Detection of Partially Simultaneously Alive Signals in Storage Requirement Estimation for Data Intensive Applications

Per Gunnar Kjeldsberg
Norwegian University of Science and Technology
Trondheim, Norway
pgk@fysel.ntnu.no

Francky Catthoor
IMEC, Leuven, Belgium
Also at EE.Dept. of Kath. Univ. Leuven
catthoor@imec.be

Einar J. Aas
Norwegian University of Science and Technology
Trondheim, Norway
einar.aas@fysel.ntnu.no

ABSTRACT

In this paper, we propose a novel storage requirement estimation methodology for use in the early system design phases when the data transfer ordering is only partially 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. Using representative application demonstrators, we show how our technique can effectively guide the designer to achieve a transformed specification with low storage requirement.

1. 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 [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 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. 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 certainly not all parts of one array, are alive at the same time. In this context an array element, also denoted a signal, 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 partially 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. It is also necessary to determine which signals are partially overlapping, since their combined size determines the total storage requirement of the application. The degree of overlap between signals, and 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 statements accessing the arrays.

At the beginning of the design process, little information about the execution order is known. Some is given from the data dependencies between the statements in the code, and the designer may restrict the ordering for example due to I/O-constraints. In general however, the execution order is not fixed, giving the designer large implementation freedom. As the process progresses, the designer makes 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 [9] this context and a sketch of a high-level estimation methodology was introduced. This work was continued in [10], presenting a CAD algorithm for size estimates of individual data dependencies.

In this paper the methodology is extended by a CAD technique for detection of partially simultaneously alive signals. This enables the designer to take the full global view of the storage requirement into account while designing data dominated applications. The rest of the paper is organized as follows. Section 2 presents previous work, followed by an overview of our techniques in Section 3. Section 3 also contains details regarding the new methodology for detection of partially overlapping signals. Section 4 illustrates the feasibility and usefulness of the methodology using representative application demonstrators. At the end we present our conclusions.

2. PREVIOUS WORK

By far the major part of all previous work on storage requirement has been scalar based. The number of scalars (individual signals) 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 [11]. Techniques such as clique partitioning are also exploited to group scalars that can be mapped together [14]. A good introduction to scalar based storage unit estimation can be found in [7]. 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 limitation, several research teams 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. For the rest of this paper, such a group of elements is denoted a signal. In [15] 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 [8] 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 yielding 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 loops, and the estimation is rerun. In the worst-case situation a full loop unrolling is necessary to achieve a satisfactory estimate, which is unaffordable. [16] 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 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. A major limitation for all these techniques is their requirement of a fully fixed execution ordering.

In contrast to the methods of the previous paragraph, the storage requirement estimation technique presented in [1] does not take execution ordering into account. It starts with an extended data dependency analysis resulting in a number of non-overlapping *basic sets* of array elements, and the dependencies between them. 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.

In summary, all 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 permits the designer to specify partial ordering constraints, which is really essential during the early exploration of the system level code transformations. When the execution ordering is not fully fixed, the task of finding the signals that may be partially overlapping is more complex. An efficient methodology for this step is consequently of outmost importance.

3. PARTIALLY SIMULTANEOUSLY ALIVE SIGNALS

3.1 Motivation and Context

The new estimation methodology currently employs the technique for detection of dependencies presented in [1]. It may however also use other complete polyhedral dependency descriptions as input. The technique for detection of simultaneously alive signals utilized in [1] is based on a graph traversal heuristic, which can be somewhat optimistic compared to the final implementation. At the same time the estimation of the size of each signal is too pessimistic, since available execution ordering information is not taken into account. A realistic detection of signals that may be partially alive simultaneously is needed, resulting in upper and lower bounds on the total storage requirement for the application. This is the main focus of this paper.

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 [6] and [13]. The single assignment format enables the data dependency analysis needed for generation of the underlying polyhedral dependency graph (PDG) model, [4]. It also opens for code optimizations that can be guided by the storage requirement estimates. The methodology furthermore requires that the resulting Dependency Parts, see below, is orthogonal, or is made orthogonal, as described in [9].

