full span of this dimension must be included in the DVP. Elements with iterator value larger than the spanning value for this dimension and equal to the spanning value for un-

specified spanning dimensions can furthermore be removed. For both the outermost and innermost case, the lower bound will increase, but typically much more dramatically in the first case. When there is a choice between two spanning dimensions, the one with the shortest distance from its spanning value to the border of the DP should be used as the innermost dimension. This will in general give the smallest increase in lower bound. Figure 7 shows the restriction parts of the LBL descriptions for the elements that must be added to the DVP with k as outermost dimension and the extended DVP with k as innermost dimension. In the first case the lower bound is increased from 6 to 481, and in the second case from 6 to 9 after removal of the elements with $j=1$ and $k=3, 4$ or 5 . If the other spanning dimension, j , was used as the innermost dimension, the lower bound would increase from 6 to 32.

a)	b)
$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 0 \\ -15 \\ 0 \\ -14 \\ 0 \\ -1 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \\ 0 \\ -5 \end{bmatrix}$

Figure 7: Restriction part of LBL description for a) additional elements with k outermost, b) DVP with k innermost

After having specified one dimension as the innermost or outermost, the same reasoning can be used to specify the second innermost or outermost and so on. Following these rules, the best execution ordering for the code example of Figure 1 would be to have k as the innermost dimension, j as the second innermost dimension, and i as the outermost dimension. As shown above, the upper bound is now estimated to 34 array elements after removal of the i dimension from the DP. The lower bound does not change when the last spanning dimension, in this case j , is added as the second innermost dimension. It is still 9. With somewhat more complex algorithms for removal of elements from the DP, outside the scope of this paper, the two bounds would converge at 9, which is indeed the lowest possible number of simultaneously alive array elements.

This example shows how the guidance of the designer works for one basic set. In real life applications, the best execution orderings for different basic sets may be in conflict with one another as demonstrated in the next section. There is consequently a need for a system level synthesis methodology and tool that can find the globally best solution.

4. ESTIMATION ON MPEG-4 VIDEO 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 [2]. We will now use this real life application to show how the guiding hints and storage requirement estimation can be used during the design trajectory. A part of the code is given in Figure 8 with the corresponding data-flow graph in Figure 9.

Assuming that the $prev[][]$ array is already in memory from the previous calculation, the designer must find the execution ordering that gives the lowest storage requirement for the $sad[][][]$ and $curr[][]$ arrays. The bit widths of their array elements, 16 and 8 respectively, must be taken into account. Given an external restriction that the $curr[yp][xp]$ pixels are presented sequentially and row first ($curr[0][0]$, $curr[0][1]$, ... $curr[0][15]$, $curr[1][0]$...) at the input, the required storage for the $curr[][]$ array is as small as possible with yp and xp as outermost and second outermost dimension. All calculations for each pixel can then be completed before the next pixel arrives, and we need

only one byte of storage for the $curr[][]$ array. For any other ordering, we have to buffer the $curr[][]$ elements, since they are needed in bursts. This will require $16 \times 16 = 256$ bytes. Following the guiding hints defined in Section 3.2, the best ordering for the $sad[][][]$ array is to have ys and xs as the outermost dimensions since they are non-spanning dimensions for all basic sets. To be able to choose the globally best ordering, the designer has to estimate the size penalty on the $sad[][][]$ array of having yp and xp as the outermost dimensions.

```

(ys : 0 .. 31)::
(xs : 0 .. 31)::
  (yp : 0 .. 15)::
    (xp : 0 .. 15)::
      sad[ys][xs][yp][xp] =
S.1 if ((xp == 0)&(yp == 0)) ->
      f(curr[yp][xp], prev[ys+yp][xs+xp])
S.2 || ((xp == 0)&(yp != 0)) ->
      g(sad[ys][xs][yp-1][15], curr[yp][xp],
        prev[ys+yp][xs+xp])
S.3 || g(sad[ys][xs][yp][xp-1], curr[yp][xp],
        prev[ys+yp][xs+xp])
      fi;

(ys : 0 .. 31)::
(xs : 0 .. 31)::
S.4 result[ys][xs] = sad[ys][xs][15][15];

```

Figure 8: MPEG-4 motion estimation kernel (Silage code)