Consider the simple application code example shown in Figure 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 [2]. The loops around the statements define an iteration space as shown in Figure 2 [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* is true, one B-array element is produced. 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 for 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 Non-spanning Dimensions (ND). For the DVP in Figure 2, *i* and *j* are SDs while *k* is ND.

```
for (i=0; i<=5; i++)
  for (j=0; j<=5; j++)
    for (k=0; k<=2; k++){
S.1   A[i][j][k] = f( in[i][j][k] );
S.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

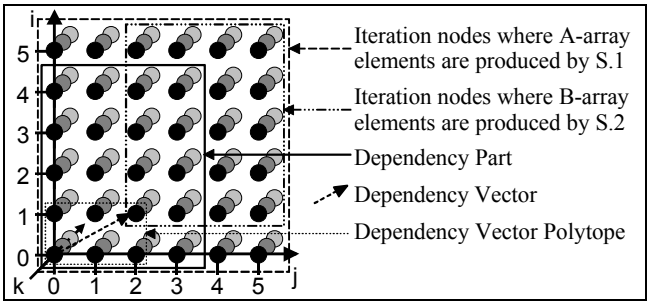


Figure 2: Iteration space with DP, DV, and DVP

Using the concepts presented above, we presented a detailed account of size estimation of individual dependencies in [10] The main contribution is the use of the DP and DVP for calculation of the upper and lower bounds on the dependency size respectively. It is also shown that the size of a dependency is minimized if SDs are fixed innermost and NDs outermost. To be able to take the global view of the storage requirement of an application, the combined size of simultaneously alive dependencies must be taken into account. The general framework of such a complete methodology is

given in Figure 3, while the details regarding detection of partially simultaneously alive dependencies are presented in the next subsections.

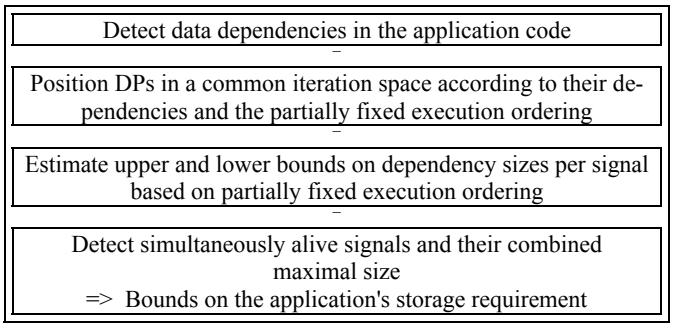


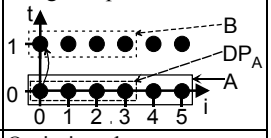
Figure 3: Framework of estimation methodology

3.2 Generation of Common Iteration Space

In the original application code, an algorithm is typically described as a set of imperfectly nested loops. At different design steps, parts of the execution ordering of these loops are fixed. The execution ordering may include both the sequence of separate loop nests, and the order and direction of the loops within nests. To perform global estimation of the storage requirement, taking into account the overall limitations and opportunities given by the partial execution ordering, a common iteration space for the code is needed. A common iteration space can be regarded as representing one loop nest surrounding the entire, or a given part of the code. This is similar to the global loop reorganization described in [5]. Figure 4 shows a simple example of the steps required for generation of such a common iteration space. Note that even though it here includes rewriting the application code, this is not necessary to perform the estimation. The common iteration space can be generated directly using the PDG model of [4].

for (i=0; i<=5; i++)
  A[i] = …
for (j=0; j<=3; j++)
  B[j] = f(A[j]);
	Original code

for (t=0; t<=1; t++)
  for (i=0; i<=5; i++)
  if (t==0) A[i] = …
for (t=0; t<=1; t++)
  for (i=0; i<=5; i++)
  if (t==1) & (i<=3) B[i] = f (A[i]);
	Add outer pseudo-dimension
	Add/enlarge dimensions and add if-clauses to enable merging

for (t=0; t<=1; t++)
  for (i=0; i<=5; i++){
  if (t==0) A[i] = …
  if (t==1) & (i<=3) B[i]= f(A[i]);
  }
	Merge loops
	 

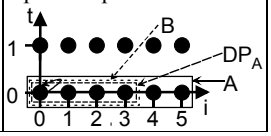
for (t=0; t<=1 t++)
  for (i=0; i<=5; i++){
  if (t==0) A[i] = …
  if (t==0) & (i<=3) B[i] = f(A[i]);
  }
	Optimize placement
	 

Figure 4: Generation of a common iteration space

The introduction of the common iteration space opens for an aggressive in-place mapping which may not be used in the final implementation. The placement of the DPs in the common iteration space entails certain restrictions on the possible execution ordering of the still unfixed parts of the code. The sizes of the individual dependencies will hence be influenced by the placement of their DPs and depending DPs in the common iteration space. To have realistic upper and lower bounds it is therefore necessary to generate two common iteration spaces, one with a worst-case and one with a best-case placement, both taking into account the available execution ordering and other realistic design constraints. [5] presents techniques for placement that enable the designer to organize the data accessing in such a way that aggressive in-place optimization can be employed. The worst-case placement can be based on a full loop body split, so that each statement is assigned individual iterator values in the *t*-dimension. For the rest of this paper, optimal iteration spaces are assumed, but the methodologies presented work equally well on alternative organizations of the iteration space.

3.3 Simultaneous Aliveness for Two Signals

Two dependencies with DPs placed at given positions in the common iteration space, may or may not be partially alive simultaneously. The deciding factor is the way the SDs and NDs of the DPs overlap. In this context, a dimension is an SD if it is an SD for any of the two DPs. For the inspection of overlap, the DP is extended in all SDs with the length of the DV in that dimension. This is necessary since the dependency is alive until all of its elements have been read, that is until all iteration nodes within this Extended DP (EDP) have been visited. In Figure 5, DP *A* is extended (dotted line) from iterator value 3 to iterator value 5 in the *i* dimension. There is overlap in a dimension if for one or more iterator values of that dimension, the two EDPs both have elements. EDP *A* and EDP *B* are thus overlapping in the *i* and *k* dimensions, but not in the *j* dimension.

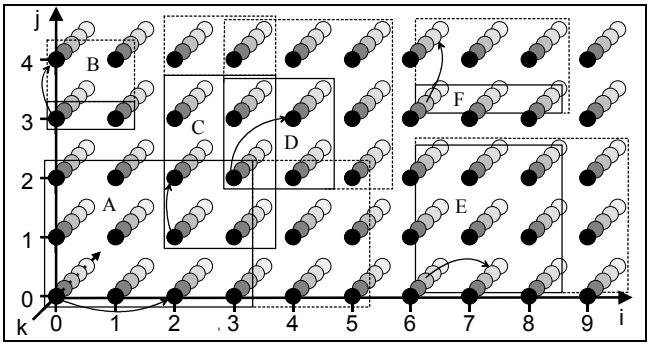


Figure 5: Overlap between EDPs

If all dimensions, SDs and NDs, are partially overlapping, that is the EDPs cover some of the same nodes in iteration space, the dependencies will be alive at least partially simultaneously, regardless of the chosen execution ordering. Elements are produced and/or consumed at the same point in time. EDP *A* and *C* have this kind of overlap in Figure 5. If none of the EDPs' SDs are overlapping, the dependencies can not be alive simultaneously. If some of the NDs are overlapping, the EDPs may however alternate in being alive. EDP *A* and *F* can never be alive simultaneously since no dimensions are overlapping. EDP *A* and *E* can not be alive simultaneously either, since none of the SDs are overlapping. They are overlapping in ND *j*, however, so if this dimension is placed outermost, the two dependencies will alternate in being alive.

The most complex form of overlap occurs when two EDPs are overlapping in one or more dimensions, including at least one SD, but still not in all dimensions. Whether the two dependencies are alive simultaneously will then depend on the execution ordering. Simultaneous aliveness is avoided if at least one dimension with-

out overlap (SD or ND) is fixed outside all overlapping SDs. EDP *A* and *B* are overlapping in all dimensions except the *j* dimension, which must hence be placed outside *i* to avoid simultaneous aliveness. An interesting consequence of this is that the lower bound on the size of the *B* dependency is doubled. The cost of avoiding simultaneous aliveness must therefore be balanced against the increased dependency size. EDP *B* and *F* are only overlapping in the *j* dimension, so ND *i* can be placed outermost, which is optimal for both dependencies. Table 1 summarizes the overlap of spanning and nonspanning dimensions for the EDPs of Figure 5.

	B	C	D	E	F
A	SD=i ND=k	SD=i,j ND=k	SD=i,j ND=k	ND=j	
B		SD=j ND=k	SD=j ND=k		SD=j
C			SD=i,j ND=k	SD=j,k	SD=j,k
D				SD=j,k	SD=j,k
E					SD=i,k

Table 1: Dimensions with overlap for EDPs in Figure 5

3.4 Simultaneous Aliveness for Multiple Signals

More than two dependencies may be alive simultaneously. The detection of this is not straightforward, since one dependency may be alive simultaneously with for example two others, but not necessarily at the same point in time. In Figure 5, EDP *A* is overlapping with both EDP *B* and *E*, but EDP *B* and *E* do not overlap. It is therefore impossible for all three of them to be alive simultaneously. As before, the most complex situation occurs when only a subset of the dimensions is overlapping. EDP *D*, *E* and *F* are all overlapping with each other but depending on the chosen execution ordering, only one, any combination of two, or all three will be alive partially simultaneously.