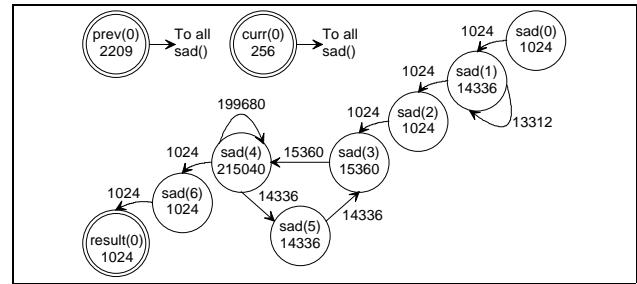


Figure 9: MPEG-4 motion estimation kernel Data-flow graph

Figure 10 gives the LBL descriptions for basic set $sad(2)$, $sad(3)$, and $sad(4)$. $sad(2)$ is produced by statement S.3 in Figure 8 and consumed while $sad(3)$ is produced by S.2. $sad(3)$ is again consumed while $sad(4)$ is produced by S.3. For both of these dependencies, all elements in $sad(2)$ and $sad(3)$ are used to produce the depending basic sets. Consequently their DPs are identical with the corresponding descriptions in Figure 10. The DV between $sad(2)$ and $sad(3)$ starts at (ys,xs,yp,xp) -point $(0,0,0,15)$ and ends at $(0,0,1,0)$. After intersection with the DP of $sad(2)$ this gives a DVP_{2-3} as shown in Figure 11. The DV between $sad(3)$ and $sad(4)$ starts at $(0,0,1,0)$ and ends at $(0,0,1,1)$. After intersection with the DP of $sad(3)$ this gives a DVP_{3-4} as shown in Figure 12. Note that DVP_{2-3} has two SD, yp and xp , while DVP_{3-4} has only one, xp .

The upper and lower bounds on the dependency sizes for the basic sets without any execution ordering specified equal the size of their DP and DVP respectively. This is shown in column a) of Table 1. Note that sizes are given with byte as their unit, so the numbers for the basic sets of the $sad[][][]$ array are twice the number of array elements. The number of simultaneously alive array elements is found through a traversal of the data-flow graph, see [1] for details. Because of the loop between $sad(3)$, $sad(4)$, and $sad(5)$ in Figure 9, parts of these may in the worst case be alive simultaneously, as aggregated in the upper bounds numbers of row "Simultaneously alive".

With yp specified as the outermost dimension, we can remove it from the DPs of basic sets where yp is a non-spanning dimension. This is for example the case for $sad(3)$, and we get a

reduced upper bound for this basic set. For basic sets having yp as a spanning dimension, we have to extend the DVP for all other dimensions to the border of the DP for elements with yp values smaller than its spanning value. Doing so for $sad(2)$ gives an increased lower bound. Column b) in Table 1 presents the results for all basic sets and for the size of the simultaneously alive basic sets with yp as outermost dimension. The new estimate of the lower bound for the $sad[][][]$ array is larger than the whole of the $curr[][]$ array. The designer can therefore already at this early stage decide to instead continue with ys and xs as the outer dimensions.

$$\begin{array}{c}
 sad(2) \\
 sad(3) \\
 sad(4)
 \end{array}
 \begin{array}{c}
 u \\
 v \\
 w \\
 x
 \end{array}
 =
 \begin{array}{c}
 \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \begin{bmatrix} ys \\ xs \\ yp \\ xp \end{bmatrix} \\
 \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix}
 \begin{bmatrix} ys \\ xs \\ yp \\ xp \end{bmatrix} \\
 \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix}
 \begin{bmatrix} ys \\ xs \\ yp \\ xp \end{bmatrix}
 \end{array}
 \geq
 \begin{array}{c}
 \begin{bmatrix} 0 \\ -31 \\ 0 \\ -31 \\ 0 \\ 0 \\ 15 \\ -15 \end{bmatrix} \\
 \begin{bmatrix} 0 \\ -31 \\ 0 \\ -31 \\ 1 \\ -15 \\ 1 \\ -15 \end{bmatrix} \\
 \begin{bmatrix} 0 \\ -31 \\ 0 \\ -31 \\ 1 \\ -15 \\ 1 \\ -15 \end{bmatrix}
 \end{array}$$

Figure 10: LBL for basic set $sad(2)$, $sad(3)$, and $sad(4)$. Common index function upper left. Restriction parts, upper right, lower left and lower right, respectively.

$$DVP_{2-3} \begin{array}{c} w \\ x \end{array} = \begin{array}{c} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} yp \\ xp \end{bmatrix} \\ \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} yp \\ xp \end{bmatrix} \end{array} \geq \begin{array}{c} \begin{bmatrix} 0 \\ 0 \\ 15 \\ -15 \end{bmatrix} \\ \begin{bmatrix} 0 \\ 0 \\ 15 \\ -15 \end{bmatrix} \end{array}$$

Figure 11: DVP for dependency between $sad(2)$ and $sad(3)$

$$DVP_{3-4} [xp] = [1][xp] \begin{bmatrix} 1 \\ -1 \end{bmatrix} [xp] \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Figure 12: DVP for dependency between $sad(3)$ and $sad(4)$

The remaining question is whether to have yp or xp as the innermost dimension. xp is a spanning dimension for all basic sets, while yp is only a spanning dimension for $sad(2)$ and $sad(5)$. As can be seen from the DP of $sad(2)$ in Figure 10, both yp and xp consist of only one value, 0 and 15 respectively. The size of DVP_{2-3} , and with it the lower bound, will therefore not change whatever dimension is added to it. For $sad(5)$ the situation is different, as Figure 13 shows. Since we have to extend the innermost dimension of the DVP to the border of the DP, the lower bound will remain unchanged with xp and increase with a factor of 14 with yp as innermost dimension. For the remaining basic sets where the only spanning dimension is xp , it is obviously also best to have xp as the innermost dimension. The estimate for the storage requirement with the execution ordering fully specified is given in column c) in Table 1. By closer inspection of the code, it can be seen that the lower bound is indeed reached, since only one element from $sad(3)$, $sad(4)$ or $sad(5)$ is alive at the same time.