```
for each iterator value of pseudo-dimension t {
  for each dimension d_i {
    generate list ( length = |d_i| )
    for each EDP
      place start and end points at corresponding list location
    traverse list and group EDPs overlapping in dimension d_i
  }
  sort groups of EDPs according to their number of EDPs
  for each group starting with the largest group {
    intersect the group with all smaller groups
    remove groups that are fully covered by the current group
} }
return groups of possibly partially simultaneously alive EDPs
```

Figure 6: Simultaneous aliveness for multiple dependencies

To overcome the complexity of deciding whether three or more dependencies really are alive simultaneously the task is divided into two consecutive steps. First dependencies that may be alive simultaneously are grouped, followed by an inspection to reveal if they can really be alive simultaneously for a given partially fixed execution ordering. An algorithm that groups EDPs is given in Figure 6. Its use will now be demonstrated using the iteration space of Figure 5. In this case, the pseudo dimension *t* has only one iterator value and is therefore ignored. Table 2 displays the list for one of the three dimensions, the *j* dimension, after insertion of each EDP's start (X_s) and end (X_e) points in that dimension. The traversal of the lists detects overlapping EDPs by adding EDPs to a group as long as start points are encountered. When one or more end points are encountered, the current group is terminated and a new one is started without the EDPs that have ended. For the *j* dimension in Table 2 EDPs *A*, *E*, *C*, and *D* are added to the first group at iterator values 0, 1, and 2. At iterator value 2 the end points of EDP *A* and *E* are encountered, so the first group is finished and a second one is started containing *C* and *D*. At iteration node 3, EDPs *B* and *F* are added to this second group. The groups for this and the other dimensions are summarized in Table 3. It also shows the sorting of all groups from all dimensions and the removal of covered groups. Finally it lists the combined upper bounds on the size for all DPs in each group as estimated using the technique presented in [10].

Val.	0	1	2	3	4
	A_s E_s	C_s	D_s A_e E_e	B_s F_s	B_e C_e D_e F_e

Table 2: List of start and end points for EDPs in *j* dimension

i	AB, ACD, EF	ABCD	<u>ABCD</u>	ABCD	41
j	AECD, CDBF	AECD	AECD	AECD	49
k	ABCD, CDEF	CDBF	CDBF	CDBF	35
		CDEF	CDEF	<u>CDEF</u>	43
		ACD	~~ACD~~	~~EF~~	
		AB	~~AB~~		
		EF	EF		
In each dimension		Sorted	Covered by ABCD	Covered by CDEF	Combined size

Table 3: Groups of overlapping EDPs

The next step is to perform a check of the possible simultaneous size of the different groups, starting with the biggest group. The global upper and lower bounds are found when the lower bound for one group is bigger than the combined size of the next group. Starting with group AECD it can be seen from Table 1 that EDPs *A* and *E* only overlap in ND *j*. The group is hence split into two new groups, ACD and ECD, both of which are covered by larger groups. CDEF is now the largest group. *C* and *D* are overlapping in all dimensions, and will consequently be alive simultaneously independent of the chosen execution ordering. *E* and *F* are overlapping in SDs *i* and *k* and will thus be alive simultaneously unless the dimension without overlap, *j*, is placed outermost. Both *C* and *D* are overlapping with *E* and *F* in SDs *j* and *k*, so these pairs will be alive simultaneously unless *i* is placed outermost. For all four to be alive simultaneously *k* must consequently be placed outermost. The actually simultaneously alive EDPs for three partially fixed execution orderings, and their upper and lower bounds are given in Table 4. The upper and lower bounds of groups of EDPs are found through addition of the bounds of the individual DPs using the methodology from [10]. Since these may require conflicting orderings they may not be reachable simultaneously. Even closer inspections of the groups are therefore needed to ensure that the bounds are reachable. In this case no conflicting orderings exists and it can be concluded that for group CDEF, the lowest storage requirement can be reached if *i* is placed outermost. Note that the ordering that results in the largest number of simultaneously alive dependencies is not the ordering with the largest storage requirement. This is because this ordering results in smaller individual dependencies.

Outermost	Simultaneously alive	UB	LB
i	C and D or E and F	20	12
j	C and D and E or C and D and F	31	27
k	C and D and E and F	23	18

Table 4: Actual simultaneous aliveness for group CDEF

The discussion above covers simultaneously alive dependencies. Multiple dependencies may however stem from the same, or partly

the same, array elements. This must be taken into account in such a way that only the dependency with the longest lifetime for a given partial ordering is counted.

4. 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, [17]. An important part of the coding of this data stream is the motion estimation (ME) of moving objects. See [3] for a more detailed description of this part of the standard. This real-life application will now be used to demonstrate the effectiveness of the new methodology for detection of simultaneously alive dependencies during storage requirement estimation. The code for a part of the ME algorithm is given in Figure 7. The sad-array is the only one that is both produced and consumed within the boundaries of the loop nest. For this example, the dependency detection technique presented in [1] is utilized. This results in a number of basic sets (signals) and their dependencies, placed in a common iteration space. There are eight basic sets, *sad0* to *sad6* and *res0*. Dependencies exist from *sad(n)* to *sad(n+1)* and also from *sad5* to *sad3* and from *sad6* to *res0*. Table 5 lists the start and end points for the dependencies' EDPs in dimensions *y_p* and *x_p*. *sad01s* indicates the starting point for the EDP of the dependency from *sad0* to *sad1*. For dimensions y_s and x_s, all dependencies start at iterator value 0 and end at 31. These dimensions are furthermore NDs for all dependencies, and have consequently no influence on the degree of simultaneous aliveness, except for causing dependencies to alternate in being alive. These dimensions are therefore ignored in the sequel. Using the algorithm in Figure 6, Table 5 is inspected to reveal groups of possibly simultaneously alive dependencies. After removal of fully covered groups, seven groups remain, differing in size from 60416 to 4096.