The last two rows of Table 1 compare our estimate with those found through pure multiplication of the array dimension sizes and those found by the methodology described in [1]. In addition to giving the designer hints for the system level synthesis of the loop ordering, the estimates can be used while considering alternative realizations. Assume for instance that it is necessary to determine whether a certain algorithm can be implemented using only on-chip RAM. Using multiplication of dimensions for estimation, the answer will obviously be *no* with today's technology. Estimating with [1] it might be possible, but it would be evaluated as very expensive. Using our methodology the designer

gets a lower bound indicating that it is probably possible, and also an easy way to verify if the lower bound is reachable. Even when the sub-optimal yp is used as outermost dimension, the upper bound on the estimate indicates that on-chip SRAM is feasible. It is thus very well possible to avoid the use of a more costly solution which was due to overestimates.

$$sad(5) \begin{array}{c} u \\ v \\ w \\ x \end{array} = \begin{array}{c} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} ys \\ xs \\ yp \\ xp \end{bmatrix} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} ys \\ xs \\ yp \\ xp \end{bmatrix} \end{array} \geq \begin{array}{c} \begin{bmatrix} 0 \\ -31 \\ 0 \\ -31 \\ 1 \\ -14 \\ 15 \\ -15 \end{bmatrix} \\ \begin{bmatrix} 0 \\ -31 \\ 0 \\ -31 \\ 1 \\ -14 \\ 15 \\ -15 \end{bmatrix} \end{array}$$

Figure 13: LBL description for basic set $sad(5)$

basic set	a)	b)	c)
$sad(0)$	2/2048	2/2048	2/2
$sad(1)$	2/2048	2/2048	2/2
$sad(2)$	2/2048	2048/2048	2/2
$sad(3)$	2/30720	2/2048	2/2
$sad(4)$	2/30720	2/4096	2/4
$sad(5)$	2/28672	2050/28672	2/2
$sad(6)$	2/2	2/2	2/2
Simultaneously alive	2/90112	2050/34816	2/8
$curr(0)$	1/256	1/256	256/256
Storage requirement	3/90368	2051/35072	258/264
Mult. of dimensions	524544	524544	524544
Method from [1]	90368	90368	90368

Table 1: Estimated lower/upper bounds [bytes] for storage requirement of $sad[][][]$ and $curr[][]$ arrays with a) no ordering, b) yp outermost, c) fully ordered: ys, xs, yp, xp

5. CONCLUSIONS

We have shown how a novel technique for estimating storage requirements for algorithms with partly fixed execution ordering can be used to effectively guide loop transformation decisions. Hints for the system level synthesis step of deciding the execution ordering are presented to the designer. Together with estimates on the upper and lower bounds on the storage requirement, these can be used effectively in early system trade-offs. By means of a real life MPEG-4 VIDEO application, we have demonstrated use of the methodology with a minimum sized memory requirement as result.

6. REFERENCES

- [1] Balasa, F., Cathoor, F., and De Man, H., "Background memory area estimation for multidimensional signal processing systems", IEEE Trans. on VLSI Systems, Vol.3, No.2, June 1995, pp. 157-72
- [2] Brockmeyer, E., Nachtergaele, L., Cathoor F., Bormans, J., and De Man, H., "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
- [3] Cathoor, F., Wuytack, S., De Greef, E., Balasa, F., Nachtergaele, L., and Vandecappelle A., "Custom memory management methodology-Exploration of memory organization for embedded multimedia systems design", Kluwer Academic Publishers, 1998
- [4] Gajski, D.D., Vahid, F., Narayan, S., and Gong, J., "Specification and design of embedded systems", Prentice Hall, 1994
- [5] Grun, P., Balasa, F., and Dutt, N., "Memory size estimation for multimedia applications", Proc. Sixth Int. Workshop on Hardware/Software Codesign (CODES/CACHE), March 1998, pp.145-9
- [6] Kjeldsberg, P.G., Cathoor, F., and Aas, E.J., "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
- [7] Verbauwhede, I.M., Scheers, C.J., Rabaey, J.M., "Memory estimation for high level synthesis", Proc. 31st DAC, 1994, pp.143-8
- [8] Zhao, Y., and Malik, S., "Exact memory size estimation for array computation without loop unrolling", Proc.36th DAC,1999, pp.811-6