```
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++)
        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]);
        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]);
        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]);
        if ((y_p==15)&((x_p==15)) result[y_s][x_s]=
                                        f4(sad[y_s][x_s][15][15]);
```

Figure 7: MPEG-4 motion estimation kernel

The final step is now to inspect each group to find the dependencies that are really alive simultaneously for a certain partially fixed execution ordering. For this example, data dependencies in the code forces *x_p* to be fixed inside *y_p*, since dependencies exist from (*y_p*,*x_p*)-node (*i*,15) to (*i*+1,0) for $0 \le i \le 14$. Apart from this, the ordering of the dimensions, including *y_s* and *x_s*, can be chosen arbitrarily. The largest group consists of dependencies *sad23*, *sad34*, *sad44*, *sad45*, and *sad53*. Table 6 summarizes in which dimensions the dependencies are overlapping. Note that if an overlap only covers the extension of one of the EDPs, and a dependency is present between the two basic sets in question, then the two dependencies will not be alive simultaneously. Direct in-place mapping can be performed between the two basic sets. Most of the remaining overlap either occurs only in NDs, e.g. between *sad44* and *sad53*, or in an SD that, due to the partially fixed execution ordering, is known to be placed inside a non-overlapping dimension, e.g. between *sad23* and *sad45*. These overlaps will not cause simultaneous aliveness between the dependencies. The remaining two overlaps to be investigated further are thus the ones between *sad23* and *sad44*, and between *sad34* and *sad53*. The SD *y_p* is in both cases caused by the negative dependency discussed above, and can be removed when *x_p* is ordered inside *y_p*. The conclusion is thus that no dependencies are overlapping in this group, and similar reasoning reveals that this is the case for the other groups as well. This is however very important information for the designer, since the focus can then be solely on the task of minimizing the size of individual dependencies [10].

Dim	0	1	…	14	15
y_p	sad01s sad11s sad12s sad23s sad01e sad11e sad12e	sad34s sad44s sad45s sad53s sad23e		sad45e	sad46s sad6res0s sad53e sad34e sad44e sad46e sad6res0e
x_p	sad01s sad34s sad23e sad53e	sad11s sad44s sad01e sad34e		sad12s sad45s sad46s sad11e sad44e	sad23s sad53s sad6res0s sad12e sad45e sad46e sad6res0e

Table 5: Start and end points for EDPs

	sad34	sad44	sad45	sad53
sad23	Not sim.	SD=y_p	SD=x_p	SD=x_p
sad34		Not sim.	ND=y_p	SD=y_p,x_p
sad44			Not sim.	ND=y_p
sad45				Not sim.

Table 6: Overlapping dimensions for the largest group

4.2 SVD Updating Algorithm for Beamforming

The results from the storage requirement estimation can also be used as feedback during interactive or tool driven global loop reorganization, [5]. This will now be demonstrated using the Singular Value Decomposition (SVD) algorithm, for instance required in beamforming [12]. The SVD algorithm continuously updates matrix decompositions as new rows are appended to a matrix. Figure 8 shows the two major arrays, *R* and *V*, and the important loop nest and statements for their production and consumption. The figure also contains two minor arrays used for the production of *R* and *V*, the *theta* and *phi* arrays. After orthogonalization, see [9], statement S.3, S.4, and S.5 each produce elements at all iteration nodes within their *i* and *j* loops.

The following dependencies exist in the USVD code of Figure 8: S12, S13, S24, S25, S34, S41, S42, S43, and S55. S12 indicates that there is a dependency from statement S.1 to statement S.2. Due to data dependencies not shown in Figure 8, the *k*-dimension must be placed outermost during the production of the *R*-array. This is however not necessary for the production of the *V*-array, as long as the necessary *phi*-values are available. A comparison of the resulting storage requirement for two alternative loop organizations, with or without a loop body splitting that places the V-array in a separate loop nest, is therefore needed. The second situation, where the loop structure inside the *k*-loop is kept, is focused first.

A common iteration space is generated as outlined in Figure 4. A pseudo *t* dimension with three iterator values is in this case added inside the *k* dimension since *k* is fixed outermost. Statements S.1

and S.2 are executed for *t=0*, S.3 for *t=1*, and S.4 and S.5 for *t=2*. An investigation of the overlap between the dependencies' EDPs in the *i* and *j* dimensions reveals that all EDPs but one, S34, are overlapping for *t=0*. For *t=1* three EDPs are not overlapping (S12, S42, and S41). Finally for *t=2*, two EDPs are not overlapping (S12 and S13). None of these groups are fully covered by others, so all three must be investigated further. Starting with the group for *t = 0*, intersection of the DPs produced by S.4 reveals that all array elements are covered by the DP of the S43 dependency. Since it also has the longest lifetime independent of the chosen execution ordering, the others can be ignored. Furthermore, the DPs of S24 and S25 overlap completely, so only one needs to be taken into account. The same holds for the DPs of S12 and S13. S13 has the longest lifetime, so it is considered further. Similar reasoning can be used for the two other groups, leaving simultaneously alive dependencies as shown in Table 7a. Again using the estimation methodology for individual dependencies presented in [10], Table 7b shows the combined sizes for each group.

```
     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] = f6( R[i][j][2*k+1], phi[k] );
           ...
S.5        else V[i][j][k+1] = f9( V[i][j][k], phi[k] );
     }   }
```

Figure 8: USVD algorithm, diagonalization loop nest

	t=0	t=1	t=2
a)	S13 S24 S43 S55	S13 S24 S43 S55	S24 S43 S55
b)	202	202	201

Table 7: a) Simultaneously alive dependencies without loop body split and b) combined size with *k* outermost (n=10)

To produce a common iteration space after a loop body split, two pseudo dimensions are introduced, *t1* outermost and *t2* inside the *k* dimension. Statements S.1 and S.2 are executed when *t1=0* and *t2=0*, S.3 is executed when *t1=0* and *t2=1*, S.4 is executed when *t1=0* and *t2=2*, and S.5 is executed for *t1=1* and *t2=0*. Note again that the introduction of the pseudo dimensions is only needed during the estimation. They are not necessarily used for the final implementation. Table 8 gives the results of the investigation into the simultaneously alive dependencies and their combined storage requirements. As can be seen, the maximal size of simultaneously alive dependencies is approximately halved compared to the result without a loop body split. This information is vital for the designer when comparing different implementation alternatives.

	t1=0&t2=0	t1=0&t2=1	t1=0&t2=2	t1=1&t2=0
a)	S13 S24 S43	S13 S24 S43	S24 S43	S24 S55
b)	110	110	109	
c)				10/19

Table 8: a) Simultaneously alive dependencies after loop body split, b) their combined size with *k* outermost, and c) bounds on combined size with no ordering fixed (n=10)

5. CONCLUSIONS

This paper presented a novel technique for detection of partially overlapping signals in high-level application code. This is a crucial step during storage requirement estimation to allow the designer to get the global view during design exploration, and to minimize storage requirements. In contrast to previous techniques, we provide upper and lower bounds with a partially fixed execution ordering. Design examples have shown how the methodology can be used to guide the designer, and compare implementation alternatives, resulting in a solution with low storage requirement.

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

6. 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] K. Danckaert, F. Catthoor, and H. De Man, "A preprocessing step for global loop transformations for data transfer and storage optimization", Proc. Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES), Nov 2000, pp.34-40

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

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

[8] 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

[9] 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, May 2000, pp. 56-60

[10] P.G. Kjeldsberg, F. Catthoor, E.J. Aas, "Automated Data Dependency size Estimation with a Partially Fixed Execution Ordering", Int. Conf. on Computer Aided Design, ICCAD 2000, Nov. 2000, pp. 44-50

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

[12] 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

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

[14] 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

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

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

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